57 HIGH-POWERED SYSTEMS REVIEWED

APRIL 1994

BYTE

THE MAGAZINE OF TECHNOLOGY INTEGRATION

Pournelle's Product Awards

Object-Oriented Databases Come of Age

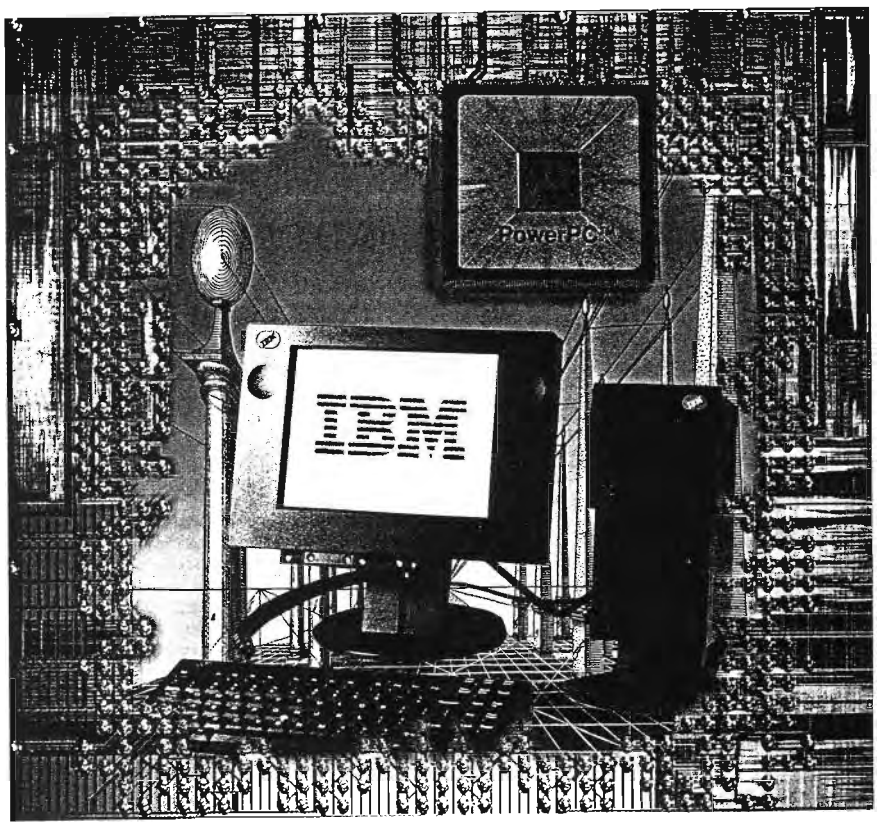
A Nifty Multiprotocol Print Server for Ethernet LANs

First PowerPCs

Apple's Power Macintosh and IBM's Power Personal Systems

EXCLUSIVE

IBM's PowerPC product plans, plus Power Mac vs. Pentium benchmark results



SPECIAL REPORT

CPU Wars: Should you move from CISC to RISC?

PLUS

Intel pushes the 80x86 envelope

02662 04
0 440235 0
$3.50 U.S.A./$4.50 IN CANADA
A McGraw-Hill Publication/0360-5280

UNDER THE HOOD

The Power Mac's Run-Time Architecture

RANDY THELEN

JOHN PATRICK © 1994

If you put a 680x0-based Mac Quadra 800 next to a new PowerPC-based Power Macintosh 8100/80, you might think they were identical except for the nameplates. Glancing at the screens wouldn't help, since the menus, icons, and windows are exactly the same. The applications also look the same; in fact, you could install the same ones on both machines. But if you used both computers for a few minutes, one difference would jump out at you: The Power Macintosh is distinctly faster.

This is just what Apple's software engineers planned. Power Macintoshes maintain 100 percent compatibility with existing Macintosh software. This was accomplished through PowerPC implementations of the Macintosh API, a 68LC040 emulator, a new Mixed Mode Manager, and modifications to the Process Manager. (A *Manager* is a set of related functions that work with a given series of data structures. The Process Manager has routines that manage processes. A *process* is a running application.)

However, backward compatibility wasn't the only goal of the Power Macintosh's operating-system design. While support for existing applications is crucial, the system software was also engineered to support future developments, where powerful new applications will take full advantage of the PowerPC's speed.

In this discussion, I'll take a look at how Apple achieved these two contradictory goals. I will concentrate on the new portions of the design where appropriate, since much of the compatibility issues are covered elsewhere in this issue (see "Emulation: RISC's Secret Weapon" on page 119).

An integration of PowerPC code and 680x0 code yields compatibility and speed while providing new capabilities

Application Structures

I'll start by examining the structure of an existing 680x0 application. (From this point on, I'll use the term *68K* to denote any of the 680x0 processors.) Macintosh files are composed of two structures called *forks*. Each file has a data fork and a resource fork.

Physically, there's no difference between these two types of forks. They're just streams of bytes located somewhere on disk. However, the Mac OS treats them differently. A file's data fork contains data—typically the output from an application, such as text from a word processor or numbers from a spreadsheet. A file's resource fork contains information on the file's creator (this is how the Mac OS knows what application to launch when you double-click on a document), the icon that is displayed on the Desktop, and other information.

For 68K applications, the resource fork also contains program code. When you double-click on a file icon, the Finder summons the Process Manager to start—or *launch*, in Macintosh parlance—the application. The Process Manager then uses a part of the Mac OS called the Segment Loader to read the code resources from this fork into memory.

The 68K Macintosh application code resources are divided up into *code segments* that the Segment Loader loads into and out of memory. Code segments are typically 32 KB in size, because Mac applications use PC-relative (program counter) instructions. Such instructions are used so that code is address independent and capable of being placed anywhere within scarce physical memory. These segments might be used briefly, purged from memory to

The RISC Decision

make room for other code segments, and then reloaded as necessary into another portion of memory.

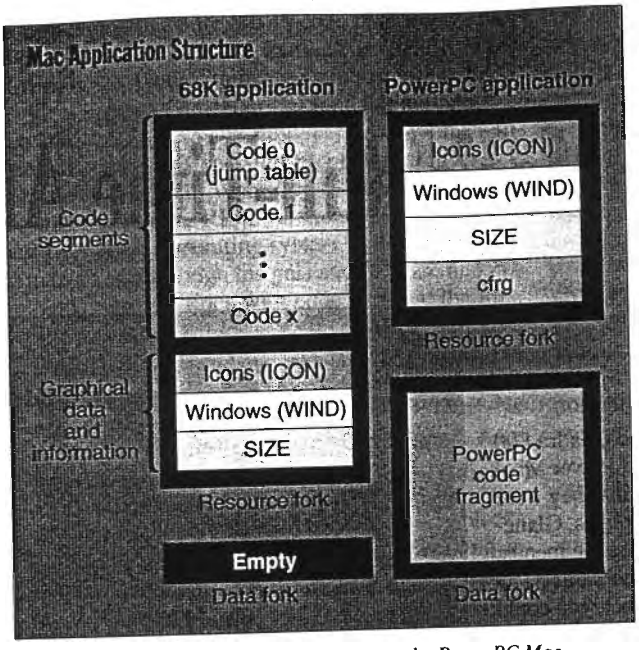
Because the 128-KB Macintosh used a 68000 processor, the offset values of these instructions were limited to 15 bits in size. The sixteenth bit was a sign bit to indicate the direction of the offset (either forward or backward in memory). This limits references to within ±32 KB of the instruction. Subsequent 68K processors had larger offset values, but PC-relative instructions and segments are still being used to implement address-independent code.

The Segment Loader loads code segments on demand as functions within them are called. Essentially, any function call outside of the current code segment is made through a nonpurgeable code block called the *jump table*. If the code block with the called function isn't in memory, its entry in the jump table is actually a call to the Segment Loader. The Segment Loader loads the missing code block into memory and then modifies the corresponding jump-table entry, along with all the jump-table entries associated with that code block.

Instead of acting as calls to the Segment Loader, these jump-table entries have jump instructions to the functions themselves. When the code block is purged from memory (an operation that only the program has control over), the jump-table entries are reset so that they are again calls to the Segment Loader.

The Power Macintoshes use a significantly different design (see the figure "Mac Application Structure"). Applications are a single code fragment (except for imported library functions, which reside in other code fragments). *Code fragments* are the atomic units for libraries and applications in a Power Mac application, and they can be any size.

An entire PowerPC application's code is stored as one continuous unit in a file's data fork. Code fragments can export internal entry points (e.g., a Mac OS function library) and can import entry points of other code fragments (e.g., an application that requires a Mac OS function). The system software is responsible for dynamically linking the entry points of code fragments at run time. As you might expect, the part of the operating system called the CFM (Code Fragment Manager) deals with loading and managing code fragments.



The structure of a 68K Mac *application and a PowerPC Mac application. The program code for the PowerPC Mac (i.e., the code fragments) is located in the data fork of the file, while resources for windows, icons, and controls still reside in the resource fork.*

The process of launching a PowerPC Mac application is similar to that for a 68K Mac application. The Finder hands the job to a slightly modified Process Manager, which calls the CFM to load in a code fragment. From there, the CFM handles the details of dynamic entry-point resolution, which I will cover later.

But on a Power Mac, the Process Manager faces a dilemma when you double-click on a file. How does it know whether to use the Segment Loader or the CFM? The answer is a special cfrg resource that has flags that inform the Process Manager whether the application is a PowerPC application or a "fat binary" (i.e., a combination of PowerPC and 68K code that can run on any Mac). The Process Manager uses this resource to determine whether to use the CFM or the Segment Loader to launch the application. If the Process Manager fails to find this resource, it assumes the application has only 68K code and uses the Segment Loader.

Code Fragments Revealed

While Power Mac applications are single code fragments, they often depend on functions in other code fragments, such as libraries or system software. In fact, portions of the Power Mac ROMs are packaged as code fragments. One of the CFM's jobs is to resolve all dependencies of a given code fragment after it loads the fragment into memory.

Code fragments exist in two executable formats, XCOFFs and PEFs. XCOFF is IBM's Extended Common Object File Format, while PEF is Apple's Preferred Excecutable Format. Here I will focus on the PEF file structure. A PEF is a container of code, data, and loader information. The PEF container is the code fragment itself, and the loader information spells out imported functions and data, exported functions and data, and version information.

To see how this all fits together, consider the example of when the CFM launches a Power Macintosh application. It first loads and locks the given code fragment into memory. The CFM then searches through the import portion of the PEF container to obtain a list of all the libraries that the application depends on. Iterating through the list of dependencies, the CFM builds a list of all entry points into each code fragment that the application needs. The CFM loads each fragment required by the application. This process is recursive.

Once a fragment that has no other dependencies is loaded, its globals and statics are built within the application heap. Then the recursive function of loading fragments is unraveled via a two-step process. First, each dependent fragment receives the addresses of the entry points into the fragments that they use. Then the dependent fragment's globals are created.

A concrete example of this is where application code fragment A depends on code fragment M, which in turn depends on fragment X. The Process Manager first allocates a heap space for application A. Next, code fragment A is loaded by the CFM. (Note that the code fragment might not be loaded into the application heap space, as is the case with 68K applications.) Then fragment M is loaded, followed by fragment X.

The CFM, knowing that X doesn't rely on other libraries, creates X's globals within A's heap space. Then the CFM preinitializes M's jump table with the addresses of all entry points within X that M is dependent on (i.e., addresses of functions, procedures, gobal data structures,

and other global variables). Then, M's global variables are created. Finally, A is preinitialized with the entry points and addresses of M. Then A's own global variables are built by the CFM. Finally, A's `main()` function is called, which begins program execution.

Statics and Globals

A critical part of the Power Macintosh's application setup is the creation and initialization of a fragment's global variables and data. The CFM gives the code fragments access to global variables, static data, and a jump table through a data structure called the Table of Contents, or TOC. The TOC contains a list of pointers to the various data elements and entry points within the global data space and to other shared libraries to which the code fragment needs access.

After the CFM loads and resolves all of a fragment's dependencies, it prepares and initializes the fragment's globals and statics. First it allocates memory for the globals' data space—which also contains the TOC—within the application's heap space. Shared libraries that are required by an application fragment build their data structures within the application's heap space as well. Then the CFM initializes the pointers within the TOC.

The TOC has three kinds of pointers. They can reference the code fragment's own globals and statics, the globals and statics of another code fragment, or entry points within other code fragments (which is essentially a jump table). See the figure "The Structure of Dynamic Links for Code and Data."

References to globals require two assembly language references to memory. The first retrieves the address of the global, while the second actually gets and sets the global's value. The question that's often asked is, "Why two references?" There are two benefits that code fragments get from using double indirection. First, TOC entries are referenced using a fixed 16-bit offset from a base register. This means that code can have only 32 KB of global data (64 KB if negative offsets could be used). In the double indirection model, code can have 32 KB (or 64 KB) of pointers to data, yielding up to 8192 (or 16,384) individual items, each of which can be any size. A second benefit is that one fragment might wish to access a variable used in another fragment. Double indirection allows this type of memory sharing, since both fragments can have pointers to the same shared location.

Consider in detail how the mechanism for calling another code fragment works. The PowerPC physically has 32 general-purpose registers. One of those registers, which is a pointer to the globals, is known as GPR2 (General Purpose Register 2). It's commonly called the TOC register because it points to the TOC for the currently executing code fragment.

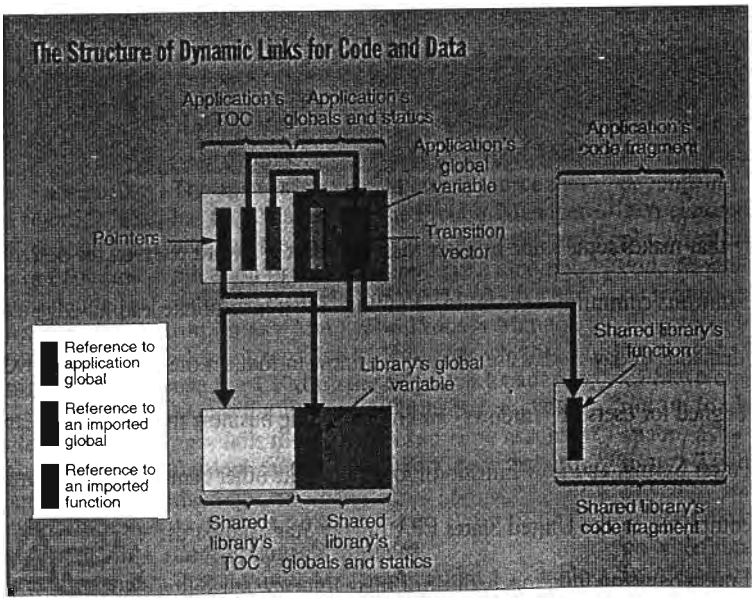
If code fragment A calls a function in code fragment M, what's going to set the TOC register to point to M's globals? The Power Macintosh run-time architecture assigns this responsibility to the caller. In other words, whenever a code fragment executes, it can rely on the TOC to be a valid pointer to its globals (except, perhaps, for some native interrupt handlers).

Therefore, the application needs to have not only the address of an entry point into a code fragment, but also the address of that code fragment's globals. This information is stored within the globals' space in a structure called a *transition vector*. This structure contains two elements: the pointer for the target code fragment's TOC, and the entry point of the function being called.

The process of calling another code fragment is called "making a cross-TOC call." The code to perform this must do four things. First, the caller saves the current TOC GPR within the linkage area of the stack. Second, it sets the TOC GPR to point to the called fragment's globals. Then the caller makes the function call. Finally, when execution returns to the original code fragment, the TOC gets reset to point back to the caller's globals, which completes the cross-TOC call.

This dynamic linking strategy works to minimize the copies of various libraries in RAM during concurrent execution of applications that rely on the same libraries. Each application that relies on a library invokes an "instance" of the library. Each instance has its own global variables, unless the library implements a shared global-memory strategy.

One major benefit of this design is that access to global information is significantly easier than was possible with the 68K run-time architecture. Previously, extensions, plug-in modules, and various periodic tasks had to resort to assembly language code to access globals within the operating system or in an application. Now global data access is a characteristic of the Power Macintosh run-time architecture itself; no special programming is required to use information inside another code fragment.

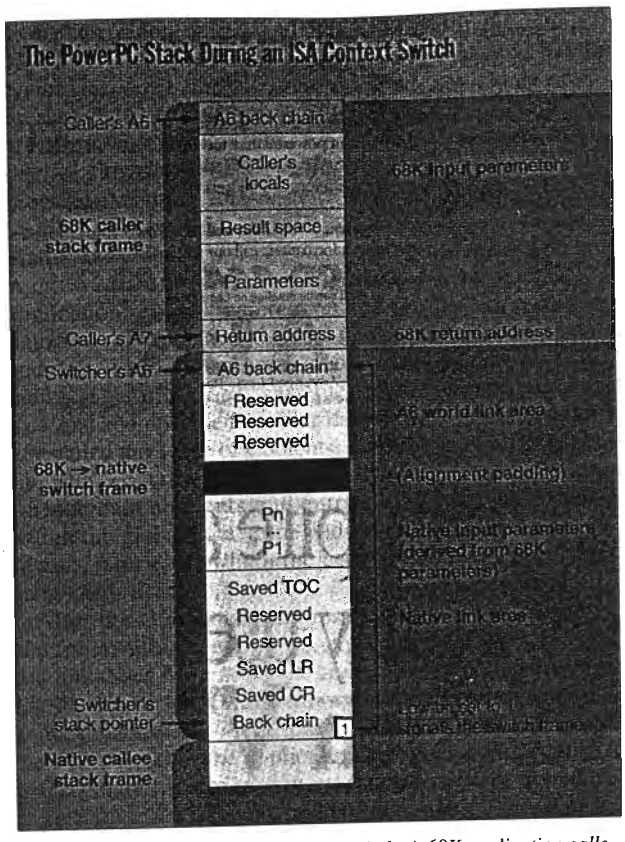


A PowerPC application *uses a TOC to point to various structures required by the application. The TOC points to the application's own global and static variables, other fragments' globals, and transition vectors that point to the TOC and function-entry points of shared libraries that the application uses.*

Compatibility Components

As mentioned earlier, the Power Macintoshes support existing 68K applications using the Macintosh API, a 68LC040 emulator, and a new Mixed Mode Manager. Macintosh applications rely on the services of system software through published entry points, which are collectively called the Macintosh API.

This API is made up of numerous Managers, including QuickDraw (which handles screen drawing), the Window Manager (which uses QuickDraw to draw windows), and the Font Manager (which handles the display of text in a variety of typefaces and



The PowerPC stack *during a mode switch. A 68K application calls a PowerPC function, which invokes the Mixed Mode Manager, which in turn uses information in a routine descriptor to build a switch frame. The switch frame contains information about the function to be called, the state of various registers, and the parameters passed to the function. Register A7 is the 68K stack, and A6 is the 68K link register. The 601's Link Register (LR) points to code that cleans up the stack and restarts the emulator.*

styles). The Macintosh API also provides high-level, hardware-independent access to low-level functions, such as sound generation (via the Sound Manager), expansion boards (via the Slot Manager), and serial I/O (via the Communications Toolbox).

Because applications use only these well-defined published entry points, Apple software engineers could replace the code behind the API without requiring huge changes to existing applications. Furthermore, replacing the API code with PowerPC code improves the performance of these applications dramatically because they rely so heavily on API calls.

The 68LC040 emulator deals with those portions of the application code that do not make calls to the Macintosh API. It maintains the stack frames, user and supervisor mode, interrupt handling, and other processor characteristics on which programmers depend. The emulator supports all 68LC040 user-mode instructions. However, it does not emulate either the FPU or the MMU (memory management unit).

The applications that query the system software for the processor type discover that a 68020 is operating. The 68020 is used because this processor marked the greatest expansion of the feature set of the 68K processor line. The 68020 introduced many new user instructions, several addressing modes, and support for a coprocessor. Subsequent processors have become faster, not more complicated.

The Mixed Mode Manager

At any given moment, a Mac application might be running emulated 68K code or executing native PowerPC code when it makes a call to the Macintosh API. This is further complicated by the fact that, in the interest of getting the Power Macintoshes on the market rapidly with a minimum of compatibility problems, the designers did not write all the Macintosh API calls in the PowerPC code.

The new Mixed Mode Manager is at the heart of making disparate PowerPC code and 68K code work together, while providing the benefit of both ISAs (instruction set architectures). It allows functions in the PowerPC ISA to call functions in the 68K ISA and vice versa.

Essentially, the Mixed Mode Manager is a stack-frame transformation engine. Switching between 68K emulation and PowerPC execution is fairly straightforward, while converting a 68K stack into a PowerPC stack can be quite involved. The calling conventions used by the Macintosh 68K model are dependent on the language (Pascal, C, and 68K assembly language each use a different calling convention), while the PowerPC has a unified strategy for all languages.

This problem is resolved by supplying a UPP (Universal Procedure Pointer) for all exported functions. The UPP points directly to 68K code (on a 68K Mac) or to a *routine descriptor* (on a Power Mac). A routine descriptor is a data structure that gives the Mixed Mode Manager the necessary pointers to the actual implementation(s) of the function, either in 68K or PowerPC code. The routine descriptor also provides information on the function's language-calling convention (Pascal, C, or assembly language), the number of arguments used, and their size. This way, the Mixed Mode Manager can determine what ISA to use when jumping to a called function, as well as how to massage the stack parameters if an ISA context switch is involved (see the figure "The PowerPC Stack During an ISA Context Switch").

For calls made to the parts of the Mac API that are written in PowerPC code, the thread of execution proceeds as follows. First, a routine descriptor is encountered, which invokes the Mixed Mode Manager. The Mixed Mode Manager uses the routine descriptor information to place any passed parameters into a switch frame for use by the PowerPC function. The routine descriptor also points to the transition vector, which in turn points to the code fragment's globals and code. The Mixed Mode Manager uses the transition vector to pass control to the target code fragment.

Apple has supplied headers that define UPPs for every Macintosh API function, so porting existing code to a Power Macintosh should be transparent to the programmer. You have to write a UPP only if you are writing a plug-in module, an extension, or a custom procedure. This UPP lets the Mixed Mode Manager know what to expect when functions in your code are called.

Memory Management

By and large, system-level memory management on the Power Macintoshes has not changed from that of 68K Macs. The design decision for this was strongly influenced by the desire to maintain compatibility. There is, however, one major enhancement: *file mapping*, which is essentially virtual memory where the backing-store data for the application is the code fragment itself. Put another way, an application's code fragment on disk is mapped into a logical address space above the backing-store file. (The backing-store file is where virtual memory is written out to disk.)

As other applications run, a background application's variables might be swapped

out to the backing-store file. The only time that code fragments are loaded into memory is when they execute. When the section of memory in which a code fragment resides must be reused, that fragment simply gets purged, because fragments are read-only code: No changes need to be swapped out to the backing store. When necessary, the fragment is read back into memory. This minimizes disk I/O, because the only data actually written to the backing-store file is an application's variables, not the invariant code in the fragment.

The major benefits of file mapping, besides virtual memory, are that PowerPC-based applications do not consume valuable virtual memory space in the swap file; and application heaps do not need to be so large, because the application code itself is not within the heap. Therefore, a user can run more applications within the same-size virtual memory footprint. The Macintosh 68K segmented application strategy, on the other hand, is not a flat memory model, it supports self-modifying code (e.g., the jump table), and in general it does not lend itself well to file mapping.

Back to the Future

The speed and power of the PowerPC processor has enabled Apple to accomplish what many thought couldn't be done: incorporate a RISC chip into a mainstream consumer product. The 68LC040 emulator allows the existing base of 68K applications to operate with good performance. The Macintosh API provides public entry points that enable existing 68K applications to access system resources. It also taps into the speed offered by the operating-system functions that are written in PowerPC code. The new Mixed Mode Manager seamlessly integrates the two incompatible processor ISAs into one smoothly operating whole.

Nevertheless, this major design improvement is not just for backward compatibility. The new Power Macintosh application run-time architecture is also ready for the time when applications can more easily communicate with one another and share resources. It lays a solid foundation on which a microkernel-based operating system with memory protection, preemptive multitasking, and multiple threads will evolve. ■

Randy Thelen is a system software engineer for Apple Computer (Cupertino, CA). You can reach him on AppleLink as "RANDOM," on the Internet at random@applelink.apple.com, or on BIX c/o "editors."

Save Disk Space

PKZIP®

PKZIP version 2.0

PC WORLD

WORLD CLASS AWARD

PKWARE introduces the next generation of its award winning compression utility. PKZIP 2.0 yields greater performance levels than achieved with previous releases of the software. PKZIP compresses and archives files. This saves disk space and reduces file transfer time.

Software developers! You can significantly reduce product duplication costs by decreasing the number of disks required to distribute your applications. Call for Distribution License information.

Put Your Executables on a Diet

Software developers! Save disk space and media costs with smaller executables. You can distribute your software in a compressed form with PKLITE Professional. PKLITE Professional gives you the ability to compress files so that they cannot be expanded by PKLITE. This discourages reverse engineering of your programs.

PKLITE

PKLITE increases your valuable disk space by compressing DOS executable (.EXE and .COM) files by an average of 45%. The operation of PKLITE is transparent; all you will notice is more available disk space!

Compression for YOUR Application

PK

The PKWARE Data Compression Library allows you to incorporate data compression technology into your software applications. The application program controls all the input and output of data, allowing data to be compressed or extracted to or from any device or area of memory.

All-Purpose Data Compression Algorithm compresses ASCII or binary data quickly. The routines can be used with many popular DOS compilers. A Windows DLL and an OS/2 32-bit version is also available.

PKWARE INC.

The Data Compression Experts

Circle 279 on Inquiry Card.