The future of minicomputer programming

by D. J. WAKS and A. B. KRONENBERG

Applied Data Research, Inc.
Princeton, New Jersey

"Here is Edward Bear, coming downstairs now, bump, bump, bump, on the back of his head, behind Christopher Robin. It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way, if only he could stop bumping for a moment and think of it. And then he feels that perhaps there isn't."

Beginning of *Winnie-the-Pooh*, A. A. Milne

INTRODUCTION

The minicomputer syndrome

The programming of minicomputers to date has resembled Pooh Bear coming downstairs because although we feel that there really is another way, we rarely stop bumping. Programmers for small computers have long exhibited what we call "the minicomputer syndrome", which is a very low expectation of program support and facilities on a minicomputer and a lack of appreciation of the hardware and software tools one can employ to attack and solve a programming problem.

Minicomputers came on the market when it was first realized that small general-purpose stored-program computers could be competitive with complex hard-wired systems built from general-purpose logic modules. Not at all surprisingly, the manufacturers of logic modules quickly became the leading developers of minicomputers. Since the programming process was viewed as an unfortunately necessary step in making the mini behave like the hard-wired system it was replacing, very little general-purpose programming support was provided with the minis. Thus, the minicomputer syndrome—the belief that primitive computers require primitive programming support—was born virtually at the same time as the mini itself.

Computer manufacturers have fostered this syndrome by typically providing little general-purpose software* with their minis. Although manufacturers have recently attempted to provide reasonable program-creation software, the majority of available assemblers still lack such desirable features as macros, conditional assembly, literals, and multiple location counters. Program-execution software for most small computers is often so weak that the user usually is left to write his own disk handler, communications package, or real-time executive. Even in cases where a manufacturer does provide such software, it usually requires a configuration that is uneconomically large.

The manufacturers' software planners seem to exhibit the minicomputer syndrome more than many of their customers. They continue to design separate, single-application monitors apparently assuming that the user will employ his mini only for real-time applications, file processing, communications, or report writing. Thus, the user who wants a single monitor system to support some combination or all of these must either adapt a manufacturer-supplied monitor or build one from scratch.

This lack of coherent manufacturer software support has also been characterized by a lack of standardization and conventions or standardization of poor or unworkable conventions in the software. This in turn has led to a proliferation of software to dismay any ecologist. For example, how many different assemblers now exist for the DEC PDP-8 family? We were able to count a baker's dozen in two minutes of trying; there must be an equal number we didn't think of or don't know about.

Finally, this lack of standards and conventions has resulted in incompatibility from one user to another. Software modules created for one user's executive and assembler can be neither assembled nor run at another user's site. We have been involved in more than one

* General-purpose software can be classified as either program-creation software (text/file editors, language processors), or program-execution software (monitors, device drivers, real-time executives).

project requiring that a program be manually transliterated from one dialect for a computer to another.

What is a minicomputer?

The paper is intended to describe programming for minicomputers. What do *we* mean by "minicomputer"?

Let us begin by characterizing the *small computer* as having:

(a) A short word and register length, always 18 bits or less; a small computer, therefore cannot perform *single-precision* floating-point arithmetic through either the hardware or software.
(b) A main memory provided in modules of 8K words or less (typically 4K words).
(c) An instruction set and, commonly, its addressing mode which are restricted, usually having a rather short operation code field in its machine-language structure.

One more characteristic is required to define what we mean by a mini: if a machine passes the test for a small computer, and you can't ordinarily buy it without an ASR Teletype®, it is a *mini*. If you can't ordinarily buy it without a *card reader*, it *isn't* a mini, it's just a small computer.

The virtual machine and the extensible machine

Let us digress for a moment and define some useful concepts we will refer to throughout the remainder of this paper. The concept of the "extended machine" was first propounded by Holt and Turanski;[1] we will use the term "virtual machine" interchangeably with extended machine. Here we are using the term "virtual" in its dictionary[2] sense: "Virtual, apparent, as contrasted with actual or absolute." That is, the virtual or extended machine is what the computer user, on some level, sees and uses. Another way of saying this is to describe it as the *set of primitives* with which he must work. Watson[3] defines a "virtual processor" in a timesharing system in a similar way. The "virtual memory" concept in paged computers is a similar derivation.

As it is, the computer user rarely sees or uses the "actual or absolute" machine. To do so, he would have to write all his code in machine language. Language processors, executive systems, device drivers, *et al.*, are all extensions to the machine which substantially affect the user's view of the machine. For instance, a minicomputer with a disk operating system appears to be a very different machine from one with only paper-tape-oriented software. A mini with a manufacturer-provided real-time executive appears to be a very different machine from one without it to the user with a data acquisition and reduction problem. In this sense, then, all available software and hardware options act as real extensions to the "actual" machine.

When certain types of extensions are preplanned into the computer hardware and software design, the resultant machine is considered "extensible"—to the extent that such extensions are planned. We characterize these extensions as being expansions of primitives on three distinct levels: the hardware or "firmware" level; the system software level; and the applications software level. On each level, the user can implement new extensions, typically visible at the next higher level (further away from the hardware) although sometimes available at the same level.

The *first* and *lowest level* of extensibility is at the *hardware* or *firmware* level. By hardware extensibility, we mean the ability to add new operation codes, typically for I/O but also for CPU purposes. Thus, a *supervisor call* facility was added to a PDP-8 four years ago to provide for user-supervisor communication in a real-time system for TV control room automation. This form of extensibility is designed-in, and encouraged, although only rarely used to augment CPU primitives. By firmware extensibility, we mean extensions through modifications or additions to microcode on those computers (such as all machines in the Interdata family) which have a two-level (micro and user) programming structure.[6] This ability to modify or augment the instruction repertoire of the machine is a powerful way, albeit rarely exploited, to extend the machine at the lowest level; the new primitives thus created are fully comparable to the original instructions of the actual machine.

The *second level* of extensibility is at the software level through creation of *system software*. By system software, we mean such obvious extensions as operating systems, device drivers and interrupt handlers, and any program-creation software, especially language processors, which can drastically change the primitives available to the end user. *Planned* extensibility at this level implies such hardware features as:

(a) A user mode/supervisor mode distinction, providing a trap of some kind against the execution of reserved instructions in user mode, and providing some kind of "supervisor call" facility.
(b) An approach to I/O on the hardware level which

® Teletype, a registered trademark of the Teletype Corporation.

provides for substantial compatibility of the I/O mechanisms from device to device, particularly with respect to interrupts and status information.

(c) Some method for extended address mapping which provides for hardware protection and relocation, thus relieving the mini user of his worst problem—severe limitations on addressing.

The *third* and *highest level* of extensibility is on the user software level. Planned extensibility provides the end user with the capability of augmenting the apparent set of primitives he uses in writing his programs. Such long-used techniques as subroutines and macros both constitute expansions of primitives, particularly when used together, e.g., a macro used to generate a calling sequence for a subroutine which performs some complex function.

Many sophisticated programmers for minis, particularly those coming from a large-computer background, habitually define large numbers of macros to provide extended primitives comparable to those found in the actual instruction set on large machines. In this way, the experienced programmer of a minicomputer views the problem of programming a mini as being no different from that for a large computer, effectively countering the purveyors of the "minicomputer syndrome" by using the same techniques, approaches, and skills developed for large computers. The manufacturer can plan for extensibility on this level by providing the user with macro features in the assembler and by providing flexible hardware and software techniques for invoking subroutines.

Let us close this introduction by noting that each layer of extension, although providing a new set of primitives to the next higher level, may also reduce the original set of primitives available on that level. A set of device drivers for I/O handling, once defined, usually prevents the user from writing code to use a new I/O device in a substantially different way from that standardized by the drivers added as extensions; he also loses, in the process, any capability of doing I/O operations himself, since attempts to do so are trapped by the hardware.

Additionally, a Basic system—editor, compiler, and operating system combined—gives the Basic user a different and more powerful set of primitives than he has in assembly language, but deprives him of the ability to perform operations in assembly language, even though this might be far preferable for some aspects of a given problem. Thus, the extended machine may look to the user quite different from the original machine, with some primitives added and others deleted, at the extender's discretion.

The remainder of this paper is devoted to describing many kinds of extensions which we see occurring now and in the future. Since this paper is principally about software, we will concentrate on extensions to the second and third levels. However, we feel that current hardware developments will encourage software-like extensions at even the lowest level, and we will discuss these implications also. Hopefully, viewing the minicomputer as more closely related to the large-scale computer than the hard-wired controllers it was originally designed to replace will lead to alleviating the underlying symptoms of the minicomputer syndrome.

THE MINI AS A GENERAL-PURPOSE COMPUTER

We will first discuss the minicomputer as it may be viewed by the applications programmer in the future. To do this, we will first examine a significant difference in the evolution of larger computer software as opposed to minicomputer software.

Both scales of computers developed from the same base: machine-language programs loaded into memory via manually produced media (cards or paper tape) followed by console debugging with the user setting switches and reading lights on the computer. In the history of large computers, the first executive systems were batch monitors, which were quite widely developed by 1960. These batch monitors provided for submitting "jobs" on decks of cards to the computer center where they were eventually entered into the "job stream" and run. All programming and debugging tools developed for large computers during most of the 1960s were oriented to batch operation; and today virtually all operating systems for today's large computers are optimized for batch processing. Nearly all commercial programming, and most university programming, is today done using batch monitors on computers including the IBM 360 and 370, the Univac 1108, and the CDC 6600 and 7600. Recently, a trend has started toward providing some sort of support for interactive programming, debugging, and program execution. Except for several large computers explicitly designed for interactive operation in time-shared applications (specifically the XDS 940 and the DECsystem-10, both produced by companies previously known for minis), the machines were originally designed for batch operation and modified in both hardware and software to support interactive operation. The IBM 360/67 is a modification to the 360/65, the Honeywell (GE) 645 to the 635, the Spectra 70/46 to the 70/45. These machines seem to have been grudgingly provided by the manu-

facturers to meet what they saw as a small, uneconomical, but prestigious market; none of them provide nearly as good interactive operation as machines and software designed to be interactive from the beginning. In particular, a system like the DECsystem-10 can provide program interaction with the user on a character-by-character basis through the user's teletypewriter; no machine based on a 2741-type terminal can possibly do so, since the terminal and its supporting computer-based interfaces are designed to operate strictly on a half-duplex, line-at-a-time basis. But the trend would appear to be toward expanded interactive and conversational uses of large computers, particularly as management information systems make terminal access to data bases desirable in real time.

Minicomputer programming started in the same way as the large computers. Paper tape was the primary I/O medium, read in by an ASR-33 Teletype®, with the computer console used for all debugging. Since minicomputers cost so much less than large computers, there was much less pressure to get away from console debugging, particularly as it was recognized as being a very cost-effective tool for a competent programmer. When it became clear that it was possible to create software tools to facilitate debugging, it was natural to use the Teletype®, which was there anyway (recalling our definition of a minicomputer), as the I/O device around which to design the "monitor." The first interactive programs for minicomputers, using the "console teletypewriter" to provide interaction with the user, were symbolic editors and debugging tools (particularly DDT,[7] a debugging system produced at MIT for the PDP-1). Many other interactive systems for minis are described in an earlier paper by one of the authors.[8] Interactive systems of all kinds are common today on virtually all minicomputers. In addition to symbolic editors and debugging systems, there are full-scale Basic systems for one or several users on many minis; several minis have Disk Operating Systems designed for conversational control from the console teletypewriter, single-user and multi-user interpretive systems derived from JOSS® running on one or more terminals, etc.

Now that minicomputer configurations often include peripherals, such as mass storage on removable disk packs, card readers, line printers, magnetic tapes, formerly found only on large computers, we see a trend toward the use of batch operating systems on minicomputers. Such systems, typically based on RPG and/or Fortran as higher-level languages, are closely modeled after large-machine batch systems. *The resulting virtual machine thus looks almost, if not completely, like the virtual machines erected on larger-machine bases.* A user programming in Fortran or RPG, both relatively standardized languages, cannot tell, when he submits a deck and gets back his reports, whether it was run on a large computer or on a mini (except, perhaps, by the size of the bill!).

Thus, large computers and minis are becoming more and more alike—at least from the point of view of the applications programmer. The authors hope that the significant advantages of interaction between the user and the computer, so prominent in the development of minicomputer software, will not be disregarded by the mini manufacturers in their seemingly headlong rush to "me-too" compatibility with larger computers and their batch operating systems.

THE MINICOMPUTER EXTENSION SPECIALIST

Until now, the people who have played the largest role in extending minicomputers have been the system software designers and implementors and the hardware engineers. We feel that the systems software designers will have an increasing role as extension specialists in the future by becoming knowledgeable in hardware techniques.

Hardware/software extensibility

Already, the manufacturers of minis are providing richer instruction sets and more designed-in extensibility in their newer computers. As an example, the DEC PDP-11 provides a rich instruction set, a novel I/O structure, and virtually a complete elimination of the addressing problem which once plagued most minis. The Interdata Systems 70 and 80 provide substantial encouragement for the user to employ microprogramming to extend the base machine, i.e., to produce firmware. Almost all new minis provide modular packaging, in which a hardware function, to add a new primitive, can be easily wired on a single module and plugged into the machine.

We see the continued growth of microprogrammed processors, built around Read-Only Memories (ROM's), as being in the vanguard of user-extensible machines. The newest hardware facilities include Programmable ROM's (PROM's) which can be written by the user on his site; Erasable PROM's—which can be manually

® JOSS, a registered trademark of the RAND Corporation.

erased after being written; and Writable ROM's (slow write cycle compared to read, intended to be written relatively infrequently), sometimes called "Read-Mostly Memories" (or RMM's). All of these lead us to see a trend toward making this form of extensibility available by design, rather than by accident.

Thus, the extension specialist will be encouraged to work on all levels of extensibility. He will be able to add firmware in the form of lower level programming (microprogramming) or as pre-wired primitives which he can literally plug in as new functions (as on the GRI-909 family). Writable ROM's promise additional extensibility, even providing the systems programmer with the ability to optimize the instruction set for each application by loading a writable ROM from his system initialization code. Thus, he might choose an instruction set built around four-bit decimal numbers to run Cobol or RPG programs, one featuring binary floating point on multiple-precision numbers for Fortran or Algol, one with extensive byte manipulation for Snobol, and one with primitive expression evaluation for PL/I.

Systems programmers will become more competent at lower-level extension work, either by the firmware being brought closer to the user in the form of writable ROM's, or by the software designer being cross-trained in hardware techniques. For many minicomputer systems programmers, an oscilloscope is a familiar tool in debugging complicated programs and systems. Every sign indicates a growing encouragement for the systems programmer to create his own desired environment through extensibility. The DEC PDP-16 carries this to its logical extreme by letting the user build his own computer from a pre-determined set of modules using software tools to aid the design and implementation process.

Thus, we predict that the systems designer of the future will increasingly be at home in both cultures—hardware and software.

Programming automation systems

We also see a substantial growth and extension to what we call "programming automation systems"—techniques that provide the programmer with computer aids in the programming process.[9] All programming systems are designed to help the programmer through the critical loop of program development, i.e., program modification, language processing, debugging, and back to modification. Thus, systems such as Basic and JOSS® provide for complete control over the critical loop within the context of a single system with a single language used to control all three functions. On minicomputers, symbolic editors, language processors and debugging aids are provided to "automate" these three steps. For the systems programmer working on logically complex and large programs, however, the minicomputer does not really provide an optimum environment. Unless it provides bulk storage and a line printer, it is not well suited to editing and assembly. Unless it has a console much better designed than most mini consoles, it is also not particularly well suited to debugging (and the newest consoles are even less useful for debugging). Thus, there seems to be a trend toward the use of larger computers, particularly time-shared systems, for supporting programming automation systems designed to support smaller machines. A large-machine-based editor and assembler can do a lot to facilitate the creation of programs, particularly since the assembler does not have to be restricted in either the features or the number of symbols which can be accommodated. A good simulator, designed for debugging, can significantly improve the productivity of a programmer by providing him with the necessary tools to debug a program without the limitations of the smaller machine. The authors have invested quite some time over the last few years investigating this approach;[10] several other organizations have also done so, with some success. Several new computers, notably the GRI-909 and the IBM System 7, accentuate this trend; they are only supported by larger host computers; the actual machines are configured only to run programs, not to create them.

For years, systems programmers for small machines have felt, like Pooh Bear, that there must be a better way than assembly language to write programs for small machines. The so-called "implementation languages" have been in use for some time on larger computers; languages such as AED,[11] BLISS,[12] PL/I and ESPOL[13] have been used to create systems software, including compilers and operating systems, for large computers. We regard it as unlikely that implementation languages will soon be operational on minicomputers, due to their high initial cost and inefficiency of generated code. (Only if substantial computer time is invested in optimizing time and space will minis be able to support such implementation languages.) We do feel that the trend toward using larger computers to support minis will continue, and that it will soon be possible for systems programmers to use large-computer-based implementation language processors and debugging systems as accepted tools of the trade. Already a compiler for the BLISS language, an implementation language developed at Carnegie-Mellon originally for the DEC system-10, has been produced to generate code for the PDP-11 on the DEC system-10.

THE MINI AS A BASE FOR DEDICATED SYSTEMS

Over the past five years (which represents virtually the entire history of the minicomputer in terms of number installed), there has been a noticeable shift in the hardware/software tradeoffs in pricing and using minicomputers in dedicated system applications. Several people have long maintained that the price of the software for a dedicated system should, as a rule of thumb, equal the price of the hardware. Although apparently true four years ago, this cannot possibly be true today. Hardware costs of minicomputer mainframes and memories have been decreasing exponentially at an apparent rate of about 25 percent a year, while software costs have been slowly rising, at the rate of inflation. Thus, if the hardware/software cost ratio was around 1/1 four years ago, it is more like 1/2.5 today, and the gap is steadily widening.

All of the extensibility promised by recent hardware developments, including the modular construction of the new machines and the increasing use of ROM's, should have an accelerating affect on the applications of minis to dedicated systems, steadily reducing the cost of the hardware required for a given task.

We see implementation languages beginning to relieve the trend toward higher software costs and hope they will continue to do so in the future. But we see problems in the high cost of training existing programmers to use implementation languages, the lack of acceptance by lower-level programmers, and general questions arising as to whether such languages and compilers really appropriately solve more difficult problems (such as those that are space and/or time critical).

We predict that in the next few years the current problems regarding implementation languages will be vigorously attacked on several fronts, and we feel reasonably certain that they will be increasingly used in dedicated systems, particularly those which are neither space nor time critical. For critical systems, we feel that some time will be required before compilers for implementation languages, and indeed the languages themselves, are improved and tested to the point that they can be of real value.

Several people, including one of the authors, have used interpretive languages to simplify the construction of dedicated systems. In this technique, an existing interpreter, for a language such as ESI[14] or FOCAL,[15] is modified so as to minimize the core requirements for the interpreter by removing unwanted features and adding additional ones. The resulting extended machine is thus an ESI machine, or a FOCAL machine; the user program in the higher level language is stored in memory and executed completely interpretively. John C. Alderman, in a paper presently in manuscript form, has referred to such languages as "plastic languages", in the sense that the systems programmer, in designing the dedicated system, can modify the interpreter for the language, and thereby change the virtual machine, in much the same way as writable ROM's can be used. Indeed, the two approaches can easily be combined; one could wire the interpreter in an ROM and plug it into the computer, thus creating an unalterable virtual machine for FOCAL or ESI.

It should be noted that the use of the "plastic language" technique is limited to those systems which are not time critical, since the interpretive nature of program execution makes the resulting virtual machine relatively slow. One of the authors has been quite successful in applying this technique to a proprietary dedicated system, where its advantages are quite significant particularly regarding ease of modification of the higher-level application code.

SUMMARY AND SOME SPECULATIONS

In closing, we would like to summarize a few points made earlier, and to exercise our license to speculate a little on the near future of minicomputers.

First, the user of minicomputers for the solution of general-purpose applications in data processing, scientific programming, or engineering, will find minicomputers increasingly indistinguishable from larger computers. With luck, he will not find that interactive control, which now distinguishes most minis from larger systems, has been thrown out in the wash.

Second, the cost/performance ratio of minicomputer hardware will continue to improve at the same rate as it has over the last five years.

Third, the minicomputer user will continue to receive the benefits of cost/performance ratios through decreases in cost for the same, or slightly improved, performance while the large computer user will generally continue to receive more performance at the same cost.

Fourth, and as a consequence of the above, all applications of minicomputers will become increasingly more economical. Thus, many applications which are not performed on minis today, or many which are not done on computers at all, will utilize minis in the near future as prices continue to drop.

It might be argued that time sharing could just as easily be used in the future to solve problems which do not use computers at all now. It is undoubtedly the case that time sharing will continue to be used for those

problems which require its unique assets, i.e., essentially as much processor time, as much I/O, as much core and disk space as required, when you need it, purchased on an "as you use it" basis. But even at the most favorable rates available today, time sharing is *much* more expensive than using a mini. The lowest price we know of for time sharing is about $8 an hour for as much time as you can use, though this is on a relatively small system with quite heavy loading and a lot of contention.

On the other hand, even at today's prices, a mini can be bought for $5000. If this is written off over three years, and used 40 hours a week, an effective price of only about one dollar an hour can be approximated. A few years from now, this should drop to around 25 cents an hour.

We see the possibility of people providing a variety of "plug-in packages" for popular minis, quite likely software provided in the disguise of ROM hardware to provide the supplier with proprietary protection, product standardization, and integrity against unauthorized user changes. Some of these standard plug-in packages might be:

(a) The COGO virtual machine for the civil engineer.
(b) The JOSS® virtual machine for the engineer and statistician, replacing today's electronic desk calculators.
(c) The small business virtual machine, providing the retailer with a small machine capable of performing routine bookkeeping.
(d) The homemaker virtual machine, providing the busy housewife with a menu planner, household controller, alarm system, and checkbook balancer.

In conclusion, if the designers and product planners of minis think more clearly on what minis can do in both program creation and program execution, we may see an end to the minicomputer syndrome.

"He nodded and went out . . . and in a moment I heard Winnie-the-Pooh—bump, bump, bump—going up the stairs behind him."

Ending of *Winnie-the-Pooh*, A. A. Milne

REFERENCES

1 A W HOLT W J TURANSKI
Extended machine—Another approach to interpretive storage allocation
Under Army contract DA-36-039-sc-75047 Moore School of EE U of Pa 1960

2 C J SIPPLE
Computer handbook and dictionary
Bobbs-Merrill Indianapolis 1966

3 R W WATSON
Timesharing system design concepts
McGraw-Hill New York 1970

4 *PDP-7 CORAL manual*
Applied Data Research Inc Princeton N J 1967

5 *Broadcast programmer user's manual*
Applied Data Research Inc Princeton N J 1968

6 *Interdata microprogramming manual*
Interdata Ocean Park N J

7 A KOTOK
DEC debugging tape
Memo MIT-1 rev MIT December 1961

8 D J WAKS
Interactive computer languages
Instruments and Control Systems November 1970

9 D J WAKS
The MIMIC programming automation system
Applied Data Research Inc Internal memorandum July 1971

10 *MIMIC user's guide*
Applied Data Research Inc Ref #01-70411M May 1971

11 *AED manual*
Softech Cambridge Massachusetts

12 *BLISS-10 manual*
Carnegie-Mellon Univ Pittsburgh Pa

13 *ESPOL manual*
Burroughs Corp Detroit Mich

14 D J WAKS
Conversational computing on a small machine
Datamation April 1967

15 *FOCAL user's manual for PDP-8*
Digital Equipment Corp