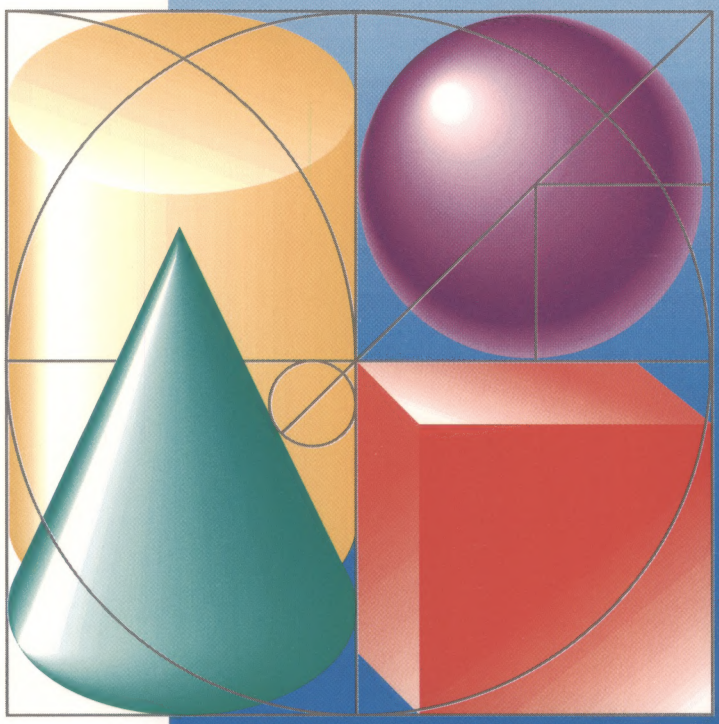
APW™ Tools Reference

APDA™ A0240LL/A



Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

To reorder products, please call:
Apple Programmers and Developers Association
1-800-282-APDA

Guide to
APW Tools & Interfaces
Release v.1.1

This release consists of four items:

- Two disks containing the APW Tools software
- A disk containing APW C interfaces and APW Assembly equates and macros as well as the interface release notes
- The *APW Tools Reference* manual
- These release notes.

These release notes describe the differences between the APW Tools software and the APW documentation, as well as the changes between "Programming Tools & Interfaces for APW" and this product.Release notes for the Assembler and C interfaces can be found on the **APW.Interfaces** disk in the **Interfaces/IIGSSupport/** folder. There are seperate files for Assembler and C interface release notes. These files are standard text files formatted for use with the APW editor.

NOTE:

This software will work with APW Version 1.0 through 1.0.2, but the new utilities require System Disk 5.0.2 or later.

NOTE:

We believe we have done a thorough and complete job of testing of the product that you receive. However, please report any problems that you find with the APW Tools software or documentation; you may use the enclosed "APW Tools BUG REPORT" form. In order to facilitate fixing interface problems, a special "INTERFACE BUG REPORT" form has been added. Please use that form to let us know of any problems you may find with any of the interface files.

APW Product Manager
Apple Computer, Inc.
20525 Mariani Ave., MS: 75-8X
Cupertino, CA 95014

Installation

NOTE:

APW requires one megabyte of *available* memory. System Disk 5.0.2 requires more memory than previous versions, thus reducing the total amount of memory available to APW. Please keep this in mind if you suddenly find yourself pinched for memory while using APW under System Disk 5.0.2.

NOTE:

Do not use APW's COPY command to install the new tools. Some of them contain resource forks and APW's COPY command will NOT copy the resource forks. This will result in incomplete files that do not work! Use the installation instructions presented here or use the Finder v.1.3 or later to copy these files instead. A new utility (DUPLICATE) that will copy resource forks correctly is in this package. That tool is an important part of the installation procedures presented below and will be your best tool for moving and copying resource forks until such time as the COPY command can be revised.

Hard Disk:

- Launch APW and insert the **/APW.Tools1** disk into a disk drive.
- Set the current directory to **/APW.Tools1** by issuing the shell command **PREFIX /APW.Tools1**
- Execute the installation script by typing: **NEWSTUFF** and pressing <RETURN>. This script will completely install a utility which is capable of copying resource forks; and transfer the new utilities, updates to older utilities, and help files to prefix 6 (which is the APW UTILITIES prefix). The script will send a list of files to standard out as they are being transferred so that you can track what is going on.
- Insert the **/APW.Tools2** disk into a disk drive.
- Set the current directory to **/APW.Tools2** by issuing the shell command **PREFIX /APW.Tools2**
- Execute the installation script by typing: **MORENEWSTUFF** and pressing <RETURN>. This script will transfer the remaining utility, the resource compiler, the RInclude files; will append the resource manager TAB settings to the **4/SYSTABS** file; and will append the proper lines to the **4/SYSCMND** file so that the new tools and the resource compiler are completely installed. The script will send a list of files to standard out as they are being transferred so that you can track what is going on.
- Edit the **4/SYSTABS** file to eliminate any duplicate entries the installation script's auto-appending may have caused. Eliminate the first tab setting for the resource compiler (Rez) that you find since the lines added by the installation script are correct and should supersede the previous entries. Rez is language number 21.
- Edit the **4/SYSCMND** file to eliminate any duplicate entries the installation script's auto-appending may have caused. Eliminate the original listing for any utility or language that you find since the lines added by the installation script are correct and should supersede the previous entries.
- The **/APW.Tools2** disk contains an icon for APW itself. Install this icon by issuing the shell command **COPY /APW.Tools2/Icon/APW.Icon */Icon/APW.Icon**
- The **/APW.Tools2** disk also contains a revised dictionary for use with Canon. To install this new dictionary, issue the shell command **COPY /APW.Tools2/Utilities/Canon.Dict 6/Canon.Dict**
- Insert the **/APW.Interfaces** disk into a disk drive.
- Set the current directory to **/APW.Interfaces** by issuing the shell command **PREFIX /APW.Interfaces**

- To install both the new Assembly interface files and the new APW C interface files, execute the installation script by typing: **NEWALL** and pressing <RETURN>. This script will transfer the E16 and M16 interface files to prefix **2/AINCLUDE** and the APW C interface files to prefix **2/CINCLUDE**. The script will send a list of files to standard out as they are being transferred so that you can track what is going on.
- To install only the new Assembly interface files, execute the AInclude installation script by typing: **NEWASM** and pressing <RETURN>. This script will transfer the E16 and M16 interface files to prefix **2/AINCLUDE** (which is the APW LIBRARIES folder). The script will send a list of files to standard out as they are being transferred so that you can track what is going on.
- To install only the new APW C interface files, execute the CInclude installation script by typing: **NEWAPWC** and pressing <RETURN>. This script will transfer the APW C interface files to prefix **2/CINCLUDE** (which is the APW LIBRARIES folder). The script will send a list of files to standard out as they are being transferred so that you can track what is going on.

LAST STEPS:

- Verify that all of the files in **2/AINCLUDE** and **2/CINCLUDE** have creation dates of 4/1/90. If any of them have older creation dates, then they are not part of the new interfaces. Some files have had subtle name changes, be certain that you are indeed using only the new interfaces.
- Either execute the command **COMMANDS 4/SysCmnd** or Quit APW and relaunch it so that the changes to **4/SysCmnd** have been executed. This will provide access to all of the new tools.

NOTE:

A hard disk is highly recommended for APW users since APW now requires several disks in order to support the full complement of utilities available from Apple. Shuffling three or four floppies in and out of the drives when different utilities, compilers, the editor, or your source files are needed will reduce your efficiency. However, we recognize that many APW users will be using floppy-based systems and hope that the following installation instructions meet their needs. In the steps below, the disks named /APW and /APWU refers to the two disks that came with the original APW product. Installation of these new tools may be easier if you make a fresh copy of /APW and /APWU and use them as needed during the installation. The installation instructions assume you have at least two 3.5" disk drives.

Floppy Disk:

- Make a backup copy of the three disks from this package and perform the installation using the backup copies. Set the originals aside in a safe place. Make sure that the backup disks have the same name as the original disk.
- Make a fresh copy of System Disk 5.0.2 and install the UniDisk 3.5" and/or the Apple Disk 5.25" drivers if you will be using those types of disk drives. Do not install AppleShare at this time.
- Name this new disk **APW.Boot502**
- From the Finder (use Finder v.1.3 or later for this), remove the following files/folders from **/APW.Boot502** to make room for APW:
  - The file **/APW.Boot502/BASIC.System**
  - The file **/APW.Boot502/BASIC.Launcher**
  - The folder **/APW.Boot502/Icons** and its contents
  - The folder **/APW.Boot502/Tutorial** and its contents
  - The folder**/APW.Boot502/AppleTalk**
  - The file **/APW.Boot502/System/Start**

- Insert **/APW** and use the Finder to copy the following files:
  /APW/APW/APW.SYS16 to the root directory of **/APW.Boot502**
  /APW/APW/SYSTEM/SYSCMND to the **/APW.Boot502/System** folder
  /APW/APW/SYSTEM/SYSTABS to the **/APW.Boot502/System** folder
  /APW/APW/SYSTEM/LOGIN to the **/APW.Boot502/System** folder
  /APW/APW/SYSTEM/EDITOR to the **/APW.Boot502/System** folder
  /APW/APW/SYSTEM/SYSEMAC to the **/APW.Boot502/System** folder
  /APW/APW/SYSTEM/SYSHELP to the **/APW.Boot502/System** folder
- Create a new folder in the root level of **/APW.Boot502** and rename it to **Utilities**
- Create a new folder in the root level of **/APW.Boot502** and rename it to **Work**
- Remove the **/APW.Boot502** disk from the desktop.
- Use the Finder to rename **/APW.Tools1** to **/APW.Utils**
- Insert **/APW** and use the Finder to copy the following files:
  /APW/APW/UTILITIES/CRUNCH to the **/APW.Utils /Utilities** folder
  /APW/APW/UTILITIES/INIT to the **/APW.Utils /Utilities** folder
  /APW/APW/UTILITIES/MACGEN to the **/APW.Utils /Utilities** folder
- Set **/APW.Utils** aside for now.
- From the Finder, initialize a 3.5" disk and name it **APW.Other**
- Insert **/APW.Tools2** and use the Finder to copy the following files:
  The folder **/APW.Tools2/Libraries** and its contents to the root directory of **/APW.Other**
  The folder **/APW.Tools2/Languages** and its contents to the root directory of **/APW.Other**
  The folder **/APW.Tools2/Utilities** and its contents the root directory of **/APW.Other**
- Remove **/APW.Tools2** and insert **/APW.Interfaces**
- Copy the **/APW.Interfaces/AInclude** folder and its contents to **/APW.Other/Libraries**
- Delete the following files:
  /APW.Other/Libraries/AInclude/E16.AppleShare
  /APW.Other/Libraries/AInclude/E16.AppleTalk
  /APW.Other/Libraries/AInclude/E16.Scheduler
  /APW.Other/Libraries/AInclude/M16.AppleShare
  /APW.Other/Libraries/AInclude/M16.AppleTalk
  /APW.Other/Libraries/AInclude/M16.Types
- Remove **/APW.Interfaces** and insert **/APW**
- Copy the the following files (do not copy the folders, just the files):
  /APW/APW/LANGUAGES/LINKED to the **/APW.Other/Languages** folder
  /APW/APW/LANGUAGES/ASM65816 to the **/APW.Other/Languages** folder
- Remove **/APW** and insert **/APW.Tools2**
- Copy the the following file (do not copy the folder, just the files):
  /APW.Tools2/UTILITIES/CANON.DICT to the **/APW.Other/Utilities** folder
- Reboot your computer using the **/APW.Boot502** disk and insert the **/APW.Utils** disk into your second disk drive.
- Set the current directory to **/APW.Utils** by issuing the shell command **PREFIX /APW.Utils**
- Execute the installation script by typing: **F.NEWSTUFF** and pressing <RETURN>. This script will install a utility that is capable of copying resource forks onto **/APW.Boot502** and will add several lines to your **4/LOGIN** file. These commands will define aliases to simplify adjusting prefixes 2, 5, and 6 to **/APW.Other/Libraries**, **/APW.Other/Languages**, and **/APW.Utils/Utilities** respectively.
- Remove **/APW.Utils** and insert **/APW.Tools2**
- Set the current directory to **/APW.Tools2** by issuing the shell command **PREFIX /APW.Tools2**
- Execute the installation script by typing: **F.MORENEWSTUFF** and pressing <RETURN>. This script will append the resource manager TAB settings to the **4/SYSTABS** file; will append the proper lines to the **4/SYSCMND** file so that the new tools and the resource compiler are accessible; and will transfer the **SysErrs** file to the **/APW.Boot502/System** folder. The script will send a list of files to standard output as they are being transferred so that you can track what is going on.
- Edit the **4/SYSCMND** file to eliminate any duplicate entries the installation script's auto-appending may have caused. Eliminate the original listing for any utility or language that you find since the lines added by the installation script are correct and should supersede the previous entries.

- Edit the **4/SYSTABS** file to eliminate any duplicate entries the installation script's auto-appending may have caused. Eliminate the first tab setting for the resource compiler (Rez) that you find since the lines added by the installation script are correct and should supersede the previous entries. Rez is language number 21.
- To install the APW C interfaces and compiler, perform the steps above, but substitute **/APW.Interfaces/CInclude** for **/APW.Interfaces/AInclude** and **/APWC/LANGUAGES/CC** for **/APW/APW/LANGUAGES/ASM65816** when copying files to the **/APW.Other** disk.

LAST STEPS:

- Verify that all of the files in **2/AINCLUDE** and **2/CINCLUDE** have creation dates of 4/1/90. If any of them have older creation dates, then they are not part of the new interfaces. Some files have had subtle name changes, be certain that you are indeed using only the new interfaces.
- Either execute the command **COMMANDS 4/SysCmnd** or Quit APW and relaunch it so that the changes to **4/SysCmnd** have been executed.

NOTE:

For editing, the /APW.Boot502 disk must be online. For compiling, the /APW.Other disk must be online. To use a utility other than CANON, the /APW.Utils disk must be online. To use CANON, set prefix 6 to /APW.Other/Utilities. The utility DUPLICATE is present on both the /APW.Utils disk and on the /APW.Boot502 disk–this will allow you to manipulate files that have resource forks without having to have the /APW.Utils disk online. Set prefix 6 to /APW.Boot502/Utilities when you need to use DUPLICATE without having the /APW.Utils disk online. Be sure to set prefix 6 back to /APW.Utils/Utilities in order to be able to use the other utilities. You may wish to set up an alias or two to simplify this juggling.

Errata and Enhancements

General:

- The tools in this package now support the "spinning cursor" during operation to provide visual feedback to the user while the tools are in use. Pressing any key (except COMMAND-PERIOD) will pause these tools. When paused, the spinning cursor will change to an hour glass. Pressing any key (except COMMAND-PERIOD) will allow the tool to resume operation.
- Pressing COMMAND-PERIOD will abort these tools during their operation. Most of these tools will send a string to standard error output stating that they aborted because COMMAND-PERIOD was pressed.
- The utilities in the package (but not the resource compiler) will give version/copyright/progress information if the **-p** option is specified on their command line. The only exception to this rule is DiskCheck.
- Canon, Equal, Files, Search, and the new tools in this product do not support the "+option" syntax for specifying options for this utility. The default values for all options are listed in the manual. To turn on an option that is off by default, use the "-option" syntax. Example: Using the **-p** option will cause tools that support it to produce a progress report, leaving **-p** off of the command line will result in no progress report.

Canon:

- Canon will return a status code of -1 if the user aborts it by pressing **COMMAND-.** (COMMAND-PERIOD).
- Canon now allows the user to specify only -p on the command line (to display just the version number).

Compact:

- All known bugs in this tool have been fixed.

CrunchIIGS:

- This utility will combine the **.root** and **.a** through **.z** files into one single **.obj** file. This is required in order for LinkIIGS to use files that were compiled using any of the ORCA languages and those using certain features of APW Assembler. This does **NOT** adversly affect partial compiles.
- The syntax for this utility is: **CrunchIIGS rootname1 [rootname2...]**
- **rootname1** is the full or partial pathname, including the filename but minus any file name extensions, of the object file you wish to combine.
- If more than one rootname is specified, each is processed separately.
- A complete example of using CrunchIIGS along with LinkIIGS would be:

```
compile myUtil.Asm keep=$
* this produces two files: myUtil.root, myUtil.a
CrunchIIGS myUtil
* this produces one file: myUtil.obj
LinkIIGS myUtil.obj -o myUtil
```

DeRez:

- This new tool was added to this disk as part of the v.1.1 update. DeRez is a port of the MPW DeRez tool for the Macintosh.
- DeRez puts TAB characters in the resulting ouput file. These characters will cause problems if the file is sent to the printer via the IIGS's built-in printer port (or via a Super Serial card). Either edit the file and replace the TABs with several spaces or use a deTAB utility if you want to print the output of DeRez.

DiskCheck:

- DiskCheck searches ProDOS volumes and identifies block conflicts (two files using the same block) and provides a list of files and blocks in conflict.
- DiskCheck will also display errors in the disk's bitmap and allow the user to correct those errors.

DumpObj:

- A new option has been added to this tool to allow disassembly of object files using the MPW IIGS Assembler's syntax rather than the APW Assembler's. The default for this option is to use the syntax of the APW Assembler. In order to force the use of MPW IIGS assembler syntax, use the "**-mpw**" option.
- The **-l** option has been changed to specify short LCONST dumps.

- DumpObj now supports OMF versions 1.0, 2.0, and 2.1.
- The formatting of SUPER record dumps has been enhanced.
- Added **-s** option
- Corrupt segment bodies are handled much better than version 1.0.
- The second column for -x and -xa dumps is now the segment offset (previously zero at all times)

Duplicate:

- Duplicate was added to this disk to allow the copying of files containing resource forks. APW's **COPY** command can **NOT** currently handle resource forks (they will be stripped off without warning).
- This utility will **NOT** expand the directory walking operator (../). Only legal ProDOS pathnames are accepted.
- The destination pathname is now optional.
- Wildcards are not supported.
- Only one file can be duplicated at a time.
- If the pathname being passed to **Duplicate** contains spaces, then that pathname must be enclosed in quotes.
- The **-m** option is no longer supported. This option was of no value to normal users, so its loss will probably go unnoticed.

Equal:

- Both the data and the resource forks are compared by default.
- The **-F resource** option will force Equal to compare only the resource forks.
- The **-F data** option will force Equal to compare only the data forks.

Express:

- This new tool was added to provide quick access to the IIGS's new Express Loader (which is part of IIGS System Disk v.5.0 and later) and may not work on all files. If a file can not be "Express'd" completely, then it will not be expressed at all.
- Files must be in OMF 2.0 format to be Express'd. Compact can be used to convert OMF 1.0 files into OMF 2.0.
- LinkIIGS creates "Express'd" files automatically, so newer files will not need to be converted with Express.
- If the file being Express'd has already been Express'd, then it will simply be copied.
- Early versions of this tool create a new segment named "Express", this version creates a segment named "~Express". Express will treat either type as "already Expressed" but it is best to recreate the original file and Express it with this newer version.

Files:

- Several new options have been added to this utility. All of these are documented in the *APW Tools Reference* manual.

Join:

- This utility will combine two files into a third by copying the first, then appending the second file.
- This utility does not support wildcards or the directory walking aoperator.
- The syntax for this utility is: **Join file1 file2 file3**
- File1 is copied to create file3, then file2 is appended. An example of using this tool can be found in the CrunchIIGS script in the UTILITIES folder.

LinkIIGS:

- This is a port of the MPW IIGS Linker and is a scriptable linker.
- LinkIIGS supports OMF Version 2.1 which includes: Object Segments having a tempORG (temporary origin); Skip Object Segments; and Bank Relative Object Segments.
- LinkIIGS generates load files in ExpressLoad format by default. This can be suppressed by using the **-x** option on the linker command line.
- The auxtype of the ouput file can be set using the **-at** option. The format for this option is: **-at $xxxx** where xxxx is the desired auxtype. Decimal numbers may also be used if the **$** is left off.
- The **-t** option can be used to set the type of the output file The syntax is **-t $xx** where xx is the desired Load File type. A decimal value can be used if the **$** is left off. Several mnemonics are recognized and will be converted to their corresponding file type. These mnemonics are: **S16**, **EXE**, **RTL**, **NDA**, **CDA**, **PIF**, **TIF**, and **TOL**. If the output Load File type is set to **RTL**, the Load File will be converted to a Run Time Library.
- The combination of **'-t'** and **'-at'** allow both a file type and auxtype to be assigned at link time.
- Larger files may be linked in the same amount of memory when the **-b** option is used.
- A "bigger link" (**-b2** option) is also available. All features of the "big link" (**-b** option) will be performed and generation of SUPER records will be suppressed to save memory. SUPER records can still be created using the "big link" option (**-b**) or using a "regular" link.
- When linking Run Time Libraries, the **-stamp** option should be used to put the Run Time Library file's time/date information into the corresponding Pathname Segment entry. This information will be used by the System Loader when it opens the Run Time Library to make sure that it is the same file that was used during the link. If this option is not used, the time/date stamp will be 0 and the System Loader will not make the run time check.
- If the **-w2** option is used on the linker's command line, warnings will be treated as fatal errors.
- If standard input is redirected from a file, the linker will read the file and interpret the textual data as additional parameters. No special continuation characters are required. For the sake of readability, you may wish to place each linker option on a different line. An example of a simple script file would look like:

```
2/start.root
direct.obj
myUtil.obj
myStdStuff.obj
-lib 2/clib
-t exe
-b
-o myUtil
```

- The earlier release of LinkIIGS could not handle large script files due to a bug in one of the libraries that were used to create LinkIIGS. This problem has been fixed and large script files can now be used successfully.
- LinkIIGS does not support APW's partial compiles. Each object file must be listed or it will not be linked. To avoid listing all the **.a** through **.z** files (along with their **.root** file), simply use the CrunchIIGS script provided with this product. For details, see the Release Notes sections on CrunchIIGS.
- Complete filenames must be provided. The standard **.root** and **.a**, **.b**, etc will **NOT** be appended for you.
- The **-org** option has a new meaning if the Load Segment is a Bank Relative Segment.

- The progress report and symbol table printouts have more information than LinkIIGS v.1.1B4. All diagnostic messages that contain the Segment Name and Segment Offset now also contain the File number and Segment Number as hexadecimal values preceding the Segment Name. For example:

LinkIIGS - warning - 1:2:"main"+12 - Missing symbol - "printf"

- In the example above, segment "main" is in file 1 segment 2. The file names associated with the file numbers are displayed when the **-p** option is used for the Link.

MakeBin:

- The **-t** and **-at** options are supported by this tool also and can be used to set the file type and aux type respectively of the output file. See the description of **-t** and **-at** under LinkIIGS.
- MakeBin now supports Load Files that are in ExpressLoad format.
- Bad Load Files and non Load Files are sensed and no longer cause unexpected results.

MakeDirect:

- The stack files in the version 1.0 Includes folders have a potential incompatibility problem and are no longer needed. They have been replaced by this new tool. Rather than linking in several stackx.obj files when you need a certain stack size, you would generate a file with a stack segment explicitly for your application and link that segment. The name of this file is "**Direct.Obj**".
- For example, if your application needs a 8K direct page and stack, the command lines used would be:

```
MakeDirect $2000
LinkIIGS Your.Obj Direct.Obj -lib Your.Lib
```

- Decimal values can be used if the **$** is left off. In the example above, **MakeDirect 8192** would create the same file.
- MakeDirect will truncate the specified segment size if it is more than four hex characters (i.e.: $1FEDCB will be treated as $EDCB).
- This special segment will override the direct page that is built into the APW C library. The stackx.obj files did not do this correctly. APW C programmers should always use MakeDirect rather than relying on the library's built-in stack segment.

NOTE: APW C programs require at least a 4K stack!

MakeLib:

- OMF 2.0 and 2.1 are now supported. OMF 1.0 libraries are converted to OMF 2.0 whenever a library is modified. OMF 2.0 libraries will be left as OMF 2.0 libraries and OMF 2.1 libraries will be left as OMF 2.1.

ResEqual:

- This new tool was added to this disk as part of the v.1.1 update. ResEqual is a port of the MPW ResEqual tool for the Macintosh.

Rez:

- Please note that **Rez** is installed as a language and not as a utility. Its language number is 21 (hex $15) and an entry in the SysTABs file should be made for this compiler.
- This new tool was added to this disk as part of the v.1.1 update. Rez is a port of the MPW Rez tool for the Macintosh.
- Two new commands were added to Rez to support Apple IIGS specific features. These two commands are **Optional** and **ReverseBytes**. Details on these two new commands can be found in the Rez Chapter of the accompanying manual.

RInclude:

- There is a new folder inside of the **Libraries** folder. Its purpose in life is to hold the interfaces for Rez and DeRez. This is the folder that Rez and DeRez will search by default for **#include** files.

Search:

- Search will return a status code of -1 if the user aborts it by pressing **COMMAND-.** (COMMAND-PERIOD).
- Search now allows the user to specify only -p on the command line (to display just the version number).
- Several new options have been added to this utility. All of these are documented in the *APW Tools Reference* manual.
- The search algorithm has been changed. It is now much faster than the version 1.0 release.

SysCmnd:

- Some of the new tools are restartable and others are not. Please refer to the **SysCmnd** file found on the **APW.Tools1** disk (in the **SYSTEM** folder) to determine which tools are restartable and which ones are not. Restartable tools will have an asterisk in the **SysCmnd** file in the second field. Restartable tools are left in memory (but are marked as purgable) so that subsequent uses are faster.
- A restartable tool may be installed as non-restartable (simply remove the asterisk and relaunch APW), but do **NOT** install non-restartable tools as restartable. The results are often unpredictable and typically cause system hangs, crashes, and other nasty effects.

SysErrs:

- The resource fork of this file contains a collection of system error messages for use by several tools in this package. If this file is missing, those tools will simply issue generic error messages instead of detailed ones.
- The data fork of this file contains the Rez source code that created the resource fork. This will allow you to study those error messages and use them from within your own utilities.
- Do not include these error message in your actual utilities, applications, etc. The correct way to utilize them is by opening the resource fork of the SysErrs file and loading the error message you need. This allows detailed error reporting by many tools without adversely affecting the amount of disk space required to support such detail.

M16.Util2:

- There are now two versions of the M16.Util macro file in the AInclude folder. The one named **'M16.Util'** is unchanged (except for bug fixes) while the one named **'M16.Util2'** is a copy of **M16.Util** with the **long** and **short** macros changed to **longmx** and **shortmx** and several stack frame management macros added. Setting the register size to 16-bits is now done by the **longmx** macro. This frees up **long** for the new purpose. For consistency, the macro **short** was also changed to **shortmx**.
- If you do not want to take advantage of the new stack macros then continue using **M16.Util** in your source files. The old usage for the macro **long** was to change the register sizes to long (16-bit). In order to take advantage of the new stack macros, you will need to change your current use of **long** and **short** to **longmx** and **shortmx**, then use **M16.Util2** instead of **M16.Util**.
- The new stack frame management macros assist in defining stack frame usage. Using these macros can make code more readable and prevent simple programming errors. The new macros are: **DSect**, **DefineStack**, **byte**, **word**, **long**, **block**, **DSectSize**, **EndLocals**, **CheckSize**.
- The new stack frame management macros are:

DSect value

This macro is used to define a dummy PC register which can be used in equates to avoid error-prone linked equates for setting up direct page or stack offsets. The dummy PC begins at value and is updated by the appropriate amount by the **byte**, **word**, **long**, and **block** macros.

DefineStack

This macro is used to set a dummy PC starting at 1 in order to define a series of stack offsets using the **byte**, **word**, **long**, and **block** macros.

label byte

This macro defines the given label with the current dummy PC, then updates the dummy PC by one byte.

label word

This macro defines the given label with the current dummy PC, then updates the dummy PC by 2 bytes.

label long

This macro defines the given label with the current dummy PC, then updates the dummy PC by four bytes.

label block value

This macro defines the given label with the current dummy PC, then updates the dummy PC by the amount indicated by **value**.

DSectSize

This macro equates the given label with the current dummy PC defined by the DSect, GlobalDSect, or DefineStack macros. This is useful for determining the size of data sections or parts of data sections.

EndLocals

This macro equates the given label with the current DummyPC-1 defined by the DSect, GlobalDSect, or DefineStack macros. This is useful for determing the stack usage of locals.

CheckSize

This macro is used to make sure that a range of equates stays within appropriate limits. A stack frame is generally less than 257 bytes. If the assumption is that it will be, and the code depends on this assumption, then the **CheckSize** macro can be used to guarantee that this limit is not exceeded. If it is exceeded, a warning will be issued.

Using M16.Util2:

- An example of using the new M16.Util2 macros:

```
                    DefineStack
wptr        long
ctlID       long
count       word
someData    block   20

stkFrameSize EndLocals

dpageptr    word
dbank       byte
retaddr     block   3

passedParam1 long
passedParam2 word

result      long

            phb             ;Save dataBank register.
            phd             ;Save directPage pointer.

            tsc             ;Make room for locals.
            sec
            sbc     #stkFrameSize
            tcs
            tcd

* your code goes here.

exit        tsc             ;Get rid of local space.
            clc
            adc     #stkFrameSize
            tcs
            pld             ;Restore directPage pointer.
            plb             ;Restore dataBank register.
            lda     1,s     ;Move return address.
            sta     6+1,s
            lda     2,s
            sta     6+2,s
            ply             ;Pull passed parameters.
            ply
            ply
            rtl             ;All done.
```

APW BUG REPORT

Date_____

APW Version_____**System Disk Version**_____

Area (circle as many as needed):

Shell
Language_____
Library
Utility_____
Performance
Documentation

Configuration:

Hard Disk Size _____MB. Brand_____
Free Space on Hard Disk_____ Memory Size_____
Floppy Disk Drives: _____3.5" _____5.25"
System Software version_____

Bug Description (please include the steps needed to duplicate the problem):

_____

_____

_____

_____

_____

_____

_____

_____

_____

Contact Information:

NAME _____ **Phone#**_____
Company Name _____
Address _____
City, State Zip_____
Country _____

Please return this completed form to:
APW Product Manager
Apple Computer, Inc.
20525 Mariani Ave., MS: 75-8X
Cupertino, CA 95014

APW INTERFACE BUG REPORT

Name of file with bug:_____

Description of the structure/record/tool call/macro/etc in error:_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

How should it be defined/declared/described/etc: _____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please return this completed form to:

APW Product Manager
Apple Computer, Inc.
20525 Mariani Ave., MS: 75-8X
Cupertino, CA 95014

Disk Contents

':APW.Tools1:' DISK
- ':APW.Tools1:F.NewStuff'
- ':APW.Tools1:NewStuff'
- ':APW.Tools1:Utilities:'
  - Compact
  - CrunchIIGS
  - DeRez
  - DiskCheck
  - DumpObj
  - Duplicate
  - Equal
  - Express
  - Files
  - Help
  - Join
  - LinkIIGS
  - MakeBin
  - MakeDirect
  - MakeLib
  - ResEqual
  - Search
- ':APW.Tools1:Utilities:Help:'
  - Canon
  - Compact
  - CrunchIIGS
  - DeRez
  - DiskCheck
  - DumpObj
  - Duplicate
  - Equal
  - Express
  - Files
  - Join
  - LinkIIGS
  - MakeBin
  - MakeDirect
  - MakeLib
  - ResEqual
  - Rez
  - Search

':APW.Tools2:' DISK
- ':APW.Tools2:F.MoreNewStuff'
- ':APW.Tools2:MoreNewStuff'
- ':APW.Tools2:Languages:'
  - Rez
- ':APW.Tools2:Libraries:RInclude:'
  - Pict.rez
  - Types.rez
- ':APW.Tools2:System:'
  - AddToSysCmnd
  - AddToSysTABs
  - SysErrs
- ':APW.Tools2:Utilities:'
  - Canon
  - Canon.Dict

':APW.Interfaces:' DISK
- ':APW.Interfaces:NewAll'
- ':APW.Interfaces:NewAPWC'
- ':APW.Interfaces:NewAsm'
- ':APW.Interfaces:AInclude:'
  - E16.ACE
  - E16.ADB
  - E16.AppleShare
  - E16.AppleTalk
  - E16.Control
  - E16.Desk
  - E16.Dialog
  - E16.Event
  - E16.Font
  - E16.GSOS
  - E16.IntMath
  - E16.LineEdit
  - E16.List
  - E16.Loader
  - E16.Locator
  - E16.Memory
  - E16.Menu
  - E16.MIDI
  - E16.MiscTool
  - E16.NoteSeq
  - E16.NoteSyn
  - E16.Print
  - E16.ProDOS
  - E16.QDAux
  - E16.QuickDraw
  - E16.Resources
  - E16.SANE
  - E16.Scheduler
  - E16.Scrap
  - E16.Shell
  - E16.Sound
  - E16.StdFile
  - E16.TextEdit
  - E16.TextTool
  - E16.Types
  - E16.Video
  - E16.Window
  - M16.ACE
  - M16.ADB
  - M16.AppleShare
  - M16.AppleTalk
  - M16.Control
  - M16.Desk
  - M16.Dialog
  - M16.Event
  - M16.Font
  - M16.GSOS
  - M16.IntMath
  - M16.LineEdit
  - M16.List
  - M16.Loader
  - M16.Locator
  - M16.Memory
  - M16.Menu
  - M16.MIDI
  - M16.MiscTool

```
    M16.NoteSeq
    M16.NoteSyn
    M16.Print
    M16.ProDOS
    M16.QDAux
    M16.QuickDraw
    M16.Resources
    M16.SANE
    M16.Scheduler
    M16.Scrap
    M16.Shell
    M16.Sound
    M16.StdFile
    M16.TextEdit
    M16.TextTool
    M16.Types
    M16.Util
    M16.Util2
    M16.Video
    M16.Window
':APW.Interfaces:CInclude:'
    ACE.h
    ADB.h
    AppleShare.h
    AppleTalk.h
    Control.h
    CType.h
    Desk.h
    Dialog.h
    ErrNo.h
    Event.h
    FCntl.h
    Font.h
    GSOS.h
    IntMath.h
    IOCtl.h
    LineEdit.h
    List.h
    Loader.h
    Locator.h
    Malloc.h
    Math.h
    Memory.h
    Menu.h
    MIDI.h
    MiscTool.h
    NoteSeq.h
    NoteSyn.h
    Print.h
    ProDOS.h
    QDAux.h
    QuickDraw.h
    Resources.h
    SANE.h
    Scheduler.h
    Scrap.h
    SetJmp.h
    Shell.h
    Sound.h
    StdFile.h
    StdIO.h
    StdLib.h
    String.h
    Strings.h
    TextEdit.h
    TextTool.h
    Types.h
    Values.h
    VarArgs.h
    Video.h
    Window.h
':APW.Interfaces:Notes:'
    New.Asm.Intfs
    New.C.Intfs
```

Apple IIGS® Programmer's Workshop Tools Reference

APPLE COMPUTER, INC.

20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and tables

Preface

The Apple® IIGS Programmer's Workshop (APW™) is Apple Computer's **native development** environment for the Apple IIGS® computer. APW is a set of programs that enable developers to create application programs on the Apple IIGS. This manual, the *Apple IIGS Programmer's Workshop Tools Reference,* describes the **APW Tools** package: a separate set of utility programs that aid in the development process. In addition to a variety of small utilities, The APW Tools include a resource compiler and decompiler, and the advanced linker (LinkIIGS), a sophisticated linker with capabilities beyond the standard linker provided with APW.

The principal APW manual, *Apple IIGS Programmer's Workshop Reference,* provides information on the APW Shell, Editor, linker, and standard utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use. That manual also provides the information necessary to write an APW utility or a language compiler or assembler for APW.

The Apple IIGS Programmer's Workshop includes several programming languages, such as 65816 assembly language and APW C. Each compiler or assembler is described in a separate manual because each language can be added to your system independently.

This manual is intended for experienced programmers and developers; that is, it assumes that you are familiar with either assembly language or a high-level programming language such as C or Pascal and with the Apple IIGS computer and the Apple IIGS operating system.

How to use this book

This manual is divided into four parts, describing in turn the utility programs, the advanced linker, the resource compiler, and error messages. It is not necessary for you to read the entire manual before you start using the APW tools; depending on the kind of programming you're doing, there may be chapters or sections of chapters that you'll never have to read at all. In particular, you will need to read about the advanced linker only if you are writing large or complex programs; you will not need this material for short or simple programs. Likewise, you will not need to read about the resource compiler unless you include resources in your programs.

Visual cues

Look for these visual cues throughout the manual:

◆ *Note:* Notes like this contain sidelights or information that you will probably find useful.

△ **Important** "Important notices" like this contain information that you should read before proceeding. △

▲ **Warning** A warning directs your attention to something that could cause loss of data or damage to the software. ▲

Boldfaced terms are defined in the glossary.

A special typeface is used for characters that you type or that can appear on the screen, such as commands, assembly-language instructions and directives, filenames, or system prompts:

```
It looks like this.
```

Icons are used in tables and command input lines to indicate the arrow keys and the Command key. If you must press two keys simultaneously, they are shown with a hyphen between them. For example, the following sequence indicates you must press the Control and Y keys simultaneously, followed by the Up Arrow key:

Control-Y ↑

- *Note:* The Apple IIGS Command key, indicated with the apple icon (), corresponds to the Open Apple key on the Apple IIe® keyboard. The Apple IIGS Option key corresponds to the Closed Apple key () on the Apple IIe keyboard. The Clear and Enter keys on the Apple IIGS keyboard have no Apple IIe equivalents.

△ **Important** On the Apple IIGS keyboard, the Reset key has a triangle (△) on it rather than the word *reset.* △

Italics are used in commands to indicate parameters that must be replaced with a value. For example, the word *pathname* in the following command refers to any valid ProDOS pathname:

`DELETE` *pathname*

If the file you want to delete is `:APW:MYPROGS:DONOTHING`, this command would be as follows:

```
DELETE :APW:MYPROGS:DONOTHING
```

Other materials you'll need

The manuals and software you need to develop applications that run on the Apple IIGS depend on the type of programming you are doing. For starters, you must be familiar with the Apple IIGS computer, including the Control Panel. The *Apple IIGS Owner's Guide* that came with your Apple IIGS describes routine operation of the computer.

The Apple IIGS does not restrict developers to a single programming language. Apple currently provides a 65816 assembler and a C compiler. Other compilers can be used with the workshop. You can write different parts of a program in different APW languages and link them into a single load file with APW.

These are the manuals for the languages Apple provides for APW:

- *Apple IIGS Programmer's Workshop Assembler Reference*
- *Apple IIGS Programmer's Workshop C Reference*

Other manuals in the Apple IIGS technical suite that may be useful in your programming include

- *Technical Introduction to the Apple IIGS*
- *Programmer's Introduction to the Apple IIGS*
- *Apple IIGS Hardware Reference*
- *Apple IIGS Firmware Reference*
- *Apple IIGS Toolbox Reference*
- *GSBug and Debugging Tools Reference*
- *GS/OS Reference*
- *GS/OS Device Driver Reference*
- *ProDOS 8 Technical Reference Manual*
- *Apple Numerics Manual*

For more information

APDA™ provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. For information about APDA, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 75-6A
Cupertino, CA 95014-6299

1-800-282-APDA or 1-800-282-2732
Fax: 408-562-3971
Telex: 171-576
AppleLink®: DEV.CHANNELS

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 75-3T
Cupertino, CA 95014-6299

Chapter 1 **The APW Tools**

The Apple® IIGS® Programmer's Workshop Shell includes the command interpreter that you use to control the Apple IIGS Programmer's Workshop, and it provides the interface between APW™ and the Apple IIGS operating system.

This chapter is a reference guide to most of the utilities supplied with the APW Tools package and accessed through the Shell as APW Shell commands. You should turn to this chapter whenever you need a full explanation of a command or command parameters for these tools. Two especially important tools, the advanced linker and the resource compiler/decompiler, are described in separate chapters. The utilities supplied with APW are described in the *Apple IIGS Programmer's Workshop Reference*.

Command types and the command table

The Apple IIGS Programmer's Workshop includes a large number of commands that perform a variety of functions, from listing a disk directory to compiling a program. There are three types of commands in APW: internal, external, and language.

- An **internal** command is one included in the APW Shell. Internal commands are resident in memory whenever the Apple IIGS Programmer's Workshop is running. They are not described in this manual.
- A **language** command changes the default APW language type. Any new file opened for editing with the `Edit` command is given the default language type. `Rez` is the only language command described in this manual.
- An **external** command is a separate APW utility program. These programs are loaded from disk when you execute the commands. Those external utilities that are sold as part of the APW Tools package are described here; others, included with APW, are described in the *APW Reference*.

The commands described in this manual are shown in Table 1-1.

- **Table 1-1** APW commands described in this manual

Command	Use	Type
`Canon`	Replace words with the canonical spelling specified in a dictionary file	External
`Compact`	Convert load file to compact form	External
`DeRez`	Decompile a resource	External
`DiskCheck`	Check a disk for block conflicts and verify the integrity of the disk's bitmap	External
`DumpObj`	List the contents of an OMF file to standard output	External
`Duplicate`	Copy files with resource forks	External
`Equal`	Compare two files or directories for equality	External
`Express`	Convert a load file to GS/OS™ ExpressLoad format	External

■ **Table 1-1** APW commands described in this manual (Continued)

Command	Use	Type
`Files`	List the contents of a directory, including subdirectories	External
`LinkIIGS`	Link an object file using the advanced linker	External
`MakeBin`	Create a ProDOS® 8 binary file from a GS/OS load file	External
`MakeDirect`	Create a direct-page/stack object segment of a specified size	External
`MakeLib`	Generate a library file from an object file	External
`ResEqual`	Compare two resources for equality	External
`Rez`	Change default language to `REZ`	Language
`Search`	Search a file for a string	External

Command descriptions

The following notation is used to describe commands:

`plain` Plain letters in Courier typeface indicate a command name or an option that must be spelled exactly as shown. The command interpreter is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters. Segment names *are* case sensitive. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered in uppercase.

italics Italics indicate a variable that you must replace with specific information, such as a pathname or file type.

directory This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *directory* must start with a colon (`:`) or a slash (`/`); if *directory* does not start with a colon, the current prefix is assumed. For example, if you were copying a file to the subdirectory `SUBDIRECTORY` on the volume `VOLUME`, the *directory* parameter would be `:VOLUME:SUBDIRECTORY:`. If the current prefix were `:VOLUME:`, you could use `SUBDIRECTORY` for *pathname*.

filename This parameter indicates a filename not including the prefix.

pathname This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. For example, if a file were named `FILE` in the subdirectory `DIRECTORY` on the volume `VOLUME`, the *pathname* parameter would be `:VOLUME:DIRECTORY:FILE`. If the current prefix were `:VOLUME:`, you could use `DIRECTORY:FILE` for *pathname*. A full pathname (including the volume name) must begin with a colon (`:`) or a slash (`/`); do not precede *pathname* with a colon if you are using a partial pathname.

`|` A vertical bar indicates a choice. For example, `+1|-1` indicates that the command can be entered as either `+1` or `-1`.

`a|b` An underlined choice is the default value.

`[]` Parameters enclosed in square brackets are optional.

... Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

The following pointers will help you use the APW Shell command interpreter:

- The command-line prompt is a number sign (`#`); when a number sign appears at the left edge of the screen followed by a cursor, you can enter a command.
- You must separate the command from its parameters by one or more spaces.
- You can use the Right Arrow key to expand command names as described in "Entering Commands" in Chapter 2 of the *APW Reference;* you can use the Up and Down Arrow keys to scroll through previously entered commands.
- There are no abbreviations for command names, except for the aliases you add to the system as described under the `Alias` command in Chapter 4 of the *APW Reference.*
- All commands and parameters, except for segment names, can be entered in any combination of uppercase and lowercase characters. Segment names *are* case sensitive. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.
- When you are calling an assembler, compiler, or linker, if there is a conflict between a parameter in a command line and a source-code command, the command-line parameter takes precedence. When neither a source-code command nor a command-line parameter has been used, the default parameter is used.
- If you fail to enter a required parameter, you may be prompted for it, as described in Chapter 2 of the *APW Reference.*
- Any of the APW commands can be placed in an Exec command file for automatic execution. Exec files are described in Chapter 5 of the *APW Reference.*

Canon

`Canon` [`-a`] [`-c` *n*] [`-i`] [`-s`] *dictionary* [*inputfile*]

This utility compares the spelling of words in the input file with words in the dictionary file. Any words in the input file that are included in the dictionary file are replaced with the canonical spelling specified in the dictionary file. The result is written to standard output (by default, the screen).

`-a` If you specify `-a`, the following characters are treated like letters by `Canon`:

```
$ % @
```

If you omit this parameter, these characters cannot be included in the character strings in the dictionary, except as leading characters for search strings (as explained below).

`-c` *n* If you specify `-c` followed by a number, only *n* characters are considered significant when matching patterns. If you do not specify this parameter, all characters are considered significant. Note that there must be space between `-c` and *n*.

`-i` If you specify `-i`, the APW `CaseSensitive` shell variable is ignored and pattern matching is case insensitive. If you omit this parameter (and the `-s` option), pattern matching is determined by the value of the `CaseSensitive` variable as explained below.

`-s` If you specify `-s`, the APW `CaseSensitive` shell variable is ignored and pattern matching is case sensitive. If you omit this parameter (and the `-i` option), pattern matching is determined by the value of the `CaseSensitive` variable as explained below.

dictionary The full pathname or partial pathname (including the filename) of the dictionary file. You can use wildcard characters but `Canon` only uses the first filename that matches the specification.

inputfile The full pathname or partial pathname (including the filename) of the input file. You can use wildcard characters in the filename.

`Canon` works like a global search-and-replace function in a text editor, except that any number of different character strings can be searched for simultaneously. The dictionary file is a text file that specifies the character strings to be replaced and the replacement strings (the canonical spellings).

Canon uses the value of the APW CaseSensitive shell variable to determine whether pattern matching will be case sensitive or not. If you define the variable as any nonzero value, Canon will perform case-sensitive pattern matching; that is, the characters in two strings being compared must match exactly, including uppercase and lowercase. If you define the value as 0 or do not define it at all, Canon will perform case-insensitive pattern matching. In this case, for example, an uppercase *S* and a lowercase *s* will match. You can override the setting of this variable with the -i and -s command-line options.

◆ *Note:* If you define CaseSensitive, be certain to export it so that Canon can use it.

The maximum length of a line in the dictionary file is 256 characters. Each string must begin with a letter and can contain any number of letters and numerals. The underscore character (_) is considered a letter by Canon. If you specify the -a option, the dollar sign ($), percent sign (%), and at symbol (@) are also considered letters. You can include comments in the dictionary file by using the pound sign character (#) to begin a comment.

Each line of the dictionary file can contain either one or two strings and an optional comment. Here is an example:

```
nil    NULL          # change all "nil" to "NULL"
```

If a line contains one string, Canon uses that string as both the search string and the replace string. For example, suppose the dictionary contains the line

```
main
```

If your search is not case sensitive (that is, the command line does not include the -s parameter), the following strings are all converted to main:

```
Main

MAIN

mAIN
```

If your search is restricted to four characters (that is, the command line includes the -c 4 parameter), the following strings are all converted to main:

```
mainstream

mainly

maintenance
```

If a line of the dictionary file contains two strings, `Canon` searches for the first string and replaces it with the second. For example, suppose the dictionary contains the line

```
Main main
```

If your search is not case sensitive (that is, the command line does not include the `-s` parameter), this line functions exactly like a line containing the single string `main`. If the search is case sensitive, however, only the string `Main` is converted to `main`. In this case, the following strings would *not* be converted to `main`:

```
MAIN

mAIN

MaiN
```

The search string can include a prefix consisting of any number of characters that are not recognized as letters or numerals by `Canon`. `Canon` replaces only the strings that match the entire search string, including the prefix, but does not replace the prefix. For example, suppose the dictionary contains the line

```
.seconds tenths
```

In this case, `Canon` would convert the string `hours.minutes.seconds` to `hours.minutes.tenths`. The string `hours/minutes/seconds` would not be changed, however.

`Canon` writes the converted file to standard output (by default, the screen). To save the result in a file you must redirect output to a pathname. For example, to process the file `MYPROG.SRC` with the dictionary `C.CONVERT`, creating the new file `MYPROG.CC`, you could use the command line

```
Canon C.CONVERT MYPROG.SRC > MYPROG.CC
```

Compact

`Compact` *infile* `[-o` *outfile*`] [-p] [-r] [-s]`

This utility converts a load file to the most compact form provided for by version 2.0 of object module format (OMF).

infile — The full pathname or partial pathname (including the filename) of the load file that you wish to compact. You can use wildcard characters in the filename.

outfile — The full pathname or partial pathname (including the filename) of the output file. If you do not specify `-o` *outfile*, *infile* is overwritten.

`-p` — If you specify the `-p` option, a progress report is written to standard output. The progress report first shows the version number of `Compact` that you are using and then shows the number of the segment being processed and the operation being performed on that segment.

`-r` — If you specify the `-r` option, any load segment named `~globals` or `~arrays` is made a reload segment. The `~globals` are forced to banksize $10000.

`-s` — If you specify the `-s` option, a summary report is written to standard output when `Compact` is finished. The summary report shows the total number of segments in the file and the number of each OMF record type compacted, copied, or created.

Press Command-period to cancel the command.

The `Compact` utility can decrease the size of load files by 20 to 70 percent and make them load up to 25 percent faster. The amount of reduction in size and loading time achieved for a particular file depends on the number and nature of symbolic references in the file.

In addition to compacting a file, the `Compact` utility converts OMF version 1.0 files to version 2.0. If you specify the `-r` option and `Compact` finds a load segment named `~globals` or `~arrays`, `Compact` makes it a reload segment. See "Command Types and the Command Table" in Chapter 4 of the *APW Reference* for a discussion of restartability. See "Object Module Format" in the *GS/OS Device Driver Reference* for a description of OMF format and reload segments.

△ **Important** To load a compacted load file you must have version 1.2 or later of GS/OS and the System Loader. △

Use `Compact` as the last step in program development, after the program has been completely debugged, to maximize the performance and minimize the size of the load file.

- *Note:* You do not have to compact load files created by `LinkIIGS` (unless you specified the `-b2` option).

DeRez

DeRez [*option...*] *resourceFile* [*resourceDescriptionFile...*]

The DeRez utility creates a text representation (resource description) of the resource fork of an Apple IIGS load file and writes it to standard output. The resource description consists of `resource` and `data` statements that can be understood by the resource compiler.

If the output of DeRez is used as input to the resource compiler, the resource compiler produces the same resource fork that was originally used as input to DeRez. DeRez is not guaranteed to be able to recreate the original resource description from a resource fork; if it can't, it produces a `data` statement instead of the appropriate `resource` statement.

See the Equal command and Chapter 3, "Resource Compiler and Decompiler," for related information.

option... You can specify as many of the following options as you wish by separating the options with spaces. If you select two mutually exclusive options (such as `-only` and `-skip`), the last one listed is used.

`-d[efine]` *macro*[=*data*]

This option defines the macro variable *macro* to have the value *data*. If *data* is omitted, *macro* is set to the null string—note that this still means that *macro* is defined. Using the `-d` option is the same as writing

`#define` *macro*[*data*]

at the beginning of the input. The `-d` option may be repeated any number of times.

`-e[scape]` When this option is specified, characters that are normally escaped (such as `\0xff`) are no longer escaped. Instead they are printed as extended Apple IIGS characters. (*Note:* Not all fonts have all the characters defined.) Normally characters with values between $20 and $7E are printed as Apple IIGS characters. With this option, however, all characters (except null, newline, tab, backspace, form feed, vertical tab, and rubout) are printed as characters, not as escape sequences.

`-i` This option lets you specify one or more pathnames to search for `#include` files. It may be specified more than once. The paths will be searched in the order they appear on the command line.

```
DeRez -i 13:newrinclude -i :hd:tools...
```

`-m[axstringsize]` ***n***

This option sets the maximum string size to ***n***; ***n*** must be in the range 2–120. This setting controls string width in the output.

`-o` ***filename***

This option directs output to the specified filename. `DeRez` creates the file with a filetype of `SRC` and a language stamp of `REZ`. If ***filename*** is an existing file, `DeRez` will change its filetype and language stamp and overwrite the file without warning. Note that this option is incompatible with command-line redirection.

`-only` ***typeExpr*** [(***ID1*** [:***ID2***])]

This option reads only resources of resource type ***typeExpr***. If an ID or range of IDs is given, it reads only those resources for the given type. This option may be repeated.

`-only` ***type***

This is a simpler version of the option just described: No quotation marks are needed to specify a literal type as long as it starts with a letter. Items such as escape characters are not allowed. For example,

```
DeRez -only 0x8009 ...
```

- *Note:* The `-only` option cannot be specified together with the `-skip` option.

`-p`

This option displays progress and version information.

`-rd`

This option suppresses warning messages if a resource type is redeclared.

`-s[kip]` ***typeExpr*** [(***ID1*** [: ***ID2***])]

This option skips resources of type ***typeExpr*** in the resource file. See the note under `-only`. The `-s` option may be repeated any number of times.

`-s[kip]` ***type***

This is a simpler version of the `-s` option; no quotation marks are needed to specify a literal as long as it starts with a letter.

`-u[ndef]` ***macro***

This option undefines the macro variable ***macro***. This is the same as writing

```
#undef macro
```

at the beginning of the input file. It is meaningful to undefine only the preset macro variables. This option may be repeated.

resourceFile

A file that contains the resource fork of an Apple IIGS load file.

resourceDescriptionFile

A **resource description file** is a file of type declarations in the format used by the Resource Compiler. You can use one or more resource description files to give optional formatting information to `DeRez`. If no resource description file is specified, the output consists of `data` statements giving the resource data in hexadecimal form, without any additional format information.

`DeRez` ignores all `include` (but not `#include`), `read`, `data`, and `resource` statements found in the resource description file. (It still parses these statements for correct syntax.)

For the format of resource type declarations, see Chapter 3, "Resource Compiler and Decompiler."

The following search rules are applied for all resource description files on the command line:

1. `DeRez` tries to open the file with the name specified "as is."
2. If rule 1 fails and the filename contains no colons or does not begin with a colon, `DeRez` appends the filename to each of the pathnames specified by any `-i` options and tries to open the file.
3 `DeRez` looks for the file in the `13:RInclude` directory.

Here is an example of a command that displays all of the menu resources (type $8009) used by the file `foo`.

```
DeRez "foo" -only 0x8009 Types.rez
```

DiskCheck

`DiskCheck` *volume*|*device*

This external command scans the disk for active files and lists all block allocations, including both data and resource forks of any extended file types. It notifies you of block conflicts, where two or more files are claiming the same block(s), and provides an opportunity to list the blocks and files involved. Finally, it verifies the integrity of the disk's bitmap. Bitmap errors are reported, and you can choose to repair the bitmap.

volume|*device* The GS/OS volume name or device name of the disk to check. The volume name can be specified with or without a beginning colon; for example,

```
DiskCheck :HARDISK
DiskCheck HARDISK.
```

A device name requires a period before the name—for example, `.SCSI1.`

◆ *Note:* `DiskCheck` only verifies a ProDOS volume. It does not work with any non-ProDOS volume, such as an HFS volume.

In normal display mode data scrolls continuously on the screen. While `DiskCheck` is running, press the Space bar to place `DiskCheck` in single step mode. In this mode block allocations are displayed one at a time each time the Space bar is pressed. Press Return to return to normal display mode.

`DiskCheck` checks volumes with up to 65,535 blocks of 512 bytes (32 MBy).

`DiskCheck` makes the following assumptions:

- Blocks 0 and 1 are always in use and contain boot code.
- Enough disk integrity exists to make a `GetFileInfo` call on the volume.
- Block 2 is the beginning of the volume directory and contains valid information regarding the number of blocks, bitmap locations, entries per block, and entry size.

- All unused bytes at the end of the last bitmap block are truly unused; that is, they will be set to 0 whenever the bitmap is repaired.

△ **Important** `DiskCheck` may not catch invalid volume header information as an error. Likewise, `DiskCheck` does not check all details of the directory structures. Therefore, if large quantities of errors are displayed, it is likely that the volume header information or directory information is at fault. △

DumpObj

`DumpObj` [*option*...] *pathame*

The `DumpObj` tool writes the contents of an object file to standard output. The default format for the listing is object module format (OMF) operation codes and records. You can also list the file in 65816 assembly language or as hexadecimal codes.

pathname — The full or partial pathname (including the filename) of the file you wish to dump. The file may be a library file, the output of an APW assembler or compiler, an Apple IIGS load file, or any other file that conforms to Apple IIGS object module format. `DumpObj` does not check the file contents or the file type to ensure that it is an OMF file.

Press the Space bar to pause the `DumpObj` listing, and press any key to start the listing again.

Press Command-period to cancel the `DumpObj` listing and return to the shell.

If the file consists of more than one segment, each segment is listed separately. Each segment listing begins with the segment header followed by the segment body. The segment header is displayed in three columns, as follows:

- The first column shows the name of the header field.
- The second column gives the hexadecimal value of the field, except for the load-segment name and object-segment name, which are shown as ASCII characters.
- The third column shows the decimal value of numeric fields, except for the segment kind, for which the segment type and attributes are listed. The third column is blank for the segment name fields.

A typical segment header is shown in Figure 1-1. The differences between headers in different versions of the OMF are as follows:

- Version 1.0 has a Language Card field that is not present in later versions.
- Versions 2.0 and 2.1 have a Byte Count field instead of the Block Count field in version 1.0.
- Version 2.1 has a Revision field and a TempOrg field not present in earlier versions.

The fields in the segment header are described in the *GS/OS Device Driver Reference*.

■ **Figure 1-1** Sample DumpObj segment header (OMF version 2.1)

```
Byte count      : $0000071d                          1821
Reserved space  : $00000000                             0
Length          : $00000469                          1129
Kind            : $0000                 static code segment
Label length    : $00                                   0
Number length   : $04                                   4
Version         : $02                                   2
Bank size       : $00010000                         65536
Org             : $00000000                             0
Alignment       : $00000000                             0
Number sex      : $00                                   0
Revision        : $01                                   1
Segment number  : $0001                                 1
Entry           : $00000000                             0
Disp to names   : $002c                                44
Disp to data    : $0037                                55
TempOrg         : $00000000                             0
Load name       : DIALOG
Segment name    : WarningDialog
```

The format in which the body of the segment is shown depends on the option used. For example, a typical OMF segment dump (the default format) is shown in Figure 1-2.

◆ *Note:* The OMF dump is provided to aid in the debugging of compilers and other tools that manipulate OMF files. If you are not highly familiar with OMF, the OMF `DumpObj` listing will not be of much use to you. You can, however, use the other options provided to examine the contents of an object file in 65816 assembly-language format or as hexadecimal codes.

The first column of an OMF segment dump shows the actual displacement into the segment, in bytes, of that record. For example, if the segment header takes up 62 bytes (that is, it ends at byte $3D), the first record in the segment starts at $3E. The second column shows the setting of the program counter for that segment, that is, the cumulative number of bytes that the linker will create in the load file. The third and fourth columns show the record type and operation code of the OMF record shown on that line. The last column shows the contents of the record.

For CONST and LCONST records, the contents of the record are shown on the line or lines following the record type and operation code. The first two columns of these continuation lines show the displacement into the segment and the value of the program counter. The program counter value is followed by up to 16 bytes of hexadecimal data, listed in four columns of 4 bytes each. The hexadecimal data is followed by the ASCII characters that correspond to these data bytes. Nonprintable characters are displayed as periods (`.`).

Unless you select the `-t` option, expressions are shown in postfix form: the values being acted on are written first, followed by the operator. OMF format is described in the *GS/OS Device Driver Reference.*

■ **Figure 1-2** `DumpObj` OMF-format segment body

```
000041 000000 | CONST    ($0a) |
000042 000000 | 0b3b38e9 f5005b69  f000              | .;8...[i..
00004c 00000a | CONST    ($04) |
00004d 00000a | 1ba5fb8f                            | ....
000051 00000e | LEXPR    ($f3) |  3 : _lpb
00005a 00000e | CONST    ($07) |
00005b 00000e | a6ffa5fd 488a8f                      | ....H..
000062 000015 | LEXPR    ($f3) |  3 : _lpb 0004  +
000071 000015 | CONST    ($02) |
000072 000015 | 688f                                | h.
000074 000017 | LEXPR    ($f3) |  3 : _lpb 0002  +
000083 000017 | CONST    ($06) |
000084 000017 | 22a800e1 0a00                       | ".....
00008a 00001d | LEXPR    ($f3) |  4 : _lpb
000093 00001d | CONST    ($01) |
000094 00001d | af                                  | .
000095 00001e | LEXPR    ($f3) |  3 : _lpb 0004  +
0000a4 00001e | CONST    ($02) |
0000a5 00001e | aaaf                                | ..
0000a7 000020 | LEXPR    ($f3) |  3 : _lpb 0002  +
0000b6 000020 | CONST    ($03) |
0000b7 000020 | da4822                              | .H"
0000ba 000023 | LEXPR    ($f3) |  3 : ptocstr
0000c6 000023 | CONST    ($02) |
0000c7 000023 | 7a7a                                | zz
0000c9 000025 | CONST    ($08) |
0000ca 000025 | 7b1869f5 001b2b6b                   | {.i...+k
0000d2 00002d | END      ($00) |
```

If you specify the `-l` option, LCONST records are dumped in a shortened format. In this format, only two lines of data are dumped. If the record data is longer than two lines, the first and last lines of the data and a line in between containing three dots are dumped. (See Figure 1-3.)

■ **Figure 1-3** DumpObj LCONST record short format

```
000040 000000 | LCONST      ($f2) | $0000a5ff
000045 000000 | 48e220a9 0048abc2  20688d04 008e1400 | H. ..H.. h.......
000045 000000 |   ...
00a635 00a5f0 | fa86f205 f2a87b18  69f3001b 982b6b   | ......{.i....+k
```

For SUPER records the first line contains the word SUPER, followed by the operation code, length, and type. The lines after this contain one of the following:

- the skipped page range, the actual data in the file in parentheses, and the number of pages skipped
- the number of the current page, the number of offsets, and the offsets (one or more lines per page

See Figure 1-4 for an example of a SUPER record dump.

■ **Figure 1-4** DumpObj SUPER record format

```
00a66f 00a5ff | SUPER       ($f7) | 00000099 | RELOC2
00a675 00a5ff | 00-10 (91) 11 pages skipped
00a676 00a5ff | 11    (05) 606264666876
00a67d 00a5ff | 12-1b (8a) 0a pages skipped
00a67e 00a5ff | 1c    (27) 6a6c6e70727476787a7c7e8082848688
00a68e 00a5ff |            8a8c8e90929496989a9c9ea0a2a4a6a8
00a69e 00a5ff |            aaacaeb0b2b4b6b8
00a6a7 00a5ff | 1d    (01) 0d19
00a6aa 00a5ff | 1e-54 (b7) 37 pages skipped
00a6ab 00a5ff | 55    (00) 6c
00a6ad 00a5ff | 56-63 (8e) 0e pages skipped
00a6ae 00a5ff | 64    (00) cd
00a6b0 00a5ff | 65-69 (85) 05 pages skipped
00a6b1 00a5ff | 6a    (03) ced0d2df
00a6b6 00a5ff | 6b-6e (84) 04 pages skipped
00a6b7 00a5ff | 6f    (03) d6d8dae7
00a6bc 00a5ff | 70-7d (8e) 0e pages skipped
00a6bd 00a5ff | 7e    (03) a5a7a9ba
```

A typical assembly-language dump (-d option) appears in Figure 1-5. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The second column shows the setting of the program counter for that segment, that is, the cumulative number of bytes that the linker will create in the load file. The third column shows the disassembled instructions, and the fourth column shows the data for the insturctions if any exist. At the start of the disassembly, DumpObj assumes that the 65816 processor is running in full native mode; that is, that the accumulator, index registers, and all memory references are 16 bits; you can use the -i and -m options to change these defaults.

■ **Figure 1-5** `DumpObj` disassembly-format segment body

```
000042 000000 |    PHD
000043 000001 |    TSC
000044 000002 |    SEC
000045 000003 |    SBC    #$f5
000048 000006 |    TCD
000049 000007 |    ADC    #$f0
00004d 00000a |    TCS
00004e 00000b |    LDA    $fb
000050 00000d |    STA    _1pb
00005b 000011 |    LDX    $ff
00005d 000013 |    LDA    $fd
00005f 000015 |    PHA
000060 000016 |    TXA
000061 000017 |    STA    4 + _1pb
000072 00001b |    PLA
000073 00001c |    STA    2 + _1pb
000084 000020 |    JSL    $e100a8
000088 000024 |    DC     X'a'
00008a 000026 |    DC     AL4(_1pb)
000094 00002e |    LDA    4 + _1pb
0000a5 000032 |    TAX
0000a6 000033 |    LDA    2 + _1pb
0000b7 000037 |    PHX
0000b8 000038 |    PHA
0000b9 000039 |    JSL    ptocstr
0000c7 00003d |    PLY
0000c8 00003e |    PLY
0000ca 00003f |    TDC
0000cb 000040 |    CLC
0000cc 000041 |    ADC    #$f5
0000cf 000044 |    TCS
0000d0 000045 |    PLD
0000d1 000046 |    RTL
```

If you select the `-x` option, the segment body is displayed in hexadecimal format. A typical hexadecimal segment dump appears in Figure 1-6. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The second column is not used and contains all 0's. The next four columns show the next 16 bytes in the file in hexadecimal. The last column shows the ASCII equivalents of those bytes (nonprintable ASCII characters are displayed as periods (`.`). The hexadecimal dump starts with the first byte after the segment header (unless you specify the `-h` option, in which case the segment header is included in the hexadecimal dump), and ends at the last byte before the next segment header.

The `-xa` option displays the entire file in hexadecimal format, without distinguishing between segment headers and segment bodies, and without putting breaks between segments.

■ **Figure 1-6** `DumpObj` hexidecimal-format segment body

```
00003D  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00004D  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00005D  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
00006D  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
00007D  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00008D  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00009D  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
0000AD  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
0000BD  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
0000CD  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
0000DD  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
0000ED  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
0000FD  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00010D  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00011D  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
00012D  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
00013D  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00014D  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00015D  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
00016D  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
00017D  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00018D  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00019D  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
0001AD  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
0001BD  000000  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
0001CD  000000  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
0001DD  000000  0004DA5A 4820ED02 83057E43 4F555400  |     ZZH m    ~COUT
0001ED  000000  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)    ` COND
0001FD  000000  E40444                               |  d D
```

option... You can specify as many of the following options as you wish, but be careful not to set two mutually exclusive options (such as `-x` and `-d`), because the results may not be what you expect. If an option can't function due to the other options set, it is ignored. For example, if you select `-h` to suppress segment headers and also specify `-s` to select short headers, the `-s` is ignored.

`-a` This option suppresses all information but the operation codes and operands for each line of an OMF-format or 65816-format disassembly. The default is to include the displacement into the file and the program counter for each line at the beginning of the line. You can use `-a` to produce APW Assembler source code.

- *Note:* Some things in the file dump, such as relocation information embedded in the code of a load file, will not be correct as APW Assembler source code. For example, after processing by the linker, a load file might contain the following `JSL`:

  ```
  JSL 123456
  ```

  However, as assembler source code after the disassembly, this code does not actually mean a `JSL` to that address.

`-d` This option writes the file dump as a 65816 disassembly rather than as OMF records.

`-h` This option lists the headers as hexadecimal codes.

`-i` For 65816 assembly-language listings, this option assumes that the CPU is set to short index (X and Y) registers at the start of the disassembly, rather than starting in full native mode. The option has no effect on OMF-format and hexadecimal listings.

`-l` This option displays only the first and last lines of an LCONST record. It lets you see the structure of an OMF file without listing all the data in the file.

`-m` For 65816 assembly-language listings, this option assumes that the CPU is set to short memory (accumulator) registers at the start of the disassembly rather than starting in full native mode. The option has no effect on OMF-format and hexadecimal listings.

`-mpw` This option dumps the file with MPW™ IIGS syntax. The default is to dump the file with APW syntax.

`-n "`*seg1 seg2 ...*`"` This option processes only the segments specified, where *seg1, seg2,* and so forth are the names of the segments you wish to process. Segment names must be entered exactly as they appear in the source code, including the case of the characters. If you specify more than one segment, you must enclose the string of segment names in quotes. You can use more than one `-n` option on a command line.

To get a list of segments in the file, use the `-o` and `-s` options with the `DumpObj` command.

`-o` This option doesn't show the contents of the segments; that is, it lists the headers only.

`-p` This option writes progress information, the version number of `DumpObj`, and the date and time to standard error output.

`-s` This option writes only the byte count, length, version, org, segment type (kind), revision, and segment name for the segment headers. The default is to include all of the information in the segment header.

`-t` This option writes expressions in infix format; for example, 4 minus 2 is written as 4 – 2. The default is postfix form; for example, 4 minus 2 is written as 4 2 –.

`-x` This option writes the file dump in hexadecimal codes rather than as OMF records. Segment headers are always printed in ASCII text unless you also select the `-h` option.

`-xa` This option writes the entire file in hexadecimal, without any other formatting. The only difference between the `-xa` option and the combination of the `-x` and `-h` options is that the `-xa` option does not skip lines between segment headers and segments.

Duplicate

`Duplicate [-p] [-r | -d]` *pathname1* `[`*pathname2*`]`

`Duplicate` copies the file specified by *pathname1* into a file specified by *pathname2*. Unlike the APW `Copy` command, `Duplicate` can copy the resource fork as well as the data fork of a file.

▲ Warning The APW `Copy` command does not copy resource forks. It automatically strips them off without warning. ▲

`Duplicate` accepts only legal ProDOS pathnames. It does not expand APW prefixes (such as `2/`) or the directory-walking operator (`../`). It does not support wildcards in pathnames. If the source or destination pathname contains spaces, the pathname must be enclosed in quotes. If the pathname is quoted, the quotes must surround the entire pathname. The source pathname must be specified; the destination pathname is optional.

`Duplicate` copies only one file at a time.

`-p` Displays the copyright notice and version number of the `Duplicate` tool itself.

`-r` Copies the resource fork only. (If neither `-r` nor `-d` is specified, `Duplicate` copies both forks.)

`-d` Copies the data fork only. (If neither `-r` nor `-d` is specified, `Duplicate` copies both forks.)

pathname1 The full or partial pathname (including the filename) of the source file (the file you wish to copy from).

pathname2 The full or partial pathname of the destination file (the file you wish to place the copy into). If *pathname2* specifies a directory, the file of *pathname1* is copied into the directory of *pathname2* and given the same filename as it had in *pathname1*. If *pathname2* is omitted, the file of *pathname1* is copied into the current directory and given the same filename as it had in *pathname1*.

- *Note:* If *pathname2* is omitted, and if the directory specified by *pathname1* is the same as the current directory (meaning that you would be copying the file onto itself), `Duplicate` disallows the copy and returns an error.

Equal

`Equal` [*option...*] *pathname1 pathname2*

The `Equal` utility compares two files or directories for data equality and can show differences in file dates or types. `Equal` compares both the resource and data forks of a file unless the `-f` option is used to restrict the comparison to one or the other.

option ... You can specify as many of the following options as you wish by separating the options with spaces.

`-d` If you specify `-d`, `Equal` does not compare the creation and modification dates and times of files. If you do not include this option, creation and modification dates and times are compared.

`-f resource|data`
This option compares the resource or the data fork of a file only. If this option is not used, the default compares both the resource and data forks.

`-m` If you specify `-m`, `Equal` does not list the names of missing files: that is, files that are present in one of the directories you listed but not in the other. If you do not include this option, missing files are listed.

`-n` *n* This option displays the first *n* mismatched bytes. If you specify a value for this option, the output for each file stops after *n* bytes of the files that do not match have been listed. If you set *n* to 0, no mismatches are displayed. If you do not select this option, the display stops after ten mismatches for each file. Note that there must be space between `n` and *n*.

`-p` If you specify `-p`, `Equal` shows progress information. Progress information consists of brief messages that tell you what the utility is currently working on: for example, which subdirectory is currently being processed. If you do not include this option, progress information is not produced.

`-t` If you specify `-t`, `Equal` does not compare the file types of files. If you do not include this option, file types are compared.

pathname1 pathname2
The full or partial pathnames of the two directories or files that you want to compare. If you name two directories, all files and subdirectories in the two directories are compared.

You can use `Equal` to determine whether two files are identical or whether the contents of two directories are the same. If you list the pathnames of two directories, a file-by-file comparison is made of the directories. `Equal` reads the filename, file type, and creation and modification dates of a file in the first directory and then checks to see if a file of the same name exists in the second directory. If it does, `Equal` compares the files byte by byte, leaving the files and going on to the next pair after listing 10 bytes if they are not identical. You can set options to suppress the comparison of file types, to suppress the comparison of file dates and file times, and to specify a different number of bytes to compare before going on to the next file. By default, `Equal` lists any filenames of files that exist in one file but not in the other. You can suppress that output as well.

By specifying filenames instead of directory names, you can compare two files with different filenames.

Express

Express [-p] *infile* -o *outfile*

The Express tool reformats an Apple IIGS load file so that it can be loaded by the ExpressLoad routine that is provided on the GS/OS 5.0 system disk. When loaded by ExpressLoad, these files will load much faster than when loaded by the standard System Loader. Files reformatted for use by ExpressLoad can still be loaded by the System Loader under GS/OS 4.0 and ProDOS 16, version 3.2.

Express works only with files in OMF version 2.0 load file format. You can use DumpObj to determine the OMF version number of your file. You can use compact to convert a pre-OMF 2.0 file into OMF 2.0.

infile	The full or partial pathname of a load file.
-o *outfile*	The -o *outfile* parameter identifies the file to which output is written. This parameter is required.
-p	If you specify this option, Express displays progress information. If you omit it, progress information is not displayed.

ExpressLoad does not support multiple load files; therefore, you cannot use Express with any program that references segments in run-time library files.

If Express is unable to convert a Relocation record for use by ExpressLoad, it will quit with error status. However, if you relink your object files using either the APW 2.0 Linker or the advanced linker (LinkIIGS), without the -x option (that is, with the default settings), the linker will create a load file that will already be in ExpressLoad format.

The following GS/OS system loader calls are not supported by ExpressLoad:

- GetLoadSegInfo ($0F). The internal data structures of ExpressLoad are not the same as those of the System Loader.
- LoadSegNum ($0B). Because Express changes the order of segments in the load file, an application that uses this call and has been converted by Express cannot be processed by the System Loader. Use the LoadSegName function instead.
- UnloadSegNum ($0C). Because Express changes the order of segments in the load file, an application that uses this call and has been converted by Express cannot be processed by the System Loader. Use the UnloadSeg ($0E) function instead.

Files

`Files` [*option...*] [*pathname...*]

`Files` is an APW utility that can display the contents of a directory or subdirectory on a GS/OS storage device. `Files` augments the `Catalog` command by providing filter facilities and many more options to format the directory listing.

With `Files` you can list the full contents of a directory, including the contents of all included subdirectories, and use options to supply as little or as much information about the file as you wish. You can use additional options to limit the display to files of a specified filetype or auxiliary type, or to files whose names contain a specified string.

option... — You can specify as many of the following options as you wish on the command line:

`-a` — If you specify `-a`, `Files` displays all the files in a directory, including files that have the invisible bit set in their access attribute. If you do not specify `-a`, invisible files are not displayed.

`-at` *auxtype* — If you specify this option, `Files` displays only those files whose auxiliary type matches the one you have specified. If there are no matches, no files are displayed. You can specify this option more than once.

You can specify *auxtype* as a decimal or hexadecimal number. Precede hexadecimal numbers with a `$` character. The value of type must be 0 through 65535 ($00 through $FFFF) inclusive.

`-b` — If you specify `-b`, `Files` places a backslash (\) character at the end of each output line. The backslash is useful when you are using `Files` to generate lists of files to be used in shell scripts (Exec files). You cannot use this option with the `-l` or `-x` options.

`-c` *value* — With this option you can specify the number of columns to use when displaying the filenames of a directory. By default `Files` displays its output in a one-column listing. You can specify from 1 to 15 columns. You cannot use this option with the `-l`, `-x`, or `-r` options. If you use this option with the `-d` option, the `-d` option will be ignored without a warning.

`-d` — If you specify the `-d` option, `Files` displays the full GS/OS pathname to the directory entry being displayed. Normally `Files` displays only the filename portion of a full pathname. This option is ignored if either the `-l` or `-x` option is also used.

`-f` *string*

If you specify this option, `Files` displays only those files whose filenames contain the characters specified in *string*. To include one or more space characters in the search string, enclose *string* in quotation marks. You can specify this option more than once.

`-h`

If you specify `-h`, `Files` displays the filetype and all numeric information in hexadecimal format. By default, `Files` displays the filetype using the standard 3-character mnemonics used by the `Catalog` command, and displays numeric values in decimal notation.

- *Note:* The auxiliary type is an exception to displaying numerics in decimal notation. `Files` always displays the auxiliary type in hexadecimal format.

`-i`

If you specify `-i`, `Files` treats a directory as a file; that is, it lists directory names only, not the contents of directories.

`-l`

If you specify `-l`, `Files` lists detailed information about each file. The order and contents of the displayed fields are identical to the display of the `Catalog` command. By default, `Files` displays only filenames. You cannot use this option with the `-b`, `-c`, or `-m` options.

`-m` *value*

With this option you can specify the number of columns to use when displaying the filenames of a directory. By default `Files` displays its output in a one-column listing. You can specify from 1 to 15 columns. You cannot use this option with the `-l`, `-x`, or `-r` options. If you use this option with the `-d` option, the `-d` option will be ignored without a warning. Note that this option is the same as the `-c` *value* option; it is provided for compatibility with the MPW environment.

`-n`

This option is valid only with the `-l` or `-x` options. If you specify `-n`, `Files` does not display column headings. Normally when you specify `-l` or `-x`, `Files` does displays column headings.

`-p`

If you specify this option, `Files` will display its copyright notice and version number.

`-q`

If you specify this option, `Files` will not put quotes around filenames containing spaces. By default, when `Files` displays a filename containing one or more spaces, the filename will be in quotes.

`-r`

If you specify this option, `Files` processes subdirectories contained in directories recursively; that is, `Files` will display the contents of subdirectories (and sub-subdirectories, etc.). As each subdirectory is opened and displayed, the filename field of the display will be indented one character, which makes it easier to visualize the hierarchical structure of the directories.

By default, `Files` displays only the contents of the directory named on the command line. You cannot use this option with the `-c` option.

`-t` *filetype*

If you specify this option, `Files` displays only those files whose filetype matches the one you have specified. If there are no matches, no files are displayed. You can specify this option more than once.

You can specify *filetype* as a decimal or hexadecimal number, or a 3-character mnemonic. Precede hexadecimal numbers with a `$` character. The value of *filetype* must be 0 through 65535 ($00 through $FFFF) inclusive. If you specify a mnemonic, `Files` searches its internal table for the matching numeric filetype. If `Files` cannot find the mnemonic, it will abort with an error message.

`-x` *string*

With this option you can specify exactly which fields of a directory entry to display and in what order. You cannot use this option with `-b`, `-c`, `-d`, or `-m`.

Use the following characters for *string* to specify which fields to display. Do not put spaces between the characters in *string*.

`a`	displays auxiliary type of file
`b`	displays total number of blocks of disk storage used by file
`c`	displays creation date and time of file
`d`	displays number of blocks used by data fork of file
`e`	displays number of bytes contained in data fork of file (eof)
`f`	displays access attribute flags of file
`m`	displays modification date of file
`q`	displays number of bytes contained in resource fork of file
`r`	displays number of blocks used by resource fork of file
`s`	displays file system that manages the file
`t`	displays filetype of file
`z`	displays all the fields

If you display all possible fields, the output device must be capable of displaying 136 columns of output.

pathname... A valid GS/OS pathname. You can specify more than one pathname. If the specified pathnames contain spaces, you must enclose them in a pair of double quotation marks (" "). `Files` will process each pathname in the order that it was entered on the command line. If you do not specify a pathname, `Files` will display the contents of the current directory. `Files` determines whether each *pathname* is a directory or a file. If it is a file, `Files` displays the directory entry for the named file (including all fields that you requested with the command-line options). If the specified file is a directory file, `Files` will read and display the contents of the directory (unless you specified the `-i` option).

The following examples show some of the typical uses for the `Files` command-line options. In all of the examples assume that the current directory is set to `:HD:APW:`:

This example shows the simplest use of `Files`; that is, if you type `files`, you get the listing:

```
APW.SYS16
System
Languages
Libraries
Utilities
Source.ASM
Source.C
Source.Pascal
```

The following example shows the use of the `-l` option, when reading a directory of an AppleShare volume. The `-l` option lists detailed information about the files in the directory. Notice these points about this example:

- The volume name is quoted on the command line because it contains spaces.
- Filenames in the listing that contain spaces are quoted.
- In the listing some of the directories show a block allocation of 0. This means that the user does not have the necessary access privileges to access the contents of the directory.
- In the listing, filenames that are too long to fit in the field width are truncated and an equal sign (=) is displayed.

Here are the command and the output it generates.

```
Files -l ":DSG Server"

Filename        Type  Blocks     Modified         Created       Access    Aux

":DSG Server:"
3,861,418       DIR      0  15 Jun 89 15:33  15 Jun 89 15:33  DNBW    $0000
APW             DIR      1   3 May 89 11:37  23 Mar 89 14:01  DNBWR   $0000
Discipline      DIR      1   2 Jun 89 10:44  24 Apr 89 16:19  DNBWR   $0000
"For Craig"     DIR      0   7 Jun 89 14:59   7 Jun 89 14:58  DNBW    $0000
Groups          DIR      1  27 Feb 89 15:00   8 Nov 87 12:06  DNBWR   $0000
"In Box"        DIR      0  26 Apr 89 17:19  26 Apr 89 17:19  DNBW    $0000
MacsBug         DIR      1  10 Jun 89 10:17   6 Jun 88 14:24  DNBWR   $0000
MAX             DIR      1  15 May 89 12:10   8 Nov 87 12:15  DNBWR   $0000
Misc.           DIR      1  18 May 89  8:17   8 Nov 87 12:11  DNBWR   $0000
MPW             DIR      2  16 Jun 89 15:04   8 Nov 87 11:55  DNBWR   $0000
MultiFinder     DIR      1   4 May 89 17:02   4 Jan 89  8:37  DNBWR   $0000
"Out Box"       DIR      0  26 Apr 89  9:29  26 Apr 89  9:29  DNBW    $0000
PIIGSIncludes   DIR      0  10 Jun 89 10:15  19 Jan 89 20:44  DNBW    $0000
"Project Coor=" DIR      1  27 Apr 89 15:23  25 Jan 89 12:11  DNBWR   $0000
ResEdit         DIR      1  23 Apr 89 20:04  19 Apr 88 10:33  DNBWR   $0000
SADE            DIR      0   6 Jun 89 19:44  24 Oct 88 10:33  DNBW    $0000
Trash           DIR      0  22 May 89 13:52  22 May 89 13:52  DNBW    $0000
"Trash Folder"  DIR      1  30 May 89  9:51  20 Dec 88 13:43    BWR   $0000
```

The following example shows a recursive directory listing. Notice these points about this listing:

- In the listing a plus sign after a file type signifies that the file is an extended file; that is, it contains a resource fork and as well as a data fork.
- Subdirectories are indented.
- Equal signs indicate that filenames have been truncated to fit the display.

Here is the command that generates recursive output, followed by the output.

```
Files -x tafmc -r *:system
```

```
Filename          Type  Aux   Access      Modified          Created

system:
Start.GS.OS       P16   $0018 DNBWR   12 Jun 89 20:32  21 May 89 20:42
GS.OS             P16   $0000 DNBWR   12 Jun 89 20:33  21 May 89 20:42
GS.OS.Dev         LOD   $0000 DNBWR   12 Jun 89 20:32  21 May 89 20:39
Error.Msg         LOD   $0100 DNBWR    6 Jun 89  7:57  21 May 89 20:33
ExpressLoad       P16   $0000 DNBWR   14 Jun 89 18:11  21 May 89 20:33
Drivers           DIR   $0000 DNBWR   15 Jun 89 12:02   9 Nov 88 14:49
 SCSIHD.DRIVER    DVR   $013F DNBWR   12 Jun 89 15:49  21 May 89 20:35
 SCSI.MANAGER     DVR   $0140 DNBWR   12 Jun 89 15:50  21 May 89 20:47
 AppleDisk3.5     DVR   $0102 DNBWR    6 Jun 89 16:21  21 May 89 20:22
 AppleDisk5.25    DVR   $010E DNBWR    6 Jun 89  7:48  21 May 89 20:23
 Console.Driver   DVR   $0101 DNBWR    6 Jun 89  7:57  21 May 89 20:33
 ImageWriter      DVR   $0001 DNBWR   14 Jun 89  6:35  21 May 89 21:14
 Printer          DVR   $0002 DNBWR    6 Jun 89  8:40  21 May 89 21:16
 Modem            DVR   $0002 DNBWR    6 Jun 89  8:40  21 May 89 21:17
 ATalk            DVR   $0110 DNBWR    6 Jun 89  7:51  21 May 89 20:26
 IWEM             TXT   $0000 DNBWR    6 Jun 89  8:44  20 Jan 88 16:13
 SCC.Manager      DVR   $0140 DNBWR    6 Jun 89  8:10  21 May 89 20:46
 ATP1.ATROM       DVR   $0201 DNBWR   13 Jun 89 18:44  21 May 89 20:30
 ATP2.ATRAM       DVR   $0203 DNBWR   13 Jun 89 18:43  21 May 89 20:29
 AppleTalk        DVR   $0003 DNBWR    6 Jun 89  8:41  21 May 89 21:17
 Printer.Setup    BIN   $0000 DNBWR   15 Jun 89 12:02  25 Jul 88  7:18
 LaserWriter      DVR   $0001 DNBWR   14 Jun 89 15:02  21 May 89 21:13
FSTs              DIR   $0000 DNBWR   15 Jun 89 11:20  10 Nov 88 10:38
 Pro.FST          FST   $0000 DNBWR   14 Jun 89 12:07  21 May 89 20:45
 Char.FST         FST   $0000 DNBWR    6 Jun 89  7:57  21 May 89 20:32
 AppleShare.FST   FST   $0000 DNBWR   12 Jun 89 14:03  21 May 89 20:25
System.Setup      DIR   $0000 DNBWR   16 Jun 89 18:10   9 Nov 88 14:46
 Debug.Init       PIF   $0000 DNBWR    3 Mar 89 19:40   3 Mar 89 18:45
 Tool.Setup       PIF   $0000 DNBWR   21 May 89 20:49  21 May 89 20:49
 TS2              LOD   $0000 DNBWR   13 Jun 89 21:03  21 May 89 22:26
 TS3              LOD   $0000 DNBWR   12 Jun 89 20:34  21 May 89 22:29
 RESOURCE.MGR     PIF   $0000 DNBWR    8 Jun 89 15:56  21 May 89 21:07
 Sys.Resources    P16+  $0000 DNBWR   11 Jun 89  9:28  21 May 89 21:11
```

```
 CDEV.INIT        TIF  $0000 DNBWR    8 Jun 89 16:02  21 May 89 21:34
 ATInit           ATK  $0000 DNBWR   16 Jun 89 18:10   6 Jun 89  7:51
 ATResponder      PIF  $0000 DNBWR    6 Jun 89  7:54  21 May 89 20:29
 ROM.CDA          TIF  $0100 DNBWR    1 Feb 88 23:47   9 Nov 88 14:47
 APPLESHARE.PR=   ATK  $0000 DNBWR   12 Jun 89 13:16   9 Jun 89 16:37
Desk.Accs         DIR  $0000 DNBWR   13 Jun 89 12:04   9 Nov 88 14:48
 CTLPANEL.NDA     NDA+ $0000 DNBWR   12 Jun 89 17:40   7 Jun 89 16:40
 SDUMP.EA         CDA  $0100 DNBWR    8 Apr 87 10:37   9 Nov 88 14:48
 ASCII            CDA  $0100 DNBWR   27 Feb 88 16:47   9 Nov 88 14:48
 PRINTSCRN        CDA  $0100 DNBWR   13 Feb 89 12:50  13 Feb 89 12:50
 SCRAMBLER.DA     CDA  $0000 DNBWR   22 Dec 88 18:56  22 Dec 88 18:56
 Exerciser.DA     CDA  $0000 DNBWR    2 Mar 89 17:08  27 Feb 89 14:29
 LOADDUMP.DA      CDA  $0000 DNBWR   27 Feb 89 13:40  20 Feb 89 13:32
 NIFTYLIST.CDA    CDA  $0100 DNBWR    4 Oct 88 23:13   9 Nov 88 14:49
 NLIST.DATA       TXT  $0000 DNBWR   30 Sep 88  1:07   9 Nov 88 14:48
 CALENDAR         NDA  $0100 DNBWR   20 Nov 87 10:01  10 Nov 88  9:06
 HEXCALC          NDA  $0000 DNBWR   26 Jul 88 21:21   9 Nov 88 14:48
 MELTDOWN         NDA  $0100 DNBWR   26 Dec 88 11:22  26 Dec 88 11:22
 MEMORYBAR        NDA  $0000 DNBWR   14 Feb 89 20:27  14 Feb 89  7:28
 MOUSETRACK       NDA  $0100 DNBWR   16 Dec 88  2:16  16 Dec 88  2:16
 SHOWTEXT         NDA  $0100 DNBWR   20 Dec 88 17:26  20 Dec 88 17:26
 SCRAPBOOK        NDA  $0100 DNBWR   20 Nov 87 10:01   9 Nov 88 14:48
CDevs             DIR  $0000 DNBWR   13 Jun 89 11:58   9 Dec 88 12:17
 GENERAL          $C7+ $0000 DNBWR    8 Jun 89 16:04  21 May 89 21:36
 MONITOR          $C7+ $0000 DNBWR    8 Jun 89 16:03  21 May 89 21:35
 RAM              $C7+ $0000 DNBWR    8 Jun 89 16:03  21 May 89 21:35
 KEYBOARD         $C7+ $0000 DNBWR    8 Jun 89 16:04  21 May 89 21:36
 SOUND            $C7+ $0000 DNBWR    8 Jun 89 16:05  21 May 89 21:37
 MOUSE            $C7+ $0000 DNBWR    8 Jun 89 16:05  21 May 89 21:37
 PRINTER          $C7+ $0000 DNBWR    8 Jun 89 16:05  21 May 89 21:38
 TIME             $C7+ $0000 DNBWR    8 Jun 89 16:06  21 May 89 21:38
 MODEM            $C7+ $0000 DNBWR    8 Jun 89 16:06  21 May 89 21:39
 SLOTS            $C7+ $0000 DNBWR    8 Jun 89 16:07  21 May 89 21:39
 AppleShare       $C7+ $0000 DNBWR    6 Jun 89  9:19  21 May 89 21:48
 ATIWriter        $C7+ $0000 DNBWR    6 Jun 89  9:12  21 May 89 21:41
 ATLQIWriter      $C7+ $0000 DNBWR    6 Jun 89  9:13  21 May 89 21:42
 ATLWriter        $C7+ $0000 DNBWR    6 Jun 89  9:15  21 May 89 21:44
 CDEV.DATA        $00+ $0000 DNB R   13 Jun 89 11:58  13 Jun 89 11:58
 CDev             $C7+ $0000 DNBWR    9 Jun 89 10:26   9 Jun 89 10:24
 ALPHABET         $C7+ $0000 DNBWR    8 Jun 89 16:07  21 May 89 21:40
 DIRECTCONNECT    $C7+ $0000 DNBWR    6 Jun 89  9:15  21 May 89 21:45
START             S16+ $0000 DNBWR   16 Jun 89 10:58   7 Jun 89 12:41
Finder            S16  $0000 DNBWR   13 Jun 89 11:23  22 May 89 12:18
```

The following example shows the use of the -d option to produce full pathnames, as well as a recursive search of the startup volume for the string .asm:

```
Files-r -f .asm *: -d

:HD:APW:Libraries:AInclude:STANDARD.ASM
:HD:APW:Source.ASM
:HD:APW:Source.ASM:CLRFF:CLRFF.ASM
:HD:APW:Source.ASM:CLRFF:PURGE.ASM
:HD:APW:Source.ASM:CLRFF:MEMORY.ASM
:HD:APW:Source.ASM:CLRFF:SUPERMEM.ASM
:HD:APW:Source.ASM:PAUSE:PAUSE.ASM
:HD:APW:Source.ASM:SIEVE:SIEVE.ASM
:HD:APW:Source.ASM:SHUTDOWN.ASM
:HD:Assemblers:EDASM:EDASM.ASM
:HD:Disk.Utils:SCSI.UTILITIES:HD.SCSI.ASM.0
:HD:Graphics:SHOWSCREEN:SHOWSCREEN.ASM
:HD:REZ.test:CDEV.SAMPLES:ASSEMBLY:CDEV.ASM
:HD:REZ.test:CDEV.SAMPLES:ASSEMBLY:MAKE.CDEV.ASM
```

The following example shows how to use the -t flag to specify a filetype filter, and the -x flag to specifiy which directory fields to display:

```
Files -t s16 *:graphics -x tdreqs -r

Filename          Type  DF Blk  RF Blk   Data EOF   Rsrc EOF  File Sys ID

graphics:
 DELUXEPAINT      S16      550       0     279397          0  ProDOS/SOS
 DRAW.PLUS        S16      387       0     196608          0  ProDOS/SOS
 PWGOLD           S16      574       0     291840          0  ProDOS/SOS
 SHOWSCREEN       S16        9       0       3926          0  ProDOS/SOS
 SHRCONVERT       S16      141       0      71680          0  ProDOS/SOS
 SHRCONV.FINDER   S16        1       0        512          0  ProDOS/SOS
 TOPDRAW          S16      264       0     133474          0  ProDOS/SOS
```

LinkIIGS

`LinkIIGS [`*option...*`]` *objectfile...*

This external command calls the advanced linker for linking object files. Use this command instead of the `Link` command if you want to do any of the following:

- select specific segments from an object file
- assign object-file segments to specific load-file segments
- assign a segment kind, such as static or dynamic, to a load-file segment
- specify the exact order in which to search libraries
- control the diagnostic output of the linker

The advanced linker, including a complete command description for `LinkIIGS`, is described in Chapter 2, "Advanced Linker."

MakeBin

`MakeBin [`*option*`] `*loadfile*

The `MakeBin` tool converts an Apple IIGS load file to a ProDOS 8 binary load file (GS/OS file type $06). The `MakeBin` utility does no checking to make sure that your program will run under ProDOS 8.

loadfile The full or partial pathname of an Apple IIGS load file.

The load file must consist of only static load segments. They can be all relocatable or all absolute, but there cannot be both relocatable and absolute load segments in the load file.

Multiple relocatable load segments are relocated one after the other in one ProDOS 8 binary file (known as a BIN file). A load file that consists of relocatable load segments is converted to one BIN file; each of the relocatable load segments becomes an absolute binary image within the BIN file. The starting address of each load segment is the absolute address of the previous load segment plus the previous load segment's length. The first load segment's starting address is whatever you specified with the `-org` option. If you do not specifiy an `-org` option for a relocatable load file, `MakeBin` uses a starting address of $2000.

A load file with multiple absolute load segments is converted into multiple BIN files, each with a different starting address. In this case, no `-org` parameter is allowed, because each load segment already contains its own starting address in its segment header. When you convert a load file with multiple absolute load segments, the `-o` *binfile* parameter is required and `MakeBin` uses the filename portion of *binfile* as the first part of the name for each output BIN file. The first output BIN file is given the exact name you specify with the *binfile* parameter, and each subsequent BIN file is given the *binfile* name with a numeric suffix (.2, .3, and so forth).

The maximum value accepted by ProDOS 8 for the starting address of a BIN file is $FFFF. If `MakeBin` finds a starting address in an absolute load segment that is more than 2 bytes long, or if you specify an `-org` option with a value greater than $FFFF, then `MakeBin` uses the lowest 2 bytes for the auxiliary file type of the binary file.

The following types of load files cannot be converted by `MakeBin` and result in errors if conversion is attempted:

- load files with both relocatable and absolute load segments
- load files containing dynamic segments
- load files containing references to run-time library segments

△ **Important** To run `MakeBin` enough memory must be available to store the largest relocation dictionary record in the input load file plus all the load segments at the same time. △

option... You can specify as many of the following options as you wish.

`-at` *$nnnn* This option sets the auxiliary file type of the output file to *$nnnn*. The default auxiliary file type is the value specified by the `-org` option.

`-o` *binfile* This option specifies the full or partial pathname of the binary file you want to create. Multiple relocatable load segments are relocated one after the other in one binary file; multiple absolute load segments are converted to multiple binary files. The first binary file has the filename specified by *binfile*; the second is named *binfile*.2; the third is named *binfile*.3; and so forth.

▲ **Warning** If you do not specify *binfile*, the input load file is overwritten with the BIN file. ▲

`-org` *val* This option gives the binary file a fixed start location at *val* and relocates all code for execution starting at the address *val*. You can use a decimal number for *val* or you can specify a hexadecimal number by preceding *val* with a dollar sign (`$`).

The maximum value for the start location accepted by ProDOS 8 is $FFFF. `MakeBin`, however, accepts any value up to $FFFF FFFF for *val*. `MakeBin` uses the full 4-byte value you specify for relocation when it processes the file, but only the lowest 2 bytes are used for the auxiliary file type of the binary file.

If you omit the `-org` option and the load file contains relocatable load segments, the load file is relocated to start at $2000. The `-org` parameter is not permitted if you are converting a load file with absolute load segments.

`-p` This option writes progress information to standard error output. It causes the following information to be written to the screen

- the version number of `MakeBin`
- the date
- the name of the file `MakeBin` is reading
- the name of the segments `MakeBin` is reading
- the size of the LCONST OMF record that `MakeBin` is processing

`-s` Writes a summary report to standard output. It causes the following information to be written to the screen

- the total number of segments read
- the total number of each OMF record type that was processed
- the number of BIN files created

`-t $`*nn* This option sets the file type of the output file to $*nn*. The default file type is $06.

MakeDirect

`MakeDirect [-p] stacksize`

`MakeDirect` creates a linkable object file (`Direct.obj`) that contains a direct-page/stack object segment of size *stacksize*. When you link this file with an application, the advanced linker will create a direct-page/stack load segment of size *stacksize*, which the System Loader uses to assign a direct-page/stack for the application at execution time.

If you also link the C or Pascal libraries, the default direct-page/stacks in these libraries will not be used because the name of the object segment in `Direct.obj` is `STACKMIN`.

If you don't specify a direct-page/stack segment for your application with `MakeDirect`, GS/OS provides a default 4 KB stack (or, if you link the C or Pascal libraries, they provide a 4 KB stack). Use `MakeDirect` only if you want a direct-page/stack larger or smaller than 4 KB.

◆ *Note:* ProDOS 16 provides a 1 KB stack by default.

stacksize — The size of the direct-page/stack load segment to create. You can specify it as a decimal or hexadecimal number. Be sure to precede a hexadecimal number with a dollar sign (`$`). The minimum value allowed is 256 (hex, $100) and the maximum is 65535 (hex, $FFFF).

`-p` — This option writes progress information to standard error output.

The following command creates the object file `Direct.obj` containing one direct-page/stack load segment of size $2000:

```
MakeDirect $2000
```

You can then link this file with an application, as follows:

```
LinkIIGS Your.obj Direct.obj -lib Your.Lib
```

You could also link `Direct.obj` with an application that is linked to C libraries, as follows:

```
LinkIIGS 13:Start.obj Your.obj Direct.obj -lib 13:CLib
```

MakeLib

`MakeLib` [*option...*] `-l` *libraryfile*

The `MakeLib` tool creates, modifies, or searches a library file. An APW library file contains all the segments from one or more object files, plus a library dictionary segment that the linker uses to find the object segments it needs. Library files are of file type $B2.

`MakeLib` can also be used to convert an OMF version 1.0 library file to version 2.0.

`-l` *libraryfile* — The full or partial pathname (including the filename) of the library file to be created, modified, or searched.

If you specify the `-r` option, `MakeLib` adds object files to an existing library file, replaces object files in an existing library file, or creates a new library file. (See the first example following the option descriptions.) If you specify `-x`, `MakeLib` locates the specified component object files in the library file, creates object files (file type $B1), and stores them in the current directory. If you specify `-d`, `MakeLib` deletes the specified object files from the library file. You can specify only one of these three options each time you run MakeLib. You must specify at least one of the options `-c`, `-d`, `-f`, `-r`, or `-x`.

`MakeLib` reads the files you specify with the `-l` parameter and the object files you specify with the `-r` option. It does not read standard input.

If you use the `-c`, `-d`, or `-r` option and do not specify the `-o` option, `MakeLib` writes output to the library file you specify with the *libraryfile* parameter. If you use the `-o` option, output is written to the file you specify with that option. If you use the `-x` option, `MakeLib` creates an object file in the current directory for each of the filenames you specify with that option. If you specify the `-f` option, `MakeLib` writes a list of the object files contained in the library file to standard output.

option... — You can specify only one of the following three options, and you can specify the option only once:

`-d` *objectfile ...* — This option deletes the specified object files from the library file. When you use this option you must specify only the filenames of the object files to be deleted, not a full or partial pathname. The filenames of component object files are case insensitive. You can use the `-f` option to obtain a list of the component object files. Use the `-o` option to prevent *libraryfile* from being overwritten by the output file.

`-r` *objectfile ...* This option adds the specified object files to the library file. You must specify the full or partial pathname, including the filenames, of the object files to be added. The filenames of component object files are case insensitive. If any of the object filenames you specify are already in the library file, the new file replaces the one currently in the library file. Use the `-o` option to prevent *libraryfile* from being overwritten by the output file.

`-x` *objectfile ...* This option extracts the specified object files from the library file and stores them in the current directory. When you use this option you must specify only the filenames of the object files to be extracted, not full or partial pathnames. The filenames of component object files are case insensitive. You can use the `-f` option to obtain a list of the component object files. The `-x` option does not delete the specified object files from the library file. To extract and delete an object file, you must run `MakeLib` twice, first with the `-x` option and then with the `-d` option.

If you use any of the options `-c`, `-d`, or `-r`, you can also specify the following option:

`-o` *outputfile* Write any output file to the file *outputfile*. The parameter *outputfile* can be a full or partial pathname, including a filename. You can use this option to prevent the original library file from being overwritten when you modify a library file. You cannot use this option when you are creating a new library file (that is, when *libraryfile* does not already exist) or when you are not modifying the library file (such as when you use the `-x` option or when you use the `-f` option alone to list the component object files in a library file).

In addition, you can specify any of the following options:

`-c` This option converts *libraryfile* from OMF 1.0 to OMF 2.0. Library files created by `MakeLib` are always in OMF version 2.0, so if you are using the `-d` or the `-r` option to modify the file, the `-c` option is not necessary. You can use the `-c` option to convert an existing library file to OMF 2.0 when you are not otherwise modifying the library file. Use the `-o` option to prevent *libraryfile* from being overwritten by the output file.

`-f` This option lists all the component object files of the library file to standard output. You can omit the *objectfile* parameter and use this option to obtain a list of component object files in the library file.

`-p` This option writes the version number of `MakeLib`, the date and time, and progress information to standard error output.

For example, the following command adds `MyFile.obj` and `YourFile.obj` to `OurFile.lib`:

```
MakeLib  -l OurFile.lib -r MyFile.obj YourFile.obj
```

`MakeLib` executes this command in one of the following four ways, depending on whether the library file already exists and whether it already contains the specified object files:

- If `OurFile.lib` does not exist, `MakeLib` creates the library file `OurFile.lib` containing object files `MyFile.obj` and `YourFile.obj` and writes the library file to disk.
- If `OurFile.lib` already exists and does not already contain object files `MyFile.obj` and `YourFile.obj`, then `MakeLib` adds the object files `MyFile.obj` and `YourFile.obj` to `OurFile.lib`, overwriting the previous version of the library file on disk.
- If `OurFile`.lib exists and already contains object files `MyFile.obj` and `YourFile.obj`, `MakeLib` replaces `MyFile.obj` and `YourFile.obj` in `OurFile.Lib` with the new versions and overwrites the previous version of the library file on disk.
- If `OurFile.lib` already exists and already contains one (but not both) of the object files `MyFile.obj` and `YourFile.obj`, `MakeLib` adds the new object file to `OurFile.lib`, replaces the existing object file in `OurFile.lib` with the new version, and overwrites the previous version of the library file on disk.

Use the following command to delete `Black.obj` and `White.obj` from `Colors.lib`, writing the output library file to `NewColors.lib` and writing a progress report to standard error output and a list of component object files to standard output:

```
MakeLib  -d Black.obj White.obj -p -f -l Colors.lib
-o NewColors.lib
```

Use the following command to convert the library file `Alpha.lib` from OMF 1.0 to OMF 2.0, extract the object file `Able.obj` from the library file `Alpha.lib`, store the object file in the current directory, and write the converted library file to the file `Beta.lib`:

```
MakeLib  -c -x Able.obj -l Alpha.lib -o Beta.lib
```

ResEqual

`ResEqual [-p]` *pathname1 pathname2*

`ResEqual` compares the resources in two files and writes their differences to standard output.

`ResEqual` checks that each file contains resources of the same type and identifier as the other file, that the sizes of the resources with the same type and identifier are the same, and that their contents are the same.

pathname1 pathname2
: The full or partial pathname (including the filename) of the two files whose resources are to be compared.

`-p`
: This option writes progress information to diagnostic output.

When the contents of resources are compared and a mismatch is found, the mismatch and the subsequent 15 bytes are written to standard output. `ResEqual` then continues the comparison, starting with the byte following the last displayed. The following messages appear when reporting differences (text in italics is specific to the particular resources being compared):

`In 1 but not in 2`
The resource type and ID are displayed.

`In 2 but not in 1`
The resource type and ID are displayed.

`Resources are different sizes`
The resource type and ID are displayed.
The size of the resource in each file is displayed.

`Resources have different contents`
The resource type and ID are displayed.

`Contents of resource in file 1 at offset`
The offset to the differing bytes from the start of the resource is displayed.
The 16 bytes at the offset are displayed.

`Contents of resource in file 2 at offset`
The offset to the differing bytes from the start of the resource is displayed.
The 16 bytes at the offset are displayed.

If more than ten differences are detected in the same resource, the rest of the resource is skipped and processing continues with the next resource.

The following command compares the resources in `Sample` and `Sample.rsrc`, writing the results to standard output.

```
ResEqual Sample Sample.rsrc
```

See also the `-f resource` option of the `Equal` command, which compares the resource forks of files, including the resource map, on a byte-by-byte basis.

Rez

`Rez`

This language command sets the shell default language to `Rez`, the language used by the resource compiler. Chapter 3, "Resource Compiler and Decompiler," describes the resource compiler.

Search

`Search [option...] string [pathname...]`

The `Search` utility searches a file or files for the string you specify. By default, `Search` displays the name of the file in which it finds the search string and then searches the next file. You can use command-line options to list the number and contents of all lines in the file containing the search string

option... You can specify any of the following options.

`-a` If you specify `-a`, `Search` will search through all files that match the pathname specification. If you omit this option, `Search` searches only those files that have a filetype of `TXT` ($04) or `SRC` ($B0).

`-c` If you specify `-c`, the APW `CaseSensitive` shell variable is ignored and pattern matching is case sensitive. If you omit this parameter (and the `-i` option), pattern matching is determined by the value of the `CaseSensitive` variable, as explained below. This option is identical to the `-s` option.

`-i` If you specify `-i`, the APW `CaseSensitive` shell variable is ignored and pattern matching is case insensitive. If you omit this parameter (and the `-c` and `-s` options), pattern matching is determined by the value of the `CaseSensitive` variable, as explained below.

`-l` If you specify `-l`, `Search` lists the filename, the line number, and the contents of the line in which it found a match for the search string. You can modify this output with the `-q` option. If you omit this option, `Search` lists only the name of the file in which a match was found.

`-p` If you specify `-p`, `Search` displays progress information. Progress information consists of brief messages that tell you what the utility is currently working on—for example, which file is currently being searched. If you omit this option, no progress information is displayed.

`-q` If you specify `-q`, `Search` displays only the contents of the matching lines. It does not list the filename or the line number. This option automatically turns on the listing of lines (`-l`).

`-r` If you specify `-r`, `Search` displays those lines that do *not* match the search string. This option automatically turns on the listing of lines (`-l`).

`-s` If you specify `-s`, the APW `CaseSensitive` shell variable is ignored and pattern matching is case sensitive. If you omit this parameter (and the `-i` option), pattern matching is determined by the value of the `CaseSensitive` variable, as explained below. This option is identical to the `-c` option.

string The string for which you wish to search. To specify a string that includes spaces, enclose the string in double quotation marks (`" "`).

pathname... The full or partial pathname, including filename, of the file or files you want to search for *string*. You can include wildcard characters in this filename. If you specify more than one pathname, `Search` will search the files in the specified order. If you omit this parameter, `Search` reads characters from standard input.

`Search` uses the value of the APW `CaseSensitive` shell variable to determine whether pattern matching will be case sensitive or not. If you define the variable as any nonzero value, `Search` will perform case-sensitive pattern matching; that is, the characters in two strings being compared must match exactly, including uppercase and lowercase. If you define the value as 0 or do not define it at all, `Search` will perform case-insensitive pattern matching. In this case, for example, an uppercase *S* and a lowercase *s* will match. You can override the setting of this variable with the `-c`, `-i`, and `-s` command-line options.

◆ *Note:* If you define `CaseSensitive`, be certain to export it so that `Search` can use it.

Here are some examples that illustrate how to use the various options of the `Search` command.

The following command searches all files in the directory `13:cinclude:` that have the extension `.h` (that is, C header files) and contain `#include` statements. It displays the names of all such files.

```
search #include 13:cinclude:=.h
```

If you want to perform the same search as in the previous example but to display the lines containing the search strings as well as the filenames, use the command

```
search -l #include 13:cinclude:=.h
```

To perform a case-insensitive search of the APW `UserStartUp` file and list all lines that do not contain the string `Export`, you could use the following command. (Note that `-q` suppresses generation of the filename and of the line numbers of the listed lines.)

```
search -r -q -i Export 15:UserStartup
```

The next example shows how you can pipeline the output of the `Catalog` command to the `Search` command to get a list of all files that were created or modified during June 1989.

```
catalog*: | search -l -q -i "jun 89"
```

Chapter 2 **Advanced Linker**

This chapter describes the advanced APW Linker (accessed through the command `LinkIIGS`), including its command syntax and command-line options. The standard linker is described in Chapter 7 of the *APW Reference*.

Operation of the linker

This section describes

- the purpose of the linker
- the formats and types of input files (object files) to the linker
- the formats and types of output files (load files) that the linker produces
- the diagnostic output from the linker

What the linker does

A linker is a program that locates individual program segments, resolves references between segments, and combines them into a complete, executable program. The APW linkers are independent of source-code language. They can extract specific code segments from multiple library and object files and can create segmented load files.

The APW linkers work with any assembler or compiler that generates files conforming to the Apple IIGS object module format (OMF). The linkers can join separate files produced by Apple IIGS–compatible assemblers and compilers and convert them into the form needed by the System Loader for loading into the computer. Together, these three components (assembler or compiler, linker, and loader) provide a very powerful and flexible programming facility.

The principal tasks of a linker are to bring together the segments needed for a program and to resolve global references. Because most Apple IIGS code is relocatable, the APW Linker must work with the System Loader to resolve and relocate global references. The linker provides the relocation information necessary for the loader to relocate all references after loading. Much of the work of the linker therefore consists of constructing tables of information for the loader to interpret, so that it can load and relocate the linker's output correctly.

For most purposes, the standard linker is completely adequate. It can be called by the APW `Link` command. It is integrated with the APW assemblers and compilers, so it can also be called automatically by the `AsmL`, `AsmLG`, `CmpL`, `CmpLG`, and `Run` commands after a successful assembly or compile.

The advanced linker is provided for programmers who require maximum flexibility from the system; it is called by the `LinkIIGS` command. You can perform these options with the advanced linker:

- select specific segments from an object file
- assign object-file segments to specific load-file segments
- assign load-file segments as static or dynamic
- specify the exact order in which to search libraries
- control the diagnostic output of the linker

Object files: Input to the linker

Object files are the output from an assembler or compiler and the input to a linker. Although both object and load files conform to the Apple IIGS object module format, only object files can be processed by the linker. Only object-file information specifically related to the operation of the linker is discussed in this chapter; see the *GS/OS Reference* manual for more detailed information about the Apple IIGS object module format.

Object files (GS/OS file type $B1) contain data or program code that has been translated (assembled or compiled) into machine language but may contain unresolved references to external subroutines or data. The linker processes object files, resolves external references, and produces load files. Load files contain all the information necessary to relocate external references and are ready to be loaded into the computer by the System Loader.

◆ *Note:* By default, `LinkIIGS` produces files of type $B3, GS/OS application load files. Use the `-t` option on the `LinkIIGS` command line to specify a different file type.

Each object file consists of segments. Each segment is a separate entity that contains all the information necessary to link it with other segments. A segment consists of a header followed by a body; the header contains name, size, type, and other information about the segment, while the body consists of sequential records, each one of which consists of either program code or information for the linker or loader. Segments are discussed in "Program Segmentation" in Chapter 1 of the *APW Reference*. See also the "Object Module Format" appendix in the *GS/OS Device Driver Reference*.

Library files

Library files (GS/OS file type $B2) contain object segments useful to many programs. The linker can search library files to resolve references unresolved within the program source code. Library files are normally kept in the APW library prefix (prefix 13). The advanced linker searches only those library files that you specify with the `-lib` option.

Library files differ from object files in that each library file includes a segment called the *library dictionary segment* (segment `KIND` = $08). The library dictionary segment contains the names and locations of all segments in the library file. The linker can look through the library dictionary segment for the names of segments it needs, so the library dictionary segment allows the linker to find segments much more quickly than if it had to scan through the entire file. Library files are created from object files by the `MakeLib` utility program (described in Chapter 1, "The APW Tools"). Each library file can be created from any number of object files.

△ **Important** The advanced linker searches through library files as many times as necessary to resolve references (see the discussion of the `-lib` option in "`LinkIIGS` Command Description" later in this chapter). The order of subroutines within a single library file can affect the time necessary to complete a link but is otherwise not important. △

In addition to ordinary library files, the advanced linker supports **run-time library files** (file type $B4 or RTL). Run-time library files are load files that can be searched by the advanced linker as libraries. When the advanced linker resolves a reference to a run-time library file, however, it does *not* extract the segment from the run-time library file and place it in the file that referenced it as it does for ordinary library files. Instead the advanced linker puts an entry in a special segment called a *jump table segment,* which the loader uses during program execution to find the run-time library file. The loader loads a run-time library segment only when it is needed (similar to the way it treats dynamic segments).

You can use the `-t RTL` option to the `LinkIIGS` command to create a run-time library file. For a complete description of the `LinkIIGS` command options, see "`LinkIIGS` Command Description" later in this chapter. Pathname segments and run-time library files are described in the *GS/OS Device Driver Reference.*

Partial assemblies and filename conventions

When you assemble or compile a program, you can use a KEEP directive (or the equivalent for the language you are using) in the source code or the keep parameter in the command line to specify a filename for the object files that are created. If you are assembling or compiling an entire program consisting of more than one segment, the first segment to be executed when the program is run is placed in one file and the remaining segments are placed in a second file. If the filename you specify is MYPROG, the first file is named MYPROG.ROOT and the second one is named MYPROG.A.

△ **Important** The root filename cannot be longer than 10 characters for files to which the .ROOT extension will be appended because the ProDOS file system limits the entire filename to 15 characters. Using more than 10 characters in such a filename will result in a fatal assembler or compiler error (`Unable to open output file`). △

There are two circumstances under which a file with a higher alphabetic suffix (.B, .C, and so on) is created; see Chapter 7 of the *APW Reference*. However, this APW partial assembly file structure is not supported by LinkIIGS. If you wish to link such object files, first perform a complete assembly under APW or run the APW CrunchIIGS utility script, to combine everrything into one .obj file, and then call LinkIIGS. Note, however, that you can specify individual object files and segments to process with the advanced linker.

Load files: Output from the linker

Load files (types $B3 through $BE) are the result of the processing of object files by the linker. They contain segments that are ready to be loaded into memory by the System Loader. Load files conform to a subset of the Apple IIGS object module format and do not contain any unresolved symbolic references.

Both object files and load files are segmented, but a load segment may contain more than one object segment. In assembly language, both the object-segment name and the name of the load segment to which that object segment is to be assigned can be specified with a START, DATA, PRIVATE, or PRIVDATA directive. APW C provides the `segment` function to allow you to assign subroutines to specific load segments. As a default, some APW compilers assign one load-segment name (a string of spaces) to all code segments, and another (`~global`) to all global variables.

With the advanced linker there are three ways you can assign object segments to load segments: by using the default assignments used by the compiler, by making load-segment assignments in the source code, or by using `-lseg` options on the `LinkIIGS` command line.

When you list object files on the command line without using `-lseg` options, the linker assigns object-file segments to load-file segments based on the load-segment names assigned in the source code, or, if there is no load-segment name assignment in the source code, to the load segment assigned by the compiler. (There is always a load-segment name; however, the name may be null or blank.) All object-file segments with the same load-segment name are collected by the linker into a single load segment with that name. When you list only an object-file name on the command line, all object segments in that file are linked. If you wish, you can specify that only certain object segments in the object file should be linked.

You can also use `-lseg` options to assign object segments to specific load segments. You can use `-lseg` options in a number of different ways, as follows:

- You can link all the object segments in a specified object file to a load segment.
- You can link specific object segments in a specified object file to load segments.
- You can link all the object segments in a specified object file to a load segment and then reassign particular segments in that object file to different load segments.

The advanced linker can produce a single load file from a single object file or from several object files. For details see "`LinkIIGS` Command Description" later in this chapter.

`LinkIIGS` produces load files of type $B3 (GS/OS application) by default. You can use the `-t` option on the command line to specify a different type if you wish.

Table 2-1 lists the entries in a load-segment header and shows how the linkers determine the value for each.

■ **Table 2-1** How the linker assigns values to load-segment header fields

Field	How determined
BYTECNT	Actual size of segment in the load file
RESSPC	0
LENGTH	Sum of lengths of component object segments
LABLEN	0
NUMLEN	4
VERSION	OMF version number (currently 2)
BANKSIZE	Value from first object segment linked into this load segment
KIND	Value from `-lseg` option, if specified; otherwise value from first object segment linked into this load segment
ORG	Value from `-org` option if specified immediately after the `-lseg` option; otherwise value from first object segment linked into this load segment
ALIGN	Value from first object segment linked into this load segment
NUMSEX	0
REVISION	Number following the decimal point in current OMF version number (currently 0)
SEGNUM	Number of segment in load file, starting with 1
ENTRY	Offset from start of load segment to beginning of first object code segment linked into load segment; 0 if only data segments have been linked into this load segment
DISPNAME	Offset within the segment of the LOADNAME field
DISPDATA	Offset within the segment of the body of the load segment
LOADNAME	0000000000
SEGNAME	Value from `-lseg` option if specified; otherwise load-segment name from object-segment headers

For a complete description of load files see the *GS/OS Reference.*

Diagnostic output

In addition to the load file itself, the linker produces diagnostic output to show what it has done and to aid debugging. There are three types of diagnostic information: error messages, a summary and progress report, and a symbol table. Each of the types of information output by the linkers is described in the following sections.

Error messages

Errors can be caused by mistakes in the command line or by problems encountered while trying to link an object file. The Appendix gives a full list of error messages and their meanings. Error messages cannot be suppressed.

- *Note:* `LinkIIGS` produces warning messages as well as error messages. Warning messages can be suppressed with the `-w` option.

Summary and progress report

The linker enables you to list summary information, such as segment name, length, and type. Use the `-l` option of the `LinkIIGS` command to list summary information. See the description of the `-l` option in "`LinkIIGS` Command Description" later in this chapter, for details on the type of data that is displayed.

`LinkIIGS` enables you to list the following progress information with the `-p` option: linker version number, date and time, the object files being processed, and all the steps of the link.

Symbol table

The linker enables you to print a symbol table. Use the `-s` option of the `LinkIIGS` command to print an alphabetized global symbol table. See the description of the `-s` option in "`LinkIIGS` Command Description" later in this chapter for details on the type of symbol data that is displayed for the respective linker commands.

- *Symbol types:* The Apple IIGS object module format defines three types of symbols: global, private, and local. Global symbols can be referenced in any segment. For APW assembly-language programs, for example, global symbols include object-segment names defined by `START` and `DATA` directives and any symbols defined in an `ENTRY` or `GEQU` directive. Private symbols are available to any segment in the same object file, but not to segments in other object files that are part of the same program. For APW assembly-language programs, private symbols include object-segment names defined by `PRIVATE` and `PRIVDATA` directives. Local symbols are labels that are defined only within individual code or data segments.

Local symbols are normally accessible only within the segment in which they appear. However, a segment may gain access to local symbols in another *data* segment by issuing a USING assembler directive. The USING directive cannot refer to a code segment.

The assembler or compiler resolves local references, so the linker never sees them. Therefore, local symbols never appear in the symbol table, with the exception of local labels in a data segment named in a USING directive.

- *Note:* LinkIIGS ignores USING directives and automatically converts any local symbols to global symbols.

Be sure that no two global symbols (or local symbols in data segments) with the same name appear anywhere in the program. Two private symbols with the same name cannot appear in the same object file but can appear in separate object files that are part of the same program.

LinkIIGS command description

LinkIIGS [*option...*] *objectfile...*

LinkIIGS combines object files and searches library files to create load files that can be loaded by the Apple IIGS System Loader.

objectfile... The full or partial pathnames of the object files to be linked; it can include wildcard characters (=). Wildcard characters are discussed in Chapter 2 of the *APW Reference.*

Object files must be of type $B1; however, you can link a library file as if it were an object file by including it in the list of object files rather than in the -lib option.

To link only specific object segments from an object file instead of linking the entire file, use the following syntax for *objectfile:*

objectfile(*segname1*, *segname2*, . . .)

To link only those object segments that have a specific load-segment name, use the following syntax for *objectfile:*

objectfile(@*loadsegname1*, @*loadsegname2*, . . .)

You can use an equal sign (=) as a wildcard character to replace any string of characters in object-segment names and load-segment names. Object-segment names and load-segment names are discussed in Chapter 1 of the *APW Reference* and in *Programmer's Introduction to the Apple IIGS.*

The linker assigns the object segments that you specify on the command line (or that are in the object files you specify) to load segments based on the load-segment names assigned in the source code. If an object segment does not have a source-code load-segment name, the compiler assigns it a default load-segment name or, if no other load-segment name is assigned to an object segment, the linker assigns it to a load segment with a blank load-segment name.

You can reassign the load segment into which specific object segments should be linked by using the -lseg option. Any object files or object segments for which you want the linker to use the source-code load-segment assignments (or the default load segments assigned by the compiler) must be listed *before* the first -lseg option on the command line.

Notice that any object segment not in an object file listed with the *objectfile* parameter and not specified with the `-lseg` option will not be linked. Because some compilers create object segments in addition to the ones you specify in the source code, it is recommended that you include all the object files in the *objectfile* parameter—even if you also use `-lseg` options to specify load segments for every object segment in your source code.

You can link the same object segment more than once into a given load segment or into more than one load segment. (However, the linker will issue a warning if you do either of these things.) If the linker finds two or more definitions for the same symbol, it uses the following priority scheme:

1. If there is a private definition in the same object file as the reference, the linker uses that definition.
2. If there is no private definition, but there are public definitions in the load segment currently being processed, the linker uses the last definition it encounters in the current load segment.
3. If there is no public definition in the load segment currently being processed, but there are public definitions for the symbol in other load segments, the linker uses the last definition it encounters.

`LinkIIGS` requires enough memory to hold simultaneously all object files, library file dictionaries, the symbol table, and the load segment that is currently being created. If there is insufficient memory available, the error message `Ran out of memory in function` ***nnn*** is written to standard error output.

`LinkIIGS` generates Apple IIGS version 2.0 OMF load files. Version 2.0 OMF is defined in the *GS/OS Device Driver Reference*. Object segments that are processed by `LinkIIGS` must have the following values in their segment headers:

- NUMSEX = 0
- NUMLEN = 4

`LinkIIGS` ignores certain OMF records, and performs special handling of others:

- ignores USING records and prints no warning messages
- ignores MEM records and prints no warning messages
- converts all LOCAL records in data segments to GLOBAL records
- converts all EQU records in data segments to GEQU records

- *APW partial assemblies:* The APW partial assembly file structure (that is, multiple object files containing different versions of the same object segments) is not supported by `LinkIIGS`. If you wish to link such object files, first perform a complete assembly under APW or run the APW `CrunchIIGS` utility script, and then use `LinkIIGS` to link both the `.obj` files as an independent object file; or use the standard linker. See "Partial Assemblies and Filename Conventions" in this chapter, for more information on partial assemblies.

`LinkIIGS` reads the object files you specify, and searches the library files you specify with the `-lib` option. Object files must be file type $B1 and library files must be file type $B2 or $B4. If standard input is redirected to be from a file, `LinkIIGS` reads the file and interprets the data in the file as command-line parameters. `LinkIIGS` does expand APW variables and wildcards and recognizes the comment character (`*`) and the line continuation character (`\`) within files used for standard input.

`LinkIIGS` creates a load file with the name you specify with the `-o` option. If you do not use the `-o` option, the load file is given the filename `Link.Out`. The output load file has the file type $B3 unless you specify a different file type with the `-t` option. If you specify the `-l` option, `LinkIIGS` writes a summary report (link map) to standard output. If you specify the `-s` option, `LinkIIGS` writes the symbol table to standard output. Progress and summary reports are described in "Progress and summary report" earlier in this chapter. The symbol table output is described in "Symbol Table" earlier in this chapter.

option... You can specify as many of the following options as you wish.

`-a` *symbol=alias* Each time the symbol named *symbol* is encountered, substitute the name *alias*. You can also specify an absolute location by using a decimal or hexadecimal number for *alias*. To specify a hexadecimal number, precede *alias* with a dollar sign (`$`). You can use this option to redefine references by relinking a program, so that a particular reference executes a different subroutine, for example. You can also use this option to change the absolute hardware locations to which specific symbols refer.

`-at $`*nnnn* This option sets the auxiliary file type of the output load file to `$`*nnnn*. To use a decimal number for the auxiliary file type, omit the dollar sign.

`-b` This option performs a "big" link. The linker uses less memory so that a larger program can be linked. It will be slower than an ordinary link. Currently, the `-b` option causes the following differences from an ordinary link:

- The output file buffer is reduced to 2 KB from 16 KB.
- Object segment headers only are kept in memory. The linker reads in the entire segment only when it needs to.
- The size of memory zones is reduced from $1000 to $200.

`-b2` This option performs a "bigger" link. The linker uses less memory than used with the `-b` option so that an even larger program can be linked. When you specify the `-b2` option, the linker does not generate SUPER records. In all other respects a link performed with the `-b2` option is identical to one performed with the `-b` option. See the "Object Module Format" appendix in the *GS/OS Device Driver Reference* for a definition of SUPER records.

`-l` This option writes summary information to standard error output. Figure 2-1 shows sample summary output.

■ **Figure 2-1** Sample of the linker's summary output

```
--------------------------------------------------------------------------------
0001 Load Segment "~ExpressLoad" relocatable LENGTH=$133 KIND=$8001 (Data)(Dynamic)
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
0002 Load Segment "main" relocatable LENGTH=$8F7D KIND=$0000
--------------------------------------------------------------------------------
0001 00000000 00000013 __start0
0002 00000013 00000141 __start1
0003 00000154 0000022A main
0004 0000037E 0000044C Get_Input
               .
               .
               .
0090 00008AE1 00000018 _syClose
0091 00008AF9 000000BD _fsClose
0092 00008BB6 0000014D _coWrite
0093 00008D03 0000027A _fsOpen

--------------------------------------------------------------------------------
0003 Load Segment "~globals" relocatable LENGTH=$70 KIND=$4401 (Data)(Private,Reload)
--------------------------------------------------------------------------------
0001 00000000 0000001A ~globals
0002 0000001A 00000030 ~globals
0003 0000004A 00000002 ~globals
0004 0000004C 00000002 ~globals
0005 0000004E 00000004 ~globals
0006 00000052 00000016 ~globals
0007 00000068 00000002 ~globals
0008 0000006A 00000006 ~globals

--------------------------------------------------------------------------------
0004 Load Segment "~arrays" relocatable LENGTH=$5724 KIND=$4401 (Data)(Private,Reload)
--------------------------------------------------------------------------------
0001 00000000 0000002E ~arrays
0002 0000002E 00005070 ~arrays
0003 0000509E 0000010B ~arrays
0004 000051A9 0000007D ~arrays
0005 00005226 00000046 ~arrays
0006 0000526C 00000190 ~arrays
0007 000053FC 00000101 ~arrays
0008 000054FD 00000048 ~arrays
0009 00005545 0000001B ~arrays
000A 00005560 00000078 ~arrays
000B 000055D8 00000022 ~arrays
000C 000055FA 0000001D ~arrays
000D 00005617 0000010D ~arrays

--------------------------------------------------------------------------------
0005 Load Segment "~Direct" relocatable LENGTH=$1000 KIND=$0012 (Direct Page/Stack)
--------------------------------------------------------------------------------
0001 00000000 00001000 STACKMIN

Output Load File "Link.Out" contains 5 Load Segments with a total length of $F844
bytes.
```

For each load segment created, the summary output displays the segment number, the name of the segment, the length of the segment in bytes (hexadecimal), and the **segment kind.** If the segment is absolute (that is, nonrelocatable), the origin address of the segment is shown. Segment kinds are described in "Object Module Format" in the *GS/OS Device Driver Reference* and can be assigned with the `LinkIIGS` option `-lseg`.

Following the load-segment header information, the summary report lists the object segments that were linked into the load segment. For each component object segment, the report shows a sequence number, the starting offset, the length of the object segment in bytes, and the object-segment name. Notice that the sequence numbers and object-segment names are for your convenience only. Object-segment names are not retained in the load segment; there is no way to extract object segments from a load segment.

`-lib` *libfile*

This option searches the library file that has the full or partial pathname specified by *libfile*. The *libfile* parameter can include APW wildcard characters. You can include only one *libfile* parameter per `-lib` option, but you can have as many `-lib` options on a command line as you wish.

Library files must be of type $B2 or $B4. You can use the `MakeLib` tool to create or modify library files. To link a library file as if it were an object file, include it in the list of object files instead of in the `-lib` option. (See the description of the *objectfile* parameter earlier in this section.)

Library files are searched after all object files have been linked, and only segments that contain symbols referenced in previously linked segments are extracted (or noted, in the case of run-time library files). Segments in a library file can reference other segments in the same library file or in other library files. Library files are searched in the order in which they are listed on the command line and are searched as many times as necessary to resolve all references.

The linker uses the same rules to determine the target load segment for library segments that it uses for object segments. Load-segment names can be assigned in the source code for the library file, or they can be assigned by the compiler. If a library-file segment has no load-segment name, the linker assigns it to a load segment with a blank name. You can reassign a library-file segment to another load segment by using the `-lseg` option in addition to the `-lib` option. (See the last example at the end of this section.) Notice that only the library-file segments specified by the `-lseg` options are retargeted; other library segments already extracted by `-lib` options are not affected.

`-lseg`[:*kind*...] *loadsegname* [*objectfile*...]

This option links object segments from the object files that have the full or partial pathnames specified by *objectfile* and places them in a load segment named *loadsegname*. You can specify as many object files as you wish for each `-lseg` option, and you can use the `-lseg` option as many times as you wish on a command line.

To link only specific object segments from an object file instead of linking the entire file, use the following syntax for *objectfile:*

objectfile`(`*segname1*`,` *segname2*`, . . .)`

To link only those object segments that have a specific load-segment name, use the following syntax for *objectfile*:

objectfile`(@`*loadsegname1*`,@`*loadsegname2*`, . . .)`

The *objectfile* parameter can include APW wildcard characters (=). You can also use an equal sign (=) as a wildcard character to replace any string of characters in object-segment names and load-segment names.

The purpose of the `-lseg` option is to allow you to assign object segments to specific load segments. All object segments listed following an `-lseg` option are linked into the load segment specified with that `-lseg` option. For example, to assign all the object segments in the object file `ObjFile` to a load segment named `LoadSeg`, use the following `-lseg` option on the command line:

```
-lseg LoadSeg ObjFile
```

If you use more than one `-lseg` option, the options are processed in the order in which they appear on the command line. You cannot use the same load-segment name in more than one `-lseg` option.

You can specify the same object file both before the first `-lseg` option and following an `-lseg` option. In this case, the object segments specified before the first `-lseg` option are first linked into the load segments specified in the source code or by the compiler. Then, any already linked object segments also specified after the `-lseg` option are reassigned to the load segment specified by that `-lseg` option.

For example, suppose you have two object files, `ObjFile1` and `ObjFile2`, each with two object segments, as follows:

Filename	Object segment	Load segment name
`ObjFile1`	`ObjSeg1`	`LoadSeg1`
	`ObjSeg2`	`LoadSeg2`
`ObjFile2`	`ObjSeg3`	`LoadSeg3`
	`ObjSeg4`	`LoadSeg4`

Now suppose you link these files using the following command:

```
LinkIIGS ObjFile1 ObjfFile2 -lseg LoadSegX
ObjFile1(ObjSeg1)
```

In this case, the object segments would be assigned to load segments as follows:

Load segment	Contains object segment
`LoadSegX`	`ObjSeg1`
`LoadSeg2`	`ObjSeg2`
`LoadSeg3`	`ObjSeg3`
`LoadSeg4`	`ObjSeg4`

When you use `-lseg` options to reassign object segments to an existing load segment, the reassigned object segments are linked into the load segment *before* the other object segments assigned to that load segment.

For example, suppose you have the same two object files as in the previous example, but you use the following command to link them:

```
LinkIIGS ObjFile1 ObjfFile2 -lseg LoadSeg1
ObjFile2(ObjSeg3)
```

In this case, the object segments would be assigned to load segments as follows:

Load segment	Contains object segments
`LoadSeg1`	`ObjSeg3`
	`ObjSeg1`
`LoadSeg2`	`ObjSeg2`
`LoadSeg4`	`ObjSeg4`

If no segment kind is assigned in the source code, the load segment has a kind of $0000; that is, it is a static code segment. If a kind is specified in the source code for the first object segment linked into the load segment, the linker gives this kind to the load segment. If you include the optional ***kind*** parameter with the `-lseg` option, the linker gives to the load segment the kind you specify. The ***kind*** parameter overrides any source-code kind specification.

Table 2-2 shows the segment attributes that you can use for the ***kind*** parameter. You can also use a hexadecimal value for the ***kind*** parameter by specifying a number with a preceding dollar sign, or you can use a decimal number by specifying a number without a preceding dollar sign. Segment kinds are discussed in the *GS/OS Device Driver Reference*.

■ **Table 2-2** Segment attributes and ***kind*** parameter values for the `-lseg` option

Attribute	Value of ***kind*** parameter
absbank*	$0800
bankrel*	$0100
code	$0000
data	$0001
direct	$0012
dynamic	$8000
initial	$0010
nospecial	$1000
posind	$2000
private	$4000
reload	$0400
skip	$0200
static	$0000

* See the description of the `-org` parameter.

You can specify more than one segment kind as long as the specification is not redundant or contradictory. For example, the specification `-lseg:initial:dynamic` is legal, but `-lseg:static:dynamic` is not allowed. Likewise, the specification `-lseg:$8010:dynamic` is not allowed, because `$8010` fully specifies the kind, so the additional specification of dynamic is redundant.

If you do not specify a segment kind, then KIND = $0000 (static code segment) is used. You can omit *objectfile* and use this option to change the type of an existing load segment.

If you specify a library file with a `-lib` option, and also include that library file in an `-lseg` option, the linker first links the object segments in that library file that satisfy unresolved references, and then reassigns any segments that are included in the `-lseg` option to the specified load segment. The last example at the end of this section illustrates this feature.

Notice that some compilers create object segments in addition to the ones you specify in the source code. It is recommended, therefore, that you include in the *objectfile* parameter all the object files in the program—even if you also use `-lseg` options to specify load segments for every object segment in your source code.

`-o` *outputfile*

This option writes the output of the linker to the file that has the full or partial pathname (including the filename) specified by *outputfile*. If you do not specify this option, the filename `Link.Out` is used.

`-org $`***nnnnn***

This option sets an absolute address for a load segment. If you place the `-org` option immediately after an `-lseg` option's load-segment name and before the object-file names, the load segment becomes an absolute load segment starting at address $***nnnnn*** unless the load segment is specified as an absolute bank segment or a bank relative segment. See the examples at the end of this section. If the load segment is specified as an absolute bank segment (segment kind = $0800), $***nnnnn*** must be a multiple of $10000 to correspond with a bank boundary. If the load segment is specified as a bank relative segment (segment kind = $0100), then $***nnnnn*** becomes the starting address of the load segment relative to the starting address of the bank into which it is loaded. For a bank relative segment, therefore, $***nnnnn*** must be less than $10000. An absolute address of 0 is not allowed.

If the load segment is already an absolute load segment, you can place the `-org` option between two object-segment names. In this case, the linker pads the load segment with enough 0's to make the next object segment start at the address $***nnnn***. The current location counter must be less than $***nnnn*** for you to use the `-org` option in this manner.

You can also place the `-org` option before an object-file name that appears before the first `-lseg` option on the command line. In this case, the load segment with the default (blank) segment name starts at address $*nnnn*. Notice that if every object segment in the object file is targeted to a specific load segment, no load segment with a blank name is created and the `-org` option is ignored.

You can use a decimal number for the origin address by specifying a number with no preceding dollar sign.

`-p` This option writes progress information to standard error output. Progress information is described in "Progress and summary report" earlier in this chapter.

`-path` *prefix* This option uses *prefix* as the pathname prefix for each run-time library file to which the linker finds a reference. The string that you specify with this option is stored in the pathname segment as the prefix to the filename of each run-time library file in the program.

The Apple IIGS System Loader uses the pathnames in the pathname segment to find the run-time library files that must be loaded during execution of the program.

You can specify partial pathnames with this option. A partial pathname begins with one of the 32 prefixes supported by the operating system; these prefixes have the form *n*:, where *n* is a number from 0 to 31. The following three prefixes have fixed definitions, as follows:

`8:` system prefix (initially the volume from which the operating system was booted)

`9:` application subdirectory (the subdirectory out of which the application is running)

`13:` system library subdirectory (initially `:`*boot_volume*`:SYSTEM:LIBS:`)

If you omit this option, the linker places the run-time library-file names into the pathname segment without a prefix and the System Loader defaults to prefix 8 when looking for the run-time library files.

Pathname segments are described in the *GS/OS Reference* and in Chapter 2 of the *APW Reference*

`-r` This option sets the Reload bit in the KIND field of any load segments that have the name `~globals` or `~arrays`. A reload segment is reloaded from the load file on disk each time the program is launched, even if the other load segments in the program are restarted from memory. When you use the `-r` option with an APW C program, the program becomes restartable on the Apple IIGS.

Restartable programs and the Reload bit are discussed in the *GS/OS Reference.*

`-s` This option sorts the symbol table and writes it to standard output after the link. Figure 2-2 shows a sample symbol table.

■ **Figure 2-2** Sample symbol table

```
-----------------------
Symbol Table
-----------------------

                        |Object----------------------|Load--------------|
Name                  Type|File      Seg        Offset  |File Seg  Offset  |Value
                          |                             |                  |
CQuit                   GD|0003:000D 0106:0062 00000000|0001 0002 0000661A|
Check_Stop              GD|0002      002A:002C 00000000|0001 0002 00002B3F|
Compact_File            GD|0002      0003:0005 00000000|0001 0002 000007CA|
Dictionary              GD|0002      002D:0002 00000020|0001 0003 0000003A|
FileInfo                GD|0002      002E:0002 00000A65|0001 0004 00000A93|
Get_Input               GD|0002      0002:0004 00000000|0001 0002 0000037E|
Get_Sizes               GD|0002      0028:002A 00000000|0001 0002 0000299F|
LConst_Table            GD|0002      002E:0002 00004998|0001 0004 000049C6|
One_File                GD|0002      002D:0002 00000022|0001 0003 0000003C|
PDosInt                 GD|0003:0011 0104:005A 00000000|0001 0002 00004FAD|
Progress_Report         GD|0002      002D:0002 00000024|0001 0003 0000003E|
STACKMIN                GD|0003:000C 0107:0001 00000000|0001 0005 00000000|
SUPER_Table             GD|0002      002E:0002 00004FD8|0001 0004 00005006|
Segment_Name            GD|0002      0029:002B 00000000|0001 0002 00002A38|
Set_Reload              GD|0002      002D:0002 00000028|0001 0003 00000042|
StandAlone              GD|0003:0005 0127:0003 00000000|0001 0003 0000004A|
Summary_Report          GD|0002      002D:0002 00000026|0001 0003 00000040|
Total                   GD|0002      002E:0002 00004972|0001 0004 000049A0|
                                       .
                                       .
                                       .
~arrays                 PD|0001      0004:0001 00000000|0001 0004 00000000|
~arrays                 PD|0002      002E:0002 00000000|0001 0004 0000002E|
~arrays                 PD|0003:0002 0133:0005 00000000|0001 0004 00005226|
~arrays                 PD|0003:0003 012F:000A 00000000|0001 0004 00005560|
~arrays                 PD|0003:0005 0124:0004 00000000|0001 0004 000051A9|
~arrays                 PD|0003:0006 011C:000D 00000000|0001 0004 00005617|
~arrays                 PD|0003:0007 0116:000C 00000000|0001 0004 000055FA|
~arrays                 PD|0003:0009 010B:0009 00000000|0001 0004 00005545|
~arrays                 PD|0003:0015 00E1:0006 00000000|0001 0004 0000526C|
~arrays                 PD|0003:001D 00CA:0003 00000000|0001 0004 0000509E|
~arrays                 PD|0003:003A 0042:0007 00000000|0001 0004 000053FC|
~arrays                 PD|0003:0056 0017:000B 00000000|0001 0004 000055D8|
~arrays                 PD|0003:005A 000E:0008 00000000|0001 0004 000054FD|
~globals                PD|0001      0003:0001 00000000|0001 0003 00000000|
~globals                PD|0002      002D:0002 00000000|0001 0003 0000001A|
~globals                PD|0003:0005 0127:0003 00000000|0001 0003 0000004A|
~globals                PD|0003:0006 0121:0008 00000000|0001 0003 0000006A|
~globals                PD|0003:0008 0111:0004 00000000|0001 0003 0000004C|
~globals                PD|0003:0015 00E2:0005 00000000|0001 0003 0000004E|
~globals                PD|0003:0018 00DE:0007 00000000|0001 0003 00000068|
~globals                PD|0003:0023 00AB:0006 00000000|0001 0003 00000052|
~string0                PD|0001      0004:0001 00000000|0001 0004 00000000|
~string0                PD|0002      002E:0002 00000000|0001 0004 0000002E|
                                       .
                                       .
                                       .
~string9                PD|0002      002E:0002 00000109|0001 0004 00000137|
~string9                PD|0003:005A 000E:0008 00000031|0001 0004 0000552E|

328 symbols are in Symbol Table and require 11808 bytes of memory
```

The fields of the symbol table are as follows:

- The symbol type, which can be one of the codes shown in Table 2-3.

- **Table 2-3** Codes for symbol types

Code	Symbol type
GD	Global definition
PD	Private definition
LD	Local definition
??	Unknown

- The file number of the input file that contains the object segment. If the input file is a library file, a second file number is also shown in this column. This second number is the file number used within the library file to identify the component object file that contains the object segment.
- Two segment numbers for the object segment. The first segment number is the number in the SEGNUM field of the segment header of the object segment. The second number indicates the relative position of the object segment within the load segment; that is, it shows the sequence in which the object segments have been linked into the load segment.
- The offset into the object segment at which the symbol is located.
- The file number of the load file. This number is 1 unless the symbol is in a run-time library file, in which case the number listed is the file number used in the relocation records that refer to this symbol.
- The segment number of the load segment into which the object segment has been linked.
- The offset into the load segment at which the symbol is located.
- The value of the symbol, if it is a constant or is located in an absolute load segment.

`-stamp`

This option puts the time and date information in any run-time library files into the corresponding pathname segment entries. If you use this option, the System Loader compares the time and date in the header of a run-time library file to the time and date in the pathname segment entry and aborts the load if they do not match. If you do not use this option, the time and date in the pathname segment entries are 0 and the System Loader does not check the time and date in the run-time library file.

`-t` *filetype*

This option sets the file type of the output load file to *filetype*. You can specify a hexadecimal number (start the number with a dollar sign), a decimal number, or a mnemonic for the file type. If the `-t` option is not specified, the file type of the load file is $B3. GS/OS load-file types and corresponding mnemonics are shown in Table 2-4.

■ **Table 2-4** GS/OS load-file types

Hexadecimal value	Decimal value	Mnemonic	Meaning
$B3	179	S16	GS/OS application
$B4	180	RTL	Run-time library
$B5	181	EXE	Shell utility
$B6	182	PIF	Permanent initialization
$B7	183	TIF	Temporary initialization
$B8	184	NDA	New desk accessory
$B9	185	CDA	Classic desk accessory
$BA	186	TOL	Tool set file

To set a GS/OS auxiliary file type in addition to the file type, use the `-at` option.

If you specify a file type of RTL or $B4, the linker converts the output file to the format of a run-time library file. Run-time library files are described in the *GS/OS Device Driver Reference.*

`-w`

This option suppresses warning messages. This option is useful, for example, if you are linking the same object segment into more than one load segment and you do not want to list warning messages for all duplicate symbol definitions. Fatal errors cannot be suppressed.

`-w2`

This option treats all warnings as fatal errors.

`-x`

This option does not put the load file in ExpressLoad format. By default, the linker creates a file that is in ExpressLoad format. With this option, the load file is in regular OMF format. You should specify this option if

- You want to use the Loader Dumper desk accessory to debug your program, because the Loader Dumper cannot monitor files that are in ExpressLoad format.
- You have code resources in your program, because code resources shouldn't have an ExpressLoad segment.

The following examples show how to use some of the LinkIIGS options.

The following command creates the Apple IIGS load file Sample. First, the linker links the C object files 13:Start.obj and Sample.obj. The object segments in these object files are placed in load segments according to the load-segment names assigned in the source code or by the C compiler. Then the linker searches the C library 13:CLib for unresolved references.

```
LinkIIGS 13:Start.obj Sample.obj -lib 13:CLib -o Sample
```

The following command creates the load file Link.Out, which contains two load segments, MAIN and Seg2. Load segment MAIN contains object segment main, and load segment Seg2 contains all the object segments in Sample.obj whose names start with *sub*.

```
LinkIIGS -lseg MAIN Sample.obj(main) \
-lseg Seg2 Sample.obj(sub=)
```

The following command is identical to the one in the previous example, except Seg2 will be loaded at location $1000.

```
LinkIIGS -lseg MAIN Sample.obj(main) \
-lseg Seg2 -org $1000 Sample.obj(sub=)
```

The following command creates the Apple IIGS load file Sample. First, the linker links the C object files 13:Start.obj and Sample.obj. The object segments in these object files are placed in load segments according to the load-segment names assigned in the source code or by the C compiler. Next, the linker searches the C library 13:CLib for unresolved references. Finally, all those object segments in the C library that have already been linked and whose load-segment name is main are retargeted to a dynamic load segment named Subs.

```
LinkIIGS 13:Start.obj Sample.obj -lib 13:CLib \
-lseg:dynamic Subs 13:CLib(@main) -o Sample
```

Chapter 3 Resource Compiler and Decompiler

This chapter explains the use of the resource compiler and resource decompiler. The decompiler is invoked by the `DeRez` command, which is described in detail in Chapter 1, "The APW Tools."

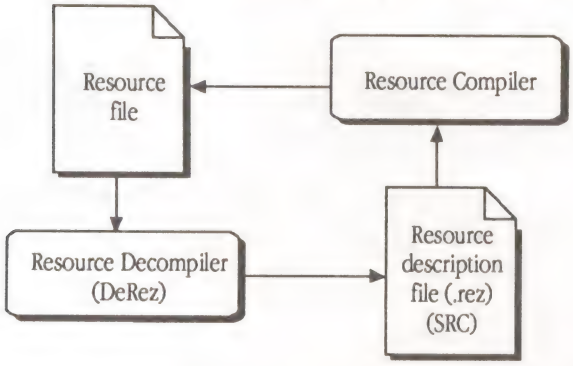
The APW Resource Compiler is an APW language compiler. As with other compilers and assemblers that run under the shell, it is invoked by the `Compile` (or `Assemble`) shell command. See Chapter 4 of the *APW Reference* for general information on the `Compile` command, and see "`Compile`" in this chapter for specific information on how this command is used with the resource compiler.

Complete background information on Apple IIGS resource files is given in the *Apple IIGS Toolbox Reference,* Volume 3.

About the resource compiler and decompiler

The APW Resource Compiler compiles a text file (or files) called a **resource description file** and produces a resource file as output. The resource decompiler, DeRez, decompiles an existing resource and produces a new resource description file that can be understood by the resource compiler. Figure 3-1 illustrates the complementary relationship between the resource compiler and DeRez.

■ **Figure 3-1** Resource compiler and decompiler



Resource description files are of language type `rez`. By convention they end in `.rez`. The `Rez` shell command enables you to set the language type to the rez language. See Chapter 4 of the *APW Reference* for more details on APW language types.

The resource compiler can combine resources or resource descriptions from a number of files into a single resource file. The resource compiler supports preprocessor directives that allow you to substitute macros, include other files, and use if-then-else constructs. (These are described under "Preprocessor Directives" later in this chapter.)

Resource decompiler

The DeRez tool creates a textual representation of a resource file based on resource type declarations identical to those used by the APW Resource Compiler. (If you don't specify any type declarations, the output of DeRez takes the form of raw data statements.) The output of DeRez is a resource description file that can be used as input to the resource compiler. This file can be edited in the APW Editor, allowing you to add comments, translate resource data to a foreign language, or specify conditional resource compilation by using the if-then-else structures of the preprocessor.

◆ *Note:* The APW Equal tool compares the resource fork as well as the data fork of files.

Type declaration files

The resource compiler and DeRez automatically look in the `13:RInclude` directory (as well as the current directory) for files that are specified by name on the command line. They also look in this directory for any files specified by a `#include` preprocessor directive in the resource description file.

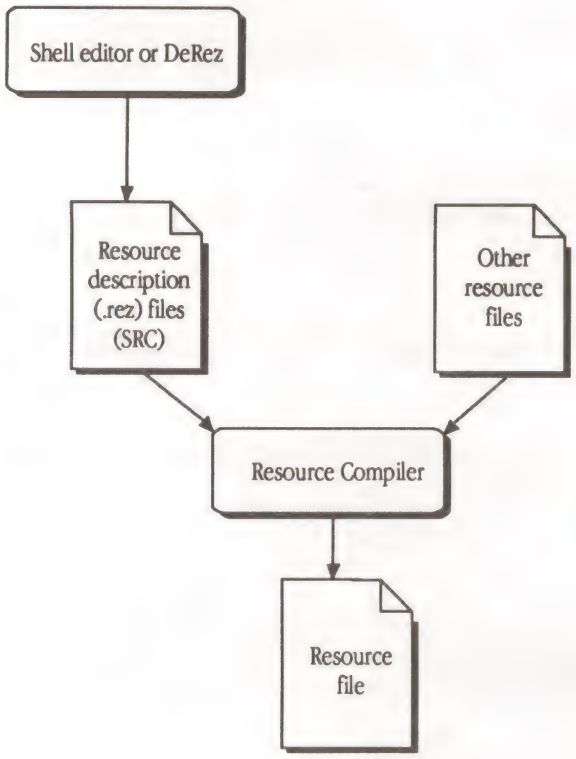
Using the resource compiler and DeRez

The resource compiler and DeRez are primarily used to create and modify resource files. Figure 3-2 illustrates the process of creating a resource file.

The resource compiler can also form an integral part of the process of building a program. For instance, when putting together a desk accessory or driver, you could use the resource compiler to combine the linker's output with other resources, creating an executable program file.

◆ *Note:* Although you can use them, resources are not required to make a desk accessory or driver (or any program other than a CDEV) executable on the Apple IIGS.

■ **Figure 3-2** Creating a resource file



Structure of a resource description file

The resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and resource decompiler have no built-in resource types. You need to define your own types or include the appropriate `.rez` files.

A resource description file may contain any number of these statements:

`data`	Specify raw data.
`include`	Include resources from another file.
`read`	Read data fork of a file and include it as a resource.
`resource`	Specify data for a resource type declared in a previous `type` statement.
`type`	Declare resource type descriptions for subsequent `resource` statements.

These statements are described in the sections that follow.

A type declaration provides the pattern for any associated resource data specifications by indicating data types, alignment, size and placement of strings, and so on. You can intersperse type declarations and data in the resource description file as long as the declaration for a given resource precedes any resource statements that refer to it. An error is returned if data (that is, a `resource` statement) is given for a type that has not been previously defined. Whether a type was declared in a resource description file or in an `#include` file, you can redeclare it by providing a new declaration later in a resource description file.

A resource description file can also include comments and preprocessor directives:

- **Comments** can be included any place white space is allowed in a resource description file, by putting them within the comment delimiters `/*` and `*/`. Note that comments do not nest. For example, this is *one* comment:

  ```
  /* Hello /* there */
  ```

  The resource compiler also supports C++ style comments:

  ```
  type 0x8001 {   // the rest of this line is ignored
  ```

- **Preprocessor directives** substitute macro definitions and `include` files, and provide if-then-else processing before other resource compiler processing takes place. The syntax of the preprocessor is very similar to that of the C language preprocessor.

Sample resource description file

An easy way to learn about the resource description format is to decompile some existing resources. For example, the following command decompiles only the `rIcon` resources in the `Sample` application, according to the type declaration in

```
13:RInclude:Types.rez.
DeRez Sample -only 0x8001  Types.rez  > DeRez.Out
```

Note that DeRez automatically finds `Types.rez` in `13:RInclude`. After executing this command, `DeRez.Out` would contain the following decompiled resource:

```
resource 0x8001  (0x1)  {
      0x8000,
      20,
      28
      $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
      $"FFFF FF00 0000 0000 0000 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
      $"0000 0000 0000 0000 0000 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0000 0000 0000 0000 0000 0000 0000"
```

Note that this statement is identical to the resource description in the file `Sample.rez`, which was originally used to build the resource. This resource data corresponds to the following type declaration, contained in `Types.rez`:

```
/*-------------------------- rIcon ----------------------*/
type rIcon {
     hex integer;       /* Icon Type bit 15  1 = color, 0 = mono */
image:
     integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
     integer;                     /* height of icon in pixels */
     integer;                     /* width of icon in pixels */
     hex string [$$Word(image)]; /* icon image */
mask:
     hex string;                  /* icon mask */
};
```

`Type` and `resource` statements are explained in detail in the reference section that follows.

Resource description statements

This section describes the syntax and use of the seven types of resource description statements available for the resource compiler: `include`, `read`, `data`, `type`, `delete`, `change`, and `resource`.

Syntax notation

The syntax notation in this chapter follows the conventions given in the preface of this book. The following additional conventions are used:

- Words that are part of the resource description language are shown in the Courier font (following the conventions used in documentation of the C language) to distinguish them from surrounding text. The resource compiler is not sensitive to the case of these words.
- Punctuation characters such as commas (`,`), semicolons (`;`), and quotation marks (`'` and `"`) are to be written as shown. If one of the syntax notation characters (for example, `[` or `]`) must be written as a literal, it is shown enclosed by "curly" single quotation marks ('...'); for example,
  `bitstring '['` *length* `']'`

  In this case, the brackets would be typed literally—they do *not* mean that the enclosed element is optional.
- Spaces between syntax elements, constants, and punctuation are optional; they are shown for readability only.

Tokens in resource description statements can be separated by spaces, tabs, returns, or comments.

Special terms

The following terms represent a minimal subset of the nonterminal symbols used to describe the syntax of commands in the resource description language:

Term	Definition
resource-ID	*long-expression*
resource-type	*word-expression*
ID-range	*ID*[: *ID*]

- *Note: Expression* is defined later in this chapter under "Expressions."

A full syntax definition can be found at the end of this chapter.

`Include`—Include resources from another file

The `include` statement lets you read resources from an existing file and include all or some of them.

Syntax

An `include` statement can take the following forms:

- `include` *file* [*resource-type* [' ('*ID*[:*ID*] ') ']] ;

  Read the resource of type *resource-type* with the specified resource ID range in *file*. If the resource ID is omitted, read all resources of the type *resource-type* in *file*. If *resource-type* is omitted, read all the resources in *file*.

- `include` *file* `not` *resource-type* ;

  Read all resources not of the type *resource-type* in *file*.

- `include` *file resource-type1* `as` *resource-type2* ;

  Read all resources of type *resource-type1* and include them as resources of *resource-type2*.

- `include` *file resource-type1* ' ('ID[:*ID*] ') '
  `as` *resource-type2* ' (' *ID* [, *attributes...*] ') ';

  Read the resource of type *resource-type1* with the specified ID range in *file*, and include it as a resource of *resource-type2* with the specified ID. You can optionally specify resource attributes. (See "Resource Attributes" later in this chapter.)

Some examples follow:

```
include "otherfile"; /* include all resources from the file */
include "otherfile" rIcon; /* read only the rIcon resources */
include "otherfile" rIcon (128); /* read only rIcon resource 128*/
```

AS resource description syntax

The following string variables can be used in the AS resource description to modify the resource information in `include` statements:

`$$Type`	type of resource from include file
`$$ID`	ID of resource from include file
`$$Attributes`	attributes of resource from include file

For example, to include all `rIcon` resources from one file and keep the same information but also set the `preload` attribute (`64` sets it):

```
INCLUDE  "file"  rIcon  (0:40)  AS
            rIcon  ($$ID, $$Attributes  |  64) ;
```

The `$$Type`, `$$ID`, and `$$Attributes` variables are also set and legal within a normal resource statement. At any other time the values of these variables are undefined.

Resource attributes

You can specify **attributes** as a numeric expression (as described in the *Apple IIGS Toolbox Reference,* Volume 3) or you can set them individually by specifying one of the keywords from any of the sets in Table 3-1.

■ **Table 3-1** Resource attribute keywords

Default	Alternative	Meaning
`unlocked`	`locked`	The Memory Manager cannot move locked resources.
`movable`	`fixed`	The Memory Manager cannot move a fixed block even when it is unlocked.
`nonconvert`	`convert`	Convert resources require a resource converter.
`handleload`	`absoluteload`	An absoluteload resource must be loaded at an absolute address.
`nonpurgeable`	`purgeable1`, `purgeable2`, `purgeable3`	Purgeable resources can be automatically purged by the Memory Manager. Purgeable3 resources are purged before purgeable2, which are purged before purgeable1.
`unprotected`	`protected`	Protected resources cannot be modified by the Resource Manager.
`nonpreload`	`preload`	Preload resources are placed in memory as soon as the Resource Manager opens the resource file.
`crossbank`	`nocrossbank`	A cross-bank resource can cross memory bank boundaries. Only data, not code, can cross bank boundaries.
`specialmemory`	`nospecialmemory`	A special-memory resource can be loaded in banks $00, $01, $E0, and $E1.
`notpagealigned`	`pagealigned`	A page-aligned resource must be loaded with a starting address that is an even multiple of 256.

You can specify more than one attribute by separating the keywords with a comma (`,`).

`Read`—Read data as a resource

The `read` statement lets you read a file's data fork as a resource.

Syntax

`read` *resource-type* '(' *ID* [, *attributes*] ')' *file* ;

Description

This statement reads the data fork from *file* and writes it as a resource with the type *resource-type* and the resource ID *ID*, with the optional resource attributes (as defined in the preceding section). Here is an example:

```
read rText (0x1234, Purgeable3) "filename";
```

`Data`—Specify raw data

Use the `data` statement to specify raw data as a sequence of bits, without any formatting.

Syntax

`data` *resource-type* '(' *ID* [, *attributes...*] ')' '{'
    data-string
'}';

Description

This statement reads the data found in *data-string* and writes it as a resource with the type *resource-type* and the ID *ID*. You can optionally specify resource attributes.

For example,

```
data rPString (0xABCD) {
      $"03414243"
};
```

- *Note:* When `DeRez` generates a resource description, it uses the `data` statement to represent any resource type that doesn't have a corresponding `type` declaration or cannot be decompiled for some other reason.

`Type`—Declare resource type

A `type` declaration provides a template that defines the structure of the resource data for a single resource type or for individual resources. If more than one `type` declaration is given for a resource type, the last one read before the data definition is the one that is used. This lets you override declarations from include files or previous resource description files.

Syntax

```
type  resource-type ['(' ID-range ')'] '{'
        type-specification...
'}' ;
```

Description

This statement causes any subsequent resource statement for the type *resource-type* to use the declaration { *type-specification...* }. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Type-specification is one of the following:

`array`	Array data specification—zero or more instances of data types
`bitstring [`*n*`]`	
`boolean`	
`byte`	
`char`	
`cstring`	
`fill`	Zero fill
`integer`	
`longint`	
`optional`	
`point`	
`pstring`	
`rect`	
`reversebytes`	
`string`	
`switch`	Control construct (case statement)
`wstring`	

These types can be used singly or together in a `type` statement. Each of these type specifiers is described in the following sections.

- *Note:* Several of these types require additional fields. The exact syntax is given in the following sections.

You can also declare a resource type that uses another resource's `type` declaration by using the following variant of the `type` statement:

`type` *resource-type1* ['(' *ID-range* ')'] `as` *resource-type2* ['(' ID ')'];

Data-type specifications

A data-type statement declares a field of the given data type. It can also associate symbolic names or constant values with the data type. The data-type specification can take three forms, as shown in this example:

```
/*---------------------- rToolStartup ---------------------*/
type rToolStartup {
    integer = 0;                /* flags must be zero */
    Integer mode320 = 0,mode640 = $80;  /* mode to start quickdraw */
    Integer = 0;
    Longint = 0;
    integer = $$Countof(TOOLRECS);      /* number of tools */
        array TOOLRECS {
            Integer;                /* ToolNumber */
            Integer;                /* version */
        };
};
```

- The last two `integer` statements declare an integer field; the actual data is supplied by a subsequent `resource` statement.
- The `integer` statement `Integer mode320 = 0,mode640 = $80;` is the same as the last two `integer` statements, except the two symbolic names `mode320` and `mode640` are associated with the values 0 and $80. These symbolic names can be used in a subsequent `resource` statement.
- The first integer statement declares an integer field whose value is always 0. In this case, no corresponding statement would appear in the `resource` data.

Numeric expressions and strings can appear in `type` statements; they are defined later in this chapter under "Expressions."

Numeric types

The numeric types (`bitstring`, `byte`, `integer`, `longint`) are fully specified like this:

[*unsigned*] [*radix*] *numeric-type* [=*expr* | *symbol-definition*...];

- The *unsigned* prefix signals DeRez that the number should be displayed without a sign—that the high-order bit can be used for data and the value of the integer cannot be negative. The *unsigned* prefix is ignored by the resource compiler but is needed by DeRez to represent correctly a decompiled number. The resource compiler uses a sign if it is specified in the data. Precede a signed negative constant with a minus sign (–); $FFFFFF85 and –$7B are equivalent in value.

- *Radix* is one of the following string constants:

  `hex  decimal  octal  binary  literal`

  You can supply numeric data as decimal, octal, hexadecimal, or literal data.

- *Numeric-type* is one of the following:

`bitstring['`*length*`']`	Bitstring of *length* bits (maximum 32).
`byte`	Byte (8-bit) field. This is the same as `bitstring[8]`.
`integer`	Integer (16-bit) field. This is the same as `bitstring[16]`.
`longint`	Long integer (32-bit) field. This is the same as `bitstring[32]`.

The resource compiler uses integer arithmetic and stores numeric values as integer numbers. The resource compiler translates Booleans, bytes, integers, and long integers to bitstring equivalents. All computations are done in 32 bits and truncated.

An error is generated if a value won't fit in the number of bits defined for the type. The valid ranges for values of `byte`, `integer`, and `longint` constants are as follows:

Type	Maximum	Minimum
`byte`	255	–128
`integer`	65,535	–32,768
`longint`	4,294,967,295	–2,147,483,648

Boolean type

A Boolean is a single bit with two possible states: 0 (or `false`) and 1 (or `true`). (`True` and `false` are global predefined identifiers.) Boolean values are declared as follows:

`boolean [ =` *constant* | *symbolic-value*... `];`

The type `boolean` declares a 1-bit field; this is equivalent to

```
unsigned bitstring[1]
```

◆ *Note:* This type is not the same as the Boolean variable defined by Pascal.

Character type

Characters are declared as follows:

`char [ =` *string* | *symbolic-value*... `];`

Type `char` declares an 8-bit field (this is the same as writing `string[1]`).

Here is an example:

```
type 0x0001  {
      char  dollar = "$",percent = "%";
};
resource 0x0001 (128)  {
      dollar
};
```

String type

String data types are specified like this:

string-type['['*length*']'] [= *string* | *symbol-value*...];

String-type is one of the following:

`[hex] string` Plain string (no length indicator or termination character is generated). The optional `hex` prefix tells DeRez to display it as a hexadecimal string; `string[`*n*`]` contains *n* characters and is *n* bytes long. The type `char` is shorthand for `String[1]`.

`pstring` Pascal string (a leading byte containing the length information is generated). `pstring[`*n*`]` contains *n* characters and is *n*+1 bytes long; `pstring` has a built-in maximum length of 255 characters, the highest value the length byte can hold. If the string is too long to fit the field, a warning is given and the string is truncated.

`wstring` Word string is a very large `pstring`. Its length is stored in the first two bytes. Therefore, a word string can contain up to 65,535 characters; `wstring[`*n*`]` contains *n* characters and is *n*+2 bytes long. The resource compiler reverses the order of the first two bytes (the length word) when creating a resource file; that is, the ordering of the length word is standard 65816, low-byte/high-byte order.

`cstring` C string (a trailing null byte is generated); `cstring[`*n*`]` contains *n*–1 characters and is *n* bytes long. A C string of length 1 can be assigned only the value `""`, because `cstring[1]` has room only for the terminating null.

Each string type may be followed by an optional *length* indicator in brackets (`[`*n*`]`). *Length* is an expression indicating the string length in bytes. *Length* is a positive number in the range $1 \le$ *length* $\le$ 2,147,483,647 for `string` and `cstring`, and in the range $1 \le$ *length* $\le$ 255 for `pstring`, and in the range $1 \le$ length $\le$ 65,535 for `wstring`.

◆ *Note:* You cannot assign the value of a literal to a string type.

If no length indicator is given, a `pstring`, `wstring`, or `cstring` stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right. The padding characters for all string types are nulls. If the data contains more characters than the length indicator provides for, the string is truncated and a warning message is given.

▲ **Warning** A null byte within a `cstring` is a termination indicator and may confuse DeRez and C programs. However, the full string, including the explicit null and any text that follows it, will be stored by the resource compiler as input. ▲

Point and rectangle types

Because points and rectangles appear so frequently in resource files, they have their own simplified syntax:

`point` [= *point-constant* | *symbolic-value...*];
`rect` [= *rect-constant* | *symbolic-value...*];

where

point-constant = '{'*x-integer-expr*, *y-integer-expr* '}'

and

rect-constant = '{'*integer-expr*, *integer-expr*, *integer-expr*, *integer-expr*'}'

These type statements declare a point (two 16-bit signed integers) or a rectangle (four 16-bit signed integers). The integers in a rectangle definition specify the rectangle's upper-left and lower-right points, respectively.

Fill type

The resource created by a resource definition has no implicit alignment. It is treated as a bit stream, and integers and strings can start at any bit. The `fill` specifier is a way of padding fields so that they begin on a boundary that corresponds to the field type.

The `fill` statement causes the resource compiler to add the specified number of bits to the data stream. The fill is always 0. The form of the statement is

`fill` *fill-size* ['[' *length*']'];

where *fill-size* is one of the following strings:

```
bit   nibble   byte   word   long
```

These declare a fill of 1, 4, 8, 16, or 32 bits (optionally multiplied by the *length* modifier). *Length* is an expression ≤ 2,147,483,647.

The following `fill` statements are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

The full form of a `type` statement specifying a fill might be:

`type $33 {`*data-type specifications;* `fill bit[2];};`

- *Note:* The resource compiler supplies 0s as specified by `fill` statements. DeRez does not supply any values for `fill` statements; it just skips the specified number of bits, or until data is aligned as specified.

Array type

An array is declared as follows:

`[ wide ]  array [`*array-name* | `'['`*length*`']'] '{'` *array-list* `'}';`

The *array-list,* a list of type specifications, is repeated zero or more times. The `wide` option outputs the array data in a wide display format (in DeRez)—the elements that make up the *array-list* are separated by a comma and space instead of a comma, return, and tab. Either *array-name* or `[`*length*`]` may be specified. *Array-name* is an identifier.

If the array is named, a preceding statement should refer to that array in a constant expression with the `$$countof(`*array-name*`)` function; otherwise DeRez will treat the array as an open-ended array. For example,

```
type rToolStartup {
    integer = 0;                    /* flags must be zero */
    Integer mode320 = 0,mode640 = $80;  /* mode to start quickdraw */
    Integer = 0;
    Longint = 0;
    integer = $$countof(TOOLRECS);      /* number of tools */
        array TOOLRECS {
            Integer;                /* ToolNumber */
            Integer;                /* version */
        };
      };
```

The `$$countof` function returns the number of array elements (in this case, the number of strings) from the resource data.

If [*length*] is specified, there must be exactly *length* elements.

Array elements are generated by commas. Commas are element separators. Semicolons are element terminators. In this example, however, it may be a good idea to use semicolons as element separators:

```
type 0x0099 {
      array Increment {
            integer = $$ArrayIndex(Increment);
      };
};
resource 0x0099 (0) {
      {     /* zero elements */
      }
};
resource 0x0099 (1) {
      {     /* two elements */
            ,
      }
};
resource 0x0099 (3) {
      }     /* two elements */
            ;;
      }
};
/* The only way to specify one element in an array that has all /*
/* constant elements is to use a semicolon terminator./*
resource '$99' (4) {
      {     /* one element */
            ;
      }
};
```

Switch type

The `switch` statement specifies a number of case statements for a given field or fields in the resource. The format is

`switch '{'` *case-statement...* `'}';`

where a *case-statement* has this form:

`case` *case-name* `:` [*case-body* `;`]...

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

`key` *data-type* = *constant*

Which `case` applies is based on the key value. For example,

```
type rControlTemplate {
        integer = 3+$$optionalcount (Fields);  /* pCount must be at */
                                               /* least 6 */
        longint;                              /* Application-defined ID */
        rect;                                  /* controls bounding */
                                               /* rectangle */
        switch {

        case SimpleButtonControl:
            key longint = 0x80000000;    /* procRef */
            optional Fields {
                integer;                       /* flags */
                integer;                       /* more flags */
                longint;                       /* refcon */
                longint;                       /* title ref */
                longint;                       /* color table ref */
                KeyEquiv;
            };

        case CheckControl:
            key longint = 0x82000000;          /* procRef */
            optional Fields {
                integer;                       /* flags */
                integer;                       /* more flags */
                longint;                       /* refcon */
                longint;                       /* title ref */
                integer;                       /* initial value */
                longint;                       /* color table ref */
                KeyEquiv;
            };
                ...and so on.
    };
```

Sample `type` statement

The following sample `type` statement is the standard declaration for an `rIcon` resource, taken from the `Types.rez` file:

```
type rIcon {
        hex integer;                    /* icon type bit 15
                                           1 = color, 0 = mono*/
image:
        integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
        integer;                      /* height of icon in pixels */
        integer;                      /* width of icon in pixels */
        hex string [$$Word(image)]; /* icon image */
mask:
        hex string;                   /* icon mask */
};
```

The `type` declaration consists of header information followed by a series of statements, each terminated by a semicolon (`;`). The header of the sample window declaration is

```
type rIcon
```

The header begins with the Type keyword followed by the resource type being declared—in this case, an icon. You can specify a standard Apple IIGS resource type, as shown in the *Apple IIGS Toolbox Reference,* Volume 3 (and as defined in the `Types.rez` file in the `13:RInclude` directory), or you can declare a resource type specific to your application.

The left brace (`{`) introduces the body of the declaration. The declaration continues for as many lines as necessary until a matching right brace (`}`) is encountered. You can write more than one statement on a line, and a statement may be on more than one line. Each statement represents a field in the resource data. Recall that comments can appear anywhere that white space can appear in the resource description file; comments begin with `/*` and end with `/*` as in C.

Symbol definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

name = *value* [, *name* = *value*]...

The "= *value*" part of the statement can be omitted or numeric data. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out; if *value* is omitted, it is assumed to be 1 greater than the previous value. (The value is assumed to be 0 if it is the first value in the list.) This is true for bitstrings (and their derivatives, `byte`, `integer`, and `longint`). For example,

```
integer     Emily, Kelly, Taylor,
            Evan, Trevor,
            Sparkle=8, Twinkle=16;
```

In this example, the symbolic names `Emily`, `Kelly`, `Taylor`, `Evan`, and `Trevor` are automatically assigned the numeric values 0, 1, 2, 3, and 4.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value; for example,

```
integer     Emily=0, Kelly=1, Taylor=2, Evan=3,
            Trevor=16, Sparkle=0, Twinkle=1, Raphael=2,
            Michaelangelo=3, Nagel=16;
```

`Delete`—Delete a resource

`Delete` allows you to delete a resource from an existing resource. You might need to delete resources from system disks and applications that need to translate menu and dialog text being shipped to international markets.

Syntax

`delete` *resource-type* ['(' *ID* [: *ID*] ')'];

Description

This statement deletes the resource of *resource-type* with the specified *ID* or *ID* range from the resource compiler output file. If *ID* or *ID* range is omitted, all resources of *resource-type* are deleted.

- *Note:* The `delete` function is valid only if you specify the `-a` (append) option on the resource compiler command line. (It wouldn't make sense to delete a resource while creating a new resource file from scratch.)

You can delete resources that have their protected bit set only if you use the `-ov` option on the resource compiler command line.

`Change`—Change a resource's vital information

This function allows you to change a resource's vital information, which includes the resource type, ID, and attributes.

Syntax

`change` *resource-type1* ['(' *ID* [: *ID*]')'] *resource-type2* '(' *ID* [, *attributes*...]')' ;

Description

This statement changes the resource of *resource-type1* with the specified *ID* or *ID* range in the resource compiler output file to a resource of *resource-type2* and the specified *ID*. If *ID* or *ID* range is omitted, all resources of *resource-type1* are changed.

- *Note:* The `change` function is valid only if you specify the `-a` (append) option on the resource compiler command line. (It wouldn't make sense to change resources while creating a new resource file from scratch.)

`Resource`—Specify resource data

`Resource` statements specify actual resources, based on previous `type` declarations.

Syntax

```
resource resource-type '(' ID [, attributes] ')' '{'
        [ data-statement [ , data-statement ]... ]
'}';
```

Description

This statement specifies the data for a resource of type *resource-type* and ID *ID*. The latest `type` declaration declared for *resource-type* is used to parse the data specification. *Data-statements* specify the actual data; *data-statements* appropriate to each resource type are defined in the next section.

The `resource` definition generates an actual resource. A `resource` statement can appear anywhere in the resource description file, or even in a separate file specified on the command line or as an `#include` file, as long as it comes after the relevant `type` declaration.

Data statements

The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration.

Base type	Instance types
`string`	`string, cstring, pstring, wstring, char`
`bitstring`	`boolean, byte, integer, longint, bitstring`
`rect`	`rect`
`point`	`point`

Switch data

Switch data statements are specified by using this format:

switch-name data-body

For example, the following could be specified for the `rControlTemplate` type given earlier:

```
.
.
.
CheckControl { enabled, "Check here" },
.
.
.
```

Array data

Array data statements have this format:
'{' [*array-element* [, *array-element*]...] '}'

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the `rStringList` resource:

```
resource rStringList (280) {
      {     "this",
            "is",
            "a",
            "test"
      }
};
```

Sample resource definition

This section describes a sample resource description file for an icon. (See the *Apple IIGS Toolbox Reference*, Volume 3 for information about resource icons.

Here again is the `type` declaration given previously under "Sample `type` Statement":

```
type rIcon {
        hex integer;                    /* icon type bit 15  1 = color,
                                           0 = mono */
image:
        integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
        integer;                        /* height of icon in pixels */
        integer;                        /* width of icon in pixels */
        hex string [$$Word(image)]; /* icon image */
mask:
        hex string;                     /* icon mask */
};
```

Here is a typical example of the icon data corresponding to this declaration:

```
resource rIcon (1) {
            0x8000,                          /* Kind */
            9,                               /* Height */
            32                               /* Width */
            $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
            $"FFFFFF00000000000000FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00000000000000FFFFFFFFFFFF"
            $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

            $"00000000000000000000000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFF0000000000000"
            $"00000000000000000000000000000000"
};
```

This data definition declares a resource of type `rIcon`, using whatever type declaration was previously specified for `rIcon`. The `8` in the resource type specification (`0x8000`) identifies this as a color icon.

The icon is 9 pixels high by 32 pixels wide.

The specification of the icon includes a pixel image and a pixel mask.

Labels

Labels support the more complicated resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels. The `rIcon` resource, for example, uses labels to specify the pixel image and mask of the icon.

Syntax

```
label ::=      character {alphanum}* ':'
 character ::=       '_' | A | B | C ...
alphanum ::=        character | number
 number ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Description

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear on a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See "Declaring Labels Within Arrays" later in this chapter for more information.

The value of a label is always the offset, ***in bits***, between the beginning of the resource and the position where the label occurs when mapped to the resource data. In this example,

```
type 0xCCCC {
    cstring;
endOfString:
    integer = endOfString;
};

resource 0xCCCC (8) {
  "Neato"
}
```

the integer following the `cstring` would contain:

```
( len("Neato") [5] + null byte [1] ) * 8 [bits per byte] = 48.
```

Built-in functions to access resource data

In some cases, it is desirable to access the actual resource data to which a label points. Several built-in functions allow access to that data:

- `$$BitField(`*label, startingPosition, numberOfBits*`)`
  Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.
- `$$Byte(`*label*`)`
  Returns the byte found at *label*.
- `$$Word(`*label*`)`
  Returns the word found at *label*.
- `$$Long(`*label*`)`
  Returns the long word found at *label*.

For example, the resource type `rPString` could be redefined without using a `pstring`. Here is the definition of `rPString` from `Types.rez`:

```
type rPString {
      pstring;
};
```

Here is a redefinition of `rPString` using labels:

```
type rPString {
len:  byte = (stop - len) / 8 - 1;
      string[$$Byte(len)];
stop: ;
};
```

Declaring labels within arrays

Labels declared within arrays may have many values. For every element in the array there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to *n* where *n* is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 0xFF01 {
      integer = $$CountOf(array1);
      array array1 {
            integer = $$CountOf(array2);
            array array2 {
foo:              integer;
            };
      };
};
resource 0xFF01 (128) {
      {
            {1,2,3},
            {4,5}
      }
};
```

In the example just given, the label `foo` takes on these values:

```
foo[1,1] = 32     $$Word(foo[1,1]) = 1
foo[1,2] = 48     $$Word(foo[1,2]) = 2
foo[1,3] = 64     $$Word(foo[1,3]) = 3
foo[2,1] = 96     $$Word(foo[2,1]) = 4
foo[2,2] = 112    $$Word(foo[2,2]) = 5
```

Another built-in function may be helpful in using labels within arrays:

`$$ArrayIndex(`*arrayname*`)`

This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label limitations

Keep in mind the fact that the APW Resource Compiler and DeRez are basically one-pass compilers. This will help you understand some of the limitations of labels.

- *Note:* To decompile a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can have only one undefined label:

```
type 0xFF01 {
      /* In the expression below, start is defined, next is undefined.*/
start:      integer = next - start;
      /* In the expression below, next is defined because it was used
         in a previous expression, but final is undefined.*/
middle:     integer = final - next;
next: integer;
final:
};
```

Actually, the resource compiler can compile types that have expressions containing more than one undefined label, but the `DeRez` cannot decompile those resources and simply generates `data` resource statements.

- *Note:* The label specified in `$$BitField()`, `$$Byte()`, `$$Word()`, and `$$Long()` must occur lexically before the expression; otherwise, an error is generated.

An example of using labels

In the following example, the definition for the `rIcon` resource uses the labels `image` and `mask`.

```
type rIcon {
     hex integer;      /* Icon Type bit 15  1 = color, 0 = mono */
image:
     integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
     integer;                     /* height of icon in pixels */
     integer;                     /* width of icon in pixels */
     hex string [$$Word(image)]; /* icon image */
mask:
     hex string;                  /* icon mask */
};
```

In the data corresponding to that definition, pixel images are provided for the image and mask.

```
resource rIcon (1) {
            0x8000,                          /* Kind */
            9,                               /* Height */
            32                               /* Width */
            $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
            $"FFFFFF00000000000000FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
            $"FFFFFF00000000000000FFFFFFFFFFFF"
            $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

            $"00000000000000000000000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000FFFFFFFFFFFFFF0000000000000"
            $"00000000000000000000000000000000"
};
```

Preprocessor directives

Preprocessor directives substitute macro definitions and `include` files and provide if-then-else processing before other resource compiler processing takes place.

The syntax of the preprocessor is very similar to that of the C-language preprocessor. Preprocessor directives must observe these rules and restrictions:

- Each preprocessor statement must begin on a new line, be expressed on a single line, and be terminated by a return character.
- The pound sign (#) must be the first character on the line of a preprocessor statement (except for spaces and tabs).
- *Identifiers* (used in macro names) may be letters (A–Z, a–z), digits (0–9), or the underscore character (_).
- Identifiers may be any length.
- Identifiers may not start with a digit.
- Identifiers are not case sensitive.

Variable definitions

The `#define` and `#undef` directives let you assign values to identifiers:

```
#define  macro data
#undef  macro
```

The `#define` directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (\), which functions as the resource compiler's escape character. Here is an example:

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

Quotation marks within strings must also be escaped.

The `#undef` directive removes the previously defined identifier *macro*. Macro definitions can also be removed with the `-undef` option on the resource compiler command line.

The following predefined macros are provided:

Variable	Value
`true`	1
`false`	0
`rez`	1 or 0 (1 if the resource compiler is running, 0 if DeRez is running)
`derez`	1 or 0 (0 if the resource compiler is running, 1 if DeRez is running)

`include` directive

The `#include` directive reads a text file:

`#include` *file*

The maximum nesting is to ten levels. For example,

```
#include ($$Shell("APW")) "MyProject MyTypes.rez"
```

Note that the `#include` preprocessor directive (which includes a file) is different from the previously described `include` statement, which copies resources from another file.

If-then-else processing

These directives provide conditional processing:

```
#if expression
[ #elif expression ]
[ #else ]
#endif
```

- *Note: Expression* is defined later in this chapter. When used with the `#if` and `#elif` directives, *expression* may also include the expression:

  `defined` *identifier* or `defined '('` *identifier* `')'`

The following may also be used in place of `#if`:

```
#ifdef macro
#ifndef macro
```

For example,

```
#define Thai
Resource  rPstring  (199)  {
#ifdef English
      "Hello"
#elif defined (French)
      "Bonjour"
#elif defined (Thai)
      "Sawati"
#elif defined (Japanese)
      "Konnichiwa"
#endif
};
```

`printf` directive

The `#printf` directive is provided to aid in debugging resource description files.

`#printf` (*formatString, arguments...*)

The format of the `#printf` statement is exactly the same as that of the `printf` statement in the C language, with one exception: There can be no more than 20 arguments. This is the same restriction that applies to the `$$format` function. The `#printf` directive writes its output to diagnostic output. Note that the `#printf` directive does not end with a semicolon.

Here's an example:

```
#define           Tuesday           3
#ifdef Monday
#printf("The day is Monday, day #%d\n", Monday)
#elif defined(Tuesday)
#printf("The day is Tuesday, day #%d\n", Tuesday)
#elif defined(Wednesday)
#printf("The day is Wednesday, day #%d\n", Wednesday)
#elif defined(Thursday)
#printf("The day is Thursday, day #%d\n", Thursday)
#else
#printf("DON'T KNOW WHAT DAY IT IS!\n")
#endif
```

The file just listed generates this text:

```
The day is Tuesday, day #3
```

`append` directive

This directive allows you to specify additional files to be compiled by the resource compiler.

`#append` *filename*

This directive must appear between `resource` or `type` statements. The *filename* variable is the name of the next file to be compiled. The same search rules apply here that apply to the `#include` directive. Normally you should place this directive at the end of a file because everything after it is ignored. Do not place an `#append` directive in an include file.

If you use more than one `#append` directive, the order in which you put them is important. When the resource compiler sees an `#append` directive, it checks the language type of the appended file. If it is the same language, that is, `REZ`, the effect is the same as if the files had been concatenated into a single file. If they are in different languages, APW quits the resource compiler and begins a new assembly or compilation. Two examples will illustrate why the order is important.

In the first example, suppose you have the following three files, each appended to the preceding file.

```
file1.rez
file2.rez
file3.asm
```

The `Compile` command calls the resource compiler to process `file1.rez` because the language is `REZ`. When the resource compiler encounters the `#append` directive for `file2.rez` it continues processing as if `file.rez` and `file2.rez` had been concatenated into a single file. When it encounters the `#append` directive for `file3.asm`, the resource compiler finishes processing and returns control to the APW Shell which calls the assembler to assemble `file3.asm`.

The result is different if the order of the files is changed, as follows:

```
file1.rez
file3.asm
file2.rez
```

The resource compiler processes `file1.rez`. When it encounters the `#append` directive for `file3.asm`, the resource compiler finishes processing and returns control to the APW Shell because the language stamp is different. The APW Shell calls the assembler to processes `file3.asm`. When the assembler is finished processing, it returns control to the shell which calls the resource compiler to process `file2.rez`. However, since this is a separate compilation from that of `file1.rez`, the resource compiler knows nothing about symbols from `file1.rez` when compiling `file2.rez`.

`DeRez` handles `#append` directives differently from the resource compiler. For `DeRez` the file being appended must have a language stamp of `REZ` or `DeRez` will treat the `#append` directive as an end-of-file marker. `DeRez` will not return control to the APW Shell after finishing processing. Therefore, in the previous example, `DeRez` would process `file1.rez` only and then finish processing.

Resource description syntax

This section describes the details of the resource description syntax.

Numbers and literals

All arithmetic is performed as 32-bit signed arithmetic. The basic constants are shown in Table 3-2.

■ **Table 3-2** Numeric constants

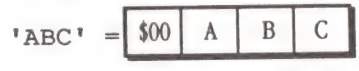
Numeric type	Form	Meaning
Decimal	*nnn...*	Signed decimal constant between 2,147,483,647 and –2,147,483,648.
Hexadecimal	`0x`*hhh...*	Signed hexadecimal constant between 0X7FFFFFFF and 0X80000000.
	`$`*hhh...*	Alternate form for hexadecimal constants.
Octal	`0`*ooo...*	Signed octal constant between 017777777777 and 020000000000.
Binary	`0B`*bbb...*	Signed binary constant between 0B11111111111111111111111111111111 and 0B10000000000000000000000000000000.
Literal	`'`*aaaa*`'`	One to four printable ASCII characters or escape characters. If there are fewer than four characters in the literal, the characters to the left (high bits) are assumed to be $00. Characters that are not in the printable character set, and are not the characters `\'` and `\\` (which have special meanings), can be escaped according to the character escape rules. (See "Strings" later in this section.)

Literals and numbers are treated in the same way by the resource compiler. A **literal** is a value within single quotation marks; for instance, `'A'` is a number with the value 65; on the other hand, `"A"` is the character *A* expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that `"A"` is not a valid number and `'A'` is not a valid string.) The following numeric expressions are all equivalent:

```
'B'
66
'A'+1
```

Literals are padded with nulls on the left side so that the literal `'ABC'` is stored as shown in Figure 3-3.

■ **Figure 3-3** Padding of literals



Expressions

An expression may consist of simply a number or a literal. Expressions may also include numeric variables, labels, and system functions.

Table 3-3 lists the operators in order of precedence with highest precedence first—groupings indicate equal precedence. Evaluation is always left to right when the priority is the same. Variables are defined following the table.

■ **Table 3-3** Resource description expression operators

Operator	Meaning
1. (*expr*)	Forced precedence in expression calculation
2. *−expr*	Arithmetic (two's complement) negation of *expr*
~expr	Bitwise (one's complement) negation of *expr*
! expr	Logical negation of *expr*
3. *expr1 * expr2*	Multiplication
expr1 / expr2	Division
expr1 % expr2	Remainder from dividing *expr1* by *expr2*
4. *expr1 + expr2*	Addition
expr1 − expr2	Subtraction
5. *expr1 << expr2*	Shift left—shift *expr1* left by *expr2* bits
expr1 >> expr2	Shift right—shift *expr1* right by *expr2* bits
6. *expr1 > expr2*	Greater than
expr1 >= expr2	Greater than or equal to
expr1 < expr2	Less than
expr1 <= expr2	Less than or equal to
7. *expr1 == expr2*	Equal
expr1 != expr2	Not equal
8. *expr1 & expr2*	Bitwise AND
9. *expr1 ^ expr2*	Bitwise XOR
10. *expr1 | expr2*	Bitwise OR
11. *expr1 && expr2*	Logical AND
12. *expr1 || expr2*	Logical OR

The logical operators !, >, >=, <, <=, ==, !=, &&, and || evaluate to 1 (true) or 0 (false).

Variables and functions

Some resource compiler variables contain commonly used values. All `Rez` variables start with `$$` followed by an alphanumeric identifier.

The following variables and functions have string values (typical values are given in parentheses):

`$$Date` — Current date. It is useful for putting time-stamps into the resource file. The format of the string is: *weekday, month dd, yyyy*. For example, `August 10, 1989`.

`$$Format("`*formatString*`",` *arguments*`)` — Works just like the `#printf` directive except that `$$Format` returns a string rather than printing to standard output. (See "Print Directive" earlier in this chapter.)

`$$Resource("`*filename*`",'`*type*`',` *ID*`)` — Reads the resource `'`*type*`'` with the ID *ID* from the resource file *filename*, and returns a string.

`$$Shell("`*stringExpr*`")` — Current value of the exported shell variable {*stringExpr*}. Note that the braces must be omitted, and the double quotation marks must be present.

`$$Time` — Current time. It is useful for time-stamping the resource file. The format is: `"hh:mm:ss"`.

`$$Version` — Version number of the resource compiler. ("V1.0")

These variables and functions have numeric values:

`$$Attributes` — Attributes of resource from the current resource.

`$$BitField(`*label, startingPosition, numberOfBits*`)` — Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte(`*label*`)` — Returns the byte found at *label*.

`$$CountOf (`*arrayName*`)` — Returns the number of elements in the array *arrayName*.

`$$Day` — Current day (range 1–31).

`$$Hour` — Current hour (range 0–23).

`$$ID` — ID of resource from the current resource.

`$$Long(`*label*`)` — Returns the longword found at *label*.

`$$Minute` — Current minute (range 0–59).

`$$Month`	Current month (range 1–12).
`$$OptionalCount (`*OptionalName*`)`	Returns the number of items explicitly specified in the block *OptionalName*.
`$$PackedSize (`*Start, RowBytes, RowCount*`)`	Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the toolbox routine `UnpackBytes` *RowCount* times. `$$PackedSize( )` returns the unpacked size of the data found at *Start*. Use this function only for decompiling resource files. An example of this function is found in `Pict.rez`.
`$$ResourceSize`	Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the number of bytes that have been compiled so far for the current resource.
`$$Second`	Current second (range 0–59).
`$$Type`	Type of resource from the current resource.
`$$Weekday`	Current day of the week (range 1–7, that is, Sunday–Saturday).
`$$Word (`*label*`)`	Returns the word found at *label*.
`$$Year`	Current year.

Strings

There are two basic types of strings:

Text string	`"`*a...*`"`	The string can contain any printable character except ' " ' and '\'. These and other characters can be created through escape sequences. (See Table 3-4.) The string `""` is a valid string of length 0.
Hexadecimal string	`$"`*hh...*`"`	Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string `$""` is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.)

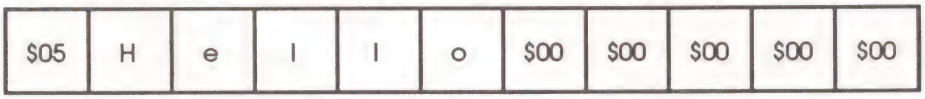
Figure 3-4 shows a Pascal string declared as

```
pstring [10];
```

whose data definition is

```
"Hello"
```

■ **Figure 3-4** Internal representation of a Pascal string



In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma) or brace signifies the end of the string data. A side effect of string continuation is that a sequence of two quotation marks ("") is simply ignored. For example,

```
"Hello ""out "
"there."
```

is the same string as

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash like this:

```
\"
```

Escape characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence

```
\n
```

Valid escape sequences are shown in Table 3-4.

■ **Table 3-4** Resource compiler escape sequences

Escape sequence	Name	Hexadecimal value	Printable equivalent
`\t`	Tab	$09	None
`\b`	Backspace	$08	None
`\r`	Return	$0A	None
`\n`	Newline	$0D	None
`\f`	Form feed	$0C	None
`\v`	Vertical tab	$0B	None
`\?`	Rubout	$7F	None
`\\`	Backslash	$5C	\
`\'`	Single quotation mark	$27	'
`\"`	Double quotation mark	$22	"

◆ *Note to C programmers:* The escape sequence `\n` produces an ASCII code of 13 in the output stream, while the `\r` sequence produces an ASCII code of 10. The `stdio` functions in the APW C library interpret the ASCII code 13 as a return character and do not perform any special processing when this character is output. ASCII code 10 is interpreted by the library functions as a newline character, and the functions will output both the character (which is a line-feed character) and then a carriage return character. This produces a carriage return/line-feed sequence, which correctly moves the cursor to the beginning position of the next line on the screen. If a carriage return only is output, the cursor simply moves to the beginning position of the current line of the screen. To produce a newline character in a string and have it recognized as a newline character by the C library functions, you must use the `\r` escape sequence in your resource definitions.

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are:

Base	Number form	Digits	Example
2	`\0B`*bbbbbbbb*	8	`\0B01000001`
8	`\`*ooo*	3	`\101`
10	`\0D`*ddd*	3	`\0D065`
16	`\0x`*hh*	2	`\0X41`
16	`\$`*hh*	2	`\$41`

Here are some examples:

```
\077  /* 3 octal digits */
\0xFF /* '0x' plus 2 hex digits     */
\$F1\$F2\$F3      /* '$' plus 2 hex digits      */
\0d099      /* '0d' plus 3 decimal digits */
```

- *Note to C programmers:* An octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write `AB\007CD`, not `AB\7CD`.

You can use the DeRez command-line option `-e` to print characters that would otherwise be escaped (characters preceded by a backslash, for example). Normally, only characters with values between $20 and $7E are printed as Apple IIGS characters. With this option, however, all characters (except null, newline, tab, backspace, form-feed, vertical tab, and rubout) will be printed as characters, not as escape sequences. See the DeRez command in Chapter 4 of the *APW Reference.*

Using the APW Resource Compiler

The APW Resource Compiler is a one-pass compiler; that is, in one pass it resolves preprocessor macros, scans the resource description file, and generates code into a code buffer. It then writes the code to a resource file.

The resource compiler is invoked by the APW Shell `Compile` (or `Assemble`) command. This command checks the language type of the source file (in this case, `rez`) and calls the appropriate compiler or assembler (in this case, the resource compiler). The next section describes the syntax and use of the `Compile` command, specifically as it is used to call the resource compiler.

Compile

`Compile` [*option...*] *file1* [*file2*] [...] [`keep=`*outfile*]
[`names=` (*seg1* [*seg2*] [...])] [`rez=` (*option...*)
[*language2*`=` (*option...*)] [...]

The `Compile` internal command is an alias for the `Assemble` command. It checks the language of the source file specified on the command line and calls the appropriate compiler, resource compiler, or assembler.

This section describes the `Compile` command as it is used to call the APW Resource Compiler, which compiles resource description files (`rez`-language source files) to create the resource fork of an Apple IIGS load file. This description includes the `rez`-language-specific options of the `Compile` command. See the *APW C Reference* manual for the C-language options, and Chapter 4 of the *APW Reference* for additional information on the `Assemble` and `Compile` commands.

The data used to build the resource file can come from other text files (via `#include` and `read` directives in the resource description file) and from other resource files (via the `include` directive) as well as directly from the resource description file(s).

`Compile` for the `rez` language includes macro processing, full expression evaluation, and built-in functions.

You can include more than one source file from different languages, or use `#append` directives (or the equivalent) to tie together source files written in different computer languages; APW compilers and assemblers check the language type of each file and return control to the shell when a different language must be called. All parameters are passed to every compiler or assembler. Each compiler or assembler uses only the parameters that it recognizes and ignores the others. The reference manual for a compiler or assembler contains a list of the options that it accepts. See Chapter 3 of the *APW Reference* for a description of the assembly and compilation process.

option... You can specify as many of the following options as you wish (up to 32) by separating the options with spaces.

- *Note:* Command-line parameters (those described here) override source-code options when there is a conflict. Like other compilers and assemblers, the APW Resource Compiler does not use all the command-line parameters; for example, it ignores the `+l|-l` and `+s|-s` options. If you include a parameter that the resource compiler cannot use, it ignores the parameter without generating an error.

`+e|-e` If you specify `+e`, when the compiler terminates execution due to a fatal error, it calls the APW Editor. The editor displays the source file with the offending line on the fifth line on the screen (or as far down on the screen as possible, if the error is in one of the first four lines of the file). If you specify `-e` and a fatal error occurs, the compiler returns you to the shell's command line or to the Exec file that executed the command. The default for this option is `+e` when the command is executed from the command line, and `-e` when the command is executed from an Exec file.

`+l|-l` The APW Resource Compiler ignores this option.

`+m|-m` The APW Resource Compiler ignores this option.

`+s|-s` The APW Resource Compiler ignores this option.

`+t|-t` If you select `+t`, any error causes the compile to terminate. If you omit this option or select `-t`, only fatal errors cause immediate termination of the compile. Note that if you select both `+t` and `+e`, any error causes the shell to call the APW Editor and display the offending line as the fifth line on the screen.

`+w|-w` If you select `+w`, the compiler stops and waits for a keypress when any error occurs, to give you the opportunity to read the error message and decide whether to continue (that is, to continue the compilation in case of a nonfatal error or to call the editor in case of a fatal error). The compiler signifies that it is waiting for a keypress by displaying an hourglass character on the screen. Press Command-period to halt execution, or press any character key or the Space bar to continue. If you omit this option or select `-w`, execution continues without pausing when an error occurs.

file1 file2 ... The full or partial pathnames (including the filenames) of the source files to be assembled (or compiled).

You can include as many source, object, and library files as you choose, and they can be of different APW language types. The APW Shell will finish compiling the first file on the command line and then check the language type of the next file and call the appropriate compiler or assembler.

You could compile various resource description files in this way to create multiple resource files. However, if you want to compile several resource description files into a single resource file, use `#include` and `#append` directives within the resource description file to be compiled.

`keep=`*outfile* Use this parameter to specify the pathname or partial pathname (including the filename) of the output file. No output file is written unless you specify one with this parameter. There must be no spaces between `keep` and the equal sign (=).

Keep the following points in mind regarding the `keep=` parameter:

- You can specify a default filename for object files by using the `KeepName` shell variable. Shell variables are described in Chapter 3 of the *APW Reference.*
- To use the `keep=` parameter with multiple source files, you must use one or more wildcard characters in the `keep=` parameter.
- If resource files with the root filename *outfile* already exist, they are overwritten without a warning when this command is executed; that is, the resource fork of the existing file, not the data fork, is overwritten.

`names=(`*seg1* [*seg2*] [...])

This parameter is ignored by the APW resource compiler, which always compiles all resources within the specified description file. The APW Assembler uses this parameter for partial assembly.

`rez=(`*option* ...) This parameter allows you to pass parameters directly to the APW Resource Compiler. Between the parentheses insert one or more of the options listed next. Note that the APW Shell does no error checking on this string before passing it through to the resource compiler. Note that no spaces are permitted immediately before or after the equal sign in this parameter. This parameter is a special case of the *language1*`=(`*option* ...) parameter.

△ **Important** You can specify up to 31 options in this way. The resource compiler will ignore any subsequent options that you specify without generating an error message. △

The `rez` language has the following options:

`-a[ppend]` This option appends the resource compiler's output to the output file rather than replacing the output file.

`-d[efine]` *macro*`[=`*data*`]`

This option defines the macro variable *macro* to have the value *data*. If *data* is omitted, *macro* is set to the null string—note that this still means that *macro* is defined. Using the `-d` option is the same as writing

`#define` *macro* [*data*]

at the beginning of the input.

`-flag SYSTEM` This option sets the resource file flag for the system.

`-flag ROM` This option sets the resource file flag for ROM.

`-i` *pathname(s)*

This option searches the following pathnames for `#include` files. It can be specified more than once. The paths are searched in the order they appear on the command line.

```
…rez=(-i 13:rinclude:stuff.rez-i 13:rinclude:newstuff.rez)
```

`-m[odification]`

Don't change the output file's modification date. If an error occurs, the output file's modification date is set to 0, even if you use this option.

`-ov` This option overrides the protected bit when replacing resources with the `-a` option.

`-p[rogress]` This option writes version and progress information to diagnostic output.

`-rd` This option suppresses warning messages if a resource type is redeclared.

`-s` *pathname(s)* This option searches the following pathnames for resource `include` files.

`-t[ype]` *typeExpr*

This option sets the type of the output load file to *filetype*. You can specify a hexadecimal number, a decimal number, or a mnemonic for the file type. If the `-t` option is not specified, the file type of the load file is $B3.

`-u[ndef]` *macro*

This option undefines the macro variable *macro*. It is the same as writing

`#undef` *macro*

at the beginning of the input. It is meaningful to undefine only the preset macro variables.

- *Note:* A space is required between an option and its parameters.

language2=(*option* ...) ...

This parameter, like the `rez=(`*option* ...`)` parameter, allows you to pass parameters directly to specific APW compilers or assemblers. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as shown by the `Show Languages` command), an equal sign (=), and the string of options enclosed in parentheses. The contents and syntax of the options string is specified in the compiler or assembler reference manual. Note that the APW Shell does no error checking on this string before passing it through to the compiler or assembler. You can include option strings in the command line for as many languages as you wish; if a language compiler is not called, the string for that language is ignored.

- *Note:* No spaces are permitted immediately before or after the equal sign in this parameter.

Press Command-period to stop the compilation after it has begun. The compiler may respond by halting execution and calling the editor with the first line of your source file at the top of the screen, or it may return you to the shell.

Appendix **Error Messages**

This appendix lists all LinkIIGS error messages.

LinkIIGS errors

This section lists the LinkIIGS error messages as they appear on the screen. Following each message is a reference number in parentheses. The reference number does not appear on screen with the message.

No Object Segments have been specified (1)

I/O error "*value*" occurred during *function* - file "*filename*" (2)

File not found (3)

Ran out of memory in function *function* ($*value* needed $*value* available) (4)

Not enough parameters were specified. (5)

Usage - LinkIIGS [options] objectFiles -lib libraryFile [-o output_file] (6)

File name missing after -o option (7)

Only one output Load File can be specified (8)

File Type missing after -t option (9)

Alias missing after -a option (10)

Segment name missing after -lseg option (11)

Origin missing after -org (12)

Redundant -org specifications (13)

Origin of 0 cannot be specified (14)

Wildcard spec "*xfilename*" has not been satisfied (15)

Library file missing after -lib (16)

-org value is not a number (17)

No KIND subfield specified after the -lseg option (18)

Unknown Segment Kind mnemonic (*value*) (19)

Unknown File Type mnemonic (*value*) (20)

Value missing in alias specification (21)

option (*value*) is not Hexadecimal (22)

option (*value*) is not Decimal (23)

option (*value*) is too large (max=$%1X) (24)

Unknown option (*value*) specified (25)

File "*filename*" is not an Object or Library File (27)

File "*filename*" is not a Library File (28)

Library File "*filename*" will be processed as an Object File (29)

Unsupported USING record ignored (30)

Unsupported LOCAL record ignored (31)

Unsupported EQU record ignored (32)

Unsupported MEM record ignored (33)

Unsupported ALIGN record ignored (34)

Load Segment record ($*value*) encountered in Object Segment *value* (35)

Unknown OMF record ($*value*) encountered (36)

Location Counter mismatch ($*value* != $*value*) in Object Segment *name* (37)

Duplicate symbol *name* in LSeg *name* OSeg *name* and LSeg *name* OSeg *name* (38)

File Aux Type missing after -at option (39)

ZEXPR truncation error (40)

BEXPR truncation error (41)

RELEXPR references an absolute address (42)

RELEXPR is out of range (value=$*value* max=$*value*) (43)

BEXPR reference is outside Load Segment (44)

RELEXPR reference is outside Load Segment (45)

Address of relocation patch not found in Load Segment *name*+$*value* (46)

Expression starts with an operator ($*value*) (47)

Expression contains consecutive operators ($*value* $*value*) (48)

Missing symbol - *name* (49)

Expression subtracts two labels (*label1 label2*) in different Load Segments (50)

Expression does not evaluate to single entity (51)

Expression contains an illegal op (%s) on the reloc (%s) (52)

Unsupported OMF Version (%*value*) found in Segment $*value* in file *filename* (53)

Bad Segment Header found in Segment $*value* in file "*filename*" (54)

Illegal segment name specification *name* (55)

value extends past Bank Size ($*value*>$*value*) (56)

expected org ($*value*) not equal to actual org ($*value*) (57)

ObjSegName is absolute but Load Segment is not (58)

Segment specification not satisfied: (59)

Illegal ORG for an Absolute Bank Segment (60)

Library File "*filename*" specified more than once (61)

More than one -t option specified (62)

Dangling org before -lseg *value* (63)

Bad Library Dictionary Segment found in file *filename* (64)

Run Time Library contains a ~globals segment (65)

Read-only file system (66)

No space left on device (67)

Expression contains an unknown operator ($*value*) (68)

Terminated by operator (69)

ALIGN of $*value* is not supported by the System Loader (only $100 and $10000) (70)

ObjSegName has inconsistent ALIGN ($*value*) and ORG ($*value*) (71)

ObjSegName is aligned but Load Segment *name* is not (72)

ObjSegName is a Skip Segment but is not absolute (73)

Unsupported OMF Revision (%*value*) found in Segment $*value* in file *filename* (74)

Multiple Segment Kind (*name*) redundant or inconsistent (75)

Incorrect Segment Number ($*value*) found in Segment $*value* in file *filename* (76)

Run Time Library prefix is missing (77)

Trace size is missing (78)

Illegal private symbol reference - *reference* (79)

Illegal ORG for an Bank Relative Segment (80)

Expression contains an illegal op (*value*) on two relocs (*value value*) (81)

Negative ORG record (-$*value*) too large (82)

ObjSegName has ALIGN=$*value* which is greater than ALIGN=$*value* of Load Segment *name* (83)

More than one -at option specified (84)

Unknown File Aux Type mnemonic (*value*) (85)

Parameter in Standard Input File is too large (>*value* chars) (86)

Too many symbols to sort (MAX=*value*) (87)

Load Segment *name* is empty (88)

Internal Linker Error (attempt to free an odd address - $*value*) (89)

Internal Linker Error (attempt to free an illegal address - $*value*) (90)

Direct Page/Stack Segment *value* is not being linked into bank 0 (91)

A \~globals\ Object Segment is being linked into Load Segment *name* (92)

Internal Linker Error (attempt to write to an illegal address - $*value*) (93)

A zero-length label encountered (94)

Glossary

absolute-bank segment: A load segment that is restricted to a specified bank but can be relocated within that bank.

absolute code: Program code that must be loaded at a specific address in memory and never moved.

absolute segment: A segment that can be loaded only at one specific location in memory. Compare **relocatable segment.**

access byte: An attribute of a ProDOS 16 file that determines what types of operations, such as reading or writing, may be performed on the file.

accumulator: The register in the 65C816 microprocessor of the Apple IIGS used for most computations.

address: A number that specifies the location of a single byte of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (in decimal) or from $00 00 00 to $FF FF FF (in hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($00 00 to $FF FF).

alias: An alternative name for a command. You can use an alias for a command in exactly the same way as you would use the command.

Apple key: See **Command key**.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS .

application prefix: The prefix of the last application launched.

APW Linker: The linker supplied with APW.

APW Shell: The shell program of APW. The APW Shell provides the interface between APW programs and ProDOS and between the user and APW.

assembler: A program that produces object files from source files written in assembly language.

bank: A 64 KB (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

binary file format: The ProDOS 8 loadable file format, consisting of one absolute memory image along with its destination address. A file in binary file format has ProDOS file type $06 and is called a BIN file. The System Loader cannot load BIN files.

BIN file: A file in binary file format.

block: 1. A unit of data storage or transfer, typically 512 bytes. 2. A contiguous, page-aligned region of computer memory of arbitrary size, allocated by the Memory Manager. Also called a memory block.

boot prefix: See **startup prefix**.

catalog: See **directory.**

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as 1-byte values.

code segment: An object segment that contains program code. Code segments are provided for programs that differentiate between code and data segments.

Command key: A modifier key on the Apple IIGS keyboard marked with an Apple icon. It performs the same functions as the Open Apple key on standard Apple II machines.

command line: See **shell command line.**

command table: A text file containing a list of command names, command types (internal, or *command;* external, or *utility;* and *language*), and command or language numbers. The APW Shell checks the command table each time you execute a command. If it finds the command in the command table, it executes that command; if it doesn't find the command in the command table, it looks for a program with that name and attempts to run that program.

compiler: A program that produces object files from source files written in a high-level language such as C.

conditional assembly: A feature of an assembler that allows the programmer to define macros or other pieces of code so that the assembler assembles them differently under different conditions.

conditional compile: In a high-level language such as C, the use of preprocessor commands to vary the output depending on compile-time conditions.

controlling program: A program that loads and runs other programs without itself relinquishing control. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell is a controlling program.

Control Panel: A **desk accessory** that lets you change certain system parameters, such as speaker volume, display colors, and configuration of slots and ports.

cross development: Using MPW on a Macintosh computer to develop programs for the Apple IIGS. See also **native development**.

current application: The application program currently loaded and running. Every application program is identified by a **User ID** number; the current application is defined as that application whose User ID is the present value of the `USERID` global variable.

current language: The APW language type that is assigned to a file opened by the APW Editor. If an existing file is opened, the current language changes to match that of the file.

current prefix: The prefix used by the APW Shell if a partial pathname is used.

data segment: An object segment that consists primarily of data. Data segments are provided for programs that differentiate between code and data segments.

default prefix: See **current prefix.**

desk accessory: A small, special-purpose program that is available to the user regardless of which application is running. The Control Panel is an example of a desk accessory.

directory: A file that contains a list of the names and locations of other files stored on a disk. Directories are either **volume directories** or **subdirectories.** A directory is sometimes called a catalog.

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (1-byte) address because its high-address byte is always $00 and its middle-address byte is the value of the 65C816 processor's direct register. Coresident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page.** The term *direct page* is often used informally to refer to the lower portion of the **direct-page/stack space.**

direct-page/stack segment: A load segment used to preset the direct-page and stack registers and to set the initial contents of the direct-page/stack space for an application.

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's **direct page** and **stack.** Initially, the 65C816 processor's **direct register** contains the base address of the space, and its stack register contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

dispose: To deallocate a memory block permanently. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge.**

dormant: Said of a program that is not being executed, but whose essential parts are all in the computer's memory. A dormant program may be quickly restarted because it need not be reloaded from disk.

DumpObj utility: A program that lists the contents of an object file. The user can specify that the listing bd object module format (OMF) operation codes and records, 65816 assembly-language code, or hexadecimal codes.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare **static segment.**

emulation mode: 1. The state in which the Apple IIGS's 65C816 processor functions like a 6502 processor in all respects except clock speed. 2. The state in which the computer functions like an 8-bit Apple II.

event: A notification to an application of some occurrence (such as an interrupt generated by a keypress) to which the application may want to respond.

event-driven program: A kind of program that responds to user inputs in real time by repeatedly testing for **events.** An event-driven program does nothing until it detects an event such as a keypress or a click of the mouse button.

Exec file: A file of APW Shell commands that, when executed, executes each command in turn as if it had been entered from the keyboard. You can pass parameters into Exec files and can include them in the command table as utilities.

external command: An APW utility program that functions like an APW Shell command.

external reference: A reference to a symbol defined in another segment. External references must be to global symbols.

fatal error: An error serious enough to make the computer halt execution.

field: A string of ASCII characters or a value that has a specific meaning to some program. Fields may be of fixed length, or they may be separated from other fields by field delimiters. For example, each parameter in a segment header constitutes a field.

field delimiter: A character or value that designates the start or end of a field. For example, in a BASIC file each field begins and ends with a return character.

filename: The string of characters that identifies a particular file within a disk **directory.** ProDOS 16 filenames can be up to 15 characters long and can specify directory files, subdirectory files, text files, source files, object files, load files, or any other ProDOS 16 file type. Compare **pathname.**

file number: A reference number assigned to a specific file. The loader assigns a file number to each load file in a program; the MakeLib utility program assigns a file number to each object file incorporated in a library file.

file number cross-reference: The part of the pathname table that contains load-file numbers and pointers to their corresponding pathnames.

file type: An attribute in a ProDOS 16 file's directory entry that characterizes the contents of the file and indicates how the file may be used. On disk, file types are stored as numbers; in a directory listing, they are often displayed as three-character mnemonic codes.

finder: A program that performs file and disk utilities (formatting, copying, renaming, and so on) and also starts applications at the request of the user.

full pathname: The complete name by which a file is specified. A full pathname always begins with a slash (/), because a volume directory name always begins with a slash. See also **pathname.**

global symbol: A label in a code segment that is either the name of the segment or an entry point to it. Global symbols may be referenced by other segments. Compare **local symbol.**

GS/OS: A 16-bit operating system developed for the Apple IIGS computer. GS/OS replaces ProDOS 16 as the preferred Apple IIGS operating system.

GS/OS string: An ASCII character string preceded by a (2-byte) word whose numeric value is the number of 1-byte characters in the strings. A GS/OS string can be much longer than a **Pascal string.**

handle: See **memory handle.**

hexadecimal: The base-16 system of numbers, using the ten digits 0 through 9 and the six letters *A* through *F*. Hexadecimal numbers can be converted easily and directly to binary form, because each hexadecimal digit corresponds to a sequence of four bits. In Apple manuals hexadecimal numbers are usually preceded by a dollar sign ($).

high-level language: A programming language that is relatively easy for people to understand. A single statement in a high-level language typically corresponds to several instructions of machine language. Compare **low-level language.**

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory.

index register: A register in a computer processor that holds an index for use in indexed addressing. The 6502 and 65C816 microprocessors used in the Apple II family of computers have two index registers, called the X register and the Y register.

initialization segment: A segment in an initial load file that performs any initialization the program requires.

initial load file: The first file of a program to be loaded into memory. It contains the program's main segment and the load-file tables (jump table segment and pathname segment) needed to load dynamic segments and run-time libraries.

internal command: An APW Shell command that is executed by the shell program itself, rather than by a utility program.

INTERSEG record: Part of a relocation dictionary. It contains relocation information for external (intersegment) references.

jump table: A table contructed in memory by the System Loader from all jump table segments encountered during a load. The jump table contains all references to dynamic segments that may be called during execution of the program.

jump table directory: A master list in memory, containing pointers to all segments that make up the jump table.

jump table segment: A segment in a load file created by the linker that provides the information the loader needs to locate dynamic segments as they are needed during program execution. The loader creates a linked list in memory called the **jump table,** which indicates the location of all jump table segments in memory.

KB: 1024 bytes.

kind: See **segment type.**

language card: Memory with addresses between $D000 and $FFFF on any Apple II–family computer. It includes two RAM banks in the $D*xxx* space, called bank-switched memory. The language card was originally a peripheral card for the 48 KB Apple II or Apple II Plus that expanded its memory capacity to 64 KB and provided space for an additional dialect of BASIC.

language command: A command that changes the APW current language.

launch: To cause a program to be loaded into memory and begin execution.

library dictionary segment: The first segment of a library file. It contains a list of all the symbols in the file and their locations in the file. The linker uses the library dictionary segment to find the segments it needs.

library file: An object file containing object segments, each of which can be used in any number of programs. The linker can search through the library file for segments that have been referenced in the program source file. A library contains a **library dictionary segment.**

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

link map: A listing produced by the linker that gives the name, length, and starting location of each segment in a load file.

load file: The output of the linker. Load files contain memory images that the system loader can load into memory, and **relocation dictionaries** that the loader uses to relocate references.

load segment: A segment in a load file. Any number of object segments can go into the same load segment.

local symbol: A label defined only within an individual segment. Other segments cannot access the label. Compare **global symbol.**

loop: A section of a program that is executed repeatedly until a limit or condition is met, such as an index variable's reaching a specified ending value.

low-level language: A programming language, such as assembly language, that is relatively close to the form the computer's processor can execute directly. One statement in a low-level language corresponds to a single machine-language instruction. Compare **high-level language.**

macro: A single keystroke or command that a program replaces with several keystrokes or command. For example, the APW Editor allows you to define macros that execute several editor keystroke commands; the APW Assembler allows you to define macros that execute instructions and directives. APW also provides a library of predefined assembler macros.

macro assembler: A type of assembler that allows the programmer to define sequences of several assembly-language instructions as single pseudo instructions called **macros.**

main segment: The first static segment (other than initialization segments) in the initial load file of a program. It is loaded first and never removed from memory until the program terminates.

MakeLib utility: A program that creates library files from object files.

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

memory block: See **block.**

memory handle: The identifying number of a particular block of memory. A memory handle is a pointer to a master pointer to the memory block.

memory image: A portion of a disk file or segment that can be read directly into memory.

Memory Manager: A program in the Apple IIGS Toolbox that manages memory use. The Memory Manager keeps track of how much memory is available and allocates memory **blocks** to hold program segments or data.

memory-resident: 1. A program or data stored permanently in memory as firmware (ROM). 2. A program or data held continually in memory even while not in use. For example, ProDOS is a memory-resident program.

memory segment table: A linked list in memory created by the loader that allows the loader to keep track of the segments that have been loaded into memory.

Monitor: A program built into the firmware of Apple II computers, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine-language level.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block. Opposite of *fixed*. Only **position-independent** program segments can be in movable memory blocks. A block is made movable or fixed through Memory Manager calls.

native development: Using APW on the Apple IIGS computer to develop programs for the Apple IIGS ; as opposed to using a Macintosh computer and MPW to develop programs for the Apple IIGS. See also **cross development**.

native mode: The 16-bit operating state of the 65C816 processor.

object file: The output from an assembler or compiler and the input to the linker. In APW an object file contains both machine-language instructions and instructions for the linker. Compare **load file.**

object module format (OMF): The general format used in object files, library files, and load files.

object segment: A segment in an object file or in a library file.

OMF: See **object module format.**

OMF file: Any file in **object module format.**

op code: See **operation code.**

open: To allow access to a file. A file may not be read from or written to until it is open.

operand: 1. In assembly language, a value used by an instruction or directive as an address or to calculate an address. 2. In object module format, an operation code that is followed by a single value that constitutes part of an expression. The value following the operand opcode is acted on by an **operator.**

operation code: The part of an instruction or command that specifies the operation to be performed. Often called op code. In machine language, the operation code precedes the value to be acted on by the processor. In OMF, operation codes are used to identify types of records and operations in instructions.

operator: In object module format, an operation code that specifies an arithmetic or logical operation in an expression to be performed on one or two variables that precede it. The variables acted on by an operator are identified by **operand** opcodes that precede them.

page: 1. A portion of Apple IIGS memory that is 256 bytes long and begins at an address that is an even multiple of 256. A memory block whose starting address is an even multiple of 256 is said to be page-aligned. 2. An area of main memory containing text or graphical information being displayed on the screen.

parameter: A value passed to or from a command, function, or other routine.

parameter block: A set of contiguous memory locations, set up by a calling program to pass parameters to and receive results from an operating system or shell function that the program calls. Every ProDOS 16 and APW Shell call must include a pointer to a properly constructed parameter block.

partial assembly: A procedure whereby only specific segments of a program are assembled. If you have performed one full assembly followed by one or more partial assemblies on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial compile: A procedure whereby only specific segments of a program are compiled. If you have performed one full assembly followed by one or more partial compiles on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial pathname: A **pathname** that includes the **filename** of the desired file but excludes the volume directory name (and possibly one or more of the subdirectories in the path). It is the part of a pathname following a **prefix;** a prefix and a partial pathname together constitute a **full pathname.** A partial pathname does not begin with a slash because it has no volume directory name.

Pascal string: An ASCII character string preceded by a single byte whose numeric value is the number of characters in the string. Pascal strings are limited to a maximum of 255 characters. Compare **GS/OS string.**

patch: To replace one or more bytes in memory or in a file with other values. The address to which the program must jump to execute a subroutine is *patched* into memory at load time when a file is **relocated.**

pathname: The complete name by which a file is specified. A pathname is a sequence of filenames separated by slashes, starting with the filename of the volume directory file and including every subdirectory file that the operating system must search to locate the file, in descending sequence of the subdirectory hierarchy. A full pathname always begins with a slash (/) to indicate that the first name is a volume directory. See also **full pathname, partial pathname, prefix.**

pathname list: The part of the pathname table that contains the file pathnames.

pathname segment: A segment in a load file that contains the cross-references between load files referenced by number (in the jump table segment) and their pathnames (listed in the file directory). The pathname segment is created by the linker.

pathname table: A table constructed in memory by the loader from all individual pathname segments encountered during loads. It contains the cross-references between load files referenced by number (in the jump table) and their pathnames (listed in the file directory).

PC: See **program counter.**

pipeline: 1. To execute automatically two or more programs in sequence, where the output of the first file is the input to the next file and so on. 2. The entire sequential set of programs executed in this way; a program or file being processed by this sequence of programs is said to be *in the pipeline* or *in the pipe.*

pointer: A memory address at which a particular item of information is located. For example, the 65C816 stack register contains a pointer to the next available location on the stack.

position-independent: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be relocated.

position-independent segment: A load segment that is movable when loaded in memory.

prefix: A portion of a **pathname** starting with a volume name and ending with a subdirectory name. It is the part of a full pathname that precedes a **partial pathname;** a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

prefix number: A code used to represent a particular prefix. Under ProDOS 16, there are eight prefix numbers, each consisting of a numeral followed by a slash: 0/, 1/, ..., 7/.

private code segment: A segment in an object file whose name is available only to other object-code segments within the same object file. The labels within a private code segment are local to that segment.

private data segment: A segment in an object file whose labels are available only to object-code segments in the same object file.

ProDOS: A family of disk operating systems developed for the Apple II family of computers. ProDOS stands for *Professional Disk Operating System* and includes both ProDOS 8 and ProDOS 16.

ProDOS 8: A disk operating system developed for standard Apple II computers. It runs on 6502-series microprocessors. It also runs on the Apple IIGS when the 65C816 processor is in 6502 **emulation mode.**

ProDOS 16: A disk operating system developed for 65C816 **native mode** operation on the Apple IIGS . It is functionally similar to **ProDOS 8** but more powerful.

program counter (PC): A number, usually expressed in hexadecimal format, that indicates the position of a byte in a machine-language program, counting sequentially from the beginning of the program.

purge: To deallocate a memory block temporarily. The Memory Manager purges a block by setting its master pointer to 0. All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purgeable: A memory block attribute indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels,** or priorities for purging; these levels are set by Memory Manager calls.

purge level: An attribute of a memory block that sets its priority for purging. A purge level of 0 means that the block is unpurgeable.

RAM disk: A portion of memory (RAM) that appears to the operating system to be a disk volume. Files in a RAM disk can be accessed much faster than the same files on a floppy disk or hard disk.

record: A component of an object module segment. All OMF file segments are composed of records, some of which are program code and some of which contain cross-reference or relocation information. Each record begins with an operation code that indicates the type of information to follow.

reference: 1. The name of a segment or entry point to a segment; same as **symbolic reference** . 2. To refer to a symbolic reference or to use one in an expression or as an address.

reload segment: A segment that the System Loader loads each time a program is started or restarted, even if the program is restartable and is restarted from memory.

relocate: To modify a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded. Relocation consists of **patching** the proper values into address operands. The loader relocates load segments when it loads them into memory. See also **relocatable code.**

relocatable code: Program code that includes no absolute addresses and can therefore be relocated at load time.

relocatable segment: A segment that can be loaded at any location in memory. A relocatable segment can be static, dynamic, or position independent. A load segment contains a **relocation dictionary** that is used to recalculate the values of location-dependent addresses and operands when the segment is loaded into memory. Compare **absolute segment.**

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image immediately preceding it. When the memory image part of the segment is loaded into memory, the relocation dictionary is processed by the loader to calculate the values of location-dependent addresses and operands. Relocation dictionaries also contain the information necessary to transfer control to external references.

RELOC record: Part of a **relocation dictionary** that contains relocation information for local (within-segment) references.

resolve: To find the segment and offset in a segment at which a symbolic reference is defined. When the linker resolves a reference, it creates an entry in a **relocation dictionary** that allows the loader to **relocate** the reference at load time.

restart: To reactivate a **dormant** program in the computer's memory. The System Loader can restart dormant programs if all their static segments are still in memory. If any critical part of a dormant program has been purged by the Memory Manager, the program must be reloaded from disk instead of restarted.

restartable: Said of a program that reinitializes its variables and makes no assumptions about machine state each time it gains control. Only restartable programs can be resurrected from a **dormant** state in memory.

result: In shell-call descriptions, a numeric quantity, one or more bytes long, that the shell places in the parameter block for the caller to use. A result is an output parameter.

root filename: The filename of an object file minus any filename extensions added by the assembler or compiler. For example, a program that consists of the object files `MYPROG.ROOT`, `MYPROG.A`, and `MYPROG.B` has the root filename `MYPROG`.

run-time library file: A load file containing program segments—each of which can be used in any number of programs—that the system loader loads dynamically when they are needed.

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

segment body: The part of a segment that follows the **segment header** and contains the program code, data, and relocation information for the segment.

segment header: The first part of a program segment, containing such information as the segment name and the length of the segment.

segment kind: See **segment type.**

segment number: A number corresponding to the relative position of the segment in a file, starting with 1.

segment type: A classification of a segment based on its purpose, contents, and internal structure, as defined in the object module format. The segment type is specified by the KIND field in the segment header.

shell: A program that provides an operating environment for other programs and that is not removed from memory when the those programs are running. For example, the APW Shell provides a command processor interface between the user and the other components of APW, and it remains in memory when APW utility programs are running.

shell call: A request from a program to the APW Shell to perform a specific function.

shell-call block: A set of instructions and directives used to make a shell call from an assembly-language program.

shell command line: The line on the screen where the number-sign prompt (#) appears when you are in the APW Shell. When you enter a command, the characters you type appear to the right of the prompt on the command line.

shell load file: A load file designed to be run under a shell program. Shell load files are ProDOS 16 file type $B5.

65C816: The microprocessor used in the Apple IIGS.

source file: An ASCII file consisting of instructions written in a particular language, such as C or assembly language. An assembler or compiler converts source files into **object files.**

special memory: Memory banks $00, $01, $E0, and $E1. These are the memory banks used by programs running under ProDOS 8 in standard Apple II emulation mode.

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. The term *stack* usually refers to the top portion of the **direct-page/stack space;** the top of this stack is pointed to by the 65C816 stack register.

stack pointer: The contents of the 65C816 stack register, consisting of a memory address pointing to the next available location on the 65C816 **stack.**

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

standard input: The default file or device (such as the keyboard) from which input is taken. If your program uses Text Tool Set calls or APW macros and libraries to get input, standard input is used.

standard output: The default file or device (such as the screen) to which output is sent. If your program uses Text Tool Set calls or APW macros and libraries to control output, standard output is used.

startup prefix: The volume name of the disk from which the computer was started up.

static segment: A segment that is loaded at program start time and is not unloaded or moved during execution. Compare **dynamic segment.**

string: An item of information consisting of a sequence of text characters (a **character** string) or a sequence of bits or bytes.

subdirectory: A directory within a directory; a file (other than the volume directory) that contains the names and locations of other files. Every ProDOS 16 directory file is either a volume directory or a subdirectory.

symbol: A character or string of characters that represents an address or numeric value; a symbolic reference or a variable.

symbolic reference: A name or label that is used to refer to a location in a program, such as the name of a subroutine. When a program is linked, all symbolic references are **resolved;** when the program is loaded, actual memory addresses are **patched** into the program to replace the symbolic references.

symbol table: A table of symbolic references created by the linker when it links a program. The linker uses the symbol table to keep track of which symbols have been resolved. At the conclusion of a link, you can have the linker print out the symbol table.

System Loader: The program that relocates load segments and loads them into Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

System Monitor: See **Monitor.**

system program: 1. A software component of a computer system that supports application programs by managing system resources such as memory and I/O devices. Also called system software. 2. Under ProDOS 8, a stand-alone and potentially self-starting application. A ProDOS 8 system program is of file type $FF; if it is self-starting, its filename has the extension `.SYSTEM`.

text-file format: The Apple IIGS standard format for text files and program source files.

token: The smallest unit of information processed by a compiler or assembler. In C, for example, a function name and a left bracket ({) are tokens.

toolbox: A collection of built-in routines on the Apple IIGS that programs can call to perform many commonly needed functions. Functions within the toolbox are grouped into **tool sets.**

tool set: A related group of (usually firmware) routines available to applications and system software that perform necessary functions or provide programming convenience. The Memory Manager, the System Loader, and QuickDraw II are tool sets.

top of form: The position on the paper in the printer to which the printer scrolls when it receives a form feed (Control-L) command.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager either to **purge** or to **dispose** of the memory block in which the code segment resides. The loader then modifies the memory segment table to reflect the fact that the segment is no longer in memory.

User ID: An identification number that specifies the owner of every memory block allocated by the Memory Manager. User IDs are assigned by the User ID Manager.

utility: In general, an application program that performs a relatively simple function or set of functions such as copying or deleting files. An APW utility is a program that runs under the APW Shell and performs a function not handled by the shell itself. MakeLib is an example of an APW utility.

value: In shell-call descriptions, a numeric quantity, one or more bytes long, that the caller passes to the shell through the parameter block. A value is an input parameter.

volume: An entity that stores data; the source or destination of information. A volume has a name and a volume directory with the same name. Volumes typically reside in devices; a device such as a floppy disk drive may contain one of any number of volumes (on disks).

volume directory: The main directory file of a volume. It contains the names and locations of other files on the volume, any of which may themselves be directory files (called **subdirectories**). The name of the volume directory is the name of the volume. The pathname of every file on the volume starts with the volume directory name.

wildcard character: A character that can be used to represent a sequence of characters in a pathname. In APW, the equal sign (=) and the question mark (?) can be used as wildcard characters.

word: A group of bits that is treated as a unit. For the Apple IIGS, a word is 16 bits (2 bytes) long.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

Index

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft Word software. Proof and final pages were created on Apple LaserWriter® printers. Line art was created using Adobe Illustrator. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of ITC Garamond®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

11/10/89

APPLE SOFTWARE LICENSE

PLEASE READ THIS DOCUMENT CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration and system software (the "Software") and related documentation are licensed to you by Apple. You own the disk on which the Software is recorded but Apple retains title to the Software. This License allows you to use the Software on a single Apple computer and make one copy of the Software in machine-readable form for backup purposes only. You must reproduce on such copy the Apple copyright notice and any other proprietary legends that were on the original copy of the Software. You may also transfer the Software, the backup copy of the Software, the related documentation and a copy of this License to another party provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Software contains copyrighted material, trade secrets, and other proprietary information and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease, loan, sell, distribute or create derivative works based upon the Software in whole or in part. You may not electronically transfer the Software from one computer to another over a network.

3. Support. You acknowledge and agree that Apple may not offer any technical support in the use of the Software.

4. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Software and all copies thereof. This License will terminate immediately without notice from Apple if you fail to comply with any provision of this License. Upon termination you must destroy the Software and all copies thereof.

5. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from Apple, nor the direct product thereof, will be exported outside the United States except as authorized in advance by Apple, in writing, and as permitted by the laws and regulations of the United States.

6. Government End Users.
(a) If this Software is acquired by or on behalf of a unit or agency of the United States Government this provision applies. This Software: (i) was developed at private expense, and no part of it was developed with government funds; (ii) is a trade secret of Apple for all purposes of the Freedom of Information Act; (iii) is "commercial computer software" subject to limited utilization as provided in the contract between the vendor and the governmental entity; and (iv) in all respects is proprietary data belonging solely to Apple.
(b) For units of the Department of Defense (DOD), this Software is sold only with "Restricted Rights" as that term is defined in the DOD Supplement to the Federal Acquisition Regulations ("DFARS") 52.227-7013 (c) (1) (ii) and use, duplication or disclosure is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013. Manufacturer: Apple Computer, Inc., 20525 Mariani Avenue, Cupertino, California 95014.
(c) If this Software was acquired under a GSA Schedule, the Government has agreed: (i) to refrain from changing or removing any insignia or lettering from the Software that is provided or from producing copies of manuals or disks (except one copy for backup purposes); (ii) title to and ownership of this Software and any reproductions thereof shall remain with Apple; (iii) use of this Software shall be limited to the facility for which it is acquired; and (iv) if use of the Software is discontinued at the installation specified in the purchase/delivery order and the Government desires to use it at another location, it may do so by giving prior written notice to Apple, specifying the type of computer and new location site.
(d) Government personnel using this Apple Software, other than under a DOD contract or GSA Schedule, are hereby on notice that use of this Software is subject to restrictions which are the same as, or similar to, those specified above.

7. Limited Warranty on Media. Apple warrants the disks on which the Software is recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Apple's entire liability and your exclusive remedy will be replacement of the disk not meeting Apple's limited warranty and which is returned to Apple or an Apple authorized representative with a copy of the receipt. Apple will have no responsibility to replace a disk damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKS, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

8. Disclaimer of Warranty on Software. You expressly acknowledge and agree that use of the Software is at your sole risk. The Software and related documentation are provided "AS IS" and without warranty of any kind and Apple EXPRESSLY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. APPLE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE WILL BE CORRECTED. FURTHERMORE, APPLE DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

9. Limitation of Liability. UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL APPLE BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.
In no event shall Apple's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Software.

10. Controlling Law and Severability. This License shall be governed by and construed in accordance with the laws of the State of California, except that body of California law concerning conflicts of law. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

11. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Software and related documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter.

APW™ Tools & Interfaces v.1.1

This package contains

1	Manual	*APW Tools Reference*
1	Set of Release Notes	*APW Tools & Interfaces Release Notes*
3	Disks:	*APW Tools 1, Disk 1* *APW Tools 2, Disk 2* *APW Interfaces, Disk 3*
1	1-inch binder	
1	1-inch binder cover	
1	1-inch spine identification	

If you have any questions, please call

1-800-282-2732 (U.S.)
1-408-562-3910 (International)
1-800-637-0029 (Canada)