AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

Debugging C Programs With the Blit

By T. A. CARGILL*

(Manuscript received August 1, 1983)

The Blit terminal is changing the way we debug C programs. Using multiple virtual terminals on the Blit, a programmer can interact simultaneously with several of the tools needed when debugging. This makes existing tools more useful and influences the design of new tools. In particular, the Blit cleanly separates the programmer's communication with a debugger from communication with the program being debugged. Moreover, `joff`, a debugger for C programs that run in the Blit, demonstrates the advantage of operating a debugger asynchronously with the subject process and the effectiveness of a source-level user interface based on pop-up menus. The graphics user interface supports "pointer chasing" through arbitrary data structures and graphical display of graphics data objects.

I. INTRODUCTION

This paper begins with a synopsis of debugging technology (see surveys published by Model and Myers).[1,2] This is followed by a discussion of the Blit terminal's effect on debugging C programs running under the *UNIX*™ operating system and then an example of `joff`, a debugger for C programs running on the Blit itself. The observations are pertinent to other languages used on *UNIX* systems, but only C has been used on the Blit. For programs on a *UNIX* system

* AT&T Bell Laboratories.

host, the multiplexed virtual terminals of the Blit increase the effectiveness of debugging with the standard tools. The Blit's hardware and software make its debugger quite unlike the debuggers used for *UNIX* programs. Several small scenarios illustrate tools and techniques used in debugging. (These examples are unrealistic and therefore require the reader to extrapolate to the effect in real debugging.) Some appreciation of the Blit terminal[3] and a reading knowledge of C[4] are assumed.

II. DEBUGGING TOOLS

Debugging is a complicated activity. A program isn't doing what it should, and the programmer has to find out what it is doing, so that the problem may be rectified or documented. Locating and understanding the errant part of the program is usually much harder than deciding how to correct the problem.

Initially, the programmer does not even know where to look; only the symptoms are known—the program's *external* behavior. The programmer constructs hypotheses about what may be wrong in the program and devises ways to test them. The results of each test are clues about the program that lead to other hypotheses. The more specific the hypotheses become, the more information the programmer needs about the internal behavior of the program, which is not normally observable.

A *debugger* is a tool for observing the internal behavior of a program. Generally, a debugger lets the programmer examine the state of the program at some point in its execution. Debuggers present the state of the subject program in different ways. They vary in the level of abstraction at which the program is viewed, from source programming language to machine language, and in the degree of user interaction:

- The most primitive debuggers give *dumps*: they print the contents of every memory location in the address space of the program at the time of a failure. The subject program executes no further; there is only information about its final state.
- Other debuggers *trace* the program: they print messages about selected events that occur in the execution of the program. Typical events are variable assignments and function calls. If the set of events must be fixed when the program is compiled or starts to run, the debugger is a batch tool, even if it runs in time sharing.
- *Interactive* debuggers involve the programmer in the execution of the program: when an event occurs the programmer enters a dialogue with the debugger and interactively examines the state of the program or modifies the set of events before restarting the program. The interactive nature is a great advantage; it is only after seeing the values of some variables that the programmer knows where to

look for other critical data. Each run of the subject yields more information than it would with a batch debugger.

The characteristics of a debugger are most influenced by the architecture of the machine executing the subject program; the machine architecture determines the ease with which the debugger can access and control the internal state of the program. An interpreter, a software machine, can easily provide ample support for a debugger. Hardware processors usually provide much less support. For example, with an interpreter it may be easy to implement a class of events based on changes in the values of variables by invoking the debugger after the completion of each statement. Hardware processors vary but may provide no more than a *breakpoint* event, halting the program when it reaches a particular instruction.

Debuggers are also influenced by the architecture of their operating environment. Under an operating system that permits users to execute only a single process, the debugger and its subject must be merged into one process. Several reasons make it undesirable to combine the debugger and the subject into a single process:

1. The debugger's presence in the subject process may result in different behavior, even to the point where the bug is no longer apparent.
2. The debugger is not protected; the subject process may overwrite it.
3. If process address space is limited, there may not be room for the debugger.
4. If the debugger and the subject must be bound before the subject starts to execute, the debugger cannot be invoked after something goes wrong in a production program.

If possible, it is therefore better to make the debugger a separate process, supported by operating system primitives for accessing the subject process.

These reasons for making the debugger a separate process have more to do with the implementation of the debugger than with its use. The programmer still perceives the debugger and subject as united if communication with them is through a single terminal. To the programmer, the drawbacks of a shared terminal are:

1. The process involved with each line of input and output must be determined.
2. The shared terminal may not behave properly if the debugger and the subject require it to operate in different modes.
3. Even in the same mode, Input/Output (I/O) may not interleave properly because of unflushed buffers, cursor control, and so on.

The solution is to use two terminals, one for the debugger and one for the subject. But whether the two processes can drive separate terminals

depends on the operating system again, and also on the availability of terminals.

A debugger is only one of the tools used in debugging. The programmer uses a full set of software tools to manipulate a great deal of information: the source program, data files, test results, other programs, subroutine libraries, documentation, news bulletins, mail messages, etc. Even though experienced programmers write programs with debugging in mind, they can rarely plan much of how to tackle a particular bug. It is hard to anticipate the course of a debugging session or what information will be needed; the results of each step determine where to look, what to consider, and what tool to use next. A dextrous programmer may rapidly apply a wide variety of tools.

III. USING THE BLIT TO DEBUG *UNIX* PROGRAMS

The Blit can multiplex a number of *UNIX* system shells.[3] Each shell runs in its own *layer*, a rectangular region of the screen that, by default, behaves like an ASCII terminal. The shells run asynchronously, writing to their respective layers at any time, ignorant of the multiplexing. The user creates, moves, reshapes, and deletes layers with a graphics mouse. The mouse also controls the way in which the layers overlap, and it selects the current layer, to which input from the keyboard is directed. Any obscured portion of an overlapped layer remains active; it can be written to at any time, and is restored when the layers are rearranged to make it reappear. The effect, for the user and the *UNIX* system alike, is as though the user had an array of terminals. A layer can also be tailored for an application with an arbitrary graphics program, down loaded from the *UNIX* system to run in the Blit's processor. For example, `jim`, a mouse-based multifile text editor, down loads its user interface process to a Blit layer.[3]

The Blit has a considerable impact on debugging, even when no debugger is used, as in the ever-popular method of debugging C programs by inserting print statements. When a program is being debugged, the ability to run multiple streams of *UNIX* system commands simultaneously is useful because the programmer has to perform so many different tasks. The subject program can run in one layer while the source text of the program is viewed in another layer. Perusing the source text and following the behavior of the subject program simultaneously is a great help, even if the text editor only displays text from one file at a time. The text editor written for the Blit, `jim`, makes it possible to flip rapidly among as many as 20 files, and arrange the files in overlapping windows within its layer. In a layer occupying less than half of the Blit's 800 × 1024 pixel display, `jim` can show a block of source text with a function call from one

source file, the body of the called function from another, and a set of definitions from a common header file.

None of the context of an editor or the subject program is lost when other tools must be used. Examples of the kind of tools that might be needed at any time are:

`grep`—to find occurrences of an identifier,

`diff`—to see how a file has changed,

`man`—to obtain a section of the *UNIX* system manual.

If executing a command takes a long time, the programmer need not wait for output before doing something else; each shell and tool responds independently. Without some discipline this can become chaotic, and it takes a little practice to use the Blit's layers to the best effect. Many programmers establish an idiosyncratic layout of the Blit screen, with fixed tools in layers at fixed positions. It is then easy to keep track of a few extra layers, handling other tasks as they arise.

Where they would not otherwise work, print statements can still be used for debugging on a Blit. Consider using print statements to debug a conventional *UNIX* system screen editor running behind a Blit layer. (A Blit layer can be programmed to emulate an arbitrary ASCII terminal.) As the editor moves the cursor around the screen, print statement output will overwrite editor text and vice versa; the editor also will lose track of the cursor's location. However, on the Blit the trace can be directed to a different layer, as follows:

1. The debugging output is written to another stream, say the standard error device:

```
fprintf( stderr, "keyboard( ) = %o\n", c);
```

2. The "pseudo-teletype" device associated with the layer to receive the trace is determined by using the `tty` command in that layer:

```
$ tty

/dev/pt/pt26
```

3. The editor's standard error output is directed to that device:

```
$ editor 2>/dev/pt/pt26
```

The editor now executes in one layer and the trace output scrolls by in another layer; there is no interference. Flow control characters from the keyboard can stop and start the trace output to prevent it from scrolling away too quickly. Of course stopping the output from the trace will not stop the editor until it blocks on full buffers.

In this case the print statements write unconditionally to the layer receiving the trace. A conditional trace is possible by adding a level of software to remove unwanted output. A file of directives, supplied by

the programmer, can be used to control which print statements are active and which should be ignored. Checking the control file periodically to see if it has changed provides asynchronous control of the trace; the control file can be edited (in a third layer) while the program is running, to select dynamically which trace output is produced.

So far, there has been no mention of the *UNIX* system debuggers `adb` and `sdb`. These tools are functionally alike. Both debuggers examine dump files from aborted processes and interactively control the execution of processes to be debugged. They differ in the level at which the subject program is interpreted: `adb` presents the program in terms of symbolic assembly language; `sdb` presents it in terms of its C source text. The *UNIX* system supports interactive debuggers as separate processes, but the subject must be a child process, created by the debugger.

For `adb` and `sdb`, isolation of the subject's I/O is handled easily. Both debuggers have a run command to start the execution of the subject process. The command takes arguments to be passed to the process, including I/O redirection. So the standard I/O devices for the subject process can be chosen to make it communicate with another layer. As with the other examples, the *UNIX* system I/O abstraction makes the technique possible. The Blit merely places a personal set of asynchronous devices at the programmer's disposal.

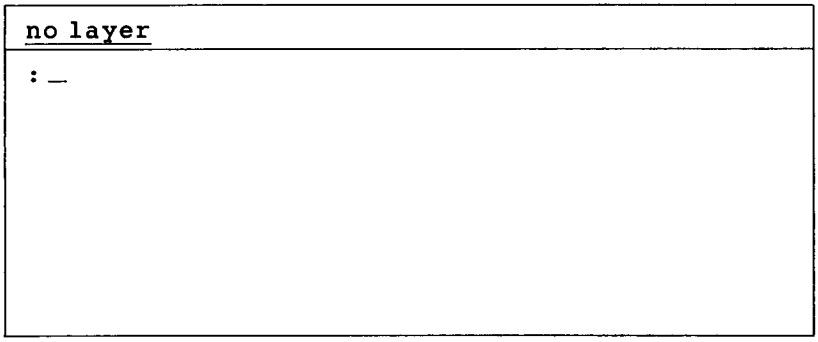
IV. DEBUGGING BLIT PROGRAMS

C programs down loaded into the Blit must also be debugged. The Blit environment is quite unlike the *UNIX* system environment and affects the way Blit programs are debugged:

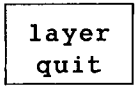
- Control flow in many Blit programs is driven by asynchronous input from the mouse, the keyboard, the clock, and a corresponding process on the host. This introduces some of the problems of debugging real-time software, particularly the difficulty of recreating conditions that produce an error. However, one classic bane of real-time programs is absent—response to interrupts is handled entirely by Blit system software.
- The primitive operations of the layer in which a program runs are those of bitmap graphics, not those of an ASCII terminal. A print statement only works if the program incorporates a set of output routines that interact properly with the graphics.
- The Blit has no memory management. Addressing errors may not be detected before a process has overwritten memory other than its own. However, one common addressing fault, indirection through location zero, is trapped by hardware.
- There is no preemptive scheduling. A looping process seizes the

processor; this prevents other processes from running. When this happens a special key on the keyboard must be used to kill the looping process.

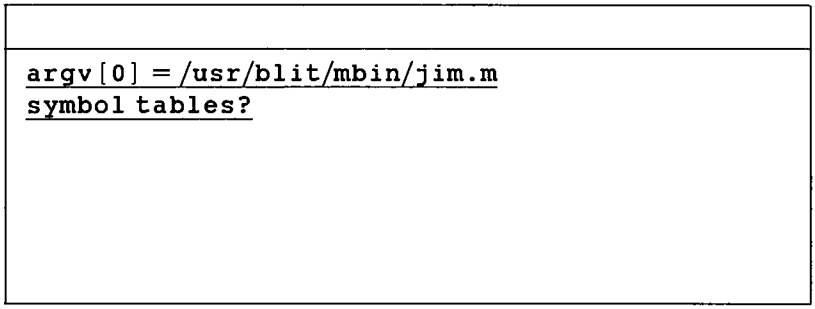
The `joff` debugger is the principal tool for debugging Blit processes. It is described more fully in Ref. 5, which includes some details of its implementation. `Joff` is quite unlike the *UNIX* system debuggers in the way it interacts with the programmer and the subject process. It is invoked in its own layer before being bound to the subject process to be examined. In a layer the command `joff` invokes the *UNIX* system process of `joff`, which immediately down loads the part of `joff` that runs in the Blit. Once loaded, `joff` is in an idle state with no layer to debug, indicated by the message in the status line at the top of its layer. The part of the display that has changed is underlined.



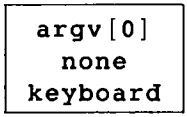
The remainder of the `joff` layer scrolls text up the screen and off the top when it fills. The "`: _`" in the scrolling region is a prompt for a keyboard command. In fact, keyboard commands are used very little; all of the common commands are from the pop-up menu on the right-hand mouse button. At the outset, the menu is just:



If `layer` is picked, the cursor changes to a bullseye icon. Moving the bullseye to a layer and pressing the right-hand button selects the process running in that layer as the subject of `joff`. Assume the layer selected is running the Blit text editor, `jim`. By examining the arguments with which `jim` was invoked, `joff` attempts to determine the host object file from which the process was down loaded, in order to find the symbol tables. The name of the object file should be element 0 of a vector of arguments, known by convention as `argv`, passed to the function `main`. This is printed in the scrolling area followed by a prompt, with the cursor switched to an icon calling for a menu selection:

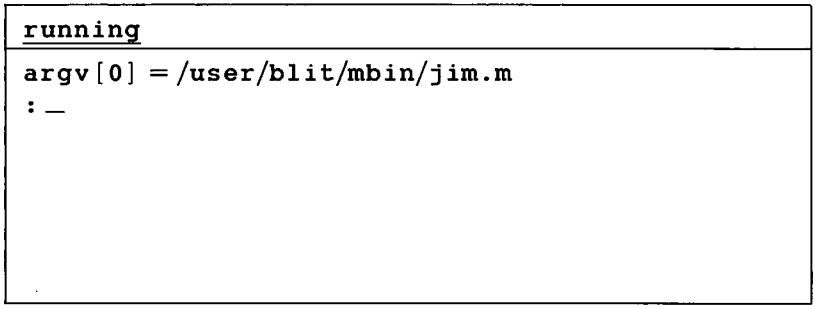


The menu presented is:



The expected response is `argv[0]`, but the other entries permit the special cases of proceeding without symbol tables, or entering the name of another file from the keyboard.

Having successfully bound itself to `jim`, `joff` displays the state of its subject in the status line:



In this case `jim` is running, that is, executing normally. If `jim` were stopped because of a run-time error or suspended by the down-loader before starting to execute, it would be selected as the subject in the same manner. The right button menu is now:

Notice that `layer` is still there; `joff` can be switched to another process at any time. Three new entries have appeared:

`breakpts`—to set and clear breakpoints,
`globals`—to examine global variables,
`halt`—to suspend the subject process.

A menu entry appears only when its use is valid. There is no need to breakpoint or halt `jim` before using `globals` to see the values of its global variables. Picking `globals` changes the menu on the right button to:

```
Drect       glb
F_rectf     glb
Jdisplay    glb
Null        glb
P           glb
_string     glb
boxcurs     glb
bullseye    glb
butfunc     glb
complete    glb
current     glb
deadmouse   glb
```

This shows only the top 12 items from a sorted list of the 40 global variables of `jim`. A scroll bar (not shown) beside the menu scrolls the 12-item window quickly through the full list. Each variable is identified as global by the `glb` tag; showing the class of each variable is needed to resolve ambiguity in some menus. Picking a variable from this menu, for example, `current`, requests that its type and value be displayed; `current` is a pointer to the portion of text displayed from the file currently being edited by `jim`:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
struct Textframe * : current?
```

Note that there was no need to refer to the source text of `jim` to find this variable. To compose this entire example I used only `joff` to feel aound inside `jim` until I found interesting objects. Of course the blind alleys have been removed from the transcript. In general, it is quite practical to examine the data structures in a working program without reference to the source text.

The value of `current` is a pointer to a `Textframe`[†] structure at

[†] To ease reading, license is taken with the length of identifiers. In the symbol tables, all identifiers are truncated to eight characters.

address 53180. The prompt is an invitation to use a menu to construct an expression based on `current`, and examine the data structure. This menu begins:

```
    ~[?]
Textframe{~}
   ->rect
~->scrollre
~->totalrec
   ~->str
    ~->s1
    ~->s2
 ~->scrolly
   ~->file
~->obscured
```

Each entry is an expression in which tilde represents the active expression, `current`. The rectangle where text is displayed is stored in the `rect` field of a `Textframe` structure, `current~->rect`, selected by picking `~->rect`:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
struct Rectangle: current->rect?
```

Now the active expression is a `Rectangle` structure. No value has been shown—it is not a scalar or a pointer. There is a new prompt to extend the expression and the menu is:

```
Rectangle{~}
  ~.origin
  ~.corner
%outline(~)
newframe(~)
  rXOR(~)
```

`Rectangle{~}`, at the top of the menu, is not a C expression. It is a request to display each field of the structure and its substructures,

recursively. The standard Blit representation of a rectangle is `struct Rectangle`:

```
typedef struct Point {
                short x;
                short y;
} Point;
typedef struct Rectangle {
                Point origin;
                Point corner;
} Rectangle;
```

Three functions—`%outline( )`, `newframe( )`, `rXOR( )`—also appear in the menu, for reasons discussed below. Picking `Rectangle{~}` produces:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
current->rect={origin={x=27,y=452},corner
={x=787,y=984}}
struct Rectangle: current->rect?
```

This selection has not moved deeper into the data structure and `current->rect` reappears as the prompt, with the same menu. Picking `~.origin` gives:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
current -> rect = {origin = {x = 27,y = 452},corner
= {x=787,y=984}}
struct Point: current->rect.origin?
```

and the menu for a `Point`:

```
Point{~}
  ~.x
  ~.y
%point(~)
pttoframe(~)
```

In this menu, %point(~), and in the previous menu, %outline(~), are examples of functions built into joff for graphically displaying the standard Blit graphics data structures. A point is shown graphically by a flashing a cross hair at its position on the screen, and a Rectangle by drawing its outline in exclusive-or mode. Graphic display of graphics objects is the natural way to debug graphics programs; many bugs are immediately apparent. For example, it might be obvious from an image that a rectangle has been rotated and translated, an observation that might not emerge from the numeric coordinates.

The Point menu also contains pttoframe(~). This is the function in jim that maps a screen position to a pointer to the Textframe covering the position; it determines to which of the jim files the mouse is pointing:

```
Textframe *pttoframe(pt)
Point pt;
```

This function is included by virtue of being applicable, that is, its only argument matches the type of the active expression. In general, this brings into the menu many useful functions, such as coordinate transformers and special display functions. Picking pttoframe(~) makes

```
pttoframe(current->rect.origin)
```

the new active expression and evaluates it:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
current->rect={origin={x=27,y=452},corner
={x =787,y=984}}
struct Textframe * : pttoframe(current->
rect.origin)=53180
struct Textframe * : pttoframe(current->
rect.origin)?
```

All is well—the pointer returned by pttoframe is the value of current, 53180.

Throughout this interaction with joff, jim continues to run—idling, waiting for mouse or keyboard input, its data structures unchanging. At any time it is possible to switch layers and interact with jim to manipulate it and see how it behaves. With jim executing asynchronously, joff does not try to present a consistent view of the internal state of jim; each expression is evaluated separately and reflects the values of the jim variables at the time of evaluation. To guarantee a consistent view, jim must be suspended, by using the halt or breakpts command from the main menu. Picking

`breakpts` yields a menu containing the one hundred functions in `jim`, beginning:

```
gcalloc( )
Rectf( )
Send( )
addstring( )
adjustnames( )
box( )
buttonhit( )
buttons( )
center( )
charofpt( )
closeall( )
closeframe( )
```

Picking one of the functions, say `box ( )`, produces a further menu for setting breakpoints:

```
   call
   return
   both
 >none
```

The ">" tag on `none` indicates that no breakpoints have yet been set on box. Picking `call` sets a breakpoint on any call to `box`. Reshaping the current text frame in `jim` results in a call to `box`, to clear a rectangle and draw a border around it:

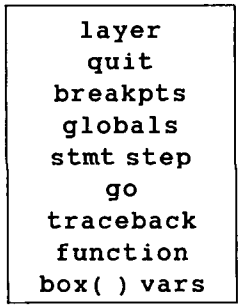
```
box(t)
Textframe *t;
```

Next, `joff` announces the breakpoint in the status line:

```
call: box(t=53180)
argv[0]= /user/blit/mbin/jim.m
struct Textframe * : current=53180
current->rect={origin={x=27,y=452}, corner
= {x=787,y=984}}
struct Textframe * : pttoframe(current->
rect.origin)=53180
: _
```

Correctly, the `box` argument, `t`, has the same value as `current`.

With `jim` suspended, the `joff` menu becomes richer:



The new entries are:

`stmt step`—to execute one source statement from the subject,

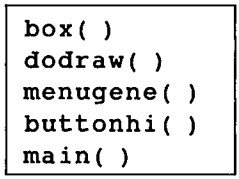
`go`—to restart the subject,

`traceback`—to list the functions on the callstack,

`function`—to select the current function from the callstack,

`box( ) vars`—to examine local variables in the current function, `box( )`,

A menu of local variables behaves like the menu of global variables. The current function can be changed by picking function from the main menu. This produces a menu of the functions on the callstack:



Picking `dodraw( )`, for example, makes it the current function; `dodraw( ) vars` then appears in the main menu and its local variables are accessible instead of those of the `box`.

Though far from exhaustive, this demonstration of `joff` emphasizes the characteristics that make it an effective tool:

1. It is bound dynamically to an arbitrary subject process, in any state.
2. It executes asynchronously with its subject.
3. A simple, mouse-based user interface supports all the basic commands and expressions for "pointer chasing."
4. Graphics data are displayed graphically.

V. DEBUGGING DISTRIBUTED PROGRAMS

Applications for the Blit are usually composed of two communicating processes, one running on the Blit processor and one running in the *UNIX* system. The example above ignored the other process of `jim`—managing the files on the host. There is no difficulty when both processes must be debugged simultaneously. Debugging the *UNIX*

system process does not interfere with debugging the Blit process. None of the debugging techniques makes any assumption about what is happening elsewhere. For example, if the *UNIX* system process is executed under `sdb`, and `joff` is applied to the Blit process, three layers are used: one for the application and two for debuggers. Neither debugger is aware of the other.

VI. CONCLUSION

Using the Blit to debug *UNIX* programs makes existing debugging tools and techniques more effective. The Blit's multiple virtual terminals make it easy to exploit the *UNIX* system's inherent character. Multiple shells help to handle the diversity of tasks involved in debugging. I/O on the *UNIX* system cleanly isolates debugging activity from the program's normal communications.

A debugger for C programs on the Blit takes advantage of the Blit's hardware/software architecture to provide more function and a better user interface than the *UNIX* system debuggers. The Blit debugger is bound dynamically to a running process and then executes asynchronously beside it. With a menu-based user interface driven by the mouse, the keyboard is rarely needed, even when using expressions to examine complex data structures.

VII. ACKNOWLEDGMENTS

I wish to thank Brian Kernighan and Rob Pike for their comments on drafts of this paper.

REFERENCES

1. M. L. Model, "Monitoring Systems Behavior in a Complex Computational Environment," CSL-79-1, Xerox Corp., 1979.
2. B. A. Myers, "Displaying Data Structures for Interactive Debugging," CSL-80-7, Xerox Corp., 1980.
3. R. Pike, "The *UNIX* System: The Blit: A Multiplexed Graphics Terminal," AT&T Bell Lab. Tech. J., this issue.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
5. T. A. Cargill, "The Blit Debugger," J. of Syst. Software, *3*, No. 4 (December 1983), pp. 277–84.

AUTHOR

Thomas A. Cargill, B.S. (Mathematics/Computer Science), 1973, University of Reading, England; M. Math., 1975, and Ph.D., (Computer Science), 1979, University of Waterloo, Ontario; AT&T Bell Laboratories, 1982—. Mr. Cargill was Assistant Professor at the University of Waterloo from 1980 to 1981. At AT&T Bell Laboratories he is a member of the Computing Science Research Center. His research interests are in software support for software development.