REX/3000 AS A PROGRAMMER PRODUCTIVITY TOOL

Lance Carnes

Consultant

Mill Valley, CA

March 1981

ABSTRACT.

Programmer productivity is an important issue today: programming effort is the single largest factor in the design, implementation and maintenance of software systems. This paper surveys the major aspects of productivity: reducing programming effort, increasing program reliability, providing run-time efficiency, and reducing the cost of software production. Each of these aspects and its relationship to the others is explored. A unique high-level language, REX/3000 is recommended as a solution to boosting programmer productivity. REX/3000 programs are used to illustrate points made.

I. INTRODUCTION.

As the costs of labor and money increase, so do the pressures to control and reduce operating costs. In data processing departments programming personnel costs are the single largest cost factor. Organizations which have dealt with this problem attribute much of their success to the use of productivity tools [1].

Productivity tools are designed to reduce the time and cost required to produce software. The project manager can implement software more economically and with more efficient use of personnel through appropriate use of these tools. Equally as important is the user satisfaction with the software product developed using the productivity tool.

The interest in increasing programmer productivity is evident from the number of articles in trade journals and conference proceedings dealing with the subject. Most software departments have a larger backlog of programming requests than they have staff to do the work. Of the time spent in programming, typically 60% to 70% is involved with maintaining existing systems, while only 30% to 40% is utilized in new development. The cost of a person-month of programming effort is high and will continue to increase. New hardware is being developed faster than the software it will run.

Typically, productivity tools meet the traditional productivity requirement: decrease the amount of code required to implement the system and, therefore, reduce the programming effort. This is an obvious path to take and has proven successful. However, the result is often that with the reduced effort comes a reduction in the quality of the software. These tools are frequently specialized for certain applications and have limited scopes; once the limits are exceeded, the application must be redone using a different, usually less reliable and less productive, method. The system produced with the productivity tool is often less efficient than the same product implemented using standard methods.

REX/3000 is a high-level language and compiling system designed to meet the requirements of increased productivity. It has specialized constructs for report writing which result in a 50% to 90% reduction of effort over using general-purpose languages. The language was designed to encourage structured programming and thereby increase program reliability and correctness. As a compiling system it generates efficient program modules. There is sufficient scope in the range of applications which can be implemented so that it is rare that programs must be redone using more general-purpose languages.

In Section II the concept of productivity is defined and expanded. Section III is a brief introduction to the REX/3000 language. Section IV examines productivity quantitatively, i.e. reducing the effort to produce software. Section V deals with productivity qualitatively, i.e. maintaining reliability and efficiency. Section VI summarizes the cost savings realized with reduced quantity and improved quality. Section VII is a summary of the points made.

II. WHAT IS PRODUCTIVITY?

Traditionally, programmer productivity is the rate of software production, i.e.

$$\frac{\text{\# lines of code}}{\text{person-months}}$$

This ratio is derived by taking the total number of lines of code required to produce a software system and dividing by the total personnel time used. The total lines of code may refer to the final code put into production or it may be all code written, e.g. for documentation, test programs, discarded modules, etc. The total personnel time may refer only to actual coding time or may include all time spent in design, training, travel etc. This ratio can be used for predicting the effort which will be required to produce future similar systems.

This ratio has limited usefulness because it indicates only the rate of code production and tells us nothing of the total time or cost of developing a system. For example, a system which could be developed in 24 months using 20,000 lines of assembler code has the same productivity rate as if it were developed in 12 months using 10,000 lines of COBOL code. While the rates are the same, the total time and costs are doubled.

In recent literature, less importance is being given to the quantity of software produced and more consideration is being given to the quality of software [2,3]. The aspects of software reliability, correctness and efficiency are being explored. These aspects are as important as reducing programming effort, and in fact play an important part in reducing the maintenance effort. Quality can also be measured for the purpose of modelling or estimating as

$$\frac{\text{\# bugs found}}{\text{\# lines of code}} \qquad \frac{\text{actual efficiency}}{\text{expected efficiency}}$$

The first ratio measures software reliability, which may be required to fall within a certain tolerance to be considered usable software. The second ratio measures machine resources used versus the allowable or avaliable resources, and a minimum tolerance may be specified to indicate whether the software has been successfully implemented. For example, the daily production must run within a 24-hour period.

It has become apparent that too much energy has been spent on increasing productivity rates and not enough on maintaining or improving program quality. One of the main reasons for this is the dramatic cost reduction in the development phase as a result of increased productivity. However, the savings are often cancelled when the cost of maintaining the error-prone code is considered. Therefore, productivity tools must treat the issue of quality with equal importance as quantity.

In the following discussions, we will be concerned with the issue of quality as well as quantity. The relationship between increased quality and reduced effort will be covered.

III. WHAT IS REX/3000?

REX/3000 is a high-level language and compiling system useful for report writing and general data processing. It was designed to boost productivity significantly through use of a combination of special-purpose and general purpose language constructs. Special-purpose constructs, also called non-procedural constructs, are the heart of the language, allowing programs to be written quickly. The general-purpose constructs, also called procedural constructs, build onto the special-purpose program and increase the flexibility of the language.

In the following discussion, we will show how the nature of the language promotes productivity. For a more detailed treatment of the language, see [4,5].

REX can be used to develop useful programs quickly. These same programs can be expanded as requirements grow. The following example illustrates this point.

```
<< WAREHOUSE PARTS SUMMARY >>
DATABASE parts PASSWORD "ANY" ACCESS 5
  DATASET part-stock
REPORT
  GET parts.part-stock
    LIST whse  AS "WAREHOUSE",          &
         part# AS "PART#",              &
         qty   AS "QUANTITY"            &
      SORTED BY whse, part#             &
      SUMMARIZING qty ON whse, part#
  LOOP  << end of GET ... LOOP >>
END.
```

This is a complete REX program, which when compiled and run will produce the report shown in the Appendix.

The sections of the program are as follows:

1) DATABASE declaration. This special-purpose construct performs the following functions:
   a) it specifies the database name, password and access mode to be used.
   b) at compile time, all attributes of the database, the datasets indicated, and the items within each of the datasets are known - the programmer need not redeclare the database layout, provide buffers, etc..
   c) at execution time, the database is opened using the parameters from (a).
   d) access to the database is available through the GET construct.
   e) at the end of the program the database is closed.

   This is a non-procedural construct, that is, it performs all of the logic necessary to access the database. The programmer is insulated from all mechanics of database use.

2) The REPORT block. This special-purpose construct performs all of the steps required to produce a sorted, formatted report:
   a) The items indicated in the LIST statement are read from the database and written to an extract file. The extract file is formatted and maintained by REX.
   b) At the end of the input phase, the extract file is sorted in the given sequence (SORTED BY ...).
   c) The report is now printed with the column headers (AS "...") and control breaks (ON ...) indicated.

   This is a non-procedural construct, since the mechanics of formatting the extract file, sorting it, setting up column headers, testing for control breaks, etc. are part of the REX system.

3) The GET statement. This is a special-purpose construct which reads data from the dataset and performs the following function:
   a) The dataset mentioned is read entry by entry (serially in this case, although chained access mode is also available).
   b) As each entry is read, the statements between the GET... and LOOP are executed (in this case, the LIST statement).
   c) After the last entry is read, transfer is passed to the statement following the LOOP.

   GET is a non-procedural construct in the sense that the mechanics of access are hidden from the programmer. The programmer does have to place the LOOP in the right place; if the LOOP is omitted, the compiler assumes the loop includes all statements up to the END.

This code could be written and running correctly in a matter of minutes. The user would be pleased with the results for at most two days, and then, of course, would want to expand the function to include the following:

1) Print the unit price and total value in stock for each part;
2) Place an asterisk in the column next to part #'s for which the quantity is zero;

Typically, this is beyond the scope of a non-procedural report writer. To perform the first requirement, the price will have to be extracted from a second dataset (PART-MSTR) and multiplied by the quantity. Some logic will have to be implemented to allow the quantity to be checked for zero and an asterisk inserted.

These requirements are not beyond the scope of REX, and in fact the original program may be modified to include the enhancements. The following is the REX program which will satisfy the above requirements.

```
<< WAREHOUSE PARTS SUMMARY >>
<< enhanced to print price and value  >>
<< will print an asterisk if qty = 0 >>
DATABASE parts PASSWORD "ANY" ACCESS 5
  DATASET part-stock
PROGVAR star A1
        value P5.2
REPORT
  GET parts.part-stock
    IF qty = 0 THEN star = "*" ELSE star = " "
    GET parts.part-master              &
      WITH part# = parts.part-stock.part#
    LIST whse  AS "WAREHOUSE",         &
         star,                         &
         part# AS "PART#",             &
         qty  AS "QUANTITY",           &
         price AS "PRICE",             &
         value = price * qty           &
                AS "VALUE"             &
      SORTED BY whse, part#            &
      SUMMARIZING qty ON part#,whse
  LOOP  << end of GET ... LOOP >>
END.
```

The following parts were added to the program:

1) PROGVAR declaration. A program controlled variable, star, was declared which can contain one alphaumeric character (A1). The variable value is a five-digit packed-decimal number.
2) IF THEN ELSE statement. This statement checks the qty for zero and sets the variable star accordingly.
3) GET parts.part-master WITH... . This statement accesses the master set (PART-MSTR) keyed by part# to locate the price.
4) value = price * qty. This calculation is performed to compute the total value of parts in stock.

The PROGVAR declaration, the IF THEN ELSE and the calculation are procedural constructs, that is, the programmer has to specify the mechanics of the function.

Notice that the enhancement was made by adding procedural (general-purpose) constructs into the original non-procedural (special-purpose) program. With most non-procedural report writers, the enhancement could not be made, and the application would have to be recoded using a fully procedural language (e.g. COBOL).

In summary, REX allows the creation of non-procedural programs which can be coded quickly and by less experienced staff members. In addition, enhancements and more complex programs can use the rich set of procedural constructs. The special-purpose (non-procedural) constructs and the general-purpose (procedural) constructs can be combined in the same application.

IV. REDUCING THE PROGRAMMING EFFORT.

The major emphasis of any productivity tool is to reduce the effort to produce software. That is, reduce the number of lines of code, and therefore the time, which would have been required to implement the system using a general-purpose programming language.

The use of productivity tools has proven effective [1]. The time and costs for software development have been significantly reduced using such tools, by as much as 50% to 90%.

In practice, productivity tools generally are not versatile enough to be used exclusively. This is the chief drawback to such tools making a significant impact on the software development process. Typically they are designed for a limited scope of applications and work well within these limits. Too often, the limits of the tool have the following negative effects:

1) Enhancement requests which exceed the limits of the tool are not done, denying the user timely access to useful information.
2) The corresponding general-purpose program which includes the enhancements costs so much to develop that the user will rationalize that the data is not important enough to justify the cost.

For example, consider the following application written in QUERY, a useful but limited tool:

```
DATA-BASE = PARTS
PASSWORD =>> ANY
MODE =>> 5
FIND ALL PART-STOCK.PART#
REPORT
H1,"WAREHOUSE PARTS REPORT",30
H2,"WAREHOUSE  PART#  QUANTITY",32
D,WHSE,15
D,PART#,22
D,QTY,30
S1,PART#
S2,WHSE
END
```

This code could be put into production in a short time and would provide useful information. However any enhancement requests must be looked at with the limitations of QUERY in mind.

For example, if the requests were the same as those in the example in the previous section, QUERY could not be used:

1) Print the unit price and total value in stock for each part (QUERY can access only one dataset at a time and

cannot perform multiplications);

2) Place an asterisk in the column next to part #'s for which the quantity is zero (QUERY does not have alphanumeric variables or conditional statements).

The application would have to be coded in a general-purpose language.

The COBOL program which includes the enhancements is given in the Appendix; it is in excess of 230 source lines.

The main point here is the great disparity in the sizes of the programs. QUERY has 13 lines where the same application with two minor enhancements takes nearly twenty times the number of source lines in COBOL. The cost of enhancements in this case is much greater than would be imagined, especially by the user.

REX, however, provides a reasonable solution. The enhancements mentioned require only seven additional lines of code and a few minutes of time. Furthermore, the same source code may be built upon, avoiding a rewrite in a more general-purpose language.

In summary, REX combines the features of QUERY and COBOL. The programmer can produce simple programs in a short time, and simple or complex enhancements can be made by building onto the original source.

Two additional benefits result from using productivity tools to reduce programming effort:

1) Throw-away programs become feasible. Code can be written for a "what if" inquiry and then discarded. This would not be possible with high development costs.
2) Maintenance effort is reduced. The effort, and therefore the cost, of correcting bugs and making enhancements is reduced. The maintenance duties can be performed by a less experienced programmer. The savings are dramatic when considering the cost of supporting several systems over an extended period of time.

V. QUALITY - MAINTAINING OR IMPROVING IT.

In the previous section we noted that a frequent problem with using productivity tools is their lack of flexibility. Two other problems are often identified:

1) While it is easy to write code, it is difficult to use structured programming disciplines or other techniques which encourage error-free, reliable code.
2) The run-time modules are inefficient, consuming far more machine resources than the equivalent program written in a general-purpose language.

These issues arise when dealing with general-purpose languages as well. The first point concerns the reliability of programs, i.e. how bug-free the programs are. The second point concerns the efficiency of the program, i.e., the amount of machine required to execute the program.

Specialized productivity tools are generally reliable. They do not have the capability of performing complicated sequences, making it difficult to introduce bugs. The reliability will be lower, however, when the tool is pressed to its limits - programmers often code 'clever' but difficult to understand programs, or use side-effects of the system to circumvent the limitations of the language. Where the language does have some procedural constructs, they are often prone to the usual logic errors found when using non-structured languages.

Reliability can be increased by 1) training the programming staff in one of the structured programming techniques and/or 2) using a programming language which encourages error-free code. The first is a common technique when a software department {_is committed to using FORTRAN or COBOL; it is usually necessary to set up careful coding guidelines and review all code produced. The second is less common, though increasing with the availability of structured languages, e.g. Pascal, JOVIAL, Ada; these languages, however, are not suited for commercial applications or report writing.

REX was designed to encourage reliable coding. The non-procedural constructs perform reliably due to the fact that their function is well-defined and not alterable by the programmer (e.g. REPORT ... LIST). The procedural constructs in REX are borrowed from PASCAL, a structured, high-level language [6]. Coding is done using constructs such as PROCEDURE and REPORT blocks, IF THEN ELSE, WHILE DO, REPEAT UNTIL and GET LOOP, etc. REX has no GOTO. In short, the programmer must work with constructs which encourage reliable coding; those constructs known to be error-prone (e.g. GOTO) are not available.

Efficiency is an important issue, since all programs must eventually run in production and produce their results in an acceptable amount of time. A program which is inefficient will not be used and must be designed and implemented again. A program which is marginally efficient, i.e. runs slowly but within an acceptable range, will be subject to many costly attempts to speed it up. A program which was easy to develop but must be tuned constantly once in production has produced no real savings.

While many specialized tools are inefficient at run-time, REX is actually as efficient or more efficient than general-purpose language systems. The main difference is that most tools are interpretive, whereas REX is a compiling system. An interpreter is a general-purpose system which has heavy demands on the machine: it is a large program which has many code segments and uses large data areas. In contrast, a compiler produces an efficient runtime module: the program and data area requirements are only a fraction of those needed by an interpretive system. Reducing code and data memory requirements can greatly improve performance [7].

REX produces efficient run-time modules, similar to those resulting from a general-purpose compiling system. Segmentation is done automatically to speed the operation of REPORT blocks - the input phase code is in one segment while the print phase code is in another. Segment switching is minimized by generating as much code inline and avoiding PCALs whenever possible. Data segment usage is kept to a minimum through efficient code generation and the use of local variables, i.e. avoid global variables [7]. Since the programs run efficiently, there is seldom a need to optimize, saving maintenance effort.

Using REX allows high quality code to be generated with little additional effort or expense. The resulting programs are easier and less costly to maintain. The benefits are efficient production programs without the effort of extensive tuning. Overall, user and programmer satisfaction will be high.

VI. HOW ARE COSTS CUT?

Whenever there is a reduction of effort, increased program reliability and dependable machine efficiency, there is a corresponding cost savings. These savings may be immediately noticable, e.g. when reducing development costs. Or they may occur over an extended period of time, e.g. in the maintenance phase of the software life cycle. In addition to the savings from reducing effort, costs can be cut through use of less experienced personnel.

These are some of the ways costs are cut using productivity tools such as REX/3000 which not only reduce programming effort but encourage high quality:

1) Higher coding productivity results in fewer person-months of effort with a direct cost savings.
2) Higher reliability and efficiency reduce the number of person-hours required for maintenance over the life of the software system.
3) Less experienced and therefore lower cost personnel can implement and maintain software systems. The more experienced staff members can devote more time to designing current

and future software systems without worrying about whether there will be time enough for implementation.

VII. SUMMARY AND CONCLUSIONS.

Productivity tools do exactly what they claim - reduce the time and cost to produce software. Those tools which also increase the quality of produced code have the additional benefits of reducing maintenance time and effort. Overall, using a productivity tool allows more careful design and planning and better personnel allocation, since the pressure of the great amount of programming effort is relieved.

The quantity of code is reduced through the use of special-purpose constructs. Where these constructs typically reduce the scope and flexibility of the language, REX/3000 has met this shortcoming by allowing general-purpose constructs to be built onto the special-purpose core of the program.

The quality of code produced by productivity tools typically is not so high as that produced by general-purpose languages. REX/3000 allows high-quality coding through the use of structured programming techniques and efficiently compiled program modules.

The features of these tools are attractive and the wise programming manager will use them to produce economical, timely systems. However those projects implemented using tools in any capacity are few in number. The overwhelming majority of software systems produced use general-purpose languages, and overall show low productivity.

The reasons for not using tools are varied: some are legitimate, e.g. machine portability requirements; most, however, are the result of the fear of using something "new", or something which appears simplistic. There is a streak of the old-time wizard in every programmer, and the fact that the non-data processing user cannot comprehend the nature of the business is comforting and even protective. Some see the use of productivity tools as a threat to this mystique. Another common reason for not using tools is the reluctance to try something other than the standard methods, unproductive as these are. With the cost of person-power increasing, the obvious move is towards increased productivity.

One observer noting the lack of use of productivity tools drew the following analogy:

> [They] are so busy digging ditches with pick and shovel that they haven't the time to go watch the bulldozer demonstration [8].

With the cost of manpower increasing, it is imperative that tools be used in the near future. Those managers who cannot control costs and time schedules because of low programmer productivity will have to compete with managers who can make a difference. Productivity tools, like REX/3000, will play a major role in making that difference.

ACKNOWLEDGEMENTS.

Many thanks to Grace Gentry and Jean Danver for taking the time to read this paper and make useful suggestions.

REFERENCES.

1. Government Accounting Office report on data processing costs, GAO report #FGMSD-80-38, Washington, D.C., 1980.

2. DACS, A Bibliography of Software Engineering Terms, IIT Research Institute, October 1979.

3. DACS, Quantitative Software Models, IIT Research Institute, March 1979.

4. REX/3000 USERS MANUAL, Gentry Inc., 1980.

5. Carnes, Lance, "Design and implementation of REX/3000", HPGSUG Meeting Proceedings, Lyon 1979.

6. Jensen and Wirth, Pascal User Manual and Report, Springer-Verlag, 1974.

7. Green, Robert, "HP3000 / Optimizing On-line Programs", HPGSUG, Denver, 1978.

8. McClure, Bob in a speech to the Software Underground, San Francisco, CA, April 1980.

APPENDIX.

This section contains the database schema and program source code and output mentioned in the paper:

Listing of the schema for the PARTS database, and the contents of each dataset.

REX example report.

QUERY example report.

COBOL example report.

```
 HP32216A.04.01 QUERY/3000  MON, JUL 28, 1980,  3:59 PM
 QUERY/3000 READY
B=PARTS
PASSWORD =
ANY
MODE =
1
FORM

DATA BASE: PARTS                              MON, JUL 28, 1980,  4:00 PM

SET NAME:
   PART-MSTR,MANUAL

      ITEMS:
         PART#,                Z4            <<KEY ITEM>>
         PART-NAME,            U16
         PRICE,                P8

      CAPACITY: 101             ENTRIES: 3

SET NAME:
   PART-STOCK,DETAIL

      ITEMS:
         PART#,                Z4            <<SEARCH ITEM>>
         WHSE,                 U6
         QTY,                  Z4

      CAPACITY: 414             ENTRIES: 7


LIST PART-MSTR

PART#  PART-NAME          PRICE
 3122  MANUAL #177          275
 2142  BRACKET               75
 1785  BOLT 1 X 1/4           5

LIST PART-STOCK

PART#  WHSE      QTY
 1785  101      2000
 2142  100       750
 3122  100       100
 2142  102       250
 2142  101       100
 1785  100      1000
 3122  102         0
```

Listing of the schema for the PARTS database, and the contents of each dataset.

```
                    REX/3000  VERSION A.1.0623
                     (C) GENTRY, INC. 1980

1   1  << WAREHOUSE PARTS SUMMARY >>
1   1  DATABASE PARTS PASSWORD "READER" ACCESS 5
1   1    DATASET PART-STOCK
1   2  REPORT
2   2  GET PARTS.PART-STOCK
2   3    LIST WHSE AS "WAREHOUSE",                       &
2   3         PART# AS "PART#",                          &
2   3         QTY AS "QUANTITY"                          &
2   3      SORTED BY WHSE, PART#                         &
2   3      SUMMARIZING QTY ON WHSE
2   3  LOOP
2   2  END  << REPORT BLOCK >>
2   2  END.
```

```
WAREHOUSE PART# QUANTITY

100          1785     1000
100          2142      750
100          3122      100

                      1850

101          1785     2000
101          2142      100

                      2100

102          2142      250
102          3122        0

                       250
```

REX example report

```
                    REX/3000  VERSION A.1.0623
                     (C) GENTRY, INC. 1980

 1   1  DATABASE PARTS PASSWORD "READER" ACCESS 5
 1   1    DATASET PART-MSTR
 1   2      PRICE P6.2
 1   3    DATASET PART-STOCK
 1   2
 1   2  PROGVAR VALUE P7.2
 1   1          STAR A1
 1   1
 1   1  REPORT
 2   2  GET PARTS.PART-STOCK
 2   3    IF QTY = 0 THEN STAR = "*" ELSE STAR = " "
 2   3    GET PARTS.PART-MSTR WITH PART# = PARTS.PART-STOCK.PART#
 2   3    LIST WHSE AS "WAREHOUSE",                            &
 2   3         STAR,                                           &
 2   3         PART# AS "PART#",                               &
 2   3         QTY AS "QUANTITY",                              &
 2   3         PARTS.PART-MSTR.PRICE AS "  PRICE",             &
 2   3         VALUE = QTY * PARTS.PART-MSTR.PRICE             &
 2   3                 AS "   VALUE"                           &
 2   3      SORTED BY WHSE, PART#                              &
 2   3      SUMMARIZING "SUMMARY ",QTY,VALUE                   &
 2   4                        ON WHSE                          &
 2   4      TOTALING "GRAND TOTAL",QTY,VALUE
 2   3  LOOP
 2   2  END  << REPORT BLOCK >>
 2   2  END.
```

WAREHOUSE		PART#	QUANTITY	PRICE	VALUE
100		1785	1000	0.05	50.00
100		2142	750	0.75	562.50
100		3122	100	2.75	275.00
SUMMARY			1850		887.50
101		1785	2000	0.05	100.00
101		2142	100	0.75	75.00
SUMMARY			2100		175.00
102		2142	250	0.75	187.50
102	*	3122	0	2.75	0.00
SUMMARY			250		187.50
GRAND TOTAL			4200		1250.00

REX example report.

```
HP32216A.04.01 QUERY/3000  TUE, JUL 29, 1980,  2:10 PM
QUERY/3000 READY
DATA-BASE = PARTS
PASSWORD =
ANY
MODE =
5
FIND ALL PART-STOCK.PART#
7  ENTRIES QUALIFIED
REPORT
H1,"WAREHOUSE PARTS REPORT",30
H2,"WAREHOUSE  PART#  QUANTITY",32
D,WHSE,15
D,PART#,22
D,QTY,30
S1,PART#
S2,WHSE
END


        WAREHOUSE PARTS REPORT
      WAREHOUSE  PART#  QUANTITY
         100      1785     1000
         100      2142      750
         100      3122      100
         101      1785     2000
         101      2142      100
         102      2142      250
         102      3122        0
exit
```

QUERY example report.

```
$CONTROL USLINIT
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    PARTCOB.
 DATE-COMPILED.
            MON, JUL 28, 1980,  3:57 PM.
 REMARKS.
                THIS PROGRAM READS THE 'PARTS' DATA BASE
                LOOPS THRU MASTERS, GETS ASSOCIATED DETAILS
                AND SORTS THEM; IT THEN READS THE SORT FILE,
                OUTPUTING THE SORTED RECORDS, GIVING A SUMMARY
                OF TOTAL QUANTITY & COST AT EACH CHANGE IN
                'WAREHOUSE'
                *
                PRIMARY SORT KEY    -  WAREHOUSE
                SECONDARY SORT KEY  -  PART
                *
                NO PAGE CONTROL PRESENT
                *
                SET FILE EQUATION :FILE LP=$STDLIST;CCTL
                BEFORE EXECUTING
                *
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. HP3000.
 OBJECT-COMPUTER. HP3000.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT REPORT-FILE    ASSIGN TO "LP  ".
     SELECT SORT-FILE      ASSIGN TO "SORT,DA".
 DATA DIVISION.
 FILE SECTION.
 FD  REPORT-FILE
     RECORD CONTAINS 72 CHARACTERS
     LABEL RECORD IS OMMITTED.
 01  REPORT-FILE-REC.
     05  REPORT-FILE-REC-LINE     PIC X(72).
 SD  SORT-FILE
     RECORD CONTAINS 24 CHARACTERS.
 01  SORT-FILE-REC.
     05  SORT-FILE-REC-KEY.
         10  SORT-FILE-REC-WHSE   PIC X(08).
         10  SORT-FILE-REC-PART   PIC 9(04).
         10  FILLER               PIC X(12).
 WORKING-STORAGE SECTION.
 01  SORT-RCD.
     05  SORT-KEY.
         10  SR-WHSE              PIC X(08) VALUE SPACE.
         10  SR-PART              PIC 9(04) VALUE ZERO.
     05  SORT-DATA.
         10  SR-QTY               PIC 9(04)  VALUE ZERO.
         10  SR-PRICE             PIC S9(05)V9(02)
                                       VALUE ZERO.

 01  CONTROLS-AND-SUMS.
     05  SUM-QTY                  PIC S9(09) VALUE ZERO.
     05  SUM-COST                 PIC S9(12)V9(02) COMP-3
                                  VALUE ZERO.
```

```
006600        05  TOTAL-QTY               PIC S9(09) VALUE ZERO.
006700        05  TOTAL-COST              PIC S9(12)V9(02) COMP-3
006800                                    VALUE ZERO.
006900 01  HDR-LINE.
007000        05  HL-CC                   PIC X(01) VALUE SPACE.
007100        05  FILLER                  PIC X(52) VALUE
007200            "WAREHOUSE   PART#  QUANTITY    PRICE         VALUE  ".
007300 01  DTL-LINE.
007400        05  DL-CC                   PIC X(01) VALUE SPACE.
007500        05  DL-WHSE                 PIC X(08) VALUE SPACE.
007600        05  DL-FILLER               PIC X     VALUE SPACE.
007700        05  DL-STAR                 PIC X(02) VALUE SPACE.
007800        05  DL-PART                 PIC Z(06) VALUE ZERO.
007900        05  DL-QTY                  PIC Z(09) VALUE ZERO.
008000        05  DL-PRICE                PIC Z(07).9(02) VALUE 0.0.
008100        05  DL-COST                 PIC Z(12).9(02) VALUE 0.0.
008200 01  SUM-LINE.
008300        05  SL-CC                   PIC X(01) VALUE SPACE.
008400        05  TEXT-LINE               PIC X(17) VALUE "SUMMARY      ".
008500        05  SL-QTY                  PIC Z(09) VALUE ZERO.
008600        05  FILLER                  PIC X(10) VALUE SPACE.
008700        05  SL-COST                 PIC Z(12).9(02) VALUE 0.0.
008800 01  BLANK-LINE                     PIC X(72) VALUE SPACE.
008900 01  MISC.
009000        05  COST                    PIC S9(09)V9(02) COMP-3
009100                                          VALUE ZERO.
009200        05  LINE-COUNT              PIC S9(04) USAGE COMP SYNC
009300                                              VALUE ZERO.
009400        05  AT-END-FILE             PIC S9(04) USAGE COMP SYNC.
009500        05  IMAGE-MODE              PIC S9(04) USAGE COMP SYNC.
009600 01 IMAGE-DATASET-NAMES.
009700        05  IDN-PART-MSTR           PIC X(16) VALUE "PART-MSTR ".
009800        05  IDN-PART-STOCK          PIC X(16) VALUE "PART-STOCK ".
009900 01  IMAGE-STATUS-AREA.
010000        05  ISA-COND-WORD           PIC S9(04) USAGE COMP SYNC.
010100        05  ISA-DATA-LENGTH         PIC S9(04).
010200        05  ISA-RECORD              PIC S9(09) USAGE COMP SYNC.
010300        05  ISA-CHAIN-LENGTH        PIC S9(09).
010400        05  ISA-ADDRESS-BACK        PIC S9(09).
010500        05  ISA-ADDRESS-FORWARD     PIC S9(09).
010600 01 IMAGE-CONTROL-WORDS.
010700        05  ICW-TEMP                PIC S9(04) USAGE COMP SYNC.
010800        05  ICW-DBNAME              PIC X(16) VALUE "  PARTS; ".
010900        05  ICW-DATASET             PIC X(16) VALUE SPACES.
011000        05  ICW-PASSWORD            PIC X(08) VALUE "READER  ".
011100        05  ICW-MODE                PIC S9(04) USAGE COMP SYNC.
011200        05  ICW-DATALIST            PIC X(04) VALUE "@   ".
011300        05  ICW-SEARCH-ARG          PIC X(16) VALUE SPACES.
011400 01 IDB-PART-MSTR.
011500        05  IDB-PM-PART         PIC 9(04) VALUE ZERO.
011600        05  IDB-PM-NAME         PIC X(16).
011700        05  IDB-PM-PRICE        PIC S9(05)V9(02) COMP-3.
011800 01 IDB-PART-STOCK.
011900        05  IDB-PS-PART         PIC 9(04) VALUE ZERO.
012000        05  IDB-PS-WHSE         PIC X(06) VALUE SPACES.
012100        05  IDB-PS-QTY          PIC 9(04) VALUE ZERO.
012200 01 IMAGE-FIND-ITEM             PIC X(08) VALUE "PART#  ".
```

```
PROCEDURE DIVISION.
MAIN-PROCESS-CONTROL SECTION.
PAR-1.
    PERFORM OPEN-DB-E.
    PERFORM DO-THE-REPORT.
    STOP RUN.
DO-THE-REPORT.
    SORT SORT-FILE ON ASCENDING KEY SORT-FILE-REC-WHSE,
                                    SORT-FILE-REC-PART
        INPUT PROCEDURE IS GET-ENTRIES-LOOP
        OUTPUT PROCEDURE IS REPORT-ENTRIES.
GET-ENTRIES-LOOP SECTION 60.
PAR-A.
    MOVE "PART-MSTR"     TO ICW-DATASET.
    MOVE  2              TO ICW-MODE.
    CALL "DBGET" USING ICW-DBNAME,
                       ICW-DATASET,
                       ICW-MODE,
                       IMAGE-STATUS-AREA,
                       ICW-DATALIST,
                       IDB-PART-MSTR,
                       ICW-SEARCH-ARG.
    IF ISA-COND-WORD = ZERO
       PERFORM GET-NEXT-MASTER UNTIL AT-END-FILE = +11.
    GO TO END-OF-INPUT.

GET-NEXT-MASTER.
    PERFORM GET-THE-DETAILS.
    MOVE "PART-MSTR"     TO ICW-DATASET.
    MOVE  2              TO ICW-MODE.
    CALL "DBGET" USING ICW-DBNAME,
                       ICW-DATASET,
                       ICW-MODE,
                       IMAGE-STATUS-AREA,
                       ICW-DATALIST,
                       IDB-PART-MSTR,
                       ICW-SEARCH-ARG.
    IF ISA-COND-WORD NOT = ZERO
        MOVE  +11 TO AT-END-FILE.

GET-THE-DETAILS.
    MOVE "PART-STOCK"    TO ICW-DATASET.
    MOVE  1              TO ICW-MODE.
    MOVE IDB-PM-PART     TO ICW-SEARCH-ARG.
    CALL "DBFIND" USING ICW-DBNAME,
                        ICW-DATASET,
                        ICW-MODE,
                        IMAGE-STATUS-AREA,
                        IMAGE-FIND-ITEM,
                        ICW-SEARCH-ARG.
    IF ISA-COND-WORD = ZERO
       MOVE +5 TO ICW-MODE
       PERFORM PART-STOCK-LOOP
               UNTIL ISA-COND-WORD NOT = ZERO.
PART-STOCK-LOOP.
    CALL "DBGET" USING ICW-DBNAME,
                       ICW-DATASET,
```

```
                                ICW-MODE,
                                IMAGE-STATUS-AREA,
                                ICW-DATALIST,
                                IDB-PART-STOCK,
                                ICW-SEARCH-ARG.
           IF ISA-COND-WORD = ZERO THEN
                   MOVE IDB-PS-WHSE  TO SR-WHSE
                   MOVE IDB-PM-PART  TO SR-PART
                   MOVE IDB-PS-QTY   TO SR-QTY
                   MOVE IDB-PM-PRICE TO SR-PRICE
                   RELEASE SORT-FILE-REC FROM SORT-RCD.
       END-OF-INPUT. EXIT.
       REPORT-ENTRIES SECTION 70.
       PAR-C.
           PERFORM CLOSE-DB-E.
           OPEN OUTPUT REPORT-FILE.
           MOVE ZERO TO AT-END-FILE.
           RETURN SORT-FILE INTO SORT-RCD
                  AT END  DISPLAY " NO SORT RECORDS"
                          STOP RUN.
           WRITE REPORT-FILE-REC FROM HDR-LINE
                    AFTER  ADVANCING 1 LINES.
           WRITE REPORT-FILE-REC FROM BLANK-LINE
                    AFTER ADVANCING 1 LINES.
           MOVE SR-WHSE TO DL-WHSE.
           PERFORM WRITE-THE-REPORT UNTIL AT-END-FILE = +99.
           MOVE SUM-QTY    TO SL-QTY.
           MOVE SUM-COST   TO SL-COST.
           WRITE REPORT-FILE-REC FROM SUM-LINE
                  AFTER ADVANCING 2 LINES.
           WRITE REPORT-FILE-REC FROM BLANK-LINE
                  AFTER ADVANCING 1 LINES.
           ADD SUM-QTY TO TOTAL-QTY.
           ADD SUM-COST TO TOTAL-COST.
           MOVE "GRAND TOTAL" TO TEXT-LINE.
           MOVE TOTAL-QTY TO SL-QTY.
           MOVE TOTAL-COST TO SL-COST.
           WRITE REPORT-FILE-REC FROM SUM-LINE
                  AFTER ADVANCING 2 LINES.
           WRITE REPORT-FILE-REC FROM BLANK-LINE
                  AFTER ADVANCING 2 LINES.
           GO TO END-OF-REPORT.

       WRITE-THE-REPORT.
           IF SR-WHSE NOT = DL-WHSE
              THEN MOVE SUM-QTY    TO SL-QTY
                   MOVE SUM-COST   TO SL-COST
                   WRITE REPORT-FILE-REC FROM SUM-LINE
                         AFTER ADVANCING 2 LINES
                   WRITE REPORT-FILE-REC FROM BLANK-LINE
                         AFTER ADVANCING 1 LINES
                   ADD SUM-QTY TO TOTAL-QTY
                   ADD SUM-COST TO TOTAL-COST
                   MOVE ZERO TO SUM-QTY, SUM-COST
                   ADD 3 TO LINE-COUNT.
           IF SR-QTY = ZERO
              THEN MOVE "* " TO DL-STAR
```

```
    ELSE    MOVE "  " TO DL-STAR.
    MOVE SR-WHSE   TO DL-WHSE.
    MOVE SR-PART   TO DL-PART.
    MOVE SR-QTY    TO DL-QTY.
    MOVE SR-PRICE  TO DL-PRICE.
    MULTIPLY SR-PRICE BY SR-QTY
               GIVING COST.
    MOVE COST      TO DL-COST.
    ADD  COST      TO SUM-COST.
    ADD  SR-QTY    TO SUM-QTY.
    WRITE REPORT-FILE-REC FROM DTL-LINE
          AFTER ADVANCING 1 LINES.
    ADD 1 TO LINE-COUNT.
    RETURN SORT-FILE INTO SORT-RCD
          AT END MOVE +99 TO AT-END-FILE..

END-OF-REPORT.  EXIT.

SUPPORT-ROUTINES SECTION 80.
OPEN-DB-E.
    MOVE 5 TO ICW-MODE.
    CALL "DBOPEN" USING ICW-DBNAME,
                        ICW-PASSWORD,
                        ICW-MODE,
                        IMAGE-STATUS-AREA.
    IF ISA-COND-WORD NOT = ZERO
       THEN DISPLAY "ERROR IN DBOPEN",
                    ISA-COND-WORD
            STOP RUN.
CLOSE-DB-E.
    MOVE 1 TO ICW-MODE.
    CALL "DBCLOSE" USING ICW-DBNAME,
                         ICW-DATASET,
                         ICW-MODE,
                         IMAGE-STATUS-AREA.
    IF ISA-COND-WORD NOT = ZERO
       THEN DISPLAY "ERROR IN DBCLOSE",
                    ISA-COND-WORD.
```

WAREHOUSE		PART#	QUANTITY	PRICE	VALUE
100		1785	1000	.05	50.00
100		2142	750	.75	562.50
100		3122	100	2.75	275.00
SUMMARY			1850		887.50
101		1785	2000	.05	100.00
101		2142	100	.75	75.00
SUMMARY			2100		175.00
102		2142	250	.75	187.50
102	*	3122		2.75	.00
SUMMARY			250		187.50
GRAND TOTAL			4200		1250.00

COBOL example report.