high-level descriptions depend entirely upon the amount of work a developer puts into factoring the application.

To write scripts, users must have AppleScript-aware applications and must know the scripting commands for those applications.

To record scripts, users must have applications that can convert actions into commands as users perform them. Moreover, they'll probably have to look at the recorded script to determine whether the application recorded all their actions.

As shipped, the Macintosh Quadra 840AV and Macintosh Centris 660AV are not supplied with scriptable or recordable applications. Some third-party applications are currently available. The Apple Scriptable Text Editor is recordable, and Excel and FileMaker® Pro are scriptable. The AppleScript system will become more useful as more and more applications support it.

QuicKeys

QuicKeys is good for recording low-level events and thus for handling simple interactions with most applications. It suffers many of the same problems as the original MacroMaker from Apple:

- It usually just replays users actions exactly (users see the interface flying by as if they were doing it).
- Since it's just replaying low-level events, many of its commands break down if the position of the underlying object changes.
- It lacks the full expressive qualities of AppleScript; it's really its own language, but one lacking sophisticated conditionals, loops, procedure calls, and so on.

To write scripts, users must know the QuicKeys language supported by the OSA component so that they can change volatile commands such as Drag and Click At to more stable commands where possible. The Macintosh Quadra 840AV and Macintosh Centris 660AV system software supports QuicKeys, so users can create new macros. CE Software also provides a set of example macros written with QuicKeys. The QuicKeys scripting language may not include the full power of the "normal" QuicKeys system; for example, QuicKeys extensions, which circumvent the interface and set values directly, may not be supported.

User Requirements

As with AppleScript, users of the Speech Macro Editor will have to be fairly sophisticated to be able to write and edit scripts; the majority of users will have to use prewritten scripts. Recording should allow less experienced users to create voice macros, but recording must be viewed as a shortcut for typing a new script; further editing will probably be required.

Since the SME isn't trying to reproduce the full suite of scripting tools being developed for AppleScript (no debugging, no access to help on scripting commands, and so on), its users will need to know how to find the answers to these questions:

- Is the application AppleScript aware? Is it also recordable? What scripting commands does it provide?
- What scripting commands does the script system provide?

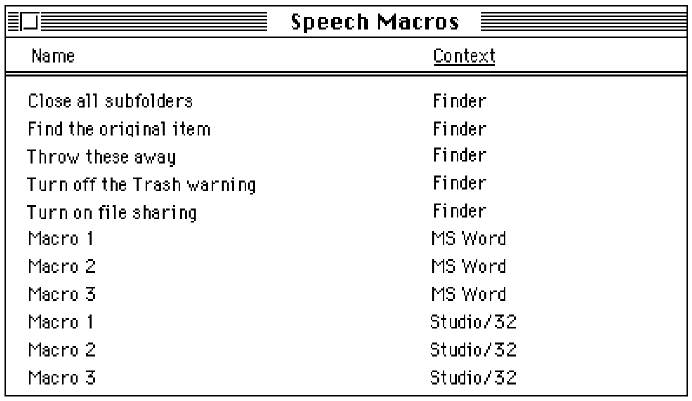
The script editor provided with AppleScript has facilities for developing more complicated scripts than are possible with SME and includes complete debugging and error reporting features.

Using the Speech Macro Editor

The Speech Macro Editor is an application in the Extras folder on the hard disk. Here's how to start it:

1. **Open the Extras folder.**
2. **Open the Speech Macro Editor by double-clicking its icon.**
   When the Speech Macro Editor starts up, by default it automatically opens the Speech Macros document from the Extensions folder. The document window for Speech Macros lists all the speech macros it contains. Initially, the SME does not select an item in the list. A typical Speech Macros document window is shown in Figure 7-5.

Figure 7-5 Typical Speech Macro document window



Speech Macros

Name	Context
Close all subfolders	Finder
Find the original item	Finder
Throw these away	Finder
Turn off the Trash warning	Finder
Turn on file sharing	Finder
Macro 1	MS Word
Macro 2	MS Word
Macro 3	MS Word
Macro 1	Studio/32
Macro 2	Studio/32
Macro 3	Studio/32

IMPORTANT

Speech macros can exist in any SME document. The Speech Macros document is just the default document shipped with the computer. An SME document must be in the Extensions folder or the System Folder for it to become part of the current grammar. ▲

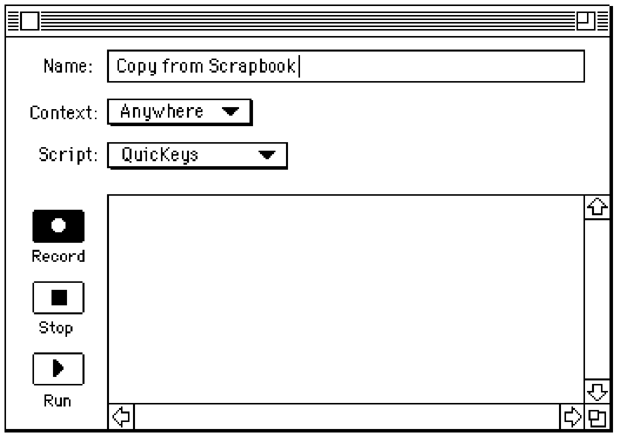
Recording a New Macro

To record a new macro, follow the steps below. As an example, we'll create a speech macro for copying the first item in the Scrapbook.

1. **Choose Create New Macro from the Macro menu.**
   A blank macro window appears onscreen. The insertion point is set in the Name field.
2. **Type the phrase that you want to speak.**
   It's best for the name to be a phrase rather than a single word. Recognition works faster and more accurately if the differences among names are more pronounced. Also note that, unlike most speech recognition technologies, Casper can recognize continuous speech.
3. **From the Context pop-up menu, choose the context in which you'll be able to speak the new macro.**
   In this case, you want to have this macro available at all times, so change the context to Anywhere.

   Figure 7-6 shows these three steps performed in a New Macro window.

Figure 7-6 Typical New Macro window


Name: Copy from Scrapbook
Context: Anywhere
Script: QuicKeys

Note
If users open the SME on a system that's not running AppleScript, they can only edit scripts. The following actions with the Record, Stop, and Play buttons and the Script pop-up menu will not be available; these items will be dimmed. ◆

4. **Choose a script language for recording the macro.**
   The choice of script system determines what applications (and events in those applications) are recordable. Users need to understand the benefits and limitations of a particular choice here. Since the system and Finder won't support AppleScript, change the script system to QuicKeys.

5. **Click the Record button.**
   The button "locks" in place, and a small recording icon blinks over the Apple menu while the user is in record mode. The icon that appears depends on the script system (AppleScript displays a small cassette, QuicKeys a small microphone).
6. **Switch to the application and perform the desired actions.**
   In this example, pull down the Apple menu, choose Scrapbook, choose Copy, and close the window.

IMPORTANT
Script systems may handle the posting of commands differently. For example, AppleScript sends commands to the SME after each one occurs. QuicKeys returns an entire script after the user stops recording. ▲

7. **Return to the SME by clicking an open SME window or by choosing SME from the Application menu, then click Stop.**
   Wait until the recording icon stops blinking. The script appears in the script area of the window.
8. **Click the close box to save the new macro.**

Renaming a Macro

To change the name of a macro, follow these steps:

1. **Select the macro you want to edit and choose Edit Macro from the Macro menu, or double-click the macro name.**
2. **A macro window appears.**
3. **Type the new name in the text field.**
4. **Click the close box.**

The window disappears, the name and context items in the list of macros change, and the list is sorted.

Saving Macro Changes

At any point, the user can save changes to an SME document by choosing one of the Save commands in the File menu. These commands are available from the main document window or any of the macro windows. If a save command is chosen when a macro window is active, the SME saves the entire document in which that macro resides.

The SME displays the standard Save Changes dialog box if the user closes a document window without having first saved changes.

Loading Macros

Casper loads macros at the following times:

- When users turn speech on from the Speech Setup control panel, Casper loads rules from any SME document that is in the Extensions folder or the System Folder on the startup disk. Casper also loads any speech rules documents found in either of these two locations.

- When users make changes to any of the SME documents loaded from the Extensions folder or the System Folder, Casper reloads the changed SME documents when the user saves the document. Casper should acknowledge that it's reloading the macros (so that users know it's happening) by posting a message to the feedback window.
- Users who place new SME documents or new speech rules documents in the Extensions folder or the System Folder must stop and restart Casper to load the new documents. Casper keeps track only of the documents it loaded when starting up.
- If an application contains speech rules in its resource fork, they will be loaded when the application is launched. For further information, see "Speech Rules Files," in Chapter 8.

Built-in Speech Rules and Grammar

Speech rules are structures used to define how words can be strung together for speech recognition. They are discussed in detail in Chapter 8, "Speech Rules." Since many commands (such as those that choose menu items) are required in all applications, a standard set of rules is built into Casper to provide a robust set of standard commands. Many menu functions are common across a wide variety of applications, and most applications will also use Finder-type commands to access the Apple menu items.

In English there are grammar rules that define the noun-verb-subject sequence. A similar sequence must be identified explicitly for the speech recognizer. For example:

"Open Chooser"

"Open the Chooser"

"Open menu item Chooser"

could all be used to open the Chooser control panel. All of the acceptable word strings must be defined in order for the Speech Monitor to select the correct command. If the user says "Chooser open," the rules in this example will not recognize that statement as an acceptable command. If the word string "Chooser open" is added to the rules, then Casper will respond with an acceptable command.

In the Macintosh Quadra 840AV and Macintosh Centris 660AV speech recognition software, all menu items and dialog box buttons are controllable by speech. Use the following command forms:

- "Open *AppleMenuItem*," where *AppleMenuItem* is any item within the Apple menu—for example, "Open Alarm Clock"
- "Switch to *ProcessMenuItem*," where *ProcessMenuItem* is the name of any process—for example, "Switch to Finder"

A new Speakable Items folder exists in the Apple Menu Items folder in the Macintosh Quadra 840AV and Macintosh Centris 660AV system software. Any item or alias to an item within it will be speakable. Some aliases to standard items are installed automatically, such as Open System Folder. The phrase to speak these items is the same as the name given to the item. AppleScript (or QuickKeys) items can be placed in the

Speakable Items folder as well. This folder is not dynamically updated at present, so speech must be shut down and restarted to load any new items placed in it.

Here are some sample Finder phrases:

- "Hello"
- "What time is it"
- "What day is it"
- "close window"
- "close all windows" (available only when the Finder is frontmost)
- "zoom window"
- "is file sharing on"
- "start file sharing"
- "stop file sharing"
- "shut speech off"

Here are sample printing phrases:

- "Print from *n* to *n*"
- "Print from page *n* to *m*"
- "Print *n* copies"
- "Print page *n*"
- "Print page *n* to *m*," where *n* and *m* are numbers from 1 through 99. (This works in all applications that use Cmd-P to print.)

Performance

Casper's speech recognition goal is a minimum in-grammar error rate for a typical task in a low-noise environment. *In-grammar error rate* is the number of times the speech recognition software does not respond as intended when a defined command is spoken. All of the variables listed in the next section affect the ability of the system to recognize speech.

Real-Time Response

Response time is a function of several variables:

- *Clear pronunciation.* The search tends to be faster if the utterance is spoken clearly in North American English.
- *Grammar complexity.* The higher the number of possible word phrases in the speech rules, the longer the search and the higher the error rate.

- *Word complexity.* The choice of words can affect the duration of the search; similar-sounding words are harder to distinguish.
- *Extraneous noise.* Additional noise affects the quality of the input and potentially increases the search time as the noise is increased.
- *Room acoustics.* Bad acoustics may degrade system performance, including response time from one acoustic environment to the next.
- *Environmental adaptation.* This algorithm adapts to changing room conditions and background noise—after every five utterances, the environmental adaptation is updated.

Types of Errors

Taken as a group, the rules for a specific application form the grammar for that application. The recognition search returns the best match from the available grammar.

One type of error occurs when the search results are too uncertain, in which case the speech recognizer rejects the sentence as unrecognizable. Another type of error occurs when an in-grammar match is selected to an incorrect sentence and the speech recognizer responds although no command was given.

Apple's naming conventions for speech recognition responses, both correct and erroneous, are shown in Table 7-1.

Table 7-1 Grammatical naming conventions

In-grammar	Out-of-grammar
Correct recognition	Correct rejection
Incorrect recognition	Correct detection of new word
Incorrect rejection (correct words/grammar not identified)	Incorrect recognition (through substitution or insertion)

For the in-grammar case the first item is the nonerror response. For the out-of-grammar case, the first two items are the nonerror responses. There are several reasons why a phrase might not be properly recognized—for example, unclear pronunciation, background noise, or bad room acoustics.

Acceptable Limits or Constraints

The system is constrained to North American adult English used in grammatically simple sentences. Note that this also implies a limited vocabulary.

The system accuracy will typically drop during changing environmental conditions. Adaptation takes approximately five utterances.

The speech recognition software currently understands only clearly spoken English words. The user must speak in well-defined sounds for all words and sentences.

Note

The speech recognizer cannot recognize most nonstandard English words. A nonstandard word could be any word that is formed as a result of concatenating words, using abbreviations, or other shortcuts, which typically result in many ways to say the same word. The current recognizer accepts only one pronunciation for a word, with only small variations from that pronunciation. As an example, a made-up word used as a filename may not be recognizable. Abbreviated forms of words (such as *MPW*) are not typically recognizable as words. ◆

C H A P T E R 8

Speech Rules

This chapter describes how the speech recognition software in the Macintosh Quadra 840AV and Macintosh Centris 660AV uses speech rules to interpret and respond to the user's utterances. It also describes the `CompileRules` tool available with the Macintosh Programmer's Workshop (MPW), which compiles the rule source files into resources. Read Chapter 7, "Introduction to Speech Recognition," before reading this chapter.

Overview

At the heart of Apple's speech recognition system is a data structure called a **speech rule.** A speech rule is a word or a sentence that is defined to perform an action within the current computer environment. Each rule performs a unique function depending on the words spoken. An application's grammar is derived from the set of speech rules and the current context.

A rule can include variables used in locations that can be more than a single word. A word within a sentence that can be substituted with another word is called a *category.* A category can be an individual word or another category. When it is a predefined category, the acceptable words are listed in that category. For example, `<number>` can be a number from 1 to 9. A `<ten>` is defined as a number in the tens location, plus a `<number>` or a 0. A `<hundred>` is defined as a number in the hundreds location, plus a `<ten>` or a 0, plus a `<number>` or a 0. This process can be continued to make up any arbitrarily large number. In each case the category is made up of previously defined categories, except for `<number>`, which is a list of individual words.

In its simplest form, a speech rule maps some spoken utterance to a value or an action. When the speech recognition software detects that the user has uttered the phrase, the corresponding value is computed or an action is performed. Here is an example of a simple speech rule:

```
%rule
  bold
%action
  tell application "MyApp"
    set style of selection to bold
  end
%end
```

The effect of this rule is that whenever the user says "bold," the application named MyApp changes the selected text to bold. The `%rule` clause signals the beginning of a new speech rule; the line containing `bold` contains the phrase that should be recognized; the `%action` clause signals the beginning of the action part of the rule; the lines from `tell` to `end` contain the script that should be executed when the rule's phrase is recognized; and the `%end` clause signals the end of the rule.

A speech rule can have any number of phrases—for example:

```
%rule
  bold
  change to bold
  bold this
  make it bold
%action
  tell application "MyApp"
    set style of selection to bold
  end
%end
```

This is a valid speech rule, the effect of which would be to cause MyApp to change the style of the selection when any of the specified phrases is recognized.

Note
Avoid using the same speech string twice. If two speech commands are identical, Casper will use only the first macro it finds. The second macro will be ignored. ◆

One problem with the foregoing rule is that it causes the MyApp application to change styles, no matter what application is currently active. So, if MacWrite® is the active application and the user says"bold," MyApp will change styles (or worse, if MyApp is not running, it will be launched and then it will change styles). One way around this problem is to specify that the rule should be active only when MyApp is the active application:

```
%rule
  bold
%context application "MyApp"
%action
  tell application "MyApp"
    set style of selection to bold
  end
%end
```

In this case, the speech recognizer listens for the phrase *bold* only when the MyApp application is active. Spoken commands that make sense only when a particular application or window is active can be marked in this fashion.

As you begin to build up larger vocabularies for your computer, you will want to avoid having to enumerate every utterance that the system should recognize. Speech rules can be used to construct entire grammars of what the user can say. For example, let's say you want to define a rule that allows the user to change the selection to any style, without having to list every utterance separately:

```
%define style
  bold
  italic
  underline
%end
```

The foregoing is called a *category rule.* It is similar to the command rule in that it defines a set of phrases that the user might say. However, it does not specify an action. Instead, this rule defines a token, `<style>`, which can be used in other rules instead of directly enumerating the category's phrases.

```
%rule
  <style>
  change to <style>
%action
  tell application "MyApp"
    set style of selection to ...
  end
%end
```

Defining the rules this way lets you specify the syntax of the style command itself separately from the syntax for the style names. Other commands can also refer to the `<style>` category.

Note that in the action for the preceding rule, an ellipsis (`...`) was used in place of the actual style. The initial example used a constant style, but in this case, the actual style depends on which style the user says. There is a way to pass that information from the category rule to the command rule, by attaching a script fragment to each phrase. The script fragment returns a value representing the meaning of that phrase—for example:

```
%define style
  plain       ; {meaning: plain}
  bold        ; {meaning: bold}
  italic      ; {meaning: italic}
  .
  .
  .
%end
```

For each phrase, the text to the left of the semicolon defines what the user can say, and the text to the right of the semicolon is the AppleScript expression. This technique allows the rule writer to assign a meaning to each of the possible phrases that the user may utter. The rule that uses this meaning, then, looks like this:

```
%rule
  <s:style>
  change to <s:style>
%action
  tell application "MyApp"
    set style of selection to meaning of s
  end
%end
```

Every reference to a category whose value is needed should be preceded with a variable name. When the subsequent script is executed, the variable will be bound to the value returned by the category rule. For example, if the user says "change to bold," the style category matches the word *bold,* producing as its value the Apple event record `{meaning: bold}`. The above command rule then matches the entire utterance, executing its script with the variable `s` bound to the value produced by the corresponding category rule. Finally, the expression `meaning of s` retrieves the style constant from the meaning record.

Note the use of the `meaning` property to access the value computed by the category. Whenever a phrase's script is evaluated, the value returned is always coerced into an Apple event record. In the example just given, a record was used as the value of each of the category's phrases. Since it was already a record, it was used as is. If the value is any other data type, it is stored as the `meaning` property of an Apple event record, and the record is used as the returned value. For example, the following two phrases are equivalent:

```
one         ; 1
one         ; {meaning: 1}
```

Thus, when accessing the value bound to a variable in a category reference, it is usually necessary to get its `meaning` property.

Here is another example of using categories to define a grammar for numeric digits:

```
%define digit
  one       ; 1
  two       ; 2
  three     ; 3
  .
  .
  .
  nine      ; 9
%end
```

This category defines a simple grammar that will recognize a single spoken digit and return the numeric value of that digit. A script can access the value returned by a category by preceding the category reference with a variable name:

```
%rule
  what is <n:digit> plus <m:digit>
%action
  .
  .
  .
  set x to (meaning of n) + (meaning of m)
  .
  .
  .
%end
```

Using the techniques described so far, you can define a category for recognizing whole numbers less than 100. First, define a rule to recognize the tens words:

```
%define tens
  twenty    ; 20
  thirty    ; 30
  .
  .
  .
 ninety     ; 90
%end
```

This rule is exactly like the definition of digit just given. Next, define a rule to recognize the teens words:

```
%define teens
  ten       ; 10
  eleven    ; 11
  .
  .
  .
 nineteen   ; 19
%end
```

Finally, define the rule that combines all the parts and returns the correct value for multiword phrases:

```
%define uhundred
  <n:digit>    ; n
  <n:teens>    ; n
  <n:tens>     ; n
  <n:tens> <d:digit>; (meaning of n) + (meaning of d)
%end
```

For the first three phrases, the value returned is simply the value recognized by the subordinate category. For example, a single digit is a valid number to be recognized by this rule, and its value is simply the value returned by the digit rule. In the case of the fourth phrase, you want to recognize spoken numbers such as *twenty-five.* The script for this phrase essentially computes the meaning of speaking these two words in sequence. This is a trivial example, but the general mechanism is a powerful one that can be used to associate meaning with a wide variety of spoken commands.

Note that in the fourth phrase above you did not write `n + d` as the script. This is because the values bound to `n` and `d` are Apple event records, not numbers. In the previous cases, you were simply passing on the values, so you could leave them as records; but when you want to do arithmetic, you need to access the meaning explicitly. An equivalent, if more verbose, expression is the following:

```
<n:tens> <d:digit>; {meaning: (meaning of n) + (meaning of d)}
```

Sometimes the meaning of an utterance resides simply in the words spoken. For example, consider a phone-dialing application in which you want to acknowledge that the spoken command is being carried out:

```
%define name
  John Doe     ; {phone: "555-7442"}
  Bob Strong   ; {phone: "555-3295"}
  .
  .
  .
%end

%rule
  call <n:name>
%action
  dial (phone of n)
  acknowledge saying "Now dialing"
%end
```

In this example, each person's phone number is attached as part of the `meaning` structure. Notice that a different property is used. This works fine; you can use any properties that you want as long as you return a record. With speech recognition, however, there is always the possibility of a mistaken recognition. It would be better to tell the user the name of the person that the system is dialing, so that if it fails to recognize correctly, the user has a chance to hang up before the call goes through. You could attach the person's name to the meaning structure:

```
John Doe       ; {phone: "555-7442", name: "John Doe"}
```

However, this would be redundant. As a convenience, the Speech Monitor always adds an utterance property to the value generated by a phrase script. The value assigned to this property is a string containing the words that were matched by the category rule. So, you can rewrite the action script of your phone-dialing rule as follows:

```
%rule
  call <n:name>
%action
  dial (phone of n)
  acknowledge saying "Now dialing " & (utterance of n)
%end
```

Speech Rules Files

Speech rules are data structures that determine how spoken commands are interpreted. They are stored as resources either in **speech rules files** or in the resource fork of an application. When speech is started, the System Folder and the Extensions folder are scanned for speech rules files. Any speech rules files found in these two locations are scanned, and the rules in those files become active and are used for spoken command recognition. Rules resources present in an application are loaded when an application is launched.

There are actually two different file types used for speech rules files: one for speech rules files proper and one for macro files. A **speech macro** is a simplified kind of speech rule that can be created with the application Speech Macro Editor. Internally, these files have identical formats, and the speech recognition system does not distinguish between the two.

The `CompileRules` MPW tool is used to generate rules files or rule resources from text files. The syntax to invoke `CompileRules` is

`CompileRules [` *options* `]` *input-file* …

Any number of input files may be specified. The valid options are as follows:

`-b`	The `-b` option causes all scripts in the file to be precompiled and stored in their binary format. If this option is not specified, the rules will be compiled by the Speech Monitor at run time, on demand.
`-base` *integer*	The `-base` option causes rule resources to be numbered beginning at the specified ID. If this option is not specified, resource IDs begin at 0. This option is useful in order to prevent resource ID collision when the rule resources are going to be installed in an application file. The rule compiler currently generates resources of types `'rule'`, `'glob'`, and `'scpt'`.
`-c` *creator*	The `-c` option may be used to specify a creator for the output file. If not specified, the creator is set to `'????'`.
`-category` *category-name*	This option is used in conjunction with the `-generate` option to cause phrases generated by a particular category to be generated. If this option is not used, then phrases are generated from the set of all possible commands.
`-generate all` `-generate` *integer*	The `-generate` option is used to print to standard output a list of utterances generated by the grammar defined by the input files. If `all` is specified, then all possible utterances are listed. Otherwise, the number of utterances specified by *integer* is printed out, generated at random according to the method defined by the `-method` option.
`-method total | phrase | mixed`	This option is used to specify the method of generating random utterances. The `total` method generates each utterance from the total set of possible utterances with equal probability. The `phrase` method generates utterances such that each phrase from a rule is equally likely to be chosen. The `mixed` method uses the `phrase` method to choose a top-level command at random and then uses the `total` method to expand any categories contained in that utterance. The default method is `mixed`.
`-o` *output-file*	The output file is designated with the `-o` option. If this option is not specified, the input files are read and checked for correct syntax, but no output is created.
`-p`	The `-p` option causes informative progress messages to be written to standard output as the rules are compiled.
`-unique`	The `-unique` option is used in conjunction with the `-generate` option in order to force all generated utterances to be unique.

The `CompileRules` tool must be run on a system that contains the scripting systems to be compiled (that is, AppleScript). Errors in the speech rules file will result in messages being written to standard output with the error and line of the file where each error occurred. The format for the text file is given in the next section.

There are two kinds of speech rules: command rules and category rules. Command rules are like speech macros; in fact, speech macros are instances of command rules. They cause a specified action to occur when the Speech Monitor hears a particular phrase. Every command rule has the following parts:

- A list of *phrases,* each of which defines a phrase that the user may utter to cause the action below to be carried out. Each phrase has an optional script that can define a semantic value to be associated with the phrase and can be accessed in the action's script. The phrase itself consists of a list of tokens that are references to either words or categories. Words are like terminals in a grammar, and categories are like nonterminals.
- An optional *context* that defines when the command rule is active. If the context is empty, the rule is always active.
- An optional *condition,* which is a script that determines whether or not the rule should be considered active. This is like the context, except that it is evaluated rather than constant, and it is evaluated after the utterance has been recognized. It is useful for resolving ambiguities when more than one rule has matched the user's utterance.
- An optional Boolean *acknowledge flag* that causes the command to be acknowledged in the standard (nonverbal) way. If it is desired to provide verbal acknowledgment, then the flag should be `false`, and the acknowledge AppleScript command should be used. Normally this flag is used for very short commands, such as menu items, dialog box buttons, and so on.
- An optional *target clause,* which indicates a default target for the condition and action scripts. If no target is specified, the default target for the scripts is the Speech Monitor itself. The target can be changed in the script by using the `tell` clause of AppleScript.
- An *action,* which is a script that is executed when one of the rule's phrases has been uttered by the user and recognized. The action's script may refer to variable bindings created by any of the phrase's scripts that have matched the user's utterance. The default target is `Speech Monitor`, so that any scripts sent to the Speech Monitor can be used without the standard `tell application` block.
- An optional *index clause,* which consists of a list of index terms used by the help system. It is recommended that this clause always be provided.
- An optional *description clause,* which is a textual description of the effect of the command. This field is used by the help system.

Category rules are used to create subgrammars, which may be used by command rules and other category rules. Each category rule defines a set of phrases that may be recognized as part of an utterance. Category rules do not have actions and are relevant only when they are referred to by command rules (directly or indirectly). Every category rule has a name, which is used to refer to the category in other rules.

In addition, there are two subtypes of category rules: internal and external. *Internal* categories are rules that have their phrases listed explicitly in the rule. The list of phrases has exactly the same form and function as in command rules. The variable bindings assigned by the phrases' scripts can be used by any rule that refers to the category. *External* categories are rules that have their phrases computed by a script. The value returned by the script should be either a list of strings or a single string of phrases separated by newline characters. External categories have an additional option called *dynamic,* which determines when the script is evaluated. Nondynamic external categories have their scripts evaluated once, when speech starts up (or if there is a context, each time the context makes a transition from not active to active). Dynamic external categories have their scripts evaluated every time an utterance is detected and their context is active (if the rule has a context).

When a speech rules file is saved in the Macintosh System Folder or Extensions folder and speech is on, a `reload rules` command needs to be sent to the Speech Monitor or speech must be shut off and restarted to load the changes. The Speech Macro Editor does this, for example, when you save a macro file that is stored in one of these two places. You can also use AppleScript to make this happen. The following script causes the named speech rules file to be loaded or reloaded:

```
tell application "Speech Monitor"
  reload rules (alias "Hard Disk:Rules:My Rule File")
end
```

Speech Rules File Syntax

Speech rules are stored as resources either in speech rules files stored in the System Folder or in the resource fork of an application. The resources in these files are created from a text form using an MPW tool called `CompileRules`. The syntax for the text form of speech rules files is described in the next section. These notations are used:

- Anything appearing between brackets is optional. The brackets themselves do not appear in the source file.
- Ellipsis points (three periods) are used to denote zero or more repetitions of the preceding element.
- A name in italics is a reference to another syntactic element, defined elsewhere. Any other item not in italics should appear in the source file exactly as indicated.

Command Rules

The general form of a rule is this:

```
%rule
phrase
   .
   .
   .
[%context [ context ... ] ]
[%acknowledge]
[%target [ signature ] ]
```

```
[%condition [ script-type ]
script]
%action [ script-type ]
script
[%index term [, term ... ] ]
[%description
description ]
%end
```

The `%target` statement is used to set the default target for all scripts that occur in the rule. This includes any phrase scripts, the condition script, and the action script. If no target is specified, the default target is used, and if no default has been set, then the default target is the Speech Monitor. Note that the script's target affects what terminology is available to the script. For example, if the values computed by phrase scripts refer to application-specific terminology, then either the script must explicitly use a `tell` statement or the rule must have a target specified. If no signature is specified in the target statement, the Speech Monitor becomes the default target for the rule's scripts.

If the `%acknowledge` statement is present, a standard (nonverbal) acknowledgment is executed according to the settings in the Speech Setup control panel.

The `%condition` statement specifies a script that evaluates to either `true` or `false` to determine whether to execute the action associated with this rule. Unlike `context`, this check is done after the rule has matched and thus is useful for resolving ambiguity when more than one rule matches the user's utterance.

The `%index` statement is used by the help system to allow the user to find speech commands that are relevant to a particular topic. This helps the user know what to say at any given point. As an example, a rule having to do with sending faxes may have the following index clause:

```
%rule
fax this to <person>
   .
   .
   .
%index email, fax, send
%end
```

The `%description` statement should be an English description (on as many lines as needed) that the help system can use to explain the effect of the command to the user.

The order of the `%` statements is flexible.

Phrases and AppleScript Clauses

A phrase defines a sequence (or set of sequences) of words that may be uttered by a user and recognized by the speech system. Syntactically, a phrase is specified as a sequence of space-separated tokens, each of which is either a word or a category reference. Words that contain nonalphabetic characters must be enclosed in quotation marks—for example:

```
"don't" save
```

A category reference consists of an opening angle bracket (<), followed by an optional label and a colon, followed by a category name, followed by a closing angle bracket (>)—for example:

```
<n:number>
```

If the value returned by the number category is not needed in the clause's script, then the label and its colon may be omitted:

```
<number>
```

If the label is used, it can be referred to as a property in the value script attached to the phrase (for category rules) or in the action script (for command rules)—for example:

```
%define ...
  .
  .
  .
print page <n:number>; {start: meaning of n, end: meaning of n}
  .
  .
  .
%end
```

or, alternately,

```
%rule
print page <n:number>
%action
... meaning of n ...
%end
```

In category rules, each phrase may have an associated script that computes its value. There is no way to specify another script type for the value scripts of phrases—AppleScript is the only script system currently supported.

For command rules, you can refer to the category reference variables in the action script itself, as illustrated in the preceding examples.

Internal Category Rules

The following format is used for defining internal categories:

```
%define name [ open ]
[%target [ signature ] ]
phrase [; value-script ]
   .
   .
   .
%end
```

In this format, *name* is the category name. It must be a single word without punctuation. The *phrase* and clause formats are explained in the previous section.

External Category Rules

The following format is used for defining external categories:

```
%define name external [dynamic] [ open ]
[%context [ context ... ] ]
[%target [ signature ] ]
%action [  script-type ]
script
%end
```

The `external` and `dynamic` properties are indications that the possible phrases of this category should be obtained by executing the script that follows, using the *script-type* indicated. If *script-type* is not specified, then the default will be used and consequently must have been specified in a prior default statement. The value returned by the script should be either a list of strings or a single string of phrases separated by newline characters. These values are then active phrases for use within the grammar.

If *context* is specified for an external (not dynamic) rule, the action will be reevaluated whenever that context becomes active.

The string containing the exact source text as opposed to the words spoken is an additional property that is present for external categories. It is available through the `source` property. For example, consider an external category that returns the names of all sounds stored in the System Folder, assuming the class `'sound'` is implemented in the application theApp:

```
%define sound external
%action AppleScript
tell application "theApp"
   name of every sound
end tell
%end
```

One of the sounds might have a name such as `Click1`. The pronunciation generated for this item might be something like *click one.* However, to cause the sound to be played, you need its exact name. For this reason, the Speech Monitor provides both the words that are recognized in the utterance property and the exact name of the item in the source property—for example:

```
%rule
play <s:sound>
%action AppleScript
utterance of s        →  "Click 1"
source of s           →  "Click1"
%end
```

The `dynamic` option of category rules causes the rule to be evaluated every time speech is detected. Extensive use of dynamic categories is not recommended, since this will slow down the apparent response time of the speech recognizer.

Here is an example category that loads the values of the first two cells of an Excel spreadsheet as possible phrases:

```
%define <possibleAnswers> external
%context application "Excel"
%action AppleScript
tell application "Excel"
  {value of cell 1, value of cell 2}
end tell
%end
```

Context Specifiers

Contexts may be specified for both command rules and external category rules. A *context specifier* is a declarative representation that tells when the rule should be considered to be active. A context specifier is simply a list of context descriptors, each of which must be "active" in order for the whole context specifier, and thus the rule, to be considered "active." The syntax for a context specifier is this:

```
%context [ context ... ]
```

Each of the one or more context descriptors can be one of the following:

`application` *name*
`application` *id*
`window` *name*
`user` *name*
`suite` *id*

The application context descriptor causes a rule to be active only when a particular application is active. The application may be specified by its name or its signature. If the name is specified, `CompileRules` looks for an application with that name and uses its signature. Examples of valid application contexts are

`%context application "MyApp"`	application named *MyApp*
`%context application 'MACS'`	the Finder
`%context application *`	any application

The window context descriptor causes a rule to be active only when a specific window of the active application is active. Windows can be identified by either their name, their kind (the `windowKind` field of `WindowRecord`), or (for dialog boxes) their resource ID. If a window descriptor is used by itself, without an accompanying application descriptor, the rule will be active whenever the descriptor matches the current context, regardless of what application it belongs to. Here is an example of a valid window context:

`%context window "Speech Setup"`	window named *Speech Setup*

The user context allows a rule to be active when a particular user name has been entered in the Sharing Setup control panel. This allows users to have their own rule sets on a machine that is used by more than one person—for example:

`%context user "Bob Strong"`	enable Pig Latin rules

Default Statements

The default statement can be used to define rule characteristics that apply to all following rules in the speech rules file. This alleviates the need to repeat the specification for each rule. Valid default statements have these forms:

`%default context` *context* ...
`%default script` *script-type*
`%default target` *signature*

In the first case, the context descriptors are as specified in the context clause of rules—for example:

```
%default context application "MyApp"
```

In order to negate the effect of a default context inside a rule definition, simply specify the desired context, or an empty context if the rule is to be globally active:

```
%default context application "MyApp"
  .
  .
  .
%rule
```

```
hello
%context
%action
acknowledge saying "Yo"
%end
```

The default itself can be undone by specifying an empty context descriptor:

```
%default context
```

A default script type can be specified as follows:

```
%default script AppleScript
```

Subsequent scripts would not have to specify the script type in their condition and action clauses.

Global Scripts

A speech rules file may contain any number of scripts that are executed when the speech rules file is loaded. These are useful for defining handlers (subroutines) that are available to speech rules. Scripts defined using the `%global` clause are executed in a global context and thus can be used to define handlers and properties that are available to all rules, even in other files. Scripts defined using the `%local` clause are executed in a context that belongs to the speech rules file itself and thus cannot be shared with rules and scripts in other files:

```
%global [ script-type ]
   .
   .
   .
%end
```

or

```
%local [ script-type ]
   .
   .
   .
%end
```

If the optional script type is not specified, the default script type is used. It must have been previously defined. Any number of these clauses can appear in a file. Note, however, that scripts using different scripting systems cannot share handlers or properties.

CompileRules Error Messages

The following are descriptions of the error messages that can be generated by the `CompileRules` tool. For syntax errors, a line is printed with the filename and line number in a form that is suitable for executing in the MPW Shell (for example, by entering triple-click–Enter). Doing so will open the source file and set the selection to the line containing the error.

`Cannot create output file`
: The file you specified with the -o option could not be created. Possible reasons are that you specified a file on a locked volume or in a read-only folder; the volume containing the file you specified is full; or the startup volume, which is used as temporary storage during compilation, is full.

`Cannot find input file`
: One of the input files you specified does not exist.

`Command doesn't need any arguments`
: Indicates that an argument was encountered for the acknowledge clause. This clause does not take any arguments.

`Couldn't get file info for ...`
: One of the input files you specified could not be accessed.

`Didn't expect this:`
: During the processing of a rule phrase, some lexical element was found that didn't make sense. Typically this is caused by unpaired angle brackets, a missing label delimiter, an improperly quoted word containing special characters, or the like.

`Empty phrase script`
: A rule phrase was encountered that had the script delimiter (semicolon), but no script was found.

`Empty rule phrase`
: An empty rule phrase was encountered. All rule phrases must have at least one word or category reference.

`External rule shouldn't have a phrase`
: Indicates that a phrase was specified on an external category rule. External categories shouldn't have phrases, since their phrases are computed externally by a script.

`File is not a text file`
: One of the input files you specified was not a text file. `CompileRules` can compile only text files.

`Invalid context`
: Indicates that the context descriptor had invalid syntax. See "Context Specifiers," earlier in this chapter, for a description of valid context descriptor syntax.

`Missing category name`
: The category name is missing from a category definition. All category definitions must specify the category name.

`Missing context descriptor`
: Indicates that a context specifier was encountered that had no context descriptor. At least one context descriptor must be included.

`Missing default type`
: Indicates that the script type is missing from a default script statement.

`Missing script type`
: This error occurs when a script specifier occurs without a script type, when no default has been specified for the file. Either a default script type must be specified or every script specifier must include a script type.

`No input files specified!`
: You must specify at least one input source file.

`Premature end of phrase`
: Indicates that the end of a phrase was encountered when expecting a closing category reference delimiter. This is typically caused by a missing right angle bracket, label, or category name.

`Rule already has a condition`
: Indicates that more than one condition clause was found in a rule definition. Only a single condition may be specified for a rule.

`Rule already has a context`
: Indicates that more than one context was specified for a rule. A rule can have only one context specified, although that specification can contain multiple context descriptors.

`Rule already has an action`
: Indicates that more than one action clause was found in a rule definition. Only a single action may be specified for a rule.

`Trouble writing to temporary file...`
: An error occurred while writing to the temporary file used during compilation. Possible reasons are that the startup volume is full or a disk error occurred.

`Unknown category option`
: Indicates that one of the category options specified was unknown. Valid category options are `open`, `external`, and `dynamic`.

`Unknown or invalid command`
: Indicates that an unknown clause was encountered. The only valid rule clauses are `%context`, `%condition`, `%target`, `%acknowledge`, `%action`, and `%end`. The only valid clause for a global script is `%end`.

`Unknown rule file command`
: Indicates that an unknown speech rules file command was encountered. Valid speech rules file commands are `%default`, `%global`, `%local`, `%define`, and `%rule`.

`Unknown scripting system`
This error occurs when a script type is specified that is not registered with the system. This can occur when the script type is misspelled or when the scripting component was not installed at system startup time.

Apple Events Speech Events

The following defines the syntax of Apple events that are implemented by the Speech Monitor and can be invoked by speech rule scripts:

```
acknowledge [ success | failure | progress ]
  [ of hearing | recognizing | understanding | responding ]
  [ saying text ]
  [ caption text ]
reload rules [ file ]
```

An Example: A Simple Checkbook

Following are the complete rules for a simple checkbook grammar. For lack of a real application to control, the rules simply type the results into the Note Pad application.

```
%define uten
   one      ; 1
   two      ; 2
   three    ; 3
   four     ; 4
   five     ; 5
   six      ; 6
   seven    ; 7
   eight    ; 8
   nine     ; 9
%end

%define digit
   zero     ; 0
   oh       ; 0
   <x:uten> ; x
%end
```

```
%define tens
   twenty   ; 20
   thirty   ; 30
   forty    ; 40
   fifty    ; 50
   sixty    ; 60
   seventy  ; 70
   eighty   ; 80
   ninety   ; 90
%end

%define teens
   ten      ; 10
   eleven   ; 11
   twelve   ; 12
   thirteen ; 13
   fourteen ; 14
   fifteen  ; 15
   sixteen  ; 16
   seventeen; 17
   eighteen ; 18
   nineteen ; 19
%end

%define utwenty
   <x:uten> ; x
%end

%define uhundred
   <x:digit>; x
   <x:teens>; x
   <x:tens> ; x
   <x:tens> <y:uten>; (meaning of x) + (meaning of y)
%end

%define uthousand
   <x:uhundred>; x
   <x:uten> hundred; (meaning of x) * 100
   <x:uten> hundred <y:uhundred>; (meaning of x)
      * 100 + (meaning of y)
   <x:uten> hundred and <y:uhundred> ; (meaning of x)
      * 100 + (meaning of y)
%end
```

```
%define money
   <x:uthousand> dollars; (meaning of x) * 100
   <x:uthousand> dollars and <y:uhundred> cents
      ; (meaning of x) * 100 + (meaning of y)
   <x:uhundred> <y:uhundred>; (meaning of x) * 100 +
      (meaning of y)
   <x:uten> <y:uhundred> <z:uhundred>
      ; (meaning of x) * 10000 + (meaning of y) * 100 +
         (meaning of z)
%end

%define merchant
   Emporium
   Sears
   JC Penney
   Marshalls
   Macys
   Nordstrom
   Pacific Gas and Electric
   Pacific Bell
%end

%rule
%context application "npad"
   pay <n:merchant> <x:money>
   pay <x:money> to <n:merchant>
%action applescript
   do menu "Clear"
   type "pay " & meaning of x & " to " & utterance of n
%index checkbook, pay
%description
   Pays the vendor the amount requested (actually simply types
      the vendor amount into the open Note Pad window).
%end

%rule
   open checkbook
%action AppleScript
   do menu "Note Pad"
%index checkbook
%description
   Opens the Note Pad to begin checkbook function.
%end
```

```
%rule
%context application "npad"
   close checkbook
%action AppleScript
   do menu "Quit"
%index checkbook
%description
   Closes the Note Pad to stop checkbook function.
%end
```

The rule to pay `<merchant>` `<money>` uses the Speech Monitor to actually type the result into the Note Pad. No "tell application" block is needed, because the Speech Monitor sets itself to be the default application. Ideally, the Note Pad should be AppleScript-aware, so the script could be

```
tell application "Note Pad"
   set ourResult to "pay " & (meaning of x) & " to " &
      (utterance of n) & "\r"
   copy ourResult to contents of selection
end tell
```

and no `doMenu` or `type` command would be needed.

P A R T F O U R

System Software Modifications

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* covers miscellaneous changes to the system software in the Macintosh Quadra 840AV and Macintosh Centris 660AV, including a new manager for the internal and external SCSI (Small Computer System Interface) ports. It contains five chapters:

- Chapter 9, “SCSI Manager 4.3,” describes the new SCSI Manager in the Macintosh Quadra 840AV and Macintosh Centris 660AV.
- Chapter 10, “DMA Serial Driver,” covers the new hardware-independent serial driver that uses direct memory access (DMA).
- Chapter 11, “Video Driver,” describes changes to the video driver for the Macintosh Quadra 840AV and Macintosh Centris 660AV.
- Chapter 12, “New Age Floppy Disk Driver,” lists changes to the floppy disk driver and tells you how they affect floppy disk compatibility with other Macintosh computers.
- Chapter 13, “Virtual Memory Manager,” details how the Virtual Memory Manager no longer disables interrupts when performing certain tasks.

C H A P T E R 9

SCSI Manager 4.3

This chapter describes the new SCSI Manager architecture for the Macintosh Quadra 840AV and Macintosh Centris 660AV computers. It contains functional specifications describing the features, interface, compatibility, and performance of the new SCSI Manager. For hardware details of SCSI support in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see “SCSI Connection,” in Chapter 2.

In addition to the capabilities of the former SCSI Manager, the new SCSI Manager

- supports major new SCSI features such as disconnect and reconnect
- supports services such as fully asynchronous SCSI input and output
- provides a more hardware-independent API that minimizes the SCSI-specific tasks that a device driver must perform
- provides full use of whatever SCSI hardware is available
- supports existing SCSI device drivers with minimum or no modifications

This chapter starts with “SCSI Manager 4.3 Features,” which describes the current feature set, compatibility issues, performance, and some of the developer issues raised by changes to the SCSI Manager. Everyone should read this section.

“Design Overview” describes the layered structure of the new SCSI Manager and lists the general functions provided by each of the layers.

“Implementation” describes specific hardware and software dependencies of the new SCSI Manager. Compatibility with the previous SCSI Manager, the virtual bus, and data transfer methods are also discussed here.

Two sections, “Guidelines for SCSI Device Driver Developers” and “Guidelines for SIM/HBA Developers,” contain information for specific types of developers. If you are developing either a SCSI device driver, a SCSI interface module (SIM), or a host bus adapter (HBA) you should read these sections.

Finally, “SCSI Manager 4.3 Reference” discusses the actual API for the SCSI Manager 4.3, and “Summary of the SCSI Manager 4.3” lists its code interface.

SCSI Manager 4.3 Features

The SCSI Manager 4.3, in its Macintosh Quadra 840AV and Macintosh Centris 660AV release, supports the following new or improved features.

- *Parameter-block programming interface for SCSI I/O requests.* A parameter block contains all the information required to complete each SCSI I/O transaction. Additionally, this interface provides a hardware-independent view of the SCSI Manager. This independence allows the same device driver to work with any supported SCSI controller.
- *Asynchronous SCSI I/O.* SCSI Manager 4.3 handles both synchronous and asynchronous I/O requests. In addition, it allows multiple device drivers to maintain multiple outstanding requests.

- *Phase-cognizant implementation.* SCSI Manager 4.3 follows the phases driven by the target and performs the appropriate operations as specified in the SCSI I/O request parameter block. Driver clients no longer need to worry about SCSI bus phases. This eliminates a major source of development difficulties found in the old SCSI Manager.
- *Disconnect/reconnect features.* Disconnect/reconnect capability helps maximize SCSI bus utilization. It allows a device to disconnect and release control of the SCSI bus while the device processes a command from the host and to reconnect when the device is ready to communicate with the host. This allows the computer to submit requests to multiple targets so that those requests are executed in parallel. An example of this is a disk array application that can issue a request to one disk, which disconnects, and then issue another request to a different disk. The two disks can be performing seek operations simultaneously, thereby cutting down on the average seek time.
- *Parity support.* For the first time, parity is completely supported. This applies both to transmission of parity (which has always been the case) and to detection and handling of bad parity on reception. For compatibility reasons, a client can disable the parity detection on a per-transaction basis.
- *SCSI-2 support.* All SCSI-2 mandatory messages and protocol actions are supported as defined for an initiator. In addition, there are several optional features, such as disconnect/reconnect, that are also supported. There are optional SCSI-2 hardware features, such as Fast or Wide SCSI, that are anticipated by the architecture and API of SCSI Manager 4.3; when compatible hardware is available, device drivers will not have to be modified to take advantage of it.
- *Autosense feature.* The SCSI Manager automatically performs a request sense operation in case of a check condition and retrieves the sense data. This provides support for contingent allegiance conditions and unit attention conditions.
- *SCSI direct memory access.* SCSI Manager 4.3 can make use of any onboard direct memory access (DMA). This feature allows the host to perform other functions while data bytes are transferred to or from the SCSI bus.
- *Full support for multiple buses.* SCSI Manager 4.3 supports a full complement of devices on each available SCSI bus; this support allows a CPU with internal and external SCSI buses to access up to 14 SCSI targets instead of 7. In addition, third parties can create NuBus or PDS cards, with advanced SCSI adapters, that drivers can access through SCSI Manager 4.3 in exactly the same manner as through the standard SCSI bus. Users install a faster SCSI bus on an accessory card and move some or all of their SCSI devices to the new bus. Those devices will continue to work even if the SCSI devices, the SCSI drivers on those devices, and the SCSI bus are all made by different vendors. If these new SCSI adapters use 16-bit or 32-bit buses, all 16 (or 32) targets are addressable.
- *Full support for multiple logical units on each target.* SCSI Manager 4.3 allows full access to all 0–7 *logical unit numbers (LUNs)* on a target. These LUNs are treated as separate entities—I/O requests are queued according to LUN, and each LUN can maintain its own internal request queue (when target queuing is supported).

Compatibility

SCSI Manager 4.3 fully supports the current Macintosh SCSI Manager interface. It supports all calls and transfer information block (TIB) pseudoinstructions, except for `scComp` (compare) which is very rarely used. The lack of `scComp` is because of the support for DMA, which does not easily permit compare operations. Future implementations of the SCSI Manager are not guaranteed to maintain this level of compatibility with the old SCSI Manager API.

▲ **WARNING**
Applications or drivers that bypass the current SCSI Manager for any part of a transaction are not supported and will probably result in a fatal error. ▲

System Performance Impact

The performance impact of the SCSI Manager can be viewed from several different perspectives. Viewed from a raw byte-to-byte transfer level on the SCSI bus, SCSI Manager 4.3 performs like the old SCSI Manager. This is mostly a hardware issue—the performance of the SCSI Manager is tied to the level of performance of the hardware underneath. For information about Macintosh Quadra 840AV and Macintosh Centris 660AV SCSI performance, see "SCSI Connection," in Chapter 2.

However, viewed from the system level, the asynchronous capability provides significant increases in performance by allowing SCSI clients to regain control of the system while a SCSI I/O request is in progress. In addition, the support for disconnect/reconnect allows the system to have multiple I/O requests in operation on multiple targets concurrently, allowing another significant gain.

Because the Macintosh Quadra 840AV and Macintosh Centris 660AV hardware supports DMA, SCSI Manager 4.3 allows even more CPU cycles to be used for non-SCSI activity while a SCSI transaction is in progress. Just how many more depends on how much time is spent transferring data bytes. Another, though smaller, factor is how much the CPU uses the memory bus during DMA operations, because the DMA and CPU contend for the bus. An example of this factor is the relative difference between 68030 and 68040 bus use. Compared to the 68030, the 68040 has a higher cache hit rate (due to the larger cache) and a correspondingly smaller bus usage for the same set of instructions.

Impact on Developers

Two main product areas can take advantage of the new features of SCSI Manager 4.3: drivers for SCSI devices and add-on SCSI buses.

Almost all existing drivers and other clients of the SCSI Manager will continue to run without problems, as described in "Compatibility," earlier in this chapter. But most developers will want to modify their drivers to make use of the features of the new manager. "Guidelines for SCSI Device Driver Developers," later in this chapter, provides general principles critical to developers of drivers.

Developers of add-on SCSI bus adapters clearly gain from the new architecture. These NuBus adapter cards provide one or more additional SCSI buses, each of which typically provides higher throughput and/or enhanced capabilities beyond the hardware supplied by Apple on the main logic board. In the past there was no standard software interface for accessing more than one SCSI bus or for accessing the advanced features of these buses. Because of this, developers of these cards have had to provide their own SCSI Manager, which controlled only their SCSI bus.

The new SCSI Manager 4.3 architecture improves this situation significantly, as further described in "Design Overview," later in this chapter. The makers of SCSI adapter cards can continue to supply the software for controlling their bus. But now, this software accesses the SCSI Manager 4.3, providing it the ability to direct I/O requests to that bus. The clients of the SCSI Manager (drivers or applications) can access a SCSI device in the same manner, with the same calls and parameters, whether that device is connected to the Apple SCSI bus or to a third-party NuBus SCSI adapter. "Guidelines for SIM/HBA Developers," later in this chapter, explains general principles critical to this effort.

The support for third-party add-on SCSI buses provides another incentive for driver writers to recode. With the new SCSI Manager's application programming interface, their drivers will work with devices that are on any available SCSI bus.

Design Overview

This section provides a high-level overview of SCSI Manager 4.3.

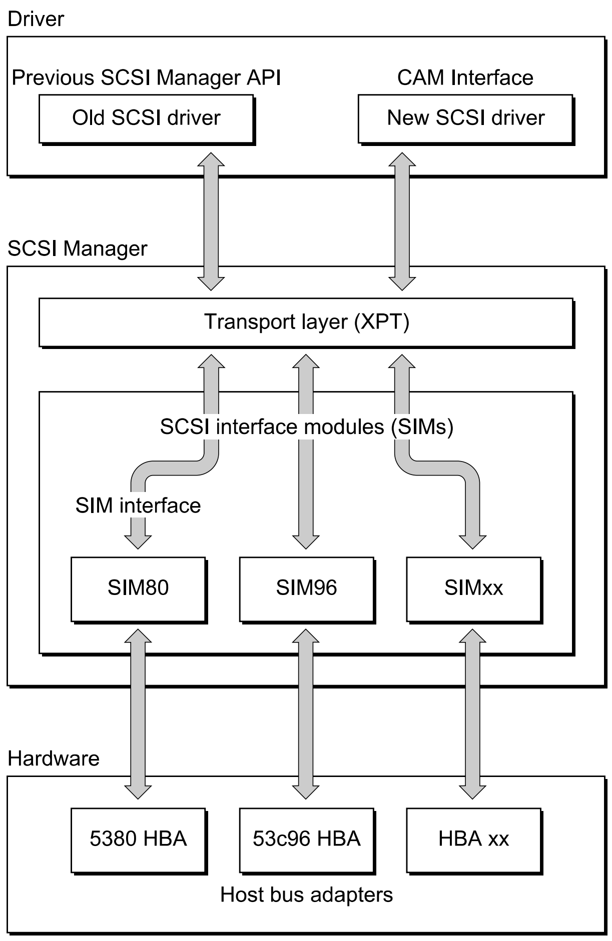
General Concepts

The SCSI Manager 4.3 application programming interface strongly resembles the software interface specified by ANSI X3T9 in the Common Access Method document (CAM). The SCSI Manager 4.3 interface however, contains Apple-specific areas for backwards compatibility and conformity with the Macintosh operating environment.

Intrinsic in the CAM-like interface is a CAM-like design. In CAM, there are two main layers—the transport (XPT) layer and the SIM layers. The XPT sits on top of multiple SIMs. Each SIM is responsible for controlling one host bus adapter (HBA), which constitutes the hardware associated with a specific SCSI bus adapter. There are a few requests that are handled entirely in the XPT layer, but in most cases the XPT simply passes the request to the SIM that has been registered to handle the HBA specified in the request.

These relationships are diagrammed in Figure 9-1.

Figure 9-1 SCSI Manager software hierarchy



Those readers familiar with the CAM document should note that Apple has adopted alternatives to some of the terms used by CAM. Table 9-1 shows the terms, their meanings, and their Apple equivalents.

Table 9-1 CAM to ACAM terminology conversion

CAM	Apple	Meaning
HBA	Bus or HBA	HBA is an acronym for host bus adapter, which contains all the hardware associated with a single SCSI bus adapter. This could be a bus on the main logic board, a NuBus card with a SCSI bus adapter, or any other native or attached SCSI bus. If there is DMA circuitry associated with that bus, it is also considered part of the HBA or bus.

continued

Table 9-1 CAM to ACAM terminology conversion (continued)

CAM	Apple	Meaning
Path	Bus	CAM uses “Path” to specify a particular HBA or bus attached to the system. Similarly, CAM functions and parameter blocks frequently include a `Path_ID` which is renamed `BusID` in this chapter.
CCB	SCSI_PB	CAM control blocks are the same as Apple’s SCSI parameter blocks or `SCSI_PB`.
In	Out	When referencing routine parameters, CAM uses the SCSI convention of direction with respect to the initiator. This is backward from the standard way of describing parameters and results for functions. For instance, if the SCSI Manager has a function with an input parameter `BusID`, this is typically considered “in,” but CAM refers to its direction as “out.” This is because, in SCSI, if a parameter is sent to the XPT, it is sent toward the target (away from the initiator), or “out.” Likewise, results returned from functions are considered “in” by CAM but are referred to as “out” in this chapter.
Out	In	See comment above.

Transport Layer

There is one XPT layer per system. This forms the access point for all clients of the SCSI Manager and has the following responsibilities:

- Provide the means to register HBAs, their characteristics, and their respective software entry points in SIMs.
- Route the request (parameter block or `SCSI_PB`) to the proper SIM.
- Provide higher-level facilities for old SCSI Manager interface compatibility. This consists of maintaining a translation list of `SCSISelect` IDs and their corresponding HBAs and directing them accordingly.
- Provide Operating System (OS) services to SIMs to isolate SIMs from OS dependencies. Such services include registration of interrupt handlers, static data space allocation and deallocation, and so on.

SCSI Interface Modules

Beneath the XTP layer lies one or more SIMs, each of which is responsible for interpreting the requests directed to it by the XPT. Each SIM “owns” one HBA. If there are multiple identical HBAs, there will be multiple identical SIMs. It is important to realize that a SIM designates not a code entity (such as a code resource), but instead represents the process or task which is responsible for controlling a particular HBA. For instance, if one SIM is coded for the Macintosh Quadra 900 (which has two identical 53c96 chips), there will be two SIM “instances,” one for the internal bus and one for the external bus.

The SIM's responsibilities can be broken into three main areas: queue maintenance, bus servicing, and assorted software services. The bus service routines are HBA-specific. The other two areas are very similar between various SIMs. Specifically, the SIM handles:

- queuing of multiple operations for all LUNs on same and on different targets and assigning tags for tag queuing (when supported)
- maintaining the queue, including freezing and unfreezing for queue recovery as necessary
- posting completed operations back to the requesting client (callback to device driver or application)
- managing the selection, disconnection, reconnection, and data pointers of the SCSI HBA protocol
- performing all interface functions to the SCSI HBA
- managing the data transfer path hardware (SCSI bus), including DMA circuitry and address mapping, and establishing DMA resource requests
- distinguishing abnormal behavior and performing error recovery, as required
- providing a time-out mechanism for tracking `SCSI_PB` execution using values provided by the peripheral driver
- supporting old SCSI Manager calls (optional on a SIM-by-SIM basis)

CAM Deviations

Apple has used the Common Access Method as a guideline during the creation of the SCSI Manager 4.3. CAM was never an attempt to provide either source-level or binary-level compatibility between different platforms. Considering this, it was viewed as more beneficial to provide a SCSI Manager interface that fit in with other Macintosh interfaces than to provide one that was similar to those on DOS or UNIX® platforms.

Implementation

The mechanics of issuing a call is slightly different than with the old SCSI Manager—instead of the stack-based `SCSIDispatch` trap, a register-based A-trap is used (`SCSIAtomic`). Several routines are accessed through one A-trap, distinguished by a routine selector word that is now one of the register parameters. C and Assembler macros and glue code are available that allow each of the routines to be called with a single line of code.

There are several routines that can be called in this manner. `SCSIRegisterBus` is used by a SCSI interface module during initialization to inform the XPT of its presence and its ability to handle SCSI requests. Various entry points and details of the SIM are passed to the XPT in a `SIMinitInfo` parameter block and, after registration, the XPT fills in other fields in the same parameter block, specifying details of the registration required by the SIM.

A SCSIDeregisterBus routine is provided to undo the effect of the SCSIRegisterBus routine. This is not likely to be needed in the current Macintosh environment.

The third routine, SCSIAction, is used by clients of the SCSI Manager to issue all other requests. Beside the selector word designating SCSIAction, the only other parameter is a pointer to a SCSI parameter block (SCSI_PB). A variety of functions are requested through this parameter block interface, the most important being SCSI_ExecIO, the function used to request a complete SCSI I/O transaction. Most of the functions available have corresponding SCSI parameter blocks defined to carry the input parameters as well as the results. For instance, the SCSI_BusInquiry function requires a SCSI_BusInquiry_PB parameter block, a pointer to which is passed to SCSIAction.

The SCSI_ExecIO_PB parameter block contains the destination of the SCSI I/O request (which bus, target, and logical unit), the command descriptor block (CDB), the description of the data buffer(s) which either supply or receive the data, and the results of the operation, as well as a variety of other fields and flags required to completely specify the transaction.

Different SIM implementations may require additional fields beyond the standard public fields in the SCSI_ExecIO_PB parameter block. Some of these may be input or output fields providing access to special capabilities provided by a SIM or they may be fields private to the SIM required during the processing of this request. In either case, the SCSI_ExecIO_PB parameter block may vary in size depending upon which bus is being addressed. In order to determine the size for a particular bus, a client must issue a SCSI_BusInquiry request (also through SCSIAction) which returns the size of the SCSI_ExecIO_PB parameter block as well as additional information about the specified bus. An appropriately sized parameter block can then be allocated, filled out, and issued to that bus to perform the requested SCSI_ExecIO function.

SCSI_BusInquiry is also used by a client to determine various hardware and software characteristics of a SIM/HBA. This may be required to adequately form a request that takes advantage of all of the SIM's capabilities.

SCSI_ExecIO calls can be made either synchronously or asynchronously. If the call is asynchronous, the caller may determine whether the action is complete either by checking the result field of the SCSI_ExecIO_PB parameter block or, preferably, by supplying an address of a completion routine (callback). Because of special interrupt handling considerations, you should issue all SCSI_ExecIO requests that need to be performed synchronously directly to the SCSI Manager, rather than issuing them asynchronously and then performing a sync-wait action in the client program. More details of this requirement can be found in "Guidelines for SIM/HBA Developers," later in this chapter.

When an error occurs during a SCSI_ExecIO request, the SIM may freeze the queue for the LUN on which the error occurred, to allow the client to perform any required error recovery. Upon completion of the recovery process, the client should issue a SCSI_ReleaseQ request to reenable the normal handling of I/O requests to that LUN.

Other SCSIActions that may affect SCSI devices are the SCSI_ResetBus, SCSI_ResetDevice, SCSI_AbortCommand, and SCSI_TerminateIO functions.

There are a class of `SCSIAction` requests that are used by SCSI device drivers and SCSI utilities to determine which Macintosh device drivers are responsible for which SCSI devices. These are `SCSI_SetRefNum`, `SCSI_RemoveRefNum` and `SCSI_GetNextRefNum`. These routines allow a client to maintain or examine the relationship between driver `refNums` and SCSI `DeviceIdents` (bus+target+LUN).

Target mode (although not supported in the initial SIM implementation) would be accessed through the `SCSI_EnableLUN` and `SCSI_TargetIO` functions.

Optional Features Not Supported in the SIM

If any of the following functions are requested of the SIM in the Macintosh Quadra 840AV or Macintosh Centris 660AV, an appropriate error code is returned:

- *Synchronous data transfer.* The SIM does not initially support synchronous data transfer. Any I/O requests designating synchronous data are rejected with a status of "Unable to provide the requested capability" and can simply be reissued without the synchronous data transfer request.
- *Target command queuing.* The SIM does not support SCSI-2 target queuing in its initial release. Eventual support is planned. Until that time, any I/O requests with the `QueueActionEnabled` bit set are rejected.
- *HBA engine support.* None of Apple's SIMs support HBA engines. A bus inquiry returns an engine count of zero. The engine inquiry and execute engine request functions return request completed with an error value.
- *Target mode.* Target mode is not currently supported.

Although these features are not supported in the initial implementation, third-party SIMs could provide support for these items because the XPT layer still delivers requests of these types to all installed SIMs.

Compatibility and Emulation

The old SCSI Manager routines (`SCSIGet`, `SCSISelect`, `SCSIComplete`, and so on) will continue to work under SCSI Manager 4.3 with very few compatibility problems.

A SIM/HBA may or may not be capable of supporting the old routines. When a SIM registers its HBA with the XPT, it must identify its `oldCallCapable` status—its ability to support the old routines or not. All Apple-supplied SIM/HBAs are capable of handling old calls.

`SCSIGet` calls set a flag that prevents any additional `SCSIGet` calls but perform no other operation. Upon the receipt of the `SCSISelect` call, the XPT issues a `SCSI_OldCall` request to the SIM which places it, like all other `SCSI_ExecIO_PB` parameter blocks, in its queue. Any SCSI parameter blocks that are awaiting initial execution (as well as any received while the old API transaction is in effect) are queued and do not begin execution until after a `SCSIComplete` is received and completed, which is when the queue is released. Any additional `SCSIGet` or `SCSISelect` calls received after an old API emulation has already begun are rejected with an error.

While the old API emulation is in progress, SCSI parameter blocks resident in or destined for queues in other HBAs are not affected—they continue to be executed as requested.

The old SCSI Manager routines supported by SCSI Manager 4.3 are listed in Listing 9-1.

Listing 9-1 Supported old SCSI Manager routines

```
OSErr SCSIReset (void);
short SCSIStat (void);
OSErr SCSIGet (void);
OSErr SCSISelect (short targetID);
OSErr SCSISelAtn (short targetID);
OSErr SCSICmd (Ptr buffer, short count);
OSErr SCSIRead (Ptr tibPtr);
OSErr SCSIRBlind (Ptr tibPtr);
OSErr SCSIWrite (Ptr tibPtr);
OSErr SCSIWBlind (Ptr tibPtr);
OSErr SCSIMsgIn (short *message);
OSErr SCSIMsgOut (short message);
OSErr SCSIComplete (short *stat, short *message, unsigned long
         wait);
```

`SCSIReset` calls are executed synchronously, but they can reset only buses that are controlled by SIMs that are capable of handling old calls.

The `SCSIStat` routine works and returns results as accurate as possible for the current old-call bus. If there is no current old-call bus, then the result indicates bus-free. If it is difficult for SCSI interface modules (SIM/HBA) to determine the exact state of the REQ signal during certain periods, for instance between the functions provided by old API calls, the SIM can be written so that the it does not return control to the XPT (for example, with an `rts` instruction) until a valid phase is on the bus.

There are several variances in support for transfer instruction blocks (TIBs). The first affects the `scComp` (compare) instruction, which is no longer supported in SCSI Manager 4.3. This results from the support for DMA hardware, which does not permit a compare operation. This should pose few compatibility problems, since it is rarely used; there were several previous versions of the SCSI Manager that did not perform compare operations properly.

Virtual Bus

SCSI Manager 4.3 has explicit support for multiple buses (HBAs), allowing a client to specify a target based on its bus number as well as its target ID and LUN. To support old API calls which understand only a target ID, the technique first used in the Macintosh Quadra 900 is expanded to include not only built-in SCSI buses but add-on NuBus and PDS buses as well.

In the Macintosh Quadra 900, SCSI transactions are directed to the first bus which responds to a select for the requested ID. The ID specified in a `SCSISelect` routine is called the "virtual ID" because it designates a device on the single "virtual bus" (which encompasses both internal and external buses). When a `SCSISelect` call is made, a selection of the virtual ID is attempted on the internal bus first, and if there is no response, the selection is attempted on the external bus. Once a successful selection of a virtual ID occurs, all subsequent `SCSISelect` calls are directed to the bus on which that selection occurred. Until a successful selection has occurred on one of the buses, the virtual ID is not assigned to a particular physical bus. Once established, a virtual ID to physical bus mapping is not changed until restart.

The virtual bus is maintained by the compatibility portion of the XPT layer, which determines the virtual-to-physical ID mapping. The XTP layer does this by attempting to select the virtual ID on each of the HBAs that can handle old calls until it finds a device that responds. The HBAs are searched in order of registration, which in most cases is first internal, then external, then additional third-party add-in SCSI cards. The Macintosh Quadra 840AV and Macintosh Centris 660AV have no separate external bus.

Data Transfer Descriptions

Clients of SCSI Manager 4.3 can use several different structures to describe the source (or destination) memory buffers for data transfer to (or from) a SCSI device. The easiest is the single buffer descriptor, consisting of a buffer address and a buffer length. A more difficult descriptor is needed when there are discontiguous areas of memory that make up a client's data transfer. These are specified by a single scatter/gather (S/G) list. Buffer descriptors include address and length. There is an additional parameter for how many items are in the S/G list.

In the old SCSI Manager, the TIB is made up of a series of transfer instructions. During the execution of a `SCSIRead`, `SCSIWrite`, `SCSIRBlind`, or `SCSIWBlind`, the TIB instructions (transfer and increment address, transfer and don't increment, add longword to address, move longword, loop, compare, and stop) are interpreted by the SCSI Manager to determine the source and destination of the data. Additional details can be found in *Inside Macintosh,* Volume IV.

Although S/G lists are simpler than TIBs, TIBs were actually designed for an additional purpose—they are also used to show the SCSI Manager where long delays (greater than 16 µs) may be found in the data transfer. This was required for the Macintosh Plus because of the lack of hardware handshaking between the SCSI chip and the CPU. It was required in all later Macintosh computers as well for slightly different reasons. SCSI Manager 4.3 only supports TIBs for old API calls. All new API calls must specify either a single buffer descriptor or S/G lists.

A second aspect of the TIB is used by the `scHandshake` field of the `SCSI_ExecIO_PB` parameter block. This field is a series of words, each of which specifies the number of bytes between potential delays in the SCSI data transfer. For instance, `1,511 TIB` is a common TIB structure needed to work with drives which have a 512 byte block and sometimes have a delay between the first and second bytes in the block as well as a delay between the last byte of a block and the first byte of the following block. This TIB structure translates to an `scHandshake` of 1, 511, 0… which translates to a request to transfer 1 byte, synchronize then transfer 511 bytes, synchronize, transfer 1 byte, and so on. As can be seen from the example, this structure is null-terminated and can have a maximum of 8 byte counts/handshake points.

Guidelines for SCSI Device Driver Developers

SCSI device drivers and other clients written to take advantage of the SCSI Manager 4.3 must continue to perform all of the operations associated with their counterparts when dealing with the old SCSI Manager. In addition, they will be have to follow some new rules that asynchronous data transfer and multiple-bus support require.

A SCSI device driver interfaces with its client (typically through the Device Manager) and with the SCSI Manager. Because the old SCSI Manager was completely synchronous, SCSI drivers were synchronous as well. If a driver is rewritten to issue asynchronous SCSI requests, it can also be rewritten to behave asynchronously as well (with respect to its clients). This is critical in order for the benefit to reach the application level.

Booting and Drive Mounting

For earlier ROMs, the OS scans the SCSI bus from ID 6 to ID 0, looking for all devices that have an `Apple_HFS` as well as an `Apple_Driver` partition. When one is found, the driver is loaded and executed and installs itself into the unit table. The driver then places an element in the drive queue for any HFS partitions that are on the drive. The Start Manager then records these `DrvQElements`, mounts the startup volume, and attempts to start up the computer.

There are six unit table slots ($20 through $26) reserved for SCSI drivers for devices at IDs 0 through 6 respectively. In the Macintosh Quadra 840AV and Macintosh Centris 660AV architecture, which allows the addition of many SCSI buses, device drivers must be able to allocate their unit table slots dynamically. All SCSI drivers, including drivers written for SCSI Manager 4.3, must attempt to install themselves in the unit table at the same location specified under the old calls, `$20+SCSI_ID`. This attempt lets both old and new drivers serve as boot devices when booting from an earlier machine. If a driver finds that the slot is already full, it should search for an empty slot in the `48 ($30)` to `UnitNtryCnt` range.

SCSI Manager 4.3 is able to distinguish between old drivers that support only the old API and new drivers that are aware of SCSI Manager 4.3. Drivers aware of SCSI Manager 4.3 are identified in the partition map as type `Apple_Driver43` instead of the previous type, `Apple_Driver`. Because the old ROMs checked only the first 12 characters of the type before loading and executing the driver, both new and old drivers will work on older machines.

Once the drivers have been loaded and executed, the ROM searches for the default startup device in the drive queue. If it is there, it is mounted and the boot process begins. In the Macintosh SE ROM and all ROMs since, the boot drive is identified by a driver reference number. The unit table slots for SCSI drivers are always in the range $20 through $26—the slots set aside for SCSI drives at IDs 0 through 6 respectively. This works fine when drivers have the same `refNum` between boots but, in the Macintosh Quadra 840AV and Macintosh Centris 660AV, drivers allocate unit table slots dynamically.

Currently, the driver reference number (a word) is stored in parameter RAM (PRAM) and is used by the Start Manager to pick the startup device. However, SCSI Manager 4.3 designates the startup device by using `DeviceIdent` (containing the bus number, SCSI ID, and LUN of a device), which supports multiple buses. To access devices on multiple-bus CPUs, you must know which device to boot from and whether the external device can be mounted at `LateLoad` time.

Some devices may be “hidden” from access to the old API calls if a device with the same ID is found on a higher-priority HBA. With earlier ROMs, the old SCSI Manager loads all drivers that are found. On multiple-bus machines, this does not include those devices with the same IDs as devices on higher priority buses. When SCSI Manager 4.3 is running, access to those devices can be made only through the new API and hence only by new drivers (`Apple_Driver43`). `LateLoad`, which runs after SCSI Manager 4.3 has been installed, scans all known buses and load all additional new drivers found, using the new API. The new drivers then mount their respective drives. SCSI Manager 4.3 also loads additional old drivers if those drivers are accessible by emulating the old API.

When SCSI Manager 4.3 is present at startup (in ROM), all new (`Apple_Driver43`) drivers are loaded from all drives found. Then pre-4.3 (`Apple_Driver`) drivers are loaded if they are found on a device (accessible via emulation of the old API) that corresponds to the virtual-to-physical mapping for their SCSI ID.

If a pre-4.3 ROM loads an `Apple_Driver43` driver, it treats it exactly like an `Apple_Driver` driver. This means that during initialization, the Start Manager makes a call to the beginning of the driver (defined as the first byte) with register D5 set to the SCSI ID of the device the driver was loaded from. To provide complete compatibility, an additional entry point has been defined for `Apple_Driver43` at 8 bytes from the start. If this entry point is called, it means that SCSI Manager 4.3 is present and that a `DeviceIdent` value is in register D5. No other registers are valid.

There could be situations where SCSI Manager 4.3 becomes active after an `Apple_Driver43` driver is loaded. This would occur if a newer system was patching an older ROM; the old ROM would load and install the driver and the system would come up later, installing SCSI Manager 4.3. To recognize the appearance of the new SCSI Manager, every `Apple_Driver43` driver should check for the presence of the `_SCSIAtomic` trap ($A089). The best place to do this check is at the first `accRun` tick (from `dNeedTime` flag). This tick happens after the system patches are in place.

Asynchronous Behavior

The successful execution of asynchronous I/O requires a whole set of rules that were not a concern when dealing with the synchronous SCSI Manager. The general form of an asynchronous SCSI driver is different than that of an old synchronous driver. When a client makes an I/O request, the Device Manager queues that request in the driver’s I/O queue and then make a call to the driver's `_Prime` routine. That routine should stuff a `SCSI_ExecIO_PB` parameter block (PB) with the parameters necessary to complete the request (or multiple PBs if required) and send them to the SCSI Manager through `SCSIAction`. The proper SIM then adds the request to its queue and possibly start working on it before returning back to the driver.

At this point, virtually nothing can be assumed by the driver about the request. If it was accepted it may have only been queued or it may have proceeded all the way to completion. If the return value is a value other than `noErr`, it is the result of input parameter errors. Either the parameter block was built incorrectly or it contains an error in one or more parameters. If the return value provided by `SCSIAction` is `noErr`, the command has been accepted and the contents of the `SCSI_ExecIO_PB` are no longer valid. This is because of the asynchronous nature of the SCSI Manager.

▲ **WARNING**
Once a parameter block has been accepted by the SCSI Manager, no attempt should be made by the driver to read it. The current parameter block being worked on by SCSI Manager 4.3 may be from a different request and completely incorrect for the driver. ▲

Typically a callback routine is supplied with the `SCSI_ExecIO_PB` parameter block. This routine allows the SIM to asynchronously notify the client that the request has completed. SCSI drivers must always use such callbacks.

▲ **WARNING**
SCSI drivers must always use a callback routine when issuing asynchronous requests. If a callback routine is not supplied the client cannot be notified asynchronously. Being notified asynchronously would require that the driver perform a sync-wait action, which is not permitted because of virtual memory compatibility factors. ▲

For SCSI requests that can be handled synchronously, such as those required during error handling or initialization, the driver should issue those requests to the SCSI Manager synchronously, rather than asynchronously, and then wait for the `scResult` field to change from `scsiReqInProg`. The latter process is effectively a synchronous request, except that the sync-wait is performed in the driver. This is not allowed. A further explanation is given in the next section.

An asynchronous I/O request issued by a client to a driver may occur at interrupt time. This eliminates the possibility of allocating memory to handle the request at the time the request arrives. This means that any `SCSI_ExecIO_PBs`, S/G lists and any other structures that are needed for the processing of the I/O should be allocated at driver initialization time. Unlike a synchronous request, none of these can be allocated on the stack because they would disappear when the driver returned from the `_Prime` routine.

When issued asynchronously the resulting action may start at any time and may end at any time. There is no implied ordering of these events with respect to earlier or later requests. An earlier request may be started later, or a later request may complete earlier. However, a series of requests to the same device (bus ID + target ID + LUN) is issued to that device in the order received.

Virtual Memory Operation

There is a possibility that an application may disable interrupts and then cause a page fault. Because this page fault translates to a synchronous SCSI driver request, the SCSI Manager handles the resulting SCSI request without the benefit of interrupts. The Macintosh Quadra 840AV and Macintosh Centris 660AV require that all sync-waits be performed either in the SCSI Manager or in the Device Manager where there are hooks that provide the sync-wait loop the ability to poll the SCSI interrupt sources.

If a driver has received a synchronous I/O request (typically from `_Prime`), the driver has two options. Either it can issue the subsequent SCSI Manager request synchronously as well, or it can issue the SCSI request asynchronously and simply return back to the Device Manager. The Device Manager sits in a sync-wait loop, awaiting the completion of the request. The driver should call (or jump to) `IODone` after it receives the SCSI completion callback. If the single driver request translates to multiple SCSI requests, the driver can issue each of those requests synchronously or it can issue them asynchronously and return back to the Device Manager. The driver should, in this case, call `IODone` after the callbacks for all of the SCSI requests have been received.

▲ **WARNING**
Under no condition should the driver use a sync-wait loop. If it does, the SCSI Manager will never be allowed to clear the interrupts and will hang indefinitely. ▲

Never perform sync-waiting in the driver itself. The wait must be controlled by the Device Manager (by returning from the `_Prime` routine) or in the SCSI Manager (by issuing the SCSI request synchronously).

As explained in Chapter 13, "Virtual Memory Manager," virtual memory (VM) in the Macintosh Quadra 840AV and Macintosh Centris 660AV executes I/O completion routines, Time Manager tasks, VBL and slot VBL tasks, deferred tasks, and `PPostEvent` actions without disabling interrupts. If a completion routine is to be run while VM is running the deferred user function queue (with interrupts enabled), VM queues this new completion routine at the tail of the deferred user function queue. This assures that routines of the types listed above will execute in their original order.

The SCSI completion routines (callbacks from `SCSI_ExecIO`) are similar to `IOCompletion` routines except for one major difference. Because they can cause page faults and typically occur at "interrupt time," `IOCompletion` routines are handled by VM; if it's safe for paging at the time the call to `IODone` is made, the `IOCompletion` routine is executed immediately. If it isn't safe for paging, VM defers execution of the `IOCompletion` routine until it is safe.

Like `IOCompletion`, SCSI completion routines are usually called at interrupt time. The difference is that VM does not intercept them. This means that SCSI completion routines are called even if it is not safe for paging, so they are not allowed to cause page faults. For SCSI drivers, that is not usually a problem—their whole world is usually held anyway. In response to a SCSI completion, a driver typically calls `IODone`, which makes a call to the client's `IOCompletion` routine, which could cause a page fault. This is not a problem, because, as mentioned already, VM defers the call until it is safe for paging.

Guidelines for SIM/HBA Developers

Developers of SCSI HBAs should ship their products with SIMs that are compatible with SCSI Manager 4.3, to enable other vendors' drivers and devices to work with their SCSI adapter.

SIM Initialization and General Operation

For the Macintosh Quadra 840AV and other machines with ROMs that contain SCSI Manager 4.3, third-party SIMs can register their HBAs as soon as their code is executing. This happens during NuBus configuration ROM setup (`PrimaryInit`).

For systems without SCSI Manager 4.3 in ROM, SIMs must wait until SCSI Manager 4.3 is up and running before registering with the XPT. This means that a drive on a third-party HBA cannot be used as a boot device nor as the backing store for VM, but can be mounted once SCSI Manager 4.3 is running and the HBA is installed. This secondary driver loading and drive mounting happens after SCSI Manager 4.3 is operational.

To initialize itself, the SIM issues a `SCSIRegisterBus` call, with a pointer to a `SIMinitInfo` structure that has been filled out with the entry points, required static data space size and `oldCallCapable` status of the SIM. The `SIMinitInfo` structure is shown in Listing 9-2. This structure can be disposed after the routine finishes, because the XPT makes a copy of the data.

Listing 9-2 SIM initialization information structure

```
typedef struct {                // used for SCSIRegisterBus call
   uchar    *SIMstaticPtr;      // <- ptr to the SIM's static vars
   long     staticSize;         // -> bytes SIM needs for static
                                //    variables
   long     (*SIMinit)();       // -> pointer to SIM init routine
   long     (*SIMaction)();     // -> pointer to SIM action routine
   long     (*SIM_ISR)();       // -> pointer to the SIM ISR routine
   void     (*NewOldCall)();    // -> pointer to the SIM NewOldCall
   long     intrptSource;       // -> interrupt source specifier
   Boolean  oldCallCapable;     // -> true if this SIM can handle
                                //     old SCSI Manager calls
   ushort   busID;              // <- bus # for the registered bus
   void     (*XPT_ISR)();       // <- pointer to the XPT ISR
   void     (*MakeCallback)();// <- pointer to the XPT layer's
                                //     MakeCallback routine
} SIMinitInfo;
```

The XPT allocates the requested number of bytes for the SIM static space and assigns the next bus number to this SIM. A pointer to the allocated memory is returned as is the `busID` that was assigned, in the appropriate fields of the `SIMinitInfo` parameter block. The XPT also fills in two fields indicating entry points into the XPT. The `XPT_ISR` entry point should be used by the SIM when the XPT has not provided sufficient interrupt registration functions. The `MakeCallback` routine should be called when the SIM completes a `SCSI_ExecIO` request and needs to make the SCSI completion callback to the client.

The `SIMinit` routine is called during the execution of the `SCSIRegisterBus` routine. It is passed the `SIMstaticPtr` (address of SIM's static data space) as well as a pointer to the `SIMinitInfo` structure after the XPT has filled in all of the required fields. `SIMinit` attempts to initialize all data structures and hardware and returns an appropriate result code indicating whether this was successful. If a failure result is returned, the XPT does not register the SIM.

Once the registration is complete, the XPT makes calls to the `SIMaction` entry point whenever a `SCSIAction` call is received that is destined for this bus. The XPT passes a pointer to the `SCSI_PB` parameter block and a pointer to the SIM's static space into the `scSIMPrivate` parameter of the `SIMaction` routine. The SIM should parse the `SCSI_PB` parameter block for illegal or unsupported parameters and return an appropriate failure code if either of these are found. All `SCSI_ExecIO` requests should be treated asynchronously by the SIM; all other types of requests should be treated synchronously.

Once the `ExecIO` request has been queued, the SIM should return back to the XPT. When the request finishes, the SIM calls the XPT's `MakeCallback` routine. This makes the call to the client's specified completion routine (if it exists).

The parameter blocks appear to the client to be queued on a per-LUN basis as queue freezing and unfreezing is performed one LUN at a time. In actuality, the SIM may queue all `SCSI_ExecIO_PB` parameter blocks in the same queue no matter which LUN is the destination. This helps maintain a first-in, first-out sequence of the parameter blocks.

Support for the Old SCSI Manager

Upon registration, every SIM should specify whether or not it supports old SCSI Manager routines. If it indicates that it does, the XPT adds it to the list of buses searched when a `SCSISelect` call is received.

The handling of an old API transaction differs from the handling of new API `SCSI_ExecIO_PB` parameter blocks. The main difference is the presence of communication between the XPT and the SIM during the transaction. This communication consists of the old API calls (to the SIM) and the results returned (from the SIM) at the completion of the routines. The XPT is responsible for catching and converting the old calls into the proper format and submitting them to the SIM. It also takes the results for each of the calls from the SIM and returns back to the client with those results.