DESIGN AND SEGMENTATION TECHNIQUES

FOR LARGE SPL PROGRAMS

Jon W. Henderson

General Systems Division
Hewlett-Packard Company

The ease of use of the HP3000 encourages the development of larger and more sophisticated applications which begin to approach the limits of the machine.
This paper deals with those constraints encountered in the compilation of very large (250,000 lines or more) SPL programs, and methods for circumventing them.

The limits encountered in SPL compilations are of two forms: absolute hardware limits and compiler-imposed ones. The most common hardware limit encountered is exceeding the addressing capacity of the DB and Q registers. Since global variables in SPL are DB-relative a maximum of 256 may be declared.
Similarly, 127 Q-relative local variables can be declared in a procedure. These two bounds are absolute and may be avoided only through programming techniques described below. Of the 127 local variables declared only 64 may be arrays. This is one of the compiler-imposed limitations. The other, more troublesome, is overflowing the compiler's symbol table.

Each identifier declared in an SPL program is inserted into the symbol table, along with relevant information about it. The number of words of symbol table used for each identifier varies from a minimum of 4 for a simple variable with a one-character name, to a maximum of 146 for a 256-character define with a 16-character name. (A new $CONTROL option, DEFINE, has been added to SPL to alleviate this problem. $CONTROL DEFINE causes the bodies of defines to be written out to disc rather than kept in the symbol table. This has

proven useful in large compilations. The DEFINE option will be available in the next version (08.00) of SPL.)

The symbol table is kept in the area between the DB and DL registers, and is expanded "downward" from DB toward DL. An initial allocation of 1000 words is provided and increments of 512 words are added as needed. When the marker at the top of the symbol table + 512 exceeds DL the SYMBOL TABLE OVERFLOW error message is output and the compilation is aborted.
Case studies have shown that on the order of 10,000 identifiers are necessary for this to happen, but overflow situations are occurring with increasing frequency as the complexity of SPL programs grows.

There are several methods for preventing these problems from arising. The most important of these is a programming practice of keeping program units small, thereby taking advantage of the powerful segmenting and linking capabilities of the HP3000's Segmenter.
The basic philosophy behind modularization is that by separating global declarations from code and keeping program units small the burden on the compiler can be lessened, thereby increasing the resources available, reducing compilation time, and, indirectly, making the program easier to maintain and enhance. This can be accomplished in two ways:

1) Use of the new SPL compiler option, $INCLUDE.

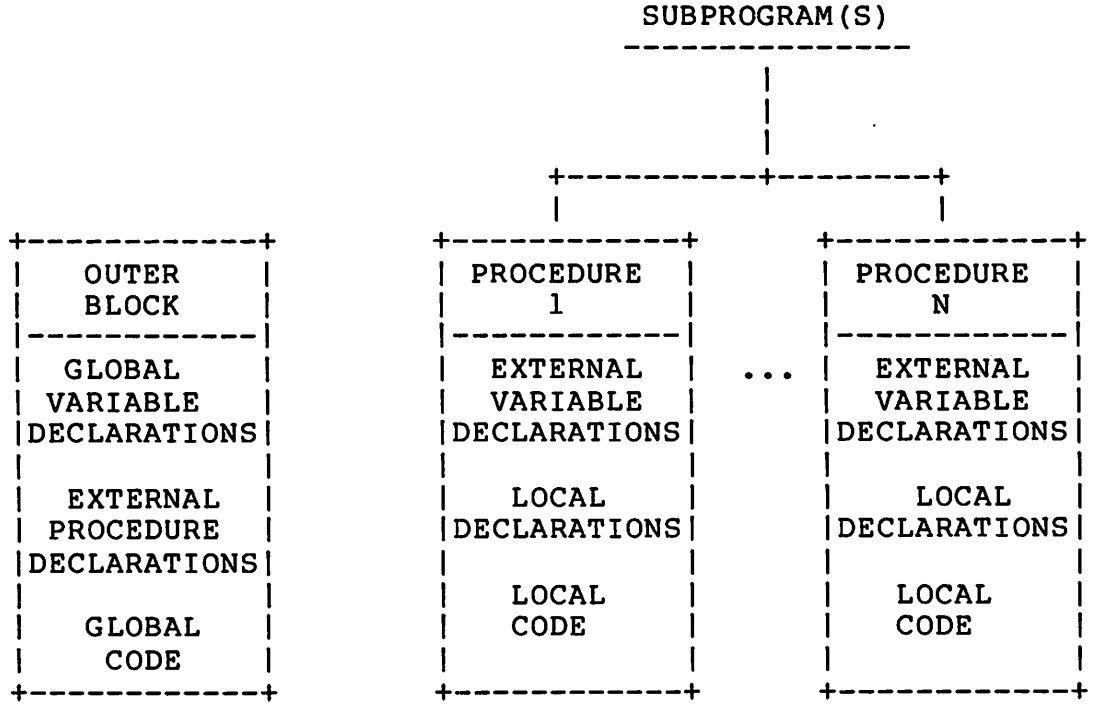
2) Use of the GLOBAL and EXTERNAL options in variable declarations.

In one case study some 10,000 variables were broken down into two classes: the 256 variables global to the entire application, and equates and defines. The equates and defines were further subdivided according to function. Then some 160 procedures were modified to $INCLUDE the file containing the global variables, plus only those equate and define files necessary for compilation. Before the restructuring the symbol table overflowed. Now the program compiles. The $INCLUDE option will also be available in version 08.00.

By declaring variables as GLOBAL and EXTERNAL the programmer allows SPL the leave the linking of the variables with their appropriate offsets to the Segmenter. Since only those global variables used by a procedure need be declared inside it, recompilation of the all the global variables whenever a procedure is recompiled is unnecessary.
The use of GLOBAL/EXTERNAL also encourages subprogram compilations. A subprogram is a discrete compilable unit, usually a procedure or group of procedures. For

this technique to be effective, the subprogram must contain all the declarations necessary for stand-alone compilation. When the compiler is placed in subprogram mode by the use of a $CONTROL SUBPROGRAM statement at the beginning of the subprogram, only the procedure(s) specified are recompiled into the existing USL file.

Ideally, the structure of a large SPL source would appear as follows:



OUTER BLOCK -

Global variables are declared in the following form:

GLOBAL <variable type> <variable name>;

i.e., GLOBAL INTEGER I;.
The GLOBAL declaration identifies the variable to the Segmenter as one to be matched up with a corresponding EXTERNAL declaration in the outer block.

An EXTERNAL PROCEDURE declaration is necessary for each separately-compiled procedure called

from the outer block. These declarations use SPL's OPTION EXTERNAL feature. This is similar to OPTION FORWARD, and again instructs the the Segmenter to do the necessary linking. For example,

```
PROCEDURE P(A,B);
   VALUE A;
   REAL A,B;
   OPTION EXTERNAL;
```

No corresponding GLOBAL declaration is necessary.

GLOBAL CODE is the outer-block code, containing the entry point to the program.

SUBPROGRAM -

EXTERNAL VARIABLE DECLARATIONS are declarations which cause SPL to place in the USL file information which the Segmenter uses to link code referencing the variable with its address. Each external variable must be declared in each procedure in which it is used. The form of the declaration is the same as that of the GLOBAL variable declaration, except that EXTERNAL replaces GLOBAL.

LOCAL DECLARATIONS and LOCAL CODE are as found in any procedure.

The process of creating a complete USL file looks like this:

1) The outer block is compiled, in program mode, into an initialized USL file. ($CONTROL USLINIT is a good idea at this point.)

2) Each module is compiled, in subprogram mode, into the USL file.

The method of recompilation varies, depending on what is to be recompiled. If the outer block is modified, then the entire source must be recompiled. However, if only a module is changed then just step 2 is necessary.

When a subset of the procedures in a subprogram need to be compiled, the parameterized SUBPROGRAM statement is useful.

```
$CONTROL SUBPROGRAM [(proc-name[*][,...,proc-name[*]])]
```

When procedure names appear, only those specified are compiled. An asterisk following the procedure name causes

that procedure to be compiled with the LIST, CODE, and MAP $CONTROL options turned on.

SEGMENTATION

Increasing the modularity of programs presents special segmentation problems. For large programs, proper segmentation can greatly increase both program and system throughput. The following is a description of some guidelines for optimum segmentation of 3000 programs. (Acknowledgements to John Page & Madeline Lombaerde of GSD.)

The 3000 is a process oriented machine, incorporating the separation of code and data, and stack architecture. This permits easy design of re-entrant code. The purpose here is to discuss ways of making a particular process:

a. Run as fast as possible

b. Have minimum effect on other processes in the system.

PROCESS ENVIRONMENT

HP3000 object code is executed by MPE in the form shown in Figure 1. The process has a single data segment (or "stack") and a variable number of code segments of varying sizes. When a program is written the following can be controlled:

a. the size of the stack

b. the number of code segments

c. the size of each segment

d. which code goes into which segment.

The diagram in Fig. 1 is actually a simplification since it does not show the externals referenced by a program (see Figure 2). If, for example, an SPL program calls FOPEN then a link will be created from the code to an MPE segment containing the FOPEN intrinsic code. Most of these intrinsics and all of the Compiler Library routines are not in memory permanently, thus they are viewed by MPE as code segments identical to the caller's. Although SPL programs have more control over which

external procedures are called than other languages, proper control of any language's program code and data decreases the run-time of a process and its impact on system load.

HOW TO DETERMINE A PROGRAM ENVIRONMENT

When a program is prepared the PMAP option shows the size of each segment, which procedures are in which segment, and the names of externals called by each segment. The MPE Commands and Debug/Stack Dump reference manuals describe the format of the PMAP in detail.

HOW MPE RUNS A PROGRAM

There are two MPE modules concerned here: the dispatcher and the memory management system. The dispatcher is responsible for the allocation of CPU time to all the executing processes. The memory management system has the job of fitting code and data segments into memory as they are required, this operaton often necessitating the decision of which segment(s) to delete to make room for others. When a process' time-slice starts, its stack is made present in memory and control is passed to the program. As the program runs, it will call procedures which are not in the segment present in memory. At this point the program is suspended while MPE arranges to make the required segment present. This can take from 20 to 100 milliseconds, since a disc access is involved. While this is going on the dispatcher tries to run the next-highest-priority process already resident in memory. When the required segment is made present, control is passed to the called procedure.
The point to be noted here is that calling a non-resident code segment is very time-consuming.

CAN THE PRESENCE OR ABSENCE OF A SEGMENT BE DETECTED?

No. The memory management system will simply attempt to keep the most "popular" segments in real memory. The smallest set of segments (both code and data) which must be in real memory for a program to execute efficiently is called the program's workng set. This dynamic set of segments may, and most often does, change continuously during the life of the executing program.

The philosophy of the HP3000 memory manager is based on the idea that there is an ideal absence frequency for an executing process. If a process gets more than the expected number of absences, the memory manager concludes that the process does not have enough segments in its working set and proceeds to add the requested (absent) segment to the process' working set.

However, if the process executes for a long time without absence faults the memory manager concludes that the working set is too large, and real memory is not being used efficiently. The least-used segments are removed from memory and made available for overlay.

From the user's point of view, the internal function of the memory manager cannot be influenced. Applications can be designed, however, with the working set concept in mind. First, it should be kept in mind that the total of all working sets active at the same time (i.e. a total of all the commonly used segments, at any given time, for all application programs running concurrently) should be no more than 75% of the memory available to the user. A rough rule of thumb for determining working set size is that the size of the working set + the size of all MPE intrinsics used should be no more than 75% of the linked memory available to the program.
This restriction on working set size is critical and directly reflects the memory management segment replacement algorithm.

RULES FOR PROGRAM SEGMENTATION

1) Minimize the number of times the program crosses a segment boundary. In other words, stay within a segment for as long as possible; when you leave it, stay out for as long as possible.

   DESIGN IS IMPORTANT

   Do not leave segmentation until the last minute. As will be shown below, it is possible to write a program which cannot be properly segmented.
   Any procedure or outer block Relocatable Binary Module (RBM) must reside entirely within a segment. Thus if it proves necessary to move a block of code into a separate segment, it will only be possible if the code is a procedure. Arbitrary sets of instructions cannot be taken and placed into a named segment, the whole RBM must be moved. Therefore, modularizing code into procedures is of vital importance during the design phase.

   CONCEPT OF LOCALITY

   The locality of a program is the degree to which control remains in the same general area of code. A high locality means that control remains in the same area for a long period of time. Poor locality means that the program branches frequently. The 3000 needs programs that have good segment locality, but does not

care about the degree of locality within a segment. Branching from segment to segment continuously is wasteful, but branching within a segment makes no difference.
If correctly applied, the principle of locality minimizes the number of possible absence traps and segment switches during execution. Although transferring control between memory-resident code segments takes less time than accessing segments on disc, it still requires more execution time (approximately 2.5 times longer) than transferring within the same segment.

FUNCTIONAL vs. TEMPORAL SEGMENTATION

As large systems are structured and modularized into smaller and smaller procedures, the grouping of these procedures into segments becomes of paramount importance. Intuitively, one segments according to the function of the procedures. That is, all the input decoding routines are put together, the input/ output routines are put together, etc. This could not be more wrong. Segmentation is a speed-enchancing operation; time, not function, is the critical dimension. Since Rule No. 1 says stay in a segment for as long as possible, control must flow smoothly from segment to segment as the program progresses.

As an example, consider a small utility program which dumps a file to the line printer in some special format. Assuming that the operator can choose the name of the file and which of three possible formats to use. The program is written with four procedures: A, B, C, and D. (See Figure 3.)
Assume also that each dump routine has a procedure to fetch a record from its file and a procedure to format a print line.
It would be tempting to put all the formatting routines in one segment, and the record fetching routines in another. This would cause a segment boundary to be crossed twice for every record dumped, perhaps thousands of times. The correct way is to put each record-reading procedure (B1, C1, D1 in Figure 4) with its corresponding output procedure (B2, C2, D2). If A is in its own segment then only three segment boundaries are crossed ofr a whole dump. In a busy system this could make large differences in program run time.

In summary, estimate the number of times a segment boundary is crossed and multiply this by 40 milliseconds. This is the time your program will

be doing no useful work and other processes will be interrupted.

Assuming that some good segmentation scheme has been devised so that good segment locality exists. The next step is reducing the size of the "working set."

2) Do not burden your working set with infrequently used code.

FREQUENCY OF CODE USE

The working set of segments is the set that consumes most of the CPU time. For example, in the program above the working set is the code that excutes the main loop such as B1-B2. If it is assumed that B1 and B2 are in a segment of their own called BSEG, then the system may spend many minutes in this segment for a large dump. It is therefore important to minimize BSEG's size in order to reduce the competition for scarce memory.
To do this, examine the code in the working set and remove any code which executes infrequently. Very often, this applies to error-handling code. When a program detects an error, the error should not be handled in-line. Instead, call an error message generating procedure. The procedure should be in a separate segment and thus not clutter up memory while normal, error-free, processing is going on. As an example, suppose in the above program that after doing an FWRITE the condition code is checked and, if end-of-file is detected, an elaborate file-extension routine is executed. If the routine is expected to execute infrequently why keep it in precious memory with the working set? Banish it to some auxilary segment and let MPE fetch it only when needed. As a reminder, this routine must be a procedure before it can be placed in another segment.

```
  WRONG                                  RIGHT

FWRITE(. . .);                         FWRITE(. . .);
IF > THEN                              IF > THEN
 BEGIN                                   EXTEND'FILE;

   .

 <<CODE TO EXTEND FILE>>                 Procedure
   .                                     EXTEND'FILE
                                         is in
   .                                     another
 END;                                    segment.
```

3) Keep the principal working set small and make infrequently used segments large.

   SEGMENT SIZES

   This is a trade-off. A lot of small segments are easier for the memory manager to place in real memory. However, a scarce rsource is being used up in the form of Code Segment Table Extension (CSTX) entries. One entry in the CSTX is needed for every program segment, and the table has a maximum of 63 entries per program executing.

   At the opposite end of the spectrum, a program might have a few large segments. While this does minimize segment-boundary crossings, the effect on memory can be devastating for other users. There is no simple answer to the question of optimum segment size. The main idea is to minimize the size of the working set.

4) Keep initialized variables, particularly arrays, out of the global declarations whenever possible. If they must be global, don't initialize them at declaration.

   SHARED CODE

   If a program is going to be run from multiple terminals then the code segments will automatically be shared by the multiple processes. Each process will have its own stack. If the program's design

incorporates data which is never altered, such as error messages, tables, etc., then by placing this data in the code, rather than in the stack, only one copy is required for all processes. There are two ways to accomplish this:

1) Initialize arrays in-line, rather than at declaration.

2) Whenever possible, declare constant arrays inside procedures as PB-relative. This has the added benefit of reducing symbol table overhead.

In both of the above cases SPL will store the initialization string in the code segment, effectively sharing it among the processes. This reduces the size of the stack.

In summary: keep modules small, compile them separately, and segment them wisely.
When the above methods of design and segmentation are applied to large SPL programs the result is reduced design, maintenance, and execution costs.

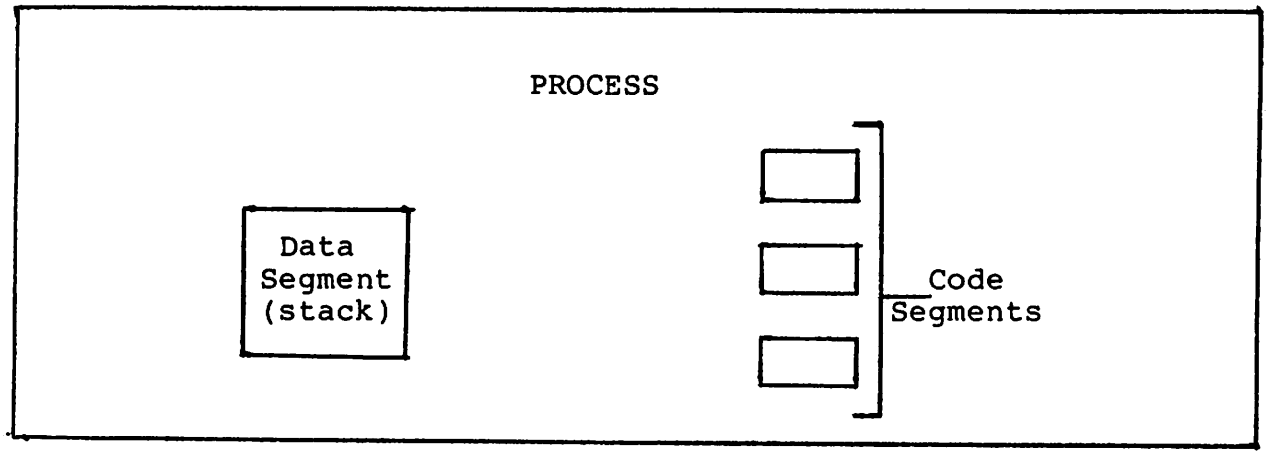


Figure 1.

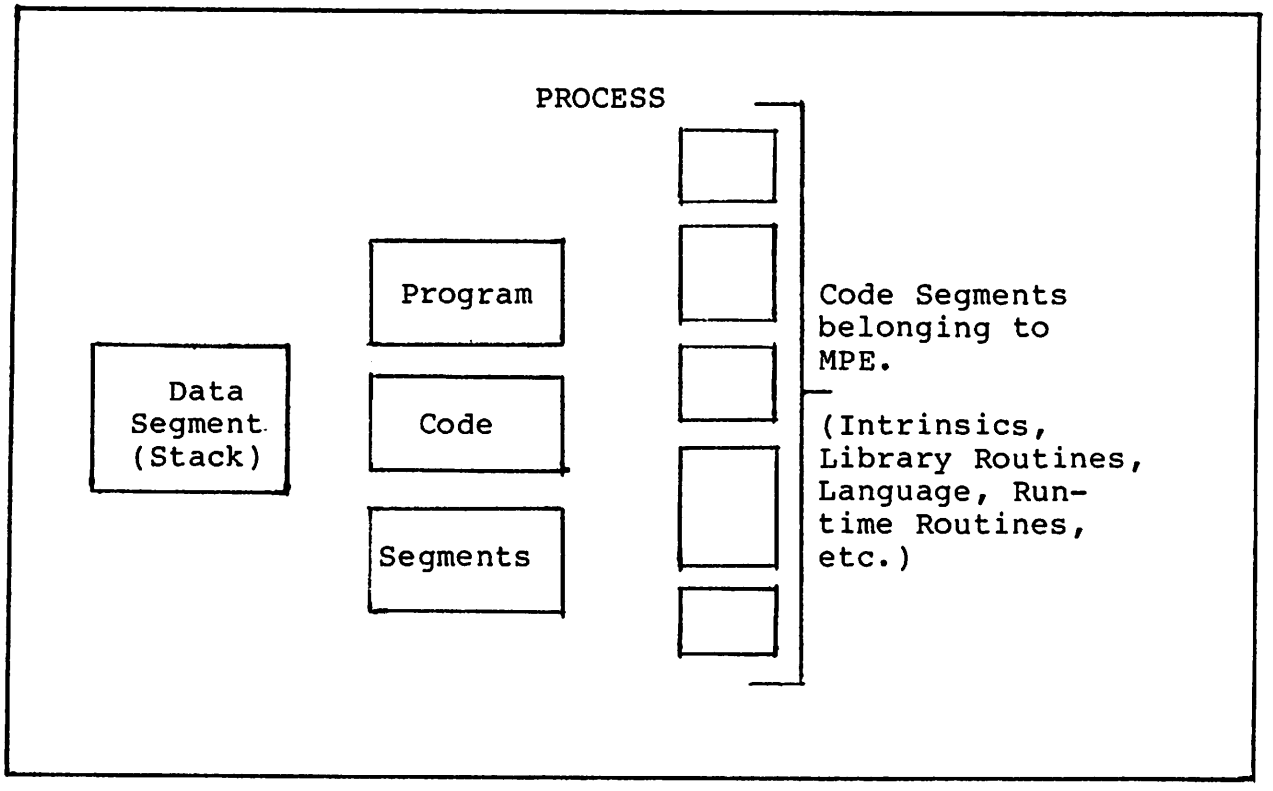


Figure 2.

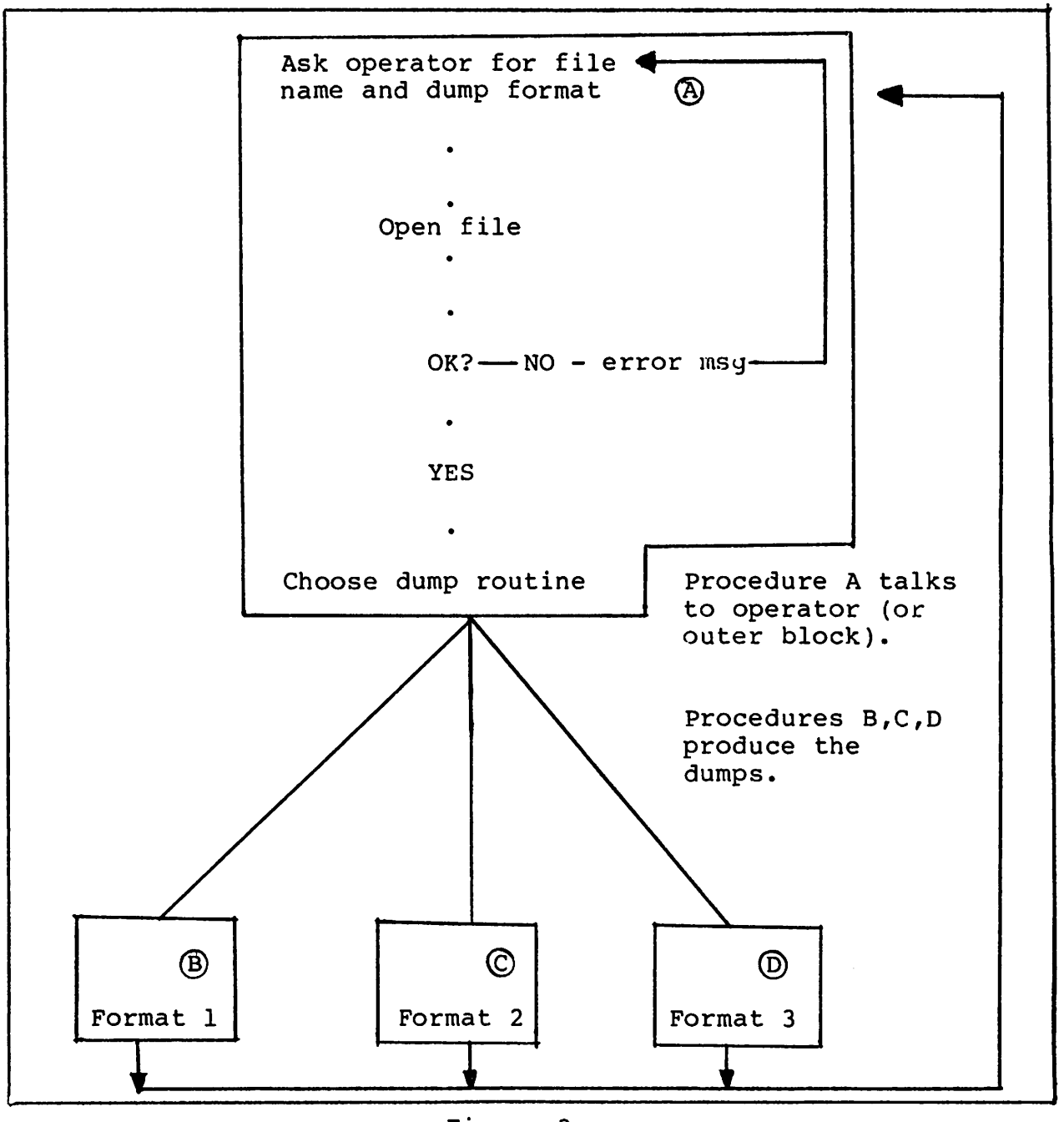

Ask operator for file name and dump format
Ⓐ
Open file
OK?
NO - error msg
YES
Choose dump routine
Procedure A talks to operator (or outer block).
Procedures B,C,D produce the dumps.
Ⓑ
Format 1
Ⓒ
Format 2
Ⓓ
Format 3

Figure 3.

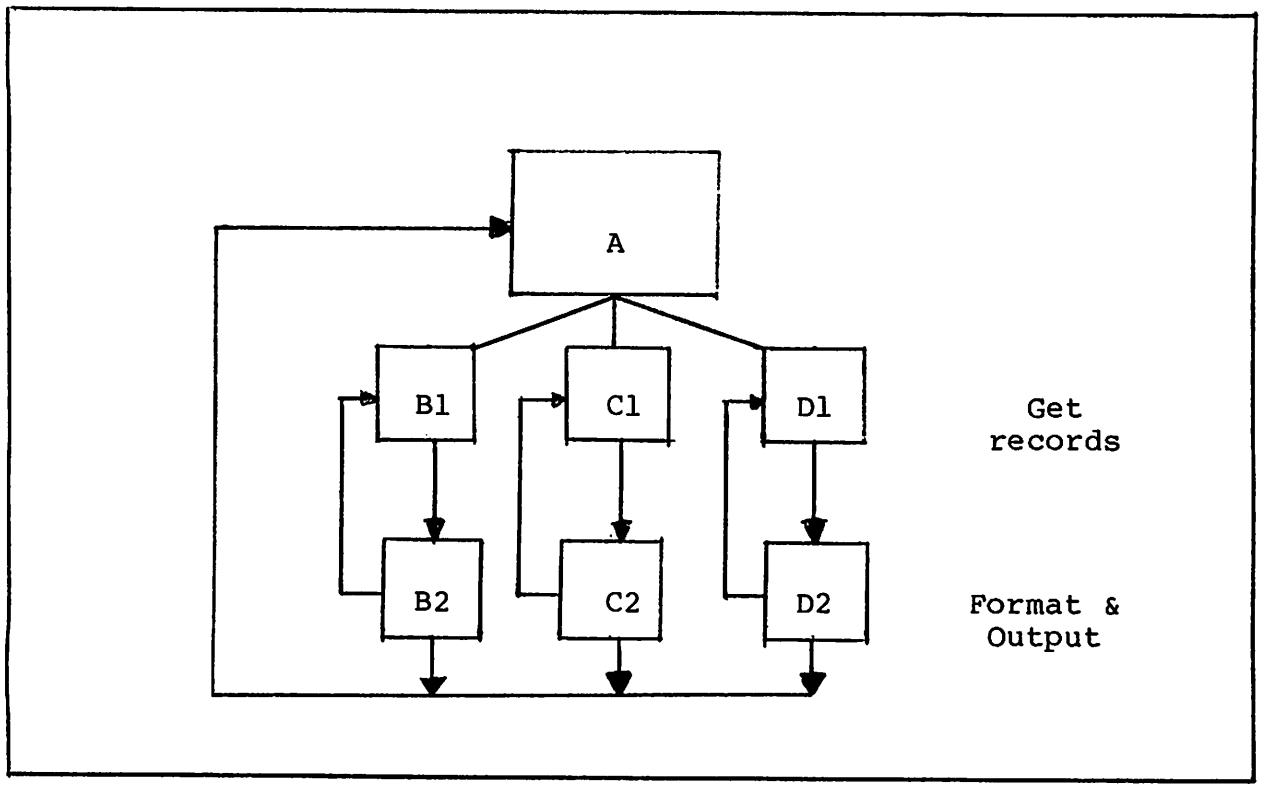


Figure 4.