SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(2), 153–168 (FEBRUARY 1985)

Implementation of the Blit Debugger

THOMAS A. CARGILL

AT & T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974, U.S.A.

SUMMARY

joff **is an asynchronous, source-level, break-and-examine debugger for C programs running on the Blit, a programmable bitmap terminal.** joff **is implemented as two processes: a small process running in the terminal and a larger process running in a time-sharing host. The constraints on its design and a menu-driven user interface combine to present an unusual set of implementation difficulties. The way the problems were tackled and the degree to which they were solved may be interesting to those designing other debuggers. Operating system designers should assess the merits of a debugger that runs asynchronously with its subject and consider providing the necessary support.**

KEY WORDS Asynchronous debugging Remote debugging Graphics debugging Symbol tables Bitmap terminal

INTRODUCTION

The Blit[1] is a bitmap graphics *terminal*. It consists of a Motorola MC68000 processor, 24K bytes of ROM, 256K bytes of dual-ported RAM, of which 100K is mapped to a 1024×800 monochrome display, a mouse, an ASCII keyboard and an RS232 port. Pike[2] describes how the Blit's control software, mpx, multiplexes several logical bitmap terminals and connects them to corresponding processes on a host computer running the Unix time-sharing system. This creates a computing environment in which the user communicates with a number of host programs simultaneously. Simple applications, such as the mail reader, require only a dumb terminal and run in a window driven by a default process. Some graphics applications require a window emulating a specific terminal, such as a Tektronix 4014. This is handled by downloading an appropriate emulator program to execute as the process controlling one of the windows. Other applications, for example the Blit's mouse-driven text editor, are programmed specifically for the Blit and download their own programs to provide custom user interfaces. Debugging the downloaded processes is the subject of this paper.

The architecture of mpxterm, the part of mpx running in the Blit, is an attractive basis for interactive debugging. The process abstraction supports a debugger as a separate process, communicating with the user through a dedicated window, known as *layer* in the Blit. A layer is a virtual terminal with mouse and keyboard input and a display bitmap; each process under mpxterm operates in its own layer. This means that the natural implementation of a debugger under mpxterm avoids the difficulties that arise when a debugger must be bound into the program to be debugged or share its terminal.

0038–0644/85/020153–16$01.60

Received 26 September 1983
Revised 24 May 1984

joff, a break-and-examine debugger, was written for the Blit. joff runs in its own layer and can dynamically switch its attention from process to process. It runs asynchronously with the subject process, allowing the user to examine variables and set breakpoints while the subject process continues to run. In the first version, all user input came from the keyboard, expressed in a typically cryptic command language. When this version worked, the idea of augmenting the user interface with pop-up menus of common commands seemed attractive; a goal emerged of making a debugger that was easy to use. After a few false starts, a successful menu-based interface was implemented. An indication of its success is that many programmers have used it effectively without reading any documentation. An evaluator for expressions was added, initially only for expressions entered from the keyboard. This was extended to support menu-built expressions for chasing through data structures. The last major addition to joff was graphical display of the standard graphics data structures: Point, Rectangle, Texture and Bitmap. The user interface will not be treated in depth; more details may be found in Reference 3.

Thus, joff is the result of a sequence of evolutionary steps rather than a coherent design. As the discussion will show, if redesigned from scratch today, joff would be very different.

DEBUGGING BLIT PROCESSES

The Blit's MC68000 supports breakpoints and single instruction execution through internal interrupts. This provides the basic mechanism for break-and-examine debugging of compiled code. There is no need to compile any special hooks into the generated code, and production programs, including those processed by the optimizer, can be handled without modification. The Blit has no memory mapping; joff can read and write the code and data of any other process or state variables in mpxterm process descriptors. The MC68000 traps divide by zero, illegal instructions and physical memory violations.

Optimized compiled code can greatly complicate a debugger, but the code generated by the C compiler is quite straightforward. It uses a uniform calling convention and stack frame format, with function arguments and local variables addressed from a dedicated register. Expressions are evaluated using fixed scratch registers. Registers are allocated to variables statically within each function. On entry, each function pushes the registers it needs for variables onto the stack; before exit, it restores them.

An object program peephole optimizer reduces code size by about 15 per cent, mostly by using shorter addressing modes. Because it treats complete object programs, it can determine when short addresses may be used between separately compiled modules. It does not reorder any code sequences, and it does update the symbol table. This peepholer replaced an earlier assembly code peepholer, which achieved about the same compression, but could not be used safely with joff because it reordered code without updating the symbol table.

An asynchronous debugging process fits smoothly into mpxterm's scheduling mechanism. Once dispatched, a process retains the physical processor exclusively; all normal interrupt service routines return to the interrupted process. Eventually, the process blocks for I/O or pauses to display an image and relinquishes the processor explicitly. A round-robin search then selects and dispatches the next process that is

able to run. The interrupt routine for a trap or a run-time error records the event in the process descriptor and calls the round-robin dispatcher.

Under this discipline, the debugger and subject process can execute asynchronously, while the user interacts freely with both. Moreover, the debugger needs no special services or privileges. To halt the subject process, the debugger can read and write the subject's process descriptor, indivisibly with respect to the subject's execution; unless the debugger gives up the processor, the subject cannot run. If the subject encounters a breakpoint trap, it is suspended by the interrupt, but the event is not handled until the debugger is dispatched and checks the subject's process descriptor. To execute a single instruction of the subject, the debugger arranges the stack so that the subject is dispatched with the *trace* bit set in its program status word. The trace bit causes the processor to generate an internal interrupt after the execution of each instruction. The debugger then yields the processor. By virtue of the round-robin schedule, the next time the debugger is dispatched the subject must have been dispatched, executed a single instruction and trapped. The interrupt routines of mpxterm handle each exception interrupt in isolation; they know nothing of what the debugger is doing and offer no special services. Since the debugger has no special privileges, one copy of the debugger may be applied to another.

Except for the original assembly code peepholer, all of the software upon which joff depends is maintained by members of a single research community. This made it easy to attack each of joff's interfacing problems at the best point in the software chain, from the C compiler through to mpxterm.

DESIGN OUTLINE

joff is implemented as two communicating processes, one in the terminal and one in the host (Figure 1). The terminal process handles the keyboard, mouse and display and has access to the subject process and mpxterm. There are really two independent functions, user interface and subject process interface, folded into a single process because the user and the subject process happen to be on the same processor. Logical control of the debugging rests with the host process. It has access to the file system and

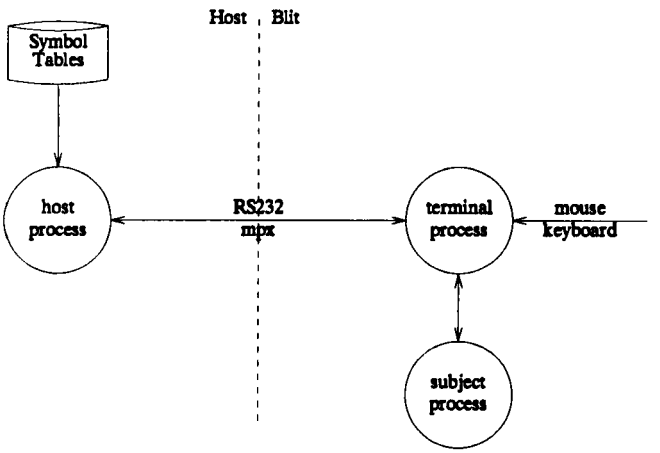


Figure 1. joff*'s process structure*

therefore the symbol tables, and has responsibility for almost all decisions calling for semantic interpretation of the subject.

From the outset it was clear that the debugger should minimize its use of terminal resources, so that it could be applied to the largest applications while sharing the terminal with the editor and other tools. The two critical terminal resources are memory for the process and physical space on the screen. The 156K bytes of off-screen RAM only go so far in the realm of multi-process bitmap graphics and it would be fruitless to have a powerful tool that could not be applied under real conditions. Bitmap display space is also memory and overlapped graphics does not mean that every process has the full screen at its disposal.

Minimizing memory results in a terminal process with the least function sufficient to communicate with its counterpart on the host and interact with mpxterm, the subject process and the user. This forces almost all the function back into the host where memory is more than ample. For example, although it would be wasteful in the terminal, a parser generated from a yacc[4] grammar is used for the keyboard command language because it runs on the host.

The host process starts as 70K bytes of code on a VAX from 6000 lines of C; it grows as it allocates symbol tables and other dynamic data structures. The terminal process is 10·5K bytes of code on an MC68000 from 1000 lines of C; it uses a fixed 3·5K bytes of data. Using fixed data structures avoids allocating from the common pool of physical memory and thereby reduces the chance of corruption from bugs in other processes. Fixed size tables are satisfactory in many cases; there is a fixed limit of 32 breakpoints. However, it imposes a limit on the size of menus held in the terminal, complicating the user interface.

SYMBOL TABLES

The C compiler for the MC68000 is a descendant of the Portable C Compiler[5] and produces symbol tables derived from those generated for the sdb debugger.[6] Symbol tables are output by the compiler as special directives interleaved in the assembly code it generates. The assembler and loader append this information to the object file as a sequence of fixed-length records, with the following fields: a variant tag, an identifier, an address and a tag-dependent value. The structure of the program is flattened to various sentinel records: START_FILE, START_FUNCTION, START_BLOCK, END_BLOCK and END_FUNCTION. Within this, each data object is described by an entry of one or more records that fully specify its type and location. The original sbd tables for data did not give complete information about C structures and arrays; this was corrected for joff.

Different record types specify the scope and storage class of variables: REGISTER, LOCAL, ARGUMENT, STATIC, GLOBAL. The field specifying the type of a variable encodes its base type with a sequence of qualifiers, packed into 16 bits. For example, in C

```
struct Line *edge[10];
```

declares a global variable edge, a 1-dimensional array of 10 elements of pointers to Line structures. The GLOBAL record for edge specifies its name and address, but has only the 16-bit field encoding the type

```
array_of-pointer_to-structure.
```

The remainder of the type is found in a subsequent DIMENSION record, specifying 10, and a TYPE_ID record specifying the identifier of the base type, Line.

Definitions of all the user-defined types appear at the end of each source file. Between START_USER_TYPE and END_USER_TYPE sentinels, each member of a type is described. The description of a field of a structure is like that of a variable of the same type, with an offset instead of an address.

Corresponding to each executable source statement there is a single-record entry, with no information about the statement's type or other control flow. For example,

```
.
.
101  while( p(x) ){
102          if( q(x) )
103                  x = f(x);
104          else
105                  x = g(x);
106  }
107  x = h(x);
.
.
```

generates 5 SOURCE_LINE records, each of which contains only a line number and program counter:

```
.
.
SOURCE_LINE:101:PC_101
SOURCE_LINE:102:PC_102
SOURCE_LINE:103:PC_103
SOURCE_LINE:105:PC_105
SOURCE_LINE:107:PC_107
.
.
```

where the program counters are the values that labels inserted at the beginnning of those statements would have:

```
.
.
101  PC_101:  while( p(x) ){
102           PC_102:    if( q(x) )
103                    PC_103:  x = f(x);
104           else
105                    PC_105:  x = g(x);
106  }
107  PC_107: x = h(x);
.
.
```

This mapping between source lines and code space is sufficient for implementing breakpoints, but it only supports single-stepping at the level of basic statements.

Without better tables, loops and conditionals cannot be treated as atomic statements. For example, there is not enough information to step over the whole loop (from PC_101 to PC_107) or once round the loop (from PC_102 to PC_102, say) in a single step.

Macros are used extensively by Blit software for constants and in-line function expansion, but no symbol tables remain from the C macro processor. The 'source' for the compiler's tables is really the macro processor's output with directives adjusting line numbers to correspond to the original source.

joff must handle two distinct symbol tables simultaneously: those of mpxterm and those of the current subject process. Since the debugger can be bound to a new subject process at any time, it must be able to substitute a new symbol table dynamically. The symbol table is therefore implemented as a two-element array of symbol tables, held in host memory. Using memory is more than a just programming convenience; as each table is read every record is augmented with a pointer to the next record with the same tag. Also as the table is read, the user-defined data types are checked for discrepancies between separately compiled source files. This generates valuable diagnostics occasionally, but the real motivation is to avoid joff being confused by two different user types with same name, even though it is legitimate C.

There are two search functions returning pointers into the symbol table:

```
lookup( table_seq, tag_seq, identifier, location )
visible( start_from, tag_seq, identifier, location )
```

The table_seq argument to lookup specifies the tables to be searched and the order in which they are to be searched. Its possible values are the constants: USER, MPX, USER_THEN_MPX. The tag_seq argument is a sequence of tag values limiting the search to records having one of the tags. identifier is a string and location is an address; either can be a 'don't care' value. lookup searches by table and tag-within-table. When successful, it returns a symbol named identifier with the greatest address less than or equal to location. The arguments to visible differ in only one respect: start_from is a pointer to a symbol table entry. When successful, visible returns the first matching symbol that is visible from the context given by start_from, under the scope rules of C.

These primitives do not hide the symbol table's representation from the rest of the program, but they are sufficient for many needs. For example, consider mapping a program counter into a list of local variables.

```
stmt = lookup( USER_THEN_MPX, STATEMENT, DONT_CARE, pc )
```

returns the entry for the statement within which the program counter lies, if there is one.

```
local = visible( stmt, LOCAL_TAGS, DONT_CARE, DONT_CARE )
```

finds the first local variable, after which the search must start from that variable,

```
local = visible( local, LOCAL_TAGS, DONT_CARE, DONT_CARE )
```

and continue until visible fails to yield another.

More complicated operations generally call for a sequential search through some part of the table. Consider trying to move from the table entry for a given statement to the following statement within the same function, to find the address at which to terminate a single-stepping loop, say. If the statement is somewhere in the middle of a

function, the next statement is found by following the extra pointer. But the pointer from the *last* statement in a function points to a statement in a different function, perhaps even from a different source file. The search must therefore be performed sequentially, checking which sentinels are encountered. It is undesirable to have this kind of detail spread throughout the debugger, obscuring the real function of the code in which it appears and making modification of the data structure more involved. If the symbol table were tree-structured, using the natural abstract syntax tree, it would simplify coding the custom search routines.

Since this is 'obvious' in retrospect, what were the reasons for the mistake? First, it is partly the result of a mental block that the author of a debugger accepts the symbol table representation dictated by the translator. Usually debuggers search symbol tables directly from the files in which the translators leave them. Indeed, that was tried initially for joff, but abandoned in favour of holding the tables in memory *without* chaining. Eventually, the chaining of like records was added, but the possibility of more radical restructuring was not considered. Secondly, the chained version of the table structure with the search routines described above was sufficient for early versions of joff. By the time joff had evolved to the point where the real inadequacy was apparent, it didn't seem to be worth going back to re-work the tables from scratch. Each ugly addition was rationalized separately and furthered the commitment to the flat structure.

Beander faced similar problems in structuring the symbol tables for VAX DEBUG.[7] That debugger builds hashed symbol tables incorporating scope information from flat tables in the object file. But the technique proved so slow for large programs that the user must explicitly request access to the symbol table for each separately compiled module before that part of the debugger's table is built. Symbol tables may remain a problem for debuggers until all compilers leave tables in a format much closer to their own internal form. That is the method used by the SWAT* debugger.[8]

ASYNCHRONOUS REMOTE DEBUGGING

Control of the execution of the subject process is distributed between joff's host and terminal processes. The basis of their communication is a subject state variable which each maintains, having one of the following values: RUNNING, HALTED, BREAKPOINT_TRAPPED, SINGLE_STEP_TRAPPED, EXCEPTION_TRAPPED. The RUNNING state and the various TRAPPED states have obvious interpretation. A process is HALTED when a flag is set in its process descriptor to prevent mpxterm from scheduling it to run. This provides clean asynchronous suspension of the subject.

The host process refreshes its copy of the state variable by requesting the subject's state from the terminal process at appropriate times. Beyond this common model, the host and terminal have different views of the subject process. Only the terminal is aware of the instruction-level control of the subject. Only the host is aware of the user-level commands driving the debugger as a whole.

The implementation of breakpoints illustrates the division of labour. Based on user commands and symbol table information, the host is responsible for deciding where breakpoints are located and maintains a table of breakpoint addresses, which it sends

*SWAT is a trademark of Data General Corporation.

to the terminal process. The terminal process is responsible for laying down the trap instructions and then later restoring displaced instructions, as the state of the subject and the breakpoint table change. The finer details of the state of the subject process are the responsibility of the terminal process. Thus, if the subject process is in any non-RUNNING state, a common GO request is used by the host to tell the terminal to restart the subject. If the subject is halted at a breakpoint, the terminal process knows to back up and single-step the lost instruction, before re-laying the breakpoints and restarting the subject.

The terminal process can gain exclusive control of the physical processor it shares with the subject and guarantee its view of the subject before modifying the subject's process descriptor, say, to alter the trace bit in the program status word. This is not true of the host. The subject's state may change after a request is issued by the host, but before the terminal process can complete it. For example, if the host believes the subject is RUNNING, it will accept a user command to halt it. But the subject may have changed state, say, by encountering a breakpoint, before the terminal process receives and executes the HALT request from the host. The terminal process therefore verifies each request from the host and ignores those that are invalid. This means that the host must follow the HALT request with a check on the state of the subject. The host cannot simply assume a subject state transition from RUNNING to HALTED.

The right division of labour between the host and terminal was not always found at the first attempt. In the earliest implementation of single-stepping C source statements the host entered a loop, in each iteration of which it told the terminal to execute a single machine instruction of the subject. The communications overhead needed for this method was too great, although the traffic between the host and terminal, per instruction, was modest: one 2-byte and one 4-byte message from the host and one 4-byte reply. But, neither the host nor terminal operating systems favour processes exchanging small messages in closely coupled interaction. The packet protocol and process context switching overhead result in real-time response that is difficult to predict, and in this case was too slow. In this respect, joff's needs are atypical among Blit applications. To improve response, the host now instructs the terminal to single-step machine instructions until the program counter leaves a specified range of addresses. However, the host still performs the outer iteration for the command to single-step multiple source statements.

The communications overhead in building call stack tracebacks also led to a redistribution of function between the debugger's two processes. Originally, the host probed each memory location separately as it moved from stack frame to stack frame. Performance was improved by introducing requests that let the host obtain multiple values, computed by the terminal process, from the stack frame and related instructions. A side benefit of this was that the complicated traceback procedure on the host was somewhat simplified, though it is still the most fragile part of the debugger with numerous special cases handled by *ad hoc* code. Some of the difficulties are: (a) neither the subject's core image nor the symbol table is sufficient to determine the start address of a function—it might require either or both; (b) peephole optimization may modify the standard function prologue and epilogue; (c) the extended precision arithmetic package uses a non-standard stack frame; (d) update to the stack frame is not an indivisible operation and it is possible to catch a process with its stack in an inconsistent state. Even with these difficulties, it is still worth verifying, as far as possible, that the traceback is valid and the core image agrees with the symbol

tables. Thorough checking complicates the debugger and calls for more communication between host and terminal, but pays dividends when the stack or code of the subject process has been overwritten or the wrong symbol tables are used by mistake.

INTRA-DEBUGGER COMMUNICATION

In the protocol between the host and terminal processes, the host is master and the terminal is slave. The byte-stream from the host is read byte-by-byte in the main loop of the terminal process. Except for an escape character, each byte is passed to a display function that interprets it as a character to be printed or as a control character. The control characters provide rudimentary functions, such as selecting whether subsequent text is written to the status line at the top of the debugger's layer or appended to the scrolling region that occupies the remainder of the layer. Control characters also clear regions of the screen.

The escape character signals the start of a sequence of bytes for special interpretation as a message in which the host requests an operation to be performed by the terminal process. The byte following the escape gives the type of the request. The format of each request is known by the terminal; the only variable length data are strings, which are terminated by the null byte. Based on the type byte, the main loop of the terminal process switches to code to handle the host's request. If the request calls for results to be returned to the host, they are transmitted back by the handler in a known format, and read by the host.

A layer of software on the host makes most of the requests to the terminal appear as remote procedure calls. For example, given an address within the target, the function peeklong requests the 32-bit integer at that address from the terminal's memory and returns it as its result, having made the necessary adjustment in byte ordering. The style of the protocol can be seen from some hypothetical host source code for displaying the value of a 32-bit integer global variable, foo.

```
symbol = lookup( USER, GLOBAL, "foo", DONT_CARE );
if( symbol ) {
    printf( "foo = " );
    printf( "%d\n", peeklong(symbol—> address) );
}
```

This produces a line on the terminal, say:

```
foo = 445
```

It is transparent in the host code that the strings of printable characters sent to the terminal from the two calls to printf have been interrupted by a request to fetch a value. The first printf sends the sequence of bytes:

```
'f', 'o', 'o', ' ', '=', ' '
```

The call to peeklong in the argument to the second printf sends

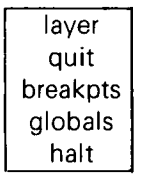
```
ESCAPE, PEEK_LONG, α, β, γ
```

where $\alpha\beta\gamma$ is the 24-bit address of foo, and reads the 4-byte reply from the terminal. The second printf then completes the line by continuing with printable characters:

```
'4', '4', '5', '\n'
```

The most complex request from the host is WAIT_EVENT, asking for an arbitrary event from the terminal. The possible responses from the terminal are a keyboard command, a mouse menu selection or a signal that the subject process changed from RUNNING to some other state. Before the host issues WAIT_EVENT, it must have primed the terminal with the menus to present in response to each of the mouse buttons. The terminal then loops, polling the mouse, the keyboard and the state of the subject process. The first *completed* event is the terminal's response to the host. For example, the user may enter some characters from the keyboard, but not complete the line with a carriage return, and then raise a mouse menu, but not make a selection. At this point, if the user pauses to think, and the subject process encounters a breakpoint, the terminal reports the breakpoint to the host. The incomplete line from the keyboard is erased, but restored as soon as the keyboard is active again.

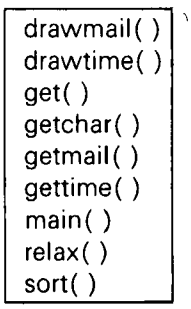
Not all the interaction between the debugger, the subject process and the user works this well. Consider what happens when menus are used to set breakpoints at the function level. Picking breakpts from the main pop-up menu on the right mouse button:



invokes the following host function, outlined here in pseudo-code:

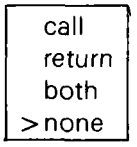
```
function_breakpoints( )
{
        build functions_in_subject_menu;
        while( selection from functions_in_subject_menu ) {
                build breakpoint_placement_menu;
                if( selection from breakpoint_placement_menu )
                        update breakpoint_table;
        }
}
```

The function builds and presents a menu of all the functions in the subject process. For a small program such as sysmon (which graphically monitors the host system load, runs a clock and announces the arrival of mail), they may even all be visible at once:



(Larger menus are discussed below.) Each time around the loop a function is selected and a menu of the points on the function where function breakpoints may be set is

presented:

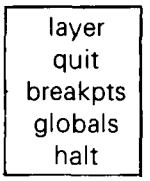


The '>' identifies the breakpoints currently set on the function.

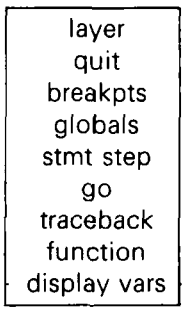
The problem is that while control flow in the host is in this loop the terminal cannot inform the host that a breakpoint has been encountered. The requests to the terminal are exclusively for menu selection and breakpoint table update. The host code is simplified by being dedicated to a single task, rather than using the more general WAIT_EVENT request. The price is that a breakpoint is not announced until the host returns to its main loop and uses the WAIT_EVENT request again. This particular instance of menus locking out breakpoints could be corrected relatively easily, but there are others, and not all could be corrected easily. It really reflects a basic flaw in the construction of the host process that should be corrected by redesign. The desired style of interaction calls for radical restructuring of the host.

The menus constructed by the host process in the code above must be transmitted to the terminal process. To receive and present them to the user, the terminal process has a fixed set of statically allocated data structures. Requests from the host write strings into these structures and bind menus to mouse buttons. Dynamic structures are not used because the terminal process uses no space allocated dynamically from mpxterm. C memory allocators do not garbage collect; programs must free memory explicitly. Indirection through dangling references to de-allocated memory is therefore a common bug in C programs and joff should depend as little as possible on the correct behaviour of other processes in the terminal.

A layer of software in the host process records the state of each of the menu structures in the terminal and suppresses redundant requests. Using this, the host can build menus from scratch with simple code. For example, if the subject is RUNNING, the menu looks like:



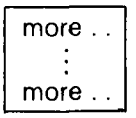
If the subject is BREAKPOINT_TRAPPED in a function display, the menu changes to:



Every time around the main loop, the host naïvely regenerates this entire menu based

on the state of the subject, but only the text of the changed menu items need be transmitted.

More complicated menus are implemented by another abstraction in the host that handles arbitrarily large menus. This is used by the breakpoint function above, since most programs have many functions. Menus too large to fit into the fixed structures in the terminal are broken into segments that will fit, with a special item at the top and/or bottom to move to the adjoining segments:



This is arranged so that the application level in the host need not know how big the menu is, where it needs to be broken or what text must be transmitted. This works well enough for logical menus of up to about 40 items, with segments showing 16 items at a time. However, chaining through menus of more than 40 items in segments of 16 is tedious. So the standard menu primitive in the terminal process was modified to scroll quickly through menus that are too large to be popped up, using a 'scroll bar' beside the menu to control which 12 items are visible. Terminal process memory is still statically allocated, and menus must still be broken into segments, but the segment size is now 64 items. (Each segment of 64 is scrolled with 12 of its items visible.) The combination of segments and scrolling handles menus of up to about 190 items acceptably as three segments.

EXPRESSION EVALUATION

joff recognizes and evaluates a subset of C expressions that includes fixed point arithmetic, subscripting, indirection, field selection, assignment and function application. Expressions entered from the keyboard are transmitted to the host and parsed. The parser was generated by extracting a subset of the compiler's yacc grammar. This parser yields an abstract syntax tree, but performs no semantic analysis. A recursive function is applied to the root of the expression to analyse and evaluate it on a bottom-up pass. Each call annotates the tree with values and semantic attributes, returning a success/failure indicator as its result. Because of the bottom-up evaluation, the evaluator does not know whether the calling level requires the *value* or the *address* of the object denoted by the subexpression to which it is applied. It therefore places both in the tree. Addresses can be computed within the host, but values must be fetched through the debugger's process in the terminal. The redundant fetching of unnecessary values is not a performance problem. Experience suggests that most expressions entered at the keyboard are small.

All arithmetic is performed on the host. As a simplification, the expression evaluator always uses 32-bit signed arithmetic, even though C semantics on the MC68000 target call for 16 and 32-bit signed and unsigned arithmetic. The intention was to correct this bug when users started to complain. Since there have been no complaints, that is how it remains. There have, however, been requests for operators not included originally. Unary minus had been omitted; that complaint came from someone applying a negative subscript to a pointer into the middle of an array:

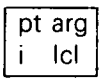
pointer[−3]

The other request was for the operator sizeof, which yields the size of an object in bytes, for computing the index of an array element, given the address of the element and the base of the array:

(*interior_address*−*base_address*)/(sizeof *array_element*)

To evaluate a function, the host lays down an argument vector in a fixed region of the debugger's terminal process and then instructs the terminal process to call the function, passing the vector as the argument. The function is therefore executed with the debugger's stack, rather than that of the subject process, and available stack space is independent of the state of the subject process. But if the function makes a system call to mpxterm, it will appear to come from the debugger process, possibly with disastrous results. Other side effects may also be undesirable; it is the user's responsibility to know about a function before calling it. It is always safe to call true functions: co-ordinate transformers, hash functions, table searches, checksums, etc. If the subject process is executing, it is suspended by joff during the call, so that its breakpoints may be lifted and the debugger's invocation of the function will complete. When the function returns, the host gathers the result and continues with the expression it is evaluating. Most results are returned in a known register. Records are returned in a fixed region, except that very short records* are returned through registers, which leads to a conflict between the host's ill-fitting models of registers and record addressing on a target machine with the opposite byte ordering.

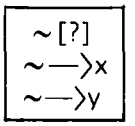
In fact, most expressions evaluated are those of a further subset that are built from menu-driven interaction. Pointers and aggregate variables can be used as the basis for building expressions, in which the user is repeatedly offered menus of applicable operators and functions. This begins with the selection of a variable from a menu, say of the variables in a function with an argument, pt, and a local variable, i:



Suppose pt is selected, a pointer to a record of type Point:

```
struct Point {
        int     x;
        int     y;
};
struct Point *pt;
```

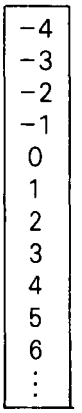
A function on the host is called with the degenerate expression pt. The expression is evaluated. Its type determines which operators are applicable and they are presented as a menu of expressions, in which the expression-in-hand, pt, is represented by a tilde:



Selecting ~—>x or ~—>y causes the function to be re-invoked recursively with the expression pt—>x or pt—>y, respectively. Both of these are the simple case where the

* The graphics package represents a point by a pair of 16-bit integers.

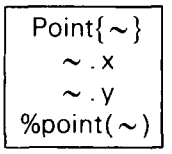
expression evaluates to a scalar; the value is printed and the function returns to the level above, where the same menu is presented again. Selecting ~[?] produces a scrolling menu of subscripts (with a somewhat arbitrary range of −4...128):



Selecting one of these subscripts re-invokes the function with a new expression such as pt[2], in which case the expression is a record, not a pointer, and the selections in the next menu are:

where the tilde now represents pt[2].

In fact, the menu above also contains special entries which are not normal expressions:



Point{~} prints the whole record, say:

{x = 226, y = 413}

%point(~) calls a graphics function in the terminal to display a cross-hair at that point on the screen. %point(~) is included in the menu because joff know that struct Point is one of the primitive graphics data structures defined for all Blit programs. Other functions display rectangles, textures and bitmaps. The 1K of code this takes in the terminal is well spent. Interpreting graphics data structures symbolically and numerically is very time consuming, in the cases where it can be done at all. A few functions makes it possible to interpret them instantly. 'Off-screen' bitmaps are a good example; the programmer can see the effects of graphics operations on images as they are constructed, before they are copied to the region of memory that is mapped to the display.

This general mechanism supports chasing through arrays and through arbitrary linked data structures. In addition to the built-in operations at each step, monadic functions in the program, whose argument is the type of the expression-in-hand, are sought from the symbol table and included in the menu. Again, it is the responsibility of the user to know when it is safe to invoke a function.

The inefficient re-evaluation of the expressions from scratch at each step might be eliminated, but the performance is acceptable. Typically, these expressions are larger than keyboard expressions, because it is easy to chase down arbitrary chains of pointers. Only expressions involving multiple function calls are a little slow. A real difficulty is that control flow in the host is tied up in a recursive function. First, this locks the user into following a single thread through a data structure. It is impossible (with a single instance of joff) to compare two linked lists by chasing down both of them in parallel. Secondly, we see again a problem that was mentioned above: control in the host is locked up and events in the subject will not be handled until the host returns to its main loop.

CONCLUSION

Experience with joff confirms that the asynchronous debugger and the menu-based user interface are successful. The implementation has worked well. The debugger is appropriately divided into two processes, though the best division of labour was only found by trial and error in several parts of the program. The debugger is bound to the subject dynamically, after which the two continue to execute asynchronously. Given operating system support, this adds relatively little to the implementation effort for considerably better assistance to the programmer trying to understand programs as they execute. The menu-driven user interface suffices for most commands, including pointer chasing, despite its introduction as an afterthought. But the full potential improvement over the keyboard interface of a conventional debugger has not been realized, by any means. Graphics data are displayed graphically. For very little implementation effort, the yield is enormous on graphics programs, especially those manipulating off-screen images.

Some difficulties in the implementation and inadequacies in the user interface leave room for improvements. A hierarchical symbol table is needed to simplify the debugger. A debugger's needs are very similar to those of a compiler; the structure of the symbol table deserves careful attention. With more care, asynchrony could be further exploited. The user's activity should never prevent the debugger from reporting events in the subject.

Operating system support, symbol tables, user interface and asynchrony are being further investigated in ongoing work towards a debugger for host processes.

ACKNOWLEDGEMENTS

Steve Johnson, Rob Pike, Dennis Ritchie and Peter Weinberger made modifications to every component of the Blit support software to accommodate joff. The complaints and suggestions of Luca Cardelli, Andrew Hume, Mark Manasse, Rob Pike, Dennis Ritchie and Tom Szymanski about early versions resulted in many improvements to joff.

REFERENCES

1. R. Pike, B. Locanthi and J. Reiser, 'Hardware/software trade-offs for bitmap graphics on the blit', *Software—Practice and Experience*, **15**, 131–152 (1985).
2. R. Pike, 'The Blit: a multiplexed bitmap terminal', *AT & T Bell Laboratories Technical Journal, Computing Science and Systems* (October 1984).

3. T. A. Cargill, 'The Blit debugger', *Journal of Systems and Software*, **3**, 277–284 (1983).
4. S. C. Johnson, 'Yacc: yet another compiler compiler', *Bell Laboratories Computing Science Technical Report 32*, 1975.
5. S. C. Johnson, 'A tour through the portable C compiler', *Unix Programmer's Manual 2*, 1979.
6. H. P. Katseff, 'Sdb: a symbolic debugger', *Unix Programmer's Manual 4.1 BSD 2c*, University of California, Berkeley, 1980.
7. B. Beander, 'VAX DEBUG: an interactive, symbolic, multilingual debugger', *Proceedings of Symposium on High Level Debugging*, Asilomar, California, 1983.
8. J. R. Cardell, 'Multilingual debugging with the SWAT high-level debugger', *Proceedings of Symposium on High Level Debugging*, Asilomar, California, 1983.