Computing Science Technical Report No. 102:

The C Language Calling Sequence

S. C. Johnson
D. M. Ritchie

Bell Laboratories
September, 1981

Introduction

A calling sequence is the conventional sequence of instructions that call and return from a procedure. Because programs tend to make many procedure calls, a compiler should ensure that this sequence is as economical as possible.

A calling sequence is influenced by the semantics of the language and by the register layout, addressing modes, instruction set. and existing conventions of the target machine. It is closely connected with allocation of storage for variables local to procedures.

This document sets forth the issues involved in designing a calling sequence for the C language, and discusses experience with various environments. The system implementer and hardware designer might find hints about the efficient support of C; the ordinary user might gain some sense of how hard it can be to support even a simple language on today's hardware. The first several sections discuss the requirements of a procedure call and return forced by the C language and conventions, including the support of recursion and variable-sized argument lists, and methods of returning values of various types (which might be ignored by the calling procedure). Then some sample designs are presented. The last sections discuss various topics, such as growing the stack, supporting structure valued procedures, and the library routines *setjmp* and *longjmp* An appendix presents the Interdata 8/32 calling sequence in detail.

The Basic Issues

What features must be provided by the call of a C procedure? Procedures may call themselves recursively without explicit declaration. Because procedures can pass the addresses of local variables and arguments to subprocedures, keeping local variables in static cells is unacceptably complex. Thus, C run-time environments are built around a stack that contains procedure arguments and local variables. The maximum amount of stack space needed by a given program is data dependent and difficult to estimate in advance. Thus, it is convenient to permit the stack to start small and grow as needed. When this cannot be done automatically, stack overflow should at least be clearly signaled and stack size easily increased.

The C language definition does not contemplate the question of procedures whose formal and actual argument lists disagree in number or type. Nevertheless, as a practical matter, the issue must be dealt with, especially for the *printf* function, which is routinely called with a variable number of arguments of varying types. In general, only the calling procedure knows how many arguments are passed, while the called procedure does not. On the other hand, only the called procedure knows how many automatic variables and temporary locations it will need, while the calling procedure does not. This issue of handling variable-length argument lists can dominate the design of the calling sequence on some machines; often the hardware support for procedure calls assumes that the size of the argument list is known to the called procedure at compile time. This assumption is correct for many languages (e.g., Pascal), but in practice not for C. (Interestingly, the assumption is also correct in theory for Fortran, but again not always in practice. Many Fortran programs incorrectly call one entry point in a subroutine to initialize a large argument list, then another that assumes that the old arguments are still valid. Were it not for the ubiquitous printf, implementors of C would be as justified as many implementors of Fortran in adhering to published standard rather than custom.)

The Basic Call/Return Process

The following things happen during a C call and return:

1. The arguments to the call are evaluated and put in an agreed place.
2. The return address is saved on the stack.
3. Control passes to the called procedure.
4. The bookkeeping registers and register variables of the calling procedure are saved so that their values can be restored on return.
5. The called procedure obtains stack space for its local variables, and temporaries.
6. The bookkeeping registers for the called procedure are appropriately initialized. By now, the called procedure must be able to find the argument values.
7. The body of the called procedure is executed.
8. The returned value, if any, is put in a safe place while the stack space is freed, the calling procedure's register variables and bookkeeping registers are restored, and the return address is obtained and transferred to.

Some machine architectures will make certain of these jobs trivial, and make others difficult. These tradeoffs will be the subject of the next several sections.

Dividing the Work

Parts of the job of calling and returning can be done either by the calling or the called procedure. This section discusses some of the tradeoffs.

As mentioned above, some of the status of the calling procedure must be saved at the call and restored on return. This status includes the return address, information that allows arguments and local variables to be addressed (e.g. the stack pointer), and the values of any register variables. This information can be saved either by the calling procedure or the called procedure. If the calling procedure saves the information, it knows which registers contain information it will need again, and need not save the other registers. On the other hand, the called procedure need not save the registers that it will not change.

Program space considerations play a role in the decision. If the status is saved by the *called* procedure, the save/restore code is generated once per procedure; if the saving is done by the *calling* procedure, the code is generated once per call of the procedure. Because nearly all procedures are called at least once, and most more than once, considerable code space can be gained by saving the status in the called procedure. If status saving takes several instructions, code space might be further reduced (at the expense of time) by sharing the status saving code sequence in a specially-called subroutine (this is done on the PDP-11). Similarly, branching to a common return sequence can further shrink code space. If the save or return code is indeed common, the calling procedure's status information must be in a known place in the called procedure's environment. Of course, the space gained by sharing save and return sequences must be balanced against the time required to branch to and from the common code.

Passing Arguments

The calling procedure must assume responsibility for passing the arguments, since it alone knows how many there are. However, these arguments must eventually be made part of the called procedure's environment.

Arguments can be passed in several ways. The calling procedure might establish a piece of memory (in effect, a structure) containing the arguments, and pass its address. This requires an additional bookkeeping register, and so increases the amount of status information to be saved, and slows down the call.

Most often. the caller knows the location of the called procedure's stack frame. Then the calling procedure can store the arguments adjacent to (or part of) the new stack frame, where the called procedure can find them. In practice, this requires that the stack be allocated from contiguous area of memory.

Because most calls pass only a few arguments, it is tempting to pass the first arguments in registers and the rest in some other way. Although this approach might be effective, it requires an exceptionally regular register architecture, and has not yet proved practical.*

Good Strategy

With enough control of the operating system environment. it may be possible to place the stack in an area that can be grown contiguously in one direction or the other. For the moment, assume that the stack is growing backwards (from high to low memory), and that it is possible to index both positively and negatively from registers (in particular, the stack pointer); other cases will he discussed later.

One simple layout for the stack frame uses only a single bookkeeping register, or stack frame pointer (fp). The stack frame looks like:

```
          . . .
          incoming arg3
          incoming arg2
          incoming arg1
          saved status information
fp´
          automatic storage area
          temporary storage,
            including space for outgoing arguments
```

where high memory addresses are at the top of the page, and the stack grows downward. The amount of status information saved is known to the called procedure; if this is *x* addressing units, then the arguments can be addressed as *x(fp), x+4(fp),* etc. (the constants reflect the sizes of the arguments in the addressing units of the hardware). Automatic variables can be accessed as *-4(fp), -8(fp)* etc. The saved status information is always addressable at *0(fp),* in the same place in every procedure, facilitating debugging and a shared save and return sequence.

The calling procedure first stores the arguments in order at the end of its stack area, and transfers control to the called procedure, passing the address of the arguments in a scratch register. The called procedure then saves the status, including the old value of *fp,* and then uses the passed argument pointer to establish the new value of *fp.* To return, the procedure value being returned is placed into a specific scratch register, the status of the calling procedure (including the old fp value) is reestablished, and control returns to the calling procedure.

A second bookkeeping register, a *stack pointer (sp)* that points to the end of the stack, may be useful. If the outgoing arguments are built at the top of the stack, the *sp* already points to them, and no scratch argument register need be computed. On many machines, a push instruction is available that adds outgoing arguments to the end of the stack (the *sp* must be adjusted after a call to undo the effect of these pushes.) To save the status, the called procedure saves the old *fp* on the stack. and copies the old *sp* into the *fp.* To obtain stack space for automatics, etc., the *sp* is then set to the desired offset from the *fp.* This is the organization used by the C implementation on the PDP-11.

On the PDP-11, stack overflow check is done as part of the usual memory protection mechanism: if a reference to a stack location generates a protection violation, the operating system attempts to allocate more memory and extend the stack segment downwards. If more memory is available, the offending instruction is restarted. otherwise the process is aborted. This overhead is incurred only when needed, but the operating system must cooperate; on some models (e.g., the PDP 11/40) this restart represents a major software investment in the kernel. When hardware support is lacking, an explicit test for overflow is only about two additional instructions per call.

One advantage of this organization is that it makes efficient use of limited address space: the heap data area grows upwards toward a downward-growing stack. Unfortunately, both the user instruction set and the system-level memory protection and allocation strategies must support this: many machines do not

* Note added 2003: it did prove practical shortly thereafter.

support protection of backwards-growing segments, for example.

What can be done in environments where the stack is encouraged to grow upwards? Clearly, the above organization could simply be turned on its head, and the result would still be an efficient and effective organization. The only incompatibility would be that arguments would now form an array running backwards in memory! Nevertheless, the scheme is viable, provided the implementor accepts the cost of modifying all procedures that (like printf) depend on variable-length calling sequences. Alternatively, the argument array might be turned around to run forwards in memory: since the called procedure cannot know (unless told) the address of the first argument, this requires yet another bookkeeping register.

A Five-per-cent Digression

When *sp* is used to push arguments onto the stack, it must be readjusted after each call to throw away the arguments. However, if an extra word is left at the end of the stack, calls with only one argument need only move this argument top of the stack, and *sp* need not be adjusted after the call. Since many calls have one argument, this is attractive. Moreover, there are special short instructions on the PDP-11 that are used to bump the *sp* by two or four bytes; these instructions now apply to procedures with two and three arguments, where before they applied to procedures of one and two arguments. Two bytes are saved for every call with either one or three arguments; on the PDP-11, this amounts to about five per cent of the code space.

When using this technique special care must be taken to handle nested calls and calls with active expression temporaries.

Positive Offsets Only

The organization discussed above requires that it be possible to index both positively and negatively from *fp*. Many machines (notably the IBM 360/370 series) take all offsets from base registers to be positive; this means that additional work is needed to preserve the good features of the above organization.

In such a scheme, the stack must almost certainly run forward in memory, because addressing locations below the current frame pointer is hard. If the saved status is to be kept at a universally known location in the frame, it must be first. The inbound arguments must come next, just above the save area; then comes the local storage. The revised organization is as follows (high memory addresses are at the top, and the stack grows upwards):

```
        outgoing arguments
        next procedure's status save area
        automatic and temporary storage
        . . .
        incoming arg3
        incoming arg2
        incoming arg1
        saved status for current call
fp´
```

To do a call, the *calling* procedure establishes the arguments of the call at the end of its stack frame. sets a scratch register to point to the beginning of the next procedure's status save area, and transfers control. The *called* procedure then saves the status (including the *fp)* and, if necessary, ensures that there is enough stack space available to begin execution. It then sets *fp* to the scratch register pointing to its status save area, and begins. To return, the old status, including the return address, is loaded from the status save area, and a return branch is done; the *fp* can often be automatically restored with the rest of the calling procedure's status. To share the return code. the saved status must either always contain the same information, or must be self-describing.

Because the *called* procedure claims local storage at the end of the argument list, this scheme implies a bound on the size of the argument list. Thus procedures that accept variable-length argument lists must declare at least as many arguments as they will actually be called with.

Argument Pointers

There is increasing interest in coroutine environments for C programs. In such environments, it is inappropriate to speak of 'the' stack, since there may be many stacks, and, in particular, the arguments to a procedure may not be adjacent to the stack frame for that procedure. Even without coroutines, the peculiarities of the previously-discussed argument passing may be discomforting. If one is willing to maintain an extra argument pointer bookkeeping register, these problems go away.

In this scheme, a register *ap* points to the argument list; in this version, it will also supply a status-save area. The called procedure still does the save. The stack frame layout looks like:

sp´
    outgoing argument space
    next procedure's save area
    automatics and temporaries
fp´
    incoming arg3
    incoming arg2
    incoming arg1
    status save area
ap´

Once again, high memory addresses are at the top, and the stack grows upward. The called procedure can address its arguments without knowing how many there are, or declaring a maximum number.

The overhead is modest. Another bookkeeping register is needed to hold *ap,* and this register must be saved and restored across calls; before a call, the new argument pointer must be established, and then, in the called procedure, the new argument pointer must be copied into ap. These operations can be done quickly on many machines, and this organization is attractive on machines with enough registers, such as the Interdata 8/32 (see the appendix) and the VAX.

Register Allocation

C compilers use registers in three different ways:

1. Bookkeeping: stack pointer, etc.
2. Register Variables
3. Scratch Registers

Register variables are usually allocated as the result of an explicit **register** keyword in the source, although experimental compilers that make this allocation automatically are under construction. Because register variables are used to store variables, they must be preserved across procedure calls. Scratch registers, on the other hand, are assumed to be overwritten by called procedures; they are used for expression evaluation, and are available for passing information to called procedures (e.g., the location of the arguments) or to receive the values returned by the called procedures.

Part of the engineering of the environment involves deciding how many registers should be devoted to each of these purposes. Few scratch registers are needed for expression evaluation; two to four is adequate on almost all machines. Although one might think that the more register variables the better, if all potential register variables are saved at each call (and this is attractive for other reasons) time is lost saving and restoring unused registers. Some operations (e.g., block copy) may require particular registers to be free; such registers are usually best made scratch registers. Finally, which particular registers are allocated to scratch and register variables is often determined by the properties of the 'save multiple' instruction, if any; it may be desirable to have the register variables adjacent to the bookkeeping registers so they can all be saved by one instruction.

Alignments

Many machines require alignment of some data on specified addressing units. For example, on the Interdata 8/32, integers must be placed on 4 byte boundaries, and doubles on 8 byte boundaries. Such a restriction forces the frame pointer and the frame size to be a multiple of 8 bytes. Alignment requirements may also generate holes in an argument list and force these lists to begin in aligned locations.

The *nargs* function, and related topics

Early implementations of C on the PDP-11 supplied a *nargs* function that purported to return the number of actual arguments with which a procedure was called, to facilitate processing of variable-length argument lists. Many difficulties ensued. First, the function never did work: it actually returned the number of words in the argument list; iB doubles take four words, *longs* take two, and structures take any number. Second, the advent of separated instruction and data space and of optimization made the implementation problematical. Third and most important, the difficulty of specifying, either formally or pragmatically, the meaning of calls in which formal and actual arguments disagree became painful enough to discourage the practice.

Nevertheless, variable-length argument lists must be dealt with in practice; *printf* has already been mentioned. The desired behavior is:

1. If the calling procedure supplies fewer arguments than are declared by the called procedure, and if the called procedure does not (in its execution) access the unsupplied arguments, no harm should result.
2. If a calling procedure supplies more arguments than are declared by the called procedure, no harm results. Moreover, if the called procedure has knowledge of the types of these extra arguments, there should be some mechanism by which it can access them.

These rules place the burden of handling variable argument lists on the called procedure; such procedures cannot be portable, but they must be possible.

Unwinding

It is desirable that the structure of an active stack be interpretable without reference to the code that created it.

First, consider a debugger that wishes to produce a stack trace, including the values of register variables in each active procedure. Unless care is taken to make the save area of each frame either self-describing or uniform, the debugger's job may be difficult or impossible. In particular, if the number of registers saved at each call varies, recovering the register variables will be hard, because they may be stored at a considerable distance from the frame that last uses them.

Similarly, the implementation of *longjmp* requires unwinding the stack to recover the values of the registers in the procedure that called *setjmp.* The task is hard enough that several systems botch the job; the values restored are not those at the time of the call leading to the call of *longjmp,* but instead those at the time of the call to *setjmp* (which is not correct).*

The problems posed by *setjmp/longjmp* appear in other contexts. For example, the multiprogramming primitives in the Unix kernel, *save* and *resume,* do similar jobs, and such primitives also appear in other coroutine and multiprogramming systems.

As was mentioned, unwinding is guaranteed to be easy if the return sequence is expressible as a (parameterless) instruction or sequence of instructions. It is almost always possible if one is willing to do enough work. It is good to have an algorithm in hand before committing oneself to a given design.

---

* Note added 2003: The ANSI committee in the 1989 standard struggled over this issue, and coopted the *volatile* keyword to help deal with it.

Interrupts

In the Unix system, interrupts (in the kernel) and signals (in user processes) are turned into procedure calls. These procedures require stack space, and it is convenient to use the same stack as the ordinary procedures. This implies that it must be possible to recognize the last location used in the current stack frame, to know where to begin the stack frame for the interrupt procedures. A register pointing to the end of the stack frame (call it *sp)* may require one additional instruction to set up the register on entry to the call, and would be an extra register of status information to be saved.

The order of events in a call must be chosen carefully so that an interrupt does not use a piece of the stack area still needed by the current procedure. For example, if the status save area is to be placed in the new stack frame, *sp* must be adjusted before the status is saved, in order that an interrupt not use the space too soon. Similarly, in returning, the remainder of the status must be restored before *sp* can be changed. In general, the registers that describe the environment must always be valid whenever an interrupt might happen.

One alternative approach is to allocate a stack for interrupts from some other area of memory. This is ugly, but might be better than consuming another register on some machines. (But make sure that *longjmp* still works!) Alternatively, the signal and interrupt procedures might leave extra space for a possible status save in progress, this is dangerous but workable. A similar issue arises when swapping processes out *(sp* marks the end of the stack area to swap) and in reentrant procedures (such as some interrupt handlers), and must be handled similarly in all cases.

Functions Returning Structures

Functions may return structures and unions. Although objects of other types usually fit nicely in a designated return register, these do not; where should the returned value be placed? If the value is left in the stack space of the called procedure, it becomes unprotected after the return and before it is copied; a signal, interrupt, or swap at the wrong time would be a disaster. If the value is left in a static area, it is more protected, but the procedure is no longer reentrant.

The proper solution is to allocate space for the return value in the calling procedure's stack, and communicate the address of this space to the called procedure; before returning, the value is copied into this space. The called procedure might be able to find this space by looking at an additional argument or a scratch register passed by the calling procedure, or (when the stack is contiguous) by mutual agreement about the area where the result should go.

One attractive scheme is for the called procedure to copy the value over its incoming arguments; this region is in the calling procedure's stack area, and is certain to be known to both procedures. If this is done, the calling procedure must supply enough space for the returned value, and must not reuse the argument area to call another procedure before the value is copied. The called procedure must ensure, when an incoming argument is being returned, that it is copied properly into the return area.

The PDP-11 uses a static area to copy the structure return value (and thus gives up reentrancy), so compiler writers who do likewise can at least claim a role model. Fortunately (or perhaps consequently), reentrant procedures returning structures are uncommon.

It is tempting to suggest that procedures returning small structures, say one or two words, should return those values in registers. Unfortunately, code sequences arise as a result of this optimization that do not arise at any other time. It hardly seems worth the trouble.

Stack Machines

Stack machine architectures are favored by some manufacturers because they permit compact code, are simple intellectually, and 'because they make compilers easier.' It has not been easy to put C on stack machines, however.

The most serious problems have been with the call and return sequences. In the call, there is often no provision for variable-length argument lists. Most stack machines use the stack to pass arguments to a procedure (there is little choice, after all!); and possess a **return** opcode that frees the stack space used by the *called* procedure, restores status, *pops the arguments,* pushes the return value onto the top of the stack, and transfers back. As the italics suggest, the *called* procedure does not know how many arguments were passed, and may thus be unable to use the return opcode (in at least one machine, the argument size has to be known at compile time). 'Hand coding' such a return sequence is long and slow, and there is no good place to save the returned value while this bookkeeping is being done.

Even when the arguments are well behaved, other problems arise returning the values, especially when the calling procedure and called procedure disagree on the type or disposition of the value. If the types disagree. it is illegal C; the consequences (misalignment of the stack because the actual and expected return value differ in size) are more bizarre than with machines that return values in registers. More seriously, if the *calling* procedure wishes to ignore a value returned by the *called* procedure, the calling procedure must know how big the value is. This requires a strict type match: moreover, it may require procedures that compute no value to return to return one anyway. so it can be thrown away! The **void** type would be a big help here if consistently used.

Finally, some stack machines have never contemplated the return of structures. so many of the problems of the previous section apply.

Call Instructions

On some machines, there are instructions designed to aid in compiling procedure calls and returns. Some of these instructions cannot deal with variable-length argument lists, and thus cannot be used. Others, far from being limited, are so excessively general that they take a long time to do the call. The VAX 11/780 instructions do exactly what is wanted, little more and no less, yet the *calls-ret* instruction pair takes about the same time as does the 16-instruction sequence that does the job on the slower PDP 11/70. It is always worthwhile to consider coding the calling sequence explicitly, especially if code space is not a critical resource. On some machines, brute force strategies have saved 30-40% in time over the built in instructions.

Microcode

Some machines have writable microstore, making it tempting to build special-purpose C call, save, and return instructions. This might work well, but several things are worth noting.

1. Writable microstore is a resource. If it is used for calls it may become unavailable for other uses. In a C-language operating system, the microstore must be loaded before execution of C programs can begin.
2. Manufacturers often provide only limited access to writable microstore. For example, on both the VAX and Interdata machines, the usual operand decode hardware is disabled. This means that to distinguish between various operand types, time-consuming conditional tests must be done in microcode.
3. The memory access time required to store relevant registers may dominate. Microcode speedup may not be significant.

On the Interdata, the writable microcode was tried; we found only a 10-15% speedup in the call of a procedure with no arguments and no register variables, and less improvement for more complicated procedures. Because of point 1, it did not seem worth the bother.

The Real World

The calling sequences described above are used on the PDP-11, the Interdata 8/32, and many other machines. On both GCOS and IBM, however, the C compiler generates different calls; other forces outweighed the considerations discussed above. On both systems, it is inconvenient to grow stack space contiguously; more important, on both systems existing system calling conventions offered advantages in compatibility.

On GCOS, the C calling sequence is the same as that for Fortran; control is transferred, and the *addresses* of arguments are left in memory following the call. The *called* procedure obtains enough space for its stack frame, and then copies the arguments into the proper place on this stack frame. Efficiency is improved by the powerful indirect addressing capabilities of the Honeywell hardware, which allow references to automatics, arguments, and constants to be compiled directly into the argument lists without executing any code at all. The scheme permits the stack to be discontiguous, so additional space can be obtained from the operating system if needed. It is necessary to avoid copying nonexistent arguments (so it costs time to cater to variable argument lists). Because the *called* procedure does not always know the size of the argument to be copied, at least with floats and doubles, it must rely on the declared size--another *printf* bother. A machine-dependent procedure solves the problem by providing a list of the addresses of the arguments to the *called* procedure.

On the IBM/370, we chose to be compatible with a local IBM BLISS calling sequence. Since C and BLISS are similar languages, we could make use of the BLISS I/O library. In retrospect this was unwise, for the BLISS library has been of no use in other IBM environments, and some of the BLISS conventions make interfacing with standard OS calls difficult.

Were the IBM calling sequence to be reimplemented, it seems that the large address space and the availability of virtual memory on most IBM systems would argue for a contiguous stack organization similar to that described above.

If any general conclusion can be drawn, it is that the problem of implementing a C calling sequence when the operating system is not ideal depends on issues far removed from C itself. If the aim is to obtain Unix utilities with little effort, it may be worth spending more time in the call to keep the fine points compatible with other environments.

Effects on the Language

The decisions about the stack frame and calling convention are barely visible at the language level, but there are a couple of places where they can be discerned. Variable-length argument lists have already been much discussed. The other area is the order of evaluation of procedure arguments.

The C language specifies that the arguments to a procedure may be evaluated in any order; if a program depends on the order of evaluation of the arguments of a procedure, it is illegal C. Reality is not so kind, and one finds programs (especially those written by novices or former assembly language programmers) that have these dependencies in them. There seem to be three natural orders of evaluation; while this is properly an issue for the compiler-writer, it will be touched on here.

If the stack grows backwards, and there is a **push** instruction that can be used to put the arguments on end of the stack, the natural order to evaluate procedure arguments is *right to left,* so that the arguments can be found on the (backwards growing) stack in increasing order. When the stack grows forward, the natural order is *left to right.*

When the outgoing arguments are built in a special area at the end of the stack frame, it is essential that all arguments containing procedure calls be evaluated before the arguments to the outer procedure are placed in the argument region. The natural order is to evaluate those arguments containing procedures first, and then the others in some order. We might term this the *inside-out* order. These three orderings are the most common, but others are possible. For example, were the first several arguments passed in registers, this would almost certainly affect the argument evaluation order. Tighter specification of the order of evaluation in C would cause significant inefficiencies with some of these stack-frame organizations, or, alternatively, considerably more complexity in the compiler.

Summary

We have discussed the major issues involved in the construction of a C calling sequence, and have given a stack frame organizations that can be adapted to many different architectures. An Appendix discusses the calling sequence for the Interdata 8/32 C compiler.

Acknowledgements

We are grateful to Mike Lesk, who coauthored the earlier version of this memorandum, and Brian Kernighan, Rob Pike, Dave Ditzel, Bart Locanthi, and an anonymous internal referee who commented on this version.

Appendix: the Interdata 8/32

The Interdata 8/32 is similar to the IBM 370 series machines, with the significant exception that most instructions permit a 'long address' mode capable of addressing any element in memory without using a base register. Thus, it is not necessary to have many base registers dedicated to establishing addressability. although base register references to data are faster and smaller than the equivalent longer forms.

The offsets from base registers are unsigned 14 bit numbers, and the memory protection does not allow segments to grow backwards, so the stack grows forward in memory.

To establish a baseline, first consider the fastest calling sequence that is marginally acceptable: this is one with only a single frame pointer register, and does not support interrupts or debugging stack traces. Throughout the following, let V be the number of register variables that are saved in the calling sequence. In the fastest calling sequence, a call looks like

```
la      nfp,offset(fp)
bal     link,procedure
```

where *fp* is the frame pointer, *nfp* the new frame pointer, and *link* the linkage register. The called procedure need merely do

```
stm     xx,0(nfp)
lr      fp,nfp
```

where *xx* is the first register to be saved. (The instruction saves registers starting at its argument through register 15). The return sequence is

```
lm      xx,0(fp)
br      0(link)
```

The registers are assigned so that the *lm* and *stm* cause the loading and storing of the V register variables, the link register, and the old *fp*. An approximate timing is

$$13.85 + 1.55V$$

microseconds. Thus, with 3 register variables saved, the cost would be 18.50 microseconds. As pointed out in the body of the text, the handling of interrupts makes it attractive to have a pointer to the end of the current stack frame. This register must be saved and restored (1.55 microseconds) and reestablished for each call (1.12 microseconds); moreover, in order that the registers not be saved beyond sp it must be copied (.4 microseconds). This gives a cost of

$$16.92 + 1.55V$$

With three register variables, this gives a time of 21.57 microseconds. The unpleasant parts of this calling organization are the need to declare more arguments than would be passed to a procedure, and the costlier implementation of coroutines. If another register is dedicated to hold an argument pointer, this register would have to be saved and restored across calls (1.55) and, in addition, the frame pointer would have to be established (.4) in each new procedure. On the other hand, *sp* would no longer have to be copied in the called procedure (saving .4 microseconds). Thus, this ornate calling sequence takes

$$18.47 + 1.55V$$

microseconds. With three register variables, this time is 23.12 microseconds; this is roughly 25% longer than the stripped down call, and less than 10% slower than the call with *sp* alone. Moreover, the Interdata, with 16 registers, is not short of registers. Thus, this calling sequence was adopted.

The stack frame organization looks like:

sp´
    new argument space
    new save area
    temporaries
    automatics
fp´
    . . .
    arg3
    arg2
    arg1
    save R15
    save R14
    . . .
    save R7
    save R6
ap´

Here, high addresses are at the top of the page, and the stack grows upwards.

To call, the arguments are placed at the end of the current stack frame. Then the calling procedure executes

```
la	nap,offset(fp)
bal	link,procedure
```

The called procedure does

```
lm	xx,yy(nap)
lr	ap,nap
lr	fp,sp
la	sp,offset(fp)
```

On return, the calling procedure executes:

```
lm	xx,yy(ap)
br	0(link)
```

Here, xx and yy are functions of V that are chosen to make the registers saved always lie at the end of the save area.

The following organization permits this calling sequence within the constraints of the *lm* and *stm* instructions:

0	scratch register
1	base register to the data area (never changed)
2	scratch register: used for procedure returns
3	scratch register
4	scratch register
5	scratch register: holds nap on calls
6-11	register variables
12	*ap*
13	scratch register: holds link on calls
14	*sp*
15	*fp*

This organization allows up to 6 register variables.

Note that the number of registers saved in a given procedure cannot be determined from the stack frame. This has caused problems; a reimplementation today would probably use fewer register variables, and save all possible ones.

Measurements and Empiricism

For several reasons, the calling sequence implemented on the Interdata is slower than claimed here. For one thing, a hardware design defect forces us to check explicitly for stack overflow; it is enough for us to generate the instruction

```
        l       r0,0(sp)
```

at the end of the *called* procedure's prologue, to generating a fault if the stack has overflowed. Another empirical problem is that, when the code is unoptimized, certain numbers (for example, the xx and offset values above) are known only at the end of the procedure. In the absence of optimization, the assembler may use longer forms for these instructions than are necessary. Finally, for various unpleasant reasons all return sequences have an extra branch in them; some have two. Thus, the measured time to do a call of a null procedure is longer than expected; it closely fits the function

$$24. + 2.3\ V$$

microseconds, or almost 25% slower than calculated.