WRITING SPL ROUTINES WHICH ARE CALLABLE FROM BASIC

BY

WARREN KUEHNER
SYSTEMS ENGINEERING SUPERVISOR

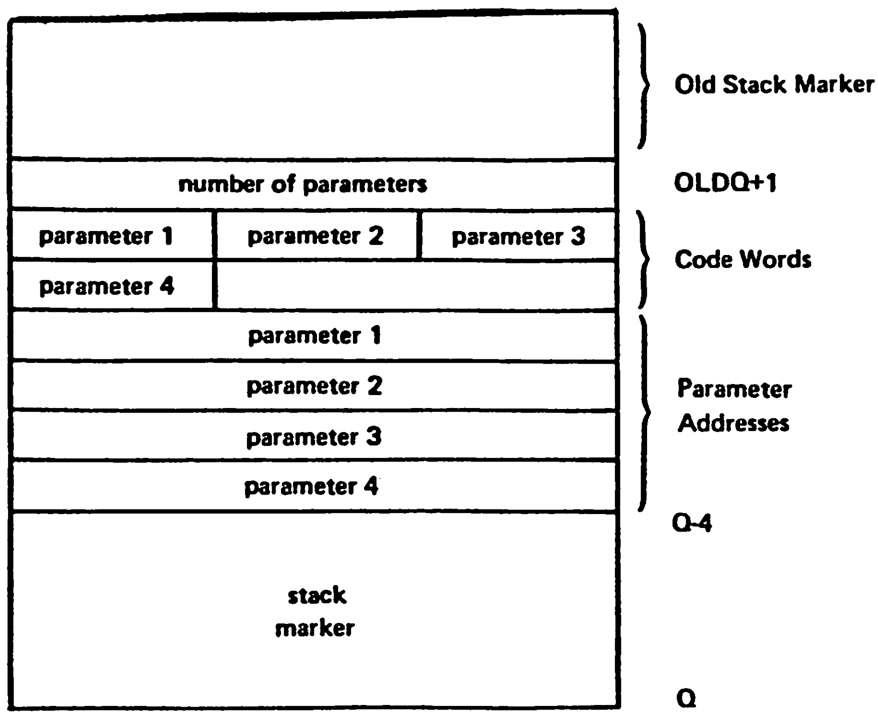
HEWLETT-PACKARD
NEELY SALES REGION
ENGLEWOOD, CO.

This paper discusses programming techniques involved in the creation of SPL routines to be called from BASIC. It specifically covers both trivial and more complex programming examples, and consideration, for putting these routines into operation with BASIC programs. This paper assumes a good knowledge of both SPL and BASIC.

HOW BASIC CALLS A SUBROUTINE:

The key to understanding the possibilities for BASIC callable SPL routines is to understand what additional information (other than that which is normally passed) BASIC provides with a call. In addition to the passed parameters and the stack marker, BASIC places two other pieces of information on the stack ahead of the passed parameters. One is the number of parameters passed. The other(s) are a code word for each passed parameter indicating what data type it is and whether it is a simple variable or an array. (Refer to Appendix F of the BASIC/3000 manual for information on the codes and the passed parameters). This information can be used by the SPL routine to verify the correctness of passed parameters. Another use is to allow the passage of a variable number of parameters to the SPL routine.

When BASIC calls an external routine, the stack is as illustrated on the following page.



BASIC first calls a routine (which generates a stack marker) which places an integer indicating the number of parameters, the parameter codes, and the addresses of the passed parameters (in that order) on the top of the stack, and then calls the desired external routine.

THE TRIVIAL PROGRAMMING CASE:

Given the foregoing background information, it should be obvious that in the trivial case, SPL routines can be written in the normal way and called from BASIC. For "quick and dirties" this could be adequate. It should be noted that BASIC does no passed parameter checking of any kind and this approach is potentially risky.

THE MORE COMPLEX CASE:

It has been pointed out that the additional information on the stack can be useful for at least two reasons; parameter checking and variable parameter passing. The following program example illustrates how to get at this information:

```
INTEGER DELTAQ = Q - Ø; (( THIS VARIABLE, WHICH IS LOCATED AT
                           Q (WHICH CONTAINS THE DISTANCE BACK
                           TO THE LAST LOCATION OF Q) ALLOWS
                           US TO FIND OUR WAY AROUND))

INTEGER POINTER NUMPARM;(( POINTERS TO THE VARIABLES CON-
                CODES,     TAINING THE NUMBER OF PARAMETERS,
                PLIST;     THE PARAMETER CODES, AND THE BASE
                           OF THE PARAMETER ADDRESSES))

@NUMPARMS: = (@DELTAQ + 1) - DELTAQ; (( NUMPARMS MUST POINT
                                        TO THE ADDRESS OF "Q"
                                        PLUS 1 LESS THE VALUE
                                        OF DELTAQ))

@CODES: = (@DELTAQ + 2) - DELTAQ:     (( SEE ABOVE))

@PLIST: = @CODES + (NUMPARMS + 2)/3; (( THIS CALCULATION WILL
                                        CAUSE PLIST TO POINT
                                        TO THE ADDRESS THE
                                        FIRST PASSED PARAMETER))
```

By indexing through PLIST and equating its contents to other pointers, one can locate any of the passed parameters. For example, to locate the third passed parameter which is a string (byte array) the following is necessary:

```
BYTE POINTER STRING;
@ STRING: = PLIST (2)
```

Obviously, the problem of checking parameters can be solved, being able to get at the number of parameters and their types.

The variable parameter passing problem can also be solved with this information.

The idea of variable parameter passing can be very useful. Good examples are the Basic callable Image routines. DBGET can, for example, be passed a variable number of strings to receive the buffer of data from the data base. This gets around the limitation on string length as well as allows the placement of logical pieces of the data in the data base in strings dimensioned to contain them.

It should also be noted that information about the length of a string variable (and also about array dimensions) is available to the SPL programmer, and that this too can be quite useful.

For example, if STRING is a string variable (byte array), then STRING (-1) contains its length as dimensioned in the BASIC program. (Refer again to Appendix F of the BASIC/3000 manual for more information.)

NOW I WROTE IT---HOW CAN I MAKE IT RUN?

If one is running his BASIC programs from the interpreter, the external routine called must be located in an SL file, either in his group, in his account's PUB group, or in the system SL. The interpreter automatically searches all three. To place the routine in the SL, one must use the SEGMENTER, and the procedure is as follows:

```
: SPL MYPROG, MYUSL
$ CONTROL SEGMENT = MYSEG
  :
  END.
: SEGMENTER
- USL MYUSL
- SL SL ((BUILD AN SL IF YOU DON'T HAVE ONE))
- ADDSL MYSEG
- EXIT
: ----NOW RUN!
```

To use the routine from a compiled program, the steps are the same as for a compiled program in any language, that is, the routine may be either:

- Compiled into the same USL file as the BASIC program and then that USL PREPED.
- Put into an RL file (with the SEGMENTER) against which the USL can be PREPED

  : PREP MYUSL, MYPROG; RL = MYRL

- Called from an SL. Remember if the SL is in your group to use the appropriate LIB = parameter, that is:

  : RUN MYPROG; LIB = G.

The purpose of this paper was to present the tools to write BASIC callable SPL routines. I would enjoy your comments, criticisms, or to hear what you've been able to do with these ideas.