Advanced Interactive Executive program development environment

by R. Q. Cordell II
M. Misra
R. F. Wolfe

The IBM RT Personal Computer uses the Advanced Interactive Executive as an operating system. This operating system provides a distinct environment for the development of programs. Some of the characteristics of application development with this operating system, some of its features that influence application design, and the basic program development tools are described.

Henry[1] has described some of the ways in which user demands for high function and performance combined with ease of use in a low-cost small-system environment have led to innovative design solutions for key technical problems. The IBM RT Personal Computer™ (RT PC™), and the Advanced Interactive Executive (AIX™) operating system represent an advance in IBM's small-system technology. The RT PC provides performance and functionality rivaling that of traditional mainframe computers in a workstation environment that puts the power of the machine at the fingertips of the end user. The capabilities of the RT PC make it an attractive system for sophisticated end-user applications requiring more power than is commonly available on a workstation. This, in turn, requires a program development environment that supports the development of such applications.

The choice of a UNIX®-based operating system for the RT Personal Computer was influenced to a large degree by the desire to provide both a platform for a variety of existing applications and a system that gives the programmer a powerful and flexible set of interfaces and tools for subsequent application development. Loucks and Sauer[2] have discussed some of the motivation behind this choice, as well as some of the characteristics and enhancements of the AIX kernel and application development environment.

Additional motivation for providing a rich and powerful set of development tools was provided by the reduced instruction set computer (RISC) architecture of the underlying RT PC hardware. Given the reliance by RISC architects on sophisticated compiler technology to shield the normal application programmer from the underlying architecture, an advanced optimizing compiler is critical to allow applications to exploit the performance benefits of the machine (see Radin[3]). The emphasis is on making function and performance available to the end user without introducing additional complexity.

Thus, the combination of a RISC architecture and a UNIX-based operating system on a high-performance workstation and microcomputer requires an advanced set of program development environment tools and a friendly development environment. The following sections of this paper describe some of the characteristics of UNIX and AIX application development, some of the features of the AIX system that influence application design, and the basic AIX program development tools.

UNIX program development tools and environment

The UNIX system was originally developed by a group of researchers at the AT&T Bell Laboratories to provide an effective computing environment within which they could pursue their own work in programming research (see McIlroy et al.[4]). From the beginning, the system was designed to be simple and extendible by the programmers who used it. The model of a trusted kernel providing a small but powerful nucleus allowed for easy extendibility with minimal complexity. Many programs that comprise part of the operating system base in more complex systems are simply application programs in the UNIX environment.

The UNIX shell provides an interactive user environment above the level of the kernel.

From this origin, the UNIX system has had a history not unlike that of the Virtual Machine/Conversational Monitor System (VM/CMS) within IBM, where the programmers who used the system enhanced it by providing numerous tools in areas such as command processors, interpretive languages, text processing, library systems, and compiler development. The UNIX philosophy, like that of CMS, has been to build small, independent tools rather than large, interrelated ones, and to build software quickly while expecting to adapt it over time to a considerable degree in order to meet users' needs (see Dolotta et al.[5]).

This process has resulted in a proliferation of tools. Over time, a subset of these tools have become standard to the AT&T UNIX System V, much as XEDIT as an editor and REXX as a language have become mainstays of VM/CMS. The AIX programming environment attempts to provide the standard set of System V tools, as well as extensions in key areas. Some of the basic tools and characteristics of the UNIX System V programming environment are discussed briefly below.

Shell. The UNIX shell is a command-processing program that provides an interactive user environment (though it will also execute commands read from a file) above the level of the kernel. The shell provides an interface to the UNIX operating system services. It includes an interpretive programming language which contains control-flow primitives, parameter passing, and string-valued variables. Although the shell is simply a utility program above the level of the trusted kernel, it provides the basic command interface and user environment and, hence, plays a very visible role in the user's perception of the system.

Many variants of the original UNIX shell currently exist, including the Bourne shell, C shell, and Korn shell. Users may write additional shells. No real distinction is made between the shell provided with the operating system and those written by users, as long as the user shells accept the basic shell conventions for invocation, parameter passing, etc.

Several shells can coexist in the same system. The AIX system, for example, provides both the Bourne and C shells, as well as a menu-oriented Usability shell and an IBM Personal-Computer-oriented DOS shell. Which shell is invoked at log-in time is an option which may be specified on a per-user basis. In the AIX system, a user may log in and execute one shell (e.g., the Bourne shell), and then open a virtual terminal, creating another process that executes a different shell (e.g., the C shell; processes and virtual terminals are described later in this paper).

Editors and text processing. The UNIX system has been from the first a general-purpose time-sharing system designed to provide an effective environment supporting programmers. This fact is one of the primary motivations behind the development of a varied and powerful set of tools. One area in which the UNIX system has had a particularly rich history is that of document preparation and text processing. UNIX tools in this area include editors and text-formatting tools.

Like the shell, these text-processing programs are all standard applications. A major advantage of the UNIX system is that the vast bulk of utilities, tools, and other programs run as application programs above the level of the kernel. This arrangement lessens the effort of developing new applications, since only the most complex programs require more than the standard user services available through the

kernel system calls. This also increases the degree to which tools can build on top of each other, since they all run in the same basic environment.

Programmer's Workbench. The Programmer's Workbench UNIX system (referred to as PWB/UNIX), developed at the AT&T Bell Laboratories in the 1970s, attempted to improve the standard UNIX programming environment. The PWB/UNIX system separated program development from execution of the resulting programs, and attempted to provide a uniform front-end development environment for UNIX code which could execute on several target systems, including the IBM System/370.

The PWB/UNIX project had the important goals of providing a single, uniform programming environment, of enhancing the existing set of UNIX development tools, and of showing that the UNIX system could be not only a productive program development environment but also an effective target execution environment. This combination of an interactive environment for application development and a powerful set of system functions supporting the resulting applications is a primary characteristic of the UNIX system, and one which the AIX system on the RT PC attempts to extend still further.

AIX extensions to UNIX program development. The AIX program development environment starts with a System V base and extends this base in several key areas, including

- Berkeley Software Distribution (BSD) 4.2 enhancements such as extended signals.
- Exploiting AIX advanced features such as distributed processing, mapped files that provide a single-level store, virtual terminals, and windowing capabilities.
- A rich set of program libraries along with capabilities to facilitate interlanguage calling for applications that are written in multiple languages.

The following sections describe some fundamental design considerations for application programmers who desire to take advantage of the capabilities of the AIX system on the RT PC and also describe the AIX program development tools available with which to implement, maintain, and debug AIX applications.

Application programming design considerations

The AIX system and the RT PC provide application designers with a large set of capabilities, including the basic features of UNIX System V and many extensions. The result is a very sophisticated system which makes available to applications many features found in much larger systems, such as multiprocessing, mapped files, shared memory, data management and relational database, high-function graphics and windowing, and distributed processing. Many of these features extend beyond traditional UNIX capabilities to provide a system more suitable for supporting commercial applications in areas such as computer-aided design/computer-aided manufacturing (CAD/CAM), decision support, workstation publishing, and expert systems.

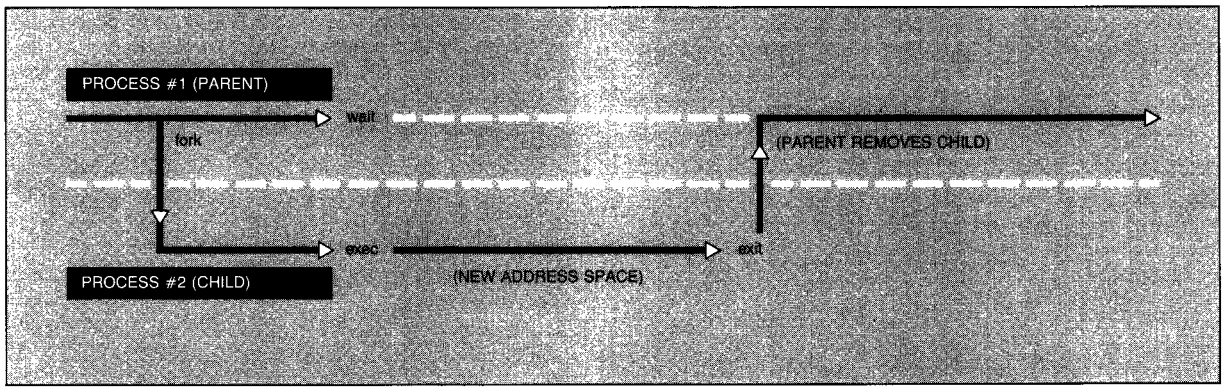
An AIX process is the current state of a program that is running.

Process model. The UNIX system was designed from its beginning to be an interactive, multiprocessing operating system, a design point combining power, flexibility, and usability. This choice has proved over time to be extremely important as computing has moved away from batch processing to an interactive emphasis, and as the subsequent user demands on machine capacity and operating system function have mandated the ability to perform multiple tasks. The ease with which application programmers are able to exploit this function has contributed greatly to the rapid growth of a base of sophisticated applications. This application base would have been significantly retarded either by a less interactive environment or by a less powerful process model.

An AIX process is the current state of a program that is running. A process is the operating system construct that is allocated CPU resources. The AIX system associates various contextual information with a process, such as a virtual address space, a current directory, and the status of open files used by that process (see the data model description below for a discussion of directories and files).

Process address space. A process includes a memory image consisting of 16 segments each up to 256

Figure 1 AIX process creation



megabytes in size, as defined by the 16-segment registers in the RT PC virtual memory management hardware (see Loucks and Sauer[2] for a description of the AIX virtual memory segment allocation). The virtual memory manager pages processes in and out of memory as necessary. Processes may run in user mode or may switch to kernel mode by issuing a system call to the AIX kernel.

Process creation and execution states. A process can create a copy of itself via the *fork* system call. The created process, referred to as the child, gets a separate address space which is a copy of that of the creating process (referred to as the parent) at the time of creation, and the two processes share open files. The program running in the child process may subsequently issue an *exec* system call, which causes the process to overlay the information it contains with new information. This is illustrated in Figure 1.

Interprocess communication. Because the UNIX system provides features allowing applications to exploit the ability to use multiple processes, the need for communication among processes becomes critical. The AIX system provides the standard UNIX System V interprocess communication (IPC) capabilities for signals, semaphores, and message queues, and extends these capabilities to provide a richer set of services supporting a multitasking program execution environment.

In addition to the standard UNIX System V signal facilities, the AIX system incorporated BSD 4.2 enhanced signals, which allow a program to mask and block each type of signal while it is executing. Blocking a signal causes that signal to be held when it is received and then handled when the signal type is unblocked.

The AIX system has extended the System V message queue services by providing a new function that returns an extended message structure containing more information, such as user ID (identification), group ID, message send time, process ID, etc.

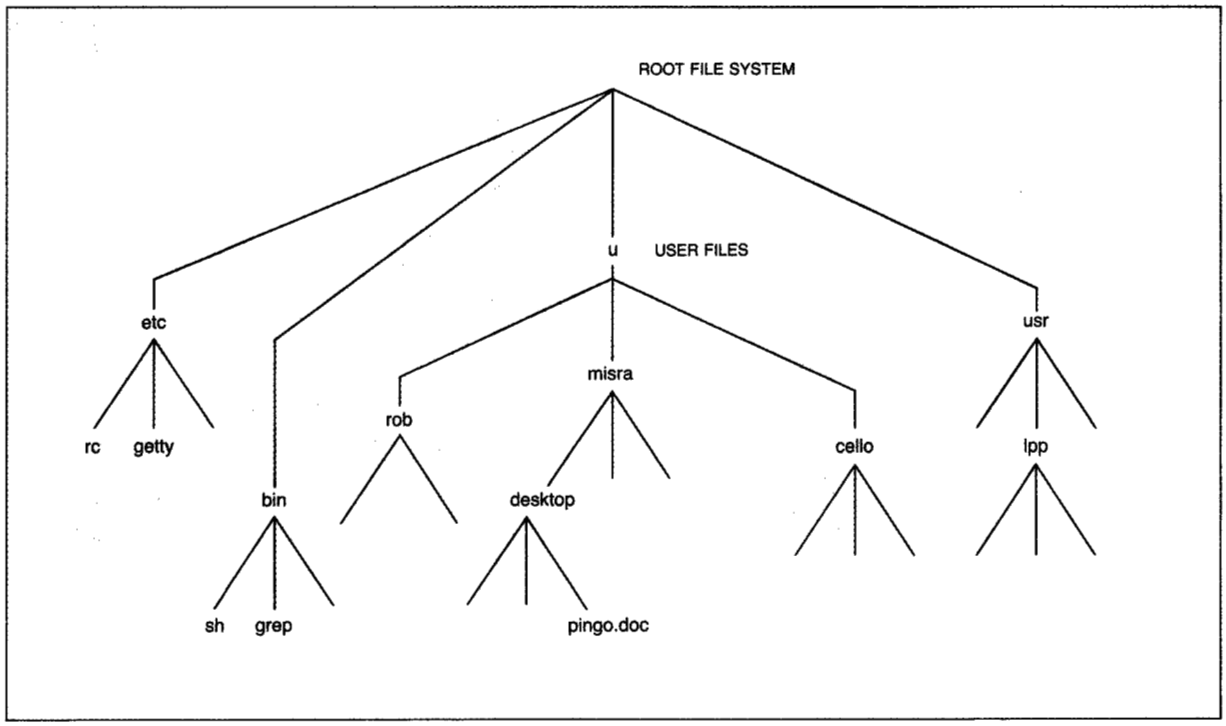
Processes can also communicate by allocating shared memory segments via the *shmat* (shared-memory attach) system call.

Data model. The AIX system provides programmers with a powerful and flexible set of data manipulation capabilities, building on the standard UNIX file system model with additional services to provide more granular and sophisticated data management capabilities to support commercial applications.

File system structure. The AIX file system is characterized by the two properties of being hierarchical in nature and of representing files as linear byte streams rather than organized into larger units such as records and fields. This model is familiar to users of the IBM Personal Computer Disk Operating System (IBM PC DOS) on the IBM Personal Computer.

Figure 2 illustrates the AIX file system structure. At the top of the file system tree is the root file system, represented as a "/". Leaf nodes (i.e., nodes which cannot have any other nodes below them) are files, which may contain user data. Directories are intermediate points in the file system tree, which may have other nodes below them in the tree, either files

Figure 2 AIX file system structure



or other directories. Directories are just AIX files with a special structure describing the files and subdirectories contained in the directory. However, an AIX user cannot write to a directory using the normal file-manipulation mechanisms, but must use a special set of directory-manipulation commands in order to preserve the directory structure. In Figure 2, the names etc, bin, usr, u, rob, misra, cello, desktop, and lpp are all directories. In the AIX system, the utility program *installp*, used to install licensed program products, uses the directory /usr, and the directory /u contains subdirectories for each user defined to the system, each of which contains user data. Thus, the directory /u/misra (i.e., the subdirectory misra in the subdirectory u off the root) contains data for the user logged in as misra.

Portions of the file system tree are referred to as filesystems and have the property of being individually mountable from a directory. For example, the directory /u is a mount point for the filesystem that contains user data. Filesystems are used for allocation purposes and for administrative control. The RT PC Distributed Services extends the notion of a mountable portion of the file system tree to provide local or remote file system transparency, as described later in this paper.

A file may always be accessed via a direct path name, i.e., a path name starting at the root and ending with the file name. For example, in Figure 2, the direct path /u/misra/desktop/pingo.doc identifies the copy of the Workstation Publishing Software™ document known as pingo available to the user logged in as misra. Alternatively, a file may be accessed via its relative path name to the current working directory. For example, if a user logged in to the AIX system as misra, the working directory would probably be /u/misra, in which case the file pingo.doc in the subdirectory desktop could simply be referenced as desktop/pingo.doc. If the current directory were changed to the subdirectory desktop, the file could be referenced as just pingo.doc, which is the actual file name. AIX file names may be up to 14 characters long. A single file can be known (accessed) by several names.

In addition to directories and ordinary files, special files are used to provide a convenient way to access the I/O mechanisms of devices. For each I/O device

there is a special file that provides an interface between the application and the kernel support for that device. Most special files are found in the /dev directory. Thus, the file system provides applications with a consistent mechanism for accessing all devices, not just disk data.

All files have read, write, and execute permissions for the file owner, group, and other users.

File permissions and sharing. All files have read, write, and execute permissions for the file owner, group, and other users. Any user with a given permission on a file is allowed access to the file for that specific type of access.

Permissions are treated differently for directories than for ordinary files, in the sense that the system does not allow users direct access to the contents of a directory entry to avoid corruption of directory information. Read permission on a directory means that the user will be able to use the standard directory utilities to read the information in the directory; write permission means that the user will be able to use the standard directory utilities to create or remove directory entries; and execute permission means that the user will be able to search the directory for a file name. This last permission provides a useful mechanism for allowing or denying users the ability to use files in that directory.

Mapped files. AIX mapped file support provides an explicit interface whereby programmers can choose to have data files mapped to the large virtual address space supported by the AIX system on the RT PC, achieving a "single-level store" rather than the traditional two-level (memory and disk) storage model. Once a file is mapped to memory, programmers can subsequently access mapped file data by direct loads and stores rather than by reads and writes, with the virtual memory paging system managing the physical I/O activity.

The *shmat* system call with the map option is used to map an open file to virtual memory. Optional flags supplied with the shmat system call specify how the file is to be mapped, with possible options including read-only, read-write, and copy-on-write. (Copy-on-write means that changes applied to the file in memory do not affect the file resident on disk until an *fsync* system call is issued for the mapped file.) In order for the *shmat* call to succeed, the file must have been opened with the appropriate access options; e.g., the file must have been opened with write access in order to map the file read-write.

A single virtual memory segment is used to support all processes that map a given file read-only or read-write. The segment remains mapped until the last process mapping the file closes the file. A separate segment is used to support a file that is mapped copy-on-write.

Mapped file support provides both a useful programming model with a single-level store and a considerable improvement in performance. These combine to make mapped files a particularly attractive feature of the AIX operating system.

Database and data management. The representation of files as linear byte streams rather than as being organized into records and fields has been noted as a major advantage of the UNIX model over many other systems. This representation allows programmers considerable power and flexibility to choose whatever structure is best suited to a particular application (see Ritchie[6]). As Bissell[7] has noted, however, this freedom of choice often results in the programmers who require random access to define substructures within a file (such as records and fields) having to develop their own access methods, thereby increasing application size, complexity, and development time. The AIX system attempts to combine the best of both worlds by providing three discrete levels of data representation and management: the base file system, an indexed data management capability, and a relational database.

The IBM RT PC Data Management Services provides a general-purpose indexed access method. Data management files may contain fixed- or variable-length records. Access to data management files may be sequential by record, by relative byte address of the record within the file, or by indexed access using B-tree techniques.

Data management also provides a set of catalog services which extend the amount of information about files beyond that available in the base file system. The catalog structure for data management files is an extension of the AIX file system directory structure, so that the user views data management files as being handled in the same manner as normal AIX files.

For programs that require still more sophisticated data facilities, such as field-level access within records and transaction control for data integrity, programmers may use the Structured Query Language/RT (SQL/RT) relational database management system. SQL/RT provides a relational data model in which data are represented as a table consisting of rows subdivided into individually accessible columns. Access to data stored in SQL/RT tables is via the IBM SQL language.

Both data management and the SQL/RT relational database exploit many of the extended features of the AIX system, particularly the mapped file support described above.

User interface and terminal support. A wide range of display terminals can be connected to the RT PC. These can be as simple as character glass teletypes, such as a Digital Equipment Corporation VT100 or an IBM 3161, or as sophisticated as bit-mapped displays, such as an IBM 6154 or an IBM 6155. The glass teletype (TTY) devices are ASCII terminals connected to the RT PC via standard asynchronous communication lines. The bit-mapped displays are directly connected to the RT PC via an associated display adapter.

The AIX system provides programmers efficient and flexible interfaces to take full advantage of the capabilities of these display terminals. These interfaces can range from simple character-based applications to complex applications running in a window, or even highly sophisticated three-dimensional graphic applications. This choice allows programmers to select the level of interface most suited for their application demands.

The simplest interface for writing application programs requiring only character support is referred to as keyboard-send-receive (KSR) mode. Services that build on the KSR mode include the curses and extended curses library routines and the AIX dialog manager included in the AIX Usability Services.

In order to take advantage of the advanced capabilities of displays directly connected to the RT PC, multiple interfaces are provided to support a wide range of graphic applications. These interfaces include multiple virtual terminals, extended ASCII mode, monitor mode, graphics support library (GSL),

The curses interface permits manipulation of data structures called windows.

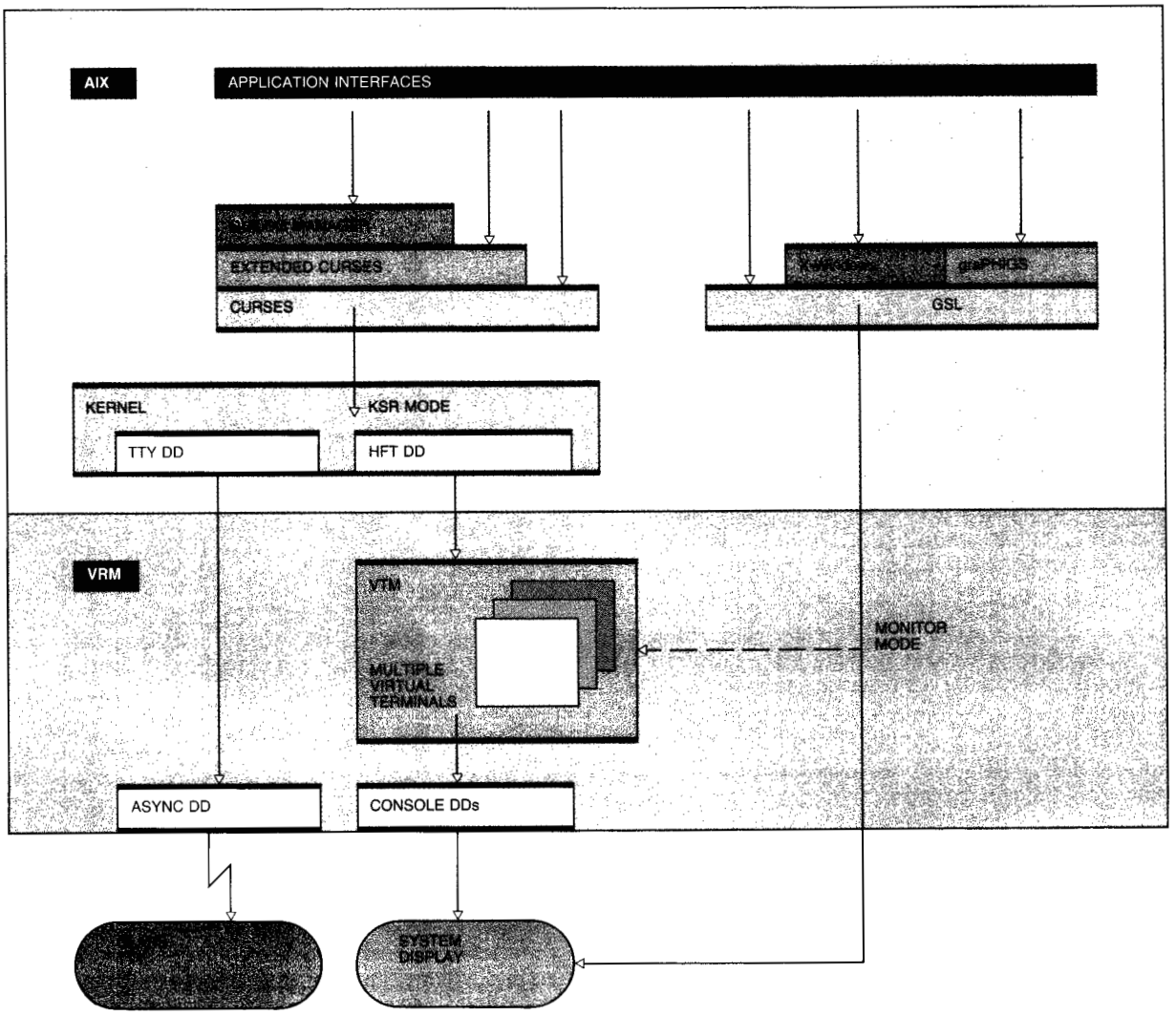
X-Windows, and graPHIGS™. Support is provided for using all-points-addressable (APA) displays, the keyboard, locator (mouse), speaker, and optional input devices such as tablets and valuators.

Figure 3 illustrates the relationship among the various terminal support components. These are described in more detail below.

Curses. To provide for migration of existing applications, the AIX system includes the UNIX curses library routines to support input and output to a terminal. The curses interface permits manipulation of data structures called windows, which can be thought of as two-dimensional arrays of characters representing all or part of a screen. Input characters received from the user can be echoed back to the window or handled directly by the application without echoing. Most interactive, screen-oriented programs require character-at-a-time input without echoing. These library routines provide programmers with the ability to control multiple, overlapping areas on the display and to assist in managing the data presented. In addition, these routines use the system "terminfo" routines, which provide access and processing for terminal description files for each type of terminal supported.

Extended curses. To take advantage of more advanced KSR displays and to provide a more efficient handling of window-oriented screen presentations, the AIX system provides extensions to the UNIX curses through the extended curses routines. The extended

Figure 3 AIX terminal interfaces



curses routines provide functions for handling expanded character sets, color, multiple character display attributes, and error detection.

These routines also enhance the programmer's ability to efficiently handle window-oriented screen presentations. Some of these enhanced capabilities include the linking and scrolling of windows, scrolling data in windows that are partially covered, the ability to stack and layer windows, the ability to subdivide windows into panels and panes, and the automatic tracking of panes. The extended curses routines can be used to develop new character-oriented display applications or to increase the function of existing programs.

Dialog manager. A dialog manager facility was developed on top of the curses and extended curses routines to provide application control and services supporting processing of interactive dialogs. The dialog manager monitors operator input and performs conditional processing based on this input as specified in the dialog. This is accomplished through dialog definitions that allow the dialog manager to direct the flow of control from one screen panel to another based on the user's actions. Some of the

additional capabilities of the dialog manager include the definition of actions to be performed on selectable field input, the ability to define user exits based on these actions, the definition of help text appropriate for the context of the dialog, and data entry validation, verification, and access by the application program through named variables.

The dialog definitions are stored outside of the program that uses them. This condition gives programmers the ability to change the dialogs without requiring any modification, compilation, or reconstruction of the program. These dialog definitions are preprocessed from a readable/editable form to one that is more efficient for run-time processing. This makes the dialog manager a useful tool for prototyping and developing user interfaces for programs (see Murphy and Verburg[8]). The dialog manager is used by user interfaces provided with the AIX system, such as the Usability Services.

Monitor mode is the lowest-level graphic interface supported by the AIX system.

The AIX facilities provided by the curses, extended curses, and the dialog manager all operate in the KSR mode of operation. Thus, applications written using these AIX facilities are supported on attached RT PC terminals and displays.

Multiple virtual terminals. One of the key design tenets of the AIX system is the concept of multiple virtual terminals. Multiple virtual terminals allow the "multiplexing" of individual full-screen images on a single system display. This multiplexing is controlled by the Virtual Terminal Manager subsystem of the Virtual Resource Manager (see Loucks and Sauer[2]). A user simply "hot keys" to a different foreground process and the screen of that process is automatically displayed by the Virtual Terminal Manager, thus permitting efficient use of the system by the user. For example, the user can have a System/370 host connection on one screen, be editing a program on another screen, have a shell running in a third screen, and be able to quickly switch from one to another among all these foreground processes.

An important feature about multiple virtual terminals is that most applications need not be aware of this capability. Additionally, the design of the multiple virtual terminals of the AIX system exploits and extends the UNIX multitasking capabilities by permitting more than one foreground application to share the display.

Extended ASCII mode. The default mode for a virtual terminal is the extended ASCII mode of operation. This mode is essentially the standard ASCII terminal character mode of operation and allows programs written to that interface to execute properly. The extended ASCII mode also permits new applications to access and make use of the locator and sound functions of the RT PC.

Monitor mode. Monitor mode is the lowest-level graphic interface supported by the AIX system. This mode allows programs to deal directly with the display hardware adapter by storing to the memory-mapped I/O section of the display buffer. As such, monitor mode provides the optimal performance for graphic display applications. However, since an application addresses the display adapter directly, support must be added for each display supported by an application.

Conventions have been established to ensure that applications written to this interface permit the virtual terminal mode of display sharing. These conventions require the application to relinquish the display upon demand by the Virtual Terminal Manager when the user switches to another foreground process. This is accomplished through AIX extensions to the UNIX signal protocols described earlier in this paper. Thus, the environment of the virtual terminal manager support of the AIX system is protected.

Dealing directly with the display hardware in monitor mode is complex but does permit applications to have the speed of direct hardware access. It is not recommended that applications use the monitor mode interface unless the optimal performance of a display is required.

Graphics Support Library. The Graphics Support Library (GSL) allows applications to perform graphics operations without the need to directly address the

display adapter hardware as in the monitor mode interface. Additionally, GSL supports the display of fixed-space characters within text lines.

GSL assumes that each application runs in its own virtual terminal using monitor mode. It provides an interface that allows a program to generate graphics interactively without detailed knowledge of the display adapter and input data formats.

The GSL routines are a set of graphics output functions that permit applications to write directly to a frame buffer—a memory storage containing a representation of a display image. A set of attributes govern the GSL functions and determine the characteristics of the display image. For example, color display adapters may be considered to have multiple storage planes; each plane acts as a single frame buffer of a monochrome display. An attribute identifies which planes of the frame buffer GSL functions will modify. Additionally, some attributes affect only a class of GSL functions, such as line style and color.

GSL provides control functions for initializing, locking, unlocking, and terminating the virtual terminal of an application, and provides various output, service, block transfer, and input functions. Some of the GSL output functions allow applications to draw straight and curved lines, mark points, write annotated and geometric text, and fill areas. GSL routines allow the definition of circular or elliptical arcs. These functions convert circular or elliptical arc definitions into a set of vertices that can be displayed via polyline draw or fill functions to produce more complex shapes. GSL also provides functions to move rectangular blocks of pixels to or from the display frame buffer to storage, or within the frame buffer or storage.

For simplicity and optimized performance, GSL does not perform any general clipping or transformation on coordinates. Most of the output functions convert coordinates as necessary to the target required for the frame buffer of the particular display. Thus, the coordinate system is device-dependent, and applications written using GSL need to be aware of the physical attributes of the display.

GSL accepts input from several sources—keyboard, locator, lighted programmed function keys, valuator, or pick device. Input from these devices is viewed as a series of discrete events, with input data associated with each event. GSL provides subroutines to enable or disable input from any device and to permit the application to suspend execution until an event occurs from an enabled input device.

GSL is a powerful tool for writing sophisticated graphic applications. This efficiency requires that applications understand the underlying display hardware capabilities, but it insulates them from dealing directly with the display hardware.

X-Windows. The X-Window System was developed at the Massachusetts Institute of Technology to provide high-performance/high-level device-independent graphics capability supporting a windowing interface on UNIX systems. This system was the basis for the RT PC X-Windows.

The RT PC X-Windows uses the functions of GSL and is a tool designed to help enhance the usability of the application-processing environment. Facilities are provided for users working with existing applications as well as for developers who are designing and implementing new applications.

X-Windows allows multiple application processes to operate within a given window and have multiple simultaneous windows on a display. These windows can be managed by the user or by the application. Each window may be hidden, with only an icon to note its existence, or it may be completely obscured, or partially obscured by other windows. Obscured windows can still be updated by the user or the application program (see Scheifler and Gettys[9]).

Each window can have a specific character style and size associated with it. Additionally, each window can have its own keyboard mapping, permitting different character sets to be supported in each window for international languages. Users can open and close windows, change window size and location, and move a window to the foreground or background.

Additionally, X-Windows provides the ability to manage local and remote windows through a client-server model. This provision allows remote display management of processes executing on other RT PCs connected via Ethernet™ or Token Ring local-area networks supporting TCP/IP.

X-Windows is designed to support character-based applications as well as graphic applications. Most character-based applications can run unmodified in a window. For graphic applications, X-Windows

provides an extensive set of graphic primitives. Each window is able to support both text and graphic data simultaneously.

Included in the X-Window program product is a set of library routines to interface with applications to support X-Window servers. These routines provide services or access functions to

- Establish the terminal server for users and applications on the same RT PC or other RT PCs in the network.
- Provide support for applications that operate in KSR mode.
- Provide automatic user log-on to the windowing system at the beginning of a terminal session.
- Print window image to a bit-map printer.
- Build keyboard mapping file to correlate keyboards to character sets.
- Display time in a separate window in either digital or analog form.
- Open and monitor full screen (virtual terminal) applications from inside a window.
- Provide access control for the display.

X-Windows provides application developers with the ability to build easy-to-use interfaces on a single display.

X-Windows provides application developers with the ability to build easy-to-use interfaces on a single display without worrying about the underlying device characteristics. Additionally, with the client-server model of X-Windows, these applications can be run remotely.

graPHIGS. graPHIGS is a tool that helps programmers in building complex, interactive graphic applications. It was developed using the facilities of GSL to provide a device-independent interface for extremely sophisticated graphic applications.

graPHIGS is based on the American National Standards Institute (ANSI) proposed Programmer's Hierarchical Interactive Graphics System (PHIGS) standard. It supports the definition, modification, and display of hierarchically organized graphics data. This system supports three-dimensional capabilities to enhance the design and visualization process of the application. The ability to organize graphics primitives into hierarchical structures makes it easy to edit, modify, and transform graphic entities.

graPHIGS is ideally suited for developing complex interactive graphic applications such as computer-aided design/computer-aided manufacturing, computer-aided engineering, and robotics.

Distributed Services. The RT PC Distributed Services extends the basic capabilities of the AIX system available to software developers across a local-area network of interconnected RT PCs. Sauer et al.[10] describe in detail the design goals and implementation decisions of RT PC Distributed Services. Key goals include

- Local/remote transparency of the services which are distributed, including no noticeable performance degradation in the remote case, and no alteration of the basic AIX and UNIX operating system semantics.
- User isolation from network media and transport mechanisms. Distributed Services runs over SNA LU 6.2 on an Ethernet network. The design allows for possible future extensions to support other protocols, such as the IBM Token Ring network.
- Administrative control. This control includes the ability to administer a set of interconnected RT PCs as a single domain, or to independently administer machines such as servers or private machines. It also includes providing security and authorization capabilities sufficient to administer the distributed configuration.

Distributed Services provides distributed operating system capabilities including local/remote file system transparency, distributed message queues, and administration of the interconnected configuration.

Distributed data. Distributed Services uses remote mounts to allow users to mount filesystems on a different machine than the directory off which the filesystem is mounted. Once the remote mount is established, the files contained in the file system appear in the same directory hierarchy across the distributed configuration, and filesystem calls gen-

erally work identically regardless of whether the file is local or remote to the user. Authorization mechanisms are provided to allow an installation to control remote mounting of file systems.

Distributed processing. Distributed Services provides distributed process support via AIX message queues, using the extended message queue capabilities described earlier. A common use of distributed message queues is to allow transparent access across the local-area network (LAN) to a server process.

Common linkage conventions were established to support applications written in more than one programming language.

Other considerations. Other capabilities of the AIX system of interest to application developers include floating-point support, interlanguage calling, and shared libraries.

Floating point. The RT PC main processor, being a RISC processor, does not contain floating-point instructions. Floating-point computations can be performed either by one of the three supported floating-point processors—the Advanced Floating-Point Accelerator (AFPA), the MC68881, or the Floating-Point Accelerator (FPA)—or emulated by software.

In the AIX system, the object code of applications can be independent of the floating-point processor without sacrificing significant floating-point performance. This is accomplished by the cooperation of the AIX compilers and the kernel through the so-called compatible mode interface.

When the kernel is initialized at initial program load (IPL) time for a machine, the kernel tests for the kind of floating-point hardware present. Depending on the result of this test, it loads floating-point routines needed for various operations, always selecting the fastest routine possible for the hardware combination. Fixed memory locations are assigned to most frequently used operations where the corresponding routines are loaded. Also, the register usage of these routines is known to the compilers.

When a program is compiled in compatible mode, the compilers generate code to invoke the kernel routines for floating-point operations. For most-frequently-used operations, these invocations are faster than subroutine calls. A direct branch is possible since the routine is at a fixed location, and only those registers that are used by the routine (which is known to the compiler) need to be saved. Thus, with minimum overhead, the compatible mode provides object code portability between different machine configurations.

Some applications prefer greater execution speed over object code portability. The AIX compilers can optionally generate AFPA instructions to meet this need.

Interlanguage calling. Many applications on the RT PC are written in more than one programming language. In order to make this possible, a common linkage convention was established. The linkage convention specifies register usage rules, program stack structure and usage rules, and identifies the responsibilities of the calling and called routines. All the AIX compilers follow the common linkage conventions in generating code for calls and for procedure entry and exits, making calls between routines compiled with different compilers possible.

However, the languages themselves impose some problems in interlanguage calls. Data types of one language are not always understood by another language. Therefore, the programmer has to be aware of the semantics of both languages involved in an interlanguage call.

Although interlanguage calling is currently possible, it can be inconvenient at times, especially when the called routine was not designed to anticipate such a call. This is true for some of the original System V library routines. New libraries provided with the AIX system (such as GSL) were designed to be callable from FORTRAN, Pascal, and C.

Shared libraries. A UNIX library is a file constructed from one or more object files using the archive utility *ar*. Usually the object files in a library are related by their function, such as the math library, or their usage, such as a language-specific library. If a program references a member in a library, the library is

linked with the object files of the program and becomes part of the executable file. When a library is referenced by multiple processes, each process has a private copy of the library files. When a large library is often used by many processes, the redundant copies lead to the following effects:

- Too much disk space is used by the multiple copies.
- The number of page faults increases.

Although the RT PC provides large virtual memory and disk space, it also supports very large applications where the redundancy of code shared by multiple processes is not acceptable. Therefore, the AIX system was extended to reduce this redundancy by enabling multiple processes to share one copy of an object file or library.

The AIX operating system uses its mapped file feature to implement sharing of libraries. An AIX object file contains a text segment and a data segment. The executable code is placed in the text segment. The initialized data used by the program and external references are placed in the data segment. When an object file is to be shared among many processes, it is transformed by a utility, *shlib*, which strips the text segment from the object file and moves it to a shared library. When a program is linked with a shared library, it gets a copy of the data segments of the members in the shared library, but the text is not duplicated.

In addition to separating the program text and data sections, *shlib* and the *linker* (*ld*) manipulate the external references so that a program can branch and return to the right place when referencing a shared routine. The linker also builds a table of shared libraries used by the program and the segment number it has assigned to it. At execution time, the kernel looks at this table and maps the shared libraries into the assigned segment.

In the spirit of the UNIX philosophy, the AIX system is its own user, and many of the library routines provided with the AIX system are shared.

AIX programming tools

The capabilities of the AIX system on the RT PC described in the previous section make it an attractive base for applications. In order to facilitate application development on the RT PC, a wide array of program development tools have been provided, building on the UNIX System V tool base with extensions in several key areas.

Editors. The AIX system provides three editors for entering and modifying programs: a line editor, *ed*, and two full-screen editors, *vi* and INed.™ Each suits the needs of a particular class of users and provides facilities to manage the editing of program source material.

Line editor. The simplest editor is *ed*, a line-editing program that allows users to work with the contents of a file in line mode. This editor works on the principle of an edit buffer—a temporary storage area that holds the file and modifications during editing. *ed* provides all the typical line-editing functions, such as appending data to the end of the edit buffer, manipulating the current line pointer, and displaying, locating, inserting, deleting, moving, and changing lines. It is useful for fast efficient editing of small programs or editing programs from printing TTY devices.

Full-screen editor vi. Based on the *ed* editor is the screen editor, *vi*. It was developed by William Joy of the University of California at Berkeley as part of the Berkeley UNIX project to remedy the shortcomings of *ed*. Whereas *ed* is simple and understands few commands, *vi* is extremely powerful and understands over a hundred commands. This difference makes *vi* difficult to learn and use.[11] However, two subsets of *vi*, *ex* and *edit*, are also provided with the AIX system for the novice or casual user.

The concept of editing files with *vi* is similar to *ed*. The file is copied to the edit buffer and edited there. However, *vi* provides a "window" on the edit buffer to display lines on a display screen. *vi* can be used to move this window around the edit buffer and to display it on the screen. Editing operations of *vi* are determined by the screen cursor position, consisting of a current line and offset within that line. In order to efficiently support glass TTY devices that send and receive a keystroke at a time, *vi* allows movement of a character as well as a word at time within the current line.

Commands are entered when *vi* is in the "quiescent state," i.e., when not inserting text at the screen cursor position. All editing commands are analogous to those of *ed* and, additionally, *vi* offers an "undo" command which can reverse the effects of most incorrect command execution.

The advantages of the *vi* editor are that it is efficient on slow-speed glass TTY devices and, even though it may be difficult to learn, once learned it is available on almost all UNIX systems.

INed full-screen editor. The other AIX full-screen text editor is INed. INed was developed by INTERACTIVE Systems Corporation and has the same concept as the *vi* editor of an "editor window" that is scrolled (positioned) on top of a file. The screen cursor position is changed by the cursor movement keys.

INed provides all of the standard facilities of a text editor for inserting, deleting, copying, and moving lines and blocks of text. INed, in addition, provides a menu-driven interface through menu and message boxes. The hierarchy of these menus is controlled through a zoom facility to display the available options at the cursor position. Associated with the menus and messages of INed is an extensive help facility for assisting the user.

Another feature of INed is the capability to divide the editor window into two or more windows for editing the same file at two or more places, editing two or more files at one time, or copying text from one file to another file.

INed has an additional feature to assist users in the manipulation of files within the AIX file directories through the file manager screen. With the file manager and command keys, one can create new files and directories, rename or delete existing files and directories, access or copy one of the displayed files or directories, and move a directory or a file to a different location in the file tree.

Text editing. The AIX system contains the UNIX text-editing utility programs, which are useful for modifying programs on the RT PC. Some of the more useful utility programs are *grep*, *egrep*, *fgrep*, *awk*, and *sed*.

The *grep*, *egrep*, and *fgrep* utilities are used to search one or more files at a time. These programs determine such information as in which files a string occurs, on how many lines in each file the string occurs (and the numbers of the lines), which lines in which files do not contain a string, and the disk block number of each place where the specified string occurs.

The *awk* utility finds and changes strings in text files. In addition, it provides numeric processing, variables, more general pattern selection for finding strings, and flow control statements. In general, *awk* is useful for processing input to find numeric counts, sums, and subtotals, verifying that a field contains only numeric information, processing data contained in fields within lines, and changing data from one program into a form that can be used by a different program.

The AIX system has a number of alternatives available for text processing.

The *sed* utility is a batch text editor that has functions similar to those of *ed*. It receives input from standard input, changes the input as directed by commands in a command file (or on the command line), and writes the resulting stream to standard output. This method of operation allows *sed* to edit very large files, perform complex editing operations many times without requiring extensive typing and cursor positioning, and perform global changes in one pass through the input.

Text processing. A programming environment is incomplete without text-processing capabilities. Normally, in a development process more documents are written than code. With the AIX system, a number of alternatives are available for processing text, including traditional UNIX tools such as *nroff* and *troff* as well as the modern typesetting system TEX™ and METAFONT.™

The *nroff* and *troff* formatters accept the same requests. *nroff* generally assumes that a printer has a single font and type size available at a fixed pitch, whereas *troff* assumes that there are four fonts available in 15 type sizes and uses smaller internal units leading to higher resolution.

The filters *mm*, *eqn*, and *tbl* make *nroff* and *troff* easier to use. Memorandum Macros (mm) is a macro

package that predefines formatting rules for various document styles, from simple memorandums to books. For formatting mathematical equations the filter *eqn* is available, and for formatting tables a filter called *tbl* is used. These filters are preprocessors for the text formatters and together provide the ability to format fairly complex documents.

The typesetting system for the RT PC contains TEX, METAFONT, LATEX, and other programs that may be used in conjunction with TEX. It also contains the necessary fonts and the programs for viewing the formatted output.

TEX[12] is the well-known typesetting system of Donald Knuth. Its powerful typesetting language has been traditionally used for writing books because the output looks just like a published book. LATEX[13] provides another layer on top of TEX and is designed to make TEX easier to use. The relationship between LATEX and TEX can be compared to that of a Generalized Markup Language and the Script/VS text formatter used on IBM mainframe computers. With LATEX, use of TEX for technical documents other than books is increasing.

The type fonts used by TEX and LATEX were designed by Donald Knuth and generated by his METAFONT program,[14] which consists of an entire family of type styles designed to look good when used together. The output from TEX is device-independent, making LATEX and TEX documents highly portable from one system to another. The device-independent output from TEX, and the fonts generated by METAFONT are then combined by a program and sent to the appropriate device. Because the fonts are generated by METAFONT, the characters look the same (except for the resolution) across different devices.

To the power and portability provided by TEX and LATEX, implementation of the typesetting system on the AIX system adds formatting speed and the capability to preview the formatted document on bitmapped displays available on the RT PC before printing it on a supported printer. The TEX source, which is in Pascal, and the view and print programs written in C were compiled using the optimizing PL.8 compilers described in the following section.

Compilers. Compilers on the RT PC are designed with two major goals. The first goal is to generate highly optimized code so that the programmer can focus on algorithm selection and program design, and not on tuning the code for a specific machine. The second goal is to recognize existing language dialects above and beyond the ANSI standards so that the programmer can easily port existing programs from other machines to the RT PC.

Optimized code generation. Since the AIX system is based on the UNIX system, the RT PC C compiler is based on the Portable C Compiler (PCC; see Johnson[15]), and the RT PC FORTRAN compiler is based on the UNIX FORTRAN 77 compiler. The quality of the code generated by the PCC and FORTRAN 77 compilers did not meet the first goal described above, that of producing optimized code. Therefore, a global code optimizer based on the PCO (Portable C Optimizer; see Kelly and McIntosh[16]) was added to both the PCC and FORTRAN 77 compilers. The global code optimizer improves the quality of the code by optimizing the intermediate code before the generation of assembly language code. Then a peephole optimizer improves it further by making local optimizations, such as reducing the number of branches. Finally, an "inliner" was added which can combine several subroutines together as one, thus extending the scope of the global optimizer.

The global optimizer eliminates unused variables and evaluates expressions involving constants whose values are known at compilation time, replacing them with their results. It analyzes loops and moves loop invariants (code that would produce the same result if it were outside the loop) out of loops. It identifies common subexpressions within a program and determines whether recomputing the subexpression is more time-consuming than storing and reusing its value. If so, it replaces the former with the latter. The optimizer also allocates registers to variables and intermediate results.

A further advance toward meeting the goal of producing optimized code is the Advanced C Compiler. This compiler is based on the PL.8 compiler (see Auslander and Hopkins[17]). The PL.8 compiler and the RT PC processor were developed together and for each other, each exploiting and influencing the design of the other. The algorithms used for global optimizations and especially for register allocation (see Chaitin et al.[18]) lead to excellent code for the RT PC, with the efficiency of the code generated by the Advanced C Compiler approaching that of code written in assembly language by an experienced assembly language programmer. The Advanced C Compiler was initially developed as an internal tool

for compiling the AIX operating system and is now available externally.

In addition to understanding program flow and making global optimizations, the AIX compilers also try to minimize subroutine call overhead. Normally, parameters passed to a routine are placed in the program stack. The called routine has to load it into a register to operate on it. The register usage conventions were established to minimize the number of loads and stores during a subroutine call and return. First of all, the first four parameters are placed in general-purpose registers rather than on the program stack. The called routine seldom has to move them elsewhere. Second, two sets of registers are identified, a volatile set used to store values with short lives, such as the intermediate result of a computation, and a nonvolatile set used to store values having longer lives that must not change across subroutine calls. The called routine only saves the nonvolatile registers it uses, which in many cases is none.

The AIX compilers try to minimize subroutine call overhead.

Support for existing dialects. The second goal, that of accommodating existing language dialects, led to the RT PC VS FORTRAN and the RT PC VS Pascal compilers. The RT PC VS FORTRAN compiler provides a VS dialect with most of the IBM extensions to FORTRAN and a VAX™ dialect with most of the DEC™ extensions to FORTRAN. Similarly, the VS Pascal compiler matches the language level of the Pascal/VS compiler on the System/370. Thus, applications written in FORTRAN and Pascal can be ported to the AIX system on the RT PC with few changes. It is possible for application developers to maintain consistency of source code among different versions of applications running on different machines.

Program maintenance facilities. The AIX system provides two UNIX facilities to assist programmers in the orderly development, building, and maintenance of a collection of files that comprise a program. These are the *make* program and the Source Code Control System (SCCS) library.

Make. The *make* program assists users in building up-to-date versions of programs. A user of *make* supplies a description file containing a set of rules that specify how executable programs are to be constructed. *make* uses time stamps to determine which source files have changed. It only builds those target files that are out-of-date and does not build files that are already current.

The *make* program is most useful for small to medium-sized programming projects.

Source Code Control System. SCCS gives program administrators the ability to control and account for changes made to source code and documentation files. It stores changes made to a file instead of storing the entire file. This method allows several versions of the same file to exist in the system. SCCS can then build versions of a file based on stored information about any previous changes.

SCCS forms a complete library system for a program repository. As such, it has its own set of commands and programs for manipulating the SCCS library. These SCCS commands allow one to

- Create an SCCS library.
- Get a version of a file stored in the SCCS library and save changes made as a new version.
- Define who can change a file and record when and why changes were made to a file.

Files stored in the SCCS library have two major sections, called the header and the body. The header has five subsections which identify who created the file, who can change it, and other administrative details. The body has one or more subsections consisting of the text portions of the file. A set of changes made to a file stored in SCCS is called a delta. Each delta is assigned an SCCS Identification (SID). An SID has up to four parts, as shown in Figure 4.

Most SCCS libraries only use release and level numbers and grow in a straight line. In these cases, the latest version of a file uses every previous delta to construct the contents of a file. However, SCCS also supports the concept of a branch, where versions of files consist only of a subset of all the deltas. For example, a common file can be used by two different

releases of a programming project. A branch is made when development work starts on the new release and maintenance changes are isolated against the original release without affecting the new development work on the file. In this case, a branch delta allows each release to add deltas to a common base. Only one text portion for each delta of the file is stored on SCCS.

The three major commands that manipulate files stored on an SCCS library are *admin*, *get*, and *delta*. The *admin* command is used to create an SCCS library or to change the characteristics and permissions of an existing SCCS library. The *get* command obtains a specified version of a file from the SCCS library for editing or compilation, and the *delta* command adds sets of changes (deltas) to the text portion of the file stored on the SCCS library.

Using SCCS reduces the storage requirements and helps track the development of a project that requires keeping many versions of large programs. Together, the facilities of SCCS and the make program allow teams of programmers to effectively maintain and build large and complex programs.

Debuggers. The function of a debugger is to help the programmer find and correct errors in application programs. A symbolic debugger usually has the following capabilities:

- Setting breakpoints in a program at source level
- Displaying program variables and their attributes
- Modifying the value of variables
- Controlling execution of the program
- Executing the program one source statement at a time as well as one machine instruction at a time
- Accessing the source file through an editor
- Keeping the history of program execution
- Displaying the status of the debugger

Debuggers vary in how well they perform in each of the above functions. The AIX system provides the UNIX System V symbolic debugger *sdb*. Although *sdb* is useful for debugging some programs, it fails when it comes to debugging complex applications with multiple processes. Also, *sdb* is not very user-friendly, and its structure makes it difficult to modify or enhance. On the basis of a survey of several debuggers, the UNIX 4.2 BSD debugger *dbx* has been chosen as the base for a prototype debugging environment. This prototype environment provides a better user interface and the ability to debug multiple processes.

Figure 4 SCCS identification (SID)

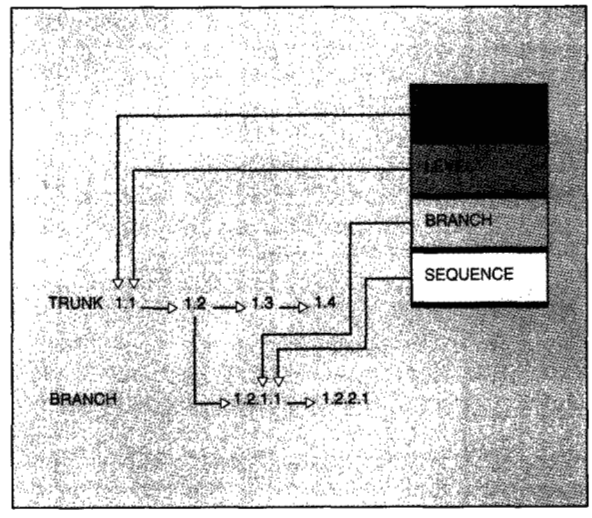
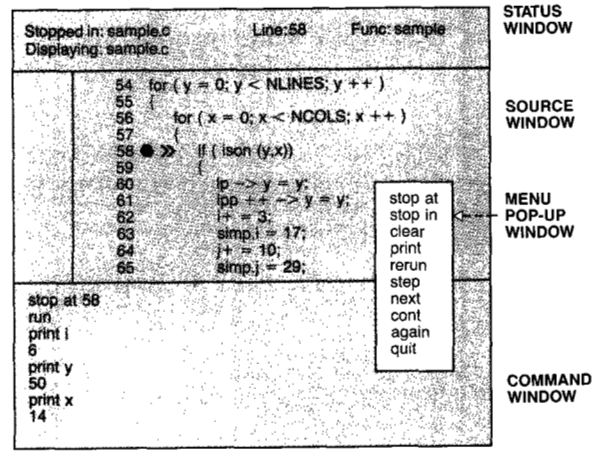


Figure 5 xdbx



Debugger interface. First, *dbx* was ported to the AIX system by making it understand the AIX debug tables. The next step was to provide a more convenient user interface for interactive debugging of programs. This was done by providing a multiwindow environment called *xdbx* that runs as an application of the RT PC X-Windows. The *xdbx* environment contains several windows, some of which are shown in Figure 5.

Shown at the top is the STATUS window, which displays status information such as the file currently being processed and the place where the program stopped.

The window shown directly below the status window is the SOURCE window, which automatically displays the source code around the line where the debugger is currently stopped. It displays markers indicating the location of breakpoints and the current line. The source window is also used to display any part of any source file as the user wishes.

The MENU window pops up when the proper button on the mouse is depressed. It displays *dbx* commands in the form of a menu. One can use the mouse to select a particular action from the menu. For example, to set a breakpoint at a line, select the line from the source window and then select the stop at command from the menu. To see the contents of a variable, select the variable from the source window and then select the print command from the menu.

When *dbx* commands are selected with a mouse from the menu window, they are echoed in the COMMAND window shown below the source window. The user may also type any *dbx* command here if the keyboard is preferred to the mouse or if the command is not selectable with the mouse.

A MESSAGE window (not shown) pops up when *dbx* or *xdbx* detects an error. The error messages are displayed in this window.

A TRACE window (not shown) pops up to automatically display the result of certain actions, such as the trace command.

The user may tailor *xdbx* by modifying the *xdbx* parameters such as the font for each window, the position of each window, the size of each window, and the name of the editor to be invoked as a result of the *edit* command. This is done by modifying the default settings in the *xdbx* profile in the user's home directory (i.e., the default directory when the user logs in to the AIX system).

Multiprocess debugging. Normally, in a UNIX environment a debugger and the program it is debugging run as parent and child processes. For a debugger to be able to debug multiple processes, a debugger should be able to debug a process that is not a child. To provide this operation for the prototype environment, the AIX kernel is modified to support a new kind of relationship called "attaching." The *dbx* debugger can debug a program that is running as a child process or as an attached process. When a user process exits, the parent as well as the attached *dbx* is notified.

When the process *dbx* is debugging forks, *dbx* also forks. The parent *dbx* still is the debugger of the user program, running in the parent user process. The child *dbx* then forks again. The child of this fork, which is a new instance of *dbx*, is attached to the child of the user process. The parent of this fork then invokes *xdbx*. Thus, there is an *xdbx*/*dbx* pair for each process that needs debugging. This scenario is illustrated in Figure 6. The multiprocess debugging capability does not require *xdbx* and can operate in a virtual environment without *xdbx*.

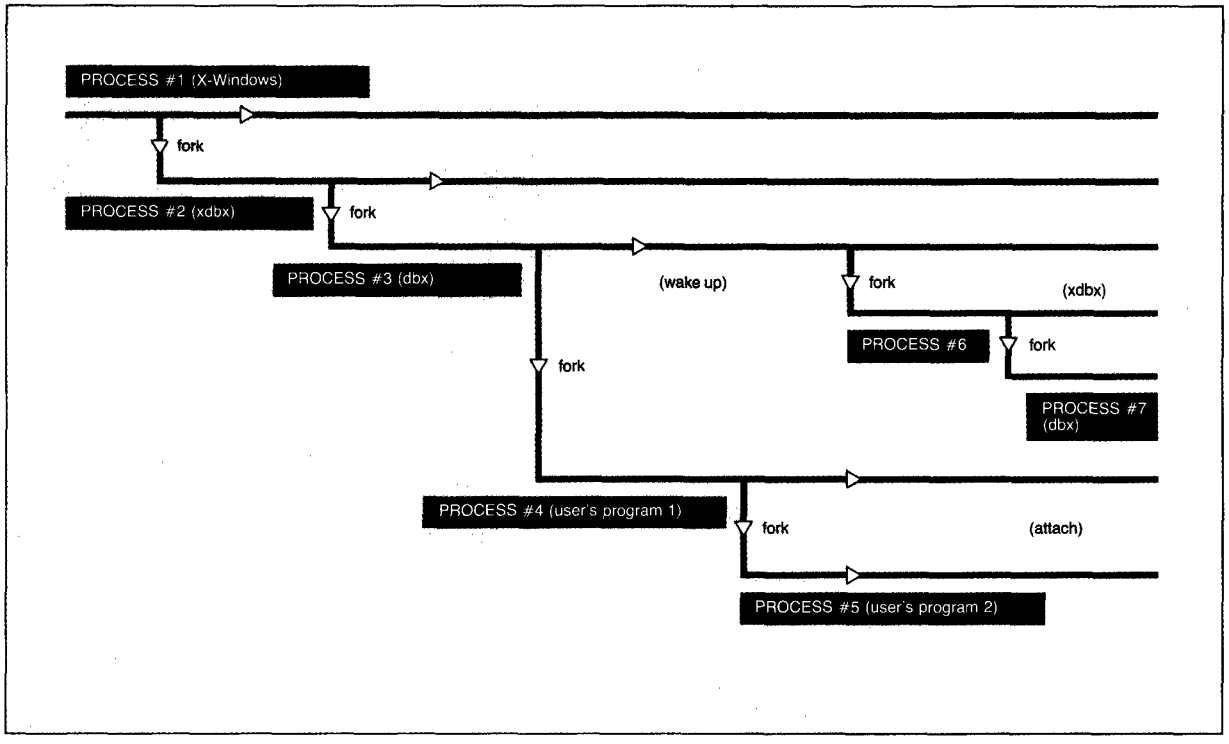
The profiling feature collects execution time data.

Other debugging tools. In addition to the symbolic debuggers, there are a few other UNIX tools that one may find useful for debugging, particularly if the symbolic information needed by the symbolic debuggers was not created or was removed because of its large size. An absolute debugger *adb* is available that can set breakpoints and display parts of a program, taking absolute addresses as input. A *dump* program produces formatted dumps of various parts of an object file. For debugging the kernel itself, a utility called *crash* is provided. *crash* works interactively with system programmers and examines and analyzes an operating system image created automatically at the time of a kernel crash.

Monitoring program activities. In large applications one often wants to identify the key areas of the application, where most of the execution time is spent. This is not usually intuitively obvious. The profiling feature collects execution time data such as the number of times a routine is called, and the average time spent in the routine. The monitoring activity is invoked by the compilers and the linker. The collected data can be examined using the *prof* command.

There is also a way to trace key events occurring in an application. Trace points can be placed at strategic places in the execution path. Each trace point

Figure 6 Multiprocess debugging using xdbx



belongs to an event class, which can be activated or deactivated. During execution, trace points of all active event classes generate trace entries that are logged in a trace log file. There is a trace formatter that formats the trace entries in the trace log file into a readable format. Predefined trace points of various events exist in the kernel components, which can be traced when needed.

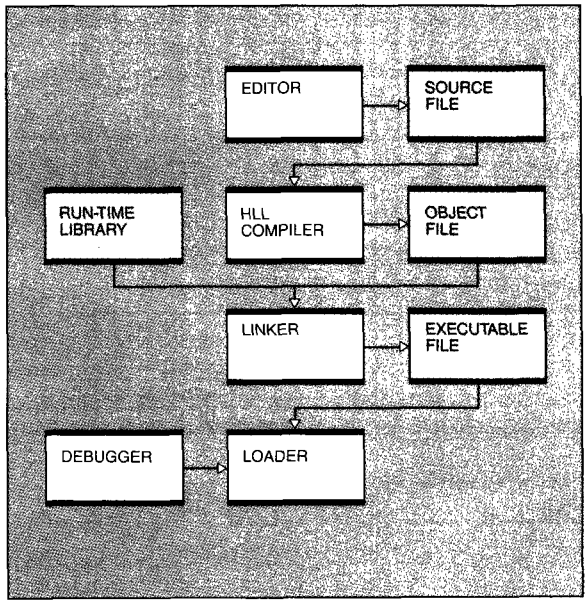
Similar to the trace facility is the error log facility to record the errors occurring in the system. The errors can be in hardware or software and can be of many different types. The error logging and the trace functions are available to the user through the run-time library.

Run-time libraries. The tools are usually accessed through commands issued from a shell. The AIX system also provides numerous services in the form of system calls and subroutines in various system libraries. These services are used by the tools and application programs.

System calls. System calls provide controlled access to the kernel. To the programmer, a system call looks exactly like a subroutine call. Normally, a program runs in user mode. When a program makes a system call, a mode change takes place, causing the system routine to run in kernel mode, which allows access to information maintained by the kernel. Only very basic primitives are identified as system calls. They perform functions of the following categories:

- Basic input/output functions for all types of devices
- File maintenance functions such as creating directories, mounting file systems, and changing access permissions
- Process control functions such as creating, operating, stopping, and identifying processes
- Interprocess communication functions such as using signals, semaphores, and message queues
- Memory management functions such as mapping files and sharing memory segments

Figure 7 AIX program development environment



- Functions to set and obtain the time from the system clock

System libraries. The AIX system provides a number of subroutines functionally grouped into libraries. These subroutines use system calls to perform more complex tasks. User applications use the library functions more than the system calls. The more general-purpose libraries are discussed below. Some of the special-purpose libraries were discussed earlier.

The C library, although initially designed for C usage, contains most of the general functions all programming languages use. The routines in this library perform input/output control functions, string and character manipulation functions, and many other miscellaneous functions such as hashing and searching binary trees.

The run-time services library provides services to configure minidisks and other devices and to log run-time error messages. It also provides services to record trace log entries and to retrieve predefined messages.

The math library consists of trigonometric functions such as sine, cosine, and tangent, other commonly used functions such as power, exponential, logarithm, and square root, and also gamma, bessel, and hyperbolic functions. Initially, the math library was the same as the UNIX System V math library. The speed of many of the math library routines is very critical to graphics applications. Therefore, these routines were moved into the floating-point services in the AIX kernel. The Advanced Floating-Point Accelerator has built-in instructions for many of these functions. The AIX kernel math routines use such functions when the AFPA is present, and are therefore very fast. Also, the math library routines in the kernel use the algorithms used by the routines in the math library of the UNIX 4.2 BSD system, which are mathematically more complete in the sense that they can operate on all floating-point numbers defined by the IEEE Standard for Binary Floating-Point Arithmetic[19] to produce accurate results.

AIX application development and build environments

The AIX program development tools described in the previous section combine to provide a powerful and integrated set of facilities, from editing to debugging. The relationship among these tools is illustrated in Figure 7. At each step, the AIX system has extended the capabilities of traditional UNIX systems, with features such as more sophisticated compiler optimization.

Maintenance of source code is an important part of any development cycle. The AIX program build and maintenance facilities provide the ability to manage source program libraries, including versioning and change management, and to merge the appropriate versions and modifications at build time. The resulting source program is then used as input to the appropriate high-level language compiler. This scenario is shown in Figure 8.

The combination of these tools results in an extremely effective and friendly program development environment. As stated at the beginning of this paper, this was a key requirement for the RT Personal Computer, given the capabilities of the system, which make it an attractive target for a variety of applications.

Concluding remarks

The UNIX system has proved over time to be an extremely good base for application programs, providing a combination of flexibility and function to the application programmer without enormous complexity. This fact, along with the historical evolution

of the UNIX system as a base for programmer activity, has led to the development of a strong application support base for application programmers. These reasons, among others, made UNIX System V a strong choice as the base for the AIX operating system on the RT PC.

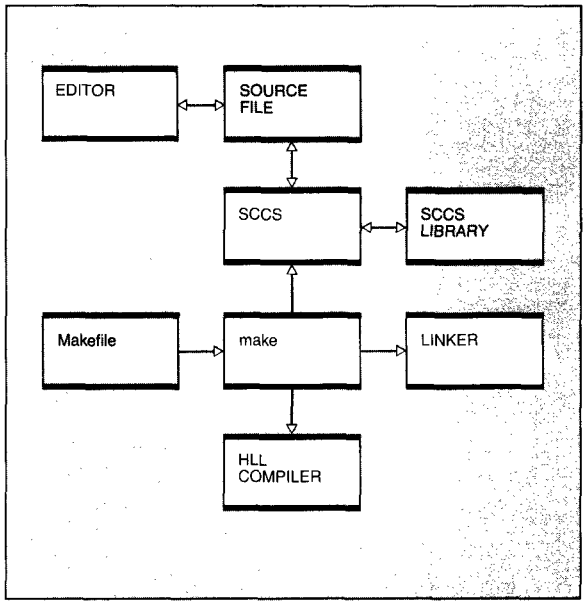
The AIX system, in turn, extends the basic UNIX capabilities for both application programs and application program development. As the operating system exploits the power of the RT PC hardware and provides enhanced capabilities to support sophisticated application programs, the program development environment is enhanced to better support the development of applications to take advantage of these capabilities. The result is a system that combines the advantages of the UNIX operating system and UNIX program development tools with new features required to support commercial applications. This combination is critical in supporting the existing variety of applications on the AIX system, such as desktop publishing (Workstation Publishing Software), artificial intelligence (LISP and the KEE™ expert system shell), office systems (Applix IA™), CAD/CAM (CATIA™, CIEDS™, AutoCAD®), and decision support (Solomon III™).

These capabilities make the AIX system on the RT PC attractive both for developing applications and for running applications which have previously required the power of a large system environment, without the concomitant complexity.

Cited references

1. G. G. Henry, "IBM small-system architecture and design—Past, present, and future," *IBM Systems Journal* **25,** Nos. 3/4, 321–333 (1986).

Figure 8 AIX program build environment



2. L. K. Loucks and C. H. Sauer, "Advanced Interactive Executive (AIX) operating system overview," *IBM Systems Journal* **26,** No. 4, 326–345 (1987, this issue).
3. G. R. Radin, "The 801 minicomputer," *IBM Journal of Research and Development* **27,** No. 3, 237–246 (1983).
4. M. D. McIlroy, E. N. Pinson, and B. A. Tague, "UNIX time sharing system: forward," *Bell System Technical Journal* **57,** No. 6, Part 2, 1899–1904 (1978).
5. T. L. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," *Bell System Technical Journal* **57,** No. 6, Part 2, 2177–2200 (1978).
6. D. M. Ritchie, "A retrospective," *Bell System Technical Journal* **57,** No. 6, Part 2, 1947–1970 (1978).
7. J. M. Bissell, "Extended file management for AIX," *RT Personal Computer Technology* (pp. 114–125), SA23-1057, IBM Corporation (1986); available through IBM branch offices.
8. T. Murphy and D. Verburg, "Extendable high-level AIX user interface," *RT Personal Computer Technology* (pp. 110–113), SA23-1057, IBM Corporation (1986); available through IBM branch offices.
9. R. W. Scheifler and J. Gettys, "The X-Window System," *Transactions on Graphics,* Special Issue on User Interface Software, Part I, Association of Computing Machinery **5,** No. 2, 78–109 (April 1987).
10. C. H. Sauer, D. W. Johnson, L. K. Loucks, A. A. Shaheen-Gouda, and T. A. Smith, "RT PC Distributed Services overview," *ACM Operating Systems Review* **21,** No. 3, 18–29 (July 1987).
11. P. P. Silvester, *The UNIX System Guidebook*, Springer-Verlag, New York (1984).
12. D. E. Knuth, *The TEXbook*, Addison-Wesley Publishing Co., Inc., Reading, MA (1984).
13. L. Lamport, *LATEX—A Document Preparation System*, Addison-Wesley Publishing Co., Inc., Reading, MA (1986).
14. D. E. Knuth, *The METAFONTbook*, Addison-Wesley Publishing Co., Inc., Reading, MA (1986).

15. S. C. Johnson, "A tour through the portable C compiler," *UNIX Programmer's Manual*, Volume 2, Bell Telephone Laboratories, Inc., Murray Hill, NJ (1979).
16. T. J. Kelly and A. McIntosh, "A portable intermediate code optimizer for C," *USENIX Conference Proceedings* (Summer 1985), pp. 577–589.
17. M. Auslander and M. E. Hopkins, "An overview of the PL.8 compiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing*, Boston (June 23–25, 1982), pp. 22–31.
18. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, P. E. Hopkins, and P. W. Markstein, "Register allocation via graph coloring," *Computer Languages* **6**, No. 1, 47–57 (1981).
19. *IEEE Standard 754 for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY (1984).

General references

M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).

S. R. Bourne, "The UNIX shell," *Bell System Technical Journal* **57**, No. 6, Part 2, 1971–1990 (1978).

D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Bell System Technical Journal* **57**, No. 6, Part 2, 1905–1930 (1978).

IBM RT PC AIX Operating System Programming Tools and Interfaces, Version 2.1, SC23-0789, IBM Corporation; available through IBM branch offices.

IBM RT PC AIX Operating System Commands Reference, Version 2.1, SC23-0790, IBM Corporation; available through IBM branch offices.

IBM RT PC AIX Operating System Technical Reference, Version 2.1, Volumes 1 and 2, SC23-0808 and SC23-0809, IBM Corporation; available through IBM branch offices.

IBM RT PC AIX Operating System Managing the AIX Operating System, Version 2.1, SC23-0793, IBM Corporation; available through IBM branch offices.

IBM RT PC Using the AIX Operating System, Version 2.1, SC23-0794, IBM Corporation; available through IBM branch offices.

IBM RT PC INed, Version 2.1, SC23-0799, IBM Corporation; available through IBM branch offices.

Introducing graPHIGS, SC23-8100, IBM Corporation; available through IBM branch offices.

Understanding graPHIGS, SC23-8102, IBM Corporation; available through IBM branch offices.

IBM RT Personal Computer Technology, SA23-1057, IBM Corporation (1986); available through IBM branch offices.

Robert Q. Cordell II *Entry Systems Division, 11400 Burnet Road, Austin, Texas 78758.* Mr. Cordell is manager of application development architecture in the Advanced Engineering Systems Development organization.

Mamata Misra *Entry Systems Division, 11400 Burnet Road, Austin, Texas 78758.* Ms. Misra is an Advisory Programmer in the Advanced Engineering Systems Development organization.

Roger F. Wolfe *IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, New York 10598.* Mr. Wolfe is a member of the Research Division currently on assignment to IBM's Entry Systems Division, Austin, Texas.

Reprint Order No. G321-5302.