AD-A077 988 NORTHWEST REGIONAL EDUCATIONAL LAB PORTLAND OR F/G 5/9
RESEARCH IN ADAPTABLE PROGRAMMING TO ACHIEVE COMPUTER INDEPENDE--ETC(U)
DEC 77 C E FRYE DAHC19-76-C-0022
UNCLASSIFIED ARI-TR-77-B4 NL

1 OF 2
AD A077988



LEVEL II

ARI TECHNICAL REPORT
TR-77-B4

(1)

AD A077988

RESEARCH IN ADAPTABLE PROGRAMMING TO ACHIEVE COMPUTER INDEPENDENCE

---

by

Charles H. Frye

THE NORTHWEST REGIONAL EDUCATIONAL LABORATORY
Portland, Oregon 97204

DECEMBER 1977

Contract DAHC 19-76-C-0022 new

Educational Technology and Training Simulation
Technical Area

DDC FILE COPY

DDC
RECEIVED
DEC 12 1979
D

Prepared for

U.S. ARMY RESEARCH INSTITUTE
for the BEHAVIORAL and SOCIAL SCIENCES
5001 Eisenhower Avenue
Alexandria, Virginia 22333

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

.9 22 5 103

U. S. ARMY RESEARCH INSTITUTE
FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the Deputy Chief of Staff for Personnel

W. C. MAUS
COL, GS
Commander

J. E. UHLANER
Technical Director

Research accomplished under
contract to the Department of the Army

Northwest Regional Educational Laboratory

NOTICES

DISTRIBUTION: Primary distribution of this report has been made by ARI. Please address correspondence concerning distribution of reports to: U. S. Army Research Institute for the Behavioral and Social Sciences, ATTN: PERI-P, 5001 Eisenhower Avenue, Alexandria, Virginia 22333.

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U. S. Army Research Institute for the Behavioral and Social Sciences.

NOTE: The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

Army Project Number
2R162725A778

Training Automation
and Simulation

ARI TR-77-B4

Technical Report 77-B4

RESEARCH IN ADAPTABLE PROGRAMMING TO ACHIEVE COMPUTER INDEPENDENCE

Charles E. Frye
Northwest Regional Educational Laboratory

Final rpt.

DAHC19-76-C-0022

Submitted by:
James D. Baker, Chief
Educational Technology and Training Simulation
Technical Area

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

December 1977

Approved by:

Joseph Zeidner, Director
Organization and Systems
Research Laboratory

J. E. Uhlaner, Technical Director
US Army Research Institute for
the Behavioral and Social Sciences

This is a report on an exploratory research project designed to meet military management requirements for research on a specific management problem. A limited distribution is made, primarily to the operating agencies directly involved.

392 779 LB

ABSTRACT

This document reports the results of four research and demonstration efforts, all of which bear some relationship to former work with the PLANIT computer-aided instructional system.

The first part of the report discusses the successful installation of PLANIT on a PDP 11 mini-computer. Because it was a feasibility study, some aspects of the installation were done in a less costly, though less efficient, way than if success had been assured. Now that feasibility has been established, suggestions are given for improving the installation.

The second part describes the utilization of the PLANIT method for transportable coding on a new application, an on-line query and retrieval system. Several aspects of this demonstration system are discussed and a User's Manual is included in the Appendix.

The third part describes a program which was developed for converting PLANIT student records into a format which the Query system can use. This provides a ready application for the demonstration of the Query system.

The fourth and final part describes the development of a test program which interactively checks out the MIOP subroutine which must be written locally in the process of installing a portable computer system, either PLANIT or Query. The Test Program makes the check out process fast, easy, and comprehensive. Using PLANIT to perform the same tests is difficult beyond the ability of most local personnel who are still new to the system.

User manuals for the last two programs are also found in the Appendix.

TABLE OF CONTENTS

	Page
INTRODUCTION	1
PURPOSE	2
INSTALLATION OF PLANIT ON A PDP-11	4
DEVELOPMENT OF THE PORTABLE QUERY SYSTEM	16
DATA BASE CONVERSION PROGRAM	28
THE MIOP INTERFACE TEST PROGRAM	30
CONCLUSIONS	34

APPENDIX A: LANGUAGE SPECIFICATIONS FOR THE MACHINE TRANSPORTABLE ON-LINE QUERY DEMONSTRATION

APPENDIX B: PORTABLE QUERY SYSTEM USER'S MANUAL

APPENDIX C: CONVERT: A DATA BASE CONVERSION PROGRAM FOR PLANIT STUDENT RECORDS

APPENDIX D: THE MIOP INTERFACE TEST PROGRAM USER'S MANUAL

APPENDIX E: DEMONSTRATION DATA BASE

RESEARCH IN ADAPTABLE PROGRAMMING TO ACHIEVE COMPUTER INDEPENDENCE

FINAL REPORT

INTRODUCTION

The research described in this report is in many ways an outgrowth of earlier work with PLANIT.

PLANIT (Programming LANguage for Interactive Teaching) is a publicly owned computer software system used for authoring and administering computer-assisted instruction scenarios interactively through time-sharing to a group of trainees. Support for its development came from the National Science Foundation and from the United States Army Research Institute (ARI).

The one feature that makes PLANIT unique in its class is its transportable design. Much of the earlier developmental effort was to achieve such a transportable package as PLANIT now is. It may be an over-simplification to say that NSF's interest in PLANIT was to see such a software design worked out, after which ARI has validated the design and put it to work.

Having successfully demonstrated that a portable software package like PLANIT can be used on a variety of hardware in a variety of operating environments to dispense training scenarios, ARI chose to expand their investigation to include software designs which are fully portable like PLANIT, but to be used in totally different applications.

Therefore, this report includes both aspects of their interest. One is the further probing of the extent of PLANIT's portability, namely involving a mini-computer this time. The other is the extension of that design to a new application, namely an interactive information retrieval subsystem of a data management system.

A third component of the effort links the first two while at the same time providing another new environment for portable programming, a batch program which moves data from PLANIT into data base format for the information retrieval system.

Finally, a much needed component, providing yet another operating environment in which to test transportable coding design, an interface test program was developed which checks out a target installation interface, either verifying the correct performance of all of its functions or providing corrective diagnostic feedback for those functions which may be executing improperly.

The contract also provided some technical assistance and some routine kinds of maintenance changes to PLANIT which have been documented in monthly reports. However, the research results which are of interest in this report center on the above four major efforts.

PURPOSE

The purpose of this effort was to extend the range of PLANIT's applicability to a new class of computers (i.e. mini-computers), and to extend the transportability design from PLANIT to different kinds of computer software applications.

Enlarging on the first part, PLANIT was first demonstrated on several conventional commercial digital computers. Although these differed in all the ways that could be expected (e.g. word sizes, character sets, input/output conventions, etc.), yet all shared operating environments which had many similarities, including FORTRAN compilers, utility software for real time applications, etc. The ARI effort that put PLANIT on TACFIRE was a test of another magnitude to see if PLANIT would operate in a completely foreign environment. PLANIT now runs on TACFIRE, performing as well or better than most of us had hoped. Not only does it execute in an identical manner to the commercial counterparts, but training scenarios prepared on either run without change on the other.

In further probing the limits of PLANIT's applicability, ARI undertook the sponsorship of the instal-

lation of PLANIT on yet a different class of computers, the mini-computer. In its various publications, the PLANIT literature has often stated that its size and design make it unsuitable for mini-computers. In the face of that, ARI determined to test its feasibility. The results were completely successful, the details of which contribute to a large section of this report.

Enlarging on the second part of the purpose (i.e. to extend the limits of the design's applicability), the results are very encouraging and clearly show the direction that further research ought to take.

In particular, this investigation sought to dissect from PLANIT a discreet subsystem that might be identified as its operating system. Called MAGIC (Machine Adaptable Generated Interactive Console), that operating system was then to be used to drive a different computer application. During the course of the effort, that application became defined to be an information retrieval subsystem for a data management system, to be patterned after the TRW GIM II information retrieval language. Being primarily a research effort, the goal did not encompass a complete system, ready for export to a user community. Rather, this version was to be suitable for demonstration purposes.

The reader is encouraged to continually be reminded of these research-oriented purposes as a frame of reference for the discussion that will follow on each of the component pieces of the project.

INSTALLATION OF PLANIT ON A PDP 11

In the early years of PLANIT's design, beginning in 1968, a class of computers were commercially available that were being called mini-computers. The Digital Equipment Corporation's PDP 8 computer was one of several that shared certain similarities, making them a part of the mini-computer class. Although the first PDP 8's were about as large as a desk, newer versions were soon marketed that would easily sit on a desk, being little larger than a breadbox. These computers generally had architecture of 12-bit words (some more, some less), and a simplified set of basic instructions.

Having had some prior experience with a prototype for PLANIT, it was completely clear that computers of this sort would not have the size or sophistication necessary to run a portable PLANIT. Therefore, it was necessary to determine a reasonable lower limit of hardware architecture in order to provide the necessary guidelines to the programmers of the system. Such things as higher level language capabilities and the magnitude and precision of numbers that could be used demanded such guidelines.

In order to choose these parameters, some estimates were calculated of what the finished PLANIT system might look like. Some of the important calculations from that effort were core sizes to run the system and data characteristics which would determine the dimensions of memory needed. Thus, PLANIT was projected to need the equivalent of 256,000 characters of core memory to run well, and a minimum of 128,000 characters of core to run a completely scaled-down version. Upper limits of core at that time for mini-computers were about 32,000 characters with a few allowing twice as much.

In addition to core considerations, mini-computers typically allowed limited mignitudes in the number representations in memory. For example, a 12-bit computer can represent at most 4095 (or ±2047). Using two computer words to increase accuracy consumes core twice as fast and involves much slower execution to do simple arithmetic.

Therefore, a class of computers exemplified by the CDC 3300 and the SDS 940 were chosen to be a practical

minimum for the future installation of the PLANIT system then being designed. These computers held numbers of the magnitude ±8,388,607, or nearly seven full signed decimal digits. In addition, these word sizes would permit the addressing of core in the amount that was projected to be needed. So PLANIT was programmed under the assumption that data could be manipulated whose magnitude did not exceed 2^{23}, nearly seven decimal digits.

PLANIT was also designed and programmed in a completely modular fashion so that the overlaying configuration could readily be adapted to fit whatever environment was presented. However, it was known that configurations of fewer and larger segments of program would execute much more rapidly than many smaller ones, and it was assumed that there would be a lower practical limit that would be much different than the lower real limit, namely that PLANIT could be cut into such small pieces that execution delays would be intolerable. That assumption is probably still true in that no known hardware is currently available which would execute PLANIT with reasonable response time if it were cut up as finely as possible.

However, ARI demonstrated through its PLANIT installation on the DEVTOS 3300 that PLANIT could be run in less core than earlier thought and still maintain acceptable interaction rates with several users. The DEVTOS computer had the required word size but was short on available core due to the tactical operating system it was running. So PLANIT ran in the equivalent of about 72,000 characters of core, or about 28% of the amount which had been estimated for an operational system, or little over half of the projected bare minimum.

Now, enter a new class of mini-computers. With Large Scale Integration and MOS chips, new tabletop computers became commercially available that were little larger physically than the PDP 8 variety but contained electronic sophistication formerly found only in the medium and large scale computers. The Digital Equipment Corporation's PDP 11 computer is typical of this class. Although word sizes were still lacking (16 bits compared to PLANIT's required 24 bits), double word addressing instructions were making it possible to concatenate two words into an effective 32-bit word length with very reasonable

execution efficiency, and addressing hardware such as Memory Management was making possible core sizes up to 248,000 characters.

With these new hardware advances, and having determined that PLANIT could operate satisfactorily in smaller amounts of core than first projected, ARI undertook to establish the feasibility of installing PLANIT on a PDP 11, chosen because it was representative of its class and one frequently found at Army sites.

Choosing The Target PDP 11 For Installation Of PLANIT

Rather than being a single computer, the PDP 11 is a family of computers, from the PDP 11/03 to the PDP 11/70. Of course, all basically have the same hardware architecture and are upward compatible through the various models. But there are also many variations, particularly with regard to their ability to provide the services formerly reserved to larger computers. For example, until recently the RSX-11D Real-time Multiprocessing Operating System required at least a PDP 11/40 plus several hardware options, with the PDP 11/45 needed for other software applications. Other operating systems such as the RT-11 run on the lower numbered computer models such as the PDP 11/04, 11/05, 11/10, etc. Newer models are being introduced such as the PDP 11/34 which provide capabilities formerly found in the higher numbered models.

Thus, it became a challenge to find the combination of the most economical model with the fewest necessary extra options that might have a chance of running PLANIT. Some of the considerations were:

- Core size
- Double-word integers
- FORTRAN IV compiler
- Ability to run an overlaid program
- Ability to communicate with multiple user terminals
- Alternate data storage on a suitably fast disk

Expanding on the above six points just a bit, minimum core size was determined to be 44,000 16-bit words (actually addressing 48,000 words but the upper 4,000 words on a PDP 11 are always reserved for device addressing). Considering the core range of 8,000 to 124,000 words, that figure was chosen out of consideration of several factors:

- The largest user program that can be task-built on a PDP 11 is 32,000 words, determined by the addressing limit of the 16-bit word. The UNIX operating system, developed by the Bell Labs for the PDP 11 will handle a user program twice that size, but it was felt that standard vender-supplied software should be used if at all possible. The additional 12,000 words are adequate to run the RSX-11D operating system utilities, the choice of which resulted from other considerations, below.

- Double-word integers were needed to handle PLANIT's larger numbers, and these were available only in the PDP 11's FORTRAN IV PLUS compiler, and this in turn needed the RSX operating system, the Extended Arithmetic option and the Floating Point Processor option. The Memory Management option was also needed to handle the addressing of core above 32,000 words. The PDP 11/40 or 11/45 was needed to accomodate the necessary configuration (actually there is little practical difference between the two with all of the above options in place). While it is true that there are other ways to get double-word integers on the PDP 11 beside the FORTRAN IV PLUS compiler with the Floating Point Processor, these all would have added significantly to the installation time where it was not yet known whether the installation would even be successful. Therefore, the more sure alternative was chosen.

- The RSX operating system also provided the necessary Object Time System utilities for most easily serving the needs of the real time interaction of the terminals, as well as the overlay structure of the PLANIT system.

If PLANIT was to run on the PDP 11 using vender supplied software, it was clear that it would have to run in 32,000 words (i.e. the equivalent of 64,000 characters). This would be even less than the DEVTOS installation before the loss due to double-word integers was subtracted. To say that the prospect of success for the installation at that time was less than certain is an understatement.

It should be noted that just having the core available is not enough. There must also be the means for shuttling program overlays through the core plus the hardware for performing double-word arithmetic. All of these factors seemed to point to the RSX operating system, where the 44,000 word core seemed to be a reasonable practical minimum for the RSX-11M and in fact is the minimum in the case of the RSX-11D operating system.

The 44,000 word core is not generally a limiting factor for PLANIT installations for a large number of extant PDP 11 sites. Nearly all of the several sites that were contacted had at least that much and usually more. Additional core is one of the lesser expensive options for the PDP 11. It would probably have been acceptable to assume even more core were it not for the 32,000-word limit imposed on standard compiled user programs. To exceed that limit would impose more difficulty on future PDP 11 installations since each would have to implement some non-standard core-extending programming technique.

Having chosen the RSX operating system in 44,000 words of core with the FORTRAN IV PLUS compiler, the problem of integer size was solved. The FORTRAN IV PLUS compiler has an "I4" switch which, if set "on", will cause all integers to be executed in double word fashion. However, sufficient core would then be critical since double word integers not only require twice as much data space but also require extra program instructions to execute properly. In order to see how much difference it did make, a segment of PLANIT was compiled both ways, with the "I4" switch off and on. Then the compile maps of the two were compared. With the "I4" switch "on", the program size expanded 42% and the program data expanded 89%. Thus it is obvious that if the double-word requirement could be removed, a significant amount of core could be put to more productive use. More will be said in this regard later.

The final consideration in choosing an appropriate PDP 11 was to insure that it had a suitable disk.

- Any of the standard disks (other than the relatively slow "floppy disk") was thought to be adequate, with the fixed-head disks being somewhat faster than the cartridge disks. However, even the 70 milli-second access time of the RK05 cartridge disk could be tolerated, giving the advantage of removable cartridges where this is often the most convenient way of taking the data out of the PDP 11 system (not too many have standard magnetic tape units attached). As a matter of fact, the system used in the installation did have RK05 disks. If the same code was used with the faster fixed-head disks, the responsiveness of the PDP 11 PLANIT would improve.

Therefore, the search was on for a PDP 11/40 or PDP 11/45 running RSX-11M or RSX-11D, having the FORTRAN IV PLUS compiler with the additional Floating Point Processor option that it required. It is a matter of record in the monthly reports that it took six months to find such a configuration on which time could be made available, and then the site lacked the FORTRAN IV PLUS compiler but was willing to purchase the software license in order to sell the time. The effort expended in the search is incidental to the project report even though it had profound effect on the scheduling and budget of the project. However, the difficulty encountered in finding such a system does say something about future changes that might be made in PDP 11 installations of PLANIT in order to make them adaptable to a more readily available configuration of the PDP 11.

As it turns out, while RSX configurations of the PDP 11 can be found relatively easily, it seems that those who need that configuration are generally engaged in work of such a special nature that they are not in a position to make time available to anyone else. This is not to say that PDP 11 time is hard to find. On the contrary, time on the PDP 11 RSTS/E time-sharing system seems to be available everywhere. Most universities who run the PDP 11 seem to run that operating system. Several commercial time-sharing venders also market RSTS/E time. RSTS/E will run overlaid programs of sufficient size and has a FORTRAN IV compiler, but unfortunately, that compiler does not accomodate double-word integers and the RSTS/E operating system does not accept the FORTRAN IV PLUS

compiler. It is this combination of circumstances that lead to the suggestion to be made a little later about modifying PLANIT to eliminate double-word requirements on 16-bit computers. Given that change, the number of PDP 11 installations which could run PLANIT would increase dramatically.

Installation Experience On The PDP 11

Arrangements were made to use a PDP 11/45 at the Center for Computer Based Education located on the campus of the University of California at Los Angeles. Two weeks were required on site to complete the installation. CCBE provided expert system assistance which was absolutely necessary to the success of the installation.

It is also worth mentioning at this point that ARI's foresight in allowing for a failure to install the system to be an acceptable alternative might easily have made the difference whether the project was ever completed. Aside from any reluctance the project team may have had to undertake the project if a successful installation was required, it also affected arrangements with the computer site. The CCBE personnel were sufficiently experienced to recognize the project as having a relatively high risk of failure and they would not have accepted the arrangements for use of the computer unless a possible failure was acceptable.

The PLANIT Generator program was not installed on the PDP 11 although it was the intention to do so. Even though there was no contractual reasons to install the Generator program, that seemed at first to be the most reasonable way to proceed. However, the Generator program is not as modular as PLANIT and would not compile because of its size. Not that the Generator program would exceed the available 32,000 words of core, but the FORTRAN compiler will only compile chunks of about 1,400 lines of code (approximately 6,000-8,000 words of core) at a time and these compiled pieces are then put through a Task Builder on the PDP 11 in order to utilize the core as fully as desired. So, rather than spend extra time making the Generator program fit onto the PDP 11, it was installed on the PDP 10 computer which is also running at the same site. The PDP 10 was then used to generate the PLANIT code (in FORTRAN) for compilation on the PDP 11. This procedure worked very smoothly.

Having worked into the second week and finally having all the pieces for PLANIT including the twenty-five overlays, the MIOP, LDBYTE and SBYTE interface programs ready to Task Build, it was then discovered that we exceeded available core by more than 2,000 words. At that point, failure appeared to be a very real alternative. However, it was discovered that the pieces could be recompiled without a debugging "trace" option, effecting a significant reduction in the size of the compiled modules. This meant that the PDP 11 system would give virtually no cross-referencing kind of help in the event of an error but we were willing to live with that, especially since the PLANIT code is relatively dependable.

Without the "trace" option, the pieces all fit into the available core. The statistics of the core map follow:

• Total core available (for PLANIT)	32,768 (words)
• Common Data	8,084
• MIOP	1,974
• LDBYTE & SBYTE	48
• Main Program	4,751
• Overlay Space	6,498
• Additional PDP 11 utilities	11,056
• Total core used	32,411

It appears that PLANIT just barely squeezed in. However, it actually fit into 30,464 words of core and the main program was later enlarged by adding some critical code from one of the overlays to speed up the execution. While the smaller version was running, we experimented with another (very small) PDP 11 program running concurrently and found that it did in fact run satisfactorily.

This configuration makes PLANIT overlay more often than any other current installation. Other measures also were required to keep the size down, such as:

- Restricting the number of frames in a PLANIT lesson to 50 (instead of the usual 100)

- Reducing the number of initial characters kept in dictionary representations of names to four (from the usual eight or more)

- Reducing Calc work space to less than usual

However, despite these reductions, every PLANIT system function is operational on the PDP 11.

Four terminals were active on the PLANIT system and all operated normally. There were three of us available to provide concurrent interaction. While there was not opportunity to gather accurate response time data over a range of activities, response times that were observed varied from one to five seconds for replies which are usually immediate (e.g. lesson building) up to twenty or more seconds for operations where some delay is normal (e.g. finishing lesson execution). Response delays were usually small enough to be tolerable.

Improving The Performance Of PLANIT On The PDP 11

It was already mentioned earlier that, where a choice existed, the more complex installation procedures were avoided since the first priority was to determine whether PLANIT would fit into the PDP 11. It would be poor judgment to spend more than an absolute minimum to find that out. Now that the project has established that PLANIT will run (in face, does run) on a PDP 11, there are several endeavors which could be undertaken to make it run faster and even to restore some or all of the space that was reduced. Three possibilities might be considered, each of which would save some core space which could be reallocated to improve the operation of PLANIT: 1) reduce the 24-bit minimum word size in PLANIT to 16 bits, 2) modify MIOP to eliminate as many of the PDP 11 utilities as possible, and 3) move some program functions into the next 32,000-word partition of core. These three possibilities will now be considered in more detail.

First, the reduction of the 24-bit requirement to 16 bits would have made a significant difference in the space used by the Common data and by the program code. It was mentioned earlier that compiler tests indicated what the savings would be. Double word compiling caused the data to expand by 89% while the program code expanded by 42%.

The 89% expansion figure for the data was due to the fact that not all of the data items were integer. If they had been, the expansion would have been 100%. However, no change occurs in the space needed for real (floating point) data when the precision of the integer data is changed.

The 42% expansion figure for the program code is due primarily to the additional instructions which become necessary to process each of the two words that make up the integer data item. Since most of the program involves integer data, much of the code is affected.

In the current PDP 11 PLANIT installation, there are 8,084 words of data and 11,249 words of PLANIT code which make reference to it. Applying the above percentages, 8,084 is an 89% expansion of 4,278 (8,084/1.89=4,278) and 11,249 is a 42% expansion of 7,921 (11,249/1.42=7,921). Therefore, if single words were used for integer data, the savings would be approximately 7,134 words ((8,084-4,278)+(11,249-7,921)=7,134). Note that 7,134 words represents 22% of the total size of the system, and is larger than MIOP or any one piece of the PLANIT code. That space could be reallocated to provide more lesson space as well as to put more of the frequently used sections of code into the main program where it would execute faster (because no overlaying would be necessary). This option alone could restore all of the usual lesson space to PLANIT and improve response time a little, too.

In order to incorporate this option, all of the present PLANIT code would need to be checked for use of integer values in excess of ±32,767, and where found, different algorithms would have to be written to reduce the size of the numbers used. This would be a big task but it would yield a sizeable benefit (22% savings of core).

Since the 24-bit minimum could have been avoided when the system was originally coded, one might wonder why now choose 16 bits? Why not less and avoid a similar mistake later? The next smaller size which would make any sense in the computer industry would be 12 bits. This would impose a limit on the integer size to ±2,047. It would not be reasonable to impose such a small limit on the coding. Indeed, that small a limit would cause the programming algorithms themselves to mushroom in size, defeating the purpose.

There will necessarily be some expansion in coding when different ways are devised for processing numbers whose size exceeds the 16-bit limit. However, 16 bits provides four full digits of accuracy which will be adequate in most cases, and will hold at least half of the number in the remaining cases. Also, in order to avoid penalizing larger sites which have the required word size, the Generator language permits alternate coding such that appropriate program statements are automatically selected according to the parameters which are used to describe the target computer. Since the number of bits in a word is already a parameter, the new coding which accomodates to 16-bit words could be selected only if that parameter value was less than 24.

The second improvement which was suggested was to improve MIOP and thus eliminate some of the PDP 11 utilities. Notice that the largest single segment of the core map of PLANIT on the PDP 11 was the collection of utility programs. These accounted for 34% (11,056/32,411=34%) of the total. Much of this is a collection of pieces belonging to the PDP 11 Object-Time System. These software utilities provide resources for real-time operations of one sort or another. It is likely that PLANIT does not use (or need not use) at least half of these utilities. Much of what they do surely is redundant with what PLANIT is already doing. For example, it is not necessary that the terminals be queued for PLANIT, that is already being done.

The task would be to become familiar enough with each of the components of the system utilities so that their use could be bypassed and they could then be eliminated from core. There is good reason to believe that 5,000 or more words could be saved in this fashion and reallocated as suggested above.

The above option suggested that the present program and data could be reduced to 7,921+4,278=12,199 words, saving 7,134 words. If another 5,000 words could be saved from the utilities, that amounts to a total of 12,134 words that could be reallocated back into the PDP 11 PLANIT, effectively doubling all of the present PLANIT capability on the PDP 11. These are not unreasonable expectations.

The third (and final) improvement which was suggested above is the moving of some of the functions into the upper regions of core, above 32,768 addresses.

This may be the most questionable of the improvements simply because it would involve some unconventional coding. In effect, a discreet part of the support package for PLANIT (such as terminal processing) would be removed and made into a separate program which then would run in the higher partition. There are ways that this new program and the remaining PLANIT program can be made to exchange data. Therefore, the support program in high core would do its work independently of PLANIT and, when finished, would pass the results over to PLANIT, and PLANIT would do likewise. The savings would be effected by the fact that another part of the 32,000 word program space would then be made available for more productive use by PLANIT. It is too difficult to project how much that might be but it could be several thousand words.

Another possibility would be to use high core to queue PLANIT overlays so they could be retrieved more rapidly than from disk. Again the coding would be unconventional but the increase in execution efficiency could be significant. This option would probably mean that at least 64,000 words of core would be needed on the PDP 11.

Concluding Remarks For The PDP 11 Installation Effort

The project has established the feasibility of installing PLANIT on the PDP 11. In addition, it appears quite certain that the installation could be optimized to both run much faster and utilize more common PDP 11 hardware/software configurations.

It now appears to be quite reasonable to run PLANIT on model numbers as low as the PDP 11/34 and perhaps lower. Also, with the benefit of the proposed changes, the FORTRAN IV compiler could be used in place of the FORTRAN IV PLUS compiler, and it may be possible to run PLANIT without the Floating Point Processor hardware with little degredation in response time.

If these options are implemented, then PLANIT ought to execute very satisfactorily on the "average" PDP 11 hardware configuration.

DEVELOPMENT OF THE PORTABLE QUERY SYSTEM

The Portable Query System is an interactive information retrieval component of a data management system.

The primary purpose of the development of PQS was to test the application of the transportable coding procedures and the application of PLANIT's operating system to non-PLANIT software. Therefore, the immediate purpose for this software, which is nearing completion as of this writing, is a successful demonstration of on-line queries from a data base supplied by ARI.

Consistent with the purpose for the PQS software, this report will analyze the application of the transportable coding procedures rather than discussing the design of the Query system. The reader who may be interested in further information on the Query system will find a statement of the system specifications in Appendix A and a PQS User's Manual in Appendix B.

There are three main topics of interest in regard to the transportable coding procedures, each of which will be the subject of one of the following sections.

Application Of Transportable Coding To PQS

Transportable coding procedures were devised several years ago in connection with the development of the PLANIT software. It amounted to an exhaustive examination of each of the elements of conventional computer programming which tend to link the resulting software to the specific kind of computer for which it was written, even occasionally limiting the execution to one specific machine.

Machine dependencies are typically introduced into a computer program from several sources, including hardware architecture, character sets, instruction repertoire, input/output device characteristics and operating systems. Any one of the above can account for machine dependencies in software, and more often than not, all of them influence the software.

Conversion of software from one computer to another is rarely a trivial task. It is common to spend more than half of the original effort to accomplish the conversion.

The PLANIT development effort determined to avoid this high program conversion cost by finding an acceptable commonality among computers for the purpose of writing software code which could run on most of them. This involved three essential elements:

- Determining what software coding conventions would be acceptable on all target machines
- Developing a method for automatically modifying the code for additional hardware differences where commonality was impractical
- Defining a clear and comprehensive set of procedures for installing the completed system on any computer where the installation interface would be functionally identical on all of them.

There are several subgoals which were of equal importance if the method was to have any chance of acceptance, such as not compromising the capability or efficiency of the target hardware. One cannot simply reduce all machines to their lowest common denominator since the result would probably not be adequate and certainly not appealing to any programmer.

One of the early decisions was to make use of some higher level language in the transport process. Compiler programs for these languages are already considered to be an absolutely essential part of nearly any computer package, and these compilers have done much to make many of the machine differences transparent to the user. For example, simple arithmetic expressions look nearly alike in many different languages and on many different computers even though the machine instruction sequences into which they get compiled will often be quite different.

FORTRAN was chosen to be the most commonly used higher level language for the mediating process of the transport of the PLANIT program. The reason for this choice was because of the wide availability of FORTRAN compilers, not because it was any more suitable. The procedure is not entirely dependent on the use of FORTRAN but has had only one case so far where

FORTRAN was not the appropriate choice, the case of the TACFIRE system on the ANGYK-12 machine, where TACPOL was used instead.

Having determined through several prior case studies that PLANIT was being installed on a variety of computers pretty much as predicted, ARI became interested in wider applications for the method. Since PLANIT is a very large and complex software system, it already contains many coding examples which have features in common with nearly any programming algorithm one can think of. Thus, of all the typical problems that one might expect in the design of this new software, the constraints necessary to maintain transportability never once entered into the problem nor were any of the design problems compounded by the need to meet these standards. The situations had all been met in some form in earlier experiences with PLANIT.

Since issues involving transport were not a particular concern, there was some interest in comparing coding efficiencies. Although there was no opportunity to make rigorous observations, it appeared that the coding effort may have been more difficult by as much as ten percent owing to the need to maintain standards for transport. However, given comparable experience, the differences might be reduced.

One aspect of this coding method soon becomes very clear, machine characteristics play no part in defining the coding task. There are many examples where program specifications undergo slight changes during coding due to accomodations to certain hardware features. Such is not the case for transportable coding for, if any particular machine feature were allowed to influence the task, the transportability would certainly be compromised.

The reader might well be wondering by now just when this transportable coding method will be explained. Unfortunately, such an explanation would go beyond the resources for this report. However, an experienced programmer would probably find little that was novel. Rather, the method consists of a collection of known procedures for improving the transportability of programs, such as character mapping, simplistic data organization, execution-time data initialization, table-driven input and output, etc. What distinguishes this method is that a carefully chosen set of such procedures

were assembled into a comprehensive "language" such that the programmer follows specific guidelines in the coding effort. Therefore the programmer does not have to be overly concerned about the matters of transport so long as the method is followed.

Those who seem to find the most difficulty with the method are those who have had prolonged experience with a particular kind of computer. They tend to think in terms of the architecture of that machine instead of the framework of the language. A typical orientation time for such an experienced programmer to this method is about six months during which three typical phases are traversed: disbelief (it can't work), unlearning and relearning. In some respects it is easier to train one who is new to the business, or at least has not become entrenched in a particular kind of hardware. One makes fewer coding errors if certain characteristics of the machine are not known (e.g. bits or bytes in a computer word, character set collation sequence, etc.).

Based on the experiences with the method, first with PLANIT, and now with the Query system, there appear to be no known drawbacks which would preclude further experiments in other applications.

Application Of The MAGIC Operating System

A computer operating system logically integrates a user's program into the operating environment of the computer, scheduling and supervising its execution, providing communication with peripheral devices, etc. The fact that PLANIT included an operating system was never in question because it couldn't function without it. However, PLANIT's operating system was not a separate, distinguishable entity.

Part of the current effort was to extract from PLANIT the part that could be identified as its operating system, and use that as the starting point for the development of the Query demonstration system. Since PLANIT is a time-sharing system, that also characterizes its operating system. That which was extracted from PLANIT was called MAGIC (Machine Adaptable Generated Interactive Console system).

The MAGIC system is of interest not only because it controls the Query system but also because of its potential use in other transportable program environments. Essentially, it consists of a set of procedures for managing the data communication between each of the peripheral devices, and allocating the resources of those devices in an orderly manner to each of the users.

It was not particularly difficult to extract MAGIC from PLANIT and to instead put it in control of the Query system. A brief examination of the two systems, PLANIT and PQS (Query), will quickly show the code that is common to both, which has been given the name, MAGIC. It was also used for the PLANIT Interface Test program which will be described later. No additional difficulty was encountered. Time-sharing systems have typically been some of the more delicate to make work properly and it some sense of achievement to see it so easily adapted to other applications without incurring problems.

However, as an operating system, MAGIC has one serious flaw. It necessarily takes on some of the characteristics of the user programs it supervises.

The MAGIC system was never developed as a stand-alone operating system. When used with PLANIT, it has some PLANIT data structured within it. When operating with PQS, it reflects some of the structure of that system. The reason that condition is considered to be serious is that the MAGIC operating system must be recompiled for each new application it controls. This would be equivalent to buying some time-sharing service and then requiring that the vender modify the system to accomodate your program. In order to be viable as an operating system, the MAGIC system will need to be redesigned to become functionally independent of the programs being supervised. Since this question is interrelated with the next topic, it will be left here and taken up again.

Communication Among Systems

Interest has been expressed in using PQS along with PLANIT so that the PLANIT tutorial capabilities could enhance the user's ability to compose good query statements. In fact, the question of using

PLANIT with other software systems on the same computer has come up frequently. Using a generalized "Program X" to identify the other software that is to operate with PLANIT, the question logically divides itself into four separate considerations:

- Is Program X to become a part of the PLANIT system?
- Are both Program X and PLANIT under the control of the MAGIC operating system?
- Is PLANIT to control more than one user?
- Can Program X be modified to suit specific needs for communication with PLANIT?

These four questions will comprise the outline for the discussion regarding the possibility of communication between PLANIT and other programs.

Is Program X To Become A Part Of The PLANIT System? This question implies that the code for the other program (Program X) is simply added to the PLANIT code, using the same data formats, in effect becoming an extension to PLANIT.

This option is certainly possible. It can easily be demonstrated in the present PLANIT code. A capability now exists within PLANIT for the creation and editing of card image files. This is separate and distinct from lesson creation and editing. In PLANIT that subsystem is called EDTEXT. It is functionally separate from PLANIT's instructional functions other than its use for manipulating decks which have been keypunched off-line.

In general, this option is no longer desirable. One of Murphy's Laws state that "a program will grow to occupy all the space available." It becomes a little difficult to relate Murphy's Law to PLANIT, at least until installation time in which case it may be discovered that it has outgrown the available space.

PLANIT's present size was already a major factor in the PDP 11 installation effort. There are critical needs for PLANIT in the area of instruction, such as the capability to display graphics. Therefore, it would seem to be unwise to encumber PLANIT with other,

non-related software tasks. There can be some exceptions. In the case of EDTEXT, for example, that subsystem almost entirely resides on disk when not in use. Its presence in the PLANIT system has added at most the equivalent of one or two FORTRAN statements to the core-resident code. Everything else it uses, including the dictionary for its language syntax, is borrowed from PLANIT's instructional components. Its only impact on the system is in the additional overlaying it causes while being used, and since it is used so seldom, that, too, is negligible.

There are not many applications like EDTEXT which could be added to PLANIT with so little impact to the system. In general, one would assume that PLANIT would grow significantly with the addition of the new package. Even though the communication between the old and the new (all within PLANIT) could then be nearly anything desired, this does not seem to be an acceptable option.

Are Both Program X and PLANIT Under The Control Of The MAGIC Operating System? This would seem to be the preferrable way to establish communication between PLANIT and some Program X. Although there is no definite design for accomplishing the communication link, it should not be too difficult to work out.

The main objection to this option is found in the MAGIC operating system itself. The earlier section dealt with the problem in the MAGIC system, that of having to incorporate some of the characteristics of the program it moniters within the compiled code. This has two implications, both unacceptable:

- The MAGIC system must be modified and recompiled for each new subprogram it supervises.
- The MAGIC system grows in size each time a new subprogram is added to its repertoire.

The MAGIC system as identified in PLANIT is slightly different from the MAGIC system as modified for PQS. If both PLANIT and PQS were to run concurrently, the MAGIC system would have to contain supporting elements for both. If the capability to run yet a third subprogram was to be added, MAGIC would grow again. This is an unacceptable trait for an operating system.

In order to make MAGIC a viable operating system and to make the concurrent running of subprograms a

good way to achieve desired intercommunication, MAGIC should be redesigned to become an independent operating system such that new subprogram applications could be accomodated without change to MAGIC.

As indicated before, the part of PLANIT which is now called MAGIC was not designed to stand alone as an independent operating system. It was not difficult to adapt it for the PQS application, but the adaptation was no more independent than the original.

It would be difficult to predict what all would be involved in producing a stand-alone version of MAGIC. The nature of the task would be to develop a machine independent, transportable operating system which functions much like present day operating systems. It would seem to be feasible to use the same guidelines for transportability as have been used for PLANIT and PQS since most operating systems are written in higher level languages. To be sure, it would have to incorporate far more generality than the present MAGIC system.

If such an effort should be undertaken, and prove to be successful, then both PLANIT and PQS could be modified to run under that system rather than the present MAGIC system. Then it would become profitable to work out intercommunication channels between subprograms.

This option would seem to have the greatest payoff in the long run, especially because of the advantages offered by a transportable operating system. Operating systems now constitute environmental differences among machines second only to the hardware differences. The ability to use the same operating system on different hardware would greatly reduce the entire transport problem. There are many informed people who would give this prospect very little chance for success, but so also they did for PLANIT.

Is PLANIT To Control More Than One User? The above discussion suggested that a stand-alone MAGIC system be developed which would supervise the various programs beneath it, and thus play a dispatcher role for any interprogram communication. If that option was adopted, this present question would be irrelevant.

Having PLANIT in control of more than one user is a matter of importance only if PLANIT, being run in one of the presently conventional ways, attempts to communicate with some other program on the same computer which is not related to PLANIT or its operating system.

Thus, this is a question of getting interprogram communication with PLANIT going today, without extensive developmental work.

The thrust of the question is whether PLANIT execution can be suspended while communication has shifted to the other program. If PLANIT is running in the one-copy-per-user mode, as it usually is under a host time-sharing system, then execution can be suspended while the user is interacting with the other program, and resumed when control is shifted back to PLANIT again.

However, if PLANIT is running more than one user, PLANIT itself cannot suspend execution since it must continue to provide service to its other users. Thus, the one user who, by reason of interprogram communication, shifts interaction to the other program, must be treated as a special case in PLANIT. It will be as though the lesson scenario was interrupted for any reason (e.g. quitting for the day), except that execution should resume automatically upon the return to PLANIT. This will require some new capabilities to be added to PLANIT, including new author directives to initiate the program switch, and probably some new MIOP interface algorithms accomplish the desired action.

If these changes are made to PLANIT, then they could be used without regard to the number of terminals PLANIT is currently running. Given sufficient planning, the same directives could be used to initiate interprogram communication if both programs were running under the MAGIC operating system as well, having the advantage that lessons would work without change in either environment. With this option, the burden would be on the PLANIT installer to determine how to shift communication over to the different program, temporarily reassigning the terminal, loading and executing the program, and all else that would be entailed. The changes to PLANIT to permit this, although somewhat difficult to design, should not be extensive.

Can Program X Be Modified To Suit Specific Needs For Communication With PLANIT? Put another way, is Program X expecting the intercommunication with PLANIT, or can it be made to? This question relates to the nature of the other program and how much freedom exists to modify it for best intercommunication results with PLANIT. In the case of PQS, one would assume that any accomodations deemed necessary could be built in since both have

a common developer and share many other similarities as well.

Some accomodation may also be possible with other software which has had no prior association with PLANIT, if the developer is available and the authorization is granted. In these cases it would not only be possible to switch the user to the other system at convenient places in the scenario, but it would also be possible to send some data along which would help make the interaction in the other program more pertinent, or in the case of a return to PLANIT, report back some indication of performance.

There will also be cases, perhaps the majority of them, where no accomodation can be made in the other program with which PLANIT desires to communicate. Two possibilities still remain for such situations:

- When switching to the other program, mark the place so that execution will return to PLANIT after the other program runs to completion.

- Write an add-on front-end processor for the other program which examines the input and output messages on the way to the computer (or user), and initiates appropriate switch action based on the content of the messages.

Either option can allow the other program to run unmodified. However, without the front-end processor, a user who has been switched to the other program will be outside of any possible supervisory control needed for training.

The front-end processor will probably be more costly and difficult to develop than building the accomodation into the other program. However some cases simply do not provide the choice.

Probably any attempt to make PLANIT communicate with another program will involve some design and modification work, almost certainly to PLANIT, and most likely to the other program as well. The addition to PLANIT should be a one-time effort which ought to handle a variety of situations. The modification to the other program will need to be specific to the requirements of PLANIT.

Assuring Portability

One of the most difficult aspects of portable programming is the discipline required to adhere to the programming conventions. This might at first look like an obvious kind of statement but the problem arises because the finished program might execute properly and still not be right. The error will not show up until it is installed on a different computer.

Some portability conventions are enforced by the software which is used to aid program development and installation. However, it is not possible to protect against all possible errors of this kind. There are several examples of this kind but the most common is mistaking a given design value in the local computer as a constant across all computers. Thus, one might make this mistake by assuming that entries in a particular dictionary are always composed of a constant number of words when in fact the number might vary depending upon how many words are used to hold a user-supplied name. This could vary from two characters per word to ten, depending on the computer.

Various ideas have been explored to detect such errors during program generation but these numbers are so interrelated with the program logic that no automatic syntax check has been found for them.

There is another procedure which does appear to hold promise. To be sure, such errors will be found as soon as the move is made to another machine. If the originating computer is sufficiently large, it might be made to simulate another computer for purposes of checking the conversion logic of the newly developed program. The simulation would not be nearly as comprehensive as it might first appear.

Most of the errors will be made through faulty references to data where the references were not properly adjusted for differences among machines. So long as the originating computer exceeds by at least one byte the minimum number of bytes per word, a second MIOP interface program could be written which assumes the same machine but with smaller word size, effectively discarding the extra byte space in each computer word.

Then, after a new program had been completely checked out on the primary MIOP installation interface and determined to be free from errors, it could be run again under the second MIOP interface which would provide many of the conditions to be encountered during transport. Such a test would significantly increase the confidence in the transportability of the finished product.

DATA BASE CONVERSION PROGRAM

PLANIT automatically keeps a detailed data record on all trainees who are receiving instruction. This data record is active during the course of the training, providing historical information so that the lesson scenario can be continually adapted to meet a particular individual's training requirement. Following the instructional session, the record is also available to the lesson author for inspection. However, as is true for so many computer-kept data bases, the mass of data is normally so great as to discourage most authors from using them beneficially.

The Portable Query System is designed for problems of extracting relevant information from large data bases. Using the PQS, one can specify in the query statement what kinds of data are to be reported for particular needs.

The Data Base Conversion Program (CONVERT) translates PLANIT's student record data bases into the format acceptable to PQS, complete with all necessary Attribute name definitions and control cards.

The input for CONVERT consists of a set of student records, all belonging to one lesson, taken from PLANIT's UNLOAD-to-tape format. This particular format was chosen for three reasons:

- A magnetic tape provides a tangible medium for an input file.
- This particular format in PLANIT is the only one that collects all relevant records together in one file.
- The production of the UNLOAD tape will be routinely done in many instructional environments.

The UNLOAD tape containing all the student records for a given lesson comprises a comprehensible collection of data. However, the user of the CONVERT program is not restricted to that one data base input format. The job may be run repeatedly, if desired, producing a different data base for each UNLOAD tape that is submitted. Then, by removing the final (terminator) card from each output data base deck, the decks can simply be stacked together to form one large data base. One of

the removed terminator cards is then put at the end of the deck collection, and the result is ready to be submitted to the PQS system as one large data base. Thus loaded, the user is ready to formulate questions of the most general sort such as:

- What a particular student did in a particular lesson
- What many (or all) students did in a particular lesson
- What a particular student did in all the lessons
- What all the students did in all the lessons

Many different kinds of information will be available from the student record data base, including specifics about any answer the trainee gave during the course of instruction, to very general kinds of information, such as the difficulty level of various questions in the scenario. The information can be useful both to assess the performance of the students and to assess the quality of the lessons.

A complete Users' Manual for the CONVERT program is included in Appendix C. It contains instructions for using the CONVERT program and examples of the kinds of information that can be retrieved as a result.

The CONVERT program is transportable, in the same way as are both PLANIT and PQS except that no locally-written MIOP program is required. Instead, the user will review a FORTRAN READ and WRITE statement and modify them if necessary to the conventions of the computer on which the program is to be run. Also, CONVERT is a batch program rather than interactive.

THE MIOP INTERFACE TEST PROGRAM

Part of the installation procedure for PLANIT is to write three subroutines called LDBYTE, SBYTE and MIOP. The first two are very simple character moving routines, normally taking only a few minutes to code. The third, MIOP, moves data between the PLANIT program and the peripheral devices.

The PLANIT program code is adapted to the architecture of the target computer through a generation step in which the characteristics of the target machine are represented in generation parameters. This part was described earlier. However, PLANIT communicates with the peripheral devices on the target computer via an interface to a subroutine, MIOP. The interface for this subroutine is defined but the code which implements the peripheral communication must be written as a part of the installation effort.

When the MAGIC operating system was extracted from PLANIT, the same interface was kept. Thus, the same MIOP program must be written to interface the MAGIC system to the target computer as if PLANIT was being installed.

The interface is relatively simple. It consists of an array of thirteen integer numbers, where different number positions contain codes for different peripheral operations. These codes convey such information as:

- What operation is to be performed (e.g. Open, Read, Write, Close, etc.)
- Which peripheral device is intended (e.g. Disk, Tape, Card Reader, Card Punch, Line Printer, etc.)
- Where to find (or put) the data in core
- How much data should be moved
- Location of the data on the peripheral device (if appropriate)
- Where to report back a status number that will indicate the success or failure of the operation

The MIOP interface is so well-defined that the coder does not need to know a thing about the operation of the computer system that uses it. It could easily be coded before PLANIT or PQS arrives. Examples of the coding of MIOP are available but the final code must be prepared at the target site by a system programmer who knows the conventions of communicating with the local peripheral devices.

Since the MIOP interface handles many functions (virtually all of the communication with peripheral devices), some of its operations will be exercised frequently and some infrequently. For example, if PLANIT was being installed on a target computer, some of the MIOP communication options might not get exercised for a week or more after the system has been made operational. It would require more sophistication than would normally be expected to run through exercises in PLANIT which would test every MIOP function.

It is for this purpose that the MIOP Interface Test Program was developed. The Test Program uses the same MAGIC operating system that is used by both PLANIT and PQS, thus it works with the same MIOP interface and requires the same MIOP.

The Test Program is interactive. It contains a fixed, built-in scenario which is designed to systematically test each and every MIOP function that can be set up in the interface. It not only tests the operations, but also attempts to move data, both to and from the peripherals. It checks the data to make sure it was moved properly.

As the scenario progresses, the user receives comments regarding the particular operation being tested. If the test is successful, the user is so informed. If not, some suggestions are made concerning the probable cause of the error (from examining the internal data), and a general description of the repairs that need to be made is also given.

Since some interface requirements might not be implemented in the first versions of the local MIOP (such as tape operations and enforced time limits on terminal reads), the Test Program asks the user at several points if the next feature has been implemented before proceeding with the test. Therefore, the coder of the MIOP subroutine can check out subsections of it, if desired.

It is also possible that the Test Program may be used several times before the MIOP subroutine is pronounced "clean." On the assumption that some users will stop the test when an error is found, the Test Program provides opportunity at the beginning to re-enter the test sequence at any point the user chooses. The user may also choose to terminate the test by typing the word, "STOP" on the terminal.

In addition to the dialog that takes place during the test interaction, a summary is printed anytime the program is stopped or reaches its end. The summary is called a "Box Score" in which are listed each of the tests that were made during that session, and the result in one word, "Okay" or "Problem." This serves as a reminder to the user of the things that still are in need of work.

Using The MIOP Interface Test Program

The MIOP Interface Test Program was developed in conjunction with a MIOP subroutine that was known to be good. Therefore, all tests should pass when the Test Program itself was working correctly.

After this stage was reached, it was possible to program some failures into the MIOP subroutine to insure that the Test Program would detect the failures.

When the Test Program appeared to be finished and working properly, a completely new and different MIOP subroutine was written. This MIOP had nothing in common with the good MIOP that was used during development.

The MIOP Test Program detected several errors in the new MIOP during the course of several checks. Each time, the error that was detected was in fact the fault of the new MIOP and was found in the place pointed out by the Test Program. When finally the Test Program produced a "clean" Box Score for the new MIOP, it should have been finished and reliable. It was. No more work was required on that MIOP after it passed inspection by the Test Program.

The use of the Test Program was an obvious savings in this one experience of writing the MIOP subroutine. Errors were pointed out and the nature of the problem was described. That makes the debugging task much easier. Also, if the Test Program can reliably declare

a MIOP subroutine to be free from errors, this will save a great many "call backs" resulting from failures encountered at a later time. There shouldn't be any.

The MIOP Interface Test Program generates and installs exactly like PLANIT and PQS. The main difference is that, whereas PLANIT and PQS assume a working MIOP subroutine, the Test Program does not. If PLANIT or PQS does encounter an error when MIOP is called, the entire system is likely to fail in very unpredictable ways. The Test Program, on the other hand, usually continues to run in spite of a MIOP failure and then surveys the current status of the data to see what went wrong. This of course would not be true if computer execution aborted within MIOP itself.

In the few months that the Test Program has been working and available, it is a little discouraging that more installers have not gone to the extra trouble of installing and using it. They probably need to be shown how effective it is and how similar to PLANIT the installation is. Doing one establishes the procedure for doing the other.

While the Test Program apparently is very effective at finding MIOP problems, it cannot detect most kinds of inefficiency. It is quite difficult to make most tests of this sort because they depend on the computer environment under which the program executes. There is an attempt to show the transmission speed to and from terminals. Time functions are also checked and a check is also made to see if time is really being released for other functions when the Test Program is supposed to be idle (e.g. waiting for a terminal input).

If new operations are added to the standard MIOP package, it would not be too difficult to add tests in the Test Program for them. In that way the Test Program can stay current with the MIOP requirements for the code on any given magnetic tape which is shipped to a target site.

For more information on the MIOP Interface Test Program, including a description of each test that is made, the reader should consult the User's Manual in Appendix D.

CONCLUSIONS

It is likely that somewhere along the way, a software application will be presented for which the transportable method of coding will be totally inappropriate. However, for the assignments to date, the method has worked well.

The most experience has been with PLANIT. So far, PLANIT has installed on every appropriate computer attempted. A few have not been satisfied with their installations due to slow response times but these are traceable to slow routines they were using, verified by the fact that it ran acceptably fast on the same kinds of computers elsewhere.

In one case, an installation attempt, the first of its kind on a computer of that type, was dropped after six months of intermittent work. Later, an installation was successfully completed on an identical computer in one week. A lot depends on the competence of the installer.

On only one computer, the Burroughs 5500, was the installation finally declared unusable. Although it installed properly, the virtual paging hardware made it impossible to keep critical code in core to achieve reasonable response times. Newer virtual paging machines have corrected this problem.

The projects comprising this effort accomplished at least three things:

- Built confidence in the validity of the transportability concepts and procedures by demonstrating its applicability to a broader class of computers than before
- Provided additional information about the kinds of software applications which could feasibly profit from transportable coding methods
- Added new and needed tools to promote further investigations of transportable coding

RECOMMENDATIONS

Recommendations were made in the discussions of each of the topics as they were presented. Therefore, their appearance again here constitutes a collected summary of the various recommendations that have already appeared.

It is recommended that:

- The minimum target installation word size for a PLANIT installation be reduced from the present 24 bits to 16 bits, providing an extension of the appropriate PLANIT sites to those who run mini-computers.

- A thorough examination be made of the system library procedures being used in the PDP 11 installation of PLANIT for the purpose of eliminating the unneeded ones and reallocating the released core space to more productive PLANIT work.

- A feasibility effort be initiated to determine whether a transportable computer operating system can be developed that is patterned after the MAGIC operating system but has the independence from subordinate programs which the MAGIC system now lacks.

- A set of author directives be designed for and implemented into PLANIT which would provide authoring capability for intercommunication between PLANIT and another (non-PLANIT) program.

- An installation procedure be designed which would suspend a particular student's work in PLANIT while interacting with another program, and permit resumption of PLANIT interaction again when the other program returned control.

APPENDIX A

LANGUAGE SPECIFICATIONS FOR THE MACHINE TRANSPORTABLE ON-LINE QUERY DEMONSTRATION

LANGUAGE SPECIFICATIONS FOR THE MACHINE TRANSPORTABLE ON-LINE QUERY DEMONSTRATION

INTRODUCTION

United States Army Research Institute (ARI) contract number DAHC19-76-C-0022 entitled, "Research in Adaptable Programming To Achieve Computer Independence," is organized into three demonstration tasks, the third of which is to demonstrate a portion of an on-line query system which has the machine transportable qualities of a computer system called PLANIT. The language details and the scope of the effort of this third demonstration was left to be negotiated but it was generally understood to encompass something analagous to the on-line query portion of the GIM II data management system. The purpose of this document is to propose language specifications to meet this requirement.

GUIDELINES

Several general guidelines were established in a meeting between the contractor and Dr. Larry Potash of ARI on April 1, 1976 for this third demonstration effort and several more specific guidelines have evolved as a logical consequence.

General Guidelines.

1. The demonstration will meet all the contract requirements, pertaining primarily to transportability.

2. The on-line query portion of a data management system will be demonstrated, using the syntax of the GIM II system as nearly as practical.

3. Constraints must be placed on the size of the language subset chosen in order to keep the scope of the effort within the bounds of the contract time and money limits.

4. The data base upon which the on-line query operations will be made will consist of alphanumeric character records, like card images but not necessarily 80 characters long.

5. There will be some components that cannot be added to the demonstration in the present time frame which nevertheless will have such obvious importance to a fully operational system that the technical feasibility of adding them in the future must be considered and reported.

6. There will be several more areas, particularly data base creation, modification and update, which will even be beyond the scope for consideration in the present effort.

Specific Guidelines:

1. The GIM II FOR-WITH-WHERE-LIST format will be the general model for the demonstration. Specific syntax specifications will be given below.

2. Data base creation and description commands will be included for the purpose of supplying a data base for the demonstration. The syntax and/or user convenience considerations will not be relevant for these commands since their utility will be solely for the demonstration.

3. The system will permit the building of a user library of query components. However, the directives for creating, maintaining and/or protecting these dictionaries will lack the features needed for a finished system. The present objective is solely to have such a

dictionary available so that its entries can be used in on-line queries.

4. System protection for login, data base access, etc., will appear in the demonstration system but only in dummy form. No actual file or password protection will be needed for demonstration purposes.

5. Multiple (linked) data base queries will not be implemented in the demonstration system. While several data bases may be active, any given query will only search one of them.

6. No heierarchical relationships will exist between data lists, hence also no inversion operation.

7. Data base entries will occur within fixed fields (as specified in the data base description library), and will be one of two types: alphanumeric or numeric. Logical entries will be simulated so that proper results are obtained in the demonstration.

8. In addition to the language features which will be included for the basic demonstration, a list of desirable options will also be articulated which, in the order of their appearance, will be added to the basic demonstration system as time and money permit.

PURPOSE

The purpose of this demonstration effort, as now defined in this document, is to build a machine transportable on-line query system which resembles insofar as is practical the operation of the on-line query subset of the GIM II system. Queries made in this system should yield print-outs similar to like queries made in the GIM II system. Where differences exist, they will be due to unavoidable differences in hardware (eg. printing devices) or lack of sophistication in the demonstration system compared to the GIM II system, or both.

It is recognized that the GIM II system is a large and comprehensive data management system which has taken many man-years to develop, which cannot possibly be fully implemented in the few short months of this project. However, the purpose of this project is not to duplicate the GIM II system or any other like system. Rather, it is to show that on-line query functions are possible in a machine transportable language, and the GIM II query format is particularly useful as a model for such a demonstration. Therefore, some deviations from the GIM II language format will be expected.

TOTAL SYSTEM

Although the remainder of this document will pertain to the specifications of a query language, it is also to be understood that the software which implements this language will be a full time-sharing system with all the usually expected system support features, including:

- The support of multiple on-line terminals
- Disk memory management
- User accounting
- Error recovery

It is also understood that an installation process will be involved before it can be operated on a given computer. The installation process is well-known, being very similar to that for PLANIT, and in fact should be able to use a PLANIT installation interface if one exists.

DEMONSTRATION QUERY LANGUAGE

The general form for the demonstration query language will closely resemble the GIM II query specifications as follows:

(FOR clause (Selection clause)) Verb clause (Limiter clause)#

In the above, "FOR" is a language particle which serves as a qualifier for the statement, and the remainder of the line will be explained below. First, the various particles of which the line is composed will be presented in appropriate categories.

Particles.

Prepositional Qualifiers:	FOR, WITH, WHERE, WHEN
Adjective Qualifiers:	FIRST, LAST, GREATEST, SMALLEST, PRESENT, NULL, NEAREST
Connectives:	AND, OR, NOT
Relationals:	GT (GREATER THAN) GE (GREATER THAN OR EQUAL TO) EQ (EQUAL TO) LE (LESS THAN OR EQUAL TO) LT (LESS THAN) NE (NOT EQUAL TO)
Verbs:	LIST, LIST-VERTICAL (& LISTV), COUNT, TOTAL, AVERAGE
Adverb Qualifiers:	ASCENDING, DESCENDING
Nouns:	A name beginning with a letter and including only letters, digits and/or hyphens.

Values: Any characters enclosed in quotes.

Symbols: letters (A - Z)
digits (0 - 9)
- -- hyphen in name, minus sign
(-- open parenthesis
) -- close parenthesis
. -- decimal point
" -- quotation mark
-- line terminator

Query Statement Format. Repeating, the general form for the query statement is:

(FOR clause (Selection clause)) Verb clause (Limiter clause)#

The parentheses in the above general form are there only to indicate that part of the form which is optional (enclosed in parentheses) versus that part which is mandatory, that is, the Verb clause. The above parentheses are not typed as a part of the query statement.

The FOR clause designates the data base name upon which the Verb clause will operate, where the name is of the Noun form.

-EXAMPLE-

FOR EMPLOYEE-LIST LIST NAMES ADDRESSES#

In this case, "EMPLOYEE-LIST" is the name of the data base. If the FOR clause is omitted, the most recently named data base will be used. If none, an error will be reported.

The Selection clause selects the data base entries upon which the verb clause will operate where the clause will begin with a WITH or WHERE particle and include some relational expression.

-EXAMPLE-

```
FOR EMPLOYEE-LIST WITH ANN-SALARY GE "15000" LIST
NAME ANN-SALARY#
```

The noun, ANN-SALARY describes one of the entry catagories within the EMPLOYEE-LIST data base. WITH and WHERE are analogous particles with equivalent meanings (although their meanings could eventually become differentiated if heierarchical data list capabilities should be added sometime in the future). The relational phrase, "ANN-SALARY GE "15000" determines which entries in the data base will qualify for verb action. The entry to the left of the relational particle will be a name of an entry category within the named data base. The entry to the right of the relational particle will be similar to the data in the data base under the named category. The category name here is ANN-SALARY, and a data sample is "15000" (i.e. the digits, 15000, enclosed in quotes).

Connectives can further sharpen selectivity in the selection clause.

-EXAMPLE-

```
FOR EMPLOYEE-LIST WITH ANN-SALARY GE "15000" AND
HIRE-DATE LE "1965" COUNT NAMES#
```

Because of the "AND" connective, both ANN-SALARY and HIRE-DATE must qualify in order for the name to be counted. The connective forms can be AND, OR, AND NOT and OR NOT. Parentheses may enclose relational phrases to clarify the intent.

In the following examples, a series of three dots (...) will designate some relational phrase such as 'ANN-SALARY EQ "15000"' or 'HIRE-DATE LE "1965".'

-EXAMPLES-

FOR clause WITH ... Verb clause#

FOR clause WITH ... AND ... Verb clause#

FOR clause WITH ... AND WITH ... Verb clause#

FOR clause WITH ... AND ... OR ... Verb clause#

FOR clause WITH (... OR ...) AND (... OR ...) Verb clause#

FOR clause WITH ... AND NOT ... Verb clause#

FOR clause WITH (... AND (... OR ...)) Verb clause#

Note in the above that the forms, "WITH ... AND ... " and "WITH ... AND WITH ... " are functionally identical and will remain so unless heierarchical data lists might be added in the future. The allowed number of relational phrases and/or parenthetical enclosures in one query statement is limited only by the total amount of space set aside to process the statement.

One additional relational phrase contains an implied "OR" of the form, "Noun Relational Data, Data, etc."

-EXAMPLE-

FOR EMPLOYEE-LIST WITH HIRE-DATE EQ "1971" "1972" "1973" LIST NAME#

The above example is functionally identical to the following:

-EXAMPLE-

FOR EMPLOYEE-LIST WITH HIRE-DATE EQ "1971" OR HIRE-DATE EQ "1972" OR HIRE-DATE EQ "1973" LIST NAME#

Recall, also, that the WHERE particle can be interchanged with the WITH particle.

The Verb clause will consist of the verb followed by one or more category names that will be acted on by the verb subject to the qualifiers.

-EXAMPLE-

```
FOR EMPLOYEE-LIST WITH DIVISION EQ "MARKETING"
LIST NAME ADDRESS TELEPHONE-NUMBER#
```

The named category data from each qualifying data base entry will be listed.

Verb operations will be discussed later.

The operation of the Limiter clause is functionally identical to the Selection clause and should be considered to be an extension of the Selection clause, providing optional placement within the statement relative to the Verb clause. Preceeding the Verb clause, the Selection clause begins with a "WITH" or "WHERE" particle. Following, it begins with the "WHEN" particle, and is called the Limiter clause. Either or both kinds of clauses may appear in a query statement. If both appear, then an "AND" condition will be implied between the two.

Verbs. The uncomplicated verbs are: COUNT, TOTAL and AVERAGE. COUNT yields a numerical report of the number of data base entries which qualify according to the selection criteria in the Selection clause. TOTAL yields the numeric sum of qualifying entries, requiring that the entries be numeric. AVERAGE yields an arithmetic average of qualifying entries, also requiring the entries

to be numeric.

The various LIST verbs imply some sort of formatting conventions for the report. Two varieties of reports are envisioned, columnar and vertical. The columnar will be of the form:

-EXAMPLE-

NAME	ANN-SALARY
JOHN SMITH	16224
ANN JONES	18913
JOE DOAKES	17323

etc.

Column width will be the larger of the number of data columns in the entry field or the number of characters in the entry name. Numeric data will be adjusted to the right margin of the column while non-numeric data will be adjusted to the left. Two spaces will separate the columns.

The vertical format will display the data vertically down the left margin of the display surface.

-EXAMPLE-

```
      NAME:  JOHN SMITH
ANN-SALARY:  16224

      NAME:  ANN JONES
ANN-SALARY:  18913

      NAME:  JOE DOAKES
ANN-SALARY:  17323

etc.
```

Indentation will be determined by the longest entry name.

The LIST verb will be used to designate the columnar display format while the LIST-VERTICAL (LISTV) verb will designate the vertical format. However, if the LIST clause encompasses a collection of entry names and/or data fields of a magnitude such that the combined column requirements of the display exceeds the number of available columns on the display device, the display will default to the vertical format.

LIST-ASCENDING and LIST-DESCENDING provide an ordered listing of output data, sorted on the basis of the first entry category name following the verb. Columnar format is assumed unless the line length is exceeded in which case the format will default to vertical.

Dictionary Name String Substitution. Any subset of any query statement may be designated by a dictionary name which has associated with it a character string to be substituted for its name in the query statement so long as no partical name gets divided. More will be said about this feature later.

-EXAMPLE-

FOR EMPLOYEE-LIST RETIREES LIST NAME#

where RETIREES = AGE GE "62"

REPORT-NO1#

where REPORT-NO1 = a complete query statement.

Normalizing. Two sets of normalizing rules will be applied in the process of executing a query statement. One set of rules will apply to numeric data and the other set to non-numeric.

For numeric data fields (whether in the query statement or data base), all blanks will be squeezed out, a decimal point will be supplied at the right if one is missing, and any leading and/or trailing zeros will then be eliminated. When two numeric values are being compared, they will be aligned by their decimal points and the columns of the smaller filled out with zeros until the number of columns of the two numbers agree. A minus sign will retain its usual meaning, appearing only to the left of the number. Any other characters in the field will cause an error to be reported for that entry in the data base. If the field contains all blanks, the entry will be considered to have the numeric value of zero.

For non-numeric data fields, leading and/or trailing blanks will be eliminated and any embedded multiple contiguous blanks will be squeezed to one blank.

Query statement selection criteria assume the comparison of normalized data. Normalization is only an internal processing operation. No change is made to data within the data base.

Adjective Qualifiers. The Adjective Qualifiers are used in a variation of the Selection clause format. Instead of the normal relational pattern (i.e. ANN-SALARY GE "15000"), the relational particle and the data item is omitted and the adjective particle is inserted ahead of the entry name noun.

-EXAMPLE-

FOR EMPLOYEE-LIST WITH GREATEST ANN-SALARY LIST NAME#

This same statement pattern is also used for the adjective particles, FIRST, LAST, SMALLEST, PRESENT and NULL. The remaining adjective particle, NEAREST, stands in place of the relational particle in the normal statement format.

-EXAMPLE-

FOR EMPLOYEE-LIST WITH ANN-SALARY NEAREST "15000"
LIST NAME#

QUERY LANGUAGE SYSTEM SUPPORT FEATURES

Although the present demonstration is limited to the on-line query component of a data management system, it will be necessary to include some limited capability for creating and cataloguing a data base and for building the necessary dictionaries that will be needed in composing the queries. Certain points need to be clearly articulated concerning the difference in approach to the on-line query component from that to the system support features:

1. It is expected that demonstrations before visitors will be restricted to the formulation of on-line queries and viewing the results. The system support features will be necessary for the setting up of the demonstration but will not be exercised during the demonstration.

2. The GIM II system is being used as a model for the on-line query component with attention given to human factors while the system support features may be relatively rigid in format and require more operator orientation.

3. The on-line query component consists of a language and system operations which are likely to survive the present demonstration project, the conventions of which might, with some refinement, be accepted as a kind of standard. However, for the system support language conventions, no such standardization is suggested since future developments of this portion of the system may want to significantly alter those conventions.

Therefore, the purpose of the system support language features is solely to make a data base available to the on-line query portion of the system. Dictionaries will

be constructed with the objective of making the queries respond appropriately, whereas a totally different approach to dictionary building would probably be taken if this part were being included in the "finished" system.

Data Bases. A data base will simply consist of a collection of card images located in sequential order on the memory unit (probably a magnetic disk) which will be searched in response to on-line queries. The number of columns in each card image data base record (entry) will be determined by a system generation parameter. The data base will be created from a card-reader-like device which shall accept standard computer cards in the following arrangement:

Card #1: DATA-BASE name#
Card #2: data
Card #3: data
.
.
Card #m: data#
Card #m+1: data
.
.
Card #m+n: data#
(etc.)
Final card: $$$$

In a narrative of the above example, the first card of a new data base contains the particle, DATA-BASE, followed by a user-chosen data base name. That name will be catalogued for use after the "FOR" particle.

The next card begins the first entry in the data base. Multiple cards may be used to supply a single entry. A "#" character on a card will signal the end of that particular entry and the remainder of that card will be blank. The result will be to create an internal entry as long as the combined number of columns on the card. If a query statement should refer to entries which have been defined to reside in columns beyond the end of the "#" character, blank entries will be assumed.

Each succeeding data base entry follows in clusters of one or more cards.

The data base creation is concluded when the "$$$$" card is received.

Error checking during data base creation will be restricted to verification of a proper data base name and the presence of an all-blank card beyond any "#" character.

Name Attribute Dictionary of Data Base Entries. The user may define noun-type names which specify the data types for any column field within a data base. This dictionary is associated only with the data base for which it has been built. The format for defining such a name is:

> FOR data-base-name DECLARE attribute-name = c1, c2, type#
> (where c1 = column 1, c2 = column 2 and type = C for
> character strings or type = N for numbers)

The attribute-name is a standard noun form to be used in query statements. The section on normalization describes

the manner in which the named attribute fields will be processed during response to query statements. Fields may be defined which overlap with other fields or coincide with them to permit added flexibility in response selectivity and print formats. However, numerical fields should only be defined to span numerical data. The column numbers in the declaration statement are those from the cards, accumulated in the case of multiple cards per entry.

-EXAMPLE-

FOR EMPLOYEE-LIST DECLARE NAME = 1, 25, C #
FOR EMPLOYEE-LIST DECLARE ANN-SALARY = 52, 60, N #

If the FOR clause is missing (i.e. the statement begins with the DECLARE particle), the last mentioned data base name is assumed.

These attribute name declaration statements, as above, may be supplied in the card stream or at the terminal (or both). If in the card stream, insert a card just prior to the end "$$$$" card with only a # in column one. Then any number of declaration cards may be supplied between the "#" card and the "$$$$" card. A command at the terminal:

READ-CARDS #

will activate the card reader and the terminal will cease to be active until the "$$$$" end card has been read. Thus, the card reader will simply be regarded as an alternate

terminal, accepting the same command formats.

Multiple DECLARE statements may be given using semi-colons to separate each attribute name as follows:

-EXAMPLE-

FOR EMPLOYEE-LIST DECLARE NAME = 1, 25, C ; ANN-SALARY = 52, 60, N #

The allowable length of such an input will be determined by a system generation parameter which governs the amount of space set aside to process a terminal response.

Dictionary of Substitution String Names. When nouns are found in a query statement which also occur in this dictionary, the character string associated with the dictionary name will be substituted for the noun in the query statement before further processing takes place. Any string may be thus defined and named in this dictionary, including strings that contain the names of other strings. Two such dictionaries will be kept, one unique to the data base being queried and the other being general for all data base queries. The format for defining the string follows:

FOR data-base-name DEFINE string-name = anything #

If the FOR clause is omitted, the string-name will be entered in the global dictionary. Otherwise, it will be entered in the dictionary belonging to the named data base.

-EXAMPLE-

FOR EMPLOYEE-LIST DEFINE RETIREES = AGE GE "62" #

The string being defined may be completely arbitrary except that is may not contain a "#" character other than the one that designates the end of that line. Note that the "#" character is not part of the string being defined in the above example. Multi-line strings may be defined.

PRINCIPLES OF OPERATION

In general, this demonstration system will accept the aforementioned statement forms at the terminal or, following a READ-CARDS command, from the card reader. If from the card reader, then activity at the terminal will cease until the "$$$$" card is received at the card reader. However, output will continue to come to the terminal printer.

The system will contain four particle dictionaries, with some of the dictionaries partitioned as needed:

1. Dictionary of documented system particles, segmented according to the kind of particle (e.g. nouns, verbs, relationals, etc.)
2. Dictionary of data base names
3. Dictionary of attribute names (one for each data base)
4. Dictionary of substitution string names, segmented into global and specific.

Where a particular kind of particle is expected in the syntax of the statement, the corresponding dictionary will be searched first to determine the meaning of the particle

name. If that fails, the specific string substitution name dictionary will be searched next. Should that fail, the attribute name dictionary will then be searched but only if an attribute name fits the syntax at that point. Still failing, the global string substitution name dictionary would then be searched.

Terminal Report. In addition to the report implied in the query statement, the query statement itself will be reproduced at the head of the report. In addition, if any string substitution occurred in the query statement, the fully expanded version of the query will also be reproduced.

At the end of the report will be a status line such as:

COUNT: 15 OF 40

where the first number refers to the number of data base entries which were selected for verb action, and the second refers to the total number of entries searched.

Halting The System. The operation of the system will be brought to a halt with the command:

HALT #

ACCEPTABLE SYSTEM REPERTOIRE

This document was produced with the realization that more is probably being proposed than there will be time to complete. The purpose for doing this is to avoid any waste of remaining time while agreement is reached on how to use the closing days of the contract period, assuming the the minimum basic system has been completed.

Rather than redefine what is being proposed as the minimum basic system, the inverse approach will be used, which amounts to a prioritized list of what will not be included if time runs out. In this list, the items listed first will be the first to be eliminated in a time crunch. Even if all listed items are eliminated, the remaining system will still support a very impressive on-line query demonstration.

1. LIST-ASCENDING and LIST-DESCENDING
2. Variable record lengths in data base entries, requiring uniform fixed length records.
3. Columnar format in printing.
4. Global substitution string name dictionary.
5. Specific substitution string name dictionary.
6. Not accept parentheses in query statements.
7. Accept a maximum of one connective in query statements.

Despite the number of items listed, the present goal only regards the first two as tentative. The others will be considered only in the event of unforeseen problems.

PORTABLE QUERY SYSTEM

USER'S MANUAL

Developed For:

The United States Army Research Institute

Alexandria, Virginia

Under Contract No.

DAHC19-76-C-0022

Charles H. Frye

The Northwest Regional Educational Laboratory

Portland, Oregon

December 1977

APPENDIX B

PORTABLE QUERY SYSTEM

USER'S MANUAL

PREFACE

An attempt was made in this document to combine the reference manual format with a narrative style. It is hoped that, with a great deal of tolerance on the part of the reader, this method will make the document readable to the new user as well as useful to the experienced one.

The section numbers used in the document imply the use headings and subheadings in a modified outline format. This organization seemed to lend itself a little better to the narrative style. If the reader is unfamiliar with this numbering scheme, a brief explanation may clarify it. In place of the common outline form:

I.
  A.
    1.
      a.
II.
  A.
    1.
      a.

the following are used:

1.0
1.1
1.11
1.111
2.0
2.1
2.11
2.111

Therefore, each "X.0" indicates an introduction to the next main topic, and within that section, each decimal place indicates the next subpoint level (level of indentation). The indentation shown in the Table of Contents should help to clarify the notation.

TABLE OF CONTENTS

		Page
1.0	INTRODUCTION	1
2.0	RULES GOVERNING CHARACTER USAGE	1
2.1	NAMES	2
2.2	NUMBERS	2
2.3	DELIMITERS	3
2.4	STATEMENT TERMINATOR	3
2.5	CONTROL STATEMENT DESIGNATOR	4
2.6	BLANKS	4
2.7	PARENTHESES	5
3.0	INPUT FORMS	6
3.1	QUERY STATEMENT FORM	6
3.11	PARTICLE NAMES FOR THE QUERY STATEMENT FORM	6
3.12	DATA PARTICLES FOR THE QUERY STATEMENT FORM	7
3.13	QUERY STATEMENT GENERAL FORMS	8
3.131	FORM NUMBER ONE EXAMPLES	9
3.132	FORM NUMBER TWO EXAMPLES	9
3.2	DATA BASE ENTRY FORM	10
3.3	CONTROL STATE ENTRY FORM	11
3.4	INPUT DEVICE FORMS, TERMINAL VS. CARDS	11
4.0	DATA BASE CREATION	12
4.1	DECLARING A NEW DATA BASE	12
4.2	ADDING TO A NAMED DATA BASE	12
4.3	ADDING TO THE CURRENT DATA BASE	13
4.4	DATA BASE ORGANIZATION	14
5.0	THE QUERY STATEMENT	17
5.1	DATA BASE IDENTIFICATION	17
5.2	THE SELECTION CLAUSE	18

		Page
5.21	COMPARISON OF SELECTION CLAUSES IN THE TWO FORMS	18
5.22	NATURE AND PURPOSE OF THE SELECTION CLAUSE	19
5.23	COMPARISON OF DATA	20
5.24	SIMPLE SELECTION CLAUSES	22
5.25	COMPOUND SELECTION CLAUSES	22
5.26	THE USE OF ADJECTIVES IN THE SELECTION CLAUSE	25
5.27	NEGATED COMPARISONS IN THE SELECTION CLAUSE	26
5.28	ABBREVIATED "OR" COMPARISONS	26
5.29	SELECTION CLAUSE VARIATIONS	27
5.3	THE REPORTING VERB	28
5.31	LIST	29
5.32	LIST-VERTICAL AND LISTV	30
5.33	COUNT	30
5.34	TOTAL	31
5.35	AVERAGE	31
5.4	THE REPORT CLAUSE	32
5.5	TERMINATOR CHARACTER	32
6.0	REPORTING FORMATS	33
6.1	NUMERIC VALUE REPORTS	33
6.2	DISPLAYING DATA	33
6.21	COLUMNAR DATA DISPLAY	34
6.22	VERTICAL DATA DISPLAY	35
7.0	MACROS	36
7.1	DEFINING AND USING MACROS	36
8.0	COMMAND VERBS	39
8.1	PRESTORING A CARD DECK	39
8.2	USING A PRESTORED CARD DECK	40
8.3	CREATING OR EXTENDING A DATA BASE	41
8.4	DEFINING A MACRO NAME	42
8.5	DECLARING ATTRIBUTE NAMES	42
9.0	CONTROL STATEMENTS	43
9.1	USER SIGNOFF, $OUT	43
9.2	OPERATOR CONTROLS	43

		Page
10.0	OPERATING PROCEDURES	45
10.1	SIGNING ONTO THE SYSTEM	45
10.2	TERMINAL PROMPTS	45
10.3	LINE ECHOING	46
10.4	ERROR MESSAGES	46
10.5	CORRECTING INPUT LINE ERRORS	47
10.6	MODE SWITCHING	47
10.7	SIGNING OFF FROM THE SYSTEM	48

APPENDIX A: INSTALLATION INFORMATION

APPENDIX B: INITIALIZATION DECK

PORTABLE QUERY SYSTEM USER'S MANUAL

1.0 INTRODUCTION

The Portable Query System (PQS) was developed under contract (No. DAHC19-76-C-0022) for the United States Army Research Institute. The purpose of the system is to provide on-line retrieval from a uniformly-defined data base in a way that can be replicated on a variety of computers. Data bases consist of multiple chained records where each record contains a string of an arbitrary number of keyboard characters.

The Query system can be moved to another computer through a well-defined installation process which is virtually identical to the installation of the PLANIT system.

The data bases for the Query system require no installation when moved from one computer to another. At most, characters which differ on the new machine may need translating.

This user's manual is valid for all installations of PQS. Variations which may occur will be noted.

2.0 RULES GOVERNING CHARACTER USAGE

The syntax of query statements dictate fairly stringent rules in the use of the characters which make up the statement. Where freedom is needed to show the arbitrary character strings which may occur in the data base, these strings will be enclosed between delimiting characters, either primes (') or quotation marks ("). Thus, when considering query statement particles, the delimited string will be regarded as a single particle.

2.1 NAMES

Names include those which are primitive in the system as well as user-defined. Names begin with a letter of the alphabet and contain letters, digits and/or hyphens (-). Letters may be upper or lower case (no distinction will be made). Any number of characters may appear in a user-defined name and all that are used will be remembered in the name dictionary.

Names are usually delimited by blanks or begin or end the line. However, other characters which may not appear in a name, if they stand immediately adjacent to the name will also serve as a delimiter of the name.

Valid name examples:

FOR	for (same as FOR)
WITH	With (same as WITH)
LIST	LIST-VERTICAL
COL1	DATA-BASE-NO-3
F4J	H----

Invalid name examples with reasons for invalidity:

DATA BASE	(two names)
FILE-NO.-3	(invalid character. FILE-NO will be taken as the name)
2-4-D	(must start with a letter)

2.2 NUMBERS

Numbers are of two kinds, those which appear as parameters (e.g. column numbers) and those which appear as data (to be compared to the data base entries).

The first kind contains only digits and is delimited by blanks, parentheses or commas. An example of a statement which contains two numbers is:

```
DECLARE ITEM NUM(12,24) #
```

The latter kind of number may also contain a sign (+ or -) and a decimal point, however neither is required. This kind of number will normally be enclosed in delimiters (' or ") although they are optional where the context makes the identity of the number clear. Examples of this kind of number are:

```
LIST SALARY WHEN EDUCATION GE '16' #

WITH GRADE EQ '4' '5' '6' COUNT #

'+21.54'

'-15'
```

2.3 DELIMITERS

The two delimiting characters for data are the prime (') and quotation mark ("). When used as delimiters, they must occur in pairs, i.e. the one used to open the delimited string must also be used to close it. A delimited string may contain one of the delimiting characters by using the other as delimiters. For example:

```
'Quotation mark (")'

"Prime (')"
```

Any character string (other than the delimiting characters) may occur between delimiting characters.

```
"Picture #42a"
```

No other characters (blanks included) may be used to delimit data strings (except in the case of numbers, above, where blank delimiters are optional).

2.4 STATEMENT TERMINATOR

Every Query statement shall be terminated by the Pound character (#). The # character may only be used at the end of the line (or in a delimited data string). If a Query statement line does not end with a # character, it will be assumed that the

statement continues on the next line.

Data base lines also use the # terminator. If a data base line does not end with the # character, it will be assumed that the next line is a continuation of (extension to) the current one.

The # character may appear alone on a line as an indication that data base inputs are complete and the next line is to be taken as a Query statement.

The # character is not used to terminate control statements as described below.

2.5 CONTROL STATEMENT DESIGNATOR

A character will be chosen at installation time to be the Control Statement Designator. For the purposes of this document, that character will be assumed to be a dollar sign ($). To be a Control Statement Designator, the chosen character must occur first on the line, i.e. the first typed character of the line. Control statements have no standard termination character. The # character does not terminate control statements. For example:

```
$QUIT ALL
```

2.6 BLANKS

The exact count of blanks on a line is significant only in the case of input strings to a data base. Otherwise, all groups of multiple blanks are treated as single blanks. Blanks are useful to clarify the syntax of a query statement especially as pertains to names and numbers. However, the usual case is that blanks (and particularly the multiple occurance thereof) are optional.

The following three statements will be functionally equivalent:

```
LIST ENTRY WHEN NAME EQ 'JOHN SMITH' #
LIST ENTRY WHEN NAME EQ'JOHN SMITH'#
LIST  ENTRY WHEN   NAME   EQ  'JOHN   SMITH  '   #
```

The following shows the omission of a blank which makes the statement invalid:

```
LIST ENTRY WHEN NAMEEQ 'JOHN SMITH' #
```

In this case, NAMEEQ is taken as a single particle, making the syntax of the statement invalid.

2.7 PARENTHESES

The rules pertaining to parentheses only affect those in the query statement which do not occur between delimiting characters or in data base input lines.

In general, parentheses are used in the query statement to clarify conditional clause groupings that might otherwise be ambiguous. Any conditional clause or any combination of compound conditional clauses may be enclosed in parentheses. However, the clause within the parentheses must be complete.

Examples of valid uses of parentheses follow:

```
LIST ENTRY WHEN(NAME EQ 'JOHN SMITH') #

WITH SEX EQ 'F' AND(EYES EQ 'BLUE' OR
HAIR EQ 'BROWN') COUNT #
```

The limit to the number of parentheses which may be used in a statement is sufficiently generous that the user will seldom, if ever, encounter it. The exact number will be a function of installation but will usually be in the order of fifty or more.

The reader will find additional explanation and examples regarding the use of parentheses in sections 5.25 and 7.1.

3.0 INPUT FORMS

There are three input formats which the Query system recognizes: (1) Query statement form, (2) Data base entry form, and (3) Control statement form.

3.1 QUERY STATEMENT FORM

The Query statement form refers to those query statements which are composed for the purpose of selecting certain data from the data base to be displayed in the report. In addition, this form also includes certain directives for creating and naming a data base and for accepting data from a card reader device.

3.11 PARTICLE NAMES FOR THE QUERY STATEMENT FORM

Prepositional Qualifier for Data Base Name:	FOR
Prepositional Qualifier for Selection Clause:	WITH WHERE WHEN
Connectives:	AND OR
Negation:	NOT
Relationals:	GT (Greater than) GE (Greater or equal to) EQ (Equal to) LE (Less or equal to) LT (Less than) NE (Not equal to)
Reporting verbs:	LIST LIST-VERTICAL (& LISTV) COUNT TOTAL AVERAGE

Adjectives:	**NULL** **PRESENT** **SMALLEST** **GREATEST** **FIRST** **LAST** **NEAREST**
Command verbs:	**PRESTORE** **READ-CARDS** **DATA-BASE** **DEFINE** **DECLARE**
DECLARE Qualifiers:	**STR** (String) **NUM** (Number)

Three additional particle names in the Query statement are user-defined. These include data base names, data base attribute names, and macro names. For the purposes of this document, three mnemonics will be assigned which will always be used to signify the three names as follows:

Data Base Name:	Dlname
Data Base Attribute Name:	Attr-Name
Macro Name:	Macro-Name

The reader should understand that wherever the above three mnemonics appear in this document, some user-chosen name will stand in its place.

3.12 DATA PARTICLES FOR THE QUERY STATEMENT FORM

Data particles in the query statement can be any arbitrary series of characters enclosed between a pair of delimiting characters as described in 2.3. The data will always qualify as being String data since this refers to a completely arbitrary string of characters. Additionally, the data may qualify as Numerical data if the data consists only of a number (and optional leading sign and decimal point).

Some query statement forms can accept the more general String data particle form while others demand the Numerical data particle form. In particular, the relationals, GT, GE, LE, and LT can only be used with Numerical data particles (or with numerical Attr-Names). Also, when a data particle is logically compared to a numerical Attr-Name, the data particle must be Numerical. This will be discussed in much more detail later.

For the present, two mnemonics will be used to differentiate the two kinds of data particles. Keep in mind that a Numerical data particle also qualifies as a String data particle but the converse is generally not true.

String Data Particle: 'String Data'

Numerical Data Particle: 'Numerical Data'

3.13 QUERY STATEMENT GENERAL FORMS

There are two general forms for the query statement. One has the data selection clause before the reporting verb, the other has it after. Brackets denote optional entries in the general form.

Form number one:

[FOR Dlname][Selection Clause] Verb [Attr-Name(s)] #

Form number two:

Verb [Dlname][Attr-Name(s)][Selection Clause] #

In form number one, if the FOR particle is used, a Dlname must follow it. In either form, if the Dlname is omitted, the most recently referenced Dlname will be assumed. If none is valid, the statement will be rejected.

The Selection Clause refers to those data comparisons which cause the selection of entries to be reported. The composition of the Selection Clause and the reporting possibilities will be discussed later.

3.131 FORM NUMBER ONE EXAMPLES

```
FOR EMPLOYEE-LIST LIST NAMES ADDRESSES #

FOR EMPLOYEE-LIST WITH ANN-SALARY GE '15000' AND
  HIRE-DATE LE '1965' COUNT #

WITH HIRE-DATE EQ '1971' OR HIRE-DATE EQ '1972' OR
  HIRE-DATE EQ '1973' LIST NAMES #

WITH HIRE-DATE EQ '1971'  '1972'  '1973' LIST NAMES #

FOR ROSTER WITH (SEX EQ 'F' OR RACE NE 'WHITE') AND
  ANN-SALARY GE '15000' LIST NAME AGE JOB-CLASS #
```

3.132 FORM NUMBER TWO EXAMPLES

```
COUNT EMPLOYEE-LIST WHEN ANN-SALARY GE '15000'
  AND HIRE-DATE LE '1965' #

AVERAGE STUDENT-BODY AGE WHEN MARITAL-STATUS EQ 'M' #

LIST EXPENDITURES CREDITOR AMOUNT WHEN DUE-MONTH EQ
  'JULY' AND DUE-DATE LE '20' #

TOTAL DRIVING-RECORDS ACCIDENTS WHEN PRESENT DISABILITY #

LIST NAME WHEN NEAREST WEIGHT '170' #
```

3.2 DATA BASE ENTRY FORM

Data base entries are composed by concatenating lines of input data until the terminal # character is encountered. The first character of the data base entry goes into column one and the following ones go into succeedingly higher column numbers until the terminating # is encountered.

Data entries may originate either at the terminal or in the form of punched cards. If from the terminal, then the terminal line length determines the number of characters (including trailing blanks) which shall be concatenated onto the data base entry for each new line. For example, if the terminal line length is 72 and the terminating # character occurs in column 21 of the fifth line, then the number of columns of data in that entry will be (4 X 72) + 20 = 308. The first five characters typed on the third line would be located in columns 145 through 149 of the entry. Thus, each entry of the data base can be visualized as one long punched computer card whose length can accomodate all the characters in the entry.

Data entries which originate from punched cards are composed by concatenating the contents of the cards in a similar manner. Each punched card will be assumed to contain 80 characters unless given a smaller number in the READ-CARDS verb, described below.

The only characters which are not appropriate in a data base entry are two dollar signs ($$) in columns one and two (which are used to signify an end-of-file to the card reader).

Total possible length of any single data entry is determined at the time the system is installed by the values chosen for certain of the parameters. It will usually be sufficiently generous, in the order of 1,500.

Lengths of data base entries may be allowed to vary. If uniformity is required for any subsequent query search, a trailing blank fill in the entry will be assumed.

Exit from the data entry mode is accomplished by typing (or punching) a # character in column one, making it the only character in the entry.

3.3 CONTROL STATE ENTRY FORM

Control statements begin with a $ character in column one and are typed in the format prescribed for each statement. Only the number of blanks are optional, requiring at least one for each blank shown in the statement form, below.

Control statements may be entered only at the terminal (not from the card reader) and only while the terminal is accepting query statements (not while it is in the data entry mode). If a control statement is mistakenly entered while the terminal is in the data entry mode, that input line will be concatenated to the data entry.

Control statements are generally only used by the system operator, except for $OUT, the user sign off.

Following the execution of the control statement, the terminal is left in the query statement mode, except in the case of a system shutdown ($QUIT ALL) in which case all files and devices would be detached and program execution would stop.

3.4 INPUT DEVICE FORMS, TERMINAL VS. CARDS

In general, input forms for the terminal and for punched cards are identical. Exceptions are the $$ (end-of-file) characters in columns one and two of punched cards which have no meaning on the terminal, and Control Statements which are not allowed from punched cards. Otherwise, the conventions apply to either.

4.0 DATA BASE CREATION

Data bases may either be created or extended by use of the DATA-BASE verb. Also, following the input of the DATA-BASE verb, the input mode is switched to data entry.

4.1 DECLARING A NEW DATA BASE

A new data base is declared by entering the line (in the query entry mode):

DATA-BASE Dlname #

The chosen Dlname must be different from any name presently catalogued on the system. Following this entry, the next input will be in the data entry mode, creating the data entries for the named data base.

When the data base is complete, or the completion is to be delayed until later, enter a # character as the first and only character of the entry. Entry mode will switch back to query form.

A data base may be declared using the DATA-BASE command without any entries by exiting with the # character immediately after naming the data base. This technique can be useful for naming a data base on the terminal which will be created from punched cards.

4.2 ADDING TO A NAMED DATA BASE

By using the same command as above, i.e.:

DATA-BASE Dlname #

where the Dlname already exists, entries may be added to the named data base. If this situation occurs at the terminal, a confirmation is required, i.e. the system will ask:

ADD TO THIS DATA BASE? (Y/N)

If the wrong name has been chosen, the user will have opportunity to correct it. No confirmation opportunity is given if the DATA-BASE command comes from a punched card.

Data bases can be arbitrarily long, having as many entries as the storage medium will allow.

4.3 ADDING TO THE CURRENT DATA BASE

By omitting the Dlname from the command form, the user can add entries to the currently active data base. The command form would be:

```
DATA-BASE #
```

If a data base is currently active through previous use, input mode would then switch to data entry form and the next input would create an additional entry for the active data base. This command form, in combination with that shown in 4.1 above, allows the user to name a data base at the terminal and fill it with data from punched cards. The sequence of commands would appear as follows:

```
DATA-BASE Dlname #
#
READ-CARDS Filename #
```

The new data base would thus be named and immediately terminated. Input would be switched to the card entry format because of the READ-CARDS command verb. Then, in order that the entries on the cards become associated with the newly named data base, no data base is named on the cards. Rather, the statements on the cards rely on the default condition, i.e. that the data base to be used will be the one most recently named. Thus, if the cards hold entries for the data base, the first card would contain the command verb:

```
DATA-BASE #
```

Notice that it does not name the data base, using instead the most recently named one. The following entries would then contain data.

4.4 DATA BASE ORGANIZATION

Assuming that the new data base has been named with the DATA-BASE command verb and the user is now in the data entry mode, ready to put the data into the data base, this section describes how to organize the entries so that the data base can be easily searched at a later time for on-line retrieval.

Recall from section 3.2 that the rules for building data entries from the terminal are exactly the same as from punched cards with the only difference being the length of the line (number of columns) that each device sends.

Recall, too, from section 3.2 that the entry should be visualized as a very long punched card that may have upwards of 1,500 columns. Many of these columns will normally be blank. However, each of these 1,500-column entries become one entry in the data base, where the data base name can refer to a collection of hundreds or thousands of these entries.

It will often require several lines (or punched cards) to make up a single entry. The end of the entry is indicated by the terminal # character.

The entry itself will be made up of "fields" of data where each field is a series of columns that are dedicated to a particular kind of data. Using a DECLARE verb, the field gets named, and that becomes the Attribute Name (Attr-Name) for that data. (See also section 8.5 regarding the DECLARE verb).

For example, suppose it was decided to make columns one through 20 to be the "first name" field, and columns 21 through 40 the "last name" field. Presumeably, then, each entry in the data base would contain data on a particular individual. Each entry would correspond to a different person. Then, for each new entry, the first name would go into columns one through 20 and the last name in columns 21 through 40. Unused columns within these fields would be left blank. However; the field must be declared sufficiently long to accomodate any name that might go into the data base because it would create problems to have any name extend beyond its field. We might choose to declare three attribute names for these two fields:

```
DECLARE FIRST-NAME STR(1,20) #

DECLARE LAST-NAME STR(21,40) #

DECLARE FULL-NAME STR(1,40)  #
```

Here, we have essentially identified three fields, 1-20, 21-40, and 1-40, with an attribute name for each field. Having decided on this organization for the data base, it now becomes important to be sure that the appropriate data are entered in the appropriate columns. If there are multiple lines (or cards) per entry, then columns one through 40 are always (and only) on the first of those lines (or cards).

To complete the example, suppose that columns 81 through 83 are chosen for the individual's age. The DECLARE verb might read:

```
DECLARE AGE NUM(81,83) #
```

where AGE would be the Attr-Name for that field. Now, suppose that three entries are to be added to the data base where each entry contains the first and last names and age. Notice that the skipped columns 41-80 are of no concern. If the card length is 80 columns, then the data entry inputs might look like the following:

```
JOHN                    JONES
 39 #
ANN                     SMITH
 27 #
GEORGE                  WASHINGTON
105 #
```

Now, let's repeat that data base in full, showing the entire deck as it might be submitted to the Query system:

```
DATA-BASE NAME-AGE-LIST #
JOHN                    JONES
 39 #
ANN                     SMITH
 27 #
GEORGE                  WASHINGTON
105 #
#
DECLARE FIRST-NAME STR(1,20) #
DECLARE LAST-NAME STR(21,40) #
DECLARE FULL-NAME STR(1,40)  #
DECLARE AGE NUM(81,83) #
$$
```

Note that the final line ($$) is necessary only if the input is coming from punched cards; if from the terminal, that line can be omitted.

In summary, the following considerations go into creating a data base:

- The data base must be named in a DATA-BASE command verb
- Data fields (column sequences) must be chosen for each attribute (type of datum) in the data base
- Data entries must be constructed in such a way that appropriate data occur in the proper columns to match the chosen data fields
- A single # line must be included to exit from the data entry mode
- DECLARE statements must name each of the chosen data fields and indicate whether they are arbitrary strings of data (STR) or numerical data (NUM)
- If the data entry is from punched cards, a final $$ card must be added to terminate the deck

Once entered, the data base may be used for on-line retrieval, or it may be enlarged by the addition of new data entries and/or new DECLARE statements. The number of attribute names which may be declared is limited only by the amount of working space in the Query system.

5.0 THE QUERY STATEMENT

Section 3.1 has already discussed the two basic general forms for the query statements. Both forms serve the same purpose. They just provide optional formats for the query.

The query statement, regardless of the form chosen, contains five essential elements:

- Identification of the data base to be used in the data search
- Selection clause to determine which of the data base entries are pertinent to the current query report
- Reporting verb to designate the kind of information desired and how it should be displayed
- Report clause, made up of a list of those Attr-Names which designate the data from the selected entries that are to be displayed
- Terminator character (#) which marks the end of a query statement, possibly a multi-line one.

Of the above five elements, only the reporting verb and the termination character are mandatory in the query statement. Default interpretations exist for the other three in most cases.

Note that the above five elements are present in both of the general forms as shown in section 3.13. Only the order is different.

Consideration will now be given to each of the above five elements.

5.1 DATA BASE IDENTIFICATION

The data base to be used in the search to satisfy a query statement is identified by its name, i.e. the Dlname. The Dlname is made up by whoever creates the

data base as described in section 4. The rules for choosing a name are described in section 2.1.

The Dlname refers to dozens, hundreds or thousands of entries, depending on the size of the data base, each of which may be several hundred characters in length. It is this data base which will be searched, entry-by-entry, to determine which of the entries qualify for the query statement report.

The Dlname is the second name to appear on the query statement line. For form number one (see section 3.13), the Dlname follows the word, FOR. For form number two, it follows the reporting verb.

If the Dlname is omitted, the most recently named data base will be used. If none is currently active, an error message will so inform the user. Note that the omission of the Dlname in form number one implies that the word, FOR, must also be omitted. In form number two, the Dlname may be omitted without changing the remainder of the statement.

5.2 THE SELECTION CLAUSE

The selection clause is the source for most of the variety in query statements. It is this clause that determines which of the data base entries will qualify for the report. The selection clause portion of the query statement resembles "IF" statement constructions in several of the computer programming languages.

5.21 COMPARISON OF SELECTION CLAUSES IN THE TWO FORMS

Selection clause construction rules are nearly identical for the two query statement forms (in section 3.13). The only difference is in the use of the primitive words, WITH, WHERE and WHEN. Whereas the words, WITH and WHERE are used (interchangeably) in form one, WHEN is used in form two. They can occur in the same positions in the selection clause, and the choice of any one of the three makes no difference in the query statement.

Therefore, in the following illustrations of the selection clause, WITH will normally be the word used, but the reader should understand that either WITH or WHERE can be used in form one, and WHEN can be used in form two with identical results.

5.22 NATURE AND PURPOSE OF THE SELECTION CLAUSE

The selection clause is one of the most critical elements of the entire query system. If the user chose to view all of the data in the data base, no query system would be necessary; he could view the listing before it was entered into the system. The purpose of the query system is to allow the user to systematically select only those data which are relevant for the occasion without being innundated with it all.

The selection clause is used to make that determination of whether a particular entry is relevant for the report being requested by the user. If judged relevant, the reporting verb clause will determine which data fields of that entry to display. If not, the entry will be passed by.

The selection clause contains a logical comparison of data fields. The comparison is judged either "true" or "false." If judged "true," the entry qualifies, otherwise it doesn't.

The "true/false" comparison is often a combination of several "true/false" comparisons where there are different alternatives given under which the entry might qualify. Therefore, using AND, OR, NOT and parentheses to formulate the comparisons of the data, it will be shown that the possibilities are virtually limitless for selecting only those data that are pertinent to any imaginable reporting need while also insuring that no data are overlooked.

Therefore, the selection clause is a comparison of data that begins with the word, WITH, or the word, WHEN, in forms one or two of the query statement, respectively. The following are simple selection clauses:

```
WITH BREED EQ 'TERRIOR'
WITH VALUE GT '4.5'
WHEN JOB-CLASS EQ 'NON-EXEMPT'
```

5.23 COMPARISON OF DATA

The important elements of the comparison of data are the attribute names (Attr-Name), the data strings and the relationals.

The Attr-Name tells where to find the data for the comparison within the data base entry, and what kind of data to expect (arbitrary strings of characters or numerical data). The Attr-Name is chosen at the time the data base is created via the DECLARE statement, and data are entered into the data base in the data fields specified by the Attr-Name declaration. (See section 4.4 for the data base organization).

The data strings, enclosed within delimiters, designate the characters to be used in the comparison with those in the Attr-Name data fields. (See sections 2.3 and 3.12 regarding the use of delimiters).

The relationals determine how the data are to be compared, and what constitutes a "true" or "false" result. The relationals are listed in section 3.11: GT, GE, EQ, LE, LT and NE.

If the comparison is between data strings which are not numerical, only the EQ and NE apply since the strings either match or they don't. However, if the strings are numerical, there can also be a greater than or less than relationship, and any of the six relationals are valid.

The comparison format is simply any two data string designations separated by a relational. Thus, the following are all valid examples of data comparisons:

```
AGE  GE  '65'
RANK  EQ  'E4'
LAST-NAME  EQ  'WASHINGTON'
CURRENT-SALARY  LT  BEGINNING-SALARY
SEX  EQ  'MALE'
TEMP  LT '-20.5'
'4'  LT  '5'
```

Note that the last comparison is valid but trivial since it will always be true and makes no reference to the data base. This kind should be avoided. In each of the others, at least one side of the comparison is an Attr-Name, referring to the data base. In one case,

both sides of the comparison has Attr-Names, indicating a comparison of two data fields within the data base entries. It is usually apparent which Attr-Names are numeric and which are not, indicated both by the delimited data strings in the comparison and by the relational being used. Wrong comparisons will invalidate the query statement. The following are some examples of obviously wrong comparisons:

```
AGE  GE  '65 YEARS'
RANK  LE  'E4'
LAST-NAME  GT  'WASHINGTON'
INITIAL  GT  'J'
```

The relationals, GT, GE, LE and LT, must only be used to compare strings of data which name numerical quantities. No relative value is given to letters of the alphabet.

It will also be necessary to understand how blanks are treated in the comparison. The general rule is that leading and trailing blanks are ignored, and embedded sequences of blanks are treated as single blanks. (See also section 2.6). For example, using the data base in section 4.4, each of the following would be a "true" comparison:

```
FULL-NAME  EQ  'GEORGE WASHINGTON'
FULL-NAME  EQ  '  GEORGE WASHINGTON  '
FULL-NAME  EQ  '  GEORGE     WASHINGTON'
```

However, the following would not be counted "true" since no comma appears in the data base:

```
FULL-NAME  EQ  'GEORGE     ,   WASHINGTON'
```

This constitutes the basic comparison format for the selection clause. The variety of Attr-Names and data strings is virtually infinite; the relationals are limited to the six that are listed. The discussion that follows will show how these comparisons can be compounded to make the selection as specific as desired.

5.24 SIMPLE SELECTION CLAUSES

Recall that the selection clause within the query statement functions to determine whether the data in a given data base entry meet the conditions specified by the user so that the entry should be used for the report, or whether it fails, in which case the entry would be passed by for this report.

The simple form of such a selection clause consists of a preposition (WITH or WHEN) and a comparison. Several simple selection clauses have already been illustrated, including those at the end of section 5.22. In section 5.23, several comparisons were listed. By adding the preposition WITH (or WHEN) to the beginning of these comparisons, they would become valid selection clauses. Thus, a simple selection clause would be:

```
WITH AGE  GE  '65'
```

5.25 COMPOUND SELECTION CLAUSES

Compound selection clauses differ from simple selection clauses only in the number of comparisons being made, where the comparisons are joined by the connectives, AND or OR. Also, compound selection clauses can be further clarified by grouping comparisons within parentheses.

Compound selection clauses permit the selection criteria to consider more than one data field in the data base entry before that entry is used in the report. For example, suppose the report is to consist of only male drivers under 25 years. The selection clause might be:

```
WITH  SEX  EQ  'M'  AND  AGE  LT  '25'
```

Note that the data base must be using 'M' and 'F' for male and female instead of spelling them out. Because of the AND connective, both comparisons must be "true" in order for the entry to qualify for the report. A third comparison might be of interest: male, under 25 and three or more accidents. That selection clause might read:

```
WITH SEX EQ 'M' AND AGE LT '25' AND ACCIDENTS GE '3'
```

As the selection clause grows in length, it soon becomes apparent why multiple lines are needed for the full query statement. Recall, too, that the selection clause is just one of the elements in the query statement as listed in section 5.0.

The OR connective permits alternative criteria for the entry to qualify. For example, suppose the interest was in those who were under 25 or over 65, then the selection clause might read:

```
WITH AGE LT '25' OR AGE GT '65'
```

As with the AND connective, so several comparisons can be joined with the OR connective. So long as a series of comparisons are joined with AND connectives (or OR connectives), there is no confusion about the intent of the selection clause. However, an element of confusion can be introduced if both connectives appear in the same selection clause, as it is often desirable to do. Consider the selection clause:

```
WITH SEX EQ 'M' AND AGE LT '25' OR AGE GT '65'
```

It is clear that males under 25 years are in view, but what about the "over 65" bracket, are they only males or both sexes? A heierarchy rule is consistently applied in cases of this sort which always causes AND conditions to be satisfied before OR conditions where there is a choice. However, if part of the clause is enclosed in parentheses, the parenthesized portion will be satisfied first. Consider what effect this rule has.

In the above example, since the AND condition is satisfied first, the SEX comparison has no bearing on the AGE GT '65' comparison so <u>all</u> over 65 are in view. If the converse had been true, that the OR condition was satisfied first, then the two AGE comparisons would have been made before the comparison was made for the sex; then only the males would have been in view.

Suppose the intention was to consider <u>only</u> the males in both age groups. Then either of the following two selection clauses would do the job:

```
WITH SEX EQ 'M' AND AGE LT '25' OR SEX EQ 'M' AND
AGE GT '65'
```

```
WITH SEX EQ 'M' AND (AGE LT '25' OR AGE GT '65')
```

The first of the two illustrations combines the sex comparison with each of the two age groups. The second combines a single sex comparison with a parenthesized grouping which contains the two age group comparisons. Both selection clauses have exactly the same meaning. However, the second is preferred because the efficiency of the search will be directly related to the number of comparisons which must be made. There are four and three comparisons, respectively, in the two illustrations.

Notice in the second illustration, the parenthesized expression occupies the place in the selection clause that a simple comparison would normally occupy. This will always be true, even when the parenthesized expression contains other parenthesized expressions within it. There is no practical limit to the complexity of the selection clause which may be constructed by joining comparisons with AND and OR connectives, and enclosing any of these in parentheses.

Consider an example presented earlier in this section but with the addition of the "over 65" age group:

```
WITH SEX EQ 'M' AND (AGE LT '25' OR AGE GT '65')
AND ACCIDENTS GE '3'
```

Now let's add a third age group to the above, the 35-40 age group:

```
WITH SEX EQ 'M' AND (AGE LT '25' OR (AGE GE '35'
AND AGE LE '40') OR AGE GT '65') AND ACCIDENTS
GE '3'
```

Actually, because of the heierarchy rule (AND before OR), the inner set of parentheses would not be necessary. However, it doesn't hurt to include them because, if nothing else, it adds to the clarity of the statement.

Continuing to modify our example, suppose the test is still to be made on the same age groups of males, but instead of combining the test with the number of accidents, we wish to know if the male drivers are either in those age groups or have three or more accidents:

```
WITH SEX EQ 'M' AND ((AGE LT '25' OR (AGE GE '35'
AND AGE LE '40') OR AGE GT '65') OR
ACCIDENTS GE '3')
```

You can see that the selection clause can take into consideration any desired combination of the data, but also soon becomes somewhat difficult to keep the original intention clear. A "MACRO" capability will be presented in section 7.0 which will permit the composing of incredibly complex selection clauses, yet they will be very clear and relatively easy to work out. However, one should keep in mind that, as the complexity increases, so does the processing time. Therefore, the selection clause should be only complex enough to achieve the desired report.

In summary, if three dots (...) can be taken to represent an arbitrary comparison of data, then a compound selection clause can be of the form:

WITH ... AND ... OR ... AND ... OR ... etc.

The AND and OR connectives can be in any order or combination. Additionally, a parenthesized expression can occupy the place of any of the three dots (...), where the parenthesized expression is of the form:

(... AND ... OR ... AND ... OR ... etc.)

And other parenthesized expressions can be substituted for any or all of the three dots (...) in the above parenthesized expression, and so on.

5.26 THE USE OF ADJECTIVES IN THE SELECTION CLAUSE

The selection clause adjectives are: NULL, PRESENT, SMALLEST, GREATEST, FIRST, LAST, and NEAREST. The adjectives are used in the comparison part of the selection clause instead of the relational. They precede the Attr-Name they modify. Some data comparisons using the adjectives might be:

```
NULL DEGREE              (No data in the DEGREE data field)
PRESENT DEGREE           (Any data in the DEGREE data field)
SMALLEST SALARY
GREATEST SALARY
FIRST RECRUIT
LAST RECRUIT
NEAREST CAREER-YEARS '25'
```

The above comparisons can be compounded using AND and OR connectives just as the other data comparisons.

Note that the NULL adjective designates a data field which contains nothing but blanks. This would be equivalent to the comparison:

```
DEGREE EQ ' '
```

where the empty data string signifies an empty DEGREE data field.

The PRESENT adjective signifies just the opposite, that some data (any data) exist.

Note, also, that the adjectives, SMALLEST, GREATEST and NEAREST imply that the data field to which they are referring is numeric.

5.27 NEGATED COMPARISONS IN THE SELECTION CLAUSE

The NOT adjective can be used to negate any comparison. When used, the NOT precedes a left parenthesis. Thus, it might be a single comparison or compounded comparisons within the parentheses.

The NOT adjective negates the interpretation of the comparison. Having the parenthesized comparison(s) in mind, the NOT adjective might be thought of as "Anything other than ...". For example, building on a previous illustration:

```
WITH SEX EQ 'M' AND NOT (AGE LT '25' OR AGE GT '65')
```

By including the NOT adjective, the group in view is now changed from males under 25 and males over 65 to males in the 25-65 age bracket.

5.28 ABBREVIATED "OR" COMPARISONS

It is often desirable to compare an Attr-Name to several data strings in a compound OR relationship. One such example follows:

```
WITH CITY EQ 'WASHINGTON, D.C.' OR CITY EQ 'CHICAGO'
OR CITY EQ 'LOS ANGELES' OR CITY EQ 'ATLANTA' OR
CITY EQ 'NEW YORK'
```

A special abbreviation exists for the kind of compound OR clause shown above which is functionally equivalent but much shorter:

```
WITH CITY EQ 'WASHINGTON, D.C.'  'CHICAGO'
'LOS ANGELES'  'ATLANTA'  'NEW YORK'
```

Thus abbreviated, the various data strings stand in an OR relationship to one another, assuming that the comparison portion (e.g. CITY EQ) is repeated for each data string. In other words, between each data string should be assumed a repeat of OR CITY EQ. Of the repeated portion, the OR is constant for all such abbreviations and the CITY EQ portion would be a repeated copy of whichever Attr-Name and relational headed the clause.

The above abbreviation also holds true for numeric Attr-Names and data strings.

5.29 SELECTION CLAUSE VARIATIONS

It is permissible to repeat the WITH or WHERE preposition (WHEN, if form two of the query statement) after any AND or OR connective. The effect of adding this preposition is the same as if parentheses were inserted at that point. For example:

```
WITH SEX EQ 'M' AND WITH AGE LT '25' OR AGE GT '65'
```

This selection clause is equivalent to the following:

```
WITH SEX EQ 'M' AND (AGE LT '25' OR AGE GT '65')
```

It is also permissible to repeat the WITH or WHERE preposition (WHEN, if form two) in place of the connective. If this option is used, the AND connective is assumed. In the first example in this section where the "AND WITH" appears, it would have been equally permissible to omit the AND as follows:

```
WITH SEX EQ 'M' WITH AGE LT '25' OR AGE GT '65'
```

Or equivalently:

```
WITH SEX EQ 'M' WHERE AGE LT '25' OR AGE GT '65'
```

AD-A077 988 NORTHWEST REGIONAL EDUCATIONAL LAB PORTLAND OR F/G 5/9
RESEARCH IN ADAPTABLE PROGRAMMING TO ACHIEVE COMPUTER INDEPENDE--ETC(U)
DEC 77 C E FRYE DAHC19-76-C-0022
UNCLASSIFIED ARI-TR-77-B4 NL

2 OF 2
AD A077988



END
DATE FILMED
1-80
DDC

Or, if it appeared in the second form:

WHEN SEX EQ 'M' WHEN AGE LT '25' OR AGE GT '65'

If there are multiple uses of the WITH, WHERE and WHEN prepositions, then the parenthesized enclosure which opens at the preposition closes at the next preposition. Thus, the interpretation of the above example would be:

WHEN (SEX EQ 'M') AND (AGE LT '25' OR AGE GT '65')

Note that the "AND" was inserted in the explanation because "AND WHEN" is assumed since "WHEN" appeared alone, and the "WHEN" then was dropped out and replaced with a parenthesized enclosure.

There is no particular processing advantage to either of the above variations. Rather, it is mainly a matter of user preference.

5.3 THE REPORTING VERB

The reporting verbs have been listed in section 3.11. The positions in the query statement, depending on form, have been shown in section 3.13 as a part of the two general forms. Both are very similar and the choice of one or the other is simply a matter of user preference.

The reporting verbs, together with the Attr-Names which follow them (if any), determine the format and content of the computer's response to the query statement. A more complete discussion of the report format will be deferred to section 6.0.

There are six reporting verbs, LIST, LIST-VERTICAL, LISTV, COUNT, TOTAL and AVERAGE, of which two (LIST-VERTICAL and LISTV) are identical, differing only in spelling. Therefore, there are five different report formats which may be requested.

Each of the reporting verbs also has some rules regarding what particle names, if any, should appear with it. Therefore, each will be taken in order.

5.31 LIST

The LIST verb requests a listing, in columns, of the data appearing in the data fields which are identified by the Attr-Names immediately following. Thus, for the LIST verb, at least one Attr-Name must follow, with as many more as desired.

The data which are to be listed will come from those fields in the data base designated by the Attr-Names, but only from those fields in entries which qualify according to the selection clause. (See section 5.22). It is not necessary that data which are tested in the selection clause also be reported. The selection clause can test data in some fields and the reporting verb can display data from others. Or they may be the same fields if desired. For example:

```
WITH SEX EQ 'M' WHERE AGE LT '25' OR AGE GT '65
LIST NAME AGE ACCIDENTS #
```

The above example is a complete query statement (using an unnamed data base, but one that is assumed to be currently active). SEX and AGE are the Attr-Names (data fields) being tested while NAME, AGE and ACCIDENTS are being reported. Only the NAME, AGE and ACCIDENTS will be reported for those data base entries which meet the requirements of the selection clause (i.e. for which the selection clause comes out "true").

The column headings of the report will be the Attr-Names (underscored), and the data from the respective fields will be listed, row-by-row, under the appropriate column headings.

If no Attr-Name follows the reporting verb, LIST, the query statement will be rejected.

If there are so many Attr-Names after the LIST verb that the report will not fit in columns across the page, then the reporting format will be switched to the LIST-VERTICAL format, described in the next section.

An example of the report for the above illustration might be:

NAME	AGE	ACCIDENTS
JOHN SMITH	23	4
WILLIAM JONES	67	2
JAMES JOHNSON	19	3

etc.

5.32 LIST-VERTICAL AND LISTV

LIST-VERTICAL and LISTV are alternate spellings for the same command. The only difference between these and the LIST verb in the previous section is the format used for displaying the report. Whereas the LIST verb attempts to display the data in columns, the LIST-VERTICAL verb always displays vertically down the page.

In the illustrations in section 5.31, if the verb had been LISTV instead of LIST, the report example would be:

```
     NAME:  JOHN SMITH
      AGE:  23
ACCIDENTS:  4

     NAME:  WILLIAM JONES
      AGE:  67
ACCIDENTS:  2

     NAME:  JAMES JOHNSON
      AGE:  19
ACCIDENTS:  3
```

etc.

The LIST-VERTICAL verb also must be followed by at least one Attr-Name, just as the LIST verb, and can have as many more as desired.

5.33 COUNT

The COUNT verb simply reports a count of the number of data base entries which qualify from the selection clause tests.

Actually, the count is given at the end of every report so that the COUNT verb is equivalent to requesting only that part of the report and nothing more.

The count will be shown as follows:

COUNT: 37 OF 142

where the first number is a report of how many entries qualify and the second reports how many entries were tested.

This report will also be given at the end of the reports for each of the other reporting verbs. For example, in the LIST report, the COUNT should correspond to the number of entries which are listed under the headings.

The second number in the COUNT report will always report the total number of entries in the data base.

No Attr-Name follows the COUNT verb, and if any are shown, the statement would be rejected.

5.34 TOTAL

The TOTAL reporting verb is meant to provide an arithmetic sum of the numeric values in the Attr-Name data fields. Exactly one such Attr-Name should appear after the TOTAL verb, and that Attr-Name must contain only numeric values in its data field.

Note that the numeric sum is not the sum of all entries in that data field but rather only the sum of the entries which qualify according to the selection clause. If there is no selection clause, or if the selection clause is written in such a way that every entry qualifies, then the sum would include every entry.

5.35 AVERAGE

The AVERAGE verb is very similar to the TOTAL verb except that the arithmetic sum is divided by the count of addends before it is reported.

Since the COUNT is included as a part of each of the other reports, one could compute the AVERAGE from the TOTAL report.

The AVERAGE value that is reported is the arithmetic mean of the values of those numbers in qualifying entry data fields.

Exactly one Attr-Name follows the AVERAGE verb.

5.4 THE REPORT CLAUSE

The report clause consists of the list of Attr-Names (attribute names) if any that appear following the reporting verb. Each verb has its own requirements for the number and type (string or numeric) of Attr-Names which may follow it. These were discussed in the several previous sections (5.3 and following) that described the reporting verbs. These requirements will be recapped briefly here.

The LIST, LIST-VERTICAL and LISTV verbs may be followed by one or more Attr-Names of either (or mixed) types.

The COUNT verb has no Attr-Name following it, therefore has no report clause.

The TOTAL and AVERAGE verbs each have exactly one Attr-Name following them in the report clause, and the data field designated by that Attr-Name must contain only numeric values.

Attr-Names in the report clause are separated only by spaces (one or more). Any other separating characters (commas included) are illegal.

5.5 TERMINATOR CHARACTER

Every query statement must be terminated with a # character. The function of the # termination character is to inform the system which line of a multi-line query is the last one, even if the entire statement happens to fit on one line.

If the final line of the query statement has been sent to the computer before it is discovered that the # character was left off, it is permissible to enter that character as the first and only one on the next line. This effectively continues the line only for the purpose of terminating it.

6.0 REPORTING FORMATS

Most of the reporting format considerations pertain to the columnar and vertical formats of the LIST and LISTV verbs, respectively. The remaining reports are trivial but will be shown for the sake of completeness.

6.1 NUMERIC VALUE REPORTS

The verbs, COUNT, TOTAL and AVERAGE each only report numeric values. In addition, the COUNT report appears at the end of each LIST and LISTV report.

Examples of the COUNT, TOTAL and AVERAGE report follow:

```
COUNT:  228 OF 591

TOTAL:  43149
COUNT:  228 OF 591

AVERAGE:  189.25
  COUNT:  228 OF 591
```

The above examples assumed that the three query statements which produced them differed only in the verb that was used (and no Attr-Name in the case of COUNT). Note therefore that the AVERAGE report can be computer from the TOTAL report and conversely.

6.2 DISPLAYING DATA

The two display formats, columnar and vertical, have already been discussed and illustrated in sections 5.31 and 5.32. Additional variations will only occur in the number of attributes for which data are displayed, and the location of the displayed data on the page. This can have some bearing on the size of the name (number of characters) one might want to choose for a given kind of attribute.

6.21 COLUMNAR DATA DISPLAY

Columnar data is displayed by using the LIST verb, but only if the data columns do not exceed the width of the display page. Otherwise the display will be identical to the LISTV (vertical) display.

One of the considerations which will keep the LIST display in columnar format will be the number of Attr-Names in the display. If a large number are specified in the display, there may not be enough room on the display page to list the data in columnar format. Two variables determine the amount of column space needed for each Attr-Name: 1) the number of columns in the data field specification, and 2) the number of characters in the Attr-Name. Both of these values are fixed at the time the DECLARE statement is entered for the Attr-Name in question. (See sections 4.4 and 8.5 for more information on the DECLARE statement). The column space needed will be the larger of those two numbers plus the two spaces that separate the columns.

Thus, if the user intends to display data in columnar format, it might be wise to attempt to limit the number of characters in a chosen attribute name to no more (or little more) than the number of columns in the data field being declared. Using the example in section 5.31 of a columnar display, the NAME field is apparently determined by the length of the data field assigned to it (21 columns in the example), the AGE field might be three, the same as the number of characters in the word, AGE, but the ACCIDENTS field is certainly determined by the size of the chosen attribute name, ACCIDENTS. With only three columns of data in the report, the difference is minimal. However, if several more Attr-Names had been specified in the query statement for the report, the ACCIDENTS column might have made the difference between a columnar or a vertical report.

By the same token, it is wise not to include more columns in the data field than are needed to represent any of the universe of data which might appear there. The more columns of data there are, the more spread out the columnar report will be.

6.22 VERTICAL DATA DISPLAY

Vertical data displays result from the use of the LIST-VERTICAL and LISTV verbs, and can result from the use of the LIST verb if the display is too wide for the columnar format.

Vertical formats are organized by repeated listings of the Attr-Names down the left margin, indented so that the colon characters line up vertically (as shown in the example in section 5.32). The data are listed after the colon. If the length of the data listing exceeds the width of the page, the remainder of the entry will appear on the next line. Thus, virtually any length entry can be accomodated.

Both displays, columnar and vertical, include a COUNT display after the last data line.

7.0 MACROS

One never has to use macros to operate the query system but the use of macros generally simplifies its operation. It is well worth the little extra time required to learn to use them.

The Macro feature simply allows one to equate a user-chosen name to any fixed sequence of characters. Then, when that Macro name is used in a query statement, the character string will be substituted in its place before the data base is searched.

The Macro feature is especially useful for any or all of the following applications:

- Abbreviation for character strings which will be repeated in several query statements for the same data base (regardless how much time separates the use of those query statements).
- Abbreviation for data comparisons (simple or compound) which clarify the writing of the query statement.
- Method of renaming primitive particle names to something more easily remembered.
- Method of attaching a name to a particular kind of reporting specification such that the same reporting format can be easily used with several different selection clauses.
- Abbreviation for the entire query statement so than an uninformed user can make that query with utmost ease.

7.1 DEFINING AND USING MACROS

The DEFINE verb is used to define the macro. Its input form is:

[FOR Dlname] DEFINE Macro-Name = anything #

Notice that the bracketed portion is optional. If omitted, the currently active data base dictionary will be the recipient of the new macro.

The Macro-Name is made up the same way as any other Query system name. (See naming conventions in section 2.1).

There are few restrictions regarding the content of the macro character string, represented by the word, "anything" above. However, a few simple guidelines will make them more useful.

Make the macro a "whole" entity. For example, if the macro is to contain part of a selection clause, make it an entire comparison (or compound comparison). An example of how not to do it may make the point:

```
DEFINE ONE = SEX EQ #

DEFINE TWO = 'M' AND AGE #

DEFINE THREE = LT '25' #
```

Now, with the above macros, the following query statement could be written:

```
WITH ONE TWO THREE COUNT #
```

While the above sequence is valid, the macros do little to help. A better way might have been:

```
DEFINE YOUNG-MALE-DRIVERS = SEX EQ 'M' AND
AGE LT '25' #
```

Now the query statement would be:

```
WITH YOUNG-MALE-DRIVERS COUNT #
```

When using macros to represent data comparisons (as was just illustrated), it is good practice to enclose the macro string in parentheses so that it will be treated as a single entity regardless where it appears within a selection clause. In the above case, it could appear:

```
DEFINE YOUNG-MALE-DRIVERS = (SEX EQ 'M' AND
AGE LT '25') #
```

In the above illustrations, the presence or absence of parentheses would have made no difference. However, if macros had been used to construct the selection clause shown toward the end of section 5.25, the use of parentheses could be very important. When the data comparisons in the macro definitions are complete, the parentheses are advisable.

Use macros in defining other macros, especially when complex query statements are necessary. For example, consider the simplicity and clarity of the query statement below after the macros have been defined:

```
DEFINE JAPAN = (MAKE EQ 'TOYOTA'  'DATSUN'
'HONDA'  'MAZDA') #

DEFINE GERMAN = (MAKE EQ 'VW' 'BMW' 'MERCEDES') #

DEFINE ITALIAN = (MAKE EQ 'FIAT') #

DEFINE BRITISH = (MAKE EQ 'ROLLS'  'JAGUAR'
'TRIUMPH') #

DEFINE FRENCH = (MAKE EQ 'RENAULT') #

DEFINE FORIEGN =(JAPAN OR GERMAN OR ITALIAN OR
BRITISH OR FRENCH) #

DEFINE GOOD-MILEAGE = (MPG GE '35') #

DEFINE FAIR-MILEAGE = (MPG GE '20' AND MPG LT '35') #

DEFINE POOR-MILEAGE = (MPG LT '20') #
```

Now, let's look at the lists of foriegn cars in each of the three mileage categories:

```
LIST MAKE WHEN FORIEGN AND GOOD-MILEAGE #

LIST MAKE WHEN FORIEGN AND FAIR-MILEAGE #

LIST MAKE WHEN FORIEGN AND POOR-MILEAGE #
```

Or, there would be many other alternatives, such as:

```
LIST MAKE MPG WHEN FORIEGN #

LIST MAKE MPG WHEN NOT(FORIEGN) AND GOOD-MILEAGE #

LIST MAKE MPG WHEN (BRITISH OR GERMAN) AND
(GOOD-MILEAGE OR FAIR-MILEAGE) #
```

(Please forgive the omission of any favorite foriegn car makes).

8.0 COMMAND VERBS

Command verbs are: PRESTORE, READ-CARDS, DATA-BASE, DEFINE and DECLARE. All except PRESTORE have been discussed at some length in earlier sections. The discussion in this section will be brief, giving the general forms for each.

8.1 PRESTORING A CARD DECK

The general form for the PRESTORE verb is:

PRESTORE Filename [n] #

The PRESTORE verb is used to load a card deck into the Query system, making it available one time only to any user who accesses it by using the READ-CARDS verb with the same Filename. Only the system operator is allowed to use the PRESTORE verb since the use of this verb must be coordinated with the placement of the card deck in the reader device.

No interpretation of the cards whatever will be done during the prestore operation. The last card belonging to the named Filename must contain dollar signs ($$) in the first two columns which signal the end of that deck. The $$ card will not be a part of the Filename reference, and will have no further function.

The purpose of the PRESTORE verb is to keep the operator in control of the actual card reader, turning the file over to the user who requested the prestore when the operation is finished. The prestore operation uses otherwise idle system time to perform its work.

The "n" parameter is optional. When omitted, "n" is understood to have the value, 80 (n=80), the number of columns on the card. Acceptable values for the "n" parameter are between one and 80, designating the last column on the card from which data are to be taken. Card columns beyond that number will be regarded as though they were blank. This permits the user to drop off such things as sequence numbers from the cards. For example:

PRESTORE NEW-DECK 72 #

In this example, Filename NEW-DECK would contain a copy of the card deck just read but with the last eight columns blanked out.

8.2 USING A PRESTORED CARD DECK

The user inputs to the Query system from card deck via a two-step operation. He first asks the operator to prestore the deck into a given Filename and also specifies the last data column (if less than the full card). The operator prestores the deck using the PRESTORE verb. (See section 8.1).

The user will be able to tell from the system messages when the deck is ready. If the prestore has not yet been initiated, the message, "PRESTORE FILE NAME IS NOT ON DISK." If it is in progress, the message will be, "PRESTORE FILE IS NOT READY YET."

The prestored file is accessed by the READ-CARDS verb. Its general form is:

[FOR Dlname] READ-CARDS Filename [n] #

The brackets denote optional entries, as before.

The cards that are read using this command verb will appear to be coming directly from the card reader, except much faster. Since the input format is generally the same for cards and for the terminal, the card deck represents a convenient method for stacking a large number of terminal inputs for efficient entry into the Query system. The most likely card entries will be the data base entries and the DEFINE and DECLARE statements. Together, these will comprise the entire data base and its dictionary. After the last card has been read, entry will switch back to the terminal.

If an error is encountered in reading the card deck, the error will be reported at the terminal and the remainder of the card deck will be discarded. It will then be the responsibility of the user to repunch the offending card and prestore again that card and the remaining ones. Using the READ-CARDS verb again with the Dlname option, the balance of the card deck will be added to the data base.

The "n" optional parameter in the READ-CARDS general form is similar in interpretation to the one in the PRESTORE verb form. (See section 8.1). However, instead of regarding the columns past "n" as being blanks, it instead regards the cards as being only "n" columns long. The difference is only pertinent while in the data base entry mode. (See section 3.2). Instead of the data entry being 80 characters per card segment, it will be taken as "n" characters per card segment. Thus, if n=20 and there were 10 cards per data base entry (i.e. the # character terminates each tenth card), then the 20 columns in the sixth card would be columns 101-120 in the data base entry, whereas they would be columns 401-420 if n=80. The user might find that 50-column cards (n=50) would make it easier to format the data base entries.

Section 4.3 shows another example of the use of the READ-CARDS verb.

After the READ-CARDS verb has been transmitted and after its execution completes, the "Filename" will no longer exist in the Query system, and that name can then be used for something else.

8.3 CREATING OR EXTENDING A DATA BASE

The use of the DATA-BASE command verb was discussed extensively in sections 4.0 through 4.3. Its general form is:

DATA-BASE [Dlname] #

The next input will be processed as an entry to the data base.

If the Dlname is given, then the associated data base will be extended by the next entries (if the data base by that name already existed), or a data base by that name will be created if the name is new. If the name does already exist and the input is from the terminal, then the user will be asked to confirm that the intention is to add to this data base.

If the Dlname is omitted, the currently active data base will be assumed, or an error will be reported if none is currently active.

8.4 DEFINING A MACRO NAME

The general form for the DEFINE verb is:

[FOR Dlname] DEFINE Macro-Name - anything #

The DEFINE verb has been discussed in detail in sections 7.0 and 7.1.

Note that defined macros become a permanent part of the data base dictionary and remain available whenever that data base is used.

8.5 DECLARING ATTRIBUTE NAMES

Attribute names are used to identify data fields within the data base when composing a query statement. The data fields may either be arbitrary strings of characters or numeric. The general form for either declaration varies only with the mnemonics STR (string) and NUM (numeric):

[FOR Dlname] DECLARE Attr-Name STR(c1,c2) #

[FOR Dlname] DECLARE Attr-Name NUM(c1,c2) #

The c1 and c2 parameters refer to the beginning column number and ending column number, respectively, of the designated data field.

If the second form is used (numeric declaration), the designated data field must only contain a numeric value, optionally including a decimal point and/or an arithmetic sign.

Note that declared attribute names become a permanent part of the data base dictionary and remain available whenever that data base is used.

9.0 CONTROL STATEMENTS

Control statements begin with a control character ($) in column one of the input and have no termination character. (See section 3.3).

The typical user needs only one control statement, $OUT, that signs that user off from the system. The remaining control statements are used only by the system operator.

9.1 USER SIGNOFF, $OUT

When any user wishes to sign off from the Query system, the control statement, $OUT, is input. The user may sign off at any time and resume again with the previous work at the next login.

9.2 OPERATOR CONTROLS

- $ADDLOG /userid/userid/userid/etc.

  This statement adds authorized login identities to the list.

- $DELLOG /userid/userid/userid/etc.

  This statement removes user identities from the authorized login list.

- $DATE=mmddyy

  This statement permits the operator to set the current date as six digits, two each for the month, day and year. In most cases, this will not be necessary as it will have been set automatically.

- $AUX n

  This state permits the operator to assign auxillary operator status to any terminal number "n". This status is limited to only one terminal at a time. Its main use is for remote maintenance of the system.

- $QUIT ALL

  This control statement halts the execution of the Query system after logging off any active users.

- Full interactive debugging is available to the operator which uses the same statement syntax as the PLANIT system (but with the addition of the $ prefix). Since this feature has very limited application among the user community, it is not described in more detail here.

10.0 OPERATING PROCEDURES

The operation of the Query system is extremely simple. There is nothing to load, nothing to save and no operations which must complete prior to sign off. Each operation is complete in a single- or multi-line entry. Although at first it may appear that the loading of a card deck is similar to loading a program, a simple examination of the deck shows it to be only a batch of terminal inputs which have been prepared ahead of time for rapid entry.

There are a few items of information which will help in using the system.

10.1 SIGNING ONTO THE SYSTEM

After connecting the terminal to the Query system (by dialing or whatever), the user will respond to the message:

PLEASE LOG IN

The login name must have previously been put into the authorized login list. The operator can do this.

10.2 TERMINAL PROMPTS

There are three terminal prompts, corresponding to the three possible kinds of entries which might be expected. They are the following:

DMS> This abbreviation for Data Management System indicates that the user can start any query statement or control statement on this line.

< > This indicates that the computer is expecting the continuation of a query statement which was started on a previous line.

1> This prompt indicates that the Query system is expecting data to be added to a data base. The number indicates the column number of the entry. Thus, the "1" indicates a new entry, and continuation lines will have correspondingly larger column numbers in the prompt. When the # character has terminated the entry, the next column number prompt will again be "1."

10.3 LINE ECHOING

Query statements will be repeated for the user. The echo of the statement will normally occur after the # terminator character has been typed and entered. However, a partial echo will occur sooner in the event of an error. This will be discussed more fully in the next section.

In addition to assisting in error detection, the echoed statement shows the user exactly what search format will be used. Actually, it is not an echo, but a reconstituted line. If the Dlname has been omitted from the statement, it will appear in the echo. If macros are used, the substitutions will appear in the echo. Therefore, the echo can be much longer than the typed query statement. The echo will show exactly what the statement would look like after default conditions have been supplied and macros have been eliminated.

The echoed line will be most useful in the event that the user suspects that the desired data are not being found for the report, to provide additional information for correcting the query statement.

10.4 ERROR MESSAGES

Error diagnostic messages will appear at the terminal in the event that the statement will not execute. In addition, if the error has been discovered in the syntax of the query statement, the echoed line will end at the point the error was discovered. The user will have both the error message and the position indication of the error.

10.5 CORRECTING INPUT LINE ERRORS

The user should follow local conventions for single character corrections (rubout or backspace) and single line corrections (line cancellation).

In addition, an entire multi-line query statement may be cancelled by entering a blank line (only the carriage return key) in place of the next continuation line. The next prompt will be:

DMS>

Note however that the result would be different if the blank line should be entered while building a data base. In that case, a number of blanks would be entered into the data base line which is equal to the columns on the terminal.

Also, in this regard, should an error be discovered by the PQS syntax check, it will be reported at the time the offending line is entered. Thus, if the query statement is to extend over several lines, the error will be reported when it is made, not when the query statement is complete. Following the report of the error, the entire query statement must be re-typed.

10.6 MODE SWITCHING

There are three basic modes of operation:

- Query mode
- Data base creation mode
- Control statement mode

The Query mode will be prompted by the "DMS>" characters. To change from the Query mode to the Data base creation mode, the DATA-BASE command is used. The data base column number prompt will remind the user of that mode. To revert back to the Query mode, the "#" character is entered as the only character on the line. (It may have to be done twice if the first one is used to terminate a data base line in progress). The Control statement mode is entered via the initial "$" character and always reverts automatically to the Query mode again.

10.7 SIGNING OFF FROM THE SYSTEM

To sign off from the system, the user types:

$OUT

Note, however, that this is a Control Statement format so it can only be entered while in the Query Statement mode. In case the user is in the Data Base Creation mode at the time the "$OUT" command is desired, the mode must first be switched. This can be done by entering a "#" character alone on the line. If a data base line was in progress, this will result in a " 1>" prompt, and a second "#" line will need to be input.

When the "DMS>" prompt is seen, the user can enter the "$OUT" control statement. (See also Section 10.6 for mode switching).

APPENDIX A-A

INSTALLATION INFORMATION

INSTALLATION INFORMATION

Installation of the Portable Query System is nearly identical to that for PLANIT (Programming LANguage for Interactive Teaching), a computer-assisted instructional time-sharing system which is also portable. Since installation documentation already exists for PLANIT, this discussion will be limited to differences in the installation of the PQS from PLANIT.

DIFFERENT INSTALLATION PARAMETER NAMES

There are seven parameter names in the PQS listing which either are not found in PLANIT or appear with a different meaning: CONTCHR, DIGITS, NUMRECS, CHKPREP, ENTRYSIZE, DICTSIZE and WORKSPACE. There are several parameter names which appear in PLANIT which do not appear in PQS that should be disregarded. The above seven names have comments in the listing which attempt to explain the purpose of the parameter names. This discussion will give a little more detail for each.

CONTCHR

This parameter simply selects the character which will be used as the Control Statement prefix character. The value chosen must be one from the list of characters in the character set listing following the parameter listing. Thus, the number for the dollar sign ($) character is 55. This can be changed to another character so long as it is not a letter or a digit and is not used for other purposes in the language.

DIGITS

This parameter allocates space for numbers associated with the primitive particle names, GREATEST, SMALLEST, NEAREST, TOTAL and AVERAGE. The number of digits chosen should be the sum of the digits to the left of the decimal point plus the digits to its right. For example, if it is desirable to provide space for a number whose maximum size will be 10^6 carried out to four decimal places, the value for DIGITS should be 10.

NUMRECS

Although NUMRECS has the same interpretation as in PLANIT, the considerations which go into choosing a value will be somewhat different. A paragraph below will discuss the disk files for PQS and will suggest an appropriate allocation of records for each file. The NUMRECS parameter value should be the sum of those numbers of records (or larger to allow for growth).

CHKPREP

In the two general forms of the query statement, the document presents a distinction in the placement of the prepositions, WITH, WHERE and WHEN. However, the system will work equally well if no distinction is enforced. A value of "0" for this parameter will drop any distinction between these three preposition names.

ENTRYSIZE

This parameter will determine the size of the data buffers for the data base, and in so doing, will limit the size of any one entry in a data base. The value chosen for this parameter is not necessarily related to efficiency since multiple entries will be packed into single buffers when sizes permit. In fact, it would be more desirable to have large buffers with several entries per buffer than to have small ones since it would tend to reduce the number of disk accesses in searches of the data base. The resulting buffer size will be slightly larger than the ENTRYSIZE divided by the number of bytes in a computer word. Therefore, values in the range of 1000 to 5000 should be appropriate.

DICTSIZE

This parameter represents an approximation of the number of particle names which can be added to the data base dictionary, including all associated with the DECLARE (Attribute Names) and DEFINE (Macros) verbs. Since the length of the macros are indeterminate, the number will not be exact. The effect of large values for this parameter will show in sizes for File number two. Values in the range of 25 to 100 should normally be sufficient.

WORKSPACE

This parameter is possibly the most difficult to assess. It determines the amount of space to be available for "parsing" a query statement into its component parts, to be ready for the data base search. The component parts include particle names (one entry each) and other (nor-name) characters (one entry each). The query statement to be parsed includes the concatenation of all continued lines up to the termination character, expanded by the substitution of all Macro texts.

The value chosen for this parameter will limit the length (and complexity) of the query statements which might be composed. A value of 300 would seem to be sufficiently generous.

The remaining parameter names are no different than in PLANIT.

DISK FILES

The PQS uses the same disk file numbering system as PLANIT, using the same file numbers for the same purposes to the extent possible. Whereas PLANIT normally uses 10 (or 11) disk files, PQS uses eight. The chart below shows the files and suggests an appropriate number of records to allocate for each. The discussion which follows the chart comments on file sizes. The "Activity index" column on the chart is meant to suggest the relative frequency of use in case there is an optimum choice of placement on disk.

File Number	Identification	No. Records	Activity Index
1	Table of Contents	2-3	4
2	Data Base Dictionary	1/Data Base	5
3	Not used	0	
4	Data Base Entries	500 plus	1
5	Not used	0	
6	User Swaps	1/Active user	2
7	Not used	0	
8	Snapshot for error recovery	1	8
9	Core Initialization	2	7
10	Prestored decks	100 plus	6
11	Messages	10-20	3

Files one and four have the same size records, the size being determined by the ENTRYSIZE parameter.

The record size of File two is set from the DICTSIZE parameter.

Files six and nine contain the "user swap" portion of the Common data, beginning with the "NTOP" item name and continuing through the end.

File eight is small, mainly containing the "IDISK" array and the "LOGNR" array entries.

Files 10 and 11 have the same size records, being set from the SPOOLSZE parameter. Since cards are handled in "batches", SPOOLSZE sets the size of those batches. Thus, SPOOLSZE multiplied by the number of records in File 10 will determine the largest single deck (or collection of smaller decks) which may be prestored prior to being used. The MSGBUG array will contain the data which are used with Files 10 and 11.

MIOP INTERFACE

The MIOP interface codes are identical to those for PLANIT. However, there are several operations which will not be used at the present time, including:

- Card punch
- Line printer
- Tape read and write
- Timed terminal read

It is entirely proper to use the same MIOP, LDBYTE and SBYTE routines for PQS as for PLANIT even though some of the operations will not be exercised. However, the disk files attached to the PQS MIOP should not be the same as those attached to the PLANIT MIOP if either are to be preserved.

APPENDIX B - B

INITIALIZATION DECK

INITIALIZATION DECK

The Initialization Deck for PQS is similar to (and serves the same function as) the Cardsfile deck in PLANIT. Of course the messages and initialization data are different, but the same purposes are served.

Among those data which should be of concern to the operator in the Initialization Deck are the following:

- "C", "W", "H" or "F" as the first character of the first card to designate Cold, Warm, Hot or Forced starts, respectively. The remainder of that card should remain unchanged once the final character set is established. The four starts are the same as in PLANIT, with the Cold start to initially start the system, the Warm start to restart the system only with data changes, the Hot start for usual restarting, and the Forced start as a very last resort solution in case the Warm and Hot starts repeatedly fail.

- The IFBTN card must contain numbers chosen for the records allocated in each file, including zeros for those files where no records are needed (Files 3, 5 and 7).

- Operator terminal number.

- Quantum size (about one second is average).

- Login identities.

In the case of a "Hot" start, only the first card will be read, thus only that card needs to be used. The remaining start types use the entire deck.

APPENDIX C - c

CONVERT

A DATA BASE CONVERSION PROGRAM FOR

PLANIT STUDENT RECORDS

CC

CONVERT

A Data Base Conversion Program For PLANIT Student Records

Prepared for:

The United States Army Research Institute

Alexandria, Virginia

Contract No.

DAHC19-76-C-0022

Ron Silberman

The Northwest Regional Educational Laboratory

Portland, Oregon

December 1977

ABSTRACT

This paper describes the CONVERT program. CONVERT is designed to convert PLANIT Student Record data into punched card format such that it can easily be input to a data management system, particularly the On-Line Query System. This enables PLANIT authors to analyze responses the students have given to their prepared lesson scenarios.

By running CONVERT repeatedly, using different PLANIT data files each time, the results of records pertaining to several PLANIT lessons can be converted into a large single data base, suitable for scanning with Query statements.

CONVERT must run on the same computer used by PLANIT to create the Student Record files, but the punched card output from CONVERT could be used on any machine.

This document also discusses the installation and operation of the CONVERT program.

TABLE OF CONTENTS

Section		Page
I.	INTRODUCTION	1
II.	CONVERT'S OUTPUT	4
III.	USING CONVERT'S OUTPUT	7
IV.	THE DECLARE FILE INPUT	9
V.	THE STUDENT RECORDS FILE INPUT	10
VI.	OPERATING THE CONVERT PROGRAM	11
VII.	INSTALLING CONVERT	12

I INTRODUCTION

CONVERT is a batch computer program which is designed to be used in conjunction with two time-sharing systems, PLANIT and the On-Line Query System. It is necessary to assume that the reader has some familiarity with both of these time-sharing systems in order to explain the function of the CONVERT program. Both systems operate independently and information about them can be obtained from their operation manuals, The PLANIT Author's Guide, TM-(L)-4422/001/01, and the Query System Manual. It is easiest to begin describing CONVERT with some preliminary information about these neighboring time-sharing systems.

The On-Line Query System is designed to scan a data base of a pre-specified format and select those records which correspond to specific data values. The operator can set the data values in a form sentence which includes logical connectives, (AND, OR, NOT). This sentence is called a query statement and can be used to direct a search for those records which correspond to the data values which were specified in the statement. The data base must be arranged as a set of data records, all of uniform format and a set of system statements which name the data base and specify the fields for each variable on the data records. The form for a data base is discussed in more detail in section II of this paper.

PLANIT is also a time-sharing system, but it serves an entirely different purpose than the Query System. In author mode, PLANIT provides its user's with all of the tools which are necessary to write a lesson which can later be used for interactive on-line teaching. The PLANIT

lessons consist of an ordered sequence of questions and decision points which are called frames. When a student executes a PLANIT lesson, (execute mode), he types answers to the sequence of questions as they appear on the computer terminal. Depending on his responses, the student may be branched to other frames in the lesson or to other lessons in order to adjust the instructional material to his rate of learning. All of the questions and the branching options must be initially constructed as frames in PLANIT by the lesson author. As a student executes a lesson, a record of his performance is automatically stored onto a disk file called the STUDENT RECORDS file. The data entries which are kept on the STUDENT RECORDS file are listed in section II of this document.

One major reason for keeping records of students' performance is to provide the author with a means for judging the effectiveness of the lesson. An author can view these records at his terminal, one student's record at a time, and determine how well each question fits into the programmed instruction sequence. For example, if a question frame in a lesson does not convey any new text material and all of the students are answering it incorrectly, the author may want to remove it from the lesson or change it into a more useful entry for teaching. To print out a student's performance record the author must first put the record into computer memory using the 'GET' and 'ATTACH' commands in PLANIT. Then to print the record the author can use the 'DISPLAY' command. For more detail on the use of these commands see the PLANIT Author's Guide, page III-35. The ability to scan these performance records for specific items can be very valuable to an author. After noting this value it should

become evident that a program which converts PLANIT student records into data base form suitable for Query System processing would be extremely useful. The CONVERT program is designed to perform this task. In relation to the discussion above, the description of the CONVERT program can proceed.

CONVERT is a batch program that converts a PLANIT STUDENT RECORDS file from the PLANIT time-sharing system into a data base which is suitable for use in the Query time-sharing system. Each record of the data base which is produced will correspond to a student's response to a particular frame in the PLANIT lesson. CONVERT works on the tape format of the PLANIT student records to produce a data base. The tape format of the PLANIT student records is obtained by entering the command 'UNLOAD' in PLANIT. For details of the UNLOAD command refer to the PLANIT Language Reference Manual, page VI-11. The tape format form of the student records does not imply that they are necessarily stored on tape. They might be stored on disk depending upon how PLANIT was installed on the given target machine. CONVERT runs as a batch job with one run for each UNLOADed STUDENT RECORDS file. The UNLOADed file consists of all the records, (one record per student), which were produced from the execution of one PLANIT lesson. Convert will produce a data base each time it is run. Each data base corresponds to a PLANIT lesson.

The Query system provides for the combining of several data bases into one large data base. This enables the operator to scan data which spans the range of several PLANIT lessons. For additional information on concatenating data bases see section III. When converted into data base form the format of each record of student performance data is uniform and all of the additional statements which are necessary to complete the data base are added by the CONVERT program. CONVERT's output file can be used directly

as input to the Query System. In most cases, PLANIT, CONVERT, and the On-Line Query System will be used on the same computer. In fact, this is required for PLANIT and CONVERT. It is possible to take a CONVERT data base which was produced on one machine and process it on another machine using the Query System. This capability makes it possible to carry out the sequence of events which are needed to analyze student performance data even though PLANIT and the Query System may not be available on the same machine.

In summary, the CONVERT program enables PLANIT authors, using the Query System, to analyze the responses to specific questions in a lesson. Each of the variables in the output format can be used as selection criteria within the Query System. More than one PLANIT lesson can be converted into data base form by running CONVERT several times. Each of the outputs which are produced by multiple runs can be prestored as a single data base in the Query System. Coordination of the operation of PLANIT, CONVERT and the Query System is possible on one machine or on separate machines. This suggests that several authors in various locations can analyze the empirical results of PLANIT lesson execution.

II CONVERT'S OUTPUT

The CONVERT source code contains one 'WRITE' statement. The output file will be written onto logical unit number 2 unless the 'WRITE' statement has been altered to do otherwise. The output device can be a card punch, tape drive, or disk, depending on how the logical unit numbers are assigned

each physical device. The size of the output file can be estimated by relating to the size of the PLANIT STUDENT RECORDS file which is to be used as input to CONVERT. Each recorded response to a frame number will be directly converted to one 80-character record of output. In addition to this there are 13 cards from an initialization file which will be transferred directly to the data base. Therefore, if the PLANIT lesson contains 100 frames and 15 students complete the lesson, the space allocation should provide for at least 15 * 100 + 13 = 1,513 records of output. An example of a data base which contains only 17 records is as follows:

```
DATA-BASE #
  1.00          Q    5 R A N  12077  720     LESSON-NAME      STUDENT-ID #
  2.00    1.00 M    10 W C N  12077  720     LESSON-NAME      STUDENT-ID #
  2.50    2.00 Q     1 N - N  12077  720     LESSON-NAME      STUDENT-ID #
  2.75    2.50 Q    65 R B Y  12077  720     LESSON-NAME      STUDENT-ID #
DECLARE FRAME-NO NUM(1,6) #
DECLARE LAST-FRAME NUM(8,13) #
DECLARE FRAME-TYPE STR(15,15) #
DECLARE RESPONSE-TIME NUM(16,20) #
DECLARE N-W-R STR(22,22) #
DECLARE TAG STR(24,24) #
DECLARE CALC STR(26,26) #
DECLARE DATE NUM(28,33) #
DECLARE SIGN-ON-TIME NUM(35,38) #
DECLARE LESSON STR(39,54) #
DECLARE STUDENT-ID STR(55,70) #
$$
```

The first card, 'DATA-BASE #', is a directive to the Query System to allow the operator to choose a name for the given data base. The 'DECLARE' cards are Query System format specifications for the named variables. The NUM(1,6)

part of the first 'DECLARE' statement indicates that the variable 'FRAME-NO' has numeric values and it can be found in columns one to six inclusive. Note that in the example above the frame numbers are similar to a FORTRAN format F6.2 (i.e. 1.00). DECLARE FRAME-TYPE STR(15,15) # indicates that the 'FRAME-TYPE' variable is string-valued and its value is located in column 15. Each of the four data records shown above are coded as follows:

VARIABLE NAME	COLUMN	POSSIBLE VALUES
FRAME-NO	1-6	The actual PLANIT frame numbers with format F6.2
LAST-FRAME	8-13	The previously executed frame number with format F6.2
FRAME-TYPE	15	P, D, M, or Q
RESPONSE-TIME	16-20	Response time in seconds
N-W-R	22	N, W or R (Neutral, Wrong or Right)
TAG	24	The answer tag which identifies the student's response
CALC	26	N or Y answering the question: was CALC used in this frame?
DATE	28-33	The date when the student signed on, in the form MMDDYY
SIGN-ON-TIME	35-38	Time when student signed on, in minutes since midnight
LESSON	39-54	The actual PLANIT lesson name
STUDENT-ID	55-70	The log in identity used by the student

The data bases created by successive CONVERT runs will all be of the same form. The above attribute name list will provide the operator with a reference guide for constructing query statements. It is possible to change the names of the attribute names by simply changing the 'DECLARE' cards in the initialization file for CONVERT before running. The attribute names must contain no spaces and can consist of letters, digits and hyphens only.

III USING CONVERT'S OUTPUT

After CONVERT has run to completion the output file can be prestored into the Query System file dictionary by the following command at an operator's terminal:

PRESTORE FILE-NAME #

The FILE-NAME in the PRESTORE command can be made up of letters, hyphens and digits but it cannot contain spaces. The FILE-NAME will be catalogued into the system dictionary so that any user can build the data base in his work area. To build a data base at a user's terminal enter the commands:

DATABASE DB-NAME #

#

READ CARDS FILE-NAME #

The first command gives a name, DB-NAME, to the data base which is about to be built. Note that the user can substitute any name of his choosing for DB-NAME. At this point the user is in data base mode. A data base can be entered directly on the user's keyboard. In order to read the CONVERT data base from FILE-NAME, the user must exit data base entry mode by entering a # mark as shown above. The READ CARDS FILE-NAME # command simply reads the CONVERT data base from FILE-NAME into the user's work area under the name DB-NAME. The system will now accept query statements to scan DB-NAME. If the name DB-NAME had existed before the command DATABASE DB-NAME # was issued, the computer would have responded with:

DATA-BASE NAME EXISTS. ADD TO THIS DATA-BASE? (Y/N)

A 'N' response from the user would return him to command mode where a new name could be chosen. If the user had answered 'Y' to the above question, the existing work area would be concatenated onto the previously stored data

with the name DB-NAME. This is a powerful option to which the reader should take note. This situation allows the user to build a data-base which contains data from more than one PLANIT lesson. By simply entering the name of a previously stored data-base and then answering 'Y' to the above question, the user is adding data to an existing set. When a CONVERT data-base has been built in the Query System the information which is in the user's work area is virtually the same as the corresponding information which is shown by the PLANIT 'DISPLAY' command but the data is collected over as many students and lessons as desired.

There are three types of query statements which are best described by the following three examples:

```
FOR DB-NAME WITH FRAME-NO EQ '2' AND (N-W-R EQ 'W' OR RESPONSE-TIME
    GT '10') LIST STUDENT-ID RESPONSE-TIME N-W-R TAG #

COUNT DB-NAME WHEN TAG EQ '-' #

LIST DB-NAME STUDENT-ID FRAME-NO WHEN TAG EQ "'" #
```

The query statements above are self explanatory. The second statement simply counts the number of records which contain a '-' in the TAG field. The third statement gives an example of a search for an apostrophe. Instead of using apostrophes to delimit the data value, the quotes were used. It should be noted that WHEN always follows the verb and WITH comes before the verb. The first query statement gives an example of parentheses nesting. The result of this statement will be a list of STUDENT-ID, RESPONSE-TIME, N-W-R, and TAG which occur on those records where the stated conditions are met. For further information on the query statements the reader is referred to the Query System Manual.

IV DECLARE FILE INPUT

CONVERT reads both an initialization file and a STUDENT-RECORDS file as input and writes a data base as output. The DECLARE file serves a dual purpose: 1) The first card, (character set card), allows CONVERT to set up a list of the local machine's character codes. Each time CONVERT is installed on a new machine it has to be able to recognize a new set of character codes internally so that it can process the binary coded input. When the character codes from the first card are read, CONVERT associates them with their respective characters by their ordered column locations. All of the remaining input can be internally identified by simply matching the listed codes with their respective characters. The DECLARE file consists of the following 14 cards:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ+-*/().%=;:'@ #_?$<>"
DATA-BASE #
DECLARE FRAME-NO NUM(1,6) #
DECLARE LAST-FRAME NUM(8,13) #
DECLARE FRAME-TYPE STR(15,15) #
DECLARE RESPONSE-TIME NUM(16,20) #
DECLARE N-W-R STR(22,22) #
DECLARE TAG STR(24,24) #
DECLARE CALC STR(26,26) #
DECLARE DATE NUM(28,33) #
DECLARE SIGN-ON-TIME NUM(35,38) #
DECLARE LESSON STR(39,54) #
DECLARE STUDENT-ID STR(55,70) #
$$
```

The first card of the DECLARE file is used to identify the local character set. Each of the remaining cards is written, exactly as shown, onto the output file. The Query System interprets the 'DECLARE' statements as

field specifications for the data variables. The second card, (DATA-BASE #), is written out as the first record of the data base. If a name was added to this card just before the # sign, the Query System would associate that name with the data base upon reading this record. Since there is no such name, the Query System will allow the operator to choose a name for the given data base. Thus, if the operator wishes to append this data base onto another one, he need only enter the name of the previously stored data base on the terminal before appending this data base.

The formatted 'READ' statement in CONVERT is set to read the DECLARE file from logical unit number 5. The unformatted 'READ' statement, used to read the PLANIT STUDENT RECORDS file in UNLOADed tape format, uses logical unit number 1.

V STUDENT-RECORDS FILE INPUT

The PLANIT STUDENT RECORDS file is read one record at a time by an unformatted 'READ' statement. The file is read from logical unit number 1 unless this 'READ' statement has been altered. The student records are read in UNLOADed tape format and as mentioned in the introduction this may be a disk read depending upon how CONVERT was installed.

The first record of the STUDENT RECORDS file, the Head Record, will contain the PLANIT lesson name and will provide some parameter values which are needed to keep track of the remaining records.

If, for some reason, CONVERT cannot read the Head Record, an error number will be written onto the output file and execution will stop. For an explanation of CONVERT's error reporting facility see section VIII.

Except for the Head Record, each record in the STUDENT-RECORDS file contains all the information which is stored from one student's execution of a lesson. In order to prepare the STUDENT RECORDS file for use in the CONVERT program, the operator needs to 'UNLOAD' the desired student records while PLANIT is running, equip the resulting file with logical unit number 1 and open the file for reading. These steps are part of the procedure for executing CONVERT.

VI OPERATING THE CONVERT PROGRAM

CONVERT's operating procedure can be described as a series of steps as follows:

1) While PLANIT is running, 'GET' the desired lesson, 'ATTACH' the desired student records and 'UNLOAD' the records into tape format.

2) Equip the UNLOADed file to logical unit 1 and open the device for reading.

3) Equip logical unit 5 to the DECLARE file device and open it for reading.

4) Estimate the space requirement for output, allocate plenty of output space on logical unit 2 and open the device for writing.

5) Link the LDBYTE and SBYTE subroutines to the CONVERT object deck and run as a batch job, one run for each UNLOADed file.

6) Collect the output data base decks in the order in which they are to be prestored for the Query System.

7) Prestore the decks into the Query System and build the data base(s) according to the Query System Manual.

The comments in the source code should be carefully studied before any attempts are made to install CONVERT on the target machine. The comments include an 'ERROR LIST' in the event that one of the following three execution errors occur:

ERROR 1 - The parameter value of VX does not match the VX value which was calculated in PLANIT when the student records were produced. This indicates that the input file read from logical unit 1 may not be the anticipated STUDENT RECORDS file. The user should check to see that logical unit 1 is ready with the correct student records before executing again. VX is a function of BYTWRD and VARENT, two parameters which must be set in the CONVERT source code to match the values which were used to produce the student records. The user should check to be sure that the BYTWRD and VARENT values are identical to those set in PLANIT.

ERROR 2 - The user should check logical unit 1 to be sure it is ready with the correct STUDENT RECORDS file. Error 2 will occur, if the Head Record is not exactly 3 * VX + 1 words in length.

ERROR 3 - The user should check the input file on logical unit 1 to be sure that it is ready with the correct STUDENT RECORDS file. The BYTWRD value which was set in PLANIT must be the same as the value set in the CONVERT source code.

In the event that an error occurs, the error number will be printed on the output unit and execution will stop. The error checking is most effective for spotting the problems which can occur during the first attempt at executing CONVERT, immediately following the installation.

VII INSTALLING CONVERT

Each of the programs which are coded in the FORTRAN META language can be installed on a target machine by following the same technique. Meta language packages include PLANIT, The On-Line Query System, The MIOP Test Program and CONVERT. The CONVERT installation requires the following steps:

- Writing the LDBYTE and SBYTE subroutines or obtaining them from a previously installed version of PLANIT or the Query System on the target machine

- Running the PLANIT Generator Program to produce the CONVERT FORTRAN code.
- Compiling the FORTRAN code and linking the object deck to the LDBYTE and SBYTE subroutines.

The PLANIT Installation Manual provides more information concerning the above steps and the reader is advised to read the following sections before attempting to install CONVERT:

- PART III CHARACTER HANDLING: LDBYTE AND SBYTE
  Sections 13.0 and 14.0
- PART IV PLANIT SYSTEM GENERATION
  Sections 15.0, 16.0, 17.0 and 18.0

The installation of CONVERT on the target machine requires only a subset of the steps needed to install PLANIT. The LDBYTE and SBYTE subroutines which were written for PLANIT can be used for CONVERT with no change, if prepared as directed. CONVERT requires no MIOP subroutine and there is no need for a MERGE file for use in the generation step. The only parameters which need to be set for generation are BYTWRD, VARENT and CRDSZE and these are explained in the listing comments. The source code contains comments which emphasize the statements which need to be examined. There are two 'READ' statements, one 'WRITE' and one 'FORMAT' statement in the source code which should be examined to insure that they qualify as legal FORTRAN statements on the target machine.

Although the input unit 1 is normally considered to be magnetic tape and the output unit 2, (the resulting data base) is described as cards, either or both of them might be a disk file, if the installer so chooses. The following table defines the requirement for the unit numbers:

Logical Unit Number	Logical File
1	Input data file consisting of the PLANIT UNLOAD tape format of the selected set of student records
2	Output data file formatted as 80-character card images consisting of the Query data base version of the student records
5	Input data file consisting of the initialization DECLARE file described in section IV

APPENDIX D -D

THE MIOP INTERFACE TEST PROGRAM

USER'S MANUAL

DD

THE MIOP INTERFACE TEST PROGRAM

USER'S MANUAL

Prepared for:

The United States Army Research Institute

Alexandria, Virginia

Contract No.

DAHC19-76-C-0022

Ron Silberman

Charles Frye

The Northwest Regional Educational Laboratory

Portland, Oregon

December 1977

ABSTRACT

This paper is a description of a program which is design to help those programmers who are attempting to install PLANIT or another PLANIT-like operating system. The program of concern is called the MIOP Interface Test Program and is used to test the MIOP subroutine (a locally-written subroutine necessary for local PLANIT execution). The description herein contains answers to the following questions:

- What is the MIOP Test Program designed to do?
- Where does the use of such a program fall within the sequence of steps which should be used to install PLANIT?
- How is the Test Program installed and used most efficiently?
- What are the functional characteristics of the tests which are administered by the Test Program?

TABLE OF CONTENTS

	Page
INTRODUCTION	1
TESTING THE PROPER OPERATION OF MIOP	2
THE MIOP TEST PROGRAM AS AN AID TO SYSTEM INSTALLATION	3
INSTALLATION OF THE MIOP TEST PROGRAM	5
DESCRIPTION OF THE TESTS	6

THE MIOP INTERFACE TEST PROGRAM

I. INTRODUCTION

This document describes the operation of the MIOP Interface Test Program, a program which exercises the locally-written machine interface subroutine for PLANIT.

Many people have become somewhat familiar with PLANIT by now. PLANIT is a time-sharing system designed for instruction via a computer, and is completely portable such that any qualified system programmer can install it on a local machine by writing three interface subroutines. Two of them, LDBYTE and SBYTE, are quite trivial and offer little challenge. The third, MIOP (Machine Input/Output Program), is somewhat more involved, usually requiring two weeks or more to produce. MIOP's function is to perform all data transfers between PLANIT and the peripheral devices.

What may be less well known is that PLANIT consists of both an author language and a time-shared operating system. In fact, the operating system has been extracted from PLANIT and is being used for new applications. MIOP interfaces that operating system to local hardware so that the exact nature of the application program which uses MIOP will usually be completely transparent to the system programmer who codes it.

It is no surprise that MIOP should be coded in as efficient a manner as possible, and must perform exactly the operations which the PLANIT operating system assumes. Rather than attempting to familiarize the system programmer with PLANIT (or other PLANIT-like application programs), this MIOP Interface Test Program was developed which systematically exercises all of the required interface points between PLANIT's operating system and the peripheral devices. Where possible, this program also performs some checks on the efficiency with which the operations are being executed. The system programmer who is doing the local installation can use this program to check the work without having to know or use PLANIT in order to do it, even checking the MIOP more thoroughly and in much less time than if PLANIT

was used to do it.

Therefore, the installer is encouraged to invest the little extra time that will be required to run this test program in order to greatly simplify the checkout process for the newly written MIOP code.

The same LDBYTE and SBYTE subroutines that are needed for PLANIT are also needed for the test program. If these are not correct, none of the programs will run. However, there is not too great a chance for error and a check of a core dump will quickly verify whether they do in fact work properly. It is up to the installer to make them operate as efficiently as possible. The PLANIT Installation Manual contains a description of each of the subroutines mentioned and the PLANIT tape contains FORTRAN versions of them as well. These are not intended to be used as such, but to help the installer understand the logic and avoid coding errors. The actual coding can almost certainly be done more efficiently in assembly language to minimize execution time.

II. TESTING THE PROPER OPERATION OF MIOP

It is MIOP's duty to access (read, write, open and close) each of the devices which will be needed to run the operating system for PLANIT. In addition, MIOP must be able to retrieve the current date and time, suspend the execution of the operating system during periods of inactivity, and respond to interrupts which are generated by user terminals. The MIOP Test Program simulates the operating system by making a series of calls, each of which checks MIOP's ability to perform the desired operation. If any of the calls results in a failure to perform the designated operation (e.g. failure to open a file, read error, write error, etc.) the Test Program will respond with a comment about the error. The comment will be sent to the operator's terminal, if possible, or the line pringer otherwise. As mentioned in the Test Program's comment cards, the line printer will be used to document all testing progress until the terminal becomes active. If a MIOP failure prevents the MIOP Test Program from communicating to the printer, then the Test Program will proceed to its termination with a "fatal error" call to MIOP and the installer can cause a core dump to display the necessary debugging information. The error number for a "fatal error" will be transmitted to MIOP in the format described in the

PLANIT Installation Manual. The explanation of these error numbers is found in the comments section of the program listing for the MIOP Interface Test Program. Also in the comments will be found the annotation of the first few calls which will be made to MIOP. It is crucial that MIOP is able to execute these calls successfully so that the Test Program can begin to communicate with the operator, first through the line printer and then at the terminal.

Once the MIOP Test Program has completed its battery of tests, it will respond with a "BOX SCORE" list of the operations which were tested and whether or not the tests were successful. Unless the entire sequence of MIOP calls are performed with no problems, the operator may wish to modify the MIOP code and rerun the Test Program. To simplify this process, the operator is given the option, near the beginning of the testing sequence, just after terminal communication has been established, to branch to various entry points within the prescribed sequence of tests. Also, the testing sequence may be stopped at any point by typing the word, "STOP" on the terminal.

III. THE MIOP TEST PROGRAM AS AN AID TO SYSTEM INSTALLATION

MIOP will require direct disk access. To run the tests it is important that space be allocated on disk as intended for PLANIT (probably before running the MIOP Test Program). It is possible to write a MIOP subroutine in such a way that disk space will be allocated as needed in which case no preliminary allocation would be necessary. However, the more common method seems to use a fixed allocation of disk space which is catalogued prior to PLANIT's execution, in which case the installer will need to generate the PLANIT system code in order to obtain disk record size parameters. After choosing appropriate machine parameters and after generating the PLANIT code, the installer can find record sizes for each of the PLANIT disk files within the generated code version of PLANIT's OFILE procedure. If the record size happens to be zero (as is usually the case for file number seven), the corresponding file number will not be used. The MAXRECSZE parameter in the MIOP Test Program listing is to take on the value of the largest PLANIT record size (i.e. the file number six record size). Thus, the installer will need the record size data for PLANIT in order to properly specify this one parameter for the

generation of the MIOP Test Program. (See also the PLANIT Installation Manual, Appendix C). This disk file information is needed to allocate disk space that will be used both by the Test Program and by PLANIT. The same information could also be obtained from the COMMON Map which the PLANIT Generator program will cause to be printed when the PLANIT code is generated.

The following check list is recommended as a guide for the order of events in the installation process:

- Read the PLANIT Installation Manual, noting particularly the IORST array values for each call.
- Read the comments on the system source code listing.
- Write the LDBYTE and SBYTE subroutines.
- Choose parameters for PLANIT's Merge File.
- Generate the Fortran code and the COMMON Map for PLANIT. Note that the PLANIT Generator also uses the same LDBYTE and SBYTE subroutines so they will be correct when the Generator runs properly.
- Obtain the disk file numbers and record sizes for each from the generated OFILE procedure, and choose the number of records that you wish to allocate for each file. Note that some will be fixed by system parameters and some will be optional, depending on how much user space is to be made available.
- Write the MIOP subroutine code.
- Read the comments on the MIOP Interface Test Program listing.
- Choose the Merge File parameters for the MIOP Test Program. Note that most values will be the same as in PLANIT.
- Generate the MIOP Test Program.
- Allocate space on disk according to the file number and record size data acquired earlier.

- Put the MIOP Test Program's Cards File in place so it can be read.
- Compile and run the MIOP Test Program, using it to debug MIOP.
- It may be necessary to generate PLANIT again, this time implementing the correct overlay structure. If changes to any parameters are necessary, the disk record sizes may change requiring a reallocation of disk. This should not affect the MIOP though unless the record sizes are used as literal numbers in the MIOP code. It would be better not to have them show up as literals in the MIOP code for that reason.
- Compile the generated PLANIT code, linking it with LDBYTE, SBYTE and MIOP.
- Make any necessary changes in PLANIT's Cards File and put it in place to be read by the PLANIT program.
- Run PLANIT.

IV. INSTALLATION OF THE MIOP TEST PROGRAM

Installation of the MIOP Interface Test Program should proceed smoothly after the installer has generated the PLANIT system. As was mentioned before, the Test Program installs exactly like PLANIT, using the same generation process and of course, the same three subroutines, LDBYTE, SBYTE and MIOP. However, the Test Program will have fewer installation parameters to set than PLANIT, and will usually not require the overlay logic since the entire program should normally fit into core. Thus, the Merge File for the Test Program is very brief, and any questions about particular parameter values can usually be resolved by referring to the corresponding parameters in PLANIT, noting their description in the PLANIT source code listing.

V. DESCRIPTION OF THE TESTS

Throughout the sequence of MIOP tests, the Test Program will check to see that the IORST array values which have been set for MIOP's use have not been altered during the call. Then the status word is checked to confirm MIOP's report of its work, and the user is notified of any problems, which might be occasioned either by MIOP's report of an error or by an internal evaluation of the results which were expected from the call. The sequence of tests which are administered to MIOP is as follows:

1. Open the line printer.
2. Open the Cards File.
3. Read the first card.
4. Print a copy of the first card on the line printer. The operator should visually confirm that the printer has executed a page eject just prior to printing this line so that the line appears just below the perforation in the paper.
5. Print the message, "1ST CARD" on the line printer. The operator should check for any misprints. This will be enough to confirm that the data on the remainder of the cards will read properly.
6. Read and process each of the remaining cards in the Cards File.
7. Close the Cards File.
8. Print a comment (on the line printer) regarding the closing of the Cards File and the operations that will take place next.
9. Open the operator's terminal.
10. Print a comment (on the line printer) regarding the opening of the operator's terminal.

11. Read the operator's terminal. The operator is asked to type the word, "HELLO", and then to confirm whether the text appeared at the terminal properly. These first eleven steps are critical in establishing communication with the operator so that further testing can be done on an interactive basis. Any failures to this point will be documented on the line printer. Note that in all cases where a YES or NO response is requested, a "Y" or "N" abbreviation may be used.

12. Send two or more buffers of text to the terminal. This will test the proper use of carriage control characters on the terminal.

13. Next is the carriage return test. This test will only be made if the "line feed" and "carriage return" are distinct characters. It will test a carriage return without a line feed.

14. The enforced time limit test is part of the question which asks whether the enforced timed terminal read has been implemented. If not, answer "NO" and proceed. Otherwise the operator will be timed at 20 seconds for a response. There will be three opportunities to experiment with this test, so an answer can be given before the 20 seconds have expired on one try and then the operator can wait and time the 20-second interval on another try. Any response submitted before the 20 seconds have expired should be accepted immediately.

15. The terminal writing speed test sends each letter of the alphabet separately, one letter per MIOP terminal write call, along with a question to the terminal. The operator will probably notice a slower typing speed on the alphabet string than on other messages. However, if the speed approaches one character per second, chances are that MIOP's data handling of messages to the terminal will adversely affect the apparent response time of PLANIT.

16. The blank line test pertains to PLANIT's need to detect blank lines. Since trailing blanks are to have been eliminated from the character count of all incoming terminal reads to PLANIT, a line in which all typed characters are blanks must be the same as the line where no key other than the carriage return is pressed. It would be well to try both variations. One form can be used for the response to this test. Then, a blank line can be given in response to any of the following tests. If the line is recognized as blank, the Test Program will just ask for the answer again.

17. The Test Program will attempt to retrieve the time and date from a MIOP call. These will be printed on the terminal for the operator to check visually. No additional comments will be made. One variation allows the setting of the date to be deferred until PLANIT is running. In that case, the date printed in this test may be incorrect.

18. The system WAIT test is next. Since the MIOP Test Program is attempting to simulate PLANIT, it is important that its execution be suspended while the program is inactive (waiting for a user input), so that CPU time is given to other programs. This is tested by reporting to the terminal the count of SYSTEM WAIT calls and the elapsed time of the longest one. A WAIT call will occur for each batch of text that goes to the terminal plus one more when the terminal is idle. To get the most value from this test, delay your response and time the delay. About 30-40 seconds should be sufficient. The time reported should correspond to your observation. The number of WAIT calls could vary up to a half dozen or so. However, if execution is not being suspended, the count would probably be large, indicating that looping is taking place and the WAIT call is being made repeatedly. Many machines have a "system wait" light on the console which can also be checked during these intervals. Note that the time test (in no. 17, above) must be correct for this and future timed tests to be reliable.

19. The line printer is now closed.

20. Next, the operator is asked to enter the number of records and record sizes for each of the disk files. The record sizes can be obtained from the PLANIT OFILE procedure (the generated FORTRAN version), or from the Generator COMMON Map by using the listing comments to show the data items and arrays belonging to each file. The number of records given in the reply can be less than the actual number which have been allocated on disk, but cannot be more. The actual number of records will also need to be shown in PLANIT's Cards File, on the "IFBTN" card. If the numbers for that card have been chosen and the corresponding records allocated on disk, then a legal reply would be to reproduce that card, character by character.

    The number of records (e.g. the "IFBTN" card) will be requested first, then the record sizes will be requested for all files whose number of records were non-zero. It is important that none of the record sizes exceed the value given for the "MAXRECSZE" parameter, and that the file number six record size exactly equal it.

21. Each of the disk files (containing non-zero record counts) is opened, a message on the terminal confirms the operations.

22. The operator is now given a choice of testing all records of each file or a sample of not more than three records in each file. The sample should be enough to confirm MIOP's work and will run faster. However, it would be better if a little extra processing time can be tolerated, to check all records since this will also confirm whether all records are properly allocated and that no disk parity errors are found.

23. Each file will be tested in order, testing its records one at a time up to the number that are to be checked. The checks include writing and reading distinguishable data, confirming direct access of the right data from a given record number, looking for overwriting and underwriting, and verifying the return status number. If any

errors are found, a message describing the error along with the file and record number where it occurred, will be reported on the terminal. If more than five errors of any kind are detected, the disk check will be terminated. If no errors are encountered, the successful write and read of the first record of each file will be reported on the terminal. This will enable the operator to follow the progress of the test. Records beyond the first will not be reported (unless errors are encountered) because of the amount of terminal output which would be generated.

24. If the MIOP Test Program has been generated for more than one user and if the "MAXRECSZE" parameter is equal to the file number six record size and if no errors were encountered while checking file six records, then time-sharing may be tested. The operator (or some assistants) may begin interacting at other terminals. The test scenario will continue only on the operator's terminal. The remaining terminals will simply acknowledge inputs by reporting back the number of characters in the input.

25. Next, the operator will be asked if magnetic tape support has been implemented in MIOP. If so, a request will go to MIOP for a tape to be mounted. This should be a scratch tape. If not implemented, the tape test will be skipped.

26. After the mount has been confirmed, the tape unit will be opened.

27. Now, five records will be written on tape. Arbitrary record sizes have been chosen for each of the five records of 4, 16, 64, 256 and 1024 words.

28. A call will be made for write end-of-file, close and rewind.

29. The tape will be re-mounted, re-opened and five records will be read. The check includes confirming the data from each record, checking overwrites, underwrites and the presence of the end-of-file mark after the fifth record. Also the MIOP status reports will be checked. The results will be reported to the terminal.

30. The tape unit will be closed (read-type) and rewound. (The rewind operation is implied in each close call, for reading or writing).

31. Next, the operator is asked whether the card punch and read operations can be tested, and, if so, to ready the card punch before typing the reply. If not, the Test Program skips to the end where the "BOX SCORE" is printed.

32. The card punch unit is opened, and the status result is tested.

33. One card is punched. The characters on the card are similar to the first line on the line printer.

34. The card punch unit is closed.

35. Now, the operator is given time to position the punched card in the card reader unit. A question at the terminal asks if the card is ready to be read. The card should be in position before replying. However, if the reply to the question is "NO", the remainder of the test will be skipped.

36. The card reader unit is opened and checked.

37. The card is read and the data compared to that which was punched. (Note that the card can also be interpreted for visual checking.)

38. The card reader unit is closed.

39. A "BOX SCORE" summary of test results for those features which were tested will be printed. When the BOX SCORE proclaims all tests to be satisfactory, MIOP is ready to be linked with PLANIT.

40. All remaining opened files are closed and the Test Program is stopped.

Having successfully completed all of the tests, the installer may be reasonably confident that the MIOP coding is correct. Unfortunately, though, this does not confirm that it is as efficient as it might be, but at least it will run so that PLANIT will run and efficiency questions can then be explored.

APPENDIX E

DEMONSTRATION DATA BASE

DEMONSTRATION DATA BASE

The following pages contain a description and listing of the Demonstration Data Base, GOBBA (Ground Order Of Battle). The data base is described both in the chart on the next page and in the data base attribute name declarations on the final page of the data base listing.

The listing assumes 80-column cards as the input medium, where each group of three lines (terminated by a '#' character) constitutes a 240-column (3 x 80) entry into the data base.

The description of the syntax for the DECLARE and DEFINE statements can be found in the Portable Query System User's Manual.

Primary Name	Descriptive Phrase	ASSIST Synonym	Collective Name	Additional Description	Character Field Length
UNAME	Unit Name	UNAME			19
CNAME	Code Name	HNAME	TNAME	Total Name	12
CNO	Code Number	----			6
PUNAME	Parent Unit Name	PUNIT			11
PUCMDR	Parent Unit Commander Name	----			13
PUCMDRRNK	Parent Unit Commander Rank	----		G/A = General of Army G/A's = General of Armies	5
UCMDR	Unit Commander Name	----		G/D = General of Division G/B = General of Brigade	15
UCMDRRNK	Unit Commander Rank	----		G/Air = General of Air COL = Colonel	5
ULOCAT	Unit Location	LONAM			46
NOOFF	Number of Officers	PEROF			4
NOEM	Number of Enlisted Men	PEREM	PERSONNEL		5
NOTL	Total Number of Military Personnel	PERTL			5
MORALE	Rating for Morale	DEPEN1		1 = High 2 = Moderate	1
DISCIPLINE	Rating for Discipline	DEPEN2		3 = Low	1
POLITREL	Rating for Political Reliability	DEPEN3	URATINGS	Unit Ratings	1
EFFIC	Rating for Officer/NCO Efficiency	DEPEN4		1 = High 2 = Moderate	1
CBTEFFECT	Rating for Combat Effectiveness	DEPEN5		3 = Low	1
CBTRED	Combat Readiness Category	CRCAT		Scale 1-3 (where 1 = Combat Ready)	1
TIME	Time to Reach Combat Readiness	CATO1	TCBTRED		4
REASON	Reason Unit Not Combat Ready	REAS1		Personnel or equipment understrength	46

```
DATA-BASE GOBBA #
      13 CAA                KLIENTO      940154ST LAURENCOKOWALSKI      G/A'S
      ZITNIKOV        G/A  NE CANADA
        70  675  745121111                                          #
      48 CAA                             804741ST LAURENCOKOWALSKI      G/A'S
      TRIMPER,L       G/A  NEW ENGLAND NEW YORK
        81  796  877221221                                          #
      21 CAA                AFANADORO    107701ST LAURENCOKOWALSKI      G/A'S
      NAZZRHULLAH     G/A  NEW ENGLAND NEW YORK
        72  734  806111111                                          #
      16 TK A               INFERULARO   741088ST LAURENCOKOWALSKI      G/A'S
      GRIMALDI,G      G/A  NEW YORK
      2754227252547911111 1                                         #
      15 AIR A                           628808ST LAURENCOKOWALSKI      G/A'S
                            NE CANADA NEW ENGLAND
       670 6005 6675111221                                          #
      67 ABN DIV            KAPTILO      243567ST LAURENCOKOWALSKI      G/A'S
                            MAINE
       855 8212 906732100212WK60% ASSIGNED PERSONNEL                #
      53 ARTY DIV           ATESTI       117481ST LAURENCOKOWALSKI      G/A'S
                            NEW ENGLAND CANADA
       463 4417 488012111215WK70% ASSIGNED PERSONNEL                #
      14 SSM DIV                         621113ST LAURENCOKOWALSKI      G/A'S
      ARCOVITO,SV     G/D  NEW ENGLAND
       495 4315 4810111111                                          #
      24 SSM DIV            KAPO         542449ST LAURENCOKOWALSKI      G/A'S
                            NEW ENGLAND
       463 4010 4473111001                                          #
      41 SAM BDE            DUONALFO     309475ST LAURENCOKOWALSKI      G/A'S
      BATALOV,B       G/B  CANADA
       183 2019 2202111111                                          #
      72 SAM BDE                         512345ST LAURENCOKOWALSKI      G/A'S
                            CANADA
       178 1928 2106111001                                          #
      15 ENG CONST REGT                  691703ST LAURENCOKOWALSKI      G/A'S
                            NEW ENGLAND
       107  798  905000001                                          #
      85 ENG AMPH REGT      DRATUR       083458ST LAURENCOKOWALSKI      G/A'S
                            NEW ENGLAND
        90  901  991000001                                          #
      24 SIG REGT                        277674ST LAURENCOKOWALSKI      G/A'S
      PUTTENHAM       COL  NEW ENGLAND
       317 1687 2004000001                                          #
      62 MT REGT                         130251ST LAURENCOKOWALSKI      G/A'S
                            NEW ENGLAND
```

```
 166 1395 1561000001                                                    #
 5 CAA                KOLOHIO      505721GOLFO       O'ROURKE,TJ  G/A'S
SOVAK,JJ         G/A  TEXAS OKLAHOMA
  71  721  792111211                                                   #
 6 CAA                FENDIGA      600731GOLFO       O'ROURKE,TJ  G/A'S
FERLINGHETTI,L G/A  LOUISIANA
  69  673  742111111                                                   #
 1 TK A               AKOMPANI     802173GOLFO       O'ROURKE,TJ  G/A'S
NEBBIA,TC        G/A  TEXAS OKLAHOMA
2559230212558011111                                                    #
 2 TK A               ANGULO       200712GOLFO       O'ROURKE,TJ  G/A'S
ULIANOV,NP       G/A  TEXAS LOUSIANA
2760240982685832111221WK75% ASSIGNED TANKS                             #
 2 AIR A              RETOFORMA    108756GOLFO       O'ROURKE,TJ  G/A'S
RADESCU,SC       G/AIRTEXAS
 655 5987 642111111                                                    #
 1 ABN DIV            TASO         325707GOLFO       O'ROURKE,TJ  G/A'S
YUAN,HJ          G/D  TEXAS
 861 8175 9036112222 9WK55% ASSIGNED PERSONNEL                         #
 2 ARTY DIV           FLUEGO       200613GOLFO       O'ROURKE,TJ  G/A'S
ODONEZ,JS        G/D  TEXAS OKLAHOMA
 478 4392 4870211111                                                   #
55 SSM DIV            ABRAMUNITO   951605GOLFO       O'ROURKE,TJ  G/A'S
JEN,WS           G/D  TEXAS
 479 4165 4644111111                                                   #
 1 SAM BDE            DRAGON       805804GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
 179 2020 2199111001                                                   #
 3 SAM BDE            TROMPETO     921313GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
 177 1840 2017111111                                                   #
 2 CML BDE                         604780GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
 145 1801 1946111111                                                   #
 9 ENG AMPH REGT                   950435GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
 103  805  908111001                                                   #
 2 SIG REGT                        455477GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
 301 1683 1984000001                                                   #
10 MT REGT            TESTUDO      317325GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
 163 1366 1529000001                                                   #
 8 MED REGT                        869959GOLFO       O'ROURKE,TJ  G/A'S
                       TEXAS
```

```
  66  291  357000001                                                  #
 7 INTEL REGT                      358418GOLFO      O'ROURKE,TJ  G/A'S
                    TEXAS
  65  278  343000001                                                  #
 2 PGND BN                         853018GOLFO      O'ROURKE,TJ  G/A'S
                    TEXAS
  98  326  424000001                                                  #
 1 CAA              ORELO          105701KARIBO     MIYAGISHIMA,TG/A'S
SANCHEZ,F      G/A  NORTH CAROLINA
  76  728  804000001                                                  #
 3 CAA              EGECO          305711KARIBO     MIYAGISHIMA,TG/A'S
GULLSTRAND,SL  G/A  SOUTH CAROLINA GEORGIA
  79  750  829000001                                                  #
12 CAA                            201711KARIBO     MIYAGISHIMA,TG/A'S
SCHUSCHNIGG    G/A  SOUTH CAROLINA FLORIDA
  69  713  782111222216WK75% ASSIGNED PERSONNEL                       #
 9 AIR A            VINERO         540661KARIBO     MIYAGISHIMA,TG/A'S
POWELL,JF      G/AIRCUBA ANTILLES SANTA DOMIGO US MID ATLANTIC ST
 668 6003 6671111111                                                  #
 4 ABN DIV          FURIOZEGA      400625KARIBO     MIYAGISHIMA,TG/A'S
                    CUBA
 805 8201 9006222231                                                  #
23 ABN DIV          KALAFO         903916KARIBO     MIYAGISHIMA,TG/A'S
VONMOHL,TH     G/D  CUBA
 890 8376 9266221221                                                  #
12 ARTY DIV         AKSETO         266291KARIBO     MIYAGISHIMA,TG/A'S
                    US MID ATLANTIC ST
 485 4192 467111111213WK65% ASSIGNED PERSONNEL                        #
 8 F SSM DIV        PARO           373014KARIBO     MIYAGISHIMA,TG/A'S
                    US MID ATLANTIC ST
 485 4067 4552111001                                                  #
 8 SAM BDE          ARGENTO        713345KARIBO     MIYAGISHIMA,TG/A'S
                    US MID ATLANTIC ST
 184 2020 2204111111                                                  #
11 CML BDE          ARKO           728771KARIBO     MIYAGISHIMA,TG/A'S
FODOR,WJ       COL  NORTH CAROLINA
 139 1785 1924111111                                                  #
22 ENG AMPH REGT                  572163KARIBO     MIYAGISHIMA,TG/A'S
                    US MID ATLANTIC ST
  87  850  937000001                                                  #
18 SIG REGT         ANSIS          614592KARIBO     MIYAGISHIMA,TG/A'S
                    US MID ATLANTIC ST
 320 1701 2021000001                                                  #
75 ENG AMPH REGT                  173745KARIBO     MIYAGISHIMA,TG/A'S
                    US MID ATLANTIC ST
```

```
        85  840  925000001                                    #
#
DECLARE UNAME STR(7,25) #
DECLARE CNAME STR(26,37) #
DECLARE CNO NUM(38,43) #
DECLARE PUNAME STR(44,54) #
DECLARE PUCMDR STR(55,67) #
DECLARE PUCMDRRNK STR(68,72) #
DECLARE NOOFF NUM(167,170) #
DECLARE UCMDR STR(87,101) #
DECLARE UCMDRRNK STR(102,106) #
DECLARE ULOCAT STR(107,152) #
DECLARE NOEM NUM(171,175) #
DECLARE NOTL NUM(176,180) #
DECLARE MORALE NUM(181,181) #
DECLARE DISCIPLINE NUM(182,182) #
DECLARE POLITREL NUM(183,183) #
DECLARE EFFIC NUM(184,184) #
DECLARE CBTEFFECT NUM(185,185) #
DECLARE CBTRED NUM(186,186) #
DECLARE TIME STR(187,190) #
DECLARE REASON STR(191,215) #
DECLARE TNAME STR(7,43) #
DECLARE PERSONNEL STR(167,180) #
DECLARE URATINGS STR(181,185) #
DECLARE TCBTRED STR(186,215) #
DEFINE HNAME=CNAME #
DEFINE PUNIT=PUNAME #
DEFINE LONAM=ULOCAT #
DEFINE PEROF=NOOFF #
DEFINE PEREM=NOEM #
DEFINE PERTL=NOTL #
DEFINE DEPEN1=MORALE #
DEFINE DEPEN2=DISCIPLINE #
DEFINE DEPEN3=POLITREL #
DEFINE DEPEN4=EFFIC #
DEFINE DEPEN5=CBTEFFECT #
DEFINE CRCAT=CBTRED #
DEFINE CAT01=TIME #
DEFINE REAS1=REASON #
$$
```