INTRODUCTION

The UCSD Pascal System is a complete program development and execution environment for small computers. Its facilities include text editors and file management utilities, as well as compilers (for Pascal, in particular), assemblers, and a linkage editor. The system is highly portable; versions have been implemented on almost twenty minicomputers and microprocessors. (A concise description of system facilities is provided in an appendix.)

The system was developed under the direction of Professor Kenneth Bowles at the University of California, San Diego (UCSD), starting in late 1974. The author, working first at the University as a graduate student, and more recently with SofTech Microsystems, played a principal technical role throughout the evolution of the software.

The original need was for inexpensive interactive access to a high level language for a large enrollment introductory course in problem solving and computer science. We decided early that we would use small, stand-alone computers as the hardware foundation for our solution rather than larger, time-shared computers. We then chose Pascal as the language to be used by students in the introductory course, and also as the implementation language for system software we would need to build for these small machines. The design goals for Pascal specifically included these two kinds of applications. We used the P-compiler as the starting point for our Pascal implementation.

We needed a stand-alone Pascal program development and execution environment suitable for computer-naive students, but also capable of being used for maintenance of the system itself. We had two primary design concerns:

1) a user interface oriented specifically to the novice, but also acceptable to experts;
2) a strategy for fitting these facilities in small, stand-alone machines. By our definition, a “small machine” had less than 64k bytes of memory, dual standard floppy disks, and a CRT terminal. We were particularly concerned about the Pascal compiler, since we knew of no implementation of Pascal within those constraints.

These two concerns have been themes of the evolution of UCSD Pascal ever since. Experts and novices have both continued to use the system, and adaptations of the system to even smaller host configurations (e.g. the Radio Shack TRS-80®) have been done.

In the user interface area, our current philosophy is still very similar to that originally developed. Given a single user host computer and a CRT terminal (the preferred environment), our approach is to keep the user continuously informed about the state of the system and the options available in that state. A “prompt-line” is maintained on the terminal screen listing these options. The user can select an option by typing a single-character command. In text editing, the high bandwidth connection to the user is exploited to provide a continuously updated “window” into the text file being perused or modified. A naive user in this environment is led tutorially through an interaction with the system. Experienced users can ignore the continuous status information unless it is needed.

The original small system implementation strategy has also largely survived. Its major component is the use of a p-machine as the foundation of the system. The UCSD p-machine is a simple idealized stack-oriented computer which can be emulated by an interpreter executing in the machine language of a conventional host computer.

The important requirement is that the instruction set of the p-machine be designed so that p-code representations of Pascal programs are very compact and easy for the compiler to produce. Compactness and ease of generation for p-code are both important, in order to minimize the size of the Pascal compiler, which is the biggest single software component of the system.

We were inspired by the code-compaction approach used at Burroughs, but had to adapt it to conventional hardware without facilities for microprogramming or bit-level addressing. Through several iterations on the p-machine architecture, static and dynamic statistics of opcode and operand frequencies were used to identify the instruction sequences occupying the most space and redesign them to be more compact. Tannenbaum has independently pursued the same sort of optimizations, but without our concern for software portability.

More generally, our concern for small host computers is pervasively reflected in our choice of functional facilities...
included in UCSD Pascal and the implementation approaches used to provide them.

As we accumulated experience with hardware and software for small systems, it became clear that while the costs of raw hardware would continue inexorably down, the costs of software were an entirely different matter. Thus, a third concern became important in the evolution of UCSD Pascal: conservation of our software investment. The bulk of this paper is devoted to explaining the software conservation strategy that we have evolved.

One final introductory matter needs to be addressed: the unexpectedly large interest in UCSD Pascal from outside the University and how that interest was dealt with. A version of our software running on PDP-11's was first distributed to a few off-campus users in the summer of 1977. Outside interest began to increase when a version for 8080's and Z80's became operational early in 1978. Shortly thereafter, a description of the system in Byte drew over a thousand inquiries. As interest in the software mushroomed through 1978, it became clear that the demand for UCSD Pascal could not be met within the available resources of the University project. For this and other reasons, investigation began at the University into ways in which support of the growing UCSD Pascal user community could be moved off-campus. This effort culminated in June, 1979, with the designation of SofTech Microsystems as the focal point for licensing, support, maintenance and continued evolution of the UCSD Pascal language and system. Advanced development work has continued at the University in the Institute for Information Systems. Some of the results of that work are described later in the paper.

CONSERVATION OF SOFTWARE INVESTMENT: AN OVERVIEW

The first component of our effort to conserve software investment is the use of Pascal as the principal system and application language. In 1974, when the choice was made, Pascal was one of the more popular academic languages, and provided the best combination of power and ease of implementation. We certainly did not foresee the current avalanche of industrial interest in the language.

The second component of our software strategy is a heavy emphasis on software portability. We feel that independence of software from differences in the underlying hardware is crucial to small machine applications in order to have:

1) the freedom to change hardware to take advantage of rapid technology developments, thus reducing hardware cost or increasing its performance.

Consider the experience at UCSD with equipment for teaching Ken Bowles' introductory computer science course. The first small machines, acquired in the fall of 1975, were nine PDP-11/10's (worth about $17,000 apiece). Since then, two equipment transitions have occurred at two year intervals: first to 25 Terak 8510a computers (worth just under $8000 apiece), and then to 45 Apple II computers (at less than $3000 each). And all of these configurations can run virtually the same software!

2) the chance to reduce the effective cost of software by widespread sharing within an application community.

At this writing there are more than 15,000 computers (with many different host CPU's) running UCSD Pascal. This number is large enough to justify significant investments in application software. High costs to the individual end-user are not required to recoup those investments.

These portability motivations are, of course, not new to the small machine environment; they have just become much more urgent as the cost of software continues to rocket while that of hardware plummets.

There are several different approaches to software portability. One can emphasize the program, picking a particular large application, like a database management package, and working to reduce effort to move it to new host operating system or processor environments. A disadvantage of this approach is that the effort must be expended anew for each additional application considered.

Another possibility is to emphasize a language and its implementation. Here the theory is that programs written in that language will port easily (by recompilation) to a new environment after the language itself is moved. Unfortunately, any sizable application program calls on I/O and other operating system resources in ways that may conflict with services available in a new environment. Therefore, changes are likely to be needed in the application programs to be moved. Figure 1 may clarify this approach.

UCSD Pascal provides a portable software environment. When the system is moved to a new host, all the conventions about file titles, disk organization, and other operating system matters are replicated. Therefore applications in UCSD Pascal can usually be moved to a new host without any modification to the source version of a program. This is shown in Figure 2. As we will see below, it is even possible to move the object version of a program to a new host.

Software environment portability has been independently pursued with the Thoth and Unix operating systems. Neither of these efforts has had our concern for the special problems of small hosts.

In two sections below, specific areas of UCSD Pascal portability are examined. In the first, independence from the host processor (ignoring system peripherals) is considered. Secondly, our approach to independence from host peripheral devices is discussed.

While our concern with portability is long-standing, recognition of the third component of our conservation strategy (organization for support and maintenance) is more recent. As UCSD Pascal entered widespread production use in 1979, it became clear that maintenance and support activities needed careful attention if a large user community using many varieties of host hardware was to be properly served.
at a reasonable cost. It also became clear that the academic environment was not well-suited to this task. One reason why SofTech was chosen to take over support of UCSD Pascal was their experience in developing and using software engineering tools of the type needed to support maintenance of UCSD Pascal.  

**INDEPENDENCE FROM THE HOST PROCESSOR**

*P-machine contributions to software portability*

The key to host processor independence is the designation of the p-machine as the foundation of the UCSD Pascal System. One result is that the entire Pascal system, including editors, compilers, operating system, etc., can be moved to a new host computer by reimplementing the p-machine and associated low-level routines in the native language of the new host. The relative simplicity of this implementation task has been widely exploited. Implementations of some variant of the UCSD p-machine have been done for the microprocessors and minicomputers listed below. (Those that have actually become products are marked with an asterisk.)

*Data General Nova
*Digital Equipment PDP-11® & LSI-11®
*General Automation GA-16
Hewlett Packard System 45
*Intel 8080 & 8086
Lockheed Sue
*Mos Technology 6502
*Motorola 6800 and 6809; 6800 is being actively pursued

The effort required for one of these implementations has ranged from 6 person weeks to 9 person months, depending on experience of the implementor and suitability of the host for p-machine implementation. We know of no other body of software as large as the UCSD Pascal System that has been moved to as many different host computers.

UCSD p-code is portable at the binary code file level (see more discussion of this under Data Representation Issues). Other pseudo-machine oriented efforts (Janus, for example) have generally standardized, instead, at the level of symbolic pseudo-machine assembly language. The choice of a binary interface for UCSD Pascal makes it much more practical for p-code to serve as a sort of "lingua franca," for communicating object programs among a wide user community.

Carl Helmers has proposed a distribution approach for small machine application packages in which the last few pages of the user document would contain a printed bar-code encoding of the object program. As he recognized, UCSD p-code fits very nicely into this approach.

P-code is a lingua franca in another sense: even though the p-machine is optimized for Pascal programs, translators can and have been written from FORTRAN and BASIC to p-code. With some strategic additions to the p-machine to support new data types, even a COBOL to p-code translator is feasible.
Concentration on small systems

We have chosen to limit the class of suitable host computers so as to enhance portability within that class. We assume that memory can be viewed as 8-bit bytes and 16-bit words and that the 7-bit ASCII character set is used. Distributed versions of the p-machine are also limited to a 16-bit address space. (A later section discusses removal of this limitation.) Our success in transcending the details of the host environment has been substantially increased by specializing in this limited class. Fortunately, most small computer systems are included.

Performance issues

What price has been paid for these portability benefits? One part of the cost is in the reduced execution performance of interpretively executed p-code compared to other implementation approaches. For many small computer applications (text editing or data capture, for instance) interpretive execution on a dedicated microprocessor is more than adequate. In other applications (e.g., compilation) the benefits due to the small size of p-code outweigh the drawbacks of raw execution speed.

It is also possible (with some reduction in portability) to code time-critical routines directly in assembly language and call them from a high level host program. Most real-time programs can meet performance requirements with only a small portion (less than 10 percent) written in assembly language.

Another performance possibility is to provide more direct hardware support for the p-machine. In the Western Digital MicroEngine, the MOS chip set used in the implementation of Digital Equipment's LSI-11 has been microcoded to implement the p-machine directly. P-code is thus the native language of the MicroEngine. Performance improvement factors of five or more have been measured for the MicroEngine over interpretive execution on the LSI-11. At least another factor of two in performance has been achieved with a micro-coded p-machine based on high speed bit-slice technology. An overall improvement factor of twenty compared to LSI-11 interpretation is probably achievable with standard low-cost components. (All of these comparisons apply to simple integer arithmetic and array operations, as in an integer sort routine.)

Data representation issues

Some concessions to efficiency over portability have been made. The biggest has to do with representations of p-machine data types. Although a standard on floating point number representation and algorithms is being developed by the IEEE,11 there are still many different formats supported by various vendors. We have standardized on a size of 32 bits for floating point numbers, but do not require a particular representation. Therefore, advantage can be taken of existing floating point software and hardware support. Where new floating point implementations have been done (on the 6502, 6800, 9900, and W.D. MicroEngine) the IEEE format is used. A machine-dependent "power of ten" table allows all high level software (even conversions between ASCII and internal floating point) to be isolated from knowledge of the internal representation. Long integer representations can also vary. Binary integer, packed BCD, and radix 10 are among the feasible representations. A dramatic performance improvement can be achieved on some hosts by choosing a suitable representation. Once again, higher level software in the Pascal system need only care about the length (in words) of a long integer.

Even ordinary 16-bit integers do not have a standard representation (at least in the ordering of their two bytes). In some host architectures (e.g. PDP-11, 8080) byte zero of a word is the least significant byte; in others (e.g. 6800, Z8000, IBM 370), byte zero contains the most significant bits of the 16-bit integer. Thus, the interpretation of two adjacent bytes as a 16-bit word is machine-dependent. Here, too, an attempt to force a single representation on all architectures would be prohibitively expensive.

What is the impact on portability of these representation decisions?

First, portability of source programs is not affected, unless a program specifically chooses to deal with the representation, by bypassing the type philosophy of Pascal. We need, but do not yet have, an analog of the LINT program under Unix,7 which could comb source programs for potential trouble spots of this type.

Second, data files containing any of these data types are not directly readable by implementations with different representation choices. It is possible to design file record structures so that an application program can automatically compensate for representation differences.

Third, code files are sensitive to the byte ordering (we call it "byte sex") of the host processor. The dependence occurs where adjacent bytes in the code stream represent words (mostly in superstructure tables). The Pascal compiler can generate code files of either type on any host, and a utility program is provided to convert object files from one type to another.

Finally, code files containing floating point constants are not directly movable between hosts with different floating point representations. It is usually possible, by doing some computation at run-time, to avoid this problem.

INDEPENDENCE FROM THE HOST PERIPHERALS

UCSD Pascal I/O hierarchy

Our approach to achieving peripheral independence is to provide a hierarchy of I/O environments. The levels of the hierarchy are chosen to further two objectives:

1) isolation of application programs and most system components from details of the host computer, and
2) reduction of effort involved in adapting to new host configurations.
The levels we identify are pictured in Figure 3 and listed and described below:

1) Screen I/O. This level presents a uniform image of a screen terminal. Capabilities include moving the terminal cursor; clearing all or part of individual lines or the entire screen; and accepting cursor control or other special commands from the terminal keyboard.

2) File I/O. At this level, devices are designated by logical volume names. Volumes can be serial (e.g. console) or random access (e.g. disk). Random access volumes can have directories of named files. Serial volumes are (possibly bi-directional) byte streams. Textual I/O (with full conversion between internal representations and ASCII) is provided from and to volumes or files. Record-oriented I/O (with automatic blocking and deblocking) is provided with random access volumes or files.

3) Unit I/O. At this level, devices have numbers that indicate their type (e.g. 1 for console terminal, 6 for printer, etc.). A serial device is still a byte stream, with knowledge of a few special output characters (blank compression codes, carriage returns). A random access device is considered an array of directly addressable 512-byte blocks. No knowledge of files, textual or record-oriented I/O is available at this level.

4) Basic I/O. Capabilities of random access devices at this level are similar to those available at the Unit I/O level. Serial devices are much simpler. Serial transfer occurs one character at a time and no special output characters are processed. Special input characters from the console terminal can cause console output to be stopped, restarted or discarded.

5) Simplified I/O. Here the random access interface is much more primitive. A device is viewed as a sequence of tracks, each containing an array of physical sectors. Transfers occur, one sector at a time, between a main memory buffer and a (track, sector) coordinate. Serial device capabilities are similar to those in Basic I/O, except that no special input characters are recognized.

Table I summarizes the degree of independence from host configuration provided at each of these levels. The Basic and Simplified I/O components are generally implemented in assembly language and clearly dependent on peripheral details. The pseudo-machine interpreter (which generally includes Unit I/O) is dependent on the host processor, but defers peripheral details to lower levels. File and Screen I/O are implemented in Pascal and are therefore independent of both peripheral and processor variations. The application program does not have to depend on anything but the virtual environment provided by the Pascal system.

<table>
<thead>
<tr>
<th>Software level</th>
<th>Is independent of:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Console</td>
</tr>
<tr>
<td>Application program and host system components</td>
<td>Yes</td>
</tr>
<tr>
<td>Screen I/O</td>
<td>No</td>
</tr>
<tr>
<td>File I/O</td>
<td>Yes</td>
</tr>
<tr>
<td>Interpreter and Unit I/O</td>
<td>Yes</td>
</tr>
<tr>
<td>Basic and Simplified I/O</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Adaptation to host configurations

Four kinds of user adaptation are intended. The easiest adaptation is also by far the most frequently needed: catering to console terminal peculiarities. Most of the effort involved goes to changing the entries in terminal description tables maintained by the system (a program, SETUP, is provided to do this). Some programming (in Pascal) of interface procedures may be required. Average effort involved for the user is an hour or two. A test program is provided to determine if the effort was successful.

Next in order of increasing difficulty and decreasing frequency of need is adaptation of the Simplified Basic I/O Subsystem (SBIOS). Given that a user is knowledgeable about his peripheral interface, and has access to existing low-level driver software, a few days of work should be sufficient to bridge the gap between those drivers and the SBIOS interface, producing a usable Pascal System. Again, a test program is provided. The approach we have taken here is based on that developed by Digital Research for the CP/M® operating system.

The Basic I/O Subsystem (BIOS) is comparable to its simplified cousin, except that is has more responsibility, and therefore more possibility for optimization. For instance, it is possible in the BIOS to take full advantage of a direct memory access interface to disk. The BIOS definition emphasizes performance and flexibility, while the emphasis with SBIOS is on ease of adaptation. BIOS implementation generally takes a week or two of effort, assuming detailed knowledge of the peripheral complement and familiarity with our I/O structures.

Finally, the most elaborate adaptation is to a new host processor (which is included in this peripheral independence section for completeness). As mentioned above, effort involved here is more than a month, but less than a year; six person months is probably a good average.

Table II gives the various levels of adaptation. For each, a reasonably realistic level of effort is given, as well as an approximate "probability of need" indicating how frequently adaptation at that level will be desired. Fortunately for the viability of our approach, these probabilities have some basis in reality.

ENHANCEMENTS

Native code  generation

Active work is under way at UCSD to provide the ability to translate selected procedures of a p-code program to native code for a conventional host computer. This possibility will alleviate many of the performance drawbacks of our p-code orientation without sacrificing portability. Programs can be written and maintained entirely in Pascal, and the p-code object version is still transferable among different kinds of host computers. If active use of a program reveals performance bottlenecks, the time-critical procedures can be translated to a native code.

<table>
<thead>
<tr>
<th>Level of adaptation</th>
<th>Approximate &quot;Probability of need&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application program</td>
<td>very little, if any</td>
</tr>
<tr>
<td>Screen I/O</td>
<td>hours</td>
</tr>
<tr>
<td>Simplified I/O</td>
<td>days</td>
</tr>
<tr>
<td>Basic I/O</td>
<td>weeks</td>
</tr>
<tr>
<td>Interpreter</td>
<td>months</td>
</tr>
</tbody>
</table>

Code generation is implemented as an optional step in the compilation process. It takes as input a complete p-code program and produces, as output, a mixture of unmodified p-code and translated native code procedures. This process is diagrammed in Figure 4. Internally, the code generator represents a p-code procedure as a forest of expression trees. Several traversals of these trees occur during the translation process.

Note that the p-code input to the code generator can come from a Fortran or Basic compiler, as well as from the Pascal compiler. Once again, software investment is conserved, since only one code generator for a particular target machine serves several languages.

Implementations are in progress of code generators for the PDP-11, TI9900, Intel 8080, Mos Technology 6502, and General Automation GA-16. At this writing, the first two versions are farthest along. For both processors, translated native code is about 50 percent larger than the corresponding p-code. Improvements in execution performance compared to interpretive execution on the same host have been around a factor of 10 for the PDP-11 and a factor of 15 for the TI9900.

New definition of "small" systems

As new microprocessor architectures (e.g., 8086, Z8000, 68000) and peripherals (e.g. low-cost Winchester disks) become widely available over the next several years, the definition of "small" low-cost computer systems will have to broaden considerably to include those with megabytes of main memory and tens of megabytes of mass storage. We are investigating ways in which the UCSD Pascal system could allow these facilities to be exploited. The situation is complicated by our need to continue supporting both 280 class and Z8000 class users for the foreseeable future. Therefore, it must be possible for user programs, and particularly system components, to run in both environments. It must also be possible, of course, to produce a user program that requires so many resources that it can only be run in an expanded environment.

As an example of the extension approach we are pursuing, consider the problem of dealing with address spaces bigger than 16 bits. In the near term, it is quite easy to apply the 16-bit limitation only to data space: object code need not be accessible within the 16-bit area. For the longer term, we are considering a scheme in which details of physical ad-
ACKNOWLEDGMENTS

The UCSD Pascal system is the work of a large group, too numerous to list here, of graduate and undergraduate students at UCSD. The role of Kenneth Bowles in inspiring, directing, and energizing this work has certainly been crucial to its success. Helpful comments on this paper from John Brackett, Winsor Brown, Al Irvine, and Richard Kaufmann are gratefully acknowledged, as is Keith Shillington’s preparation of the illustrations.

REFERENCES


APPENDIX: Facilities of the System

Program execution environment

The foundation of the System is the UCSD p-machine. It is a simple idealized stack computer which can be implemented either by direct hardware support (as in the Western Digital MicroEngine) or by an interpreter executing in the machine language of a conventional host computer. On a conventional host, a single object program can include both p-code (to be interpretively executed), and native code (for direct execution by the host).

Peripherals are accessed by logical "volume" names. Serial volumes (e.g. console terminal) are considered byte streams. Random access volumes (e.g. floppy disk) can have directories of named files. Various kinds of logical transfers involving volumes and files are supported.

Program execution and file manipulation commands

The user can execute a named object program, or use short cut commands to invoke important system programs. The user can also designate individual files or groups of files for removal, renaming or transfer among on-line devices. Other commands support various housekeeping needs: listing directories, compacting the files on a disk, and testing disks for invalid areas. Finally, the user can designate a "work file." Subsequent editing, compilation and execution commands apply to this work file by default.

Text preparation and modification facilities

Two styles of text editing are supported: one requires a video display terminal, and the other does not.
When the system console is a CRT, the “screen-oriented”
editor can usually be used. This editor maintains a cursor
into the text file being edited and a “window” into that area
of the file on the terminal screen. Modifications to the text
are made by the intuitive and mechanical process of moving
the cursor to the site where change is desired and indicating
the change. Commands are provided for moving the cursor,
finding and replacing textual patterns, making insertions and
deletions, and copying text into the cursor position from
elsewhere. Special facilities exist for processing documents.
User-specified left and right margins can be automatically
enforced by the editor and new margin requirements can
easily be applied to existing text.

The second available style of editing does not require a
screen terminal. Once again, a cursor is maintained, where
most of the action occurs. But the user is responsible for
maintaining a mental image of the cursor context. Com­
mands are available for insertion, deletion, and copying of
text, as well as for moving the cursor. A simple macro facility
is provided.

Programming languages

The principal programming language supported is UCSD
Pascal. Except for the provision of procedures as parameters,
UCSD Pascal is largely consistent with the base Pascal
language, as defined in Jensen and Wirth’s User Manual and
Report.12 UCSD Pascal is also quite consistent with the
emerging international standard for Pascal.13 We are com­
mitted to eventual complete compliance with an adopted
standard.

UCSD Pascal includes various extensions beyond the base
language. We summarize, here, the most important:

1) Dynamic character strings. A predeclared type “string”
is supported. A string variable contains a sequence of
characters. A maximum length for the sequence is spec­
ified in the declaration. Concatenation of strings; inser­
tion, deletion and extraction of substrings; and string
pattern matching are provided by predeclared service
routines.

2) Encapsulation and separate compilation. A new com­
posite declaration, the “unit,” is provided. A unit is
a group of procedures, functions, and data structures,
usually related to a common task area. A program or
another unit (a “client module”) can access these facili­
ties by naming the unit in a simple “uses” declara­
tion. A unit consists of two parts, the interface part,
which can declare constants, types, variables, proce­
dures and functions that are public (made available to
any client module), and the implementation part, in
which private declarations can be made. These private
declarations are available only within the unit, and not
to client modules. Units can be compiled separately
from their client modules.

3) Extended precision integers. A “long integer” data
-type is provided. Integers up to 36 decimal digits in size
can be represented and participate in the standard in­
teger operations: addition, subtraction, multiplication,
and division. Conversions among long integer, string
and standard integer forms are provided.

4) Concurrent processes. Another type of routine in
UCSD Pascal is the “process.” Processes are declared
with the global procedures of a program and have the
same lexical access to global variables and procedures.
A process is different from a procedure in that when
invoked, it proceeds in parallel with its invoker. Sem­
aphore variables, plus wait and signal primitives, are
provided to allow these parallel processes to synchro­
nize and communicate reliably. With the “attach” pro­
cedure, a semaphore can be associated with an external
interrupt. This association causes the semaphore to be
signaled if the interrupt is activated. Thus Pascal pro­
cesses can respond to external events.

5) Miscellaneous extensions. Other additions to UCSD
Pascal provide random access to Pascal file compo­
nents and a constrained interprocedural go to mecha­
nism. Segment routine declarations allow designation
of overlays and external procedure declarations allow
an assembly language routine to be called from a Pascal
host as if it were a Pascal procedure.

A Basic compiler exists for the UCSD Pascal system, but
is not currently being supported.

Assembly language is available for most processors on
which the system is supported. The assemblers can be used
for stand-alone programs (such as interpreters) or for pro­
cedures which will be bound into high level language host
programs. The approach is to provide (as far as possible) the
syntax for machine instructions defined by the original pro­
cessor manufacturer (e.g. Zilog for the Z80). A common syn­
tax has been defined for assembler directives and assembly­
time expressions. Naturally, all of the assemblers can run
on any host processor variant of the System, so a single type
of host can be used to support assembly language programs
for multiple machines.

Directives supported include the usual facilities for macro
definition, conditional assembly, storage allocation and list­
ing control. Additional directives allow communication with
external labels in other assembly language routines. Finally,
special provision is made for communication between an
assembly language routine and a Pascal host program. The
low-level routine can request access to host program global
variables and constants. It can also allocate its own global
storage space.

TRADEMARKS

UCSD Pascal is a trademark of the Regents of the Uni­
versity of California. PDP-11 and LS1-11 are trademarks of
Digital Equipment Corporation. Unix is a trademark of Bell
Laboratories. MicroEngine is a trademark of Western Digital
Corporation. CP/M is a trademark of Digital Research, Inc.
Radio Shack and TRS-80 are trademarks of Tandy Corpo­
ration.
Software Quality

The FAA's computerized Enroute System for controlling in-flight commercial aircraft crashes during a peak holiday period due to overloading. The DoD Early Warning System, a computerized air defense system, mistakes the rising moon for a barrage of incoming enemy missiles and shock waves travel all the way to the White House. A single erroneous statement in a small computer on-board a French weather satellite causes 71 of 142 weather balloons to self-destruct. These experiences would not have happened if there had been better software quality.

Assuring software quality has been and still is a thorn in the side of most software customers and project managers. This phenomena crosses all customer boundaries: commercial, industrial, military, other government; and crosses different application types: operating systems, information systems, process control, command and control, communication, business systems, etc. The "Software Quality" area contains four sessions that will enlighten both purchasers and developers of software with discussions and papers which will reveal not only current quality-related problems, but also suggested solutions.

Dr. Edward Miller will chair a panel session on Software Quality Testing. Panelists will discuss the need for establishing and following quality standards, and programming and testing techniques to improve the testing process.

Dr. Ned Chapin will chair a three-paper session on Software Quality Metrics. "Measuring program complexity in a COBOL environment" by Zolnowski and Simmons presents a composite measure of program complexity that provides an objective quantitative evaluation for any program or programming effort. Another paper, "The complexity of an individual program" by John McTap critiques the Zolnowski and Simmons model and proposes a model extension. The third paper, "An information theory based complexity measure" by Eli Berlinger proposes a measure of programming difficulty based on the probability with which various tokens of a program are used.

Dr. Leonard Gardner will lead a panel discussion of accomplishments, problems, and proposed solutions of present and future software standards. These will encompass machine, assembly, and high level languages, and software related to buses and their interfaces. Panelists have been selected who represent a very broad base of standardizing activities of various technical societies and workshops.

Mr. Kurt Fischer will lead a panel discussion of current software quality assurance problems and techniques. Topics of discussion will include the purpose of software QA, the techniques that are currently used, the benefits that are received from QA programs, and future directions that software QA should take. The selected panelists have all managed QA programs on major projects and will be glad to share their experience with the audience.