PROGRAMMING DEPT. MEMO #130-003-005-00

digital INTEROFFICE MEMORANDUM

SUBJECT: PDP-11 Subprogram Calling Sequence Standards

DATE: January 29, 1970

TO: PDP-11 List "C"

FROM: H. Shepardson

LOCATION:

LOCATION:

The attached paper gives proposed standards for subprogram calling sequences for PDP-11 software along with some justification for each proposal. Your comments are invited. Feel free to raise questions or points for discussion with me at your convenience.

Attachment: "PDP-11 Subprogram Calling Sequence Standards"

PDP-11 SUBPROGRAM CALLING SEQUENCE STANDARDS

The purpose of this paper is to present and discuss several different types of subroutine calling sequences. The motivation for such a presentation is the apparent fruitlessness of several meetings which were held to define standard FORTRAN subroutine calling sequences for the PDP-11. It should be noted that the definition of FORTRAN calling sequences is critical since all PDP-11 software should follow these conventions as much as possible to facilitate maximum interchangability and to avoid future embarrassment (such as suddenly finding that system diagnostic dumps are not callable from FORTRAN).

The handling of re-entrant and non-reentrant subroutines and calling programs is the problem which to date has not yielded to an agreeable solution. However, it has been agreed that any solution should satisfy at least the following criteria:

1. Any subprogram can be called in either a re-entrant or non-reentrant manner.
2. The calling sequence must be "short" (at least in the non-reentrant case).
3. Non-reentrant code must not be penalized by the possibility of re-entrant code.
4. If extra code is necessary, it should be placed in the subroutine rather than the calling program as much as possible.

Several proposed solutions will now be presented and discussed. To facilitate an accurate evaluation of the methods presented, coding examples of calling sequences and subroutines will be given. Argument passing from level to level will also be shown. The discussion is summarized in clearer form and a recommendation is made at the end of the paper.

It is hoped that by making a critical examination and evaluation of several types of calling sequences, one type will appear to be more appealing than the others.

No Re-Entrant Calls

Since the JSR instruction as implemented on the PDP-11 provides very nicely for the coding of re-entrant subroutines, but does not provide as well for the coding of re-entrant calling programs, the easiest solution is to restrict re-entrant FORTRAN programs to one level only, i.e., do not allow subroutine calls from re-entrant FORTRAN programs. This method apparently allows the shortest possible subroutine calling sequences, calls are unaffected by re-entrant considerations, and all extra coding for re-entrancy is placed in the re-entrant subroutine.

This method has the disadvantage that a considerable restriction is placed on FORTRAN, and even simple library routines such as SQRT or SIN are not available to re-entrant programs. Also, by circumventing the real problem, no universal model for re-entrant calling sequences will be defined. Since such calling sequences will surely be required by monitors, etc., individual programmers would each "do it their own way" and a real communications hodge-podge would result, although overall length of code would probably be the shortest possible since each coder would take advantage of his particular situation.

The calling sequence would appear as follows (for three arguments):

```
JSR      R5, SUB1        ; NON-REENTRANT CALL
.WORD  A
.WORD  B
.WORD  C
```

The called subroutine can handle scalar arguments in two ways; either by (1) storing them in the subroutine for quick reference (and hence always restoring the final value back into the calling program) or (2) making all references to the scalar through the calling sequence. Both are shown below (where A and B are scalars and C is an array and all arguments are passed on to the next level).

Case (1) SAVE/RESTORE

```
SUB1:     MOV @(R5)+,T1       ; MOVE A AND B
          MOV @(R5)+,T2       ; INTO SUBPROGRAM
          .
          .
          .
          MOV (R5)+,.+12      ; ARRAY ADDRESS
          JSR R5,SUB2
          .WORD T1            ; A
          .WORD T2            ; B
          .WORD Ø             ; C
          .
          .
          .
          MOV T1,@-4(R5)      ; RETURN
          MOV T2,@-2(R5)      ; ARGUMENTS
          RTS R5
```

Case (2) REFERENCE THROUGH CALL SEQUENCE

```
SUB1:     .
          .
          .
          MOV (R5)+,.+16      ; MOVE ADDRESSES
          MOV (R5)+,.+14      ; TO CALLING
          MOV (R5)+,.+12      ; SEQUENCE
          JSR R5, SUB2
          .WORD Ø             ; A
          .WORD Ø             ; B
          .WORD Ø             ; C
          .
          .
          .
          RTS R5
```

Note that SUB1 would not be re-entrant but SUB2 could be.

Note also that if there were a sufficient number of arguments the total code could be lessened if the argument fetching were done in a subroutine.

All Calls Re-Entrant

The direct opposite of the previous method would be to make all subroutine calls re-entrant so that all subroutines would be potentially re-entrant. Probably the best method of accomplishing this is to move all argument addresses to the stack immediately prior to the JSR. This method provides uniform subroutine calls at a cost of two words (the MOV) per argument. The burden on the subroutine would be lessened since argument addresses are already on the stack.

This method has the distinct disadvantages of requiring several locations of core storage for each call, and the extra locations are required for each call in the calling program.

The calling program would be coded as follows (for the same call as discussed earlier):

```
        MOV #C,-(SP)        ; PUSH ALL
        MOV #B,-(SP)        ; ARGUMENT
        MOV #A,-(SP)        ; ADDRESSES
        JSR R5,SUB1
```

The called routine would be

```
SUB1:   .
        .
        .
        MOV n(SP),-(SP)     ; PUSH CALLING
        MOV m(SP),-(SP)     ; ARGUMENTS ON
        MOV k(SP),-(SP)     ; STACK FOR NEXT CALL
        JSR R5,SUB2
        .
        .
        .
        MOV (SP),6(SP)      ; REMOVE CALL
        ADD #6,(SP)         ; ARGUMENTS FROM STACK
        RTS R5
```

Where n, m, and k depend on how the subroutine has altered the stack at the point of the instruction. For ease of reference the user could set R5 to point to the argument list on the stack upon entry to a subprogram. Note that SUB1 as well as (potentially) SUB2 are re-entrant.

Compromise (The Bell-Delagi Method)

In order to satisfy criterion 3 (non-reentrant - no penalty), it was proposed that separate calling sequences be used, one for re-entrant and one for the non-reentrant case. As proposed, the method of entry would be invisible to the subroutine. This proposal is as follows (assuming the call discussed above):

```
Non-reentrant       JSR R5,SUB1
                    .WORD Return Point
                    .WORD A,B,C

Re-entrant          MOV #C,-(SP)           ; PUSH ARGUMENTS
                    MOV #B,-(SP)
                    MOV #A,-(SP)
                    MOV SP,R4
                    MOV R5,-(SP)
                    MOV R4,R5              ; ARGUMENT POINTER
                    JSR PC,SUB1+2
```

The subroutine would have the following code to provide the invisibility:

```
SUB1:               MOV (R5)+,-(SP)        ; NORMAL ENTRY
                                           ; RE-ENTRANT ENTRY
                    .
                    .
                    .
                    MOV (SP)+,R5
                    RTS R5
```

Several versions and refinements of this method have been discussed but all appear to have the same basic problems: (1) Minimum of <u>three word</u> subroutine call, (2) Link loader must be capable of resolving SUB+2, (3) Maintaining re-entrancy through cascading calls appears messy, and (4) This method appears to be hard to document and understand.

This proposal has several advantages. It allows any routine to call any other routine, most of the non-productive code appears only in the re-entrant case (plus a small overhead in the called program), compatibility adds only one word (above the normal JSR) to each non-reentrant calling sequence, and the method of calling is invisible to the called routine.

The passing of arguments from level to level of calls in the non-reentrant case is similar to the non-reentrant method discussed earlier. This case of argument passing in the subroutine is as shown:

```
SUBl:           MOV (R5)+,-(SP)
                .
                .
                .
                MOV (R5)+,.+20          ; MOVE
                MOV (R5)+,.+16          ; IN
                MOV (R5)+,.+14          ; ARGUMENTS
                JSR R5,SUB2+2
                .WORD RTN
                .WORD Ø,Ø,Ø
                RTN:
                .
                .
                .
                MOV (SP)+,R5
                RTS R5
```

For the re-entrant case the subprogram and argument passing would be:

```
SUBl:           MOV (R5)+,-(SP)
                .
                .
                .
                MOV 4(R5),-(SP)         ; MOVE ARGUMENTS
                MOV 2(R5),-(SP)         ; TO STACK AND
                MOV (R5),-(SP)          ; MAINTAIN ORDER
                MOV SP,R4
                MOV R5,-(SP)
                MOV R4,R5               ; ARGUMENT POINTER
                JSR PC,SUB2+2
                .
                .
                .
                MOV (SP)+,R5
                RTS R5
```

TRAP

Another subroutine calling scheme (with many variations) is to discard the JSR instruction and make use of the TRAP instruction. The TRAP instruction can be used in conjunction with a Trap Vector generated by the Linkloader. This Trap Vector contains the addresses of the entry points of subroutines currently in core. The Linkloader also constructs an offset in the low order byte of the TRAP instruction. Hence a

```
    TRAP n
```

would be an effective

```
    JMP @Trap Vector + 2n
```

However, a monitor trap handling routine would be entered. This routine would amount to several instructions, but they would appear only once, and they could be executed with interrupts inhibited if necessary. This method allows a minimum one word calling sequence, and also, since the monitor is entered on each call, eventual implementation of features like scatter-loading would be facilitated. However, at least a minimal monitor would be required for all calls, and all calls would execute rather slowly because of the monitor intervention. Also, the JSR would be scrapped, thus disallowing its esthetic appeal as well as eliminating its addressing flexibility.

The TRAP instruction could be substituted for the JSR instruction in any of the above calling methods with the result that calling sequence lengths would be reduced by one word.

The following TRAP handler is presented to provide a good model for users who wish to implement a TRAP method for their own code.

```
          ; SUBROUTINE JUMP TRAP HANDLER
          ; THE ADDRESS OF TRAP34 MUST BE IN LOC 34
          ;
TRAP34:   MOV    @R6,2(R6)            ; THE CALLED ADDRESS
          SUB    #2,@R6               ; IS COMPUTED THRU
          MOV    @(R6)+,-(R6)         ; THE JUMP TABLE
          ASL    R6                   ; AND THE TRAP ARGUMENT
          ADD    #JTABLE-TRAP,@R6     ; THE STACK WILL
          MOV    @(R6)+,R7            ; APPEAR AS IF
                                      ; JSR R7,SUB
                                      ; WERE EXECUTED.
; THEN TRANSFER IS
; MADE TO THE
; SUBROUTINE
; RETURN IS BY
; RTS R7.
```

Serial Reusability

One alternative to re-entrancy will be briefly presented. This alternative is to allow any subroutine to run to logical conclusion before entering it again (Serially Reusable Routines). The monitor would maintain a priority queue of requests to use any serially reusable routine and whenever a routine becomes available the highest priority request would be allowed entry. All interrupts would be immediately serviced (if higher priority than the routine running) and if a routine which was "busy" was encountered a request would be queued and control returned to the interrupted routine.

This method would work best where different levels of interrupt required generally different routines to complete. This could be used in conjunction with "lowest level" re-entrant routines (i.e. SQRT) to speed servicing of high priority interrupts.

This scheme would be at its worst when several levels of priority had a large number of common subroutines and some interrupt servicing was highly time dependent.

Comparisons

After having discussed several types of calling sequences, we are now in a position to make comparisons among them. The following chart shows the calling sequence length (in words) for the various types of call sequences. The chart is divided into two major sections, one for "first level" calls (i.e. calls with no arguments passed from a prior call) and one for nth level calls (i.e. calls with some arguments passed from the next higher subprogram call). Each section is further subdivided to show the length of code required per argument, per subroutine call, etc.

To get any real, meaningful information from the data presented, one must decide the importance of arguments relative to calls, of passed arguments relative to arguments originating in the subroutine, etc. The alert reader will have already noticed that there are figures on the chart related to "weights". These weights are a measure of the relative importance of the column they are associated with and they are based on the following assumptions:

(1) There is an average of four subroutine calls per "program", where a program may be a main program or a subprogram.

(2) There is an average of two arguments per subroutine call: one a scalar and one an array.

(3) The relative importance of passed arguments and arguments originating at the level of the call was a matter of great debate which was finally resolved by actually counting subprogram calls in a large number of FORTRAN programs. The result of this count showed that for programs not making much use of COMMON, approximately 70% of all arguments in calls from the nth to the n + 1st level originated in the n - 1st level or before. The overall average including all types of programs showed that about 45% of arguments were "passed" arguments. It was decided that due to the large difference, both situations should be shown, and thus we have the two sets of "weights" on the chart.

The rightmost two columns on the chart show the weighted lengths of subroutine calls for each calling method. The weighted length for each row is computed by taking the sum of the products of the length as given in each column of the row being computed and the weight for its column. A column is computed for each of the weightings discussed above. The method which yields the shortest weighted length should then give the overall shortest length of calling sequence (Comparisons should be restricted to columns, i.e. no intercolumn comparisons are meaningful.).

It should be noted also that due to the addressing modes of the PDP-11 and the amount of time required to execute each, the relative overall execution time for the call methods involved should be roughly proportional to the relative weighted lengths. The reader can easily see for himself the effect of juggling the various weights.

CALLING SEQUENCE LENGTH COMPARISON CHART

		First Level Call Length			2nd Level - nth Level Call Length for "Passed" Arguments				WEIGHTED LENGTHS	
		# Words Overhead/ Call	Additional Overhead/ Argument	Register Usage	# Words Overhead/ Subroutine	Overhead/ Call	Additional Overhead/ Argument	Register Usage	70% passing	45% passing
METHODS	Non-Reentrant reference thru call sequence	2	1	any register	Ø	2	3	any	36	4Ø
	Non-Reentrant save/restore	2	1	any	6/ Scalar argument	2	Scalar 1 / Array 3	any	34	38
	All Calls Reentrant (MOVE Method)	2	2	any	4	2	2	any	34	4Ø
	Bell-Delagi non-reentrant	3	1	any	2	3	3	any	43	48
	Bell-Delagi re-entrant	5	2	any register And R5	2	5	2	any register And R5	47	56
WEIGHTS	~70% "passing" of variables	1	2		1	4	8			
	~45% "passing" of variables	2	4		1	4	8			

Recommendation

The preceding analysis yields the somewhat surprising result that making all calls re-entrant (potentially) by placing argument addresses on the stack gives only a Ø - 5% overall increase in calling sequence length over the best non-reentrant method. The reader should now consider the necessity for re-entrant calling sequences. The writer believes that they are necessary because

(1) Our competitors have widely advertised that they supply re-entrant software, and

(2) DEC has widely advertised that writing re-entrant software for the PDP-11 is "easy".

(3) The interrupt structure of the PDP-11 makes re-entrant coding useful.

(4) Many projected applications for the PDP-11 will benefit from re-entrant code.

Since system programs such as the Disk Monitor will make extensive use of re-entrant calls, a call method which gives little penalty for re-entrant calls is required for their use. Thus it appears that the All Calls Re-entrant Method is the best calling method available, and therefore it is proposed that this method be the standard to be used by FORTRAN (and others) for <u>all</u> call sequences.

Conventions for Calling Sequences

Now that the standard method for calling sequences has been determined, the following set of conventions can also be proposed as standards.

Register Usage

General register 5 (R5) will be the standard "calling register". That is, the JSR statement will take the form

```
JSR R5, Subroutine Name
```

This provides automatic saving of R5.

Argument Order

The arguments for a subroutine will be placed on the stack in reverse order to their appearance in a CALL so that the "first" argument will be on "top" of the stack (see detailed code in the earlier discussion of the adopted method for an example).

FORTRAN Functions

A FORTRAN function will return its single result on the top of the stack. This facilitates coding for such statements as

A = SQRT (COS (ABS (B)))

Also, since floating point numbers require multi-word representations, returning the result on the top of the stack requires less register manipulation than returning the result in general registers. It should be pointed out, however, that this convention precludes coding routines which can be used as both SUBROUTINES and FUNCTIONS.

Register Saving/Restoring

Since the PDP-11 addressing schemes will usually require that any program use all general registers, it will be more efficient if each subroutine save and restore any general register which it uses.