VOL. 7, NO. 1
JANUARY/MARCH 1984

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP INTERNATIONAL
USERS GROUP, INCORPORATED

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

PUBLICATIONS COMMITTEE MEMBERS

CONTENTS

Technical Editor
Dr. John Ray

Editor
Christine M. Dorffi

Technical Editor's Note

The Publications Committee is grateful for the support the IUG membership provided during 1983. Our goal is to publish a variety of articles from a cross section of the membership. The more articles *you* provide, the better your *Journal* will be. Remember, we can only publish what *you* write and send to us. The very best articles are those which let your colleagues know what you do with your hardware and software.

Again, thanks for making your *Journal* better in 1983 and keep up the good work!

HP IUG, 2570 EL CAMINO REAL WEST, FOURTH FLOOR, MOUNTAIN VIEW, CALIFORNIA 94040, (415) 941-9960

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP INCORPORATED

Toward Better COBOL Programs: A Structured Approach

David J. Greer

Robelle Consulting Ltd.
Aldergrove, British Columbia
Canada

INTRODUCTION

In order to write better COBOL programs, it is necessary to understand the concepts of structured programming, and how these concepts relate to COBOL. This article takes a break from the concrete level and explores the theoretical environment in which COBOL works best. This environment is the theory of "structured programming."

Rather than paraphrase ideas that have already been stated eloquently by others, this article will draw short quotations from published sources, which are listed at the end of the article.

STRUCTURED PROGRAMMING

> The software project had to be abandoned, and with it, over thirty man-years of programming effort. "You-know what went wrong? You let your programmers do things you yourself do not understand." How could one person ever understand the whole of a modern software product? (Dijkstra 1972)

Most of us are aware that a "software crisis" exists. Hardware continues to improve by orders of magnitude, but software improves only slightly. In fact, as the problems that we attempt to tackle with computers increase to match the capacity of the latest hardware, the software seems to get worse.

The time required to develop programs is a massive headache to managers in data processing. Projects are usually behind schedule, even when the scheduled completion date seems ridiculously generous. The delivered products are often unsatisfactory to the user, either because they have bugs, or because the user no longer needs what was originally specified.

Programs are often completed containing serious logical bugs. Bugs are difficult to detect during final integration and testing, and they can be costly to the installation if they remain undetected. When they are finally detected during production, bugs are costly to eliminate as well. One of the urgent goals of the software industry must be the elimination of logical bugs, preferably before they get into the programs. At the same time, we have another important goal: faster program development. Surprisingly, recent experience has shown that software is one of the few products where increased quality and reliability are accompanied by lower development costs.

Fortunately for us, the last 15 years have seen a revolution in the theory of programming: this revolution is called "structured programming." It may still be in a state of rapid and explosive development, but certain fundamental truths about programming are being recognized and accepted in the industry.

> A study of program structure has revealed that programs . . . can differ tremendously in their intellectual manageability. A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program . . . I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. (Dijkstra 1972)

> The Fundamental Principle of Structured Programming is that at all times and under all circumstances, the programmer must keep the program within his intellectual grasp. The standard and well known methods for achieving this have been well documented and can be briefly summarized as follows:
> 1. top-down design and construction
> 2. limited control structures
> 3. limited scope of data structures.
>
> The bald truth is that natural language is ill-suited for writing specifications. As a corollary to this theorem, the most concise and intellectually manageable form for the specification of an algorithm is the algorithm itself. (Harkron 1977)

What are the prospects for writing intellectually manageable programs in COBOL? The prospects are mediocre. As will be demonstrated with examples, COBOL provides only some of the structure facilities that are expected of a modern programming language. The mechanisms fo

Structuring the flow of control can be implemented using the COBOL constructs, but it requires good programmer discipline. Regrettably, those for logical structuring of data are even more limited. When the limitations of COBOL are pointed out, we will also provide workarounds for those limitations.

To get the most out of COBOL, you must apply the rules of structured programming. COBOL has more than enough features to completely destroy the structure of a program. We will often stress the necessity of adhering to a well-selected subset of COBOL. Many language features will have to be avoided to keep COBOL a clean programming language.

The remainder of this article will investigate three applications of the "fundamental principle" of structured programming to see how they work in COBOL:

1. *code:* limited control structures
2. *data:* limited scope of structures
3. *programs:* top-down design.

STRUCTURING THE CODE

The price of reliability is the pursuit of utmost simplicity. (Hoare 1981)

One of the primary challenges in programming is specifying the flow of control: what to execute next and under what conditions. Flow of control takes the form of branches, loops, and other logical patterns; but only four control structures are sufficiently simple to be intellectually manageable, while still providing sufficient power to solve all problems.

- Sequence
- Selection
- Test and loop
- Loop and test.

Complex control structures that cannot be directly decomposed into the four basic forms are much more complex and are not intellectually manageable. You cannot grasp what they "mean" by a quick, or even a thorough, glance. Therefore, program structure should be limited to combinations of the four basics.

A programming language should allow a clear expression of the four basic control structures, avoiding syntactic constructs such as GO TO. COBOL provides three of the four forms:

Sequence	(statement, statement, . . .)
Selection	(IF-THEN-ELSE)
Test and loop	(PERFORM-UNTIL)

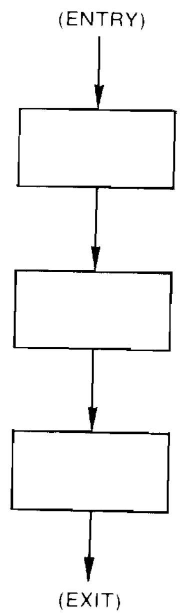
The fourth form loop and test can be simulated in COBOL by initializing the UNTIL condition of a PERFORM to FALSE, before we do the PERFORM:

```
MOVE FALSE TO END-OF-CUSTOMER-FLAG.
PERFORM 20-READ-CUSTOMER
  THRU 20-READ-CUSTOMER
  UNTIL END-OF-CUSTOMER.
```

Basic Structure #1: SEQUENCE

A "sequence" is the first form of program that most of us learn to write. Each statement follows simply after the previous one. When the last statement has been executed, the program stops.

Sequence



A sequence has one entry at the top and one exit at the bottom.

Each box above contains a COBOL statement, such as a MOVE statement, ADD statement, or a DISPLAY statement:

```
MOVE SPACES TO DETAIL-LINE.
ADD 1 TO PAGE-NO.
DISPLAY "END OF PROGRAM".
```

It is difficult to convert a "sequence" into a single unit, but it is usually done by using paragraphs:

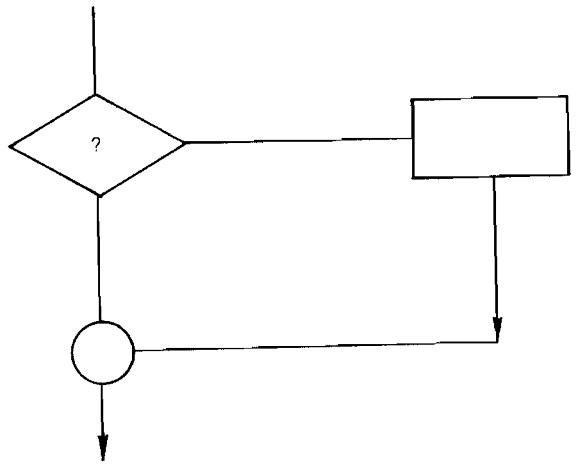
```
90-10-FORMAT-PAGE-HEADINGS.
  ADD 1 TO PAGE-NO.
  MOVE PAGE-NO          TO PR-PAGE-NO.
  MOVE "REPORT TITLE" TO PR-REPORT-TITLE.
```

This new entity must be PERFORMed from within another control structure, such as a selection, test and loop, or loop and test, to produce more complex structures. But these complex structures can always be analyzed into their component structures. This keeps them intellectually manageable.

Basic Structure #2: SELECTION

"Selection" means the selective execution of a single statement, possibly from a group of statements. Execution always "falls through" to the statement following, so there is only one entrance to the structure and one exit.

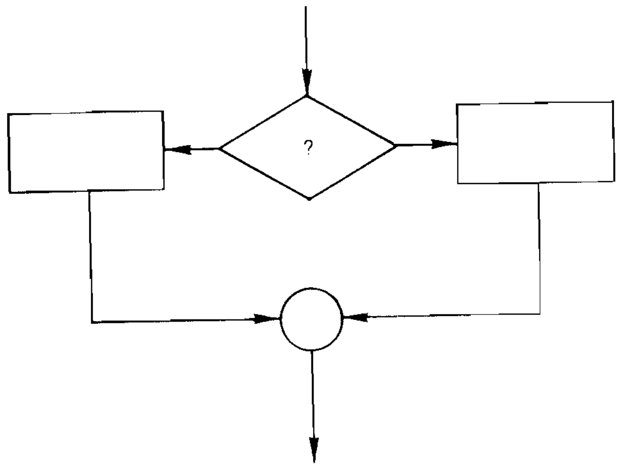
SELECTIVE EXECUTION OF A SINGLE STATEMENT



A test is performed: if the test succeeds, the statement in the box to the right is executed; if it fails, the statement in the box is not executed. This structure is represented in COBOL through the IF-THEN statement:

```
IF expression THEN statement
IF DB-END-FILE THEN
   MOVE TRUE TO END-OF-CUSTOMER-FLAG.
```

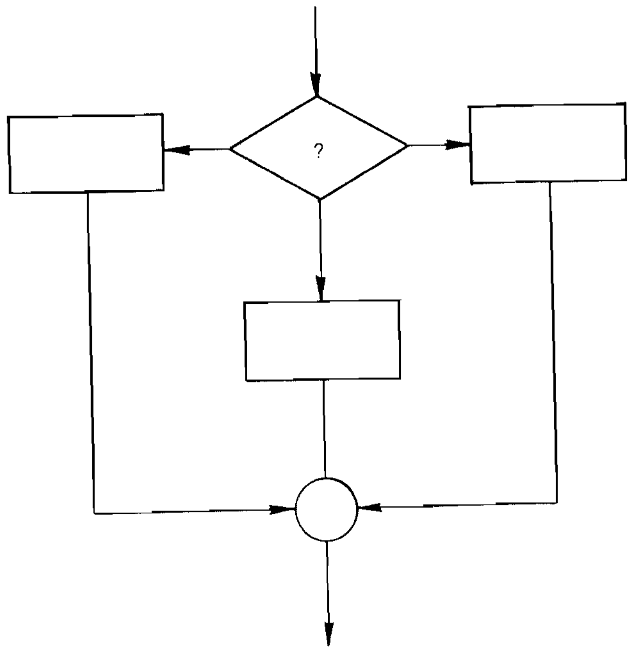
SELECTION FROM TWO ALTERNATIVES



Selection from two choices is implemented in COBOL through the IF-THEN-ELSE statement:

```
IF expression THEN statement
ELSE statement
IF DB-END-FILE THEN
   MOVE TRUE TO END-OF-CUSTOMER-FLAG
ELSE
   PERFORM 20-REPORT-CUSTOMER
      THRU 20-REPORT-CUSTOMER-EXIT.
```

SELECTION AMONG MANY ALTERNATIVES



Selection from among many alternatives is implemented in COBOL through a sequence of IF-THEN-ELSE statements that are *not* nested:

```
IF case-1 THEN
   statement-1
ELSE
IF case-2 THEN
   statement-2
ELSE
. . .
ELSE
IF case-n THEN
   statement-n
ELSE
   impossible case.
IF IN-ADD-TRANSACTION THEN
   PERFORM 20-ADD-TRANSACTION
      THRU 20-ADD-TRANSACTION-EXIT
ELSE
IF IN-DELETE-TRANSACTION THEN
   PERFORM 30-DELETE-TRANSACTION
      THRU 30-DELETE-TRANSACTION-EXIT
ELSE
IF IN-CHANGE-TRANSACTION THEN
```

```
    PERFORM 30-CHANGE-TRANSACTION
      THRU 30-CHANGE-TRANSACTION-EXIT
ELSE
    DISPLAY "Impossible transaction".
```

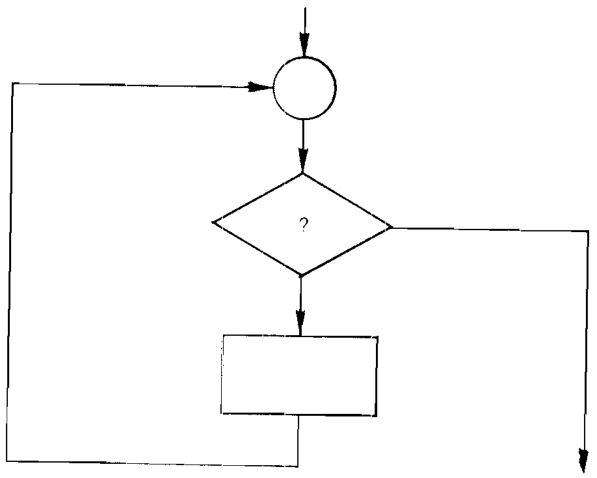
The "impossible case" is interesting. If we write our programs correctly, it should never occur. But since we know that we all make mistakes, including the "impossible case" makes sense. In actual practice, the "impossible case" usually turns up an embarrassing number of times.

In all variations of selection, as in all four of the basic control structures, there is one entry and one exit to the structure.

Basic Structure #3: TEST AND LOOP

The "test and loop" structure performs a test and falls through if the test fails. Should the test succeed, a statement is executed, after which the test is applied again. This process continues until the test fails. There is no way to get out of the loop except by having the test fail (that is important - you cannot jump out).

Test and Loop



The "test and loop" structure is implemented in COBOL through the PERFORM statement:

```
PERFORM procedure-name
  UNTIL expression

PERFORM 20-READ-CUSTOMERS
    THRU 20-READ-CUSTOMERS-EXIT
  UNTIL END-OF-CUSTOMERS.
```

Of course, the "test and loop" can be nested within the "selection" or "sequence," and vice versa, to produce more complex structures:

```
IF D-SALES-EXISTS THEN
    MOVE FALSE TO END-OF-D-SALES-FLAG
    PERFORM 30-REPORT-D-SALES
        THRU 30-REPORT-D-SALES-EXIT
      UNTIL END-OF-D-SALES.
```

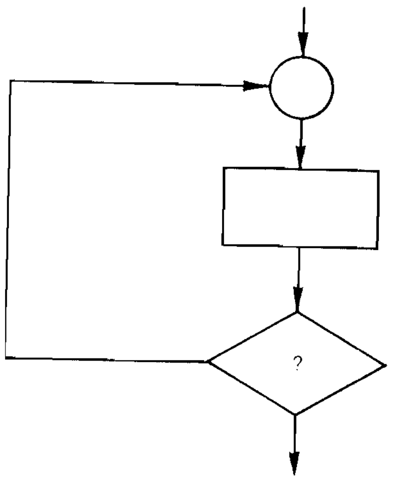
In COBOL, care must be used when nesting control structures. Any extraneous period ends the nesting of the structures. When many statements are being nested, separate them into another paragraph or section and PERFORM them. Compare the statements below with those above. The period changes the meaning entirely, and the nesting is incorrect.

```
IF D-SALES-EXISTS THEN
    MOVE FALSE TO END-OF-D-SALES -FLAG.
    PERFORM 30-REPORT-D-SALES
        THRU 30-REPORT-D-SALES-EXIT
      UNTIL END-OF-D-SALES.
```

Basic Structure #4: LOOP AND TEST

The "loop and test" structure is similar to the "test and loop" structure, except that the test is done after the statement is executed instead of before. The loop statement is always guaranteed to execute at least once.

Loop and Test



The "loop and test" structure is implemented in COBOL by initializing the terminating condition to FALSE, and by using the PERFORM statement.

```
initialize expression to false
PERFORM procedure-name
  UNTIL expression

MOVE FALSE TO END-OF-CUSTOMER-FLAG.

PERFORM 20-READ-CUSTOMER
    THRU 20-READ-CUSTOMER-EXIT
  UNTIL END-OF-CUSTOMER.
```

Note that the initialization does not have to take place exactly before the PERFORM statement, but it ALWAYS should. Doing the initialization right before the PERFORM statement alerts the reader to the fact that this is a "loop and test" structure, instead of a "test and loop" structure. This is an area that requires good programmer discipline.

Structures to Avoid

COBOL has other control structures beyond the four basic ones, but they should be avoided. Control Structures to avoid: GO TO, ALTER, STOP RUN.

THE GO TO STATEMENT

GO TO is universally banned in structured programming. Here's why: GO TO can be used to build the basic logical structures and *many others*. Because the temptation to build other structures is so strong, GO TO should never be used at all.

Programmers usually turn to GO TO when they are revising existing code to handle new exceptions: "Restructuring this code is too much work, but if I just insert a GO TO here, I think it will do what I want." One key to eliminating the need for GO TO statements is the judicious use of LOGICAL flag variables.

```
IF UPDATE-TRANS THEN
   CALL DBUPDATE USING BASE, DSET, MODE,
      STAT, LIST, BUF
   IF STAT NOT EQUAL 0 THEN GO TO DB-ERRS.
   ELSE NEXT SENTENCE
ELSE
   CALL "DBPUT" USING BASE, DSET, MODE,
      STAT, LIST, BUF
   IF STAT NOT EQUAL 0 THEN GO TO DB-ERRS.

DB-ERRS.
   CALL "DBEXPLAIN" USING STAT.
   DISPLAY "Database error - call sys manager".
   STOP RUN.
```

This piece of code may have occurred as follows: originally, the code consisted of only a DBPUT, plus the error handling; then, a DBUPDATE was added. Rather than duplicate the error-handling statements, the programmer inserted a label and a GO TO (a false economy). Adding a local flag (FAILURE-FLAG) would have allowed the error-handling statements to be shared by both calls without a GO TO:

```
MOVE FALSE TO FAILURE-FLAG.
IF UPDATE-TRANS THEN
   CALL "DBUPDATE" USING BASE, DSET,
   MODE, STAT, LIST, BUF
   IF STAT NOT EQUAL 0 THEN MOVE TRUE TO
   FAILURE-FLAG.
   ELSE NEXT SENTENCE
ELSE
   CALL "DBPUT" USING BASE, DSET, MODE,
   STAT, LIST, BUF
   IF STAT NOT EQUAL 0 THEN MOVE TRUE TO
   FAILURE-FLAG.
IF FAILURE THEN
   CALL "DBEXPLAIN" USING STAT
   DISPLAY "Database error - call sys manager"
   STOP RUN.
```

In this particular case, an even better solution would be to centralize the error handling into a single module:

```
99-FATAL-ERROR       SECTION.
   CALL "DBEXPLAIN" USING STAT.
   DISPLAY "Database error - call sys manager".
   CALL "MISQUIT" USING STAT.
99-FATAL-ERROR-EXIT.   EXIT.
```

THE ALTER STATEMENT

ALTER causes the object of a GO TO to change at runtime. As if the GO TO statement was not harmful enough, the ALTER statement guarantees that your programs can NEVER be debugged or maintained. If you have any programs that use the ALTER statement, throw them away; they will cause a major application disaster.

The ALTER statement is dangerous because a walk-through of the COBOL code is almost impossible, since the object of a GO TO can be changed dynamically. A program that uses ALTER *must* use the GO TO. Every rule of structured programming is broken when using the ALTER statement.

THE STOP RUN STATEMENT

STOP RUN causes an immediate termination of the program. A COBOL program should be well structured; it should have one entry point and one exit. The STOP RUN is equivalent to a loop with more than one exit.

An example of the damage caused by STOP RUN is when it is used to terminate on a fatal error from a sub-system such as KSAM. In some versions of KSAM, terminating a program without closing the open KSAM files leaves the KSAM files in an ABORT state. It is extremely difficult to go back and eliminate each and every STOP RUN statement.

There is one exception to the one-exit-point rule in a program. There should be a fatal exit point that is PERFORMed when a call to IMAGE fails for an unexpected reason. This module should print the reason for the IMAGE failure and call the intrinsic QUIT, which notifies the operating system that an ABNORMAL termination has occurred.

STRUCTURING THE DATA

For a well-structured program, the flow of data is just as important as the flow of control. In order to keep a progra within your intellectual grasp, it is essential to know at all times where and when the values of each variable are used or redefined. There are a number of guidelines that lead to well structured data:

- limit the scope of variables
- explicitly declare all variables
- use meaningful data names
- use hierarchical data structures.

Limit the Scope of Variables

One method of making it easier to grasp what the variables are doing in a program is to limit the area of the program in which the variable can be accessed. This is called limiting the scope. If a variable's scope is limited to a single module, you do not have to bother to look in other modules to see if they are changing it. If a variable is declared globally, it can be altered in any module of the entire program. Unfortunately, in COBOL all variables are global, unless subroutines are used. Variables in subroutines are hidden from the mainline and from any other subroutines.

A common practice that violates this guideline is that of using a global utility variable such as I, ARG, X, or T, for many purposes in a program - subscript, line-length counter, loop index, etc. Each use is local, but it is difficult to tell at many places in the program if or how it is being used. With nested module calls, it becomes quite likely that a global utility variable will be used for two conflicting purposes at the same time. This situation is really the same as the unrestricted use of GO TO.

Use each variable for one purpose only and declare it as close to the use as possible. In COBOL, this must be done by the programmer. Using comments, each module should identify which important control variables are being changed. Each module should only change a small number of variables.

One reason for using subprograms is to hide data from the mainline or from other modules. In addition, if a subprogram is dynamic, it uses the HP 3000 stack space more efficiently.

Explicitly Declare All Variables

My suggestion was to pass on a request of our customers to relax the ALGOL 60 rules of compulsory declaration of variable names and adopt some reasonable default convention such as that of FORTRAN. I was astonished by the polite but firm rejection of this seemingly innocent suggestion: It was pointed out to me the redundancy of ALGOL 60 was the best protection against programming and coding errors which could be extremely expensive to detect in a running program and even more expensive not to. The story of the Mariner space rocket to Venus, lost because of the lack of compulsory declarations in FORTRAN, was not to be published until later. (Hoare 1981)

Fortunately, COBOL requires that all variables be declared prior to use.

Use Meaningful Data Names

Variables should always be given long and meaningful names; COBOL treats the first 32 characters of a variable name as significant. Each variable should have a specific function a purpose that it serves within the structure of the program just as each module should have a function.

The objective of data naming should always be to help make this multiplicity of entities intellectually manageable by reducing the amount of arbitrary information that the programmer must remember. A programmer should be able to deduce what a piece of code does by looking at it (i.e., without searching the entire program).

For example, suppose you plan to index through two tables, a customer array and an inventory array. Even if the two indexing operations are never done "concurrently," the program will be cleaner if you declare two index variables with very specific names than if you declare a single index variable called INDEX (or even worse: X).

```
MOVE ZEROS TO X.
PERFORM PRINT-RECORD
  UNTIL X > 10 OR CUST-NAME(X) = SPACES.
MOVE ZEROS TO X.
PERFORM UPDATE-RECORD
  UNTIL X > 20 OR ITEM-NAME(X) = SPACES.

MOVE ZEROS TO CURRENT-CUST.
PERFORM PRINT-RECORD
  UNTIL CURRENT-CUST > 10 OR
  CUST-NAME(CURRENT-CUST) = SPACES.
MOVE ZEROS TO CURRENT-ITEM.
PERFORM UPDATE-RECORD
  UNTIL CURRENT-ITEM > 20 OR
  ITEM-NAME(CURRENT-ITEM) = SPACES.
```

This may require a little more typing, but it will save a lot of searching through the program later. Further, how do you know that you won't want to index both tables concurrently in some future revision of the program?

The example demonstrates how two modules can communicate. The PRINT-RECORD module uses the CUST-NAME (CURRENT-CUST) variable as input. This should be identified by a comment in the PRINT-RECORD module.

By the way, those examples contain another example of inadequate naming. What is the meaning of the constant values 10 and 20 that are used to stop the loops? The code would have much more meaning to the reader if those values were replaced by MAX-CUST and MAX-ITEM or other suitable names.

Many programmers are in the habit of using short, easy-to-type names for variables. This is a mistake. The name should be as long as necessary to describe the variable.

```
E EOF END-OF-FILE END-OF-FILE-FLAG
END-OF-CUST-FILE-FLAG
```

Which of the above communicates the most?

Another technique that can be used to enhance the meaning of names is to use a prefix or a suffix. If you have a group of variables that are used for a related purpose, give them a common prefix or suffix.

Dataset name, buffer, and field-list, with prefix:

```
05 DB-SET-M-PRODUCT   PIC X(10)
   VALUE "M-PRODUCT;".
05 DB-BUFFER-M-PRODUCT.
10 MPR-PRODUCT-DESC  PIC X(20).
10 MPR-PRODUCT-NO    PIC S9(8).
```

Parameters to subroutines (e.g., MISDATE, a date editor):

```
05 DATE-FROMTYPE     PIC S9(4) COMP.
05 DATE-TOTYPE       PIC S9(4) COMP.
05 DATE-AUX          PIC S9(4) COMP.
05 DATE-RESULT       PIC S9(4) COMP.
   88 DATE-OK            VALUE ZEROS.
```

OVERVIEW OF COBOL DATA STRUCTURES

The data structures of COBOL consist of simple variables, arrays (OCCURS), and hierarchical data structures. The basic data types include character, display, binary (COMP), and packed-decimal (COMP-3). All data is global within a COBOL program.

Constants

COBOL has no provisions for named constant values. Instead, variables initialized to a constant value should be used.

```
01 TRUE              PIC X      VALUE "T".
01 FALSE             PIC X      VALUE "F".
01 CUST-MAX          PIC S9(4) COMP VALUE 10.
```

Simple Variables

```
01 PAGE-NO           PIC S9(4) COMP.
01 TOTAL-BALANCE     PIC S9(15)V99 COMP-3.
```

Logical Variables

COBOL has no logical type (i.e., variables that have a true or false value). Instead these must be created by using 88-level declarations.

```
    01 END-OF-CUSTOMER-FLAG  PIC X.
       88 END-OF-CUSTOMER        VALUE "T".
```

You should use 88-level variables as much as possible. Variables that contain status information, input processing options, page counters, and any other variables with a small fixed number of values should be qualified with 88-level names.

LIMITING THE SCOPE OF DATA IN COBOL

COBOL has no way of declaring different scopes for variables. It is up to the programmer to limit the scope of variables, by always limiting the number of variables that are changed in any single module.

Communication between modules is done on an ad hoc basis. The programmer should identify what variables are input, and what variables are output, for each module. Clarity of purpose is enhanced if there is only one output variable. This lack of scope is the most common cause of bugs in COBOL programs.

Using subprograms makes the interface between modules explicit. A subprogram can only access variables of the main program if they are passed as parameters.

Use Hierarchical Data Structures

Up until now I have not mentioned the work "hierarchy," but I think that this is the key concept of all systems embodying a nicely factored solution. . .the only problems that we can really solve in a satisfactory manner are those that finally admit a nicely factored solution. (Dijkstra 1972)

Hierarchical systems seem to have a property that something considered as an undivided entity on one level is considered as a composite object on the next lowest level of greater detail: as a result the natural grain of space or time that is applicable at each level decreases by an order of magnitude when we shift our attention from one level to the next lower one. We understand walls in terms of bricks, bricks in terms of crystals, crystals in terms of molecules, etc. (Dijkstra 1972)

A language should permit hierarchical data declarations. COBOL allows this in the data division:

```
01  CUSTOMER-RECORD.
    05 CUSTOMER-ID-NUMBER          PIC X(6)
    05 CUSTOMER-NAME               PIC X(30).
    05 CUSTOMER-ADDRESS.
       10 ADDRESS-LINE-1           PIC X(16).
       10 ADDRESS-LINE-2           PIC X(16).
       10 ADDRESS-LINE-3           PIC X(16).
    05 CUSTOMER-PHONE-NUMBER       PIC X(12
```

Notice how the layers of definition and abstraction are mad clear in this record. PASCAL has even better hierarchic capabilities than COBOL; PASCAL allows you to assign a name to a structure and treat it as a separate entity. You can declare an array of type CUSTOMER-RECORD.

STRUCTURING THE ENTIRE PROGRAM

We all know that. . .the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called "abstraction". . .the purpose of abstraction is not to be vague, but to create a new semantic level in which we can be absolutely precise. (Dijkstra 1972)

The design of a program and the design of its specification must be undertaken in parallel by the same person, and they must interact with each other. A lack of clarity in specifications is one of the surest signs of a deficiency in the program it describes. (Hoare 1981)

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. (Hoare 1981)

Programming is preeminently a process of abstraction. Therefore, the programmer needs as many abstracting tools at his disposal as he can find. One of the most powerful abstracting tools is top-down design.

Top-Down Design and Construction

Top-down design is a method of applying the abstraction principle to the construction of complex computer programs. The pattern of top-down programming is as follows:

1. Write the main module, freely using undefined terms.
2. Define the undefined terms by writing COBOL SECTIONs, again freely using undefined terms, and by declaring data variables.
3. Repeat this process until all terms have been defined.

Throughout the top-down construction, each module contains only simple, straightforward code, written without GO TO, and using only the four basic control structures. This helps in eliminating logical errors. Programs using the top-down approach can, and should, be tested after each iteration by writing dummy procedures (called "stems") for the undefined terms. Testing is repeated at each level of abstraction. When the program decomposition and coding is done, the program has also been completely tested.

The "Virtual Computer" Concept

At each level in the decomposition process, the code is written as though the computer could understand previous undefined terms. This concept of a "virtual computer" is very powerful and helpful in making complex problems simpler. It allows you to concentrate your efforts on a clear subset of the problem, and it makes that subset of the problem solvable. If you can solve each subset of the problem, eventually you can solve the entire problem.

There are also great advantages to testing and debugging at each level of decomposition. If you do find an error during testing, you only have to look at the changes introduced at this level of abstraction. By definition, the previous levels have already been tested, and they are not normally modified when you go to the next level.

COBOL SECTIONs as an Abstracting Mechanism

A language should provide a way of abstracting complex details so they can be treated as single entities at another level. This is abstraction. It is supported in COBOL via the SECTION and paragraph declarations. You only need to know the SECTION's name and pseudo-parameters, but not how it actually works. Each time that we refer to a module we mean a COBOL SECTION. These modules should be small, about a page of COBOL code, and well defined.

Communication Between Levels

Communication between modules is facilitated by the passing of pseudo-parameters, as described above under Data Structures. Modules should not have too many parameters- never more than you can memorize easily.

It is also a good idea to document which parameters are input values and which are output. A single output parameter is desirable, for the same reason that a control structure should have only one exit: it increases the intellectual manageability of the module.

Top-Down Programming in COBOL

Modules are the key tool in the use of top-down techniques in COBOL. With the abstracting power of modules and paragraphs, you should never have to repeat similar code. You can define a common utility routine and use it in any program that needs it. Nor should you have to write a module that is more than one page long, because you can break it into several modules. At any level of abstraction, each module should perform a well-defined function, and that function should be obvious from the name assigned to the module.

Modules should often return a logical result that indicates success or failure. An advantage of returning logical results is that they remind us to deal with potential failures, no matter how unlikely, in every module.

```
PERFORM 20-GET-CUSTOMER
   THRU 20-GET-CUSTOMER-EXIT
   UNTIL END-OF-CUSTOMER.
. . .
```

```
    PERFORM 20-10-CHECK-CUST-REC.
    IF CUST-REC-OK THEN
       . . .
```

Think of your program as a potential software product. State the objectives of your program as abstractly as you can. Anticipate demands for enhancements by building a general model to solve a class of problems. The abstract solution will be cheaper in the long run.

An Example of a Program Structure

I have developed two sample programs called GOOD and BAD for a workshop in COBOL programming standards. The BAD program contains every mistake that a program could have, while still running. The GOOD version of the program accomplishes the same function, but does so using structured programming techniques. Here is a sample of the GOOD program:

```
Mainline
PROCEDURE DIVISION.
00-MAIN                    SECTION.

  DISPLAY "MIS001-", REVISION-NO.
  DISPLAY "  ".

  PERFORM 10-INITIALIZE
     THRU 10-INITIALIZE-EXIT.

  IF INIT-OK THEN
     SORT SORTFILE ON ASCENDING KEY
     SORT-CUST-ACCOUNT
        INPUT PROCEDURE IS 30-SORT-INPUT
        OUTPUT PROCEDURE IS 40-SORT-OUTPUT.

00-MAIN-EXIT.   GOBACK.
```

The mainline and the initialization module communicate with the INIT-OK-FLAG. The initialization module sets the flag to TRUE or FALSE, depending on whether it succeeds in initializing the program. The control portion of the initialization module is:

```
Initialize
10-INITIALIZE              SECTION.

  MOVE  FALSE              TO INIT-OK-FLAG.

  PERFORM 10-10-OPEN-STORE.

  IF DB-STAT-OK THEN
     PERFORM 10-20-INIT-REPORT
     PERFORM 10-30-INIT-CST-TABLE
     PERFORM 20-GET-DATE-PARAMETERS
        THRU 20-GET-DATE-PARAMETERS-EXIT.
```

The "get date" module sets the INIT-OK-FLAG to TRUE if it succeeds in obtaining an opening and a closing date for selecting records. The input phase of the sort is straightforward:

```
Sort Input
30-SORT-INPUT        SECTION.

  PERFORM 30-10-REWIND-M-CUSTOMER.

  MOVE FALSE    TO END-OF-CUSTOMER-FLAG.

  PERFORM 30-20-GET-M-CUSTOMER
     UNTIL END-OF-CUSTOMER.
```

The get M-CUSTOMER paragraph will set the END-OF-CUSTOMER flag to TRUE when it reaches the end of the input data. Finally, the sort output phase is:

```
Sort output
40-SORT-OUTPUT       SECTION.

  MOVE FALSE            TO END-OF-SORT-FLAG.

  PERFORM 50-REPORT-CUSTOMERS
     THRU 50-REPORT-CUSTOMERS-EXIT
    UNTIL END-OF-SORT.

  PERFORM 95-PRINT-CUST-STATUS-TABLE
     THRU 95-PRINT-CUST-STATUS-TABLE-EXIT.

40-SORT-OUTPUT-EXIT.   EXIT.
```

Enriching the Environment

There is one extension to the top-down method that can be extremely productive in an COBOL environment. You can think of this new method as "enriching the environment." When faced with a difficult programming task, it is easier to design the program if you can take certain details for granted. For example, when writing an accounts receivable system, if we store the records in an IMAGE 3000 database, we can ignore how the records are stored on disc, and how they are indexed and linked together. IMAGE 3000 takes care of those details for us. All we have to worry about is making the calls to IMAGE correctly. The IMAGE routines are not part of the abstract model of the program. They are utility functions, like the MOVE statement of COBOL, that are treated as basic operators at any level in the design.

The utility approach can be applied to many tasks. For example, if your program requires table look-up validation, define a separate support library for performing look-ups. The implementation of this support library is a separate task in top-down design; but once the COBOL subroutines or modules have been designed, coded, and tested, they can be taken for granted in other programs. Since a program need not be concerned with the internal details of the support library, the program can be simpler and more intellectually manageable. Since "enriching the architecture" hides from other programs the details of how the support facility is implemented, you can reprogram the support routines whenever you think of a better implementation.

Another place to apply this approach is in the checking and converting of dates. In all DP systems, you need to h ble to check whether a date is valid and within certain mits, and then you need to convert it to the format used in our database. Rather than code these edits and conversions into every COBOL program repeatedly, you should have a standard utility routine that performs all date functions—MISDATE, for example. Bob Green's SPLAIDS2 package (Green 1981) contains a good example of such a routine.

This MISDATE routine will solve an entire class of problems, not just a particular problem. The problem of dates must be solved for almost every COBOL program. By solving it once, we can thereafter ignore the details of dates, knowing that MISDATE will look after them. If MISDATE does not have all of the features needed for our application, it can be enhanced. But, MISDATE is the only module that needs to be changed.

Structured Programming for Commercial Systems

Commercial data processing projects generate a large number of programs, but only a small number of program classes. There may be 34 different reports to write which have a great deal in common. In fact, a consistent and understandable output for the end user requires a great deal of commonality in the programs.

When you have written one data entry program at an nstallation, you should be qualified to write the next and to nderstand the others. Wherever possible, new programs hould be developed by starting with a house-standard samle program of that class. Modifications are then introduced, in a standard way, of course.

Michael Kohon, formerly manager of several HP 3000 sites in Europe and now president of Datasoft International, has created an implementation strategy that he calls "step-by-step" (Kohon 1982) With his method, you analyze the end users' problems into components, until the largest project can be completed in two weeks. Then you attack the project that the user needs most. In this way, you never have to throw away more than two weeks of work. Also, you elicit end user involvement by delivering products to him every two weeks. You get immediate feedback as to his true needs, which may change from month to month. With "step-by-step," software development is a constant process of refinement and evolution. There are no big surprises, good or bad.

Contrast "step-by-step" with the traditional systems development methods. Normal data processing practice calls for an extensive investigation of the entire problem, followed by a general design. If the user approves the design, typically a large document that he doesn't quite understand, detailed design of the entire project follows. This step generates another large document and another review phase. (No code has been written yet, and the user still does not know how he finished system will "feel" in practice.) Other phases folow in succession: programming, testing, documentation, integration, user training, and production turnover. Often, projects run as long as two years with no discernible output. When the product is delivered and is unacceptable to the user, a tremendous investment must be written off. Even with a large investment in up-front work, the typical installation spends at least as much on maintenance of the existing systems as was spent to develop them in the first place. Thus, in the end, they are forced into an evolutionary approach.

With "step-by-step," data processing problems are broken down into intellectually manageable chunks. Each project is easily within the abilities of the programmer. As a result, if structured programming techniques are applied, logical bugs can be virtually eliminated. The maintenance program and the development program become indistinguishable.

SUMMARY

Programming languages should invite us to reflect in the structures of the program all abstractions needed to conceptually cope with the complexity of the design. (Dijkstra 1972)

The original goals stated for this article were to find a method of programming that would eliminate the bug problem and to improve programmer productivity. The method proposed was structured programming.

By using structured programming principles in COBOL, these goals are not only attainable, they carry with them a surprising bonus: increased program efficiency (i.e., less code, faster execution, less data storage, and better user interface).

ACKNOWLEDGMENTS

This article is based on the chapter, "Structures," from the learning guide *SPL/3000 in a Commercial Installation* (Green 1981) The "Structures" chapter and this article use the same concepts of structured programming. Each gives examples from a specific language, but these concepts reach past the confines of programming languages. I am grateful to Robert Green for his presentation ideas.

REFERENCES

All of these references have insights into the problems of programming and system development. After the material in this article has become familiar, the references should be read for more ideas.

Dijkstra, Edger W. "The Humble Programmer." *Communications of the ACM* 15, no. 10 (October 1972).

Green, Robert M. "SPL 3000 in a Commercial Installation." *SPLAIDS2-Self-Paced Learning Package*. Aldergrove, British Columbia: Robell Consulting Ltd., 1981.

Harbron, Tom. "Structured Programming: State-Descriptive Systems." *HPGSUG Journal* I, no. 4 (1977).

Hoare, C. A. R. "The Emperor's Old Clothes." *Communications of the ACM* 24, no. 2 (February 1981).

Kohon, Michel. "Introduction to Step by Step." *INTERACT*, March April 1982 (Mountain View, Calif.: HP IUG).

Suggestions for Implementing Hierarchical (Bill-of-Materials) Data Structure Using IMAGE

Paul A. Knaplund

Western Data Corporation
Bellevue, Washington

INTRODUCTION

Background

Although many business data structures may be easily represented in IMAGE, there are some structures that require careful attention in their design and use. This paper focuses on one of these more complex information structures: designing and using a Bill-of-Materials (BOM) data structure with DBSCHEMA statements and pseudo-code provided.

Perhaps a definition of a Bill-of-Materials tree is in order: the BOM structure describes the relationships between ordered pairs of entities (read: parts), such that one of the pair is an owner (assembly) and the other a member (component).

Information Sources

Information sources for this paper include over three years personnel experience designing and using IMAGE, IUG proceedings papers, and other database literature. A bibliography is provided at the end of this paper.

DESIGN AND USE OF A BILL-OF-MATERIALS DATA STRUCTURE

Design

There are several processing requirements that the design of a BOM structure should accommodate:

- a part explosion
- a where-used listing
- multiple levels or depth of the tree
- retain information associated with the assembly component relationship, such as quantity used.

It is also desireable to minimize the number of data sets and paths required to support this structure.

Not surprisingly, the solution is both simple and elegant. Utilize a part manual or automatic master data set with two paths pointing to a single component detail data set. Stored in the component data set are:

- assembly part number, path #1 from the part master
- component part number, path #2 from the part master
- quantity required and other data elements that pertain to the intersection of the assembly and component part numbers.

The DBSCHEMA instructions to create this structure look like this:

```
ITEMS:

(1)   ASSEMBLY-PN,            X16;
(2)   COMPONENT-PN,           X16;
(3)   PN,                     X16;
(4)   QTY-REQD,               J02;
```

Lines 1, 2, 3, and 4 define the items to the DBSCHEM processor. Note that ASSEMBLY-PN, COMPONENT-P and PN are identical in their definition. ASSEMBLY-PN and COMPONENT-PN will be used as aliases for the PN field in the COMPONENT-DTL data set. Line 4 is the QTY-REQD (quantity required) item definition.

```
SETS:

(5)   NAME: PART-MASTER,  MANUAL;
(6)    ENTRY:  PN(2),
            << other data, such as part description, etc. >> ;
       CAPACITY: nnnnn;

(7)   NAME: COMPONENT-DTL,  DETAIL;
(8)    ENTRY:  ASSEMBLY-PN(PART-MASTER!),
(9)            COMPONENT-PN(PART-MASTER),
(10)           QTY-REQD;
       CAPACITY: nnnnn;
```

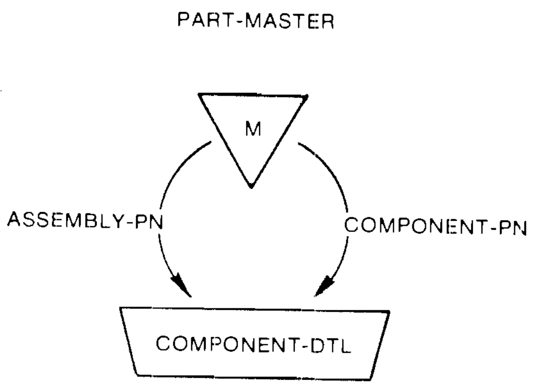
Line 5 names PART-MASTER as a manual master data set. A manual master is used for several reasons: first, it makes sense to store the part description, unit-of-measure, etc. here; additionally, we want to enforce the existence of a PART-MASTER record for every part in the BOM structure.

Line 6 establishes PN as a key element with two paths, both pointing to the COMPONENT-DTL data set.

Line 7 names COMPONENT-DTL as a detail data s

For clarity, the paths from the PART-MASTER are named ASSEMBLY-PN and COMPONENT-PN in the COMPONENT-DTL data set (lines 8, 9).

Using IMAGE pictoral representation, the design looks like this:



This design meets all of the design objectives: it allows for multiple levels (actually there is no depth limit); processes the BOM explosion and where-used reports; and minimizes overhead by employing few data sets and paths.

Creating and Using the Bom Structure

CREATING. There are several constraints that must be adhered to when creating and maintaining the BOM tree, namely:

- no assembly may have itself as a component
- if an assembly (say, a piston) is the immediate owner of a component (piston ring), then the component piston ring may not be an owner at any level above the piston.

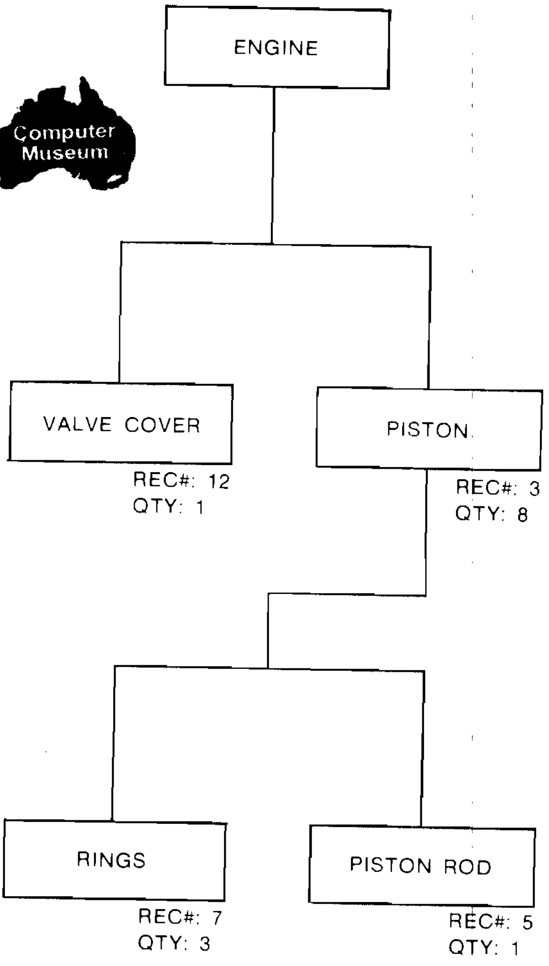
These constraints serve to eliminate the possibility of endless looping when traversing the tree. Optionally, the classic hierarchical constraint may be applied, which is that a component must have one and only one parent asembly above it.

USE. Two important uses of the BOM structure are to produce a where-used report and a parts explosion.

The where-used report is very easily produced. It requires retrieving the chain of assembly part numbers for a given component. In pseudo-code:

```
(1) MOVE "COMPONENT-PN" to ITEM-KEY
(2) MOVE <component key value> TO SEARCH-KEY
(3) MOVE "N" TO END-OF-CHAIN
(4) FIND CHAIN HEAD <<DBFIND, status check >>
(5) GET NEXT IN CHAIN <<DBGET, MODE 5/6, status
    check >>
(6) PERFORM UNTIL END-OF-CHAIN = "Y"
        PRINT <assembly part number>
        GET NEXT IN CHAIN <<DBGET, MODE 5/6,
        status check >>
(7) END-PERFORM
```

The parts explosion is considerably more complex and requires a stack data structure for processing. As the BOM tree is traversed (in preorder fashion), the stack will contain the record number of the brother of the last part processed at each level. The stack will also contain the component quantity requirements for each level.



The BOM tree will be traversed in the following order: ENGINE, VALVE COVER, PISTON, RINGS, PISTON ROD. VALVE COVER and PISTON, RING and PISTON ROD are brothers. VALVE COVER, RINGS, and PISTON ROD are leaves in the tree (without children). Processing the tree is performed in this manner:

1. The chain head for components below ENGINE is located.
2. Since VALVE COVER and PISTON are components below ENGINE, the chain head will exist. Therefore, set (DBGET), the first in the chain.
3. VALVE COVER is now present in the work area. STACK-LEVEL is set to one, and the next record number (DBSTATUS words 9 & 10) is moved to the

NEXT-REC (1). In the above example, the next record number is 3, which refers to PISTON. The current stack quantity required is 1 * 1, or 1.

4. Process the VALVE COVER part.
5. Set up to determine if VALVE COVER has any components. Move VALVE-COVER to the search key, use ASSEMBLY-PN as the path, and perform a DBFIND (similar to step 1).
6. Since VALVE-COVER has components below it, we next check to see if VALVE-COVER has a brother as yet unprocessed. If the next record in the stack is not zero, then a brother awaits further processing. In this case, the next record is equal to 3, which points to PISTON.
7. Issue a directed read against record 3 (DBGET mode 4), using the record number from the stack as the argument. Since PISTON has no other brothers (except VALVE COVER, already processed), the next record is zero so therefore NEXT-REC (1) = 0.
8. PISTON is now present in the work area. STACK-LEVEL is still at one, with the next record being zero. The current stack quantity is 1 * 8 or 8.
9. Process the PISTON part.
10. Set up to determine if PISTON has any components (see step 5).
11. Since PISTON does have components (RINGS and PISTON ROD), add 1 to STACK-LEVEL, and move the brother of RINGS(PISTON ROD's record number, 5) to the next record of the current STACK-LEVEL, which is now 2, or NEXT-REC (2) = 5.
12. Process the RINGS part number.
13. Set up to determine if RINGS has any components below it.
14. RINGS does not have components, so issue a directed read to bring PISTON ROD into the work area (see step 7). There are no more brothers under PISTON, so the next record of STACK-LEVEL = 2 is zero.
15. Process the PISTON ROD part number.
16. Set up to determine if PISTON ROD has any components below it. There are no components below PISTON ROD.
17. Since the next record at STACK-LEVEL = 2 is zero, we have processed all components below PISTON. Decrement STACK-LEVEL by 1.
18. STACK-LEVEL is now 1. Since the next record at STACK-LEVEL = 1 is zero, we have processed all components below ENGINE. Decrement STACK-LEVEL by 1 again.
19. STACK-LEVEL is now less than 1, which indicates that we have finished the traversal.

Pseudo-code to perform the tree traversal:

```
 (1) MOVE "ASSEMBLY-PN" TO SEARCH-PATH
 (2) MOVE "ENGINE" TO SEARCH-KEY-VALUE
 (3) MOVE ZERO TO STACK-LEVEL
 (4) PRINT "ENGINE"
 (5) FIND CHAIN HEAD
 (6) GET NEXT IN CHAIN
 (7) IF CONDITION-WORD = ZERO
        ADD 1 TO STACK-LEVEL
        MOVE NEXT-RECORD TO NEXT-REC (STACK-
          LEVEL)
        ENDIF
 (8) PERFORM UNTIL STACK-LEVEL < 1
 (9) CALCULATE BOM-QTY-REQ
(10) PRINT COMPONENT-PN, BOM-QTY-REQ
(11) MOVE COMPONENT-PN TO SEARCH-KEY-
        VALUE
(12) FIND CHAIN HEAD
(13) GET NEXT IN CHAIN
(14) IF CONDITION-WORD = ZERO
        ADD 1 TO STACK-LEVEL
        MOVE NEXT-RECORD TO NEXT-REC (STACK-
          LEVEL)
(15) ELSE
(16) PERFORM UNTIL NEXT-RECORD (STACK-
        LEVEL) NOT = ZERO AND STACK-LEVEL >
        ZERO
      SUBTRACT 1 FROM STACK-LEVEL
      END-PERFORM
(17) IF STACK-LEVEL > ZERO AND NEXT-RECORD
        (STACK-LEVEL) NOT = ZERO
      READ BY RECORD NUMBER
      MOVE NEXT-RECORD TO NEXT-REC (STACK-
        LEVEL)
(18) ENDIF
(19) ENDIF
(20) END-PERFORM
```

Notes on the preceeding pseudo-code:

1. Use the ASSEMBLY-PN path for searching.
2. Assume ENGINE is the part we want to explode.
3. Initialize STACK-LEVEL to zero. The STACK-LEVEL variable determines what level of the tree is being processed.
4. Print the master part number.
5. DBFIND, on the COMPONENT-DTL. Check status.
6. DBGET, MODE 5 or 6. Again, check status.
7. If ENGINE has components, increment STACK-LEVEL by 1 to enter into the perform loop.
8. Loop until all entries are processed, signaled by the STACK-LEVEL variable reaching zero.
9. BOM-QTY-REQ = CURR-QTY-REQ (STACK-LEVEL) * component qty.
10. Print the component part and required component quantity on the listing. Optionally, indent based on value of STACK-LEVEL.
11. Set up to look for children under current component

12. DBFIND, on COMPONENT-DTL. Check status. (Step 5).
13. See step 7.
14. Increment STACK-LEVEL if current part has components. Move next record from the DBSTATUS area to NEXT-REC in the stack.
15. Otherwise. . .
16. Check to see if the current level has brothers remaining to be processed. If not, back up a level and try again, until the STACK-LEVEL variable is zero.
17. If we found a brother not processed, move that part current in the work area by using a DBGET MODE 4 using the NEXT-REC (STACK-LEVEL) value. Move the next record from the DBSTATUS to NEXT-REC (STACK-LEVEL).
18. End IF for processing additional brothers.
19. End IF for processing additional component levels.
20. End of perform loop.

SUMMARY

Many information structures are disguised Bill-of-Material trees. For example, a general ledger chart of accounts and a company organization chart are similar in many respects to the BOM structure. The given DBSCHEMA statements and pseudo-codes can easily be modified to apply to these cases as well.

One nice feature of the above pseudo-code examples is that they do not affect the currency of the PART-MASTER data set as traversal occurs in the COMPONENT-DTL data set. To process all parts, merely put an outside loop with a serial read (DBGET MODE 2) against the PART-MASTER data set, before entering the inside loops of the examples. A problem arises if you want to print the component description in addition to the component part number when traversing the tree. To do this, save the current record number from the serial read on the PART-MASTER data set before the first DBFIND on the COMPONENT-DTL. Then, for each component, perform a calculated read (DBGET MODE 7) against the PART-MASTER to obtain the component part description stored in the PART-MASTER record. When exiting the loop STACK-LEVEL < 1, issue a directed read (DBGET MODE 4) to reestablish currency in the PART-MASTER, before the next serial read.

BIBLIOGRAPHY

Berquist, Rick. "Optomizing IMAGE: An Introduction." Proceedings of the HPSUG 1980 San Jose Meeting.

Codd, E. F. "Relational Database, A Practical Foundation for Productivity." *CACM* (February 1982).

Date, C.J. *An Introduction to Database Systems*. Third Edition.

Green, Robert. "Overview of Optomizing (On-line and Batch)." Proceedings of the HPIUG 1982 San Antonio Conference.

Kiefer, Karl H. "Database Design, Polishing your IMAGE." Proceedings of the HPIUG 1981 Orlando Conference.

Kroenke, David. *Database Processing*. Second Edition.

Matheson, Wendy. "IMAGE 3000: Designing for Performance and Maintainability." Proceedings of the HPIUG 1983 Montreal Conference.

Rego, Alfredo. "Database Therapy: A Practitioner's Experience." Proceedings of the 1981 Orlando Conference.

Thomas, Ray. "Entity Relationship Analysis." Proceedings of the HPIUG 1983 Montreal Conference.

Implementation of Control Structures in FORTRAN/3000

James P. Schwar

Lafayette College
Easton, Pennsylvania

RECOGNIZED STRUCTURES

The four recognized control structures available for writing structured codes are the:

- IFTHENELSE
- DOUNTIL
- DOWHILE
- CASE.

In FORTRAN/3000 these control structures can be implemented using the logical IF, unconditional GO TO, DO loop, and the computed GO TO.

The IFTHENELSE is of the form IF condition THEN true statements ELSE false statements, where condition is a logical expression. This control structure, as shown in Figure 1, represents a simple decision. In FORTRAN/3000, the IFTHENELSE becomes

```
      IF(.NOT.condition)GO TO S1
      true statements
      GO TO S2
   S1 false statements
   S2 CONTINUE
```

Consider, for example, the calculation of the real roots of F(X)=A*X**2+B*X+C=0 for any A,B,C. The decision to be made is IF B*B-4.0*A*C>=0 THEN calculate and output the real roots ELSE output 'not two real roots':

```
PAGE 0001 HP32102B.01.04 FORTRAN/3000
©HEWLETT-PACKARD CO. 1980

C     ROOTS OF THE QUADRATIC EQUATION
      DATA A,B,C/1.0,3.0,2.0/
      DISCR=B*B-4.0*A*C
C     BEGIN IFTHENELSE
      IF(.NOT.DISCR.GE.0.0)GO TO 10
      ROOT1=(-B+SQRT(DISCR))/(2.0*A)
      ROOT2=(-B-SQRT(DISCR))/(2.0*A)
      WRITE(6,*)'REAL ROOTS ARE',ROOT1,ROOT2
      GO TO 20
   10 WRITE(6,*)'NOT TWO REAL ROOTS'
   20 CONTINUE
C     END   IFTHENELSE
      STOP
      END

 PROGRAM UNIT MAIN' COMPILED

 ****        GLOBAL STATISTICS          ****
 ****   NO ERRORS,   NO WARNINGS        ****
 TOTAL COMPILATION TIME  0:00:01
 TOTAL ELAPSED TIME      0:00:02

 END OF COMPILE

 END OF PREPARE

  REAL ROOTS ARE -1.00000   -2.00000

 END OF PROGRAM
```

ALTERNATIVE IMPLEMENTATIONS

One alternative implementation would be to replace .NOT.-DISCR.GE.0.0 with logical expression DISCR.LT.0.0. A second alternative would be to interchange the position of the true and false statements and use DISCR.GE.0.0 as the logical expression. Neither alternative matches the flowchart logic as well as the implementation that uses the .NOT. condition.

The control structure DO statements WHILE a condition is true is shown in Figure 2. The FORTRAN/3000 implementation becomes

```
   S1 IF(.NOT.condition)GO TO S2
      statements
      GO TO S1
   S2 CONTINUE
```

Consider the calculation of N!, where

N!=N(N-1)(N-2) . . . . (1)

This calculation proceeds from left to right WHILE there is a value > 1. Given N, the FORTRAN/3000 statements are:

```
PAGE 0001 HP32102B.01.04 FORTRAN/3000
©HEWLETT-PACKARD CO. 1980

C     CALCULATION OF N!
      DATA N/10/
      FACTORIAL=1.0
C     BEGIN DOWHILE
   10 IF(.NOT.N.GT.1)GO TO 20
      FACTORIAL=FACTORIAL*N
      N=N-1
      GO TO 10
   20 CONTINUE
C     END  DOWHILE
      WRITE(6,*)'FACTORIAL IS',FACTORIAL
      STOP
      END

PROGRAM UNIT MAIN' COMPILED

****        GLOBAL STATISTICS        ****
**** NO ERRORS,  NO WARNINGS ****
TOTAL COMPILATION TIME    0:00:01
TOTAL ELAPSED TIME         0:00:02

END OF COMPILE

END OF PREPARE

 FACTORIAL IS    .362880E+07

END OF PROGRAM
```

This code yields 1 for N<=1.

The control structure DO statements UNTIL a condition is true is shown in Figure 3. In FORTRAN 3000 this control structure becomes

```
  S1 statements
     IF(.NOT.condition)GO TO S1
```

and the preceding factorial calculation becomes

```
PAGE 0001 HP32102B.01.04 FORTRAN 3000
©HEWLETT-PACKARD CO. 1980

C     CALCULATION OF N!
      DATA N 10
      FACTORIAL=1.0
C     BEGIN DOUNTIL
   10 FACTORIAL=FACTORIAL*N
      N=N-1
      IF(.NOT.N.LE.1)GO TO 10
C     END  DOUNTIL
      WRITE(6,*)'FACTORIAL IS',FACTORIAL
      STOP
      END

PROGRAM UNIT MAIN' COMPILED

****        GLOBAL STATISTICS        ****
**** NO ERRORS,  NO WARNINGS ****
TOTAL COMPILATION TIME    0:00:01
TOTAL ELAPSED TIME         0:00:02

END OF COMPILE

END OF PREPARE

FACTORIAL IS    .362880E+07

END OF PROGRAM
```

The DO loop in FORTRAN 3000 which is of the form

```
    DO S counter=initial value, final value, increment
    statements
  S CONTINUE
```

also implements the DOUNTIL as shown in Figure 4. The calculation of N!, using the DO loop, becomes

```
PAGE 0001 HP32102B.01.04 FORTRAN 3000
©HEWLETT-PACKARD CO. 1980

C     CALCULATION OF N!
      DATA N 10
      FACTORIAL=1.0
      DO 10 I=N,2,-1
      FACTORIAL=FACTORIAL*I
   10 CONTINUE
      WRITE(6,*)'FACTORIAL IS',FACTORIAL
      STOP
      END

PROGRAM UNIT MAIN' COMPILED

****        GLOBAL STATISTICS        ****
**** NO ERRORS,  NO WARNINGS ****
TOTAL COMPILATION TIME    0:00:01
TOTAL ELAPSED TIME         0:00:03

END OF COMPILE

END OF PREPARE

FACTORIAL IS    .362880E+07

END OF PROGRAM
```

The DOUNTIL requires that 0! be treated as a special case.

MULTIPLE PATHS

The CASE structure, as shown in Figure 5, offers multiple paths. The selected path is based on the value assigned to an integer variable. As many paths as needed can be specified. It is common practice to assume that when the integer variable is outside its allowable range — e.g., one to four for Figure 5 — the CASE structure is ignored. The computed GOTO can be used to implement the CASE structure, as illustrated by the following FORTRAN code:

```
C     J=INTEGER VARIABLE
C     J IS PREVIOUSLY DEFINED
      GO TO (10,20,30,40),J
   10 [statements]
      GO TO 50
   20 [statements]
      GO TO 50
   30 [statements]
```

```
      GO TO 50
   40 [statements]
   50 CONTINUE
```

CONCLUSION

In conclusion, it should be noted that these implementations of control structures are similar to those output from the Relational FORTRAN preprocessor for the HP 3000.

REFERENCES

FORTRAN 3000 Reference Manual. Hewlett-Packard Company (1977).

Schwar, James P. and Charles L. Best. *Applied FORTRAN for Engineering and Science.* SRA (1982).

Schwar, James P. and Charles L. Best. "Fortran 3000 and Fortran 77: A Comparison." *Journal of HP General Systems Users Group* (Winter 1980): pp. 14-15

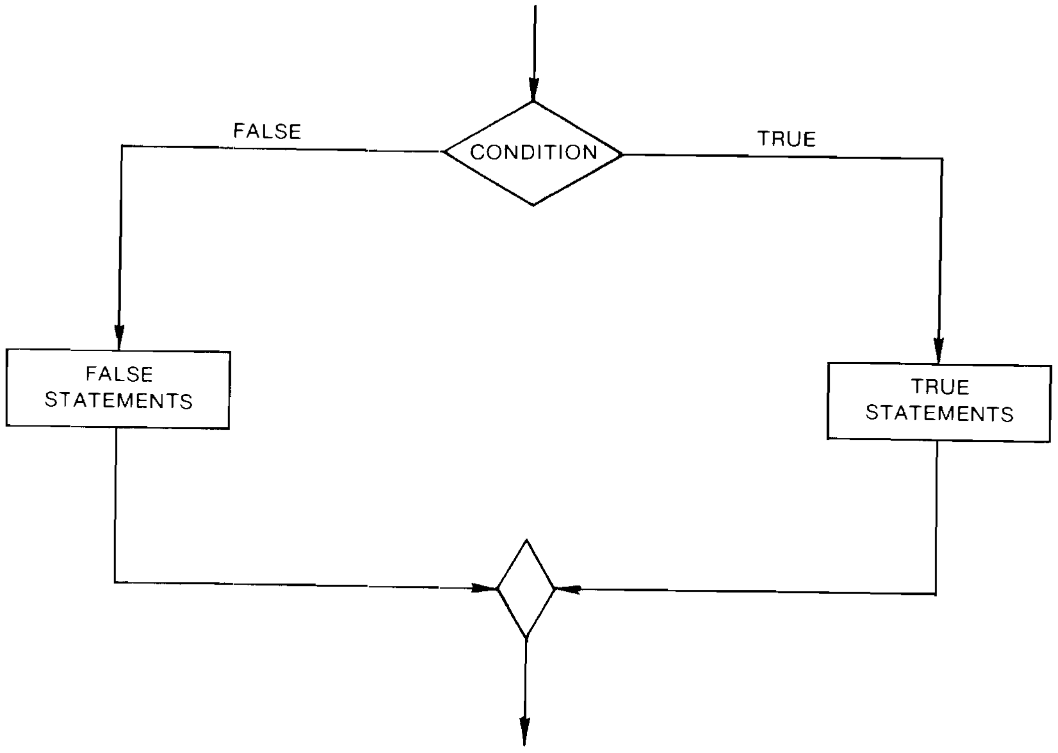
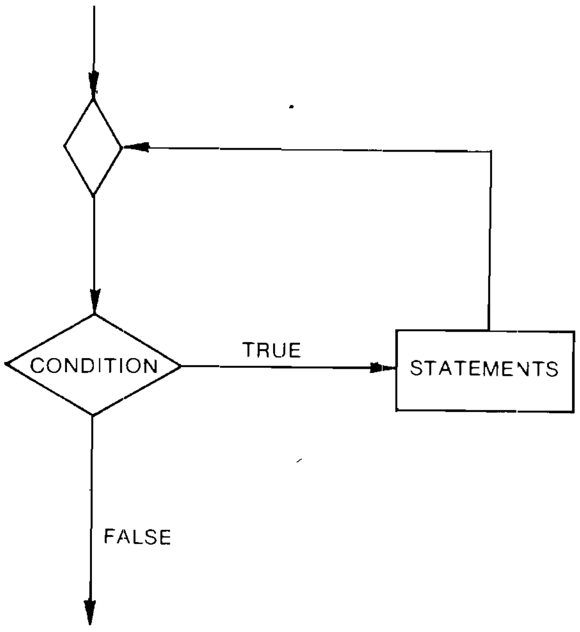


Figure 1. IFTHENELSE Control Structure



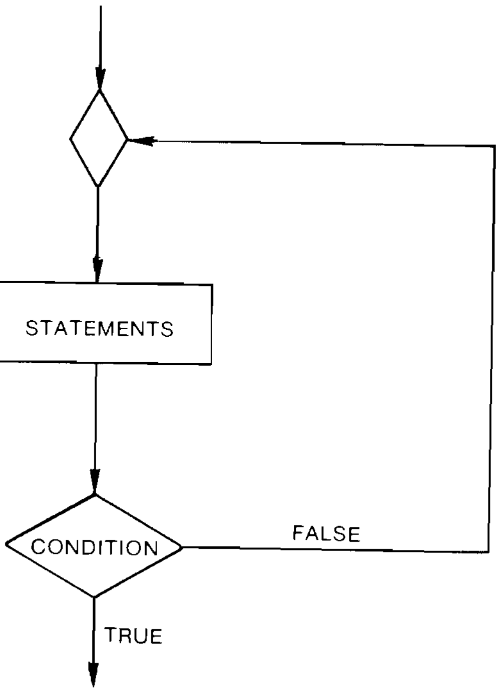


Figure 2. DOWHILE Control Structure

Figure 3. DOUNTIL Control Structure

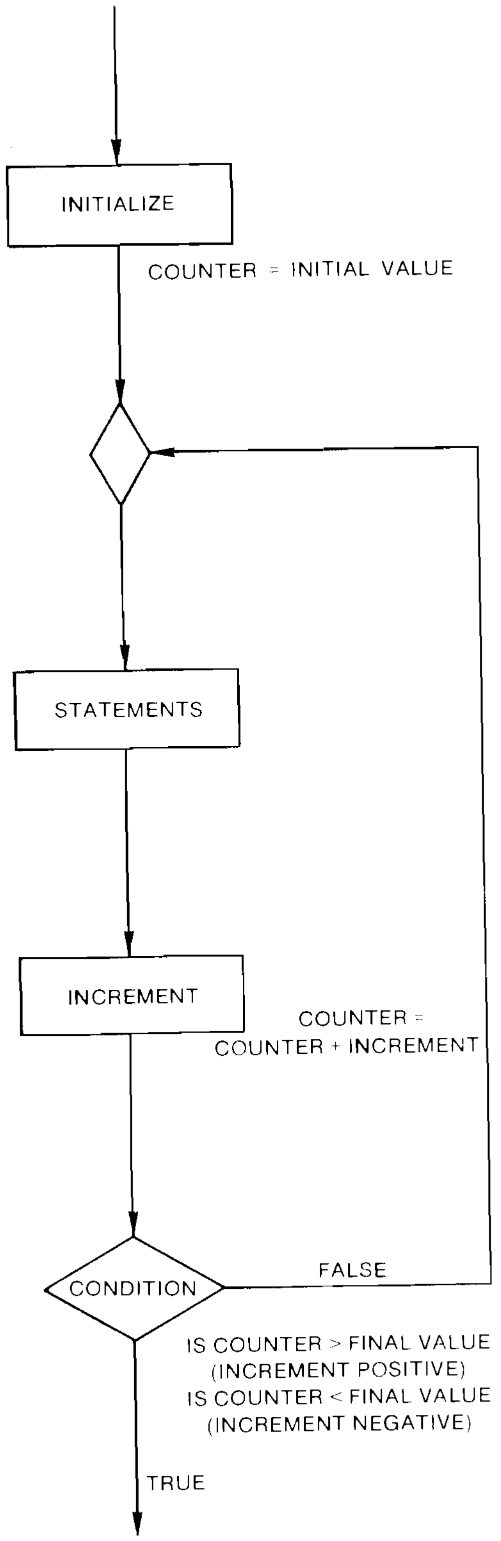

INITIALIZE
COUNTER = INITIAL VALUE
STATEMENTS
INCREMENT
COUNTER = COUNTER + INCREMENT
CONDITION
FALSE
IS COUNTER > FINAL VALUE (INCREMENT POSITIVE)
IS COUNTER < FINAL VALUE (INCREMENT NEGATIVE)
TRUE

Figure 4. DO Loop

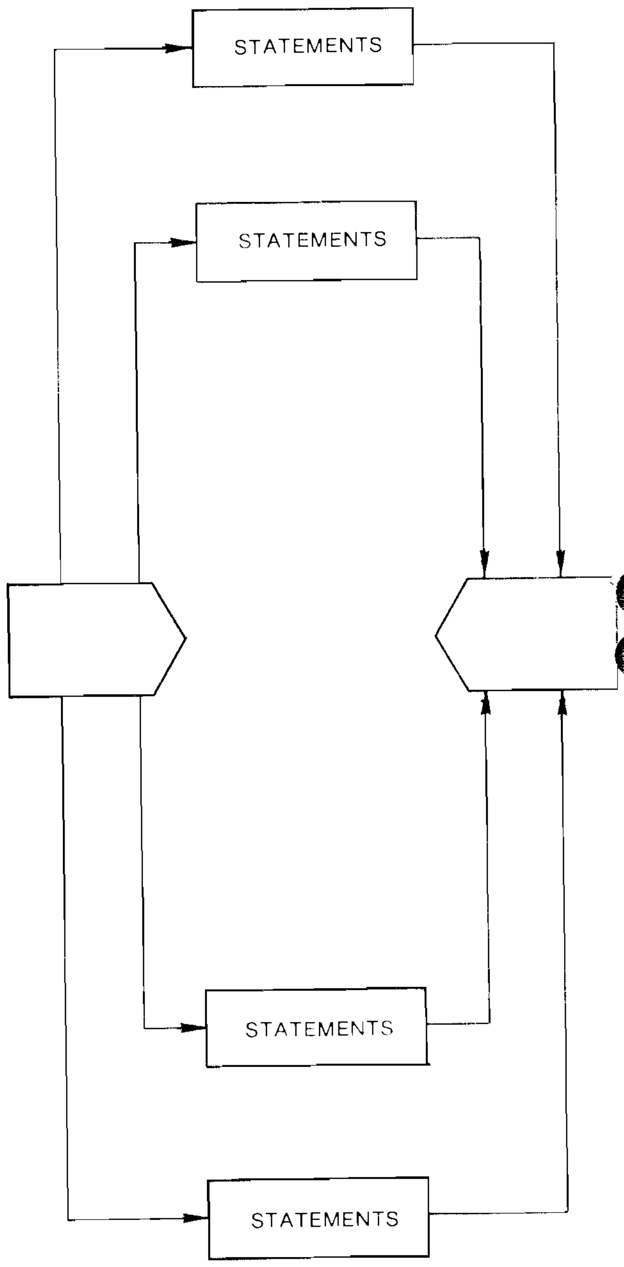


Figure 5. FORTRAN CASE Structure

MPE Disc Cache: In Perspective

John R. Busch
Alan J. Kondoff

Hewlett-Packard Corporation
Computer Systems Division
19447 Pruneridge Avenue
Cupertino, California

ABSTRACT

MPE Disc Caching is a major new performance product for the HP 3000 family of computers. MPE Disc Caching effectively utilizes the excess main memory and processor capacity of the high-end HP 3000 family members to eliminate a large portion of the disc access delays encountered in an uncached system. With disc caching, disc data is potentially available at main memory rather than disc with a probability that increases with main memory size. The MPE disc cache designers here present an overview of the purpose of disc caching, its design approach, its advantages over the alternatives, and its impact on the system price performance of the HP 3000 family.

WHY DISC CACHING?

A storage hierarchy can provide a cost-effective system organization for computer systems. Each successive level of a storage hierarchy uses lower cost, but correspondingly slower, memory components. By retaining frequently accessed code and data in the higher speed memories, the system can operate at speeds close to the access times of the fastest memories but at costs approaching those of the slowest memories. (The price and performance of a computer system is dominated by the organization and management of its storage hierarchy.)

Achievable system performance is a direct function of processor speed and utilization. Processor utilization is limited primarily by its waiting time caused by misses at various levels of the storage hierarchy. Thus, for optimal system price performance, the processor speed and the capacity, speed, and management of the levels of the storage hierarchy must be matched. In order to fully utilize the processor capacity, the system must achieve a sufficiently high probability of finding data when referenced at the highest levels rather than having to go to the lower levels of the hierarchy.

When a low hit ratio at a certain level of the storage hierarchy is causing low processor utilization (thereby limiting achievable system performance), a number of alternatives exist to resolve the problem. These include improving the management policies of the levels, increasing the capacity of the level incurring the low hit rate, speeding up the access time of the next lower level of the hierarchy, and introducing a new level into the storage hierarchy. Cost and technology determine which alternative or combination of alternatives is optimal.

This paper focuses on the innovative solution to the main memory disc bottleneck that MPE Disc Caching provides for the HP 3000 computer family.

The levels of the storage hierarchies of HP 3000 computer systems (Hodor and Woodward 1982) range from:

1. processor registers, to
2. microcode store, to
3. processor cache, to
4. main memory, to
5. discs, and to
6. tapes.

Each of these levels uses memory components that have a significantly lower cost-per-byte but a significantly longer access time.

The higher-end HP 3000 systems currently suffer from low processor utilization in some database and disc I O intensive installations. This negatively impacts the growth capability for customers. The low processor utilization is due to processor idling while misses at the main memory level are being resolved from disc storage. Main memory capacities beyond a few megabytes may provide limited incremental improvement.

The HP 3000 R&D lab teams have been working to resolve this imbalance and propose MPE Disc Caching as the solution. MPE Disc Caching optimally exploits current cost and technology tradeoffs to eliminate the imbalance.

WHAT IS MPE DISC CACHING?

MPE Disc Caching is an optional MPE subsystem that manages retrieval and replacement of disc "domains" or regions in excess main memory. It locates, moves, and replaces disc domains in main memory so that a significant portion of the references to disc storage can be resolved without incurring physical disc access delays.

The MPE Disc Caching policies are fully integrated with the MPE kernel, file system, and I/O system. This allows system performance to be optimized based on current resource availability and workload demand.

Ancillary to MPE Disc Caching, there is a set of external controls, measurement, and simulation tools to allow users to manage and predict the operation of caching on their systems. Operator commands are available to enable or disable caching on a device basis, and to display general caching statistics for a device. Measurement tools have been enhanced to display relevant disc cache usage and performance statistics. A special simulation tool has been developed, which analyzes disc access traces from HP 3000 systems and produces disc cache performance statistics for specified cache memory sizes. With this tool, main memory requirements to efficiently support disc caching can be predicted for existing HP 3000 installations.

MPE DISC CACHE DESIGN APPROACH

The MPE kernel resource management mechanisms and strategies (Busch 1982) provide an efficient, integrated approach to resource management. The MPE disc cache mechanisms and strategies are integrated with those of the kernel, and they exploit the file system's knowledge of file structure and access method to enhance prefetch and replacement decisions for disc domains. Microcode assist is exploited to rapidly locate cached domains in main memory.

The kernel's main memory placement and replacement mechanisms are extended to handle cached disc domains in the same manner as segments. Thus, cached disc domains can be of variable size, fetched in parallel with other segments or cached disc domains, garbage collected, and replaced in an integrated manner with stacks, data segments, and code segments. The relative allocation of main memory between stack, data, code, and cached disc domain objects is entirely dynamic, responding to the workload's current requirements and current memory availability.

When a request is made to access disc information, the list of currently cached disc domains of the specified device is searched using the *linked list search* instruction. If the requested disc domain is present in main memory, the data is moved between the process's data area and the cached copy of the requested disc domain. The process will continue executing without an interruption or process switch.

If the access request is a write, and there is currently a write pending against the specified disc domain, the process's request is queued until the pending write is posted to disc. If the disc domain to be written is not currently cached, an available region of memory is obtained which is used to map the corresponding disc image (i.e., no fetch of the disc domain to be written is required). When the move effecting the write takes place from the process's data area to the cached image of the disc, a post to the disc is initiated. Only the portion of the cached disc image that is modified by the write is posted. After the move to the disc image is performed and the post to disc is initiated, the writing process is allowed to continue running without having to wait for the physical post update to complete. Disc integrity is insured within the operating system and subsystems through serial posting on a global basis of writes with posting order constraints. At the user level, wait-for-post can be specified on a file basis in place of the default no-wait-for-post.

When a request is made to read data that is not currently cached, the fetch strategy uses knowledge of the file blocking, extent structure, access method, and current memory loading to select the optimal size of disc domain to be fetched into memory. The fetch of the disc domain is initiated through the memory manager on the current process's stack without a process switch. The fetch is performed in an unblocked manner so that the requesting process or another process can run in parallel with the cache fetch from disc.

When a process completes referencing a cached disc domain in sequential mode, the domain is flushed immediately from main memory since it won't be needed again. In this way, memory utilization is improved over that achievable with the kernel's approximate least recently used (LRU) replacement algorithm.

With these mechanisms and strategies, MPE Disc Caching significantly reduces the traffic between the main memory and secondary disc storage and significantly reduces delays to read or write disc information. It does so in a manner that is superior to its alternatives when evaluated by cost, reliability, and performance measures. The advantages over its alternatives are discussed in the following section, followed by a discussion of its impact on HP 3000 system price performance.

ADVANTAGES OVER ALTERNATIVES

There are several alternatives to MPE Disc Caching, which could help address the high traffic rate and long delays between main and secondary disc storage on HP 3000 systems. Software alternatives focus on providing localized caching on a subsystem, file, or application basis. Hardware alternatives focus on reducing access time to disc storage or increasing the number of concurrent paths to disc storage.

The software alternatives are all localized and unresponsive to current memory loading conditions. The amount of memory devoted to the caching of a specific file or subsystem would likely be either excessive or insufficient at any given moment, depending on the current memory availability and the current workload demand and priority structure.

Access times can be improved by speeding up the discs, introducing a new level in the storage hierarchy between semiconductor main memory and moving head devices, or caching the discs with a peripheral cache. Access capacity improvements can be accomplished by adding more parallel paths to secondary store with more discs, controllers, and channels.

Speeding up discs involves technology limitations and tradeoffs between performance, cost, and reliability. Access time improvements that compare with the order of magnitude improvement provided with MPE Disc Caching are unlikely at any cost, even with head-per-track approaches.

Introducing a new level in the storage hierarchy that exploits bubbles or CCDs as a gap-filling technology has not proven to be cost effective when compared with exploiting high density semiconductor memory technology in very large main memories.

A peripheral disc cache (Hugershofer and Shultz 1982; Krastins 1982) can be built into the controller or disc or can be a separate system component that front-ends a selected subset of the disc subsystem. There are cost, performance, and reliability trade-offs involved among each of these peripheral disc cache architecture alternatives. MPE global disc caching in the SPUs (system processor unit) is superior to the best achievable cache architecture by all three measures.

Evaluated with respect to cost, a peripheral cache requires additional power, cooling, cabinetry, and electronics, as well as the caching memory that is required for disc caching in both peripherals and main memory.

Evaluated with respect to reliability, the MPE disc cache introduces no new hardware components into the system. The reliability is identical to the uncached system, whereas any of the peripheral cache architectures necessarily degrade system reliability due to their introduction of hardware components. The software firmware complexity of the two alternatives is roughly the same, so reliability degradation due to cache management is comparable. Since the MPE Disc Caching is built into MPE, it has the potential to evolve toward improving system reliability (e.g., automatic extent sparing on write failures). Also, the posting strategy of the peripheral cache is not integrated with the system posting strategy so that a consistent level of integrity is not guaranteed.

Evaluated with respect to performance, the MPE disc cache has clear advantages in access time, access rate, and cache memory utilization. Access to cached disc domains is provided on the current process's stack with a combination of firmware and software. Not even a process switch is required. The access time is on the order of a very few ms, which scales with processor speed. Access to a peripheral cache requires at minimum a trip through the I/O software and interrupt system, a process switch, plus the cache access time of the peripheral cache.

Achievable cache access rate of the main memory cache is roughly the inverse of the cache access time (several hundred accesses per second, scaling with processor speed). This rate is achievable because the main memory cache is a parallel server. When one process encounters a miss on cache, the processor can be applied to another process that can access the cache concurrently with the resolution of cache faults of other processes. This level of parallelism would be very difficult to achieve in any of the peripheral cache architectures. Also, the best case access of any peripheral cache architecture is several times that of the main memory cache. Therefore, even if full parallelism were achievable in a peripheral cache, the best achievable access rate can only be a small fraction of that achievable with the main memory cache approach.

Cache memory utilization of global SPU caching integrated with the MPE kernel is superior due to several factors. The amount of cache memory applied to a cached device is responsive to the current utilization of the device. The size of a prefetched disc domain is tailored to the structure of the data (e.g., the cache mechanisms fetch extents instead of fetching tracks that contain unrelated data or only pieces of the required extent). The replacement policy exploits operating system knowledge of access patterns (e.g., the policy flushes a cached disc domain from the cache memory after sequential reference and on file purging).

The traditional way of addressing the disc bottleneck on a disc-bound system has been to increase disc access capacity by adding more discs, controllers, and channels. This approach is very expensive, and its improvement in secondary store access time and access capacity cannot match that provided by MPE Disc Caching.

IMPACT ON HP 3000 FAMILY SYSTEM PRICE AND PERFORMANCE

MPE Disc Caching provides a significant improvement in HP 3000 family system price performance for a large class of workloads. With the higher-end systems and a large class of HP 3000 application environments, MPE Disc Caching effectively reduces the system cost required to achieve a given performance level. It also significantly increases the system performance, which is achievable for a given cost.

MPE Disc Caching does introduce increments in system cost due to the expense of the optional software product and additional main memory for caching. However, these increases are more than offset by the reduction in system cost due to the ability to exploit the very attractive $/Mbyte advantage of HP's large capacity discs without suffering a significant performance disadvantage.

In HP 3000 systems without disc caching, exploiting the roughly fourfold $/Mbyte advantage of the 793X discs over the 792X discs comes at a significant performance cost. This is due to the effective fourfold reduction in disc access capacity caused by the reduction in concurrent disc servers. Since the disc subsystem is utilized at a much lower rate with MPE Disc Caching, this reduction in disc access capacity has a limited impact on system performance. Second-order cost impacts are also achieved, in that disc maintenance costs are reduced due to fewer drives and lower utilization of the drives.

Achievable system performance for installations with excess processor capacity is significantly higher with MPE Disc Caching than that with uncached configurations. System response times are quicker due to reduced queueing delays, and service times for the resources required to complete transactions. Disc queue lengths and frequency of visits to the discs are reduced. The queueing delays and holding times of system and application locks are significantly reduced, since the disc delay components of the holding times are reduced. Since these delays dominate the response time in uncached systems for many HP 3000 workloads, the impact on system response time due to disc caching can be significant. System throughput is improved due to the reduced response times and reduced contention for resources.

As a workload grows, the upgrade to higher performance family members becomes a very attractive alternati from a price/performance perspective.

CONCLUSIONS

MPE global disc caching in the SPU provides an innovative solution to overcoming the huge access time gap between main memory and disc storage, and the long database semaphore queueing delays caused by this gap. It provides a solution that exploits the current cost and performance trade-offs of memory and processor technologies. It is superior to the alternatives when measured by cost, performance, or reliability. MPE disc caching provides significant system price/performance improvements across the HP 3000 family, as well as an attractive upgrade for existing HP 3000 installations.

REFERENCES

Hodor, Ken M. and Woodward, Malcolm E. "A High Performance Memory System With Growth Capability." *Hewlett-Packard Journal* (March, 1982), pp. 15-17.

Busch, John R. "The MPE IV Kernel: History, Structure and Strategies." *Proceedings of the HP 3000 International Users' Group* [illegible] Orlando, April 27-May 1, 198[illegible]. Reprinted in *Journal of the HP 3000 International Users Group, Inc.* [illegible] 3-4 (July-December 1982).

Hupcheck, William and Schrader, [illegible] "Cache Buffer for Disc Accelerates Minicomputer Performance." [illegible] Feb. [illegible] 198[illegible]

Krastins, Uldis "Cache Memories Quicken Access to Disk Data." *Electronic Design* May 13, 1982, pp. 41-44.

HP IUG BOARD OF DIRECTORS

Journal
HP International Users Group, Inc.
2570 El Camino Real West, Fourth Floor
Mountain View, California 94040
U.S.A.

BULK RATE
U.S. POSTAGE
PAID
PERMIT NO. 382
MOUNTAIN VIEW, CA
and Other Locations