Programming for MPE XL Performance

Dave Trout
Hewlett-Packard Company
2 Choke Cherry Road
Rockville, MD 20850 USA

Introduction

Application performance issues are fairly well understood these days for the existing MPE V/E systems environment. With the change in architecture presented by HP's new 900 Series HP3000 systems and MPE XL, it is helpful to re-examine application design philosophies and techniques to insure optimum performance in the new environment.

Rather than attempt to present an exhaustive list of do's and don't's, a specific set of MPE XL features and techniques will be reviewed within the framework of improving application performance. Use of Native Mode (NM) versus Compatibility Mode (CM), extended addressing capabilities, and user mapped files will be examined. Coding examples and methods will be presented where appropriate to provide a background for discussion and elaboration.

With the understanding that application performance will generally improve as MPE XL tuning continues, relative performance comparisons will be presented to help quantify coding effort versus the resulting performance benefits.

Performance Opportunities

Because the 900 Series HP3000s are part of the overall commercial family of HP3000 systems, a great deal of attention was given to designing MPE XL for compatibility. The very high level of compatibility achieved has been well documented and has contributed to a constant stream of successful customer migrations.

At the same time, HP Precision Architecture (HPPA) offers a number of new opportunities for improved performance and productivity which go beyond the older HP3000 system capabilities. Since a number of these new features are used in the programming environment, it seems best to illustrate their power and benefits by actually going through a programming problem and the solutions available in MPE XL. This discussion will therefore follow the development of a program which will illustrate the benefits of NM versus CM, extended addressing capabilities, and user mapped file techniques.

The two essential goals are: 1) show *how* these new techniques are used, and 2) demonstrate that they provide *improved performance* over techniques used in the past.

The Problem

To facilitate comparisons of the various techniques, a single programming problem is desired which can be easily "modularized" in the solution. This allows leverage of common program procedures and makes it easy to "drop in" different techniques by simply changing specific program procedures.

A Table Lookup Simulation was chosen as a good problem to work on. Table lookup techniques are widely used in computer operating systems and applications and the design center is usually *fast and efficient data retrieval*, or in other words, high performance. In many applications, the requests for retrieval of entries are spaced randomly within the table and the table itself may be fairly large. These attributes tend to work against the requirement for high performance (as far as the typical operating system is concerned), so a table lookup simulation seems particularly good for determining the real power of MPE XL and the specific features to be examined.

Figure 1 shows a graphical look at the programming problem being posed.

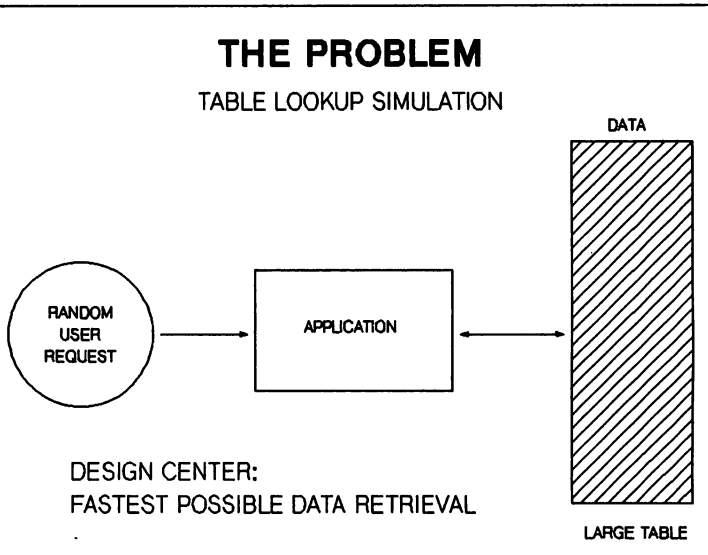

Figure 1

The design parameters for the Table Lookup Simulation program can be summarized as follows:

- The table itself will be very large and must be at least several times larger than can be represented in a single data segment on MPE V/E.
- Table entry retrieval must be extremely fast and efficient. It is desirable that the table be accessed as a memory resident structure.
- During the simulation, random entries in the table will be accessed.
- The program and associated files will be designed so that identical simulations can be run, each showing a different programming technique to do the actual table lookup.
- To gain a good performance comparison of the various techniques, a large number of table entries will be accessed during the simulation.

The Solutions

For this programming problem, there is really only one solution in the MPE V/E environment--use Extra Data Segments (XDS) to represent the table. (Note: Throughout this paper, it is assumed that a basic understanding of the Extra Data Segments capability of MPE V/E and MPE XL already exists. Our purpose here is not to show how to use the XDS intrinsics but how to use the new MPE XL features which essentially replace those intrinsics in functionality.)

In the MPE XL environment, we have four solutions to choose from:

- Table is multiple XDSs (program in CM).
- Table is multiple XDSs (program in NM).
- Table is a large array (program in NM).
- Table is a user mapped file (program in NM).

To duplicate the progression that a current MPE V/E programmer might take in migrating an existing application, we will first examine an implementation of the solution program using Extra Data Segments, then change the necessary procedures to use a large array (extended addressing), then finally change the program once again to define the table as a user mapped file. Performance improvement at each step will be noted and summarized at the conclusion.

NOTE

To illustrate the various coding techniques being discussed, Pascal code fragments will be shown and referenced. These fragments are taken from a fully tested and executable program, however code which is not germane to the discussion has been left out to improve clarity. At the end of this paper is a complete listing of the actual Pascal program used for the "mapped file" version of the Table Lookup Simulation.

The Test Environment

All tests were run on a Series 930 configured with 64MB of memory and 4-7937 disc drives. The MPE XL version was A.01.10. To provide consistency in the performance comparisons, each test was run in a dedicated batch job environment which included the actual simulation itself and performance data collection programs. Each run used exactly the same data and script files (described below). Performance data was collected with XLDCP and AMT for a 15 minute period for each test. The essential performance indicators for the simulation were defined as follows:

- Elapsed time for the simulation.
- CPU time used during the simulation.
- Switch rate during the simulation (from/to NM/CM).
- % of CPU time in NM.

The elapsed time and CPU time measurements were built into the simulation itself using the TIMER and PROCTIME intrinsics. Switch rate and % of CPU time in NM were taken from XLDCP and AMT logfiles. All graphs shown here were generated from the XLDCP logfiles.

Data Structures

Before we can begin discussing the first programming example, the essential data structures need to be defined.

The table itself is defined as having 65536 entries, each entry 16 bytes in length. (Each entry could also be described as 4 MPE XL words, 8 MPE XL halfwords, or 8 MPE V/E words. The term *halfword* is used occasionally in MPE XL reference manuals to describe a 16 bit entity, as opposed to *word* which describes a 32 bit entity.)

The table entries will be accessed by index range 0..65535. With 65536 entries, each 16 bytes in size, we therefore have a table which is 1,048,576 bytes (or 1MB) in size. This is obviously quite a bit larger than can be represented in any single data structure in MPE V/E.

For the purposes of this simulation, each entry in the table is very simply defined (a real application would of course include useful data in the table). Each entry consists of two subfields: 1) A 32 bit integer value in the first word which is defined as the index number for that entry, and 2) The "DASCII" version of subfield 1 in the remaining 3 words (left justified). It will become apparent later in this discussion why this data format was chosen. Figure 2 below shows the table structure.

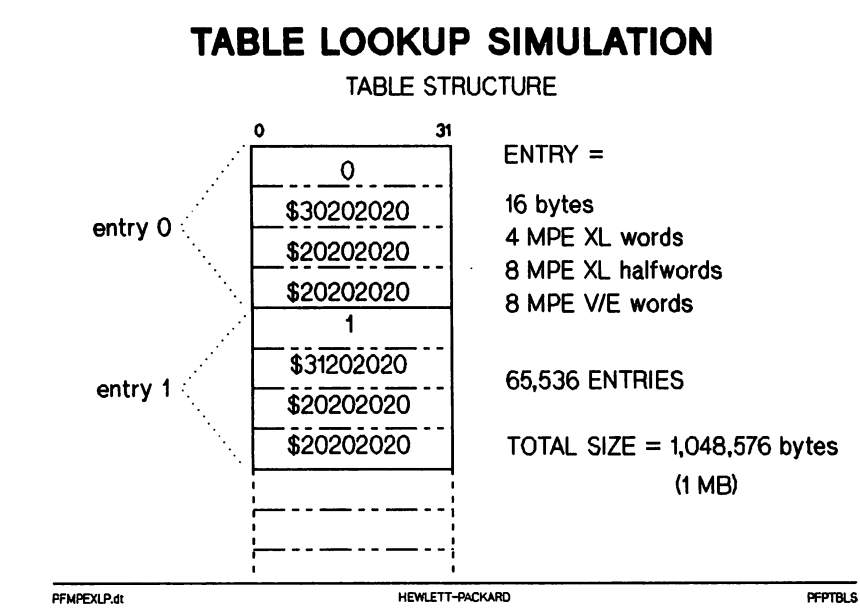

Figure 2

In our Pascal program we can define the table data structure as follows:

```
TYPE
  table_entry_type = record
    f1 : integer;
    f2 : packed array [1..12] of char;
  end;
```

Since one objective of the simulation is to access the table in memory, in the first two program examples (XDS and large array) it will be necessary to load the table from a flat data file on disc before the actual simulation of table lookups can begin. This is of course done only once during the program at the very beginning. A separate program was written to load the flat data file with the table data as described above. Our Pascal program would define the table entry and the source data file as follows:

```
VAR
  table_entry : table_entry_type;
  table_file : file of table_entry_type;
```

To simulate accessing the table, a large script file was created. Each record in this script file is an integer which has a randomly distributed value in the range 0..65535. As this file is read sequentially, each record's value will be used as the index for finding and processing the indicated entry in the table. (It should be noted that this scheme does not provide for any locality in the random table lookup. Most computer applications would exhibit at least minimal locality in this kind of data retrieval.)

The script file (hereafter called the *request file*) was created with 300,000 records to insure a good steady state simulation run that would last at least several minutes. Figure 3 illustrates the structure of the request file.

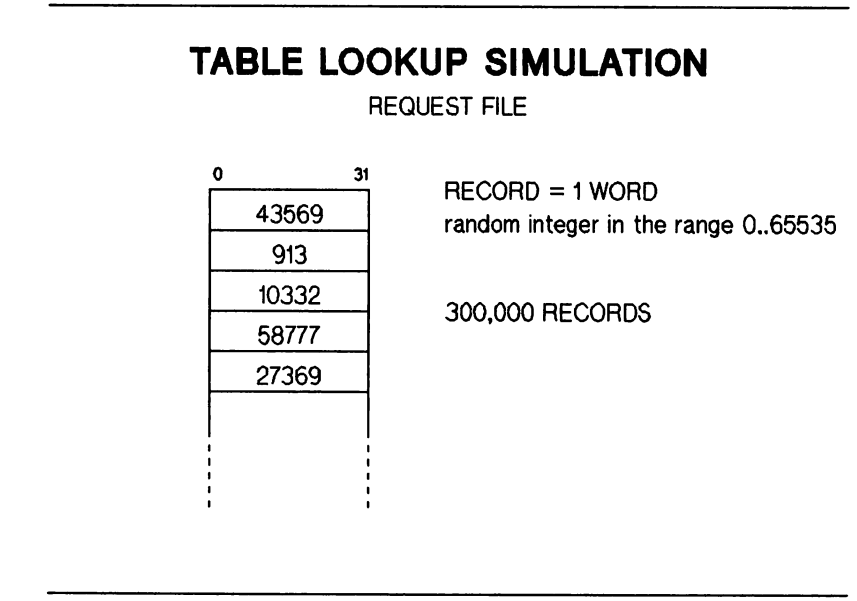

Figure 3

Table Lookup Simulation Using XDS

The data flow for this version of our program is shown in figure 4 below. Step 1 will be to create the Extra Data Segments required to hold the table. To represent a 1MB table in memory, 32 XDSs will be created, each 32768 bytes in size. Since each table entry is 16 bytes, each XDS will therefore hold 2048 entries.

In Step 2, we load the table with the data as defined above from an already existing flat file on disc (the *table file*). Once this is done, we are then ready to begin the simulation by starting a loop of reading the request file to get the table index requested, looking up the requested entry in the table, and then processing the entry (Step 3).

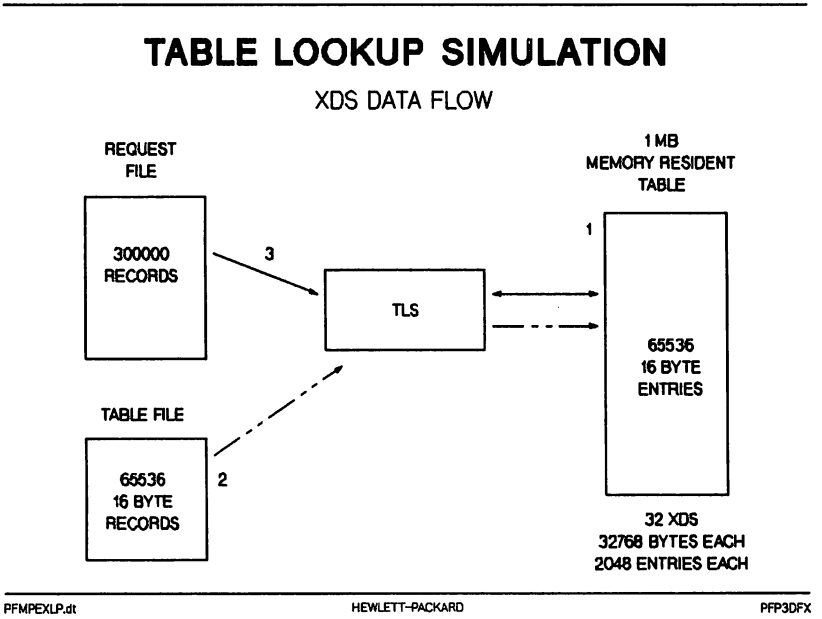

Figure 4

The commands and simulation output for the XDS test are shown below. User input is highlighted. The TLSXDSN program was compiled in native mode with the Pascal/XL compiler.

```
:FILE TABLE=TABLE.PROJECTS.TROUT
:FILE REQUEST=REQ1.PROJECTS.TROUT
:RUN TLSXDSN

Extra Data Segments created.
Loading the Extra Data Segments...
Table loaded.  Number of entries =  65536.
Request script file REQ1.PROJECTS.TROUT opened.
Starting Table Lookup Simulation...

Table Lookup Simulation completed on 300000 requests.
CPU time used =       597252 milliseconds.
Elapsed time of simulation =       610966 milliseconds.

END OF PROGRAM
:
```

As mentioned above, the design of the simulation program is structured so that each technique can be tested by simply replacing specific program procedures. The main body code for the XDS version of the Table Lookup Simulation program is as follows:

```
BEGIN  { main body }
   initialize;
   { start measurements }
   load_table;
   setup_loop;
   repeat
     get_entry_request;
     look_it_up;            { required only in the XDS version }
     process_entry;
   until no_more_requests;
   { end measurements }
   close_down;
END.
```

For the "large array" and "mapped file" versions of the simulation program, the only difference in the main body code is that there is no **look_it_up** procedure. The **look_it_up** procedure is required in the XDS example because we are managing a 1MB memory table in 32 Extra Data Segments. For each entry request, it is necessary to first determine which XDS the requested entry is in, and then calculate the offset within that XDS for the beginning of the actual entry.

This extra overhead is compounded by the fact that once we locate the entry, we must then move the data from the XDS to our user stack so that it can be processed. Already we can see that, aside from performance issues, use of XDSs is not very productive for the programmer when compared to using a simple large array or a user mapped file.

The code for **load_table** in our XDS program would look like this:

```
BEGIN  { load_table }
  xds_size := 16384;
  for dirc_ptr := 1 to 32 do
    getdseg (xds_dirc[dirc_ptr], xds_size, xds_id);
  reset (table_file);
  load_counter := 0;
  for dirc_ptr := 1 to 32 do
    for xds_entry := 1 to 2048 do
      begin
        read (table_file, table_entry);
        disp := xds_entry * 8 - 8;
        dmovout (xds_dirc[dirc_ptr], disp, 8, table_entry);
        load_counter := load_counter + 1;
      end;
  close (table_file);
END;
```

The **look_it_up** procedure would be coded as follows:

```
{ table_index has just been read from the request file }

BEGIN  { look_it_up }
  dirc_ptr := trunc (table_index / 2048) + 1;
  disp := (table_index mod 2048) * 8;
  dmovin (xds_dirc[dirc_ptr], disp, 8, table_entry);
END;
```

In this simulation, we are mainly concerned with the performance of retrieving a large number of table entries. What we do with the entry once retrieved is not really of interest. Of course, the bulk of a typical application would be in the processing of table data, not in retrieving it.

For the purposes of this simulation testing, our **process_entry** procedure is very simple. To verify that we have the correct entry, a simple comparison is made between the table index used to find the entry (from the request file) and the first subfield in the entry itself. From our description above of the table data, we know that the first subfield is nothing more than the index of the entry. This means that *our comparison should always show equal values.* If not, we have somehow retrieved the wrong entry!

So, the **process_entry** procedure will look something like this:

```
{ table_index has just been read from the request file }

BEGIN  {process_entry }
  if table_index <> table_entry.f1 then
    { error };
  req_counter := req_counter + 1;
END;
```

The other routines in the Table Lookup Simulation program need not be examined in detail since they essentially do housekeeping, initializations, etc. Refer to the complete program listing at the end of this paper.

From the simulation output given above, we know that the XDS version of our program ran for about 10 minutes (610966 milliseconds to be exact). A graphical look at what happened on the system during the test is seen in figure 5. This graph shows CPU utilization during the simulation. (Note: In all test runs, the performance data collections were started two minutes before the simulation program was started so that the effects of the simulation could be clearly seen.) As expected, the dedicated batch job environment produced 100% CPU utilization during the simulation.

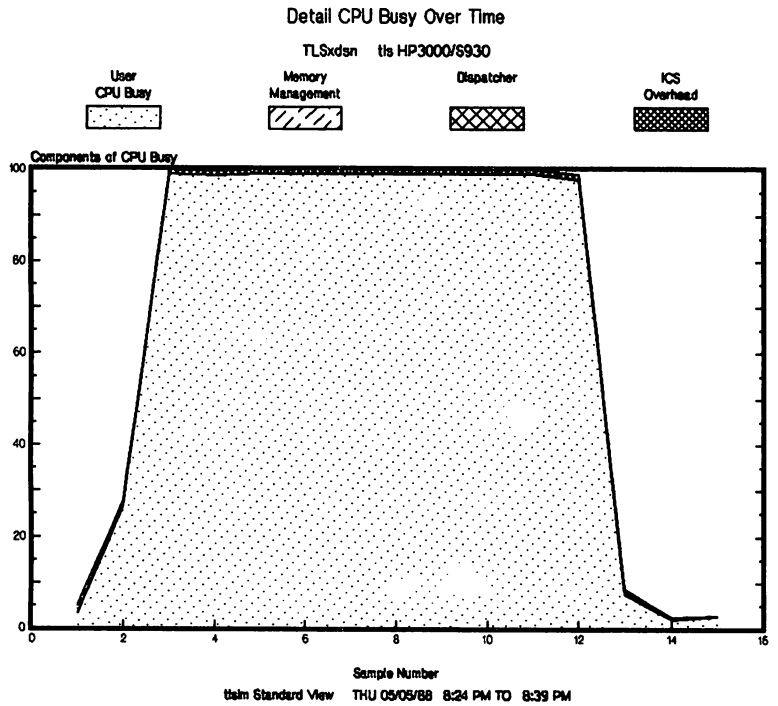

Figure 5

The really significant performance factor for the XDS example, however, is shown in the following chart. Because the XDS intrinsics reside in the compatibility mode SL (SL.PUB.SYS), a program running in native mode (as this one was) must access these intrinsics through the Switch Subsystem in MPE XL. Using the intrinsics is transparent to the native mode program because the necessary switch stubs already exist.

Figure 6 shows that during the simulation, the switch rate was sustained at about 600 switches per second. This is an extremely high switch rate and definitely contributes to degraded performance in the XDS version of the simulation.

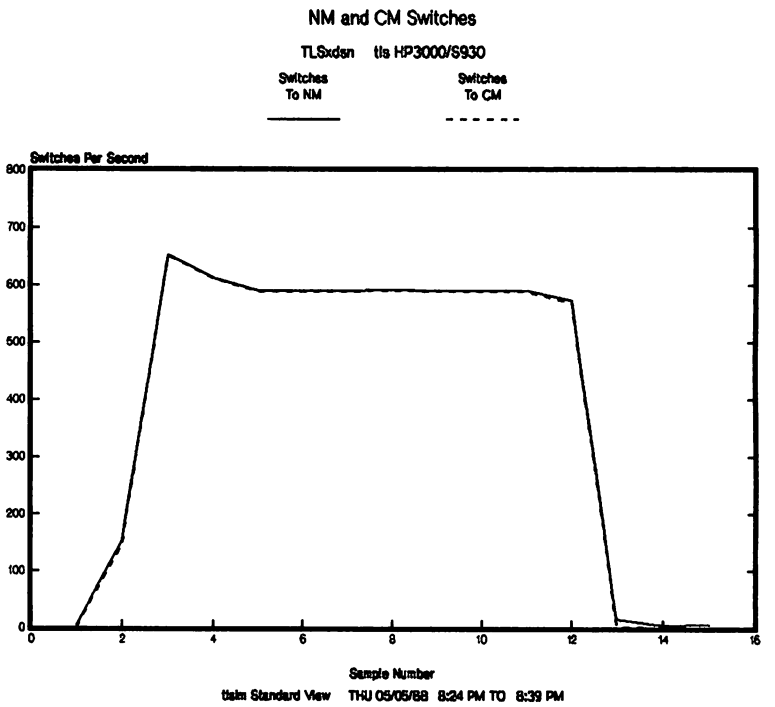

Figure 6

NM vs CM

To compare NM versus CM, the XDS version of the simulation was compiled using the CM Pascal/3000 compiler on the Series 930 and rerun in CM. In this case, elapsed time was even longer: 893342 milliseconds, or almost 15 minutes. The longer run time can be attributed to the overhead of CM versus NM for this kind of CPU intensive program and the fact that the number of switches was even higher in CM (see the summary table in Figure 12 at the end of the paper). Remember that the simulation issues 300000 sequential reads to the request file; the file system code for doing these reads is in NM, so the CM program must switch to NM for all the request file I/O.

As this test illustrates, NM performance is better than CM performance, even when part of the application is calling system routines which are in CM. This will generally be true for most applications being migrated to MPE XL, although there may be certain corner cases where it may be desirable to leave an application in CM (at least for now).

Table Lookup Simulation Using A Large Array

Now that a "past technology" solution for the Table Lookup Simulation problem has been established, the newer techniques which provide easier design and better performance can be examined.

With the extremely large addressing capability of MPE XL, it should be obvious that the table can simply be represented as a large memory resident array. Instead of having to create and manage multiple Extra Data Segments, a single block of memory space can be created which will contain the entire table. Indexing to a given entry in the table will be greatly simplified from a programming point of view and the XDS overhead is totally eliminated.

However, once the table array is created, there is still a need to load it from a disc file as in the XDS example. Figure 7 shows the data flow in the "large array" version of the simulation.

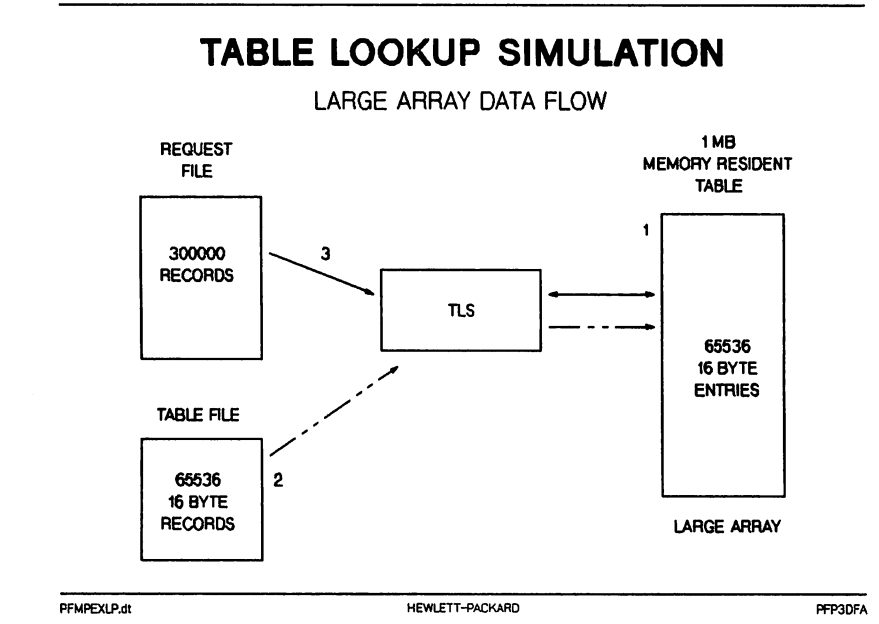

Figure 7

Although the data flow is essentially identical to the XDS example (refer to the earlier discussion of the steps involved), several procedures in the program will need to change. The main body code (refer to the listing given earlier) is the same except that there is no **look_it_up** procedure. The simulation test for the large array example resulted in the following output. Program TLSARRN was compiled in native mode with the Pascal/XL compiler.

```
:FILE TABLE=TABLE.PROJECTS.TROUT
:FILE REQUEST=REQ1.PROJECTS.TROUT
:RUN TLSARRN

Memory array created.
Loading the memory array...
Table loaded.  Number of entries =  65536.
Request script file REQ1.PROJECTS.TROUT opened.
Starting Table Lookup Simulation...

Table Lookup Simulation completed on 300000 requests.
CPU time used =        143231 milliseconds.
Elapsed time of simulation =        145824 milliseconds.

END OF PROGRAM
:
```

The code fragment below shows how the array is defined and also the declaration for the pointer which will be used to index the table.

```
TYPE
  table_entry_type = record
    f1 : integer;
    f2 : packed array [1..12] of char;
  end;
  table_type = array [0..65535] of table_entry_type;

VAR
  table : ^table_type;      { table pointer }
```

It would also be possible to index the table with a simple integer variable "pointer" which would be calculated each time an entry is desired. There is really no need to do this, however, since Pascal offers the pointer data type which can be conveniently used in the program syntax. This will be illustrated below.

One procedure which will obviously need to change is **load_table**. Instead of creating and loading 32 Extra Data Segments, the array can be created on the Pascal heap with the *new* function and then loaded by dereferencing the table pointer:

```
BEGIN  { load_table }
  new (table);           { allocates the array on the heap }
  reset (table_file);    { open the table data file on disc }
  table_index := 0;
  repeat
    read (table_file, table^[table_index]);
    table_index := table_index + 1;
  until eof (table_file);
  close (table_file);
END;
```

Notice how convenient it is to dereference the pointer `table` and specify an array element (`table_index`) in the read statement above. In one Pascal statement, we have read the table file and loaded data into the appropriate table entry in the memory array.

The only other procedure which needs to change is **process_entry**. Only one statement change is required:

```
{ table_index has just been read from the request file }

BEGIN  { process_entry }
  if table_index <> table^[table_index].f1 then
    { error };
  req_counter := req_counter + 1;
END;
```

Again, the power of Pascal syntax is evident in the `if` statement above. The pointer `table` is dereferenced by `table_index` and then the subfield variable `f1` which "retrieves" the correct table entry and subfield. In one statement, the table lookup is accomplished.

This version of the Table Lookup Simulation exhibited much improved performance over the XDS version; the elapsed time of this run (145824 milliseconds, or about 2.5 minutes) is a significant *4.2 times improvement* over the XDS test run! (Refer to the summary table at the end of this paper.) The major factor contributing to the improved performance is the almost non-existent switch rate since the XDS intrinsics are no longer being used. In addition, the path length of the table lookup is greatly shortened since there is no need to calculate XDS pointers and offsets. Using a large array provides for a much simpler and more efficient means of accessing the table data.

Clearly, using the extended addressing capability of MPE XL can result in impressive improvements in run-time performance in addition to the programmer productivity enhancement already mentioned. Can these great results be improved even more? Yes, they can.

Table Lookup Simulation Using A Mapped File

One part of the simulation that could be eliminated is the table loading. This is essentially "wasted" time in the simulation and needs to be done each time the program is run (in the XDS and large array versions). It would be far more desirable in this kind of application to make the table *durable* from one run to the next and somehow access it from our program as a memory array. This "best of all worlds" solution can be implemented by using a *mapped file.*

User mapped I/O is a feature of MPE XL which is particularly unique and powerful. The essential attributes are:

- A method of accessing data from files using a virtual pointer.
- Accessed using HPFOPEN intrinsic specifying a long (64 bit) or short (32 bit) pointer.
- File "reads" and "writes" are accomplished at the level of LOAD and STORE machine instructions.
- File System buffering and overhead is bypassed; structure of the data is user defined; access files like memory, memory like files.
- Can be much faster than normal file access, especially for non-sequentially accessed files.

User mapped I/O is possible because of the basic design of HP Precision Architecture. All objects to be accessed in memory (including files) are *mapped* into a large *virtual address space.* When a file is opened, it is assigned a virtual address range which encompasses the first byte to the last. By opening the file in such a fashion as to return to the user a *virtual address pointer* which "points" to the first byte of the file, all data in the file can then be accessed by dereferencing the pointer.

As the file is referenced, it is brought into real memory in *pages* from secondary storage (disc). The essential components of this *virtual demand paging* scheme are shown in figure 8.

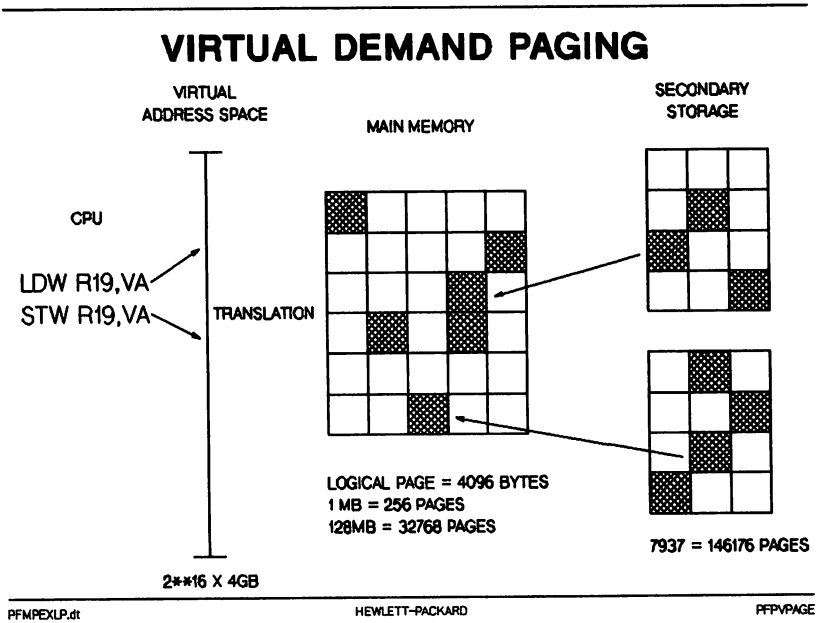

Figure 8

In the "mapped file" version of the simulation program, then, there will be no need to load the table data as a separate step. The memory table and the table file on disc (previously used to load from) will be *one and the same.* This greatly simplifies the data flow of the simulation, as shown in figure 9:

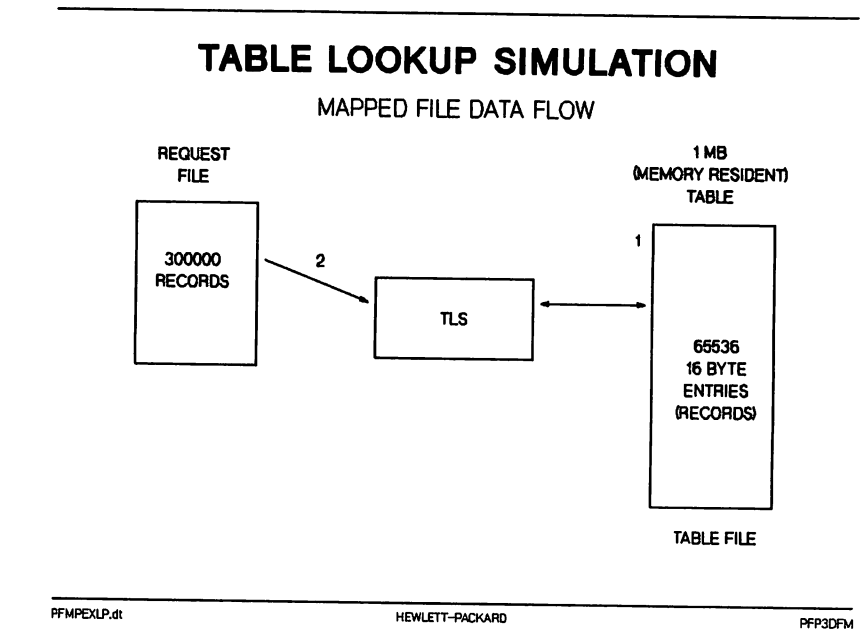

Figure 9

Only two steps are necessary: 1) Open the table file specifying mapped I/O, and 2) Loop through the request file. Although the entire table will not be memory resident at first, it will gradually become memory resident as more and more pages of the file are touched.

The simulation output for the "mapped file" test are shown below. The TLSMION program was compiled in native mode with the Pascal/XL compiler.

```
:FILE TABLE=TABLE.PROJECTS.TROUT
:FILE REQUEST=REQ1.PROJECTS.TROUT
:RUN TLSMION

Table file opened for mapped access.
Number of entries in table =  65536.
Request script file REQ1.PROJECTS.TROUT opened.
Starting Table Lookup Simulation...

Table Lookup Simulation completed on 300000 requests.
CPU time used =        116992 milliseconds.
Elapsed time of simulation =        124550 milliseconds.

END OF PROGRAM
:
```

As with the "large array" version, the main body code will be the same as the XDS version except that there is no **look_it_up** procedure (refer to the earlier listing). The **load_table** procedure will now become the place that the table file is opened for mapped access. No loading need be done, so the procedure is essentially just the HPFOPEN:

```
BEGIN  { load_table }
  hpfopen (filenum, status,
           ffd_option, table_filename,
           domain_option, permanent,
           access_type_option, read_only,
           short_mapped_option, table);  { return the pointer }
END;
```

Notice that the HPFOPEN intrinsic is called using *itemnum, item* pairs. This greatly improves coding accuracy and ease compared to the FOPEN intrinsic.

The important part of the HPFOPEN call is in the last line above; the *short mapped option* is requested and in the pointer variable `table` is to be returned the virtual address of the beginning of the file. Both the table array and the pointer `table` are declared in exactly the same way as they were previously for the "large array" version of the simulation (see code fragment above). As a result, the **load_table** procedure is the only piece of code that needs to change in order to convert the "large array" program into a "mapped file" program. Procedure **process_entry** is identical for the two versions (see code fragment above).

For a detailed look at the entire simulation and the coding technique for mapped files, refer to the complete listing of the program at the end of this paper.

From above, the elapsed time of the "mapped file" version of the simulation was 124550 milliseconds, or about 2 minutes. This is even more improvement over the XDS version than was seen with the large array technique, and results from not having to "load" the table as a first step in the simulation. To graphically see the benefits of the mapped I/O technique versus the XDS technique, compare the following graph with the earlier XDS graph showing CPU utilization:

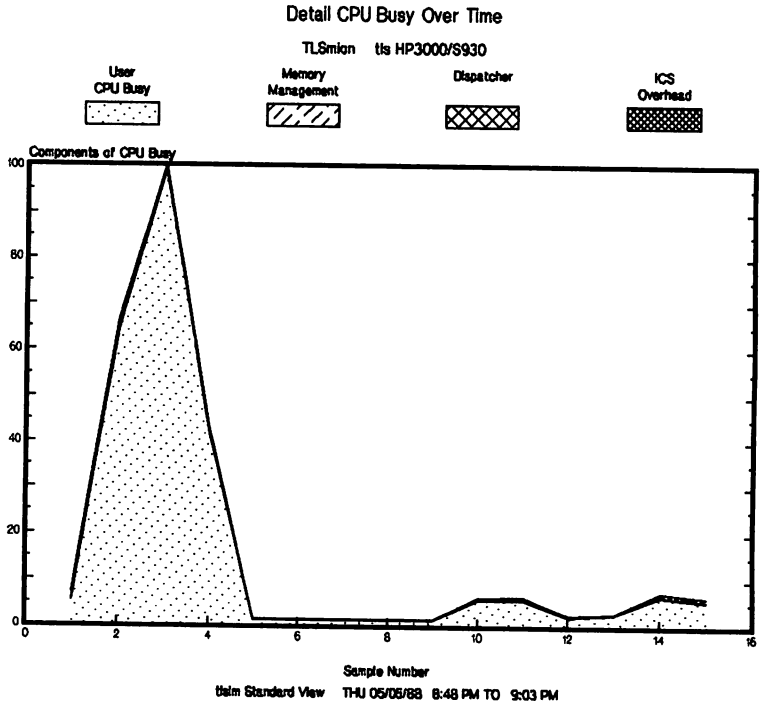

Figure 10

As expected, the switch rate dramatically improved over the XDS test. Note that the X-axis in the following graph is the same as before, however the Y-axis shows a much smaller range:

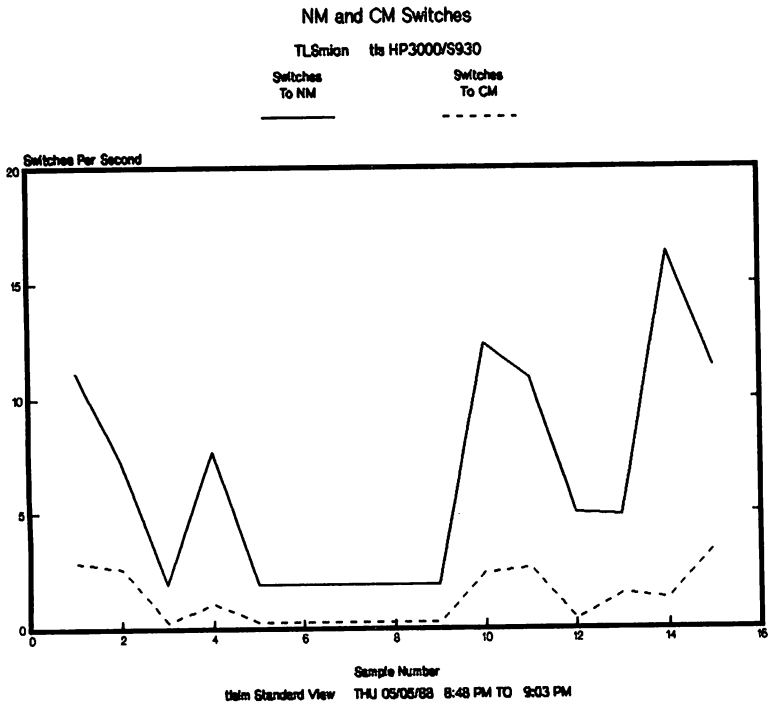

Figure 11

Mapped I/O Considerations

There are some considerations when using mapped files. Since the File System is being effectively bypassed, things like EOF and file posting are no longer being automatically managed. The user must explicitly use the FPOINT and FCONTROL intrinsics to set the EOF in those cases when records have been written beyond the existing EOF. This only need be done before the file is closed. (Since we are only *reading* the file in our example program there is no need for this.)

For critical applications, the user may want to use FCONTROL to force physical posting of file pages when appropriate, although too much of this would negate the benefits of using mapped files.

As noted above, mapped files may be opened with either a short pointer or a long pointer. For short pointer access, a file may be up to 4MB in size and a total of 6MB of mapped files may be opened at once per process. For long pointer access, a file may be up to 2GB in size and there is no limit to the total of mapped file space being utilized per process.

Short pointers are more efficient than long pointers and should be used wherever possible. If files are to be opened using a mix of techniques (FOPEN, HPFOPEN, HPFOPEN mapped, etc.) simultaneously, then use of long pointers will be required.

Conclusions

The following table shows the pertinent performance data for all of the simulation test runs. The "relative performance" column has been normalized to the XDS program running in native mode and is based on elapsed time of the simulation.

TABLE LOOKUP SIMULATION

RELATIVE PERFORMANCE
SERIES 930/64MB
MPE XL A.01.10

TEST	CPU ms	ELAPSED ms	NUMBER OF SWITCHES	% NM	RELATIVE PERFORMANCE
XDS, CM	872255	893342	931200	53.4	0.68
XDS, NM	597252	610966	731313	83.6	1
ARRAY, NM	143231	145824	112	99.9	4.2
MAPPED I/O, NM	116992	124550	110	99.9	4.9

PFMPEXLP.dt HEWLETT-PACKARD PFPRELP

Figure 12

The mapped I/O program shows the best results at 4.9 times the XDS program. The large array test also showed significant improvement at 4.2 times the XDS program. Both of these new MPE XL techniques illustrate the power and performance potential of HP Precision Architecture. In addition, it has been demonstrated that the programmer's task can be simplified and productivity improved. Clearly, the benefits of application design changes to take advantage of HP Precision Architecture are worth the effort.

Appendix A: Additional References

- Programmer's Skills Migration Guide (30367-90005)
- Accessing Files Programmer's Guide (32650-90017)
- MPE XL Intrinsics Reference Manual (32650-90028)
- *Hewlett-Packard Journal*, December 1987, "MPE XL: The Operating System for HP's Next Generation of Commercial Computer Systems"

Appendix B: TLSMION Program Listing

```
$OS 'MPEXL'$
$STANDARD_LEVEL 'HP_PASCAL'$
$CODE_OFFSETS ON$
$TABLES ON$
$VERSION '1.0'$
$TITLE 'Table Lookup Simulation Using Mapped I/O'$

PROGRAM tlsmio (input,output);

{ 1.0  05/02/88  Dave Trout, HP Rockville }

{ This program does a Table Lookup Simulation.  The purpose of this
  simulation is to compare the Extra Data Segments (XDS) capability
  of MPE for table handling versus Mapped I/O techniques and extended
  addressing capabilities in MPE XL.

  Data Structures:

    TABLE - 65536 entries, accessed by index 0..65535
    TABLE ENTRY - 16 bytes (4 MPE XL words|8 halfwords|8 MPE V/E words)

    Total table size is 1,048,576 bytes (1MB).

  The table will be accessed as a memory resident table using XDS
  intrinsics, a large array, or a mapped file.  To simulate accessing the
  table, a large script file has been created; each record in this script
  file is an integer which has a randomly distributed value in the range
  0..65535.  As this file is read sequentially, each record's value will
  be used as the index for finding and processing the indicated entry in
  the table.

  It should be noted that this scheme does not provide for any locality
  in the random table lookup.  Most computer applications would exhibit
  at least minimal locality in this kind of data retrieval.
```

```
  Since table lookup applications typically require extremely fast
  access, it is desirable to have an efficient access PATH to table
  entries which are MEMORY RESIDENT.  Performance of this simulation will
  be determined in large part on how well these objectives are met. }

CONST
  ccg = 0;
  ccl = 1;
  cce = 2;
  req_ffd = 'REQUEST';                          { request file }
  time_adj = 2073600000;                        { timer adjust }

TYPE
  xlstatus = record
    case integer of
      0 : (all : integer);
      1 : (info : shortint;                     { error number }
           subsys : shortint);                  { subsystem number }
    end;

  table_entry_type = record
    f1 : integer;                               { logical entry number }
    f2 : packed array [1..12] of char;          { DASCII version of above }
    end;

  table_type = array [0..65535] of table_entry_type;

VAR
  status : xlstatus;
  no_more_requests : boolean;
  req_file : file of integer;
  table_index : integer;
  req_counter : integer;
  timer_start : integer;
  timer_stop : integer;
  time : integer;
  table : ^table_type;                          { table pointer }
  filenum : integer;
  cpu_time : integer;

FUNCTION timer : integer; intrinsic;
FUNCTION proctime : integer; intrinsic;
PROCEDURE hpfopen; intrinsic;
PROCEDURE fclose; intrinsic;
PROCEDURE ffileinfo; intrinsic;
PROCEDURE terminate; intrinsic;

PROCEDURE stop (parm : integer);

  BEGIN
    writeln ('*** Fatal error; parm = ',parm:5);
    terminate;
  END;
```

```
PROCEDURE initialize;

  BEGIN
    no_more_requests := false;
    req_counter := 0;
  END;

PROCEDURE load_table;

  { Since the table already exists as a permanent file on disc which
    will be opened and accessed using mapped I/O, there is no "loading"
    to do.  The table will become memory resident automatically as the
    simulation touches pages of the file.  This routine will HPFOPEN
    the file specifying the short mapped option.  The short pointer
    returned by HPFOPEN will then be used in later routines to access
    entries in the table. }

  CONST
    ffd_option = 2;                          { setup for HPFOPEN }
    domain_option = 3;
    short_mapped_option = 18;
    access_type_option = 11;
    file_eof = 10;

  VAR
    table_filename : packed array [1..10] of char;
    permanent : integer;
    read_only : integer;
    entry_count : integer;

  BEGIN
    table_filename := '%TABLE%';             { setup ffd }
    permanent := 1;                          { permanent file domain }
    read_only := 0;                          { read only access }

    hpfopen (filenum, status,                { open the table file }
             ffd_option, table_filename,
             domain_option, permanent,
             access_type_option, read_only,
             short_mapped_option, table);    { return the short pointer }

    if status.all <> 0 then
      begin
        writeln ('error on hpfopen; info = ',
                 status.info,', subsys = ',status.subsys);
        stop (6);
      end;
    writeln ('Table file opened for mapped access.');
    ffileinfo (filenum, file_eof, entry_count);
    if ccode <> cce then stop (7);
    writeln ('Number of entries in table = ',entry_count:6,'.');
  END;
```

```
PROCEDURE setup_loop;

  CONST
    afd = 1;

  VAR
    req_file_afd : packed array [1..28] of char;
    afd_string : string[28];
    n : integer;

  BEGIN
    reset (req_file,req_ffd);                { open the request file }
    ffileinfo (fnum (req_file), afd, req_file_afd);
    setstrlen (afd_string, 0);
    strwrite (afd_string, 1, n, req_file_afd);
    afd_string := strrtrim (afd_string);
    writeln ('Request script file ',afd_string,' opened.');
  END;

PROCEDURE get_entry_request;

  BEGIN
    read (req_file,table_index);
    if eof(req_file) then no_more_requests := true;
  END;

PROCEDURE process_entry;

  { Since we are using mapped I/O, we simply reference the entry we
    want to retrieve it.  For this simulation, just test to make sure
    we do indeed have the right table entry.  Since the first word of the
    table entry is an integer whose value is the table index, we can
    simply compare the index used with the retrieved value.  They should
    be the same! }

  BEGIN
    if table_index <> table^[table_index].f1 then
      begin
        writeln ('we have a problem here...index/entry don''t agree.');
        stop (4);
      end;
    req_counter := req_counter + 1;          { this one is done }
  END;

PROCEDURE close_down;

  BEGIN
    close (req_file);                        { close the request file }
    fclose (filenum,0,0);                    { close the table file }
    if ccode <> cce then stop (8);
  END;
```

```
BEGIN {Table Lookup Simulation}

  initialize;
  cpu_time := proctime;                          { start measurements }
  timer_start := timer;
  load_table;
  setup_loop;
  writeln ('Starting Table Lookup Simulation...');

  repeat
    get_entry_request;
    process_entry;
  until no_more_requests;

  cpu_time := proctime - cpu_time;               { end measurements }
  timer_stop := timer;
  time := timer_stop - timer_start;
  if time < 0 then time := time + time_adj; { fix time if required }
  writeln;
  writeln ('Table Lookup Simulation completed on ',
           req_counter:6,' requests.');
  writeln ('CPU time used = ',cpu_time,' milliseconds.');
  writeln ('Elapsed time of simulation = ',time,' milliseconds.');

  close_down;

END {Table Lookup Simulation}.
```