System Software 5.0—The Times, They Are a'Changing

You've been playing with your Apple IIGS for nearly two years now. It's nice, but it's a little sluggish. You've been hearing those rumors floating around that Apple doesn't support the Apple II anymore, and you've had your eye on one of those other computers with color, a mouse and an alleged user interface.

Then you get Apple's newest version of the "System Disk"—System Software 5.0 for the IIGS. Your eyes start to widen. Windows zip open in the Finder. AppleWorks GS™ loads from your 3.5" drive in an ***eighth*** *of the time it took with the old version. A new Control Panel NDA has pop-up menus in it. Nearly every program you have that uses the desktop interface is running faster, sometimes* ***amazingly*** *faster. Scrolling NDA and CDA menus let you have more than the previous number of desk accessories and still get to them all—without that "Two Apples" thing you've been meaning to read about in the September Call -A.P.P.L.E. You can have lowercase letters in file names on ProDOS disks. The Finder actually says your ShrinkIt archives are ShrinkIt documents. And switching back from ProDOS 8 takes about 4 seconds as opposed to what seemed like 4 years. Visions of orphaned computers vanish in the light of reality.*

This is all well and good, and you spend a few days discovering how much fun it is to use your computer again, stopping occasionally to drool a little bit. But soon the fun wears off—there's only so many times you can watch the Finder "Close All" and be amazed at the speed. You're an Apple II owner—you want to know the innards. What's new in there for programmers? How can I do pop-up menus? Or scrolling menus? What do I have to do to get the speed necessary? Has anything *been done to make the desktop interface easier for programmers? Are there new tools? Do Dogcows live in the Apple II world? Doesn't "ORCA/C" evaluate to "ORA" if "C" is nonzero?*

Ah, you think to yourself, if only a magazine I was holding in my hands right now answered these questions, maybe even with a source code example....

The advent of System Software 5.0 is, in many ways, as revolutionary as the IIGS itself was three years ago this month. Without a hardware upgrade, installing this new system software on your hard drive suddenly reveals words that previously were hidden. Desktop interfaces are easier to generate than ever before—perhaps easier than on any other machine. Applications have access to networking through the operating system in ways that make server access virtually transparent. One extra step in the build procedure produces files that load two to four times faster than existing load files. New tools present the world of resources, and a package that handles full desktop text editing functions with barely any work by the application.

Non-programmers instantly understand that System Software 5.0 is all about *speed.* Things zip along at rates they previously thought required additional hardware. Programmers interested in 5.0 soon learn the other major component of 5.0—*power.* Applications do more in less code. The system works more to make the desktop interface happen, and the application has the freedom to be brilliant without thousands of user interface details interfering with a programmer's good ideas.

As in the past, the System Software is divided into three main components—the operating system (GS/OS), the Toolbox, and the Finder and other applications. The other two main components of the system, the firmware (ROM) and the hardware, are obviously not affected by new System Software. We'll examine the changes to the three areas separately.

Note that this article is not and can not be a tutorial. Early indications are that developer-level documentation for 5.0 (including GS/OS and Toolbox) changes is easily

going to be several hundred pages long. These are references that will be available from APDA and should be purchased by anyone wishing to use 5.0 features. Because of the sheer volume of information, this article can only present a general overview, and specific examples of one subset of the changes for 5.0. Many details will be presented, but encyclopedic reference is left to those books designed to provide it. I don't suggest trying to use new 5.0 features without adequate reference unless you're into heavy, intense personal pain and anguish.

An overview of System Software 5.0

GS/OS

GS/OS 3.0 is included with System Software 5.0. While the general design of GS/OS remains the same, the specific implementation of several components of it has changed greatly. New system calls have been provided; new capabilities have been given to drivers; drivers have been added as has a new File System Translator (FST); and performance has been improved.

GS/OS and the AppleTalk networking system join hand in hand on System Software 5.0 with the addition of AppleTalk drivers and an AppleShare FST. Normal file level calls may be made to a server volume and the AppleShare FST will interpret them and return the appropriate information, just as the ProDOS FST interprets calls and provides information from ProDOS disks. New AppleTalk drivers provide general AppleTalk parameters as well as GS/OS character output for printing text over the network (the Remote Print Manager is now a loaded driver) and providing implementation of some AppleTalk Filing Protocol (AFP) services as well as access to the ProDOS Filing Interface (PFI) through another driver.

This new architecture allows GS/OS to operate fully and functionally with AppleShare and AppleTalk. Additionally, methods documented in the *AppleShare Programmer's Guide to the Apple IIGS* are still supported for backwards compatibility. In fact, many services are not provided by the GS/OS drivers through GS/OS call mechanisms because those services are handled by the previously-documented methods. GS/OS programmers as well as AppleTalk programmers will find the new system software compatible and understandable.

AppleShare is GS/OS's first FST that not only has both reading and writing capabilities, but also contains a disk structure and rules for file name syntax almost completely different than those of ProDOS. Programs oriented specifically to ProDOS disks may run into problems with AppleShare volumes. On AppleShare, file names can be longer than 15 characters and can contain special characters such as "*f*" or "√" as well as spaces. Directories can not be directly read; the `GetDirEntry` call must be used (AppleShare directories do not have the same format as ProDOS directories—trying to read and interpret them wouldn't make sense). Access privileges actually mean something significant—if an application doesn't have permission to `Destroy` a file, it can't simply use `GetFileInfo` and `SetFileInfo` to change the access to something it likes better, since the file may not belong to the user running the application. The world of multiple file systems is different than the ProDOS-only world many Apple II programmers are used to, and AppleShare is the first new file system to point it out.

There are also AppleShare-specific tips to make otherwise network-*compatible* applications be network-*friendly*. Remember that more than one person could be running the application at once, so don't write configuration information to the application program file. Don't always use the same file name for temporary files (a second user running the application could cause a new temporary file to erase one currently being used). Catalog a directory until you see the end (signified by the "End of Directory" error from

GetDirEntry), not by using a count of files obtained from that call. The count could change during the catalog operation. You might also get duplicate entries since AppleShare directories are in alphabetical order, and someone could create a file during your catalog operation. Most importantly, only ask for file access that you're going to need. If you're only going to read a file, don't Open it with "as available" or "read/write" permission—no one else can then get to that file until you Close it. Also, if you Open a file with "as available" access, GS/OS has no way to tell you what access you really got (the returned access word is not suitable for this task), so you may get error $4E (access error) when trying to Write to the file later on, since you only had read-only permission to the file. If you ask for write permission when you Open the file, GS/OS will tell you right there that you don't have access to Write if you don't. Instead of checking for access problems on every call, you can resolve them at Open time.

A new capability to load specially-processed OMF load files faster than ever before is part of 5.0, and is not surprisingly called *ExpressLoad*. ExpressLoad is part of GS/OS, and resides in a file named ExpressLoad in the System folder. It is loaded at boot time automatically except on 512K machines (to save memory).

ExpressLoad operates on files that have been "Expressed", using the "Express" utility that works with APW, or the ExpressIIgs tool that works with MPW IIGS. Express takes a load file and pre-processes it, pre-expanding it in some ways. It keeps information about the file in a dynamic data segment with the name "ExpressLoad". The information includes the position and size of each segment in the file (including relocation dictionaries). It also rearranges segments for optimal disk access and converts code segments to other OMF forms to avoid extra loader work. ExpressLoad itself special cases certain common operations for maximum performance and keeps the application file open if it contains dynamic segments (to avoid having to re-Open it).

Since the Express segment is a dynamic data segment, it's only loaded if someone asks for it. The application and the regular System Loader won't, so Express format is fully compatible with existing applications. For example, AppleWorks GS™ 1.0v2 shipped in Express format, and loads just fine with or without ExpressLoad. Future linkers (Apple's or third-party) could be revised to automatically create Express format files. They are slightly bigger than non-Expressed files, but their speed in loading will nearly always make up for it.

ExpressLoad gets first crack at calls made to the System Loader, which is then called to process a file if it is not in Express format or if Express can't handle it. Express is designed to speed up the majority of applications, so certain special cases may not work. In particular, ExpressLoad will not work with files that it did not IntialLoad. The regular System Loader has worked in the past with files it had not InitialLoaded (for example, LoadSegName might succeed on a file that had not already been InitialLoaded), but this isn't really guaranteed by Apple and should be avoided. It might continue to work in the future, and it might not—it definitely prevents your application from using the advantages of ExpressLoad.

ExpressLoad also doesn't support some Loader functions and it's recommended you avoid others. Since Express rearranges segments for optimal performance, and segments are consecutively numbered, it follows that the segments will be renumbered as well. Although ExpressLoad can still load segments properly (since it keeps a mapping from old numbers to new numbers in the Express segment), the regular System Loader does not know about this mapping. Therefore, loading or unloading segments by number (LoadSegNum and UnLoadSegNum) will not work and should *always* be avoided. Always load segments by name, not by number. Furthermore, GetLoadSegInfo returns internal data structures from the System Loader that ExpressLoad does not use or

support, so `GetLoadSegInfo` is not supported by ExpressLoad. All other current Loader calls are supported.

Several changes to the driver mechanism of GS/OS provide new functionality and improved performance. The Device Manager has significantly improved performance on GS/OS "device" system calls like `DRead` and `DWrite`, as well as on single-character input and output through character devices. Also, the Device Manager now knows when an Apple peripheral needs a loaded driver that isn't present—it warns the user that the loaded driver is needed and doesn't generate a driver. Devices now have the option of being renameable and restartable. A restartable driver can start itself more than once without having to be reloaded from disk. Things such as pre-initialized data that is later changed or self-modifying code are examples of things which could prevent drivers from being restartable. All the Apple drivers included with 5.0 are restartable except the Apple 5.25" Disk driver.

The Apple 3.5 Driver (`AppleDisk3.5`) also includes a new capability referred to as "scatter read". Although the driver still can't write to the disk at 1:1 interleave (this is physically impossible), it pulls a nifty trick in reading. It reads an entire track in on one spin of the disk, decoding the interleave later. So instead of two spins to read one track at 2:1 interleave, it now takes one spin to read one track (even at 2:1 interleave). This capability happens on multi-block reads without caching to non-shadowed memory, since timing conditions are kind of tight. Although it's not guaranteed that such conditions will always invoke scatter read, not meeting them currently prevents it. For giant reads (such as entire disks), this results in an almost 2X speed increase over regular 2:1 interleave reads.

New SCSI peripheral support arrives in 5.0 with the introduction of the SCSI Manager, the full-fledged GS/OS Supervisory Driver for SCSI peripherals Apple promised to developers at 4.0 time. The SCSI Manager takes full control of the Apple II SCSI card or cards in the system (it does not work with non-Apple SCSI cards) and provides a nearly 5X performance increase, from 16 MB/second under the 4.0 `SCSI.DRIVER` to 80 MB/sec with the new SCSI Manager. Also included are two drivers that run under the SCSI Manager Supervisory Driver ("slave" drivers, if you will) for SCSI hard disks and Apple CD SC drives. Apple's Developer Technical Support department will help those interested in creating SCSI peripherals and drivers, since the details of using the SCSI Manager get quite detailed and technical, and bore the pants off those not incredibly intensely interested in SCSI.

GS/OS also now has a method for drivers and other parts of the operating system to let interested applications know when certain events have happened. Through new calls `AddNotifyProc` and `DelNotifyProc` (Add or Delete Notification Procedure), GS/OS or drivers can notify applications when the system switches GS/OS to ProDOS 8 (or back), when disks are inserted or ejected, when the system is shut down or when there is change to a volume (writing occurred to a volume). Applications like the Finder use this mechanism to reduce the amount of polling done to disk devices. (It's true that the Macintosh doesn't poll disks in the Finder, but Apple II owners can eject disks without asking the computer to do it—doing that on the Macintosh can confuse the operating system.)

GS/OS now provides for **standard input, output** and **error**. These are standard methods of obtaining input (reading from "standard in"), producing output (writing to "standard out") and handling either input or output generated by error conditions (reading or writing to "standard error"). GS/OS implements this through the use of prefixes 10, 11 and 12 respectively. Unless a launching application specifically requests that GS/OS not do this, those prefixes are changed to "`.CONSOLE`" before launching an application. The application may then simply open "10:" for standard input, "11:" for standard output and

"12:" for standard error, using GS/OS `Read` and `Write` calls to obtain the data. Additional support for this is provided in new system calls.

New system calls? Yup. In addition to the new system calls for the Notification Queue, GS/OS has added a few new system calls for dealing with open files as well. `GetStdRefNum` returns the reference number of the last `Open` call to standard in, out or error (prefixes 10, 11 and 12) so others can know which reference number to use in these instances. `GetRefNum` returns the reference number for an open file, while `GetRefInfo` returns the pathname and access information on an open file given the reference number. The new `DRename` call lets you rename a device (provided the device is renameable, naturally). The system preferences have been expanded to allow applications to suppress error dialogs (those with only one button, such as "Volume /XXX may be damaged") or to request that volume mount dialogs do not have the "cancel" button. This is now used by the Loader to load dynamic segments, since pressing "cancel" would cause GS/OS to return error $45 (Volume Not Found) to the Loader, which is always fatal when trying to load dynamic segments. GS/OS nos uses prefix 8 for partial pathnames if prefix 0 is null (useful since prefix 0 can't be longer than 64 characters). `Format` and `Erase` no longer let you destroy data by wiping out a volume on which there are currently open files.

A new symbolic prefix has been added. GS/OS sets the "@" prefix at application launch time. If the application is launched from a server, the prefix points to the user folder on the server. If not launched from a server, this prefix is the same as prefix 1: or prefix 9:—the same directory the application was launched from. This provides a convenient and standard location for configuration files. GS/OS also checks the auxiliary type of a program at launch time to see if it has identified itself as "GS/OS aware." If it has not, GS/OS displays a warning when the application is launched from a folder whose path is more than 64 characters long, since older programs using only class zero calls can not access such pathnames. The user may cancel the launch at this point.

The `option_list` has been clarified and enhanced as well. This list of FST-specific information is available as input or output from many GS/OS file calls, including `Open`, `GetDirEntry`, `GetFileInfo` and `SetFileInfo`. In the calls, your application passes a pointer to an `option_list` buffer. The buffer starts with a word describing the length of the buffer, followed by a word result space where GS/OS tells you how much information was placed in the buffer. Following that is the addition of a word file system ID (`file_sys_ID`) which tells you the file system of the file in question. Following the file system ID is file-system specific information that is important to the host file system but not to GS/OS. For example, AppleShare keeps Finder Info (for the Macintosh Finder) here, as well as access privileges and parent directory IDs. This information is important to non-Apple II users of AppleShare, so it should be copied with the file. By placing it in the `option_list`, the AppleShare FST (and GS/OS overall) provides a convenient and easy way to do it. Since most file copy operations use `GetFileInfo` and `SetFileInfo` to set the attributes of the copy of the file correctly, simply including the `option_list` in these calls assures it will be copied. The FSTs are smart enough not to use the data in the list if the file system ID indicates the source and destination file systems are different. (For example, passing an AppleShare `option_list` to ProDOS does no harm—the ProDOS FST knows not to try to interpret the information as ProDOS-specific attributes since the file system ID says "AppleShare".)

GS/OS has always allowed ProDOS 8 applications to be launched off ProDOS disks, and loads and switches to ProDOS 8 for you. This now happens as well when ProDOS 8 applications are launched from AppleShare. (This can happen since the PFI part of AppleShare allows ProDOS 8 and AppleShare to co-exist. This would be impossible for other filing systems such as DOS 3.3 or MS-DOS, since ProDOS 8 can't read files

from those disks as it can from AppleShare.) The switching of operating systems is even faster. When ProDOS 8 is launched, GS/OS now goes into a dormant state while staying in memory. Under 4.0, GS/OS disposed of itself and reloaded it from disk when it was time to return. If memory is not available to keep GS/OS around when ProDOS 8 is launched, GS/OS is disposed and you have to reboot to get back to it. Since GS/OS is now in memory instead of being reloaded, switching back to GS/OS from ProDOS 8 takes somewhere from three to six seconds. Older drivers which have not been made restartable will have to be reloaded from disk, but most of Apple's drivers provided with 5.0 are restartable. Examples of non-restartable drivers include the Apple 5.25" disk driver and most third-party drivers. Even so, the time to reload those few drivers is trivial compared to the rest of the operating system, and switching is indeed a joy compared to 4.0. Because GS/OS is in memory, ProDOS 8 applications which use IIGS extended memory will find less of it available, but the speed in going back and forth is well worth it.

The ProDOS FST has undergone some changes as well for 5.0. Handling of updated blocks in memory has been improved to handle files which will remain open all the time (such as the System Resource file, discussed later). The FST can now add a resource fork to an existing file (part of the new GS/OS capability to do this as well), and will only display the "Volume /XXX may be damaged" error once. The ProDOS FST also uses the Cache Manager more often for improved performance, and has added the idea of a "volume modification date" to know when a volume has changed. But most noticeably, the ProDOS FST allows lower case letters in file names. Two bytes of the ProDOS directory entry have been redefined to be character bits, indicating the case of the file name (the actual file name itself is still stored entirely in upper-case for ProDOS 8's benefit). More information on the details of how this works can be found in Apple's GS/OS Technical Note #8, "Filenames With More Than CAPS and Numerals".

And the Cache Manager itself uses a new toolbox feature so that when an out of memory condition is near, the cache is flushed and released, giving the memory back to the application. Under 4.0, if a user had unwisely used a 256K disk cache that had gotten full, the only way to release the memory was to make the `ResetCache` call or reboot the system. Under 5.0, it's handled automatically.

The Toolbox

Extensive though they are, the GS/OS changes for 5.0 are minor compared to the changes in the Apple IIGS Toolbox. Two new tools have been added and most of the tools have undergone profound changes, from adding support for resources to almost-automatic extensions of the user interface.

The Resource Manager

The Macintosh™ has an architecture where discrete, often small amounts of data are kept in individual chunks of a file called **resources**. Every file on the Macintosh is divided into two logically equal parts, called **forks**. The **data fork** is the "normal" part of the file, and is the part of the file IIGS owners have always dealt with. It contains the file's normal data. For example, the data fork of a text file would contain ASCII text. The other fork is the **resource fork**, and it contains resources. Resources typically contain definitions for dialog boxes, or windows, strings to be displayed, segments of code, or the like. This separation of data from program code makes it immune to changes in the program code—even to changes in the programming language itself. The program could go from Pascal to C to assembly, and the data in the resources would remain the same.

Although dividing portions of the file into standard, discrete segments is a good idea, more than that is needed. If resources were nothing but a file format, every

application would have to contain a fair amount of code just to access the individual resources. What makes them valuable is a part of the System Software—new to the IIGS with 5.0—called the **Resource Manager.**

On the IIGS, resources are identified by a two-byte resource **type** and a four-byte resource **ID.** The type indicates the generic kind of information in the resource, and the ID differentiates a particular resource from any other with the same type. It's similar to a file's file type and auxiliary type, except only one resource with any particular type and ID combination may exist in a file. Some types are defined by the System as standard resource types, which will contain data in a standard format. Other types are left to each application, and their format will vary from program to program. Resource types from $8000 to $FFFF are reserved for standard definitions and should not be defined by applications; types from $0001 to $7FFF are reserved for applications and will not be defined by the System. IDs have a similar restriction. Application-definable resource IDs range from $00000001 to $07FEFFFF. The range from $07FF0000 to $07FFFFFF is reserved for the System, and other values are invalid.

The Resource Manager manipulates **resource files,** which correspond to GS/OS resource forks. It opens them when asked, reads resources from the file, writes them to the file when needed and closes the files. The only time your application need not use the Resource Manager to manipulate resource forks is when you're just copying them and don't care about the format of the information they contain. If your application wants a specific resource from the file, always use the Resource Manager to access it.

Resources have attributes similar to Memory Manager attributes, determining how the resource can be used. Resources have purge levels, can be locked, fixed, may be restricted so not to load in special memory or cross bank boundaries, and may be page-aligned. They may also indicate that they should be **pre-loaded** (meaning the Resource Manager should load the resource when the resource file is opened, and not wait for someone to specifically load it), that they are **protected** (and should not be written to disk), **changed** (indicating the resource has changed and should be rewritten to disk, unless protected), **absolutely loaded** (loaded at a specific memory location) or needs a **resource converter** (a special routine provided by your application that converts resource from a disk format into a memory format). Resource converters are powerful tools. The System provides a code resource converter so that code resources can be converted from OMF format on disk to relocated code in memory. Another example might be an Apple Preferred format picture converter, which would take a resource containing an Apple Preferred format picture on disk and leave it as a pixel map in memory. The converter would also take a pixel map as input and write it to disk as an Apple Preferred format resource. Note that the code resource converter does not convert code which has been loaded back into OMF before writing it disk, so don't try to write a code resource that you've already loaded! (The Resource Manager will write a resource to disk automatically when you close the file if the resource has been marked as **changed**, so don't do that to code resources.)

The Resource Manager is implemented as a permanent initialization file named `Resource.Mgr`; it must be present or 5.0 will not boot. Although it is called through the Tool Locator, this implementation allows it to be available to the System even if the application has not started it up. The System uses resources in several instances as well; the code to handle a new type of control (Icon Button controls) and Event Manager translation tables, as well as other system resources are contained in a **system resource file** named `Sys.Resources`. (This is different from the Macintosh, where most of the System is contained in one file called `System`, including system resources.) The `Sys.Resources` file is always open on the boot disk, regardless of whether an application has opened it or not. It will not be closed by a `Close` call with reference number zero, although resource files your application opens (including the resource fork of your program file) will be closed by such a procedure, making it a nice thing to avoid.

When the Resource Manager is asked to load a resource, it first checks to see if the resource is already in memory. If so, the handle to the already-loaded resource is returned. If not, it is loaded and the handle to the just-loaded resource is returned. The Resource Manager owns the handles to resources; your application should not do anything but read from them, lock them or unlock them. If you need to do other things to the handle, you may take the handle from the Resource Manager with the `DetachResource` tool call. Resources may be written to disk, or simply marked as changed, in which case the Resource Manager will automatically update them on disk when the resource file is closed.

The Resource Manager becomes an instantly-useful part of the System since several of the tools have been changed to use resources. For example, instead of passing a template in memory to the Window Manager to create a new window, you can pass a resource type and ID to the Window Manager's `NewWindow2` call. The Window Manager then loads the resource and creates the window from it. As graphical resource editors become available, you will be able to design these parts of their user interface by drawing them, rather than by specifying coordinates and "fine-tuning" them one slow recompile after another. Once resources are set, they never have to be recompiled (unless the programmer desires it), saving compile time. They can make your program's memory management more flexible and reduce your programming time. In the sample program discussed later, nearly all the static data in the program is provided in resources, built through the resource compiler "Rez", soon to be part of APW and MPW IIGS.

TextEdit

Although the Resource Manager is a very important tool, it can be a little difficult to describe. TextEdit is conceptually easier, so the description will take considerably less space.

TextEdit implements text-processing primitives in the desktop interface with very little work on the part of the application. TextEdit includes vertical scrolling of text (with a scroll bar), word-wrap, multiple tab types (so when tabbed to, text may be left-aligned, center-aligned or right-aligned), ruler-based formatting similar to MacWrite (TextEdit currently supports only one ruler), intelligent cut and paste, four justification styles (left, right, center and full) and multiple styles, fonts and colors. TextEdit can work with large blocks of text, up to the limits of available memory. Text is selected through standard mouse movements, clicks and drags, as well as through standard Apple II keyboard equivalents (including Control-Y as delete to end-of-line).

To use TextEdit in your programs is almost trivial. The Fidgeter program in this article takes the most simple route, which is implementing a TextEdit field as a control. With new extensions to TaskMaster, most of the work involved even for complex, multi-part controls like TextEdit can be handled for you. TextEdit provides calls to get and set the text in the field, to get and set the offsets to the current selection, or to get information about the text in the field. It will also convert points in local coordinates to text offsets (and back), scroll to specified locations, handle standard editing functions (cut, copy, paste and clear), and draw the text as a pixel map into a specified port, which is useful for printing. Advanced routines are available to further customize TextEdit for specific purposes, although most people will never need to do this.

In most cases, applications simply want a way for the user to enter some text that could be longer than 255 characters long (the limit of LineEdit), or to contain large amounts of text in a small location (LineEdit fields don't scroll). TextEdit handles this with remarkable ease for the application. In the Fidgeter program, a TextEdit control is used. It supports cut, copy, paste and clear, as well as standard text editing (typing, deleting, moving text around) with absolutely no TextEdit-specific calls done by Fidgeter. TaskMaster and TextEdit do all the work. You may use TextEdit as TextEdit fields (as opposed to controls), if you so desire. In this case TaskMaster and TextEdit do not do

most of the work for you, but you gain some more flexibility over the way actions are handled.

The best way to get the feel for what TextEdit will and won't do for you automatically is to try it out. Fidgeter uses standard actions in its two TextEdit fields. Extensions (such as retrieving the text edited, changing styles or custom editing) can be implemented through TextEdit calls on the TextEdit control. Apple has a sample program called "Teach" which demonstrates more of these capabilities in a very simple word-processing type program.

The other tools

Just because there are only (!) two new tools included with 5.0, don't think the changes to the toolbox are minor. Many of the tools have undergone significant transformation—in fact, preliminary toolbox delta documentation is several hundred pages long!

The tool dispatcher itself is a little faster, having had a few cycles squeezed out of each call. A new capability has been added to work with the Message Center—`MessageByName` allows you to manipulate messages based on an ASCII string of your choice; the call maps that into a message number you can then use. This means anyone can use Message Center without having to get a Message number assignment from Apple.

But the most significant change to the Tool Locator is the addition of `StartUpTools` and `ShutDownTools`. The first of these calls takes a reference to a table largely formatted like the table used as input to `LoadTools`, and starts up all those tools, in the proper order, allocating direct page memory for those that need it. Code to start up all the necessary tools used to take several pages of source code in assembly — now it takes less than one screen. High-level language tool startup used to take a couple of screens—now it takes about three lines. You start the Tool Locator, the Memory Manager, and tell the Tool Locator which mode to use (320 or 640 mode) and what tools to start—it takes care of the rest. Fidgeter contains examples of these calls.

The Memory Manager has better performance. When allocating non-fixed handles, the Memory Manager remembers where the last handle was allocated and starts looking immediately following that handle, instead of at the beginning of the list each time. This increases memory fragmentation but dramatically affects the speed of `NewHandle`, which is often a bottleneck operation in both applications and in QuickDraw. An "out of memory queue" has been added as well—routines can install themselves in this list, and the Memory Manager will call them when it is unable to allocate a handle. Each routine can be called twice—once when the Memory Manager was immediately unable to allocate a handle without purging or compacting memory, and a second time when the Memory Manager was unable to allocate the memory even after purging and compaction. The Cache Manager in GS/OS uses this feature to release the cache memory when memory is critically low. This scheme allows your applications to create much more complex memory management techniques than the Memory Manager's built-in three purge levels can handle (no pun intended. Really).

The Desk Manager now limits the number of CDAs and NDAs only to available memory—both the CDA and NDA menus scroll if more desk accessories than fit on one screen are present. Also added are ways to remove CDAs and NDAs from the Desk Manager's lists, and for applications or anyone else to add themselves to the Desk Manager's "run queue"—the list of routines which get "run events" like NDAs.

The Event Manager now has key translation features. This allows a user to type a key combination like "option-e-e" and get a character like "é". The default translation is the same as on the Macintosh, so the keystrokes already familiar to many now work on both machines. The Event Manager has new calls to allow the translation to be set to default, none, or custom (using a key translation table you provide).

The Window Manager has all kinds of new goodies. TaskMaster has been extended to handle controls in windows. For standard controls (the list of which has also been extended, as will be described shortly), TaskMaster can handle almost all the work. It flashes simple buttons, handles keys for TextEdit and LineEdit controls (both of which are new), checks and unchecks check boxes, selects and deselects radio buttons, tabs between "target" controls (there's a lot of new Control Manager stuff, as you're seeing!), tracks menus—and tells you when it was done that the user selected a control, then gives you the ID of the control selected. Pretty nifty. In fact, with one exception, TaskMaster does all the work in Fidgeter, as you'll see when we explain that program in more detail. All this support has been added for Desk Accessories as well—the new TaskMasterDA call allows NDAs to use TaskMaster features for the first time. (This was previously impossible since TaskMaster assumes things about the system that NDAs don't have the right to assume—NDAs written with TaskMaster often had severe problems when opened under applications that don't use TaskMaster.)

The Window Manager has support for resources, so a new window template may be kept in a resource which the Window Manager will automatically load and use. AlertWindow also takes resources, and the code that compiles the alert strings, making the necessary substitutions, has been broken out into a separate call, CompileText, for your application to use if it desires. A new ErrorWindow call takes standard GS/OS errors (and more) and translates them into simple dialogs presenting real English messages to the user. The Window Manager now uses the default desktop pattern message previously used only by the Finder (but not previously documented). A utility creating this message could provide a custom desktop for every application until reboot with the greatest of ease.

The Menu Manager supports resources through a new **menu template** format. Menus can be defined in this way instead of as ASCII strings, making the job of graphical menu resource editors easier. You can create menus, menu items and menu bars from resources (although if you use a menu bar resource, a strange anomaly in 5.0 forces you to use a resource ID with a high word of zero, or the call will not work). The Menu Manager also allows outlined and shadowed menu items, as well as the creation of empty menus (to get an empty menu previously, you had to create a menu with one item and delete the item). And, very noticeably, menus with more items than fit on the screen automatically scroll (as long as more than three items in the menu fit on the screen, usually only a concern for menus not in the menu bar). This includes the Apple menu (which contains New Desk Accessories) and a font menu created with FixFontMenu.

The biggest new Menu Manager addition for programmers is **pop-up menus**. Pop-up menus are little rectangles on the screen that look a lot like simple rectangular buttons with drop-shadows. However, clicking in them makes a real menu "pop up" around the cursor, with the previous selection remaining directly under the cursor. The Menu Manager tracks the menu just as if it were pulled down instead of popped up. When a selection is made (if one is made), it becomes the new item listed in the rectangle.

Pop-up menus can be implemented two ways, like TextEdit fields—as controls or as themselves. If they are controls (as is the pop-up menu in Fidgeter), TaskMaster and the Control and Menu Managers do most of the work for you. As pop-up menus, you have more work to do (detecting when someone clicks in the menu, highlighting the title, calling PopUpMenuSelect, etc.), but you also gain more flexibility. The example in Fidgeter is implemented as a control, and shows a pop-up font menu (using a little trickery).

No tool that existed on 4.0 shows more change than does the Control Manager. Going far beyond adding support for creating controls from Resources, the Control Manager works with the new TaskMaster extensions to create new control records we refer to as **super controls**. Super controls are created by the NewControl2 call, and

have an extra word of flags (known as the `moreFlags` field, naturally) and a control ID, assignable by the application. Although the ID and the RefCon are both four-byte fields under the control of the application, the ID is intended to be a compile-time assignment of a value so you can write code that knows minimally which controls were selected (`TaskMaster` returns the control ID in the extended Task record when a control is selected). Also, calls are provided to transfer a control ID into a control handle and vice-versa to make such code easier to write.

There are seven more types of standard super controls than regular controls. **Static Text** controls display text in a rectangle supplied by the control creator. The text is drawn with `LETextBox2`, so all the formatting possible with that tool is possible with Static Text controls. The text can't be edited by the user. **Picture** controls draw a QuickDraw picture in a rectangle you specify, scaling it as necessary. **LineEdit** controls allow you to have line edit items as controls; **TextEdit** controls are similar. **Pop-up** controls are pop-up menus implemented as super controls, and **Icon Button** controls are simple buttons that contain icons, with or without titles. Fidgeter contains examples of each of these controls.

Standard controls may have keystroke equivalents—you can set it up so that a simple button is selected when the user types a given key (such as "return"). You also have control over modifiers, so you can specify which modifiers must be pressed and which ones can't be pressed. One of the radio buttons in Fidgeter has the key equivalent of "6", but will only recognize it if the key is pressed on the keypad and not on the main keyboard.

To have keystroke equivalents and handle things like LineEdit and TextEdit, you may have guessed that super controls may also accept events. Controls like TextEdit controls accept not only keystrokes, but mouse clicks and drags, and even menu selections—if you use standard edit menu numbers, TextEdit controls will handle cut, copy, paste and clear for you. Other controls accept events as warranted. Super controls also introduce the notion of a **target** control; the target control is the active recipient of user keystrokes. In a window with more than one control capable of taking keystrokes (the Fidgeter window is such a window; it contains two TextEdit controls and two LineEdit controls), one of them has to be the currently "active" recipient of keystrokes. It wouldn't do at all for one key to be seen in all the editable items. The Control Manager has calls to find and change the current target control, and LineEdit and TextEdit controls can allow you to set it up so pressing "tab" changes to the next target control.

Another new Control Manager feature is reciprocated in the Window Manager. Calling `SizeWindow` now sends a message to super controls that the window size has changed, and you can create grow box controls so that they automatically call `SizeWindow`.

QuickDraw II itself has been sped up considerably. `QDStartUp` offers two new options to allow QuickDraw to operate even faster if your application will allow it. One new feature to QuickDraw is to use hardware shadowing if memory in bank $01 is available. Since hardware shadowing allows reads and writes to take place to fast memory instead of slow memory, operations go faster. The other new option tells QuickDraw that you are not going to change any of the fields in a GrafPort except through standard QuickDraw calls. This allows QuickDraw to skip a fairly lengthy set-up process on many calls, which also increases speed.

QuickDraw Auxiliary has two new routines—`SeedFill` and `CalcMask`. `SeedFill` takes a specified pixel map and and a starting point and "fills" the shape around the starting point, much like a paint bucket tool would do in an art tool. `CalcMask` generates a mask from a specified pixel map and a given pattern by filling in from the boundary rectangle, an operation useful in implementing "lasso"-like tools. Each one is loaded with options for flexibility. QuickDraw Auxiliary also now allows full text justification in pictures.

The Print Manager has added calls to allow you to get and set the name of a document (mainly useful when printing over the AppleTalk network). You can also get more information about the current printer, and can get and set the page orientation of the current document. A new driver structure is provided to allow new drivers to add the new calls, but older drivers still work properly. Changes in driver structure will be documented in Apple IIGS Technical Note #35 and #36. Also, the `PrChoosePrinter` call is no longer supported—the Control Panel New Desk Accessory should be used to select printers. Making `PrChoosePrinter` results in a dialog asking you to use the Control Panel NDA to select a printer.

Standard File has been completely rewritten. The new version uses class one GS/OS calls and is fully network-aware. It features new calls that allow filter procedures to access GS/OS `GetDirEntry` records instead of synthesized ProDOS directory entries, and a new **multi-get** mode that allows one standard file "open" dialog box to open many files. More flexibility for the application is present, although some of the features may not be fully implemented on this release.

The Font Manager and QuickDraw have teamed up to support fonts with greater than 64K font strikes. Fonts up to 255 points may be created by the toolbox,although you should be very aware that large fonts take large amounts of memory—the font strike grows by both height and width for each character. In a simplified example, a 12 point font that requires 10K of memory may be scaled into a 24 point font that will require 40K of memory (double the width and the height means the font strike grows by a factor of four). So if a 48 point font takes 50K, a 96 point size of the same font will probably take around 200K! A new font format (version 1.5) is defined to allow larger fonts to exist, and the Font Manager and QuickDraw deal just fine with fonts of the older format.

The List Manager is extended to handle list controls (one of the new types of super controls). Several List Manager calls have been added to work with list control handles instead of list records. This means that for many normal purposes, applications won't have to carry around list records just as input to List Manager calls. Other new calls allow manipulation by item number rather than the less-useful list record pointer. The List Manager now also allows applications to control whether the scroll bar is created inside or outside the bounding rectangle.

Other toolsets have undergone minor changes as well. The Sound Tools have new calls to allow the setting up and the playing of sounds, as done by `FFStartSound`, to be split into two separate stages for application flexibility. There are also new calls to read and set DOC registers more easily. The Text Tools saw minor changes to make them less Slot Arbiter unfriendly (for when dynamic slot arbitration is available), but they're still pretty unfriendly and should be avoided when writing new text-based applications. And the VideoMix toolset (new to the System Software release) is updated to a 1.1 version.

Finder, Applications and Utilities

Although Finder changes normally don't affect most applications, most programmers will be pleased to see the changes implemented. First and foremost, the Finder is fully AppleShare Aware. Server volumes are dynamically updated, access privileges are respected (and changeable if you own the folder) and network-friendly programming practices were used. The overall toolbox and OS changes greatly increase Finder speed, and more enhancements were made to the Finder code itself for even more speed.

The Finder now looks for the strings to display as a file's "kind" in disk files referred to as **File Type Descriptors**. These files have a file type of $42 and are of public format, so any application can use these files to know what string to display to identify a particular file by file type and auxiliary type. Multiple files are supported by priority, so developers can supply their own description strings for the Finder and for other

programs which use this scheme. The file format is documented in Apple II File Type Note for file type $42.

The **Control Panel NDA** is new for 5.0. Through a series of small program definitions called "CDevs" (after the Macintosh Control Panel's four-character file type CDEV), the Control Panel NDA acts as a shell for a lot of minature programs that affect the operating parameters of the computer. The Control Panel NDA serves as a combination of the Macintosh Control Panel and Chooser desk accessories. System Software 5.0 comes with sixteen CDevs, not all of which will be installed on every system since many of them relate to network activities. The **Alphabet** CDev allows selection of display and keyboard languages, as well as key translation control. The **AppleShare** CDev allows you to select and log onto AppleShare file servers, providing for automatic log-on at boot time with or without requiring you to type in your password. **AT IWriter** and **AT IWriterLQ** allow selection of ImageWriter and ImageWriter LQ printers over the AppleTalk network. **DC Printer** lets you choose a directly-connected printer (what most of us have). **General** is not nearly as crammed full of stuff as on the Macintosh Control Panel, since the IIGS Control Panel NDA generally has a separate CDev for each category listed in the text-based Control Panel CDA (which is still fully functional). **Keyboard** sets key repeat and delay rates; **LaserWriter** lets you select LaserWriter printers over AppleTalk; **Printer Port** and **Modem Port** allow setting the parameters for either of those serial ports; **Monitor** sets display colors, 40 vs. 80 column defaults and color vs. monochrome graphics; **Mouse** controls mouse behavior (surprisingly enough); **RAM** gives scroll bars to set the RAM Disk and GS/OS Disk Cache sizes (the old "Disk Cache" NDA has gone away); **Slots** lets you change slot settings and assignments; **Sound** controls system volume and pitch; and **Time** sets the system clock and controls how the time is returned by the toolbox (wanna see a pop-up menu with 60 entries, one for each second you could set the clock to? This is the place!).

The Advanced Disk Utility now allows up to 32 partitions per SCSI Hard Disk, as this functionality is present in the GS/OS SCSI Manager. The Installer allows multiple script selection (continuously with shift-clicking, or discontinuously with apple-clicking), has better error reporting and can install over the network. It also allows installation on the boot volume (unlike the Macintosh Installer), but forces a reboot when done so the System can't confuse itself. CD Remote (included with Apple SC CD Compact Disk ROM drives) has been updated for the new System Software as well.

You'll find most of the user-visible features of 5.0 just by playing around with your programs and applications after installing the new system. However, you won't find the programmer-specific changes—and there are plenty of them—until you start using them in your own programs. And what better way to get the general idea than with a simple programmatic example of how the new world works?

Fidgeter—a simple program using 5.0 features

Fidgeter is a simple sample application that demonstrates one of each kind of the new super controls, along with a couple of interface tricks and the general use of resources in tool calls. It doesn't do anything incredibly useful—you can check boxes and select radio buttons and type text and pop menus, but it doesn't take action on what you do (except in the case of standard editing functions, as will be explained below). Nearly all of the static data is provided through resources, and since most of the program is the controls and their definitions, the resource source code is nearly as large as the program code itself.

The program will look familiar to many of you—it's based on a new revision of the Apple IIGS Source Code Sampler "Shell" program. The revision used here is 1.1ß2, obviously not final. The Shell has been revised to start more tools (friendlier to desk

accessories), be cosmetically more pleasing, and to use all its data from resources where possible. All of the Shell's resources have ID in the $07FExxxx range, with the exception of the menu bar resource (as noted earlier, menu bar resources currently must have IDs in the $0000xxxx range or they will not work with `NewMenuBar2`). The modularity associated with the Shell is maintained, and the style will be familiar to those who have worked with or studied the Source Code Sampler. Fidgeter is built on the Shell, but there are some properties of the resulting program that are present because I put them in Fidgeter, and some are present because of the Shell. Unless they need distinction, I'll just refer to everything as "Fidgeter".

Fidgeter uses only standard resource types. That means there are no application-defined resource types present; all the resources are of public format. With one exception, the toolbox loads all the resources needed by the program without direct Resource Manager calls from Fidgeter or the Shell. For example, to create the menu bar and all the menus within it, the Shell passes the ID of a MenuBar resource to `NewMenuBar2` along with a descriptor saying to use resources. `NewMenuBar2` knows the standard resource type for a MenuBar resource ($8008), so it takes the ID passed to it by the Shell and loads a resource with that ID and type $8008, and uses that resource to describe the menu bar. Each reference to an individual menu within that menu bar is a resource ID as well, which `NewMenuBar2` passes to `NewMenu2`, and it also loads resources of that ID with the type of a menu resource ($8009). `NewMenu2`, in turn, takes resource IDs of new menu items (type $800A) and loads them with the given IDs. The menu items then reference Pascal String resources (type $8006) for the string data displayed as the menu item text. The strings, menu items, menus and the menu bar are all linked together through resource IDs in a way totally transparent to the Shell—it's all determined by the resources themselves. In one instance, Fidgeter does specifically load a resource so it can work with the data in that one resource.

About Super Controls

Super Controls are created from **control templates**. Each template has a count of the number of parameters, followed by the control's four-byte application-assigned **ID** and the control's boundary rectangle. After that are a word of flags, a word of more flags and the four-byte `refcon`. The remaining parameters depend on which kind of control is being created. Figure 1 shows the control templates for some of the standard controls. To save a little space, I've not drawn the templates for Icon Button, Line Edit, Text Edit, List or Pop-Up controls, but their format is very similar, and it's the format that's the point.

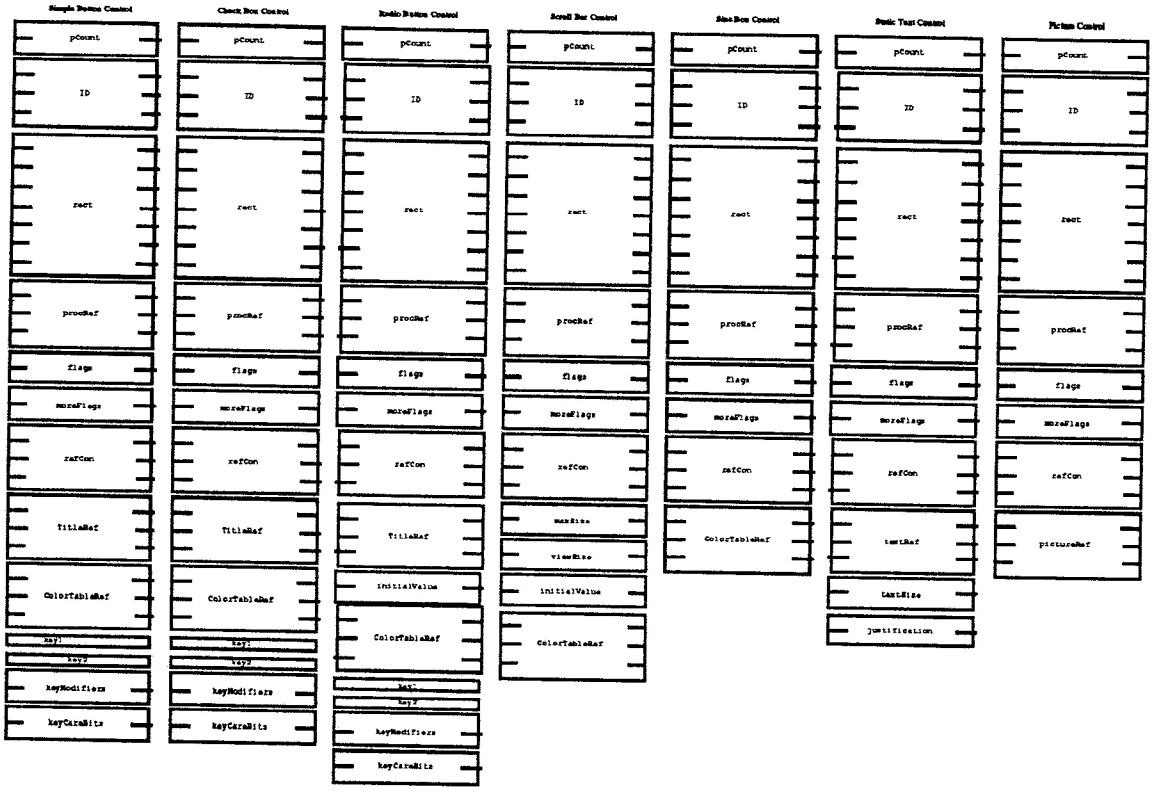

Figure 1—Obnoxious Diagram

As you can see, each template starts with the same header, and then progresses to the specific parameters for the control type. Standard controls are determined by the value of the `procRef` field. Controls handled by the toolbox have a `procRef` with a non-zero high-order byte (for example, the `procRef` value for a size box control is $88000000, and a scroll bar control is $86000000). A bit in the `moreFlags` word tells the Control Manager that the `procRef` field is not a pointer to a custom procedure, and is always set for standard Super Controls. You can see the values for the other standard controls in the Fidgeter resource source code.

The `flag` word is used to control the `ctlHilite` and `ctlFlag` bytes in the control record as the control is created. The high-order byte is used for `ctlHilite`; the low byte is `ctlFlag`. These values work just like described in the Toolbox Reference manuals.

The `moreFlags` word is new for Super Controls. The Control Manager uses the high byte for standard flags; the low byte is defined by the control being created. The standard flags are as follows:

`fCtlTarget`	bit 15	If set, this control is the current target control and is the active recipient of keystrokes.
`fCtlCanBeTarget`	bit 14	If set, this control can be the target control.
`fCtlWantsEvents`	bit 13	If set, the new Control Manager call `SendEventToCtl` may send events to this control. A control that can be the target control will always get events, even if this flag is not set.

`fCtlProcRefNotPtr`	bit 12	This bit, when set, tells the Control Manager the `procRef` field does **not** point to a custom control definition procedure.
`fCtlTellAboutSize`	bit 11	If this bit is set, the control will be notified when the window it's in has changed size.
`fCtlIsMultiPart`	bit 10	If the control has more than one part (like a list or TextEdit field with a scroll bar or grow box), this bit must be set. It helps the Control Manager hide multi-part controls (since Control Manager doesn't always know about all the parts).

The low order byte is used by standard controls (and is therefore suggested for custom controls) to define references as follows:

`titleIsPtr`	$00	The title reference is a pointer.
`titleIsHandle`	$01	The title reference is a handle.
`titleIsResource`	$02	The title reference is a resource ID.
`colorTableIsPtr`	$00	The color table reference is a pointer.
`colorTableIsHandle`	$04	The color table reference is a handle.
`colorTableIsResource`	$08	The color table reference is a resource ID.

Note that the color table references are the same as the title references shifted to the left by two bits.

The options specific to each control are far too numerous for TechAlliance to pay me to go into here. You can see many of them in the control samples for Fidgeter.

About Fidgeter's Controls

Everything in Fidgeter's window is a control, from the text displayed by items (static text controls) to the TextEdit fields. There are standard controls (a simple button, two check boxes and three radio buttons), but each has a key equivalent (the number key included in the control's title). For "Radio Button #6", the key equivalent is the "6" key, but the key equivalent format lets us be more specific than that. Not only does it give us two key equivalents (usually upper and lower case versions of the same character), but it also allows us to specify which modifiers must be set, but also which modifiers must not be set. The equivalence test takes the modifiers word from the keypress and ANDs it with the value found in `KeyCareBits`, then compares it against `KeyModifiers`. If the comparison doesn't return the two values to be equal, the Control Manager says there was no key equivalent pressed. This allows us in Fidgeter to say that Radio Button #6 has the key equivalent of "6" (or "^") as long as the keypad bit is set, but the "option" bit is **not** set. Try it for yourself—"6" works unless you press the "option" key.

Key equivalents are obviously a good thing, but there are a couple of things of which you should be aware. First, key translation is probably in effect. Having a key equivalent of "option-d" for something probably isn't a good idea, because if translation is in effect (and it usually is), you won't see "option-d". The Event Manager will translate it to "∂"—but if you use "∂" as a key equivalent and translation is off, you'll see "option-d." Second, if you have editable fields (like LineEdit or TextEdit fields, keys that are equivalents (like the number keys for Fidgeter's controls) will **not** get through to those items, so you've effectively limited the user's typing. The Icon Button controls are set up to take key equivalents in their templates, but they're turned off by clearing the bit that says

the control wants events. If you turn them back on(**OK?**), notice that the letters used to activate the buttons can't be typed in the LineEdit or TextEdit fields.

There are two TextEdit fields—one is a read-only field and one is fully editable. The TextEdit control and TaskMaster handle standard edit menu choices for us since we use one of TaskMaster's new features—send menu events to controls.

The pop-up menu is actually a font menu. This is done by calling `FixFontMenu` on the pop-up control after it's created. Note that the current menu bar has to be set for `FixFontMenu` to work, so we set the menu bar to the pop-up (each pop-up is a miniature menu bar) first.

The tricky part to this is the size of the control rectangle. What we specify as the control rectangle in the resource control template is what the Control Manager uses as the rectangle for the "unpopped" menu selection. Although calling `CalcMenuSize` on the new font pop-up menu correctly sizes the menu, nobody redraws the rectangle. However, the Menu Manager has calculated the proper size for us, so we take advantage of it. We get the control handle from the pop-up control ID (assigned by us when we wrote the program), dereference it and read the lower right corner of the correctly-calculated menu rectangle out of the control record. It's **highly** illegal to change values in the control record, so to use this new knowledge we have to dispose of the pop-up and recreate it. We load the resource containing the pop-up control template, call `DetachResource` to tell the Resource Manager we're going to take over the handle, and replace the control rectangle's lower right point with the one calculated by the Menu Manager (offset by a few pixels in each direction to account for the drop shadow). We then use that as input to a `NewControl2` call, and the pop-up is recreated. The window is then made visible (it was created invisible so no one could see all this chicanery going on).

You'll notice the `NewControl2` call to recreate the pop-up is the only `NewControl2` call in the program. Others aren't needed—one of the features of `NewWindow2` (which we call to create our window from a window template resource) is the capability to link a control template list to the window template. If one is there, the Window Manager will call `NewControl2` to create all our controls for us. Everything in the window (except the pop-up font menu, as we handle that specially) is created by the one `NewWindow2` call.

Below the pop-up control are two LineEdit controls. One shows a new feature of LineEdit—**password fields**. No matter what character you type, the "*" character appears, as a form of visual protection. The AppleShare CDev uses this so that people looking over your shoulder don't see your password for file servers on the screen as you type it. The other Line Edit field is a standard one, without the password protection feature.

Under those are the Icon Button controls. These are icons, with strings centered under them (as titles), framed as a rectangular simple button would. You have your choice of frames, as you do with simple buttons. They're an easy way to get a custom graphical button.

Below that is a picture control. The picture doesn't look like much (and it isn't; it's a very small portion of a fractal drawn on the IIGS) because a complicated picture would add even more time to that already required to type in the program. It is a QuickDraw picture, and the rectangle specified in the template is not the same size as the original picture rectangle, so true to form, the picture is scaled to fit the new rectangle. You might try playing with the control rectangle and watching the scaling effect. Or, if you have a file that contains a standard QuickDraw picture (file type $C1, auxiliary type $0001), you can replace the entire resource definition with a Rez command to read the resource from disk. Suppose the file is "/MyDisk/Picture". Replace the resource definition with:

```
read rPicture (0x3015) "/MyDisk/Picture"
```

The Fidgeter Routines

`Main` is the core of the entire program—it calls the initialization routines, the main event loop, the shutdown routines and then quits.

`Globals` contains all our global data. Since most of our data is in resources, there's not a whole lot here.

`InitTools` starts up the tools, using the new `StartUpTools` call. It then creates the menu bar (with `NewMenuBar2`), adds the New Desk Accessories to the Apple menu and draws the menu bar.

`FatalError` is called when we detect an error that we can't recover from. Being a fairly simple example, there isn't much of an error mechanism in Fidgeter. Any place where recovering from an error would be messy, we just branch to `FatalError`. This is not acceptable error handling except in a learning exercise (like this one) where we want our mistakes to jump out at us.

`CloseTools` uses the new `ShutDownTools` call to shut down all the tools we started up. Note how these two new Tool Locator calls have replaced literally pages of source code with just a few lines. The Memory Manager and Tool Locator are then shut down after `ShutDownTools` returns.

`InitApp` does our application-specific setup. In this case, it's creating the Fidgeter window (invisibly), making our pop-up control into a pop-up font menu and showing the window.

`CloseApp` does our application-specific shut down. We don't have any in this case; we only have one window, and the Window Manager closes it for us when it gets shut down by `ShutDownTools`.

`EventLoop` does most of the work in Fidgeter. It first tests to see if the top window is a desk accessory; if so, the Undo item (which Fidgeter doesn't use) is activated in the Edit menu. `TaskMaster` is then called to get an event. `TaskMaster` actually handles most everything that happens since we don't really do anything in this application. We then jump through a table based on the taskCode returned to us. Most of the possibilities are ignored because they're either irrelevant for a program as simple as Fidgeter or because TaskMaster handles them for us. We do handle menu selections, but that's it. Routines are called for clicks in the content region or events in controls, but both routines called in those instances do nothing. Before looping again, we look to see if the Quit item was selected. If so, we return from `EventLoop` to `Main`, which will then shut down everything and quit.

`MenuSelect` dispatches for the one kind of event we do actively handle—menu selections. We turn the item ID into an index and call a subroutine to handle the menu item selected. On return, we unhighlight the selected menu's title and return to `EventLoop`.

`Ignore` is called for most of the possible TaskMaster return values—it ignores them. It just returns. So does `doUpdate`, since TaskMaster handles our updates. `InControls` also returns, since we don't take any specific action on any of the controls. If we actually wanted to call `ChooseFont`, for example, when the "Choose Font" item was selected in the pop-up control, this would be where we dispatch to that code. After the selection of the "Choose Font" pop-up menu item, TaskMaster will send us to `InControls` since the event went to a control.

`DrawContent` is not called by any routine within Fidgeter; instead, it's called by TaskMaster to update our window. Since all we have in our window is controls, it's easy to update it. We just draw the controls with `DrawControls`. Note that we use long addressing to get the window pointer to push, since the direct page and data bank registers aren't guaranteed when TaskMaster calls this routine.

TestTopWindow is called by EventLoop to see if the top window has changed kind. If it has, we call DoSysChange (immediately after it) to activate or deactivate the Undo item. Code to do the entire Edit menu is present, but the deactivate portion of it is commented out.

doQuit is called when the "Quit" menu item is selected. It sets a flag to let EventLoop know it's time to quit, and returns.

doAbout gets control when someone picks "About Fidgeter..." from the Apple menu. It puts up an AlertWindow (with the data in a resource) for our about box.

doCut, doCopy, doPaste and doClear are called when their respective edit menu items are selected. Although TextEdit controls handle standard editing menu selections automatically, LineEdit controls do not. If we want to be able to cut, copy, paste and clear in LineEdit items, we have to do it ourselves. It's not too difficult, but there are a couple of fine points.

First we call IsItLineEdit to determine if the current target control is a LineEdit control. (This routine is described below.) If it comes back with the carry set, it wasn't a LineEdit item, so we just return. If it was, the handle to the LineEdit record (the LERecord) is in the A and X registers, so we push them on the stack to save them. Then we get the current port and save it, since we're going to tinker with it. Next we do a StartDrawing to prepare to change the contents of the window. Remember that cutting or pasting will force a change to the way the LineEdit item looks, and we have to set up the drawing environment before doing that. TaskMaster does it when it handles changes to controls, and so do we. Next, we get a copy of the LERecord off the stack and push it on the top of the stack (the stack-relative offsets are both 7 because the first PHA increased what would have been 5 to 7 again). We're now ready to do the editing work at hand, so we call the appropriate LineEdit tool (LECut, LECopy, LEPaste or LEDelete). If we are cutting or copying, we also make sure the scrap size is non-zero (as required by Apple IIGS Technical Note #59) and if it is, we put it to the desk scrap. If it is zero, we call ZeroScrap to clear the desk scrap. If pasting, we put the desk scrap into the Line Edit scrap earlier, before saving the current port. The rest is easy—we set the origin back to (0,0), we restore the port we previously saved, and return.

IsItLineEdit gets the current target control (Fidgeter knows that there always is one, but returns gracefully if it gets an error) and gets the control handle for it from its ID. The moreFlags field is checked to be sure the fProcRefNotPtr bit is set. If it is, the handle is dereferenced and the procRef field is examined. It must equal $83000000 for this to be a standard LineEdit control. If it is, we know that the LERecord handle is stored in the ctlData field of the control record. We retrieve it in the A and X registers and return to the caller.

Conclusions

There really isn't a conclusion here; just a beginning. Take Fidgeter, this article, System Software 5.0, the reference manuals and your favorite development environment and start playing around. Some simple extensions to Fidgeter are easily possible—change the key equivalents on the controls. Make the font chosen in the pop-up menu be the font used in the TextEdit field. Make the TextEdit fields resizable. Use your imagination; 5.0 makes it a lot easier for the system to handle the interface and for you to be brilliant. And that's what's important after all.

[Sidebar]

I change development tools like some people change shirts, and the one I was "wearing" when I wrote Fidgeter was the MPW IIGS cross-development assembler from Apple. This is the assembler that runs under the Macintosh Programmer's Workshop (MPW) but builds IIGS applications. I think it's a powerful assembler and I like it a lot, and I hope some of you feel the same way. Even if you don't, you'll find it fairly easy to translate the Fidgeter program into APW/ORCA, Merlin, Lisa, or whatever your favorite happens to be. Here are the tips you need:

- DC.*character* means, similar to APW/ORCA, "Define constant". However, the character following the period defines how big the resulting constant is. B means byte, W means word and L means long. The argument of the expression can be any type of data. For example, `DC.B 'This is a string'` is the MPW IIGS equivalent to `DC C'This is a string'`. `DC.L 13` is the equivalent of `DC i4'13'`. `DS` works in a similar way, except it means "define storage". The character after the dot means the same thing, and the argument is how many of them to define. `DC.W 1` means define one word of storage space. Storage space is automatically pre-zeroed.

- `With` is kind of like the APW/ORCA `USING` statement, except that routines that you reference with `With` must have been previously defined. `With Globals` is seen in nearly every Fidgeter routine, and `Globals` is the second segment.

- APW/ORCA's `START/END` combination is replaced with `PROC/ENDP`. Data segments are delineated wtih `RECORD/ENDR`. Also, the entire program must end with an `END` statement.

- The `PRINT` and `EJECT` statements are used to control listing options.

Matt

[Sidebar]

I tried, where possible, to use symbolic definitions in the resources rather than numbers. I'm sure I failed once or twice, but the effort is there.

The reason is that this was all written with preliminary development tools. It worked when I wrote it (months before the magazine went to press), but things can change before final release. I tried to use the symbolic parameters so you could know what the controls were trying to *do*, rather than what numbers go where. For instance, instead of saying "$1000" for `moreFlags` in a control template, I'd say "`fProcRefNotPointer`". If the defines change in the `Types.Rez` file before you see it, those will have to be changed or they won't compile. However, you have a lot better idea of what it's trying to do by reading a label than by reading a number.

Matt

[Sidebar]

The resources not only have to be built (compiled with Rez in this case), they have to make it into the resource fork of the program file. I did this with a handy tool called "duplicate". Duplicate lets you copy both forks of a file (the default), only the data fork (by specifying "`-d`") or only the resource fork (by specifying "`-r`"). If you only copy one fork, the other one is not touched. So I got the assembler and linker to output the program into a file I called "`Fidgeter.DFork`" (the data fork), and I got Rez to output the resources into a file called "`Fidgeter.RFork`" (the resource fork). The rest was easy:

```
Duplicate -d Fidgeter.DFork Fidgeter
Duplicate -r Fidgeter.RFork Fidgeter
```

And since I was using the MPW IIGS system, I added afterward:

```
DuplicateIIgs -y -mac Fidgeter :
```

I eventually got a make file built that would only build the fork whose source code had changed. Once the resources were set, the resource fork of the file wasn't rebuilt unless I changed the Rez source. Not too difficult, and speedy when developing.

Matt