Programming For Survival

by

Gerald T. Wade
Product Specialist

Hewlett-Packard
Neeley Sales Region
Englewood, Colorado

ABSTRACT

This article deals with programming techniques to generate maintainable programs which will survive the loss of their author. It is geared primarily for the individual programmer rather than the leader of a programming team but it should be useful to both. The techniques described apply both to the creation of an entire program or only to the creation of one subsystem, as is the case in a team effort. One of the desires of the author is to promote "Egoless Programming" with overall program quality rather than individual programmer mystique as the end result.

FORWARD:

The purpose of this article is to discuss programming techniques which will aid in the maintainability, hence the ultimate survival, of programs. I take as a basic premise that few if any programs are ever written such that they never require modification. This modification may be necessary due to changes in program requirements, operating systems, host computer, etc. or to 'Bugs' found in the program. Whatever the reason, very few programs escape the need for maintence. As a corollary I purpose that in many cases the modification of a program will be made by someone other than the original programmer. In general programs tend to be written to satisfy the needs of a particular site and thus tend to remain at that site even after their original programmer has left in search of a better position. The conclusion that must be drawn is that in general a program will have to be modified and that such modification may be done by other than the original programmer.

If I may digress a moment into historical speculation: In the early days of computer programming the prime constraint on the programmer was his hardware. Hardware tended to be bulky and extremely expensive. As a result an installation would have to operate on the least amount it possibly could. One of the main hardware restrictions was the amount of real memory available in the computer. Virtual memory systems came along later to ease the situation but they brought with them penalties in program speed and complexity. As a result much emphasis was placed on coding a program such that it required the least amount of memory possible. Programming techniques developed with this goal in mind. For example; Variable locations might be reused for several purposes as the program progressed. This saved using a separate location for each use but made it difficult to determine the exact contents of that location as the program ran. Sections of code were often overwritten with data arrays after they had been executed. This technique saved much memory but could be very confusing if a modification tried to use the overwritten code. A similiar technique was to modify a section of code in order to configure it to perform various functions as the program progressed. This was done primarily in assembly or machine language but it too could cause much difficulty for someone not aware of what was being done.

All of the difficulties mentioned were dismissed by the programmers by explaining that they were smart enough not to do anything like that. While this might seem to be a valid argument I shall draw on my speculative history to discredit it. Hindsite shows us two basic occurences.

First, as I have mentioned, these programs were often required to be maintained by other programmers after the original author left. This means that someone else would have to learn what tricks had been used in order to avoid errors. If the programs were well documented and told just what had been done then this would be no serious problem. This was generally not the case. Program documentation was skimpy and often even misleading. I attribute this fact to the programmers ego and his sense of survival. Remembering that the prime method for determining a good from a bad program, and by association a good from a bad programmer, was how little memory was used, the programmer was naturally reluctant to reveal all the tricks he used to achieve this objective. By keeping these techniques to himself he would be better able to stay one step ahead of his competitors and boost his own ego or 'mystique'. Extensive documentation was often omitted as a time consuming task with little reward for the programmer. Thus the program documentation was not sufficient to allow another programmer to modify or maintain the program. Second, even if a programmer was modifying his own program at a later date he might have forgotten all the tricks he had originally used and thus fall into the same traps as the outside programmer.

The result of these types of problems often was that programs became ineptly patched, slower to execute, and unreliable. As an end result they would have to be thrown out and a new program written in their place, even when the changes to the original program were all small ones. We have set the stage for much duplication of programming effort and a bad reputation for the computer industry due to the unreliability of its programs.

What of the present and future? While technology has been advancing at a blinding rate and removing most if not all of the original constraints on the programmers, they have failed to update their techniques to keep pace. Memory cost and bulk has been so reduced that most installations can now afford to buy essentially all they need. With the advent of high speed secondary storage devices, (drum, disc, and soon magnetic bubbles), virtual memory systems become much more viable. What we have then is a totally new environment for the programmer. In the past he had to conserve memory by whatever means possible but today he can sacrifice some program size for clarity and maintainibility.

The rest of this article will deal with methods to write programs such that they will (1) accomplish their objectives reliably (2) be as simple as possible to

maintain and modify, even by a programmer other than the original author (3) possibly survive and remain in use long after the original author has gone. It is my contention that the ability to write simple to understand, reliable programs that remain in use is a far better goal than to be thought a 'magic man' for the ability to write programs that no one else can understand.

The first sections of the article deal with general techniques that might be applied to any computer system. The remainder of the article deals with special techniques that may be used in order to write programs for the HP-3000 system.

PROGRAM STRUCTURE

The term STRUCTURED PROGRAMMING has become a popular buzz word in todays society. As such its definition has become so twisted as to make it almost a useless term. Let me define what I mean when I say that a program should be properly structured.

A program may be thought of as existing at several different levels. The outermost level is the grossest look at the program and answers the question. Just what is this program going to do ? or Why was this program written ? The next level breaks the program into major functions or 'blocks'. For example, there might be an initialization block, a function selection block (if the program has multiple functions), a block to perform each function, a block to deal with anticipated error handling, and a block to handle any finishing housekeeping, such as closing files, printing summaries etc.

Each block within the program may then be further divided into smaller sub-blocks. A sub block might contain the code to read the input file, or to perform a sort on the data etc. The key thing about a sub-block is that it must be small enough to be fully comprehended by someone reading the program. This means, in practical terms that it should absoultely be no longer than one page of source code, preferable about one half page in length. One and only one operation should be preformed in the sub-block. Thus it would be improper to create a sub-block that reads and sorts the input file if indeed those are two separate operations. The beginning of each sub-block should have a comment that describes the operation to be performed and the person reading the program should be able to verify that the code in the sub-block does indeed perform just that operation.

This approach to structuring a program is also known as the "Top Down" programming method or the method of "sucessive redefinition." Many papers and books have been written about these methods and I will not elaborate further here except to say that I normally only follow the methods in gross structure but not in actual implementation. Some of them involve large amounts of formal structuring that I see as time consuming and appropriate only for the largest of programming tasks. For most programs it is only necessary to keep the concepts in mind during the programming, not to generate large amounts of paperwork.

THE PROGRAM DEFINITION STATEMENT

At the start of each program there should be a comment that states the exact purpose of the program. It should answer the questions "Why was this program written ? What does it do "? The answers to these questions might seem all to obvious at first but their answers must be fully understood in order to direct any further development of the program. I was surprised to find the large number of programs that were written and actually being used without stopping to ask just what it was that they were trying to accomplish. An example might be a program that was written to "analyze the system log files". While this might sound like an admirable objective, it leaves many questions unanswered. Will this program allow me to print a report showing the number of power failures logged on a given day of the week ? Can it tell me how many lines each of the system users have printed on the line printer ? Depending on the answers to this type of question, the program might be a small task or a major programming effort. No program should be attempted with so skimpy a definition. A better statement of program function might be: "This program was written to read the system log files and to produce a summary file. The summary file will be one record for every job or session run and contain the total number or records transferred to each I/O device by that job." With this definition it is easier to tell just what the program will or will not do.

The program should adhere rigorously to its stated objectives. This is a means of avoiding the program that starts out to do a simple task and ends up growing in to a monsterous 'klugde' that attempts more than its modest original framework can support. The temptation is great to take a program that does 'almost exactly' what you want and add to it until it can perform both the old and the new function. The problem with this is that the ground rules for the program are being changed. For example, a program that was designed to read the system log files and print out a summary of powerfails might be a candidate to be built into one that can also produce a summary of I/O errors by device. Then it might be modified to create job summaries and print histograms of system usage (CPU or CONNECT TIME). This sounds like a viable thing to do since the original program already has the code necessary to read the log files and extract some of the information. It seems that a lot of time might be saved by avoiding the duplication of that code. I do hereby put it to you that you should avoid this pitfall like the plague! "Why"? you might ask. The problem arises in the fact that at the time the original program was written certain decisions had to be made as to the best way to handle the task. These decisions were made based on the original design objectives

of the program. By allowing the design objectives of the program to change after the fact, you may have locked yourself into some no longer valid decisions. For instance, when the original decision as to how to handle reading the logfiles (whether to read record at a time utilizing the file system or to invest the time necessary to write a special internal deblocker) was decided, the size of the task at hand could not justify the time spend in writing an internal deblocker vs the time lost in reading the few powerfail records. Thus the slower file system was used to read the records. In the later version of the program it would make a great deal of sense to do internal deblocking of the log records since the volume of records read was now very great. At this point, the program is already locked into the record at a time scheme of reading the log files and could not be easily converted into a new scheme. Thus, by revising the design objectives of a program you might have ended up with a program that has a much poorer performance than desired, or you will end up spending far more time to convert the old program to a more efficient scheme than if you had just started from scratch.

A further effect of allowing the design objectives to change after the program is written is that the program will end up looking 'LUMPY'. By lumpy, I mean that you will be able to see that the original program shell is here, and a 'LUMP' has been added here for this function, another one here, one there and so forth. In the end the program may be so lumpy and consist of so many different internal techniques, variable names, that it will be almost impossible to find and fix any bugs that it has. Indeed if allowed to continue, the program modifications will eventually create a program that is non functional and non maintainable, in short, totally useless.

If there is any question as to how much the initial program definition statement should encompass, I suggest the following guideline. Make the original program definition cover as large an area as the program will ever be allowed to perform. You may add qualifiers to the effect that it is envisioned that the program will perform these functions at a later time and that they are not completed at the present time. You may then plan the program such that these non-implemented functions will be possible, even to the point of including dummy program blocks to mark the places they will be added. In this way the initial decisions on the methods used by the program will be valid even if the program is expanded to its maximum.

The program definition statement serves two functions: First, it informs the person reading the program source as

to just what to expect this program to do (and conversly what to expect it not to do). Second, it serves as a guide as to what modifications fall into the original concept of the program, and thus are vaild changes to it, and what functions are outside the original concept and should best be handled in another manner. The program definition statement should always reflect the current status of the program but should be changed only after a great deal of thought and soul searching.

MAJOR PROGRAM BLOCKS

The structure of a program should be broken down into major blocks, each block responsible for a certain function. A function might be best described as that section of the program which performs a logically related set of tasks. An example might be for a program that is designed to read the system log files and then either print a summary of powerfailures or write a summary of the I/O written for each job or session run on the system. A possible division into functions might include the following functional blocks:

Initialization and selection of the desired function.

Extraction of the powerfail records from the log files and the accumulation of summaries.

Printing the powerfail summary.

Extracting the I/O records from the log files and the accumulation of summaries.

Printing the I/O summary.

Handling any internal errors or file errors encountered.

Closing the files and finishing up.

Notice that not all functions need be performed for each execution of the program.

At the beginning of each block should be comments that describe exactly what function that block is to perform. This is similiar to the program definition statement at the beginning of the program. The same rules apply to the block definition statement as applied to the program definition. That means that the block definition statement should rigorously define the function of that block. No deviations to the block's function should be allowed without verifing that the statement will still be valid. Also at this pcint any interconnections, common files, etc. between this block and any other block should be stated.

This structuring by functional blocks will make the program easier to modify and maintain. Thus if the program is having trouble in the selection of a function, only the block containing that function need be examined. If you desire to change the format of the powerfail summary report, you again need to modify only one block. A necessary feature of the program blocks then is that they

must be relatively independent. This will allow you to make a change in one block without affecting the function of the other blocks.

In certain cases it is necessary that two or more blocks will have to have a certain amount of interconnection. In the above example, for instance, the block that reads the logfile records and accumulates summaries must pass those summaries to the block that prints them. Also, if any block encounters an error that is being handled by the errors block then certain information about that error must be passed. In this case the functional blocks can not be 100% independent. In order to maintain the desired degree of maintainability to the program all that need be done is to ridgidly define the interface between the blocks. This might take the form of describing the parameters passed between subroutines or the layout of the program common areas. It becomes important to also include a description of just what values each block can modify in these communications areas. Careful attention to detail in defining the use of these interconnection areas will save untold problems later in the program life. I strongly suggest that this documentation take the form of comments within that actual source code rather than as a separate document whenever possible. This will insure that it is always carried with the program and not misplaced or lost. Once the usage of the interconnection areas has been defined and the program written, it should be strongly discouraged that any programmer be allowed to redefine them without carefully examining the entire program for consequences. Any such redefinitions should also be noted alongside, not in place of, the original definitions. In this way any problems of interconnection areas being wrong can easily be traced to the offending block.

PROGRAM SUB-BLOCKS

Within a program block the program should be broken up into sub-blocks. Each sub-block should be a single entity that performs only one operation. The size of the sub-block should be such that it can be easily comprehended without a great deal of effort. The ideal situation is where a sub-block can be quickly scanned and verified that the operation is actually performed correctly. From practical experience this relates to from one half to one page of source code.

The sub-block should begin with a comment that states the operation to be performed. Again, this sub-block definition statement is important in that is serves as the guiding rule for the sub-block. The function of the block should be reflected by simply reading the sub-block definition statements.

THE RULES OF BLOCKS

The block or sub-block is a special animal. In order to be useful several rules must be rigidly followed. Failure to do so may result in a program that only has the illusion of being properly structured. The rules are:

1). The only place for entry into the block's code is at its beginning.

2). The only place for exit from a block's code is at its ending. (The only exception should be an error exit ESCAPE)

3). Only certain easily recoginzable programming constructs will be allowed within a block. This list includes but is not limited to:
   a). straight line code (statement follows statement)
   b). if _ then _ else
   c). looping (DO UNTIL, DO WHILE, WHILE DO, REPEAT n TIMES, DO FOREVER, etc.).
   d). case (execute exactly one of the following)
   e). escape (escape a loop prematurely or error exit)

4). Certain programming constructs are to avoided whenever possible. These include but are not limited to:

   a). Ill defined or ambiguous branches (FORTRAN'S ASSIGNED GO TO, certain cases of COBOL'S PERFORM, FORTRAN'S COMPUTED GO TO while not as bad as an ASSIGNED GO TO should still be avoided)

   b). Backward branches. The major program flow should be strictly from top to bottom. Any backward branching should always be a part of one of the constructs in rule 3 and should be easily recognizable as such. (For example, FORTRAN will not support the DO FOREVER statement but it can be simulated by a single backward branch (GO TO). In this case comment statements should be used to clearly mark the code as a DO FOREVER.

These rules are not all encompassing nor are they inviolable but they do represent a collection of guidelines that I have found lead to programs that will survive the test of time and maintenance. Not all rules can be implemented in all languages but the concepts should apply to any language where the user has control of the program's flow. In cases where the syntax of the language used does not support a construct, the construct can usually be simulated using other features of the language. Always make sure that it is obvious which construct is being used. I

recomment sticking to those constructs listed since that comprise the most commonly understood constructs. One of the objectives of structuring your programs is to make them readable by others.

The reasons for avoiding such constructs as FORTRAN'S ASSIGNED GO TO is that they make it very difficult to interpret the program flow. The destination of an ASSIGNED GO TO is not known until actual program execution time. This makes it very difficult for someone reading your code to determine how your program functions without actually simulating its execution thru exhaustive cases in order to find all possible values of the assigned variable at this point in the program. FORTRAN'S COMPUTED GO TO has some of the same problems in that it is a multidirectional branch but at least the possible targets of the branch are listed in the statement. This statement may be used to construct a CASE statement but it should be clearly marked that this is so. Also note that all CASE statements will eventually return to the same point in order to properly terminate the statement. COBOL'S PERFORM verb can cause much the same confusion as FORTRAN'S ASSIGNED GO TO when it is used to perform different subsets of the same set of paragraphs. This makes it very difficult to determine if execution of the paragraphs will have the desired result. This also violates the rule of always entering a block at the top and exiting at the bottom or it confuses the definition of the blocks.

PROGRAM DESIGN VERIFICATION

The program definition statement should state just what the program is to accomplish. By reading the block definition statements it should be possible to see just what blocks should be involved in any subset of the programs operation. It should be possible, then, to verify if the blocks within the program are capable or satisfying the program definition statement. You should also be able to single out any block that is not necessary to the programs operations.

Now that the program definition can be seen to be satisfied by the block definitions it is necessary to determine if the blocks actually do what their definitions say they do. This is accomplished by reviewing the sub-block definitions. You should be able to follow the blocks structure thru each of the sub-blocks, remembering that all entry into the block is at its top and all exit from its bottom, thus all branching must be between blocks. If the rules of blocks have been followed it should be a simple matter to follow the program thru each sub-block, accepting the sub-block definition as describing properly the action of that block.

Once each block is verified to perform properly, given that its sub-blocks perform properly, all that is necessary is to verify that each sub-block will satisfy its definition statement. This is the first time that we must actually read the source code in the task of determining a programs correctness. If the code within the sub-blocks follow the rules of blocks and is of sufficiently small size then it should be a simple matter to verify that each one properly performs its function.

At this point the program will have a very good chance of performing the program definition as stated at the beginning of the program. If there are any problems then it should be easier to isolate them by placing debugging statements at first the interfaces between the blocks, then at the interfaces between the sub-blocks and finally within the sub-blocks if necessary. In this manner, you can eliminate the majority of code, simply by eliminating the functions and operations that are not in error. If the program has been properly structured then isolating a problem can be the easiest rather than the hardest part of debugging.

By the same token, program modification has alsc become much easier. In order to change a programs operation follow the following steps:

1). Examine the desired change to determine if it fits within the program definition statement. If not then seriously consider not making the change but rather writing a new program as any such change will have major ramifications.

2). Determine which block performs the function that needs to be changed. Alsc consider at this time if the change is in reality a new function, in which case it should have its own block.

3). Locate the sub-block within the block that is performing the operation to be modified or determine where in the blocks program flow a new sub-block should be located.

4). Make sure that the changes will not alter the block definition statement. If so then the entire program structure will have to be examined. If not, then you will only be concerned with this block.

5). If the changes are within a sub-block, make sure that the sub-block definition is still valid after the changes.

6). If you have altered the program flow within the block then make sure that the sub-blocks and program flow will still satisfy the block definition.

7). At this point, if the definition statements are still valid, the program flow is still good, and the rules governing the interconnecting areas have nct been violated, the modification should be properly installed. Now go back and exercise the program thoroughly in and around the affected functions to verify proper operation and to verify that the modification functions properly.

A few words need to be said about commenting and documenting a program. In the past this task was considered as separate from writing the program. Indeed, in many cases, the person writing the documentation was not the one that wrote the program. There are times when this approach is not bad, especially when the documentation in question takes the form of an extensive users manual. It would be a misapplication of talent to have a good programmer tied down for six months after each program, writing a book on how to turn on the system etc. The thing to bear in mind in the case of documentation is that there may be various levels of it. Some levels of documentation are best handled by full time documentors as in the case above, but there is a level of documentation that is the responsibility of the programmer and should always be done by them. This documentation consists of the comments within the program source, and a basic functional description of the program. The main use of this documentation will be by those persons that will have to maintain and modify the program. This means that the task of documenting should not have to be an exhaustive effort as the persons using it will at least be experienced programmers. All that is necessary is to explain the basic flow of the program and the major decisions about the program's design.

In many cases, documentation by the programmer has been put aside until the end of the program developement cycle. This was often justified by citing tight schedules, uncertainty in the programs final state etc. Documenting after the fact leads to skimpy or inaccurate documentation at best and possibly even to no documentation if the schedules are in reality tight.

It is imperitive that the programs internal documentation be written as the program is written. This will insure that it truly reflects the actual state of the program. In attempting to go back after a program is in use and document it, the reasons a certain technique was used is often forgotten. The reasons for choosing one technique over another is one of the most important things to know if you are contemplating a modification to that technique. It would take little actual time for the programmer to add a few comments at the top of a block or sub-block explaining how the block functions and either why the technique was chosen or a brief notice as to under what circumstances it would not be appropriate. This accomplishes a dual function: It informs the next programmer, or indeed yourself if you come back to this program at a later date, as to when to consider modifying the technique. It also

forces you to consider such questions as applicability at the time you are writing the program rather after a technique is already locked into the program. It is far better to discover that a technique will not work in all cases before it is installed rather that having to remove and replace it after a program is finished.

The time to write the internal program documentation, then, is while the program is being written. The place to put the it is as comments in the program source code. This makes sure that whoever is responsible for maintaining the program will always have accurate documentation at hand. At first it might seem that by documenting as the program is written will take too much time. Upon closer examination and by actual trials it can be determined that just the opposite is true. The exercise of documenting as you go will force you into thinking out just what you are doing and consequently keep you from following too many incorrect paths. The end result will be that in the time that you could have written but not documented a program, you can have written and documented a program that has a much better chance of being correct. At the very least the program will be much more maintainable.

What constitutes good commenting within a program? Contrary to the beliefs of a lot of programmers, the more comments does not imply the better the documentation. It is the quality of the comments not the quantity that determines the quality of documentation. In fact, good documentation can be ruined by adding a lot of unnecessary comments and making it difficult to weed out the essential information. I offer to you the following model. It has proven to be useful to me in the programs I write.

1). Variable and Program Names:

Variables, subroutines, and programs should be named in such a way as to make their use as obvious as possible. Thus it might be easier to relate to CUSTOMER(INDEX) in a program rather than to C(I). I realize that certain languages place restrictions on names. These restrictions are being lifted as the languages grow, for instance HP-3000 fortran now allows names to be fifteen characters long rather than the old five character maximum. In any case try to avoid the naming of items within a program by arbitrary or cryptic names. The best way I know to make a program almost unmaintainable is to go thru it and replace all the variable names with a sequence of meaningless names. The names used become an important part of your documentation. If they are chosen properly then you should have little commenting to do within your sub-blocks. By the

serves as more or less a roadmap to find the block responsible for the function you are interested in. At the very least you should include the basic program flow thru the blocks, a brief description of each blocks function and how to locate it in the source code (subroutine name, etc.).

8). Program Modification History:

A short running history of the modifications to the program should be included next. It's this history that will tell you what changes have been made for each revision of the program. Items to be included are:

- revision code after the changes
- date the changes were completed
- a brief description of the changes
- the name of the person doing the modifications if other than the original author.

9). Author and Modifier Names:

A short description of the author(s) and of those persons responsible for making modifications to the program should be listed. This might include the address and/or phone number of the persons or any other information necessary to identify them.

10). Data Definition Sections:

The usage of any data arrays or block interconnection areas should be set off and commented in such a way as to make their intended use clear. This might be as simple as identifying which array is used to buffer data into and out of the program or as complex as to describe in detail the format of an internal communications array. Comment blocking should be used in order to separate the data definition areas for the various operations into blocks, much as the program code areas are blocked. It is easier to understand the data structures in a program if you can concentrate on only the desired subset of them. For example, if the format of the output records needs to be modified then it should be a simple matter to locate the definitions dealing with it and to be assured that all definitions in that area are strictly for that purpose. This will allow changes to be made without affecting other areas of the program and allow no longer needed variables to be discarded.

11). Block Definition Statements:

Each block within a program should begin with a comment describing the operation to be performed within that block. The format of the statement and of its data definitions should be similiar to those for the main program.

12). Sub-block Definition Statements:

Each sub-block should begin with a very short statement of what that sub-block is to perform. For example "This section sorts the input array into ascending order by customer name". Very little additional commenting should need to be done within the sub-block except maybe for a simple clarification comment here and there. If the variable names are properly chosen and the proper code ccnstructs are being used then the sub-block can be almost self documenting.

If a program is internally documented well then most program maintenance and modification can be acccmplished with little more than a source listing. If an additional document is desired it can often be created by excerpting the comments directly from the program, block and sub-block headers.

A word to the modifiers: Once the program has been written following all these guidelines, it is your responsibility to insure that by modifying it you don't invalidate the original program concepts or documentation. A good rule of thumb should be to try to make any modifications such that, in the future, you couldn't tell your code from the original unless it is so marked. This means that you should not force your own programming peculiarities into the program unless they match those already in use. Use the same variable naming scheme as the original author, even if you can think of a better one. This prevents ccnfusion later with having to follow several conventions within the same program. Also try to make your code look like the original. This might mean doing the same indenting and following the same commenting scheme. You should also be responsible for updating any internal documentation that is affected by your changes. This is a good exercise in that it will force you to examine all the areas that your changes might affect. Also don't forget to add a line to the program history records to indicate the changes you have made.

and then resume execution. Again, this is a relatively fast operation (maybe 40 microseconds). If the required segment is not in memory then MPE will suspend the program and make a request to the memory manager in order to have it made present. This process involves reading the segment from disc into memory and might involve swapping some other segment out of memory to make room for it. Disc transfer time on the HP-3000 is fast but is still orders of magnitude slower than direct memory access. Swapping a segment from the disc into memory might take approximately 500 to 2000 microseconds. If this operation is required infrequently then it is not a great burden but if it must occur a great number of times then it can cause a local thrashing condition for this program.

How do you control local thrashing? The main method is to reduce the number of intersegment subroutine calls to the minimum. Lets take the program example given above. In the case of the function involving subroutine A, there is one intersegment call at the beginning of the function and then one at the end in order to return to the main program. This fits nicely in our guidelines. In the second function, however, subroutine B calls subroutine C a great many times. If these two subroutines are in different segments then we might end up in the local thrashing situation. It makes sense, then, to put subroutine B and C into the same segment. That segment will be a little larger than if they were in different segments but the time necessary to call back and forth has been greatly reduced.

Lets summarize what we've learned so far. It is good to reduce the size of our code segments by making a greater number of smaller segments. this reduces the total real memory requirements for the system. If we place two routines that call each other a lot into separate segments then we can cause a local thrashing that wastes time and resources. This means that we have some tradeoffs to evaluate in the segmentation of programs.

Here are some time tested guidelines for the segmentation of program code:

1). Try to make the program segments as small as possible. (4000 words is a good size to shoot for).

2). Minimize intersegment calls between routines even at the expense of larger segments. (Don't worry about numbers as small as 10-20 calls per function but concentrate on those that might be called hundreds of times).

3). Once you enter a segment, try to remain in that segment for as long as possible. This might mean making a copy of a small subroutine that is used by several segments and adding it (under a slightly different name) to each segment that requires heavy use of it.

4). Once you leave a segment, try to remain out of it for as long as possible.

5). Place large and seldom used sections of code into their own routines and their own segments. Error handling routines that contain a large number of error messages are prime candidates (Ascii strings occupy a great deal of code space as compared to machine code). Initialization and light user interaction sections that require messages to be written and read are also good candidates. It is possible that these large sections of necessary code might only be called once, or in the case of error routines, not at all in the program execution. This means that they will not often need to be present in real memory.

How is the best way to segment our sample program ? First of all, we have determined that subroutine A can be a separate segment. Subroutines B and C should be in the same segment since they call each other a lot. Subroutine D is the error handling segment so it can be in its own segment. The main program is rarely executed and could be in its own segment. The final segmentation would then be:

```
SEGMENT 1:    MAIN
SEGMENT 2:    SUB A
SEGMENT 3:    SUB B and SUB C
SEGMENT 4:    SUB D
```

Data segments

MPE currently has the restriction that a programs modifiable area must be in a data segment called a 'stack'. It does not currently allow the data stack to be split into separate data segments. This means that the only control we have over data stacks is in their size. A certain amount of data stack will be required for the program to execute. The areas we can control have to do with data arrays and storage and with dynamic storage.

The storage necessary for data arrays declared in the main program or in the golbal or common areas of a program is always allocated in the data stack. Anything we can do to reduce the size of this storage area will make for a

In summary, the main point I am trying to get across is that the days are gone when programming could be considered a strictly solitary project. A program may be written by only one person, but if it is to survive, it must be written in such a way as to allow others to maintain and modify it. This means that a programmer must give some thought to the basic structure of the program and to organizing its internals in such a way as to make its operation obvious to other programmers.

I have outlined several techniques that might help to achieve this objective. There are probably other techniques that would also be useful. You might, for instance, make it a practice to have another programmer try to 'read' your programs in order to determine if they are properly commented and written. This would accomplish yet another benefit in that it will be a method to share programming techniques and expertise. By having an experienced programmer read the novice's program, he could help the novice to develope the proper standards and techniques. The experienced programmer might also learn a few new techniques from the novice if he keeps his eyes open. By having the novice read the expert's programs, he could learn what techniques should be used in a given situation. The expert will be able to sharpen his documentation techniques by having the novice identify any areas of the program are difficult to follow.

A major point of this whole procedure is to combat the feeling that a program is the exclusive property of one programmer. It should be thought of as belonging to the entire group or to the company. In this way the programmers will not code cryptic programs in order to promote their own mystique. They will instead be writing prorams that will be understandable by others and as such can survive long after they are gone. A side benefit of writing programs that survive should be the writing of more reliable code and raising the standard of data processing in general.