UNIX as an environment for non-UNIX software development
A case history

Jean Renard Ward, Pencept, Inc.
39 Green Street Waltham, MA 02154

ABSTRACT:

Many of the back issues of SEN contain articles about software development environments and software tools. UNIX has historically been cited as an example of a good software development environment. For many developers still struggling with the offspring of MS-DOS, RSX-11M, OS-370 and the like, UNIX still represents a dreamed-of state-of-the-art. Many of the more enhanced environments actually sit on top of UNIX or a UNIX-like system, especially for graphics-based environments like those offered by Sun Microsystems, Apollo, and Masscomp.

Pencept's business is real-time character recognition for hand-"scribbled" text. Our products run in a completely non-UNIX environment, but all of our development is done on UNIX. This presented a unique opportunity to find out how good "standard" UNIX is for developing non-UNIX products.

Based on our experience, we have come to the conclusion that UNIX is a good software development environment for the kinds of applications that are traditionally run on UNIX, if UNIX is the target environment. UNIX (and most of its derivatives) do NOT address many of the more general needs of non-UNIX software engineering projects.

UNIX is a powerful system, partly because it comes with a large assortment of software development tools. Some of the deficiencies and problems we had to resolve were:

+ Certain UNIX and vendor software had to be modified, because it did not have all the functions we absolutely needed.
+ Several UNIX utilities did poorly for this big a project.
+ Some UNIX features were poorly designed for non-UNIX development.
+ Some tools for our particular project are not part of UNIX.

Our particular vendor is Masscomp, which competes in the high-performance graphics workstation market with companies such as Apollo, Sun Microsystems, and Digital Equipment Corporation. We believe that our experience is generally applicable to UNIX systems, regardless of vendor, and across a variety of UNIX versions (System III, System V, Berkeley 4.2, etc.)

1. What we were developing: the PENPAD (R)

Pencept has developed and now markets an interactive character-recognition device, the Penpad (R), for reading hand-printed characters. It performs the functions of:

1. data input (a keyboard).
2. graphical input (digitizer or "mouse").
3. function selection/control (function keys or menus).
4. pointing (touch-screen, or a mouse).

The design of the product is the largest micro-processor based commercial product running multi-tasking code entirely out of PROM, with all code written in a high-level programming language, that any of us have encountered. A total of four bugs in the product code have been reported in the first 18 months from the field.

1.1 DCR (R) - Dynamic character recognition

The dynamic character recognition technology (R) is quite complex, and has been under constant development since January of 1977. In itself, this technology constitutes a radical breakthrough in its own field of character recognition, since no competing technology exists which can read untrained, unrestricted hand-printed characters with even a modest level of accuracy, much less in real-time, interactively.

The major portion of the recognition algorithms are written in an applicative language of our own design, which we execute interpretively in the product. The rational for this is strictly code size: we achieve a 10:1 compression of code size for that portion of our algorithms over programming in "C" (or any other compiled language) for the 68000 instruction set.

Figure 1 shows some of the hand-printing the PENPAD reads: compare this with Figure 2, the ANSI standard for machine-readable hand-print. The major differences between our DCR (R) technology and static character recognition, such as OCR, are:

1. We read "unconstrained" hand-print: if it's not too sloppy for a human to read it out of context, we need to recognize it, too.

(R) UNIX is a Registered Trademark of A.T.T.
(R) ADA is a Registered Trademark of the Department of Defence.
(R) DCR, PENPAD, and Dynamic Character Recognition are Registered Trademarks of Pencept, Incorporated.

2. Our device follows the motion of the pen while writing, rather than scanning a static image.
3. The results are delivered interactively in real-time.

1.2 Hardware - M68000-based design

The electronics use a 10MHz M68000 micro-processor with 196Kbytes of PROM and 64Kbytes of RAM. The RAM memory size is partly due to the need for a real-time heap management package to hold variable-sized descriptors for the menuing and block-mode functions.

2. Why would anyone need a large UNIX system for micro-processor development?

Unlike most applications using dedicated micro-processors, our code is very large. Over two-thirds of the code in the product is strictly for recognizing hand-printed characters. We also make heavy use of multi-taking to control several asynchronous activities: recognition, buffering and image-processing on the digitizer input, full-duplex host communications on two ports at 19.2Kbaud, and graphics operations. This presented us with several more orders of magnitude of complexity in debugging our code than with, say, a graphics terminal emulator using a 8- or 16-bit microprocessor running one process.

Some people say the cost of computing "iron" is dropping. We think it's more accurate to say that for a given application cost, the hardware is increasing in power. However, there is a lag in the development tools for micro-processor projects: they were all developed when an embedded micro-processor typically had less than 8Kbytes of memory, and everything was written by hand in assembly language. To develop the software for this next generation of embedded applications, the engineers on the project now need all the development tools ever found on large system projects, not just the low-level tools used on small hardware projects.

Given the complexity of the character recognition code, and the difficulties debugging anything of that size on most in-circuit-emulators, we wrote all our code to run:

1. as co-operating tasks running under UNIX, with modules added to make it a suitable test and development system for character recognition development. This is where we do all our recognition debugging, and most of our system development.

2. identical co-operating tasks in a PROMmed version of the code, compiled from the same sources.

The inter-task communications use UNIX's "named pipes". All the data control information is passed with non-blocking writes and blocking reads. This approach solves all our task-synchronization problems.

For development, we add a large body of code (over 10,000 lines) conditioned on "#ifdef" statements. The bulk of it is conditionally-compiled "assertion checks" that check for bad values in internal variables (similar to array-bounds checks that are generated by other compilers)(*). The second part is a complete set of debugging and development tools so we can develop our recognition algorithms. The benefits are:

1. Since all the code is the same, from the same sources and the same compiler, if it runs on our UNIX system we are extremely confident it will run in the product.
2. Likewise, if there is a bug on the product, we can always reproduce it in the development system on UNIX.
3. We have all the tools for high-level language debugging, not just an absolute-assembler debugger.
4. Since we use named pipes for inter-task communication, we can get copies of all task messages into files for examination.

2.1 What's important in picking a UNIX (R) for a development system?

UNIX and "C" are touted as guaranteeing "portable" software. This is true, but only relatively so. Moving software from one vendor's UNIX to another will probably be much easier than moving from Data General's AOS to DEC's VAX/VMS, but not all UNIX systems are the same.

The reasons for picking a 68000-based UNIX (R) system, and for picking one vendor over all other vendors were:

1. We needed a large-address machine for our development: the development version of the recognition code is over 700 Kbytes, and must run in real time. Real-time performance meant we could not run with disk overlays (such as on a PDP-11), and the fact that "C" could not deal with segmentation of our large (over

(*) (We have found that automatically-generated array-bounds and pointer checks from a debugging "C" compiler were too simple-minded to catch many of our errors, and that the "assert" option in our vendor's "C" compiler could not do a "graceful" recovery after an error. Our run-time checks are all written specifically for the code segment they are in. There is a limit to how much an automatic tool can substitute for human judgement.)

64Kbytes) recognition tables meant we could not run easily on a segmented machine (such as an 8086).

2. We needed to support a number of co-operating software engineers who worked together on the code: most microprocessor development systems support one user.

3. We needed high real-time performance: we typically run eight users on our UNIX (R) system at once, each collecting and processing data on disk from tablets that generate 700 interrupts a second over serial-line interfaces. Writing speeds can be about 0.5 seconds per character for hand-printing, and it takes us 0.2 seconds CPU time on the average to process and recognize a character.

4. Our application is sufficiently complex that we needed the full set of general software development tools available on UNIX (R) systems, rather than the limited set available on most microprocessor development systems.

5. We could not afford the risk of cross-compilation: A VAX system, for example, would have been viable for system development, except that the bugs from the use of two separate compilers and run-time environments (such as VAX native, and 68000 cross-compilation) had almost made us unable to meet our product commitments once before.

6. ROMable code: most "C" compilers do NOT produce ROMable code. The common problems include:

   + Run-time routines (typically not all are written in "C") are not sharable in PROM because of some static data buffer that cannot be shared.

   + "C" code from the compiler is re-entrant, but not sharable in PROM (separate code and data segments).

   + The initial values of variables are set by pre-loading the data segment, which does not exist when you power up a PROM-based system.

   + The actual "C" code is not ROMable because constants are put in the data segment, not the text segment.

3. Our development environment

Our particular project differs both from more ordinary micro-processor projects and from other large software systems projects, and also shares most of the requirements of both of these. One way of looking at it is that we have all the best headaches from both environments.

3.1 UNIX

The product code is some 46000 lines (10400 statements) of "C" code, not including some 22000 lines (5300 statements) of recognition code written in a proprietary applicative language we execute interpretively in the product (the interpreter is written in "C"). Other parts of our project that are not part of the run-time code, such as the compiler for the applicative language, add about 30000 more lines of code to our total.

The code is much too large and complex to ever develop and debug in assembly language -- at the same time, the product requires us to minimize memory size (which is a large part of our product cost), and maximize execution speed.

The software in the product is designed as a set of five co-operating tasks for managing various activities asynchronously. For example, the tablet we use interrupts continuously 100 times per second with seven bytes each time over a serial line, and the terminal must support communications simultaneously at 19.2 Kbaud to and from the host. The recognition activity must occur in parallel with both of these, since the recognition is always buffered behind the incoming tablet data, and the communications activity is asynchronous.

Due to the sheer size of the code going into an embedded environment where ALL errors are fatal -- you cannot re-boot -- we have had to use every conceivable method to find and prevent bugs. We check for proper range on all the arguments at the start of every routine, and for proper range on the return value at the end. We also check for proper values between all major sections of the code in-between. Our rule of thumb has been "when in doubt, put more debugging checks in", since they are conditioned out of the product code, and give us no performance penalty where it would hurt.

3.2 Non-UNIX

The UNIX kernel is not really suitable for embedded microprocessor systems, since it was never designed with this in mind. Instead, we purchased a "C" kernel from JMI Software, Inc., in source form, which we ported from the original Whitesmith's "C" to more standard "C". We chose this one kernel over other vendor offerings because it offered by far the most compatibility with UNIX features. This kernel gave us the following facilities:

1. Asynchronous multi-tasking in PROM. In our case, the number of tasks is fixed, which simplified the use of the kernel.

2. Inter-task communication using read/write queues. This is functionally the same as using named pipes in UNIX, which let us use identical source code in the development version of our code.

3. ROMable run-time support for heap management. We use a large heap to hold varying-length run-time descriptors for the graphics, forms, and block-mode features of our terminal product. The heap is shared among the control and processing tasks in our code.

4. ROMable versions of "C" library functions. Many UNIX library functions either crash on an error and give you a core dump, or pull in a large part of the I/O library for the error messages.

4. Some problems with UNIX and their solutions

Our particular design is dictated by the requirements for the character-recognition technology, which is unique, and quite complex. However, many of the practical problems we encountered are of interest to anyone who is now starting to do development with the M68000 microprocessor.

4.1 Compiler technology -- what our vendor's C compiler does well

Since our code is written entirely in "C", and must meet extreme real-time requirements, we are very dependent on the technology of the "C" compiler we use. Our vendor uses a compiler that traces it's history to the M.I.T. "C" compiler. There are several optimizations we found in this "C" compiler that were of great benefit, compared to other compilers available on competitive systems:

1. 16-bit "short" operations: The M68000 does not support 32-bit multiply and divide operations in its instruction set. Most "C" compilers evaluate all expressions in the "natural word length" of the machine, regardless of whether that is the type of the result that is needed. In this case it would be 32-bits, and the code speed would suffer needlessly.

2. Additional "register" variables: The original "C" compiler for the PDP-11 could only use the three remaining registers on the PDP-11 architecture: this restrictions has been perpetuated in many "portable C" compilers, including many of the cross-compilers.

3. Efficient run-time support for 32-bit integer operations: we re-designed the image-processing part of our code which cleans up the low-resolution image from our tablets to use no floating-point operations. Using 32-bit integers instead, and only when needed to protect against overflow, saved us considerable execution time and code space.

4. Efficient pointer arithmetic in pointer expressions: subscript array references in "C" are usually un-optimized, unlike optimizing compilers for FORTRAN-77 and RATFOR. Our vendor's compiler produced shorter code than many other compilers for the pointer expressions we used instead.

5. ROMable run-time routines for mathematical and string operations: our vendor had written some of these in assembly-language to get greater code efficiency.

4.2 What didn't come with UNIX (R) or from our vendor, but we could do ourselves

In any large software engineering project, there is always some need to adapt the tools to get the maximum performance for the particular design you are working on: a good tool will work well for the general case, and must also allow you to take care of the special trade-offs in a particular project. For example, "C" is nice, but you will never manage to do completely without assembly language for a real-time project.

The particular facilities we added to the available tools were as follows:

4.2.1 C compiler and linker

On UNIX, the "cc" command automatically calls the separate linker "ld" to put object modules together into an executable file. Code and data segmentation is a combined feature of these two utilities.

1. BSS data segment: the original "C" language had no clear way to say which module actually set up the data storage for a static variable. Most "C" compilers now say that whichever module gives it an initial value sets up the storage. However, if no module gives it an initial value, the linker sets it up automatically in the "BSS" segment, which is SEPARATE from the initialized data

segment.

On most systems (other than UNIX), having uninstantiated data items is an error. For our application, it is fatal -- we have to tell the linker the starting address and total length of data to match the RAM addresses in our product board, and the BSS data messes this up.

For our own code, we initialize all data. Our problem is the existence of BSS data items in the vendor's run-time routines, for which we have no sources.

2. Data tables in PROM: There was no way to put data tables into the text segment with the vendor's "C" compiler. (This may be added to the "C" standard in the future.) We got an unsupported copy of the M.I.T. M68000 assembler from the vendor, and used a combination of "sed" editing macros on the assembly-language output from the "C" compiler to force the entire contents of selected source modules into the text segment.

3. Floating-point constants and string constants in PROM: For historical reasons, most "C" compilers do NOT put constants in the text segment, but in the data segment. This was the source of several bugs in our product code.

   We separated all constants into PROM-only modules. After repeated support problems with every major release of UNIX from our vendor, our vendor ended up giving us the sources for the math and I/O libraries so we could make the changes ourselves. For others, we either used the UNIX (R) debugger to dis-assemble the code and find the offending data references, or reverse-engineered the routines from the user documentation.

4. The vendor's "C" did not pass all parameters as full-word values, as the "C" standard states: The exception case is the return value of a function. Since (for efficiency) we declared almost all values to be 16 bits ("short"), we had to re-declare all these functions so that our code did not fail because of trash in the high 16 bits of the return value.

5. Pointer arithmetic: Our vendor's "C" always did 32-bit operations, and did not convert divisions by powers of 2 into shifts, when taking the difference between two pointers. We re-wrote all the routines affected to update a parallel integer variable in parallel with the updates to the two pointers, instead of taking the difference between the two pointers at the end of the loop.

6. Formated I/O routines -- type-dependence and portability: The M.I.T. and Berkeley "C" compilers assume that all format control characters refer to "int"s, and the result is that you cannot do a formated read ("scanf") into a short integer on

these compilers -- you get a pointer addressing error.

4.2.2 System utilities - getting around the hiccups

Since our code was so large, we ran into several problems with system utilities which we were able to overcome.

1. "ld" and "ar" could not handle extremely large libraries:

   We have over 700 object modules in our code, since like most well-written "C" programs, there are a relatively large number of smaller modules. We broke all our code somewhat arbitrarily into 15 separate libraries, ran "tsort" on them by hand instead of with the switch to "ar", and put the libraries multiple times on the command line to "ld".

2. The vendor did not provide documentation on how to write a device driver:

   This may be common knowledge among UNIX systems programmers, but there were not many of those around at the time. This was especially annoying to us, since (by chance) every one of us had experience writing terminal and disk device drivers for much more awkward systems than UNIX. The only hang-up was the total lack of documentation. (Note: the vendor did supply all the necessary header files under license, and now offers a device-driver course.)

   We did our own search and hired an overly-expensive UNIX consultant for one day just to show us what the integration procedure was.

4.2.3 Hardware debugging support

No hardware micro-processor vender, to our knowledge, sells an efficient multi-user software development system integrated with a compatible in-circuit-emulator (ICE). We have had good success with our ICE, and the set of absolutely-required support tools from our vendor. The most necessary, of course, is some way to transfer executable code to the ICE and to a PROM blaster.

However, there are several things we would have appreciated if there were a vendor who provided more integrated support:

1. Compatible assemblers:

   The assembler syntax on our UNIX (R) system and any ICE we found use different mnemonics, and usually radically different instruction syntax. (In fact, on our UNIX system, the "adb" debugger uses incompatible syntax with the assembler!) This continues to be a source of some confusion and annoyance.

2. Object module and debugger support:

There is no way we can get the symbol values from the UNIX (R) object and load modules over to the debugger for our ICE. With such a large program, debugging in absolute assembly-language is excruciatingly inconvenient.

Part of the problem is that not even Motorola has standardized these facilities for the M68000.

3. UNIX terminal I/O is very slow:

Our tablets generate 700 characters a second each over serial lines. We estimate the terminal driver on our system can handle no more than about 250 characters per second aggregate data rate on all the terminals on the system before the system crashes or starts dropping characters. (This is slow enough that the function keys on a VT-100 terminal, for example, cannot be used at 9600 baud.) There were two places where this problem was much more than just an inconvenience, and we had to do some UNIX engineering.

We designed and installed our own serial interfaces for the tablets, for which we wrote our own driver. In order to get adequate throughput to our digitizers under UNIX (R), we used "ioctl" calls rather than character reads and writes. (IOCTL calls generally don't work across a networked file-system, since they can be written using physically shared memory, not just copyin/copyout transfers.

We have to transfer data from the system to our ICE over the system console port, which runs faster. For transferring data to the system from the ICE, we must run at a slow baud rate (45 characters per second) or set very long delays after X-OFF signals (longer than 150 milliseconds) to keep the system from crashing when its input buffers are flooded. This means an up-load or down-load takes at least 45 minutes.

4.2.4 System utilities that we wish we could have used, but couldn't

There are also utilities for code development that we found were much more difficult to use on our large body of code than on a more normal-sized development effort. The problems were mainly logistical: it is too much work to set them up and to maintain them by hand for large systems.

MAK: Our code is in some 700 "C" source files, which are inter-dependent as you would expect. In addition, we decided to pick up the definitions of all external data items via some 400 separate header files. We found that we tended to re-design many of our data-structures in the course of development: we did not want to have to edit an arbitrary number of references in a large number of "C" source files just because we modified one "typedef" or changed the value of one "#define" symbol.

Finding and setting up all the interdependencies for MAK would have been a laborious and error-prone task for us: there would be a high likelyhood that we would be unable to keep the interdependency list that MAK uses up to date. For large projects, some automated tool for establishing the interdependencies is a practical necessity.

SCCS: Similar to the problems with MAK, SCCS works less well with a system undergoing rapid evolution: the total number of SCCS deltas produced by rapid development and modification to the source code makes it impractical to use, unless you have a LOT of file-space to burn. (We currently have some 500 Mbytes of file-space on-line for a community of 16 users.)

SCCS seems to be targeted to projects that have already reached considerable stability, where changes are relatively localized within single source files.

We have solved this practical problem for the time being by:

+ Re-compiling ALL our sources frequently as a weekend batch job with "at" to avoid version skews. The process takes about 26 hours.
+ We have a simple set of shell scripts to do check-in/check-out of single source files. This way two people do not inadvertently edit the same file in conflict with each other. This system is easy to bypass, and requires everybody's co-operation to work.
+ We release our code "internally" every couple of months by blowing new PROMs for all the in-house users of our product. This way we get frequent feedback on user problems.

LINT: "Lint" has several deficiencies that make the people in our development team reluctant to use it:

+ Spurious error messages on 16-bit operations:

Since we are very concerned with code size, and since the 68000 processor requires you to do out-of-line subroutines for 32-bit multiplies and divides, we declare almost all our values to be "short" 16-bit items. We get spurious warnings about loss of significance on any pointer (32-bit) dereferencing for a 16-bit quantity. We theorize that this is a historical bug in "lint" from when it was developed for a 16-bit machine (the PDP-11).

+ Lint assumes that all sources are presented to it at once:

Our sources are too large to feed to "lint" in one pass. We get many spurious warnings about differences in parameter lists, returned values, and external data items because the other information "lint" needs is in some file what we left out.

We theorize that either "C" needs support of external modules like that of ADA (R) or Modula-2, or that "lint" and "mak" should be supplanted by a more powerful combined utility that would automatically check for all inter-module dependencies.

5. In summation ...

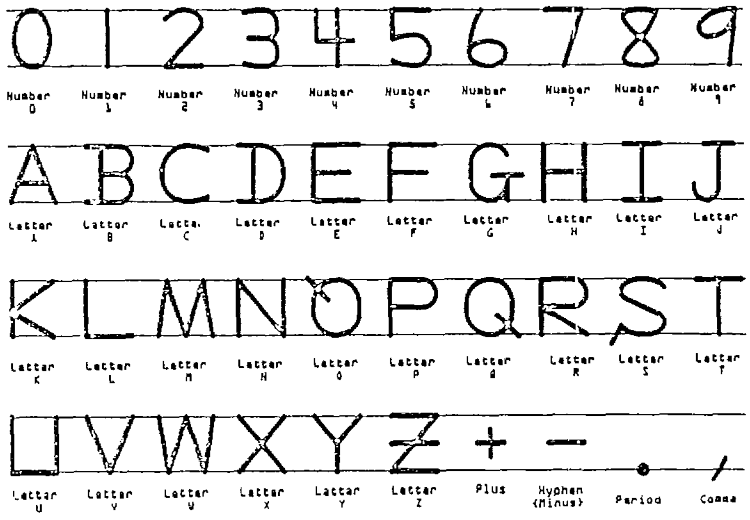
This paper relates our experiences using a multi-user UNIX (R) system for the development of an extremely large body of code for an embedded M68000 micro-processor running out of 160Kbytes of PROM.

For our application, there was significant benefit from doing our development on a large, multi-user UNIX (R) system. Among UNIX (R) vendors, there were also real differences in terms of usability and system support for our application.

Combined with certain other tools, such as in-circuit-emulators and the related software, the system has made it possible for us to produce a product of exceeding complexity in a short time, with few bugs. We would rate our overall experience as favorable.

We expect large, embedded micro-processor applications to become increasingly common in the future. We would recommend the practical benefits of our experience to any engineering group contemplating a large project of this type for the M68000. We hope that UNIX (R) developers in the industry will address the problems that we have encountered.

Pencept, Incorporated


0 1 2 3 4 5 6 7 8 9
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
A B C D E F G H I J
Letter A
Letter B
Letter C
Letter D
Letter E
Letter F
Letter G
Letter H
Letter I
Letter J
K L M N O P Q R S T
Letter K
Letter L
Letter M
Letter N
Letter O
Letter P
Letter Q
Letter R
Letter S
Letter T
U V W X Y Z + - . ,
Letter U
Letter V
Letter W
Letter X
Letter Y
Letter Z
Plus
Hyphen (Minus)
Period
Comma