MACINTOSH

DEVELOPMENT



The Macintosh Programmer's Workshop

Richard J. Meyers and Jeff W. Parrish, Apple Computer

The Macintosh development environment combines a graphical interface, Unix-like command language, and immediate command execution in any window. Here's how it works. hen the Apple Macintosh was introduced, it was clear that Apple needed to provide a complete software-development environment for both itself and third-party developers. The result was the Macintosh Programmer's Workshop.' MPW includes a comprehensive set of programming tools and handles several languages. It also combines the graphical interface common to all Macintosh applications, a command language similar to the Unix shell languages, and Smalltalk's immediate execution of commands in any window.

Innovative features include an integrated command and editing language, redirection of input and output to windows and to selected text in windows, a graphical interface for each command, and user-defined menus, which associate shell commands with menu items.

The supported languages include Motorola 68xxx assembly, C, Apple Pascal (which supports object-oriented programming), a shell command language, and a language for automating program builds. Originally developed for the Macintosh Plus, MPW now also runs on the Macintosh SE and II.

This article focuses on the program-development environment rather than on the tools in that environment. Innovations not covered here include C and Pascal extensions for extended-precision IEEE floating-point numbers, tools to compile and decompile Macintosh resources (data structures used to describe menus, windows, and dialogues), and the MacApp object-oriented generic application.

History

When the Macintosh was introduced in January 1984, most software for it was written using the Lisa Workshop, a cross-development system that ran on Apple's Lisa computer. The Lisa Workshop provided a mouse-driven editor, 68000 assembly language, Pascal, a linker, and several utilities

May 1988

0740-7459/88/0500/0059/\$01.00 ©1988 IEEE

running in an environment similar to UCSD Pascal. A second alternative, the MDS Macintosh development system, had the advantage of running on the Macintosh, but it provided only assembly language and was merely a collection of separate applications (including an editor, assembler, and linker) that were executed sequentially — it was *not* an integrated environment.

By summer 1984, it was clear that a more complete development system running on the Macintosh was needed as soon as possible. The urgency of this need was underscored by the prediction that the Lisa would soon be superseded by the Macintosh, bringing cross development to an abrupt end.

The engineering team grew from three to 15 during the first year of development. At the end of that year, early versions of the Macintosh Programmer's Workshop were being used to develop MPW itself. Much of Apple's internal Macintosh software development has been done with this software since late 1985. In September 1986, Version 1.0 was released through the Apple Programmer's and Developer's Association, a group based in Renton, Wash. just less than two years after the project began. A major update, Version 2.0, was completed and released in August 1987.

Goals

Four major design objectives helped shaped the system:

• The system had to be comprehensive, meeting most needs of most developers. Software developers inside Apple, as well as third-party developers, expected MPW to support development in 68000 assembly, Pascal, and C. Because MPW was to be the development system that Apple supplied, it was difficult to cut corners: If something was needed for software development on Macintosh, it was our responsibility to provide it, whether minor or major.

• We preferred an integrated system over the multiapplication Macintosh Development System that was in use when the design work began. The system should be a Macintosh application, providing the standard windows, menus, and dialogues. Moreover, it had to appear as a single application rather than a set of disjoint utilities. The various system components should share a common set of windows, both for reading and writing. This would let the user view the input and output of several development tools simultaneously. And an integrated system would provide a basis on which to build automated operation.

• The system had to provide automated operation for program builds. As the complexity of Macintosh software grew in response to increased hardware capabilities, building software essentially by hand one step at a time — was rapidly becoming unworkable. MPW had to provide a mechanism to automate program builds — and it had to manage the dependencies between software components.

• The system had to be easily extensible, both by Apple and by other users. The tar-

We had four major design objectives: Make the system comprehensive, integrate it with the environment, provide automated builds, and make it easily extensible.

get audience for MPW was broad. It included university students, software enthusiasts, professional application developers, and Apple engineers. It would be impossible to satisfy the individual needs, much less the preferences, of all our users, so the system had to be easy to customize and extend. The goal was to let individual users customize the system, making simple modifications and additions, and let professionals add major new components, such as additional programming languages, without our involvement.

Several systems already existed that met many of our goals. Unix provided a comprehensive set of tools, automated operation, and some degree of extensibility, but most versions lacked the window-based user interface. Unix also placed demands on the hardware that the Macintosh could not satisfy at the time. Smalltalk provided a user interface closer to our goals, integration to the point that system and user software were difficult to distinguish, and some automation and extensibility. However, it was difficult to use Smalltalk to create stand-alone applications, and its performance was not adequate.

Envisioning a system that met our goals was not a problem: A combination of Macintosh's user interface, Smalltalk's interactive nature, and Unix's command language was a good approximation. The challenge was to engineer this vision on the limited hardware and software available on early Macintosh systems: The Macintosh Plus, which would be the high-end machine when MPW was introduced, included a Motorola 68000 running at 7.3 MHz and only 1Mbyte of main memory. The maximum hard-disk capacity was 20 Mbytes.

Software constraints provided an even bigger challenge: The system architecture, encoded in ROM chips, would run only a single application in a single address space. Multiprocessing, multitasking, and virtual memory were not provided. The system did, however, provide code segmentation, resources (swappable data structures, such as descriptions of menus and windows), and a memory manager that supports a heap with relocatable objects.

Design

MPW is built around a single application program that serves as a combined editor and shell. Add-on components, called tools, run in the same address space as the primary application. A command language automates operation, and a graphical interface helps construct and execute commands.

Macintosh application. The central development-system component is MPW Shell. Because it is a Macintosh application, it can be run from Finder, which provides icon- and window-based file manipulation, by opening any MPW document. (Documents are files created by applications. Documents are opened to see their contents.) You can also use MPW Shell as the start-up application.

MPW Shell provides a multiwindow environment for editing text files (see Figure 1). Editing is done with the mouse (or cursor keys) and pull-down menus (with keyboard equivalents). You can select font styles and sizes for each window. Because the shell provides the user interface common to all Macintosh applications, the users know the basic system operation.

Integrated editor and shell. MPW Shell combines the functionality of a traditional editor and shell. Windows can contain source code, documentation, scripts, and so on; the shell provides general textediting capabilities. Editing is disk-based: The entire document need not be in memory, so you can edit very large files, helping satisfy the comprehensiveness design goal. You can use menus to create, open, and print files, to edit, to search and replace text using literal or regular-expression-based patterns, and to manipulate windows.

The shell recognizes a command language similar to the languages used by the Bourne shell² and C shell³ under Unix. Shell commands may be entered in any window. They are executed by pressing the Enter key, an idea borrowed from the Smalltalk Do-It button. Pressing Enter executes the command or commands that are selected in the topmost window. If no selection exists, the line containing the insertion point (the point where typed text appears) is executed. Thus, using Enter instead of Return lets you type and execute commands immediately. You can correct errors by editing the incorrect command and reentering it.

Because you can execute commands from any window, you can execute them directly from program documentation or comments in the source code. MPW users often create files of frequently used commands, some general-purpose and others project-specific. Integrating the shell and editor, coupled with the use of the Enter key, minimizes the typing required to run the system.

Commands. Syntactically, commands consist of a name, parameters (often file names), and options. Commands are implemented four ways:

- in the shell,
- as scripts,
- as integrated tools, and
- as applications.

May 1988

MPW provides many commands, such as those for text editing, window manipulation, file management, and compiling, linking, and executing programs. Version 2.0 contains more than 100 commands.

Integrating the shell and editor lets us use a single command language for both editing and shell operations. This has several obvious advantages — and some advantages that surprised us. The major ones are that:

• Users learn a single command language rather than separate languages for editing and operating the system.

• Selection expressions (an extension of regular expressions) are used both to match file names (normally a shell capability) and to match text in find and replace commands.

• All the power of the command language is available for editing operations, including shell variables and command substitution (using the output of one command, such as a list of recently modified files, as part of another command).

• You can use the shell's control constructs (like If, For, and Loop) to combine editing operations.

· Editing operations can be easily and

both built-in commands provide the fastest exes sevcution and require the least amount of disk space. However, because they are linked with the shell program, they can

terfaces.

ing the shell. Command-language control constructs (Begin, If, For, Loop, Break, Continue, and Exit) are implemented in the shell as part of the command interpreter. Most of the file-handling commands (like Files, Duplicate, Rename, and Delete) and editing commands (like Align, Copy, Cut, Paste, and Find) are also built in.

naturally included in scripts and Make

files; these commands have graphical in-

Commands are implemented four ways:

· Built-in. Many commands are imple-

mented directly by MPW Shell. These

only be modified or replaced by rebuild-

• Scripts. A file of shell commands is called a script. Scripts can have an unlimited number of parameters that are accessed individually or collectively as shell variables. Scripts are often developed and tested by opening the script in a window and using the Enter key to execute the commands, one at a time or in groups.

The command language and the ability

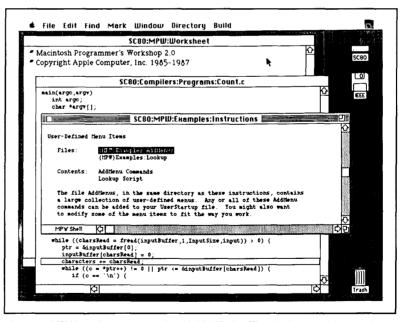


Figure 1. MPW running concurrently with Multifinder. The windows and menus shown were created by MPW. The disk and trash icons on the right belong to Finder.

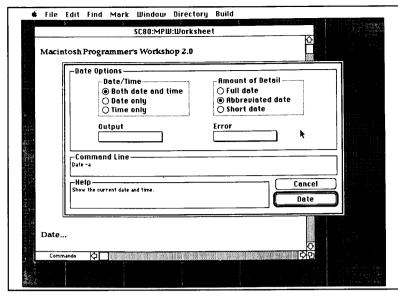


Figure 2. The dialogue box in the center of the screen provides a graphical interface to the Date command. Every command has a similar interface. Options are selected with the mouse, and context-sensitive help is available for each option. Once the parameters and options have been selected, the command can be executed immediately or saved in a window.

to write scripts helps automate program builds, one of our major objectives. Scripts are also one of the easiest ways to extend the system because they allow new commands that are combinations of existing ones.

• Tools. These are programs that are compiled and linked independently from the shell, but they run in an environment provided by the shell. Compilers, the linker, and the Make utility are typical tools. Internally, tools are similar to programs that run under a Unix shell. The command that invoked a tool is passed to the tool as a vector of strings. Tools have three files that are open by default: standard input, standard output, and diagnostic output.

Tools are the second major way to extend the system. Because tools take advantage of the windows provided by the shell, they are relatively easy to write. Most Unixstyle utilities can be ported to MPW with little or no modification.

Because the Macintosh Plus originally lacked multitasking and provided only a single address space, creating the shell support for tools was a major challenge. Tools run concurrently with the shell, automatically perform I/O to windows using the shell's editing capabilities, and share the shell's address space — yet they are written as independent programs. MPW supports a single tool running concurrently with the shell and caches other tools in memory between invocations.

• Applications. Macintosh applications, such as MacPaint and Hypercard, can also be run from the shell. These applications take over the machine and are run independently from the shell. When the user quits the application, MPW Shell is immediately restarted. Special scripts (called Suspend and Resume) are executed before the application is launched and just after the shell is restarted. They are often used to automatically save the shell's state —including open windows, user-specified menu items, aliases, and variables — and restore the state after the shell is restarted.

Parameters to applications specify which documents (files) to open or print. Because many programs developed under MPW are applications, testing is often done by running the application from the shell. You can use scripts to build and execute applications automatically.

Multifinder, which became available in late 1987, lets several applications run concurrently in fixed-memory partitions. Windows for each application appear on the screen; you switch applications by clicking the appropriate window. This lets MPW run concurrently with other applications, so you can run applications under development while you examine and modify their source code. I/O. The command language includes specifications for the source and destination of input and output. Input specifications can refer to a file, a window, selected text in a window, or text entered interactively. Output and diagnostics can replace or be appended to files, windows, or selections in windows.

Tools can use I/O routines supplied with the C and Pascal libraries or use the Macintosh's own file-system routines. The same calls are used regardless of the source or destination of the I/O, be it a file, window, or selection. Tools are normally not aware of the source of their input or the destination of their output; this simplifies the design and implementation of tools.

When a file and window have the same name, I/O operations affect the window rather than the file because windows have priority over files. This is especially important because the material in the window (such as source code) may contain recent changes not yet saved in the file. This also avoids the added complexity of providing separate mechanisms to access windows and files.

Graphical interfaces. We introduced graphical command interfaces in Version 2.0 to help users manage the many options available in the MPW commands. While the command language provides a reasonable way to automate a sequence of operations and to extend the development environment by adding both tools and scripts, remembering the various parameters and options used by a host of commands is too burdensome.

To ease this burden, command names are often English words. Options have the same meaning in several related tools. (For example, in many tools the option –p causes progress information to be written to standard output). An on-line help facility displays brief summaries of the parameters and options for each command. But even that is not always enough, so we added graphical interfaces to the commands.

Figure 2 shows the graphical interface for the Date command. The tool that implements these interfaces, called Commando, presents a graphical interface to a command by interpreting a resource description stored with the command. It handles all user interactions and builds a command string as parameters and options are chosen with the mouse. The completed command can then be passed to the shell for execution.

You select file-name parameters with the Macintosh's Standard File interface: You scroll through a hierarchical list of files and select the files to be supplied as parameters. You select options with check boxes (nonexclusive options in a menu) and radio buttons (exclusive options, named after car-radio pushbuttons). Context-sensitive help is available for each parameter and option by clicking the parameter or option with the mouse. All the commands supplied with MPW have graphical interfaces, and you can add similar interfaces to your own scripts and tools.

User-defined menus. MPW Shell provides an initial set of pull-down menu items. The shell directly executes many of these items, such as Cut, Copy, and Paste. Other menu items, such as Open... and Print Window, execute commands and thus may be implemented with both scripts and tools. (An ellipsis at the end of a menu item indicates that a dialogue will appear, letting you specify additional information.) For example, the Print Window menu item executes the command

Print {PrintOptions} "{Active}" ∂ ≥ 2 "{Worksheet}"

that runs the Print tool. (Shell variables are in braces. Using quotes allows spaces in file names. The symbol $\geq\geq$ indicates that you are appending to diagnostic output. The symbol ∂ continues the command onto the next line.)

You can create additional menu items by associating a list of commands with the name of a menu and the item in that menu. For example, the command

AddMenu File Restart 'Shutdown -r'

adds the menu item Restart to the File menu. It executes the command Shutdown –r, which restarts the computer (the –r option indicates restart). User-defined menu items can be added to existing menus or be used to create new menus. Keyboard equivalents can also be associated with user-defined menu items. MPW

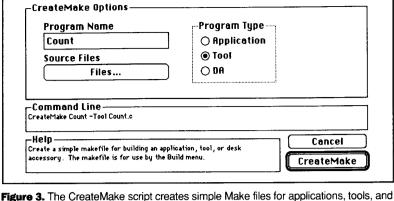


Figure 3. The CreateMake script creates simple Make files for applications, tools, and desk accessories written in assembly language, C, Pascal, and Rez. The user enters the program name, selects it source files from a list displayed when the Files... option is selected, and selects the program type.

includes several example user-defined menus.

Build menu. The Make tool is frequently used to automate program builds. Make takes as input a list of file dependencies (such as the dependence of object files on source files) and the shell commands needed to satisfy these dependencies. It uses the dependencies and the modification dates of files to determine which commands must be executed to bring the program up to date. Because the shell executes only one tool at a time, Make cannot execute other commands directly. Instead, it writes the commands to standard output. A simple script, such as

Make target > Makeout Makeout

can run Make, direct its output to file Makeout, and then execute the commands generated by Make.

Constructing the input to Make is difficult for new users and is tedious for experienced users. MPW Version 2.0 includes a script, called CreateMake, that creates a simple Make file for programs written in assembly language, C, Pascal, and Rez (a language for describing Macintosh resources; resources are swappable data or code). Parameters to CreateMake indicate the name of the program and its source files. An option indicates the type of program to construct: an application, tool, or desk accessory. CreateMake uses these parameters to construct Make input that specifies the commands needed to build the program, including the appropriate link options and a list of libraries based on the languages used in the program. Figure 3 shows the graphical interface to CreateMake.

CreateMake is executed by the first

menu item in the Build menu, Create Build Commands.... Figure 4 shows the Build menu. Other items in the menu build the program based on recent changes, completely rebuild the program, and display the build and complete build commands in a window. Each Build menu item uses Make to create the list of build commands. You can use the Make file created by CreateMake, modify it, or create a new Make file.

The Build menu is composed of a series of user-defined menu items, so it is both optional and easily extended. In addition, CreateMake is implemented as a script, making it easy to modify for other languages. The Build menu and CreateMake illustrate the system's extensibility: While Apple supplies these scripts, any user could have created them.

Internals

To provide tight integration among the shell, editor, and tools, we had to implement some services normally considered to be the operating system's responsibility. For example, to share memory efficiently,

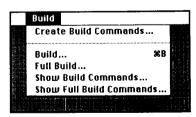


Figure 4. The Build menu provide an easy way to build most programs. The Create Build Commands... menu item runs CreateMake to construct a Make file for the program. The remaining items automatically build the program using Make.

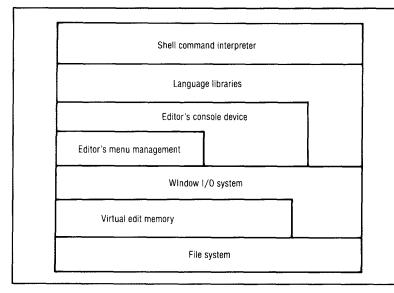


Figure 5. I/O organization. MPW Shell uses the standard language libraries for all its I/O. When the shell is reading from the console device, the data it receives may originate from several sources, including windows and menu items.

we had to implement our own code caching and garbage collection.

Editor/shell integration. Although they are highly integrated at the user level, the shell and editor are conceptually two distinct programs that are only loosely coupled. The shell is responsible for command processing, resource management, and memory management. The editor is responsible for editing and I/O management.

Command processing in the shell is designed much like any tool. The command interpreter reads a line of text from standard input, processes the line, reads the next line of text to process, and so on until a terminating condition occurs. The command interpreter uses the standard C library routines for all I/O instead of accessing the editor's data structures directly (see Figure 5). Using the standard I/O routines gives the shell the same flexibility as any other tool to read or write files, windows, or selections in windows without explicitly specifying the type of object for which the I/O is intended.

MPW provides a simple, generic console-device interface to the language libraries. The console is an on-screen command interface; in the simplest case, it is like a window-based teletype. By supplying a generic console-device interface to the language libraries, the editor gives you control over where the console data originates. For example, in addition to returning the contents of a window or the contents of the selection in a window, the console can also return text strings that have been associated with user-defined menu items.

User-defined menu items are a good example of how the editor was designed to use Macintosh features to help users customize their environments. For example, you can create command macros that can be invoked from the keyboard by combining the editor's user-defined menu items with the facility that lets you associate any key on the keyboard with any menu item.

Tools. Workshop tools are closely patterned after the programs that execute from a Unix shell. They are compiled and linked independently of the shell. Before a tool is executed, the shell sets up the tool's parameters and opens the three files, standard input, standard output, and diagnostic output.

Because the Macintosh Plus lacks multitasking, the shell must explicitly give up control to begin tool execution. The tool will retain control until it requires a service managed by the shell, such as window I/O and tool-segment loading. After the shell services the tool's request, it returns control to the tool. When the tool terminates, whether successfully or after an error, the shell ensures that all the files opened by the tool are closed and that any memory allocated by the tool is reclaimed.

You can write tools in any combination of the languages supported by MPW: assembly, C, and Pascal. We have developed runtime library interfaces for assembly language and Pascal to offer the same capabilities as C (such as standard input and standard output).

Shared memory. The Macintosh operating system was originally designed to run one program at a time. An application's global data, stack, and heap occupied most of the available memory. When it terminated, another application — usually Finder — would overlay the space occupied by the application. Running a new program always implied a complete purge of the old program, regardless of the new program's memory requirements.

Some programs, notably Multifinder and Switcher (which let multiple applications share memory and which let the user quickly switch among them), use a memory-partitioning scheme that creates separate application areas so more than one program can exist in memory. This scheme has the benefit of having well-defined boundaries between applications and makes it relatively simple to clean up memory after an application has terminated. But its major disadvantage is that memory is easily fragmented, so a memory request by an application often cannot be satisfied even though the space is available in another partition.

To maximize memory use, MPW Shell uses a shared-partition scheme where the shell and tools share a common heap and stack (see Figure 6). When the shell starts a tool, it allocates a global area for the tool in the common heap (which contains both code and data), adjusts the global register accordingly, sets up the tool's intersegment jump table, and then calls the tool.

The shell tracks some system resources allocated to tools, such as files and memory, to reclaim the resources when the tool terminates. The shell tracks these resources by overriding several ROM operating-system routines. It monitors heap allocation by intercepting the memory-allocation routines and using a block-tagging scheme to differentiate shell and tool blocks. By monitoring the appropriate file-system calls, the shell records in a shell data structure the names of files opened by a tool and removes the name when the files are closed. The shell determines if a memory-manager or filesystem call came from a tool (or from ROM on behalf of a tool) by comparing the current global context to the tool's global context.

When a tool terminates, it returns to the shell. The shell then cleans the heap by removing all tagged blocks. It also closes any remaining files left open by the tool.

To improve the time to reexecute a tool, the shell postpones closing the tool's resource file and removing any of the tool's resource blocks in memory (most notably, code segments). Instead, it marks these memory blocks as relocatable and purgeable so the memory manager may later move the blocks around to consolidate free space or reclaim the space completely if needed. The next time the tool is run, any resources still in memory are used without reloading them from disk.

In many cases, the code segments of several tools can remain memory-resident simultaneously. However, when a tool's memory requirements are large, the shell does its best to free up memory as needed. Before starting a tool, the shell makes many of its own heap objects relocatable and purgeable so they may be moved or deleted as needed. As the heap fills up, the shell will dispose memory allocated by inactive tools and write the modified portions of all files opened as windows to its scratch file on disk.

By sharing the shell's memory this way, a compiler, for example, that requires 1 Mbyte to run efficiently by itself will require just a little more than that 1 Mbyte to run — including the space for the shell. This allocation strategy takes less memory than the total memory requirements to run the compiler in a separate partition where the maximum memory requirements of both the shell and compiler would have to be met simultaneously.

Because both the shell and the tool have separate global environments, some form of context switch must occur during a call between the two environments (such as a tool's libraries calling some part of window I/O). We devised a context-switching system, called Export, to do this. A routine that may be called from another context can be registered via Export to provide the automatic switching. Export works by recording the proper context at the time of the export and returning a new address to be used for the exported routine. When

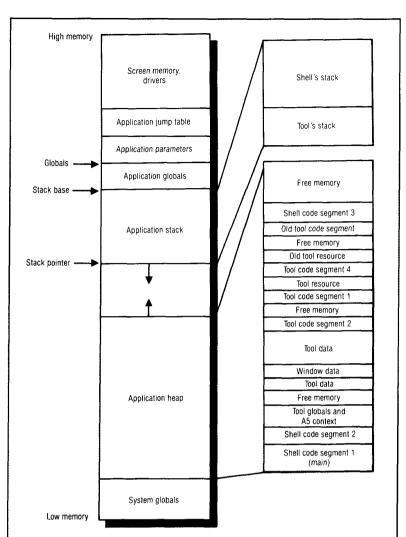


Figure 6. Memory organization. MPW Shell shares its stack with an executing tool. The application heap is shared between the shell, the executing tool, and one or more previously executed tools.

the new address is called, an intermediary ensures that the context is correct and then calls the real routine as if it were a local call. On return, the intermediary restores the context to the previous state.

Window I/O. The editor supports window I/O at two levels: by overriding device handlers in the runtime libraries and by overriding the file system. Overriding the runtime libraries' device handlers is necessary to give tools access to information, such as the tab setting of a window, that cannot be accessed through ROM routines.

MPW runtime libraries' device handlers closely resemble Unix device handlers. There are three standard handlers: the console device, file device, and system device. Each handler contains five standard entries: read, write, close, ioctl, and faccess. At initialization, the editor replaces the shell's entire console handler and the faccess and ioctl entries of the file handler with its own exported functions. A tool's device handlers are copied by its runtime libraries at initialization from the table of device handlers associated with the shell.

Tools can also use ROM calls for I/O and still benefit from the window I/O supplied by the editor. By overriding the file system, the editor can watch all ROM file-system calls and either process the call itself or let the ROM handle the call.

For example, when a tool tries to open a

file, the editor detects the call and searches its list of windows for one whose complete path name matches the complete path name specified. If there is no match, the editor lets the file system handle the request. If a window matches the request, the editor creates a data structure that will satisfy future I/O requests for that window and then returns a window-reference number that is outside the domain of all file-system reference numbers.

When a tool makes I/O requests other than opening a file, determining whether the request is for a window or file is done by testing whether the reference number in the request is a window-reference number

To extend the more conventional notion of I/O redirection (such as in a Unix environment) to include redirecting I/O to and from selections in windows, we needed a mechanism to differentiate the name of a window (or file) and the name of the selected text in that window, so we use the special suffix .§ to mean the selection in the file.

Acknowledgments

We thank Bud Tribble, who gave us the original charter to develop MPW, Jim Thomas, who has managed the engineering effort since early 1986, and Daniel Smith, a key designer and implementer who really should have been a coauthor. A hard-working team of 15 engineers designed and built the tools and languages included with MPW; another 15 writers and testers contributed their talents. Finally, our fans and critics both at Apple and in the Macintosh development community continually encourage us to try harder and find a better way.

References

- Macintosh Programmer's Workshop Reference, Version 2, Apple Programmer's and Developer's Assn., Renton, Wash., 1987.
- 2. System V Interface Definition, Issue 2, Vol. 2, AT&T, Murray Hill, N.J., 1986.
- W. Joy, "Introduction to the C Shell," Unix User's Supplementary Documents, 4.2 Berkeley Software Distribution, Univ. of California, Berkeley, Calif., Nov. 1980.
- D.B. Garlan and P.L. Miller, "Gnome: An Introductory Programming Environment Based on a Family of Structure Editors," *Proc. Software Eng. Symp. Practical Software-Development Environments*, ACM, New York, 1984.

PW was a product-development effort, not a research project. The goal was to produce the best possible product using well-understood technology. The major challenge was to design and implement such a system given the limited hardware and system-software resources. We believe we have done so. But we intend to do more.

Our short-term objectives are to add source-code version control and sourcelevel symbolic debugging. The sourcecode control system will make it easier for several people to work together on a common project. It will support all files related to a project, not just source code. In addition, projects may be nested in other projects, letting large projects be broken into subunits. The symbolic debugger will allow source-level debugging of programs written in any combination of MPW-supported languages. The debugger will run as an independent application under Multifinder, where it will be used to debug both tools and applications. These two enhancements should be completed this summer

Our long-term objectives are to increase the level of integration in MPW. Each time we have increased the level of integration, we have been pleasantly surprised by unexpected benefits. For example, combining the shell and editor languages makes the system easier to learn and lets you use shell variables, command substitution, and control constructs when editing scripts.

We are considering language-oriented editing, incremental compilation, and closer coupling between the editor, compilers, and other tools. Expected benefits include an increase in system performance, easier navigation through source code and documentation, and simpler coordination of multiple files and their dependencies.

The design of a fully integrated system supporting several languages is a significant challenge. MPW is a first step in meeting that challenge. Although there are already several systems with the level of integration we seek, such as Smalltalk and Gnome,⁴ we know of no system that is both highly integrated and provides equal support for multiple languages.



Richard J. Meyers is a principal engineer at Apple Computer. He implemented Smalltalk-80 on Apple's Lisa computer and has contributed to the system software on both the Lisa and Macintosh computers. He served as the engineering manager for the development and initial release of the Macintosh Programmer's Workshop.

Meyers received a BA summa cum laude in physics and mathematics from Hope College in Michigan and an MS in electrical engineering and computer science from the University of California at Berkeley.



Jeff W. Parrish is a staff engineer at Apple Computer. He designed and implemented Mac-Works on Apple's Lisa computer and has contributed to the system software on both the Lisa and Macintosh computers. He is engineering manager for the development of the Macintosh Programmer's Workshop shell environment.

Parrish received a BA and MS in physics and an MS in computer science from the University of Oregon.

Address questions about this article to the authors at Apple Computer, 20525 Mariani Ave., MS 27-E, Cupertino, CA 95014.