Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA ILLINOIS 61801

CAC Document Number 186
CCTC-WAD Document Number 6502

Humanizing Data Management Systems: An Intelligent Terminal Approach

by

Deborah Sue Brown

January 1976

CAC Document No. 186
CCTC-Wad Document No. 6502

HUMANIZING DATA MANAGEMENT SYSTEMS: AN INTELLIGENT TERMINAL APPROACH

by

Deborah Sue Brown

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

January 1976

This work was supported in part by the Command and Control Technical Center, WWMCCS ADP Directorate of the Defense Communications Agency under contract No. DCA100-75-C-0021 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1976.

Approved for release Peter A. Alsberg
Peter A. Alsberg, Principal ~~Investig~~ator

ACKNOWLEDGMENTS

Many people were helpful in the preparation of this thesis. I would like to acknowledge the efforts of John Mullen, Dave Willcox, Dave Healy, Jim Gast, and Steve Bunch in the design and building of the original terminal system. Also, a mai-tai goes to Jim Bailey, who kept the hardware running so we had a machine on which to develop the system. John Mullen and Peter Alsberg, my thesis advisor, deserve a special thanks. These two gave me much helpful advice on the writing of the thesis, including several proofreadings each. Additionally, Peter devoted several hours to preparing the pictures used. Last, but hardly least, a special thanks goes to Shirley Brown and Pam Hibdon for their typing of this thesis.

Thanks to everyone.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Computer systems are often difficult and awkward to use. This is no less true of data management systems (DMS), which are designed for use largely by non-computer scientists. Problems typically encountered by DMS users include: awkward command syntax; communication through artificial jargon; the need to go through one or more computer professionals for help in formulating a request; confusing error messages; and slow response time. Problems such as these are sufficiently common that many computer users simply learn to tolerate them rather than insisting on more usable human interfaces.

This thesis documents one attempt at an improved interface. An intelligent terminal was built to act as a front-end to an existing data management system. This terminal system approached several of the problems mentioned above. It attempted to give the user rapid response or explanatory messages when the response time was slow. It attempted to eliminate the confusion resulting from artificially imposed computer jargon by tailoring the front-end system to one particular application and by using the language of that application. Also, user hand-holding and feedback was built into the system to minimize training required to use the terminal.

The completed system was demonstrated to both system programmers and DMS users. The response was very favorable. The observers not only were impressed with the ease of use of the demonstration DMS, but also were interested in seeing what further extensions of these ideas might accomplish.

2. PROBLEMS ADDRESSED

The primary thrust of this project was to create a better human interface to an operational interactive data management system. The use of an interactive system can in itself minimize some of the problems mentioned above, e.g. excessively long response times. However many other problems remain to be approached by the terminal front-end system.

The problems addressed by the terminal system fall into three general areas:

1. the language of the system,
2. the method of input, and
3. reliability.

System language. Data management systems tend to be general in design, so that different applications can use the same DMS. Although this generality may make the DMS more widely applicable, it can make any specific user's job more difficult. The user, already familiar with the jargon of a specific application, is forced to learn a new, relatively artificial, machine-oriented jargon in order to communicate with the DMS. This learning process may require a training period of weeks or months. Then, since the language is artificial to the user, facility with the DMS can be quickly lost if the user does not have frequent contact with the system. Once familiarity is lost, the user will tend to make more syntactical mistakes. Often the resulting error messages are not particularily helpful, and may cause increased frustration and confusion for the user.

Input mechanism. Most computer terminals use some type of keyboard-like mechanism for user input. For non-typists and inexperienced users in general, this in itself can be a slow, awkward, and particularly frustrating procedure.

Reliability. Under ordinary circumstances, a user must rely on the availabity of a host computer in order to retrieve information from a data base. This presents three types of problems. First, processing time on the host may be quite expensive. Second, communication facilities between the terminal and the host may be slow or expensive or both. And third, if the host is unavailable either because the computer itself is down or because the communication facility is down, the user is unable to get any information from the data base.

The combination of these three aspects of most data management systems can make these systems very frustrating to many users.

3. PHILOSOPHY OF SOLUTIONS

Before describing the specific system which was built, a discussion of the general types of solutions used is in order. The final approach was based on five primary considerations:

1. The system must not require any modifications to the existing DMS.
2. A more easily usable command syntax and input mechanism than is currently available was desired.
3. The local processing and memory capabilities of the terminal should be used to maximize the user feedback and to minimize the load on the remote host.
4. The front-end system should be able to survive host system crashes.
5. The solution had to be feasible under currently available technology.

DMS interface. Since the DMS front-end could not make any changes in the host system, the terminal was designed to interact with the DMS as a normal user. Communication with the DMS would use the standard human conventions for that system and would be generated by the front-end. DMS output would be formatted by the front-end for communication with the user.

Ease of communication. The second consideration was approached from three different directions. First, the front-end was custom tailored to one particular application. This allows the creation of a syntax in a familiar jargon which is more meaningful to the user than the general purpose approach of the DMS. The second angle of attack was to use this

jargon in presenting the user with a menu of all possible choices at every step of a command. This relieves the user of the need to memorize the syntactical form of commands, and so decreases the amount of training required to use the system. Finally a touch panel rather than a keyboard was used as the primary form of user input. Using this panel, a command is selected by touching "buttons" which are displayed on the terminal screen. These buttons present all the valid options at every step. The need to remember syntactical forms and the need to type are both eliminated. Specifying a command becomes simply a matter of selecting a series of buttons on the screen. Additionally, this approach eliminates for the user the frustration of making syntactical mistakes, since they are impossible.

Local capabilities. The local memory and processing capabilities of the terminal are utilized in many ways. Primarily they serve as a local cache for data items and handle most of the user interaction with the system. As each data item is retrieved from the host for display, the item is also stored in local memory. When that item is requested later, it is available without accessing the host. This cache technique reduces the use of the communication facility and the load on the host. The local processing power is used to manage the cache, to create all displays of data items, to handle the menu selection described above, and to provide feedback to the user as to what the system is doing. The local generation of displays not only allows for the displays to appear quickly, but also for an item to be quickly redisplayed in a different format. This saves processing time on the host and reduces the load on the communication facility. Also, the local processing power allows quick feedback, e.g. indicating which button was touched or displaying messages about the state of an attempted item retrieval.

Survivability. The existence of a mini-computer in the terminal allows the front-end terminal system to proceed when the host is unavailable. When the host is down, the user may have a limited set of available commands, such as being able to display only data items which are in the terminal. Within these restrictions, the user can continue to work without the main host system.

Technology. The terminal was built from commercially available hardware and required no modifications to that hardware.

In addition to the stated front-end system goals, the approach used has two other advantages. First, the compound system of intelligent terminal and DMS appears more reliable to the user than the DMS alone. The terminal crashes less frequently than the large system. It is a basically simpler device with fewer components, connectors, etc. In addition, it can mask a host crash. As a result, the amount of time the system is unavailable to the user is decreased by using the intelligent terminal. Second, the compound system can actually be cheaper to operate than using a standard terminal to access the DMS. Initially the intelligent terminal itself is more expensive than a standard terminal. However, the intelligent terminal reduces the load on the host computer by decreasing the number of item retrievals done and by doing all display generation. The money saved by reducing the amount of time purchased on the remote host can make up for the extra cost of the smarter terminal. (See [3] for a discussion of a similar mechanism studied at Harvard University.)

4. IMPLEMENTATION APPROACH

A system was built in order to test the theories outlined in Chapter 3. It was intended merely as a demonstration vehicle to study these philosophies and to get feedback from users on what was good, on what was not so good, and on possible further improvements. In no way was this system intended to be anything other than experimental.

The discussion of this DMS front-end system will include descriptions of the hardware comprising the terminal itself, the user input mechanism, the data bases used, and the terminal actions from the user's point of view. After this overview has been given, the front-end system will be explained in more detail.

Terminal hardware. The terminal itself consisted of a Digital Equipment Corporation PDP-11/10 with 20K words of memory, a high speed plasma panel, a touch panel, and a modem for phone line communication (figure 1).

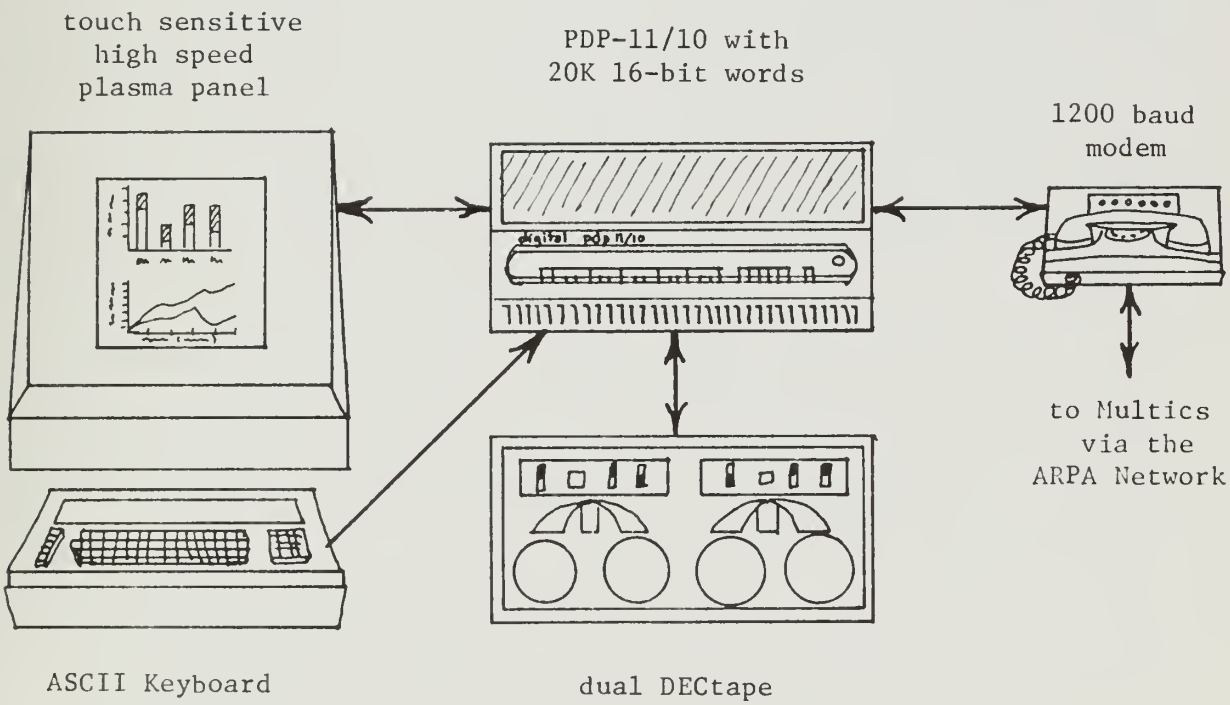

Figure 1

Intelligent Terminal Hardware

Also in the terminal, but of little logical importance, were a DECtape drive and a keyboard. The tape drive was only used for initially loading the system software into the PDP-11, and the keyboard was used only to log in to the remote host.

The plasma panel used is roughly similar to a CRT but has a flat screen and presents a flicker-free display [1]. The high speed parallel panel differs from a standard plasma panel in that a mask of sixteen dots can be written in one panel access. The high speed of this screen allows some feedback mechanisms which would not be effective on a slower device.

The touch panel consists of a square frame which fits on the front of the plasma panel, like a picture frame. The touch panel uses light emitting diodes and light sensing phototransistors to create a grid of infrared light beams approximately one quarter inch in front of the screen. As a pair of intersecting beams are broken, the panel coordinates of the touch can be read, similar to ordinary input devices.

Buttons. When user input is needed, rectangles labeled with the possible choices are displayed on the screen. These rectangles or "buttons" are positioned to be directly behind one or more touch intersections. If the user touches the screen where a button is displayed, the corresponding touch beams in front of the screen will be broken. This will allow the terminal software to ascertain which button was touched. Thus the primary form of input consists of touching a series of programatically displayed buttons.

Data Base. The terminal front end system allows access to a subset of a data base currently existing on the MIT Multics system. This data base contains the Illinois Socio-Economic Indicators for Rural Development. It resides in the Janus DMS which is a subsystem of the

Consistent System. The Consistent System is a large general purpose data management and analysis system [2]. A subset rather than the entire data base was used for the demonstration system to eliminate some problems which seemed extraneous to the topics being studied. For example, the number and size of data items were limited so that all displays could fit on one screen, and so that the amount of local storage required would not be excessive.

The data base subset used for the demonstration contained thirty data bases: one containing information on the northern twenty-nine counties of Illinois and one for each of these counties, containing more detailed information. The elements of the Illinois data base refer to counties. The elements of each county data base refer to towns with populations of greater than 2500.

Each of the data bases is essentially a matrix of information. The rows of the matrix contain information about a specific county or town in the data base. The columns contain information about a specific attribute of each county or town. The information in the data base can be referenced by column. These column vectors are referred to as "items." An item of information contains all the values of a specific attribute, such as population, of the counties or towns in the data base. In keeping with the space limitations mentioned above, the Illinois data base was restricted to twenty-nine items for each of twenty-nine counties. The county data bases were restricted to seventeen items for up to five towns.

The local memory in the terminal was managed so that up to sixteen Illinois data items and one entire county data base can be resident in the terminal at once.

Front-end system. The primary function performed by the terminal system is to display data items. To see an item, the user must specify the data base of interest, the item within that data base which is to be displayed, and the format of the display. At this point, the front-end system will formulate the request to the host for the retrieval of the data item. The retrieved item values are converted into the local format and stored in the cache memory. Finally the item is displayed in the indicated format. If the requested item was already resident in the terminal, the local copy is used for the display. In addition to retrieving and displaying data, the terminal system also provides extra facilities such as printing an explanatory paragraph about any command with which the user needs help. This system does not provide any facility for creating or updating data items. It will, of course, continue running if the remote host goes down.

5. EXTERNAL SYSTEM VIEW

A general understanding of the front-end system can be gained by observing the terminal actions from the point of view of the user. Basically the user is presented with successive screens or "pages" of buttons. Each page asks the user to specify which of the valid options is desired. After all necessary options have been specified, the requested command is executed.

Each page of buttons specifying options of a command has special "Proceed" and "Cancel" buttons. After the desired options have been selected, the user must touch the Proceed button before execution will continue. This allows the user to correct extraneous touches and to verify that the desired options have been selected before continuing. If at any time the user wishes to abort a command, touching the Cancel button will return the system to the front page of buttons.

The front page (figure 2) presents five commands:

1. Help,
2. Select Data Base,
3. Display Data,
4. Manual Override, and
5. Halt.

When the user selects one of these commands, the front page is replaced by other pages presenting options relevant to the selected command. These five commands are explained more fully below.

5.1 Help

The user is presented with a page containing a button for each command (figure 3). The user touches the button for the command which

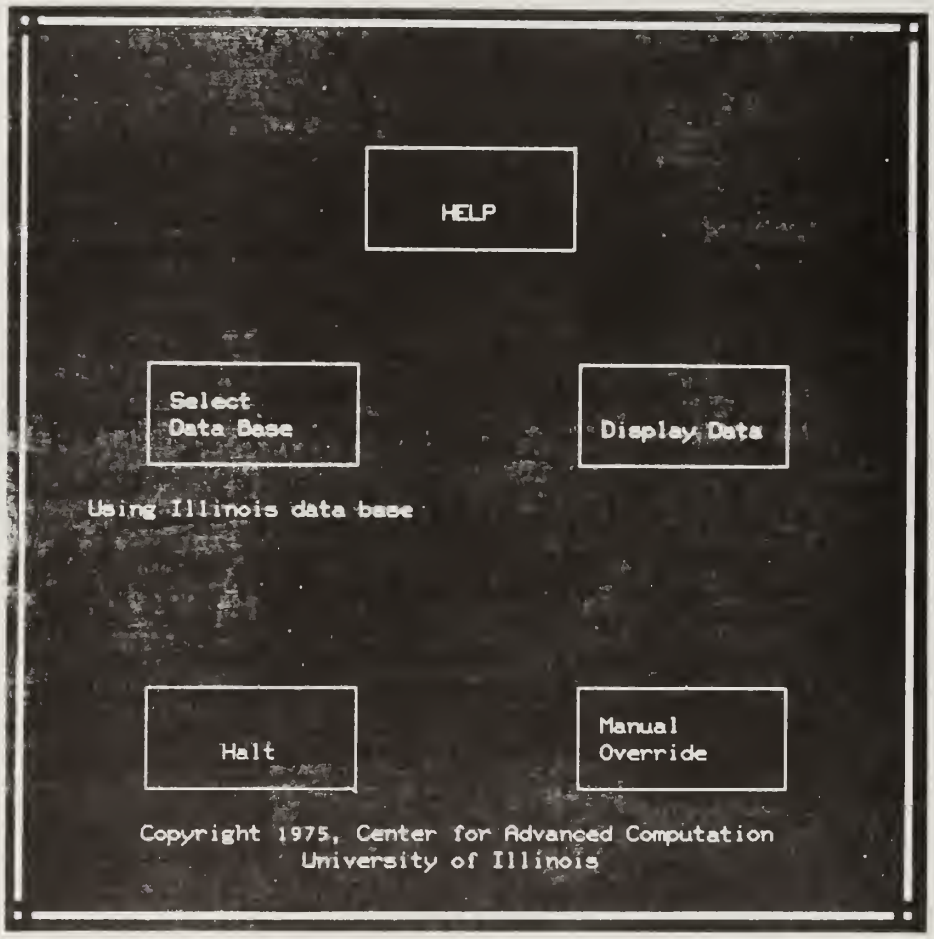

Figure 2

Front Page

Figure 3

Help Page

needs explanation. A paragraph is displayed which describes the function and use of the indicated command. Due to local memory space restrictions, these paragraphs are stored on the remote host. Thus, no help is available if the host is down. In that case the Help button does not appear on the front page.

5.2 Select Data Base

Commands such as Display Data automatically refer to the current data base. The Select Data Base command allows the user to

change the current data base. The first option which must be specified is whether the current level of interest is to be the Illinois data base or some county data base (figure 4). If the user is interested in the entire state, the change is made, and the system returns to the front page. Otherwise, the user must specify which county is desired (figure 5) before the system can complete the change.

If the user sets the current level of interest to be a county, the system will attempt to make the local copy of that data base as complete as possible. If all the items for the data base are already

Figure 4

Select Area of Interest Page

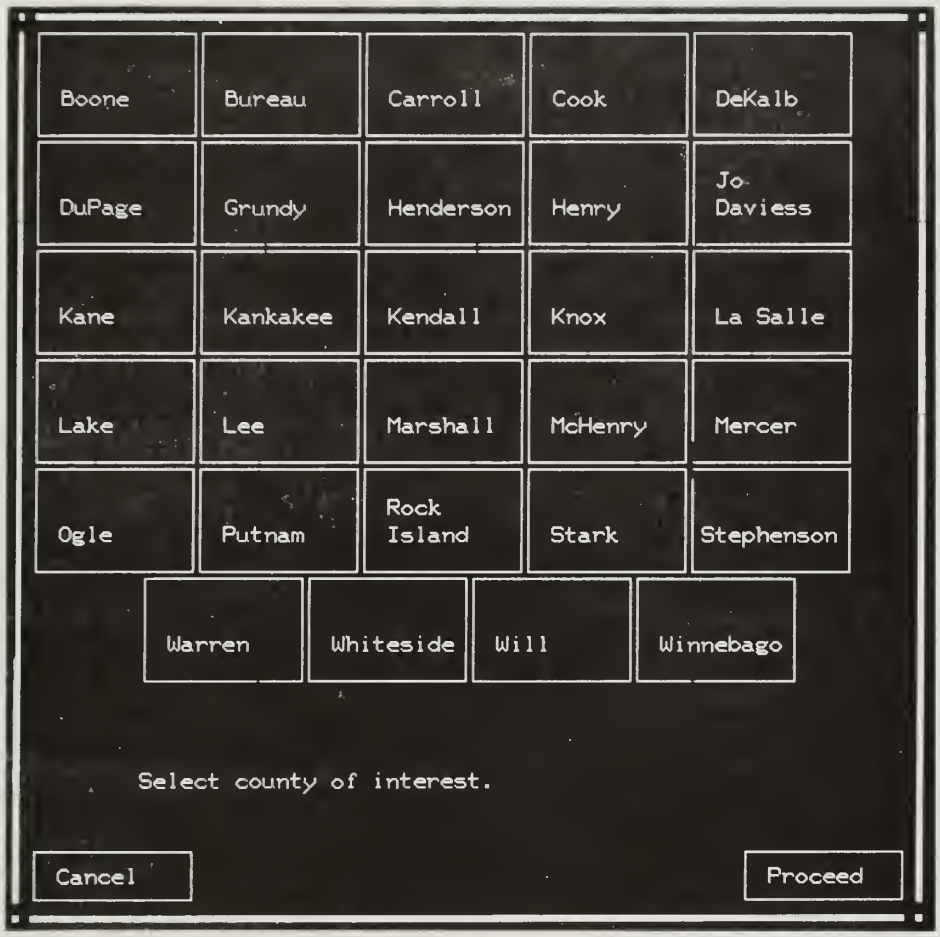

Figure 5

Select County Page

locally available, no more action is taken. However, an attempt will be made to retrieve any items which are not resident. This is done so that the user will get immediate response on future requests for item displays. The mass retrieval of county items is possible because the system has local storage space for an entire county data base and because the county items are so small that the total delay due to transmitting the data is short.

If the host system is unavailable, changing the current data base may not be feasible. In particular at least part of both a county

data base and the Illinois data base must be in the terminal before changing the current data base is a valid option. So, if the host is not available, and if only one data base is locally available, the Select Data Base button does not appear on the front page. Even if a county data base and the Illinois data base are locally available, the user has no choice as to which county data base may be selected. In this case, the page of buttons of counties (figure 5) is omitted if the user selects a county data base, and the system automatically selects the locally stored county data base.

5.3 Display Data

This command displayed up to three data items in a choice of formats. The first option which the user must specify is which data items are to be displayed (figure 6). If two or three are chosen then the display will be a table. If only one item is chosen, the user must specify whether it is to be displayed as a table or bar chart or, if the current level of interest is the state, a shaded map (figure 7). At this point the terminal system will attempt to retrieve those data items which are not locally available. As each retrieval is initiated, an explanatory message is printed to the user. If some retrievals can not be completed, a list of unavailable items is printed to the user.

If at least one of the requested items is available, it is displayed in the indicated format. Each of these displays has a Continue botton in the lower right corner. In general, the system will not replace the display until the user touches this button. (This technique is also used for messages written to the user.) If only one item is displayed, a "Redisplay" button appears in the lower left corner. Touching this button allows the user to have the current data item

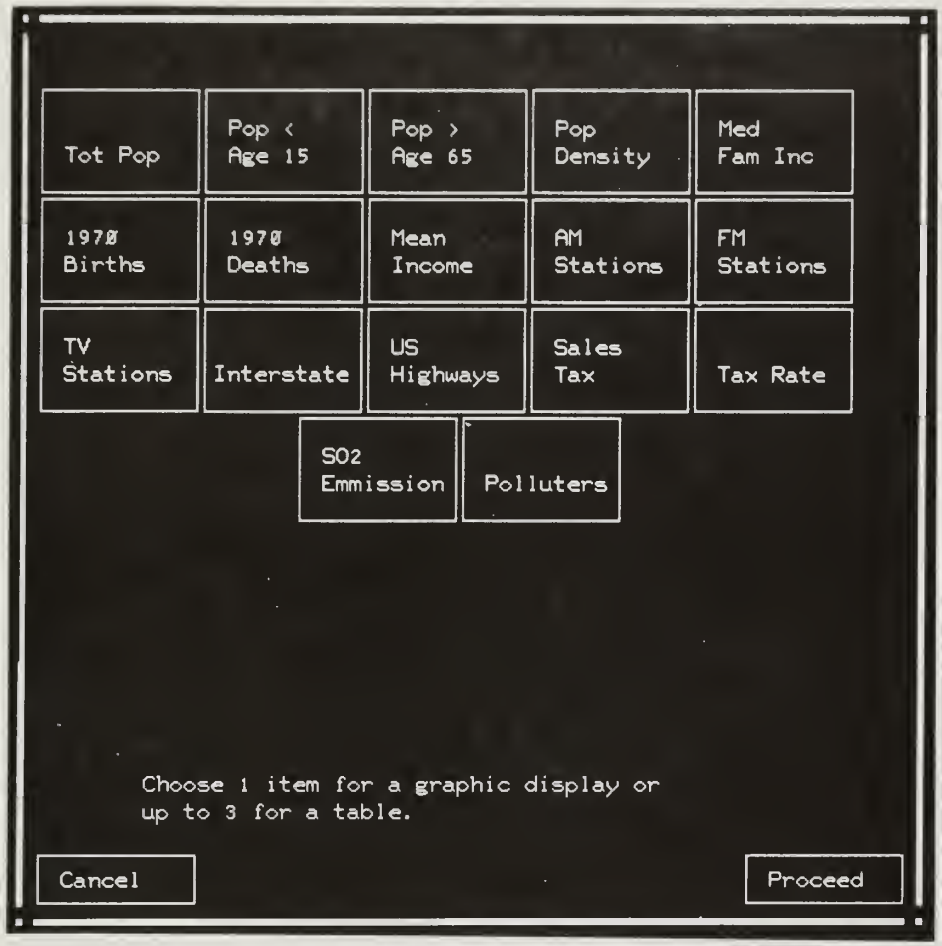

Figure 6

Display Data Page

redisplayed in a different format without going back to the front page. If this button is touched, the system goes immediately to the format selection page.

If the host system is unavailable, no item retievals can be done so only locally resident items can be displayed. In this case the page presenting all the items for the current data base is shortened to list only locally available items. If it happens that the current data base has no locally resident items when the host dies, then the level of interest is automatically switched to be a data base for which items are

Figure 7

Display Format Page

resident. If no data base has any locally resident items, then the current data base is set at the state level and the Display Data button does not appear on the front page.

5.4 Manual Override

This command allows the user to have more direct control over the actions of the front-end system than is allowed by the usual automatic mode. This command differs from the other major commands in that it has a manual override page, similar to the front page, which presents five

sub-commands (figure 8). The system will stay in manual override mode, executing the indicated subcommands, until the user specifies a return to automatic mode. The five subcommands are:

1. Directory,
2. Delete Items,
3. Retrieve Items,
4. Standard Terminal, and
5. Automatic Mode.

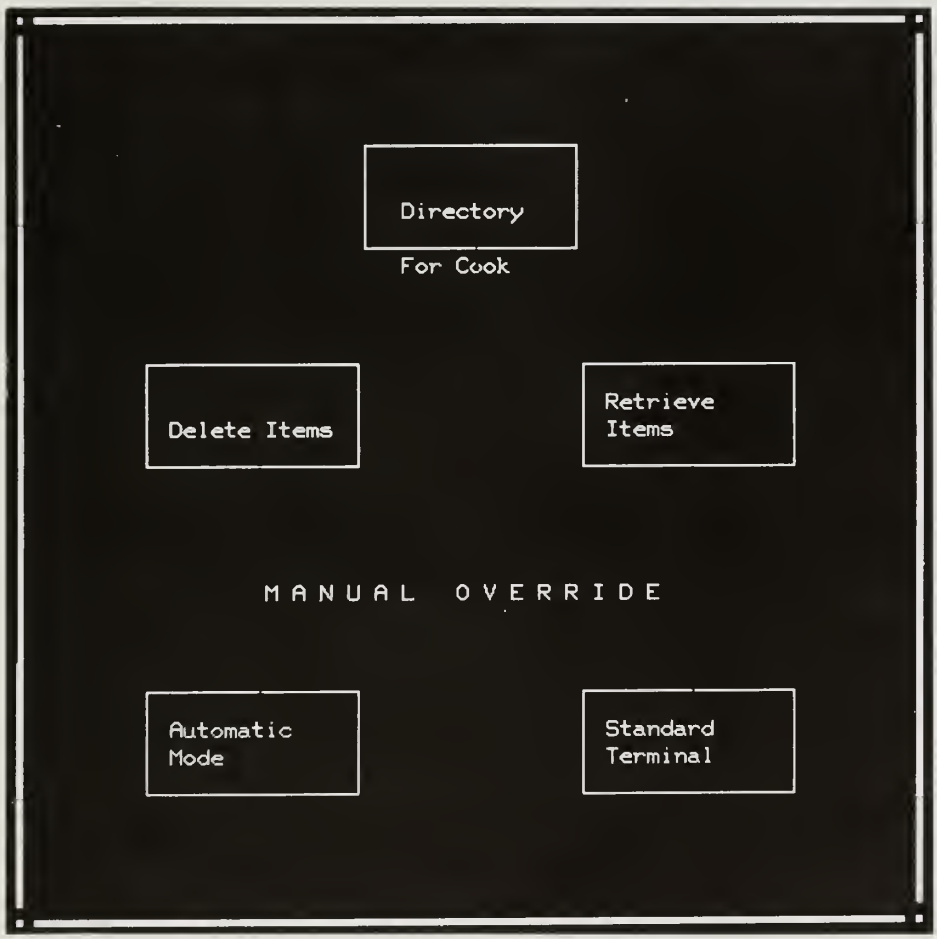

Figure 8

Manual Override Page

Directory. This produces a list of the locally resident items for the current data base.

Delete Items. This allows the user to delete specific items from local memory. It is useful if the user wants to circumvent the least-recently-used replacement algorithm used by the memory manager. If the host is down, this button does not appear on the manual override page.

Retrieve Items. This allows the user to retrieve several items at once. The number of items is limited only by the local memory size. This allows more items to be retrieved at once than by the Display Data command. If the host is down, this button does not appear on the manual override page since no retrievals would be possible.

Standard Terminal. This allows the user to use the terminal as a standard teletype. In this mode, the screen is cleared, the keyboard is enabled, and anything typed on the keyboard or read over the phone line is echoed on the screen. This mode is generally used to re-establish the connection to the host after it has been lost. Since this is vital to recovery after the host has been unavailable, this button always appears on the manual override page.

Automatic Mode. This indicates that the user wants to leave manual override.

Since Standard Terminal mode is the only way to recover from the host being unavailable, the Manual Override Button always appears on the front page.

5.5 Halt

This command causes the DMS front-end system to terminate. An unintentional halt can destroy the stand alone properties of the system

by discarding everything currently in the terminal. Thus the user is asked to verify that the system should stop (figure 9). If the Continue button is hit, the display returns to the front page. If the Halt button is hit, then the system cleans up after itself, prints a message to the user, and goes away.

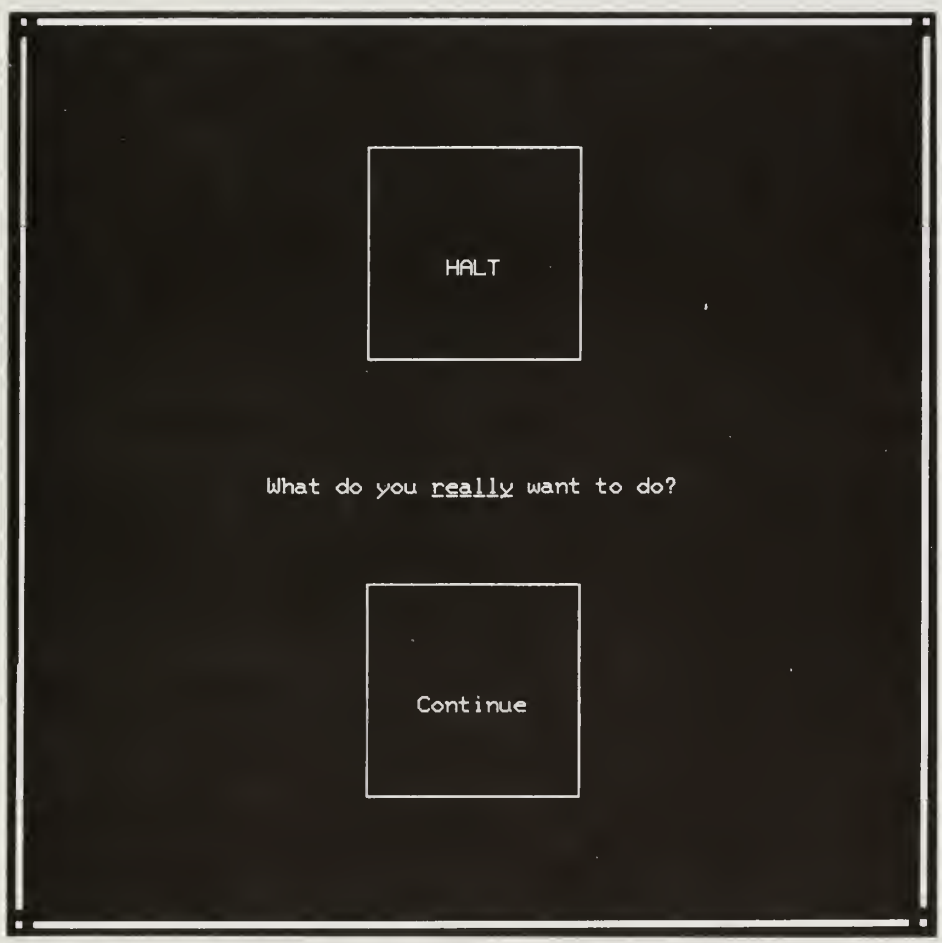

Figure 9

Halt Verification Page

6. DETAILED SYSTEM DESCRIPTION

The description of the front-end system is divided into six areas:

1. the software written to run on the host system,
2. the terminal operating system,
3. front-end system support software,
4. front-end data structures,
5. front-end memory manager, and
6. front-end user interface software.

Detailed discussions of these areas follow.

6.1 Host Support Software

One special purpose routine was written to run on the Multics host system. This routine, which interfaced to the Janus system, prepares data items for transmission by attaching leading and trailing information. The data elements are ordinarily transmitted from the host as the character representation of the numbers. The interfacing routine prepends to this list a synchronizing header sequence and the number of values to be transmitted. The trailer consists of a synchronizing sequence used to detect error conditions in transmission. For ease of interfacing between Multics and the PDP-11, all item values were transmitted as the ASCII representation of the number, and transformed into the PDP-11's internal format by software in the terminal.

6.2 Operating System

A small multi-processing operating system was written for the terminal. The facilities in this system which are of specific concern

to the DMS front-end are the I/O system and screen panel accessing facilities. These two types of routines are further explained below.

I/O system. In this subsystem, a device must be owned before it can be accessed. At most one process at a time can own each device. Accordingly the operating system supplies the subroutines open and close to allow a process to request and relinquish ownership of a device. Once a device is owned, it can be read from and written to by the read and write subroutines.

Panel accessing. Several routines provide facilities for writing to the screen. One of these is screen_clear which erases the entire screen. Two more routines allow strings of text to be written. set_cursor determines where the printing will start (i.e., positions the cursor). printf allows formatted printing of one constant string with an arbitrary number of parameters.

These routines are documented more fully in Appendix C.

6.3 Front-End Support Software

This classification includes routines which are very low level so far as the actual DMS front-end is concerned, but which are too specific to be considered operating system functions. Four of these are described below.

clr_line. This routine blanks out a specified number of character lines on the screen. It starts at a specified line and then positions the cursor at the left edge of the topmost cleared line.

terminal. This procedure causes the terminal to simulate a normal teletype. It takes input from the keyboard and the phone line. All input is printed on the screen, and input from the keyboard is written to the phone line. Input is buffered before it is printed so

that characters from the phone line and from the keyboard are not interspersed on the same line. This routine is used by the front-end to allow the user to log in to the remote host.

get janus values and pr help. These two routines retrieve information from the host into the terminal. get_janus_values is used to retrieve a data item. To do this, get_janus_values formulates and transmits a command line to the host which will cause the host system to send back the desired item. The incoming information is checked to make sure that the leader and trailer sequences are proper, and that the correct number of elements are received. If the item is in proper form, the element values are converted into internal format and stored in the specified memory location. A returned status word indicates the success or type of failure of the attempted retrieval.

pr_help displays a page of explanatory text on the screen. This text is retrieved by formulating and shipping over the phone line a command which will cause the proper explanatory paragraph to be shipped back. The resulting input from the phone line is treated as being the help text. The header sequence is stripped off, and everything up to the trailer field is printed on the screen.

These routines are documented more fully in Appendix C.

6.4 Front-End Data Structures

In order to simplify the front-end software both conceptually and in terms of the amount of code required, two common constructs were represented by the structures item_tag and level_variables. The exact definition of these structures is included in Appendix B.

item tags. As was mentioned in Chapter 5, many options cease to be valid if the host is unavailable. In order that the corresponding

buttons will not be displayed in this case, each potential button has an item_tag associated with it. This structure has two parts: a label and an availability flag. The label field contains the name of the option or the label for button to be displayed. The availability part is a flag which is true if the option is valid when the host is unavailable and false otherwise. This structure is widely used in the front-end software.

level_variables. One of the design goals of the front-end system is the ability to keep portions of two different data bases in local memory simultaneously. Although these two data bases are similar in structure, they differ in details such as item names, number of items, size of local memory, etc. In order to keep such details readily available, a level_variables structure is associated with both the state and county data base. The information kept in these structures is detailed in Appendix B. In general this structure includes all information which depends on the type of data base.

6.5 Memory Manager

The memory management routines have the responsibility of maintaining the local memory cache in an as up-to-date state as possible. This includes retrieving data items from the host and determining which locally resident items are to be overwritten, as necessary. The description of the memory manager includes an explanation of the memory organization and a discussion of four procedures:

1. lru,
2. retr_item,
3. retr_many_item, and
4. fetch_county.

Memory organization. The local memory is divided into two sections, one for each type of data base. Each local data memory is divided into slots the size of one item. Every slot has an associated time-stamp which indicates the last time that slot was referenced. These time-stamps are used to determine which item to replace as new items are retrieved. The availability flags for the items in each data base indicate not only whether or not the item is currently resident in the terminal but also which memory slot the item occupies. Figure 10 diagrams part of the local memory in a typical state. The diagram shows the memory containing three items. Item "% Pop change" is not locally resident, as indicated by an availability flag of 0.

lru. This is the placement routine for the memory manager. It implements a least-recently-used algorithm to determine which memory slot is available for use by a new data item.

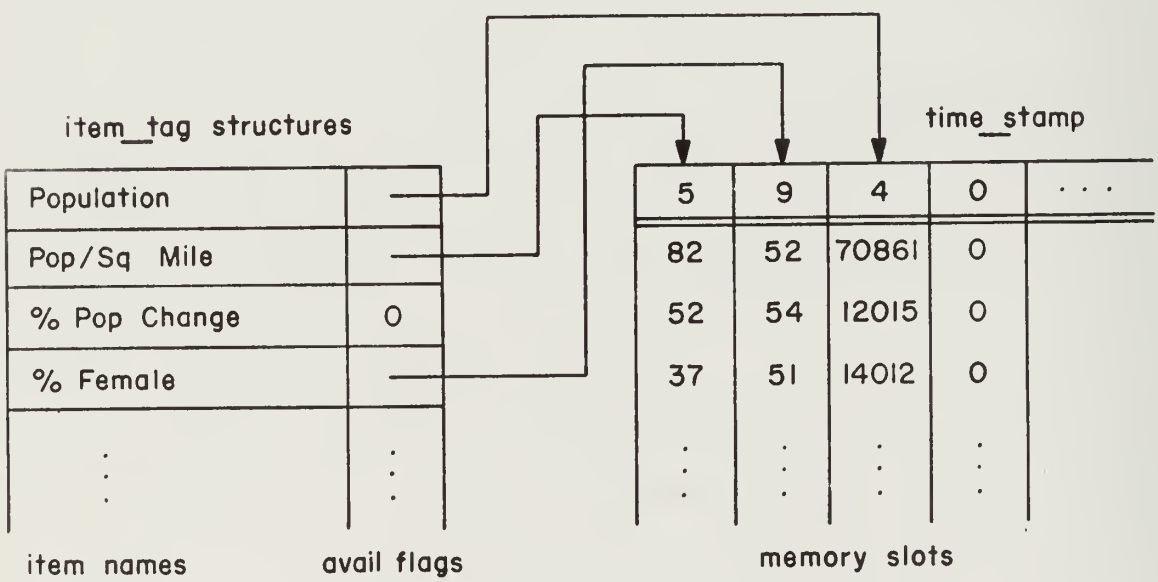

Figure 10

Memory Organization

retr item. This low level routine retrieves a data item from the host. It uses lru to find an available memory slot and get_janus_values to actually fetch the item. retr_item contains checks to assure that the item retrieved by get_janus_values was retrieved correctly. The retrieval is complete either when the item has been fetched correctly or when the host becomes unavailable. When the retrieval is completed retr_item updates the availability flag of each item to indicate the new state. This includes marking the item which previously occupied the pre-empted memory slot as being gone, as well as updating the flag of the new data item.

retr many items. When a user requests several data items, additional checks are performed by retr_many_items to assure that resident requested items are not over-written as other items are retrieved. retr_many_items makes a first pass through the list of requested items checking the availability flag of each item. If the flag is non-zero, then the item is locally resident. If the item was deleted from manual override node, it is undeleted. The time-stamp of the slot occupied by the item is updated. If some of the requested items are not locally available, retr_many_items makes a second pass through the list. On this pass, retr_items is used to retrieve the unavailable items. If one or more items are not available and cannot be retrieved because of host failure, an explanatory message listing the unavailable items is displayed to the user.

fetch county. When an individual data item is referenced, the presence of the item is determined by its availability flag. Since the same list of items is used for all the county data bases, this check is not sufficient when an entire county is to be retrieved. In this case,

the check must be for the availability of each item for the desired county data base. This requires the checking of two availability flags: one for the item and one for the county. Since the list of counties is actually a list of item_tag structures, the availability flags are used to indicate which county data base is currently resident in the terminal. When a county data base is requested, fetch_county will retrieve an item if the availability for either the item or the county is false. After all necessary retrievals have been done, or after the host becomes unavailable, the flags for the counties and the data items are updated by fetch_county. This includes resetting the availability flag for the old county and setting the flag for the new county, if any new items were retrieved. Also, if the host went down after some items were retrieved, the flags for any items remaining from the old county must be reset. If the host goes down before any items for the new county can be retrieved, the memory and availability flags are left as they were upon entrance to fetch_county. After the memory has been updated as much as possible, fetch_county returns to the calling procedure the index of the county which actually has data.

The memory management routines are presented in more detail in Appendix A.

6.6 User Interface

This portion of the system includes facilities for coordinating the various interactions between the user and the terminal system. This includes displaying and interpreting pages of buttons, displaying data items in the proper format, and controlling the flow of the menu of options presented to the user. Each of these sections is discussed further below.

Button routines. To facilitate the utilization of the touch panel, the terminal system uses the internal concept of a button as a structure. Nine routines which provide the button facilities are broken into three categories:

1. Six routines which deal directly with creating, using and deleting individual buttons.
2. get_touch, a general-purpose button handling procedure.
3. get_command and user_command, button facilities which work specifically with the front-end system.

Explanations of these routines follow.

When created, a button has specific dimensions, position on the screen, and label. The creation process consists of entering the button definition into the internal button table. This is done by the procedure add_command. The button is not actually displayed until it is activated by a call to the activate procedure. After a screen has one or more active buttons displayed the program can read the user's touch input by means of get_command. This procedure reads the coordinates of a screen touch, and compares those coordinates with the areas inside all active buttons. If the touch is determined to be inside a button, then the button on the screen is flashed and a special internal tag, which is associated with the touched button, is returned to the calling procedure. Flashing a button consists of lighting all the dots inside the button, then turning them off and re-displaying the button as before. This feedback indicates to the user that a touch was recognized by the system. If the touch is not inside any button, it is ignored.

After a button has been used, one of two things is done to it. Either it is deactivated by the deactivate procedure or it is deleted by

the del_command procedure. Deactivating is the opposite of activating in that the button is marked as inactive but the button definition is not removed from the internal button table. When a button is deleted, its definition is removed from the table. In either case the button is left displayed on the screen and must be explicitly erased. This is usually done by erasing the entire screen of buttons via screen_clear. A button may be individually erased by the lite_box procedure.

The procedure get_touch uses the above button handling routines to display a page full of buttons and to allow the user to pick up to a maximum number of them. This routine takes as input a list of button labels, a list of internal tags to be associated with the buttons, and the maximum number of buttons to be picked. It then creates and displays buttons of a standard size, centered as a group on the screen. Additionally, Proceed and Cancel buttons are created in the lower corners. Finally an explanatory message, also input to the procedure, is displayed on the screen.

As the user touches the buttons, two actions are taken. First the selected button is marked as having been touched by placing a small box in the lower right corner of the button. Second, the message to the user is updated to reflect the number choices still available. After the user has chosen the maximum allowable number of buttons, the message is changed to read "Hit Proceed to continue." If the user makes more selections after this point, the new buttons are recognized as being touched, but buttons chosen initially are deleted from the list of choices. This keeps the number of choices from exceeding the allowed limit.

If the user wishes to delete a selection, a button can be unselected by touching it again. When this occurs, the marking box is erased and the message is again updated. After the user is satisfied that the appropriate options have been specified, the Proceed button must be touched. Forcing the user to always explicitly indicate that all decisions have been made allows the user to change a bad selection or correct an extraneous touch. At any time before the Proceed button is hit the user may abort a command by touching the Cancel button. If this is done, the current page of buttons is erased, get_touch returns a special code to the calling procedure, and the terminal system returns to the front page display.

After the desired buttons have been chosen, and the Proceed button touched, get_touch cleans up after itself and returns to the calling procedure. The cleaning up includes erasing the screen and deleting all the buttons it created. The returned information includes the number of buttons chosen and a list of the internal tags of the chosen buttons.

The details of the seven general button handling routines discussed so far are included in Appendix C.

The procedure get_choice was built to specifically interface between the front-end system and get_touch. get_choice is used to build a list of valid options and pass this list to get_touch along with the other necessary information. An option is valid if either the availability flag of the button is true or a master flag, also input to get_choice, is true. The master is usually a flag which is true if the host is available, and false otherwise. The tags to be associated with the buttons are generated by get_choice to be the index of the buttons in the list input to get_choice. This procedure allows the main coordinating

procedure of the front-end to ignore the state of the backup host so far as determining valid options is concerned. This helps simplify the main routine.

The last major button handling routine is user_command. This procedure is used to display buttons in the format of the front and manual override pages. The buttons on these pages are different from the other buttons used by the front-end system. They are created as the main or manual override loop is entered and then merely activated and deactivated until the loop is exited. This saves the overhead of creating these few buttons every time they are used. The procedure user_command takes a list of pointers into the button table and the availability flags for the commands. Using the same approach as get_command to determine validity, the valid commands are activated. The user is allowed to select one command, with no provision for changing a touch. (This is not significant since every command generates a follow up page which has a Cancel button.) After the command is chosen, the buttons are deactivated and the screen is cleared. The internal tag of the chosen command is returned to the calling procedure.

The code and more detailed explanations of the procedures get_command and user_command are included in Appendix A.

Data item displays. The user interface also includes facilities for displaying data items in three different formats: table, bar chart, and shaded map.

The table is the simplest of the formats. This routine will take up to three data items and display them as columns of numbers centered as a group on the screen. The columns are labeled by the item names and separated by vertical lines. The rows are labeled by the

county or town names and every third row is marked by a dotted line. This is a very standard format. Missing data values are displayed as an asterisk (*).

The second type of display presents one item as a standard bar chart centered on the screen. The element values are represented as horizontal bars, labeled on the right with the numeric value of the element and on the left by the element name. The entire display is labeled at the bottom with the item name. The bars are scaled so that the smallest values becomes relative zero and the bar for the largest value fills the maximum allowable area. All other values are scaled linearly between the maximum and minimum. Again missing data values are represented by an asterisk.

The third type of display, a shaded map, is only available for items in the Illinois data base. This routine draws an outline map of the twenty-nine northern counties of Illinois. The element values of the items are divided into five ranges. Each county is filled in with one of five shades depending on which range includes the value for that county. Counties with missing data values are left blank.

The three display routines are outlined in Appendix C.

Flow control. The main driver coordinates the above procedures to make the terminal perform as discribed in Chapter 5. When the terminal system first comes up, the driver initializes all the level structures, item names, element names, button labels, availability flags, and textual constants which are needed. For the most part, once these values are initialized, they are never changed. Working pointers are switched to reference whatever values are currently in use. After this initialization is done, the driver opens the touch panel and the phone line, and allows

the user to log into the remote host. When this is done, the terminal system goes to the front page display. From this point on, the driver simply loops through successive commands until the user indicates that the front-end system should halt. At that point, after receiving verification, the driver closes the devices it opened, prints a final message, and returns to the operating system supervisor.

The main work of the driver is done in the loop which executes user commands. For the most part, this entails coordinating other procedures in a straight-forward manner under the guidelines of Chapter 5. However, two functions of the driver are not completely obvious to the user. First the driver must ensure that the system housekeeping is done at the beginning of every pass. Second, care is taken so that a county data base is not overwritten unnecessarily.

The system housekeeping includes updating availability flags for commands such as Display Data or Select Data Base if these commands became valid as the result of actions of the previous command. Also, the driver must ensure that the terminal system is, if at all possible, using a data base which has resident data if the host is down.

If the driver always retrieved a county data base when a user switched the level of interest to it, then some counties for which all the items had zero elements would be retrieved. (A county data base will have zero-length items if that county has no towns with populations greater than 2500.) In this case, the fact that the county has no large towns is noted, and the old county data base is left intact in the terminal. This approach prevents unnecessary loss of data, allows the user to go back to the old county of interest without retrieving the data items again, and keeps more interesting data available if the host should become unavailable.

With these two exceptions, the actions of the front-end driver closely follow the system description. A more detailed description of the dms_front_end procedure is included in Appendix A.

7. CONCLUSIONS

After the terminal front-end system was built, it was demonstrated for two groups of people. During the first set of demonstrations approximately thirty people from the Joint Technical Support Activity (the project funding agency) saw the terminal. The second set of demonstrations reached approximately one hundred civilian and military personnel from the Pacific Command. These people ranged from system programmers to DMS users to senior administrative personnel. As a result of these demonstrations and our own observations, several good and bad points of the front-end system were recognized. These results and possible future research is discussed below.

Demonstration results. The terminal system has several shortcomings. Some of these were the result of restrictions designed to keep the demonstration system simple and were not felt to be significant conceptual restrictions. Others were more basic shortcomings.

The restrictions implemented for the sake of simplicity include:

1. limiting the size of the data base so that all displays will fit on one page,
2. assuming that data items are not being updated at the host system, and
3. not making more displays touchable.

The assumption that data items are static was valid in the case of the data base used in this demonstration. However the system could be easily changed to check for updates if this was necessary. The touch facility could have been utilized for more displays. For example, the shaded map could have been made touchable so that touching a particular

county would cause the information for that county to be displayed. These extensions of the demonstration system might be worthwhile in an actual system, but were felt to be unnecessary in this original version.

A more basic shortcoming of the demonstration system is the fact that data items can only be retrieved, not updated or created. Probably the single most frequently asked question from people who viewed the terminal concerned how data might be input through this type of system. This seems to indicate that data input is a significant concern for users, and that they want a better way of inputting information. Data input capabilities were not implemented in the demonstration system primarily because of time constraints. However, any such system which is designed for actual rather than demonstration use should consider the problem of data input and updating.

In spite of these shortcomings, the demonstration system was very favorably received. The demonstrations were introduced by approximately fifteen minutes of lecture covering the concepts to be demonstrated. This was followed by a demonstration performed by a member from the audience. After the short introductory lecture, the audience members were able to successfully get displays of data, change default data bases, etc., using only the instructions included in the front-end system itself.

Overall, the demonstration system was successful in improving the human interface in the areas addressed. The combination of the touch input mechanism and the menu selection in the user's jargon, made the system easy to use. This was demonstrated by the people who saw the system. The combined front-end and host system was more reliable than

the host by itself. Several times during the demonstrations the terminal masked the failure of the host or communication facility.

Future research. Many areas of the philosophy implemented in the demonstration terminal system lead to further research. Some possible extensions are discussed below.

1. The local data manipulating capabilities might be expanded. Methods of updating data values via touch input might be studied. Also, the possibility of using the local processing power to perform calculations, such as correlations between times, on the local data might be explored.
2. The menu selection approach should be studied. This approach can be very limiting. Further study might include the problem of how the system can be made more flexible for the experienced user and still be easy for the novice to use. Further, the possibility of allowing the user to create abbreviations for a sequence of commands might be of interest as the system grows more complex.
3. As the system grows in capabilities, the structure of the system will have to be carefully considered. The demonstration system presented basically a tree of commands. That tree had only three levels and every node had only a few (two to thirty) possible sub-branches. Other systems may not easily fit into such a conceptually simple hierarchy. Structuring a system with many possible functions on a complex set of data so that a user can grasp that structure may be a significant problem.
4. Even more user feedback and hand-holding can be built into the terminal. Possibly the terminal could be expanded to utilize

two or more screens. One screen could perform the same types of functions as the current terminal. The other might serve one of several functions. It might be a user manual, constantly displaying the help text for the current command options. It might allow the user to carry on a conversation with another user about the information being displayed. It might be a scratch pad for the user's personal use. The possibility of using extra screens to perform one or more of these functions poses interesting problems.

These are some areas of possible further work which have grown out of the current project. Each of these areas holds promise for improving the human interface to data management systems. Further, this development should not be prohibitively difficult. The demonstration system, which made a first pass at improving the user interface, took only nine programmer-months to design and build. Further improvements should be possible without excessive programmer efforts. Considering the costs and potential benefits, further study of intelligent terminals as interfaces to existing systems seems worthwhile.

REFERENCES

1. Hoehn, H.J. and Martel, R.A., "A 60 Line Per Inch Plasma Display Panel," IEEE Transactions on Electronic Devices, Vol. ED-18, No. 9, Sept. 1971, pp. 659-663.

2. Janus Beginner's Manual, Overlap Project, Massachusetts Institute of Technology, July 14, 1975.

3. Wyatt, J.B., "Networking Cost Benefits: Some Opportunities," Eighth Hawaii International Conference on System Sciences, 1975, pp. 101-103.

APPENDIX A

Front-End System Code

dms_front_end

```
#
/* Uses :  activate, add_command, bar_graph, build_command_buttons, close, del_command, delete, fetch_county,
        get_choice, get_command, list_ith, map, open, open_pn, pr_help, printf, replot, retr_many_items, screen_clear,
        set_cursor, str_ith, table, terminal, undelete, user_command, user_pause, write.
*/

#include "/mnt/net/jrm/source/constants.incl"
#include "/mnt/deb/thesis/defines.incl"
#include "/mnt/deb/thesis/structures.incl
#include "/mnt/deb/thesis/demo_text.incl"

int     backed_up {false} ;                                     /* flag indicating whether the backup system is available.
                                                                   This flag may be reset by the procedures fetch_county,
                                                                   retr_many, and pr_help.  It is updated after every call
                                                                   to terminal -- it is set if the return status is 1, and
                                                                   reset otherwise. */
int     clock {1} ;                                             /* simulated clock for lru memory management scheme */



cms_front_end () {

int                     buttons [2] ;                           /* general list of internal button tags */
struct  item_tags       c_items [num_cnty_items] ;              /* county items names and their avail flags */
extern  char            *c_itm_names [] ;
extern  char            *c_j_itm_names [num_cnty_items] ;
struct  level_variables c_level ;                               /* variables associated with the county level */
int                     c_mem [c_mem_size] [max_num_towns] ;    /* county memory arrays */
int                     choices [max_num_buttons] ;             /* holds user's choices from a page of buttons */
int                     command ;                               /* the user's command from the touch panel */
struct  item_tags       counties [num_counties] ;               /* list of county names and their avail flags */
extern  char            *county_names [num_counties] ;
int                     cnty_id ;                               /* id of currently resident county */
int                     *cnty_mem [c_mem_size] ;                /* pointers to the county memory arrays */
int                     cnty_usage [c_mem_size] ;               /* time stamps for the county memory arrays */
int                     demo_over ;                             /* flag indicating when the user wants to halt */
struct  item_tags       display_types [max_disp_types] ;        /* buttons for display formats */
char                    *empty_message ;                        /* text for message telling user that there are no items here */
int                     flag ;
int                     fp_b_l [num_commands] ;                 /* internal tags for the front page buttons */
char                    *fp_text [fp_lines] ;                   /* message to be printed on front page */
int                     fp_x [fp_lines] ;                       /* x co-ord of where to print fp messages */
int                     fp_y [fp_lines] ;                       /* y co-ord of where to print fp messages */
struct  item_tags       front_page [num_commands] ;             /* button labels for front page and their avail flags */
char                    *help_file [num_commands] ;             /* name of help files at backup system */
int                     i ;
register  struct  level variables *level ;                      /* general level pointer */
int                     mo_b_l [num_mo_commands] ;              /* internal tags for manual override buttons */
int                     mo_mode ;                               /* flag which is true while the user is in manual override */
struct  item_tags       mo_page [num_mo_commands] ;             /* button labels and avail flags for mo page */
char                    *mo_text [mo_lines] ;                   /* message to be written on the mo page */
int                     mo_x [mo_lines] ;                       /* x co-ords of where the messages are to appear */
int                     mo_y [mo_lines] ;                       /* y co-ords of where the messages are to appear */
```

```
/* dms_front end                                                                    Page 2 */

char                    *names [max_num_displays] ;             /* user names of items to be displayed */
int                     num ;
int                     phone_id ;                              /* device id of the phone line */
struct  item_tags       sel_db_page [num_db_types] ;            /* buttons and avail flags for select data base page */
struct  item_tags       st_items [num_state_items] ;            /* list of state items and their avail flags */
extern  char            *st_itm_names {} ;
extern  char            *st_j_itm_names [num_state_items] ;
struct  level_variables st_level ;                              /* variables associated with the state level */
int                     st_mem [st_mem_size] [num_counties] ;   /* actual state memory arrays */
int                     *state_mem [st_mem_size] ;              /* pointers to the state memory arrays */
int                     state_usage [st_mem_size] ;             /* the time stamps associated with each memory array */
int                     status ;                                /* indicates the return status of various subroutines */
int                     temp ;
int                     touch_id ;                              /* device id of the touch panel */
extern  char            *town_names [num_counties] [max_num_towns] ;
int                     type ;                                  /* indicates the type of display desired */
int                     values [max_num_displays] ;             /* pointers to item values to be displayed */


/*  .
```

```
/* First, do the various initializations.  This includes setting up the name and avail flags for various buttons,
   setting the level variable structures to the proper values, and setting up the texts for messages that appear routinely. */

      for (i = 0 ;  i < num_cnty_items ;  ++i) {
             c_items [i].name = c_itm_names [i] ;
             c_items [i].avail = false ;
      }

      c_level.mem_array = cnty_mem ;
      c_level.mem_usage = cnty_usage ;
      c_level.num_arrays = c_mem_size ;
      c_level.num_items = num_cnty_items ;
      c_level.items = &c_items ;
      c_level.item_j_names = c_j_itm_names ;
      c_level.elt_names = town_names ;
      c_level.dsn = "<unitialized>" ;
      c_level.num_displays = num_c_disp_types ;

      for (i = 0 ;  i < num_counties ;  ++i) {
             counties [i].name = county_names [i] ;
             counties [i].avail = false ;
      }

      for (i = 0 ;  i < c_mem_size ;  ++i)
             cnty_mem [i] = &c_mem [i] [0] ;

      display_types [disp_table].name = "Table" ;                  display_types [disp_table].avail = true ;
      display_types [disp_bar].name = "Bar Chart" ;                display_types [disp_bar].avail = true ;
      display_types [disp_map].name = "Map" ;                      display_types [disp_map].avail = true ;

      empty_message = "       There are no %s items in the terminal.\n" ;

                                                                   /* the numbers put in fp_x and fp_y are magic.  They are cursor
                                                                      positions for text that appears on the front page, and are
                                                                      co-ordinated with the boxes built by build_command_buttons. */
      fp_x [0] = 21 ;  fp_y [0] = 22 ;  fp_text [0] = "Using " ;
      fp_x [1] = 27 ;  fp_y [1] = 22 ;
      fp_y [2] = 22 ;  fp_text [2] = " data base\n" ;
      fp_x [3] = 0 ;   fp_y [3] = 2 ;   fp_text [3] = "       Copyright 1975, Center for Advanced Computation,\n" ;
      fp_x [4] = 0 ;   fp_y [4] = 1 ;   fp_text [4] = "                  University of Illinois\n" ;

      front_page [comm_help].name = "    HELP" ;                   front_page [comm_help].avail = false ;
      front_page [comm_sel_db].name = "Select Data Base" ;         front_page [comm_sel_db].avail = false ;
      front_page [comm_display].name = "Display Data" ;            front_page [comm_display].avail = false ;
      front_page [comm_mo].name = "Manual Override" ;              front_page [comm_mo].avail = true ;
      front_page [comm_halt].name =      HALT" ;                   front_page [comm_halt].avail = true ;

      help_file [comm_help] = "help" ;
      help_file [comm_sel_db] =  select_db  ;
      help_file [comm_display] = "display_data" ;
      help_file [comm_mo] =  manual mode" ;
      help_file [comm_halt] = "halt" ;

      level = &st_level ;
```

```
        mo_page [mo_dir].name = "Directory" ;                   mo_page [mo_dir].avail = false ;
        mo_page [mo_delete].name =  Delete Items" ;             mo_page [mo_delete].avail = false ;
        mo_page [mo_retr].name = 'Retrieve Items" ;             mo_page [mo_retr].avail = false ;
        mo_page [mo_terminal].name =  Standard Terminal" ;      mo_page [mo_terminal].avail = true;
        mo_page [mo_automatic].name = "Automatic Mode  ;        mo_page [mo_automatic].avail = true ;

                                                                /* the numbers put in mo_x and mo_y are magic.  They are cursor
                                                                   positions for text that appears on the manual override page, and
                                                                   are co-ordinated with the boxes made by build_command_buttons. */
        mo_x [0] = 25 ;  mo_y [0] = 23 ;  mo_text [0] = "For " ;
        mo_x [1] = 29 ;  mo_y [1] = 23 ;
        mo_x [2] = 17 ;  mo_y [2] = 11 ;  mo_text [2] = "M A N U A L   O V E R R I D E\n' ;

        sel_db_page [state_db].name = "State-wide Interest" ;   sel_db_page [state_db].avail = true ;
        sel_db_page [county_db].name =  Select County" ;        sel_db_page [county_db].avail = true ;

        for (i = 0 ;  i < num_state_items ; ++i) {
                st_items [i].name = st_itm_names [i] ;
                st_items [i].avail = false ;
        }

        st_level.mem_array = state_mem ;
        st_level.mem_usage = state_usage ;
        st_level.num_arrays = st_mem_size ;
        st_level.num_elements = num_counties ;
        st_level.num_items = num_state_items ;
        st_level.items = &st_items ;
        st_level.item_j_names = st_j_itm_names ;
        st_level.elt_name = county_names ;
        st_level.dsn = "Illinois  ;
        st_level.num_displays = num_st_disp_types ;

        for (i = 0 ;  i < st_mem_size ;  ++i)
                state_mem [i] = &st_mem [i] [0] ;


        phone_id = open_ph () ;                                 /* open the phone line for our use */
        if (phone_id < 0) {                                     /* if the open failed for some reason,
                                                                   will not be able to do the demo, so give up */
                printf ("%c", form_feed) ;
                set_cursor (0, screen_center_y) ;
                printf ("              Cannot do demo -- unable to open phone\n') ;
                return ;
        }
        touch_id = open (TOUCH_PANEL, 1, &status) ;             /* open the touch panel for our use */
        status = terminal (&phone_id) ;                         /* let the user do the various connecting and
                                                                   logging in required to connect to the backup system. */
        backed_up = status == 1 ? true : false ;                /* set value of backed_up depending on return status of terminal */
        build_command_buttons (front_page, num_commands, fp_b_1) ;
                                                                /* create the buttons for the standard front page */
        demo_over = false ;

/*
```

```
/* dms_front_end                                                                    Page 5 */

/* Now that the initialization is done, we can proceed with the body of the program.  This consists of a large loop
   which gets the user's command, performs some action based on that command and then loops back for a new command.  This
   process continues until the user finally stops it by hitting the halt command. */

        while (!demo_over) {
                                                        /* need to check the status of what is currently in the terminal
                                                           and make sure that all the appropriate avail flags are set */
                if (st_level.item_resident && c_level.item_resident)
                        front_page [comm_sel_db].avail = true ;
                if (level->item_resident) {
                        front_page [comm_display].avail = true ;
                        mo_page [mo_dir].avail = true ;
                }
                else if (!backed_up) {                  /* if it is the case that there is no data in the terminal for
                                                           current level and the backup system is not available, then
                                                           need to change to a level that does have data.  If neither
                                                           level has any data, then will change to the state level. */
                        if (c_level.item_resident) level = &c_level ;
                        else level = &st_level ;
                }
                fp_text [1] = level->dsn ;      fp_x [2] = 27 + str_lth (level->dsn) ;
                                                        /* since this part of the front page message changes, it gets
                                                           set up each time thru */

                command = user_command (touch_id, num_commands, front_page, fp_b_l, backed_up, fp_lines, fp_x, fp_y, fp_text) ;
                                                        /* get the user's command */


                switch (command) {


                /* User wants help.  Find out which command needs explanation, then call pr_help to display the text. */
                case comm_help :
                        num = get_choice (touch_id, choices  front_page, backed_up, num_commands,
                                "For which command do you want help?\n", 1) ;
                        if (num <= 0) break ;
                        pr_help (help_file [choices [0]]) ;
                        user_pause (touch_id) ;
                        break ;
```

```
        /* Change default data base.  Find out whether interested in the entire state or a particular county. */
        case comm_sel_db :
                num = get_choice (touch_id, choices, sel_db_page, backed_up, num_db_types,
                        "Select desired default data base.\n", 1) ;
                if (num <= 0) break ;
                switch (choices [0]) {

                /* If interested in the state, simply switch level to point to the state variables. */
                case state_db :
                select_state_db :
                        level = &st_level ;
                        break ;

                /* If interested in a specific county, the action is more complex.  First, find out which county is desired,
                   then try to fetch the items for that county.  (If there are no "interesting" (large) towns  in the
                   county, the fetch is not done.  This allows the data for the last "interesting" county to remain in the
                   terminal.)  If the attempt to retrieve the items failed, print a message to the user explaining what
                   happened.  After all this point level at the county variables and set up necessary variables. */
                case county_db :
                        num = get_choice (touch_id, &cnty_id, counties, backed_up, num_counties,
                                "Select county of interest.\n", 1) ;
                                                /* find out which county the user wants */
                        if (num <= 0) break ;
                        if ((c_level.num_elements = list_lth (&town_names [cnty_id] [0], max_num_towns)) > 0) {
                                cnty_id = fetch_county (phone_id, &c_level, &counties, cnty_id, &status) ;
                                if (status) {           /* if the attempted retrieval failed, print the
                                                           appropriate message to the user */
                                        set_cursor (0, screen_center_y) ;
                                        printf ("       Unable to complete county retrieval as the backup\n") ;
                                        printf ("       system is unavailable.\n") ;
                                        if (status == have_old_cnty) {
                                                printf ("       Will use the %s county data base\n"
                                                counties [cnty_id].name) ;
                                                printf ("       as these items are already in the terminal.\n") ;
                                        }
                                        else if (status == have_no_cnty)
                                                printf ("       Will use the Illinois data base.\n") ;
                                        user_pause (touch_id) ;
                                        if (status == have_no_cnty) goto select_state_db ;
                                }
                        }
                        level = &c_level ;              /* set up the variable values */
                        level->elt_name = &town_names [cnty_id] [0] ;
                        level->num_elements = list_lth (level->elt_name, max_num_towns) ;
                        level->dsn = counties [cnty_id].name ;
                        break ;
                }
                break ;
```

```
/* Display data.  To do this, find out what items are to be displayed, and if there's only one of them, find out
   in what format.  Attempt to retrieve the items, and if at least one of them is available, do the display.  If
   only one item is displayed, the user has the option of seeing it redisplayed in a different format. */
case comm_display :
        if (level->num_elements == 0) {         /* if the number of elements for this level is 0,
                                                   then this is a county with no "interesting" towns. */
                set_cursor (0, screen_center_y) ;
                printf ("       %s county has no interesting towns.\n", level->dsn) ;
                user_pause (touch_id) ;
                break ;
        }
        num = get_choice (touch_id, choices, level->items, backed_up, level->num_items,
                "Choose 1 item for a graphic display or\n       up to %d for a table.\n", max_num_displays) ;
                                                /* get the list of items the user wants to see */
        if (num <= 0) break ;
        flag = true ;
        while (flag) {                          /* do the display at least once, and possibly more
                                                   if only one item is displayed */
                if (num == 1) {                 /* if only 1 item, get the choice of format */
                        temp = get_choice (touch_id, &type, display_types, backed_up,
                                level->num_display, "How do you want the data displayed?\n", 1) ;
                        if (temp < 0) break ;
                        else if (temp == 0) type = disp_table ;
                } else type = disp_table ;      /* otherwise, format must be a table */
                temp = retr_many_items (phone_id, touch_id, level, num, choices) ;
                                                /* attempt to get the items -- temp will be the number
                                                   actually available */
                if (temp == 0) break ;
                if (temp < num) {               /* if not all the items were retrieved, remove the unavailable
                                                   items from the choices list */
                        temp = 0 ;
                        for (i = 0 ;  i < num ;  ++i)
                                if (level->items [choices [i]].avail)
                                        choices [temp++] = choices [i] ;
                        num = temp ;
                }
                switch (type) {                 /* do display depending on the format requested */
                case disp_table :
                        for (i = 0 ;  i < num ;  ++i) {
                                names [i] = level->items [choices [i]].name ;
                                values [i] = level->mem_array [level->items [choices [i]].avail - 1] ;
                        }
                        table (names, num, level->elt_names, level->num_elements, values) ;
                        break ;
                case disp_bar :
                        bar_graph (bg_left_edge, bg_bottom_line, bg_width, bg_height,
                                level->num_elements, level->items [choices [0]].name,
                                level->mem_array [level->items [choices [0]].avail - 1],
                                level->items [choices [0]].name) ;
                        break ;
                case disp_map :
                        map (num_shades, level->mem_array [level->items [choices [0]].avail - 1],
                                level->num_elements, level->items [choices [0]].name) ;
                        break ;
                }
```

```
                        if (num == 1) flag = replot (touch_id) ;
                        else {
                                user_pause (touch_id) ;
                                flag = false ;
                        }
                }
                break ;


        /* Change into manual override mode.  This mode allows more direct control over the items in the terminal than is
           possible in the automatic mode.  This mode, unlike the other commands, is a self-contained loop which keeps going
           until the user explicitly indicates that it should stop. */
        case comm_mo :                          /* first some initialization is required */
                build_command_buttons (mo_page, num_mo_commands, mo_b_1) ;
                mo_text [1] = level->dsn ;
                mo_mode = true ;
                while (mo_mode) {                       /* loop until user says to stop */
                        command = user_command (touch_id, num_mo_commands, mo_page, mo_b_1,
                                backed_up, mo_lines, mo_x, mo_y, mo_text) ;
                        switch (command) {

                        /* Directory.  Produce a list of items for this level which are currently in the terminal. */
                        case mo_dir :
                                if (level->item_resident == 0) {
                                        set_cursor (0, screen_center_y) ;
                                        printf (empty_message, level->dsn) ;
                                        user_pause (touch_id) ;
                                        break ;
                                }
                                printf ("%c\nThe following items for the %s data base\n", form_feed, level->dsn) ;
                                printf ("are currently resident in the terminal :\n") ;
                                for (i = 0 ;  i < level->num_items ;  ++i)
                                        if (level->items [i].avail > 0)
                                                printf ("        %s\n", level->items [i].name) ;
                                user_pause (touch_id) ;
                                break ;

                        /* Delete items.  This will circumvent the normal least-recently-used memory replacement scheme.
                           Any items which are deleted but not over-written, will be undeleted upon leaving this mode. */
                        case mo_delete :
                                if (level->item_resident == 0) {
                                                        /* check for no items in the terminal */
                                        set_cursor (0, screen_center_y) ;
                                        printf (empty_message, level->dsn) ;
                                        user_pause (touch_id) ;
                                        break ;
                                }
                                num = get_choice (touch_id, choices, level->items, 0, level->num_items,
                                        "Choose items to be deleted.\n", level->num_items) ;
                                for (i = 0 ;  i < num ;  ++i)
                                        delete (level->items, level->mem_usage, choices [i]) ;
                                break ;
```

```
                        /* Retrieve items.  Allows up to an entire "memory-full" of items to be retrieve at once,
                           rather than the few at a time allowed by the display command. */
                        case mo_retr :
                                num = get_choice (touch_id, choices, level->items, backed_up, level->num_items,
                                        "Choose items to be retrieved.\n", level->num_arrays) ;
                                retr_many_items (phone_id, touch_id, level, num, choices) ;
                                break ;

                        /* Standard terminal mode.  The primary use of this mode is to allow the user to re-connect to
                           the backup system after the connection has been lost. */
                        case mo_terminal :
                                if (phone_id < 0) phone_id = open_ph () ;
                                status = terminal (&phone_id) ;
                                backed_up = status == 1 ? true : false ;
                                break ;

                        /* Automatic mode.  Indicates that the user is ready to leave manual override mode. */
                        case mo_automatic :
                                mo_mode = false ;
                                break ;
                        }
                }
                for (i = 0 ; i < level->num_items ; ++i) /* undelete any deleted items */
                        if (level->items [i].avail < 0)
                                undelete (level->items, level->time_stamp, i) ;
                del_command (num_mo_commands, mo_b_l);
                break ;


        /* User hit the stop button.  Since an inadvertent halt can just ruin your day, get verification that we
           are really supposed to stop. */
        case comm_halt :
                buttons [0] = add_command ("     HALT", 6, 10, 4, 4, 1) ;
                buttons [1] = add_command ("  CONTINUE", 6, 2, 4, 4, 0) ;
                activate (2, buttons, touch_id) ;
                set_cursor (0, screen_center_y) ;
                printf ("              What do you really want to do?\n") ;
                demo_over = get_command (touch_id) ;
                del_command (2, buttons) ;
                break ;
        }
}
/*
```

```
/* dms_front_end                                                    Page 10 */

/* the user wants to quit, so clean up, close connections, and go away. */

        del_command (num_commands, fp_b 1) ;
        screen clear () ;
        set_cursor (screen center x - 4, screen_center_y) ;
        printf ("Demo over\n\n\n\n\n") ;
        close (touch_id, &status) ;
        if (backed_up) {
                write (phone_id, "lv", 3, &status) ;
                write (phone_id, "exit", 5, &status) ;
                write (phone_id, "logout", 7, &status) ;
        }
        close (phone_id, &status) ;
        return ;
}
```

Other DMS Front-End Procedures

```
#
/* build command buttons -- This procedure builds five buttons in the format of the front page.
   It takes an item_tags structure containing the text to label the buttons and the number of
   buttons to be built (in case the procedure is ever generalized) and fills in the butt_tag
   array with the tags returned by add_command.

   The numbers passed as parameters to add_command are relatively magic.  The last one is the index
   of that button which will be returned whenever that button is hit.  The others specify the
   placement on the screen and size of the buttons.

   Uses :  add_command
*/

#include "/mnt/deb/thesis/structures.incl"

build_command_buttons (text, num, butt_tag)
struct  item_tags       *text ;
int     num ;
int     butt_tag [] ;
{
        butt_tag [0] = add_command (text[0].name, 6, 12, 4, 2, 0) ;
        butt_tag [1] = add_command (text[1].name, 2, 8, 4, 2, 1) ;
        butt_tag [2] = add_command (text[2].name, 10, 8, 4, 2, 2) ;
        butt_tag [3] = add_command (text[3].name, 2, 2, 4, 2, 3) ;
        butt_tag [4] = add_command (text[4].name, 10, 2, 4, 2, 4) ;
        return ;
}
```

```
#
/* delete -- This procedure explicitely deletes an item from a memory space.  (The item is not removed until
   another item is actually put in that slot.  Thus an item can be "undeleted" any time before it is actually
   over-written without a fetch from the backup system.)  The parameters are the item_tags structure for the items
   in this level, the time stamp array for this level, and the numer of the item to be deleted.
*/

#include "/mnt/deb/thesis/structures.incl

delete (items, time_stamp, item_num)
struct  item_tags  *items ;             /* list of items and their availability flags */
int     time_stamp [] ;                 /* headers associated with the memory slots */
int     item_num ;                      /* number of the item to be deleted */
{
int     temp ;

        temp = items [item_num].avail ; /* find the slot used by the item */
        if (temp < 0) temp = -temp ;    /* take care of the case of trying to delete an already deleted item */
        time_stamp [temp - 1] = -1 ;    /* mark the appropriate slot as empty */
        items [item_num].avail = -temp; /* mark the item as deleted but not gone */
        return ;
}
```

```
#
/* fetch_county -- This procedure attempts to make available as many items as possible for a given
   county.  If an item is already here, it is undisturbed.  If not, an attempt is made to
   retrieve it.  If the backup system is not available, one of several things happens, depending
   on how much information is already in the terminal.  If part of the desired county is available,
   the rest is marked as not available, and the county_id is left at that county.  If no part of the
   desired county is available then the county_id is set to be the county which is already in the
   terminal.  If no county is in the terminal, the county_id is set to be uninitialized.

   A return status is set according to :
        0               : everything ok, have all of desired county
        have_new_cnty   : backup system unavailable, using part of desired county
        have_old_cnty   : backup system unavailable, using old county
        have_no_cnty    : backup system unavailable, no county availble
   The procedure then returns county_id.

   Uses :  retr_item, undelete
*/

#include "/mnt/deb/thesis/defines.incl"
#include "/mnt/deb/thesis/structures.incl"

fetch_county (dev_id, level, counties, cnty_id, rtn_status)
int     dev_id ;
struct  level_variables *level ;
struct  item_tag        *counties ;
int     cnty_id ;                                       /* number of desired county */
int     *rtn_status ;                                   /* return status word */
{
int     count,                                          /* number of items available for this county */
        old_cnty,                                       /* number of the previously resident county */
        i,
        status ;                                        /* internal status word */
extern  int     backed_up ;

/*
```

```
/* fetch_county                                                                    Page 2 */

/* as long as the backup system is available, check each item to see if it is resident, and retrieve it necessary */
        for (count = 0 ;  count < level->num_items && backed_up ;  count =+ backed_up ? 1 : 0) {
                if (!(counties [cnty_id].avail) || !(level->items [count].avail))
                        retr_item (dev_id, level, count, &status) ;
                else
                if (level->items [count].avail < 0)
                        undelete (level->items, level->time_stamp, count) ;
        }

/* find which, if any, county used to be resident */
        for (old_cnty = 0 ;  !counties [old_cnty].avail && old_cnty < num_counties ;  ++old_cnty) ;
        if (old_cnty == num_counties) old_cnty = uninitialized ;

/* update the avail flags for counties and items depending on the value of backed_up */
        if (backed_up) {                                /* indicates everything is ok */
                if (old_cnty != uninitialized) counties [old_cnty].avail = false ;
                counties [cnty_id].avail = true ;
                *rtn_status = 0 ;
        } else {                                        /* if not everything ok */
                if (count == 0) {                       /* if retrieved no items, things are still as before */
                        if (cnty_id == old_cnty) *rtn_status = have_new_cnty ;
                        else {
                                *rtn_status = (old_cnty == uninitialized) ? have_no_cnty : have_old_cnty ;
                                cnty_id = old_cnty ;
                        }
                } else {                                /* otherwise, go with whatever have of the new county */
                        *rtn_status = have_new_cnty ;
                        counties [cnty_id].avail = true ;
                        if (cnty_id != old_cnty && old_cnty != uninitialized) {
                                counties [old_cnty].avail = false ;
                                for (; count < level->num_items ; ++count)
                                        level->items [count].avail = false ;
                        }
                }
        }
        return (cnty_id) ;
}
```

```
#
/* get choice -- This procedure takes a list of user options and their avail flags (an item_tags structure)
   and based on the value of a master availability flag, presents the valid options to the user.  It then
   returns the user's choices as indices into the original list.

   Uses :  get_touch
*/

#include "/mnt/deb/thesis/defines.incl"
#include "/mnt/deb/thesis/structures.incl"


get_choice (file_id, choices, buttons, master, num_in, text, max_out)
int     file_id ;                               /* id of input file */
int     choices [] ;                            /* array to hold user's responses */
struct  item_tags  *buttons ;                   /* button labels and their avail flags */
int     master ;                                /* master flag to be used in building the button list */
int     num_in ;                                /* max number of buttons */
char    *text ;                                 /* text of user message */
int     max_out ;                               /* max number of choices a user in allowed to make */
{
char    *butt_labels [max_num_buttons] ;        /* labels for buttons to be built */
int     butt_tags [max_num_buttons] ;           /* user tags for buttons which are built */
int     num,
        count ;                                 /* number of buttons built */

        count = 0 ;
        for (num = 0 ;  num < num_in ;  ++num) {
                if ((buttons [num].avail > 0) || master) {
                        butt_labels [count] = buttons [num].name ;
                        butt_tags [count] = num ;
                        ++count ;
                }
        }

        num = get_touch (file_id, butt_labels, butt_tags, choices, count, text, max_out) ;

        return (num) ;
}
```

```
#
/* hit_mem_slot -- This procedure simply marks a memory slot as being recently used.
*/

#include "/mnt/deb/thesis/defines.incl"
#include "/mnt/deb/thesis/structures.incl"

hit_mem_slot (level, slot)
struct  level_variables *level ;
int     slot ;                                  /* index of the slot being referenced */
{
extern  int     clock ;

        level->time_stamp [slot] = ++clock ;
        return ;
}
```

```
/* lru -- This procedure finds and returns the index of either an unused memory slot or, if they are
   all in use, the one not referenced for the longest time.
*/

lru (map_size, time_stamp)
int     map_size ;              /* number of slots in this memory */
int     time_stamp [] ;         /* the time-stamps for each slot */
{
int     oldest,                 /* index of the slot used least recently */
        i ;

        for (oldest = i = 0 ;  i < map_size && time_stamp [oldest] ;  ++i) {
                if (time_stamp [i] == 0) oldest = i ;   /* if found an empty slot, use it */
                else if (time_stamp [i] < time_stamp [oldest]) oldest = i ;
                                                        /* otherwise, remember the lru slot */
        }
        return (oldest) ;
}
```

```
/* replot -- This procedure displays two buttons giving the user the option of seeing the current
   item in a different format or of continuing.  It returns a 1 for a replot and 0 for continue.
   The numbers passed to add_command are relatively magic.  The last one is the tag which will
   be returned by get_command.  The others specify the placement on the screen and size of the
   buttons.  (Proceed will be in the lower right corner, and Replot will be in the lower left.)

   Uses :  activate, add_command, del_command, get_command
*/

replot (dev_id)
int     dev_id ;
{
int     buttons [2] ;
int     num ;

        buttons [0] = add_command ("Proceed", 12, 0, 1, 3, 0) ;
        buttons [1] = add_command ("Replot Data", 1, 0, 1, 3, 1) ;
        activate (2, buttons, dev_id) ;
        num = get_command (dev_id) ;
        del_command (2, buttons) ;
        return (num) ;
}
```

```
#
/* retr_item -- This procedure attempts to retrieve an item for a given level.  It finds the least
   recently used memory slot, and attempts to fetch the item into that slot.  Depending on the
   success of that attempt, the value of the avail flags for the involved items are updated.

   A return status flag is set according to :
        0 : everything ok
        1 : retrieve failed

   Uses :  get_janus_values, hit_mem_slot, lru, printf, screen_clear, set_cursor
*/

#include "/mnt/deb/thesis/defines.incl"
#include "/mnt/deb/thesis/structures.incl"


retr_item (dev_id, level, new_item, rtn_status)
int     dev_id ;                                /* id of the device from which to do the read */
struct  level_variables  *level ;               /* the structure of all the variables for the
                                                   current level */
int     new_item ;                              /* number of the item to be retrieved */
int     *rtn_status ;                           /* return status, indicating either success or the
                                                   type of failure */
{
int     slot,                                   /* slot in memory where the item will be stored */
        old_owner,                              /* the former owner of that slot */
        tries,                                  /* counter of the number of attempts to retrieve the item */
        num_retrieved,                          /* the number of elements actually retrieved in the item */
        status ;                                /* local status flag */
extern  int     backed_up ;


/*
```

```
/* find a slot to put the new item in */
        slot = lru (level->num_arrays, level->time_stamp) ;  /* find an available slot */
        for (old_owner = 0 ;                                 /* find the old owner of that slot */
                (level->items [old_owner].avail != slot + 1) && (- level->items [old_owner].avail != slot + 1) &&
                (old_owner < level->num_items) ;  ++old_owner) ;
        if (old_owner == level->num_items) old_owner = uninitialized ;

/* try to fetch the item -- will keep trying until either have a successful retrieve with the proper
   number of elements or the backup system is no longer available */
        num_retrieved = 0 ;
        status = 0 ;
        tries = 0 ;
        screen_clear () ;
        set_cursor (0, screen_center_y) ;
        while (((num_retrieved != level->num_elements) && !status) || status == garoled_transmition) {
                printf ("%c                Retrieving %s ", erase_line, level->items[new_item].name) ;
                                                     /* print a message so the user knows we're still here */
                num_retrieved =                      /* attempt to retrieve the item */
                        get_janus_values (dev_id, level->dsn, level->item_j_name [new_item],
                        level->mem_array[slot], level->num_elements, &status) ;
                ++tries ;
        }
        screen_clear () ;

/* mark items as available or gone, depending on the value of the returned status */
        switch (status) {
        case 0 :                                     /* everything OK */
                if (old_owner != uninitialized) level->items [old_owner].avail = false ;
                level->items [new_item].avail = slot + 1 ;
                hit_mem_slot (level, slot) ;
                level->item_resident = true ;
                break ;
        case host_died_before_transmition :
                if (tries == 1) {                    /* if died on the first try, nothing was destroyed */
                        backed_up = false ;
                        break ;
                }                                    /* otherwise, this is just like the next case */
        case host_died_during_transmition :
                if (old_owner != uninitialized) level->items [old_owner].avail = false ;
                level->items [new_item].avail = false ;
                backed_up = false ;
                break ;
        ]

        *rtn_status = status ? 1 : 0 ;
        return ;
}
```

```
/*
/* retr_many_items -- This procedure attempts to make available all the items in a given list.  First, the avail flags
   are checked to see if any of the items are already available, and their corresponding memory usage stamps are updated, so that
   they will not be over-written.  Then any items not available are retrieved.  If the retrievals fail (because the backup
   system is unavailable), then a message is written to the user.

   Uses :  hit_mem_slot, printf, retr_item, screen_clear, set_cursor, undelete, user_pause
*/

#include "/mnt/deb/thesis/defines.incl
#include "/mnt/deb/thesis/structures.incl

retr_many_items (dev_id, touch_id, level, num, item_list)
int     dev_id ;                                        /* id of the device from which to read the item */
int     touch_id ;                                      /* id of the touch panel */
struct  level_variables  *level ;                       /* structure of variables for current level */
int     num ;                                           /* number of items to be retrieved */
int     item_list [] ;                                  /* list of numbers of items to be retrieved */
{
int     count,                                          /* count of number of desired items available */
        i,
        status ;                                        /* local status flag */
extern  int     backed_up ;                             /* global variable which is true if the backup system is available */
                                                        /* this variable may be changed by a call to retr_item */

        count = 0 ;                                     /* initially, assume no items available */

/* make sure any items which are already here are available */
        for (i = 0 ;  i < num ;  ++i) {
                if (level->items [item_list [i]].avail  != 0) {
                        undelete (level->items, level->time_stamp, item_list [i]) ;
                        hit_mem_slot (level, level->items [item_list [i]].avail - 1) ;
                        ++count ;
                }
        }

/* if not all the items are here, will have to retrieve some */
        for (i = 0 ;  i < num && backed_up ;  ++i) {
                if (level->items [item_list [i]].avail == 0) {
                        retr_item (dev_id, level, item_list [i], &status) ;
                        if (!status) ++count ;
                }
        }

/* if were unable to retrieve all the desired items, print a message to the user */
        if (count < num) {
                screen_clear () ;
                set_cursor (3, screen_height - ((screen_height - (num - count + 2)) >> 1)) ;
                printf ("       Not all retrieves completed -- host system is down.\n") ;
                printf ("       The following items were not retrieved :\n") ;
                for (i = 0 ;  i < num ;  ++i) {
                        if (!level->items [item_list [i]].avail)
                                printf ("              %s\n", level->items [item_list [i]].name) ;
                }
                user_pause (touch_id) ;
        }
        return (count) ;
}
```

```
#
/* undelete -- This procedure undeletes an item previously deleted.  The parameters are the item_tags
   structure for the items in this level, the time_stamp array for this level, and the number of
   the item to be undeleted.
*/

#include "/mnt/deb/thesis/structures.incl

undelete (items, time_stamp, item_num)
struct  item_tags  *items ;                     /* list of items and their availability flags */
int     time_stamp [] ;                         /* headers associated with the memory slots */
int     item_num ;                              /* number of the item to be undeleted */
{
int     temp ;

        temp = items [item_num].avail ;         /* find the index of the slot occupied by the item */
        if (temp < 0) temp = -temp ;            /* if the item was deleted, need to get the actual slot index */
        time_stamp [temp - 1] = 1 ;             /* mark the appropriate slot as being used */
        items [item_num].avail = temp ;         /* mark the item as available */
        return ;
}
```

```
#
/* user_command -- This procedure presents the user with all the currently valid commands, and returns
   the index of the command button touched.  (A command is valid if its avail flag is set or if the master flag is true.)

   Uses :  activate, deactivate, flush, get command, printf. screen_clear, set_cursor
*/

#include "/mnt/deb/thesis/defines.incl
#include "/mnt/deb/thesis/structures.incl


user_command (file_id, num_buttons, buttons, intl_tags, master, num_msg, x_pos, y_pos, msg)
int     file_id ;                               /* id of file to read user's response from */
int     num_buttons ;                           /* number of buttons in list */
struct  item_tags  *buttons ;                   /* provides the avail flags for the buttons */
int     *intl_tags ;                            /* internal button identifier tags */
int     master ;                                /* master flag to be ORed with avail tags in determining
                                                   whether a button should be activated */
int     num_msg ;                               /* number of textual messages to be printed on this page */
int     x_pos [] ;                              /* x co-ord of starting position for lines of text */
int     y_pos [] ;                              /* y co-ord of starting position for lines of text */
char    *msg [] ;                               /* text for messages */
{
int     butt_tags [max_num_buttons] ;           /* tags for buttons actually to be activated */
int     num,
        count ;                                 /* number of buttons to be activated */

        count = 0 ;
        screen_clear () ;

/* make the list of valid command buttons */
        for (num = 0 ;  num < num_buttons ;  ++num) {
                if (buttons [num].avail || master) {
                        butt_tags [count] = intl_tags [num] ;
                        ++count ;
                }
        }

/* if there is at least one valid command  present the options and get the user's response */
        if (count > 0) {
                activate (count, butt_tags) ;
                for (num = 0 ;  num < num_msg ;  ++num) {
                                                /* print the lines of text that need to appear */
                        set_cursor (x_pos [num], y_pos [num]) ;
                        printf ("%s", msg [num]) ;
                }
                flush (file_id, &num) ;         /* throw away any leftover extraneous touches */
                num = get_command (file_id) ;
                deactivate (count, butt_tags) ;
                screen_clear () ;
        } else num = -1 ;

        return (num) ;
}
```

```
/* user_pause -- This procedure presents the user with a button marked "Proceed", and waits until
   that button is touched.  It allows the user to indicate that the current display is no longer
   needed and execution can continue.

   Uses :  activate, add_command, del_command, get_command, lite_box, user_pause
*/

user_pause (touch_id)
int     touch_id ;              /* device id for the touch panel */
{
int     button [1] ;

        button [0] = add_command ("Proceed", 12, 0, 3, 1, 35) ;
        activate (1, button, touch_id) ;
        while (get_command (touch_id) != 35) ;
        lite_box (button [0], 0) ;
        del_command (1, button) ;
        return ;
}
```

APPENDIX B

Front-End Data Structures

```
struct item_tag {
        char    *name ;
        int     avail ;
}
```

1. name — Pointer to a character string which is the name of this item/button. This string is used for labeling various displays and buttons presented to the user.

2. avail — Integer flag indicating whether this is a valid button if the host is unavailable. A value of 0 means that this button is only valid if the backup host is available; non-zero indicates validity even if the host is not available. Generally, this flag is a simple 1 or 0. For data items, extra information is stored in this flag. (See discussion of the 'items' component of the level_variables structure.)

```
struct level_variables {
        int             num_items ;
        struct item_tag *items ;
        char            **item_j_name ;
        int             num_elements ;
        char            **elt_name ;
        int             num_arrays ;
        int             *mem_array ;
        int             *time_stamp ;
        char            *dsn ;
        int             num_displays ;
        int             item_resident ;
}
```

1. num_items The number of items in this data base. (Integer)

2. items Pointer to an array of item tag structures for the items. Each item in the data base has an item tag structure associated with it. This structure contains the name of the item and its availability flag. The the avail flag is 0 if the item is not locally resident, negative if it is resident but marked as deleted, and positive if it is resident and not deleted. If the avail flag is non-zero, the value of the flag indicates which memory slot contains the item's value.

3. item_j_name Pointer to the list of character strings which are the item names as used in the Janus system. This is used in formulating the request for data transmition from the host system.

4. num_elements The number of data elements in each item in this data base. (Integer)

5. elt_name Pointer to the list of character strings which are the element names for this data base. This list is used for labeling the element values in table and bar graph displays.

6. num_arrays Number of slots in the local memory for this data base. Used in keeping track of the amount of local storage available. (Integer)

7. mem_array Pointer to a list of integers which are the starting addresses of local memory slots. Used when referencing or retrieving the values of a data item.

8. time_stamp Pointer to a list of time stamps (integers) associated with the local memory slots. The time stamp associated with a slot indicates the last time the contents of that slot were referenced. These times are used to determine which item is replaced when a new item is retrieved from the remote host.

9 dsn — Pointer to a character string which is the name of the current data base. Used in formulating the request for data from the remote host and for messages to the user. For the Illinois data base, this is the word "Illinois"; for a county data base, it is the name of the county (e.g., "Cook").

10. num_displays — The number of types of data displays available for items in this data base. For the Illinois data base it is three (table, bar chart, and shaded map); for the county data bases it is two (table and bar chart). (Integer)

11. item_resident — Integer boolean flag indicating if any data for this data base is resident in the terminal. Used for determining when certain buttons represent valid commands even if the host is unavailable.

```
/* This is the include file /mnt/deb/thesis/demo text.incl */

/* the list of county names */
char    *county_names [num_counties]
                {"Boone", "Bureau",  Carroll", "Cook", "DeKalb",
                "DuPage", "Grundy", "Henderson", "Henry",
                 JoDaviess", "Kane", "Kankakee", "Kendall", "Knox",
                "LaSalle", "Lake", "Lee", "Marshall", "McHenry",
                "Mercer", "Ogle", "Putnam", "Rock Island", "Stark",
                "Stephenson", "Warren", "Whiteside", "Will",
                "Winnebago"} ;

/* list of town names by county */
char    *town_names [num_counties] [max_num_towns]
                {"Belvidere", "", "", "", "",
                "Spring Valley", "Princeton", "", "", "",
                "Savanna", "", "", "", "",
                "Barrington Hls", "Arlington Hts", "Hanover Park",
                        "Niles", "Winnetka",
                "Genoa", "De Kalb", "Sandwich", "Sycamore", "",
                "Villa Park", "Hinsdale", "Glen Ellyn", "Lombard",
                        "Bloomingdale",
                "Coal City", "Morris", "", "", "",
                "", "", "", "", "",
                "Geneseo", "Green Rock", "Galva", "Kewanee", "",
                "Galena", "", "", "", "",
                "S Elgin", "N Aurora", "Elgin", "St Charles",
                        "Carpentersville",
                "Bradley", "Borubonnais", "Momence", "Manteno",
                        "Kankakee",
                "Plano", "", "", "", "",
                "Abingdon", "Knoxville", "Galesburg", "", "",
                "Peru", "Ottawa", "Mendota", "La Salle", "Oglesby",
                "Round Lk Bch", "Park City", "Round Lk Pk",
                        "N Chicago", "Mundelein",
                "Dixon", "", "", "", "",
                "Henry", "Toluca", "", "", "",
                "Woodstock", "Crystal Lake", "Harvard", "Algonquin",
                        "Lake in the Hls",
                "Aledo", "", "", "", "",
                "Mt Morris", "Polo", "Rochelle", "Oregon", "",
                "", "", "", "", "",
                "Rock Island", "Coal Valey", "Silvis", "Moline",
                        "East Moline",
                "", "", "", "", "",
                "Freeport", "", "", "", "",
                "Monmouth", "", "", "", "",
                "Morrison", "Rock Falls", "Fulton", "Sterling", "",
                "Romeoville", "Steger", "Crest Hill", "Bolingbrook",
                        "Plainfield",
                "S Beloit", "Rockford", "Loves Park", "", ""} ;
```

```
/* list of state item names for the user */
char    *st_itm_names [num_state_items]
                {"Population", "Pop/Sq Mile", "% Pop Change",
                "% Female", "% Urban", "% Chng Negro", "% Pop < 5",
                 % Pop > 18", "% Pop > 65", "Negro Pop", "% Spanish",
                "Birth Rate", "Death Rate", "Pres Votes",
                "Land Area", "Families", "% Low Income", "Med Fam Inc",
                "Syphilis", "Gonorrhea", "Bank Deps",
                "M. H. Admiss", "M. H. Pop", "Gas Stations",
                "Suicide Rate", "Nat Gas Prod", "# Streams",
                "Miles Stream", "Acres Stream"} ;

/* list of state item names for janus */
char    *st_j_itm_names [num_state_items]
                {"cc72002", "cc72003", "cc72004", "cc72006",
                "cc72007", "cc72010", "cc72011", "cc72012",
                "cc72013", "cc72009", "cc72017", "cc72018",
                "cc72019", "cc72005", "cc72069", "cc72070",
                "cc72071", "cc72075", "vs72001", "vs72002",
                "cc72041", "mh72002", "mh72010", "bu67023",
                "vs70027", "mn72001", "cons72001", "cons72002",
                "cons72011"} ;

/* list of county item names for the user */
char    *c_itm_names [num_cnty_items]
                {"Tot Pop", "Pop < Age 15", "Pop > Age 65", "Pop
                Density", "Med Fam Inc", "1970 Births", "1970
                Deaths", "Mean Income", "AM Stations", "FM
                Stations", "TV Stations", "Interstate", "US
                Highways", "Sales Tax", "Tax Rate", "SO2 Emission",
                "Polluters"} ;

/* list of county item names for janus */
char    *c_j_itm_names [num_cnty_items]
                {"var201", "var202", "var204", "var216", "var224",
                "birth70", "death70", "income69", "amradio",
                "fmradio", "tv", "tra256", "tra257", "tax235",
                "tax245", "so2emis", "nopollut"} ;
```

APPENDIX C

Front-End Support Software

activate

Description : Every button in the input list of buttons is activated. This includes displaying the button on the screen and marking its entry in the button table as active.

Calling Sequence : activate (number, button_list)

1. number Number of buttons to be activated.
2. button_list Array of pointers into the button definition table. Contains pointers to the entries for the buttons which are to be activated.

---

add_command

Description : The characteristics of a button are entered into the button definition table. A pointer to this entry is returned to the calling procedure. This pointer is used by other button-handling routines to reference this button. The button is not displayed and is initially inactive.

Calling Sequence : new_button_pointer = add_command (button_label, x_position, y_position, x_size, y_size, internal_tag)

0. new_button_pointer Pointer to the table entry for the newly created button. This pointer is passed to other button handling routines to access this button.
1. button_label Character string to be used to label the button when it is displayed.
2. x_position, y_position The co-ordinates on a 16 x 16 touch grid of the lower left corner of the button.
3. x_size, y_size The dimensions, in touch grid units, of the button.
4. internal_tag Number to be associated with the button. This tag is used to communicate to other procedures which button was touched by the user.

bar_graph

Description : The input data element values are displayed as a bar graph. The values are displayed as horizontal bars labeled on the left with the element name and on the right by the numeric value of the element. The bars are scaled so that the smallest element value becomes relative zero and the bar for the largest value fills the specified area, leaving room for labels. All other values are scaled linearly between these two values. Elements with values which inidcate that they are "missing data' are displayed as a single asterisk instead of as a bar.

Calling Sequence : bar_graph (origin_x, origin_y, size_x, size_y, number_values, element_labels, element_values, title)

1. origin_x, origin_y The co-ordinates, on a 64 x 32 character area grid, of the lower left corner of the space to be occupied by the graph. Space 0 of line 0 is in the lower left corner of the screen.
2. size_x, size_y The size, in character spaces, of the area to be occupied by the graph.
3. number_values Number of element values to be displayed.
4. element_labels Array of pointers to character strings to be used to label the bars of the graph.
5. element_values Array of values to be representated as bars on the graph.
6. title Pointer to a character string to be used to label the entire graph.

---

close

Description : Used to relinquish control of an owned device.

Calling Sequence : close (device_id, return_status)

1. device_id Internal identifier of the device to be closed.
2. return_status Pointer to a return status word. Used to indicate the success of type of failure to the calling procedure.

clr line

Description : Erases a specified number of character lines on the screen, and positions the cursor at the left of the topmost cleared line.

Calling Sequence : clr line (number lines, top line number)

1. number_lines Number of lines, each one character space high, to be erased.
2. top_line_number Number of the topmost line to be erased. Line Ø is at the bottom of the screen.

---

deactivate

Description : The table entry for each button in the input list is marked inactive. The buttons are not erased from the screen.

Calling Sequence : deactivate (number, button_list)

1. number Number of buttons to be deactivated.
2. button_list Array of pointers into the button definition table. Contains pointers to the entries for the buttons which are to be deactivated.

---

del_command

Description : Deletes a set of buttons from the internal button table. Takes a list of pointers into the table (returned by add_command when the buttons were created), and makes those slots available.

Calling Sequence : del_command (number, button_list)

1. number The number of buttons to be deleted.
2. button_list Array of pointers into the button definition table. Contains pointers to the entries for the buttons which are to be deleted.

flush

Description : Allows buffered input from a device to be discarded.

Calling Sequence : flush (device_id, return_status)

1. device_id Internal identifier of the device to be flushed.
2. return_status Pointer to a return status word. Used to indicate the success or type of failure to the calling procedure.

---

get_command

Description : Reads a button touched by the user. As a touch-interrupt occurs, the co-ordinates of the touch are checked to find which (if any) active button encompasses the area hit. If such a button is found, the tag of that button is returned to the caller. If not, the touch is ignored, and the routine waits for the next hit. This routine handles flashing a touched button.

Calling Sequence : get_command (touch_panel_id)

1. touch_panel_id Internal identifier for the touch panel. Used to read the co-ordinates of the touch.

get_janus values

Description : Retrieves an item from the host system. Uses the Janus names for the data base and the item to formulate a request for data to the host. If all is well, software at the host will ship back a data item in response. After sending out the request, get janus values reads from the phone line, and interprets the incoming data as an item. If the data is determined to be in valid data item format, the item values are converted intp PDP11 internal format and stored in the indicated memory slot.

A return status word is set according to :

- 0 : all went well
- 1 : host went down before any data was transmitted
- 2 : host went down during transmition
- 3 : garbled transmition

Calling Sequence : get janus values (phone_id, data_base, item_janus_name, memory_slot_address, size_of_item, return_status)

1. phone_id — Internal device identifier of the phone.
2. data_base — The Janus name of the data base from which to get the item. This is the same as the county name for county data bases, and is the string "Illinois" for the state data base.
3. item_janus_name — The Janus name for the desired item.
4. memory_slot_address — Pointer to the beginning of the memory slot where the new data item is to be put.
5. size_of_item — Number of elements in the item. Used to make check for possibly garbled transmition.
6. return_status — Pointer to a return status word. Used to indicate the success or type of failure to the calling procedure.

get_touch

Descritpion : Formats a list of labels and button tags into buttons, and allows the user to choose up to a specified number of them. The buttons are displayed centerd as a group on the screen, with Cancel and Continue buttons in the lower left and right corners. An explanitory message is dispalyed near the bottom of the screen.

Three actions are taken when the user touches a button. The tag of that button is added to the list of touched buttons to be passed back to the caller. The button is marked by a small box in the lower right corner to signify that the button has been chosen. And the user message is updated. Since this message is often of the form "Choose <n>", the value of n is updated after every touch to reflect the number of choices left.

get_touch will only allow the user to pick up to a specified number of buttons. After this number of buttons has been chosen, the message is changed to "Hit Proceed to continue." If more buttons are chosen after this time, the initially chosen buttons are removed from the list and their marking boxes are erased.

A selected button can be "un-selected" by touching it again. A button is counted as hit iff it was touched 1 mod 2 times.

The current command can be aborted at any time by touching the Cancel button.

Before get_touch returns, it clears the screen and removes all the buttons it created.

Calling Sequence : number_chosen = get_touch (touch_panel_id, button_labels, button_tags, chosen_button_list, number_buttons, message, maximum_number_to_be_chosen)

0. number_chosen — The number of buttons chosen by the user. If the Cancel button was hit, then this value is -1.
1. touch_panel_id — The internal identifier for the touch panel. Used to pass to get_touch.
2. button_labels — List of labels to associate with the buttons which are created.
3. button_tags — List of internal tags to be associated with the buttons.
4. chosen_button_list — Pointer to the array where the caller wants the tags for the chosen buttons to be stored.
5. number_buttons — The number of buttons to be created.
6. maximum_number_to_be_chosen — The maximum number of buttons which the user is to be allowed to chose.

list lth

Description : Determines the length of a list of character strings. Given the maximum number of strings in the list, the procedure scans the array of pointers to character strings and returns the index of the first null pointer.

Calling Sequence : length = list_lth (pointer_array, array_size)

0. length Number of character strings in the list, up to the maximum (array size)
1. pointer_array Array of pointers to character strings.
2. array_size Size of array -- maximum possible size of list.

---

lite_box

Description : Lights or erases all the dots inside a button.

Calling Sequence : lite_box (button_pointer, mode)

1. button_pointer Pointer into the button definition table of the desired button. Used to determined the placement and size of the button.
2. mode Determines whether the area is lit or erased. 1 for light, 0 for erase.

---

map

Description : Displays an Illinois data item as a shaded map. Draws an outline map of the northern 29 counties of the state of Illinois, then shades the counties according to the value of the data item for each county. The map is labeled with the item name under the map, and the shades are explained via a legend at the bottom of the display.

Calling Sequence : map (number_shades, item_values, item_size, item_name)

1. number_shades Number of different shades to use on the map. Must be between 1 and 7.
2. item_values Pointer to the array of values for the item to be displayed.
3. item_size Number of values in the item.
4. item_name Pointer to the name of the data item. Used to label the map.

open

Description : Allows a process to acquire possession of a device.

Calling Sequence : device id = open (device code, block flag, return_status)

0. device_id Internal logical identifier to be associated with this device. Used to communicate with other procedures which need a device id.
1. device_code System wide code number associated with the device.
2. block_flag If this value is non-zero, the open will block (wait) until the requested device is available.
3. return_status Pointer to a return status word. Used to indicate the success or type of failure to the calling procedure.

---

open_ph

Description : Special opening routine for the phone line. Includes extra code to handle problems associated with synchronizing the request to open with the physical process of establishing a carrier signal on the phone line.

Calling Sequence : phone_id = open_ph ()

0. phone_id Internal logical identifer of the phone line.

---

pr_help

Description : Prints an explanitory paragraph about one of the commands of the front-end system. Using the file name of the help text on the host, pr_help formulates a request to the host. In response, the host ships the requested file. pr_help then strips of the leading information and prints everything up to the trailer on the screen.

Calling Sequence : pr_help (phone_id, help_file)

1. phone_id Internal identifier of the phone line.
2. help_file Pointer to the character string which spells the name of the help file on the host system.

printf

Description : Prints one constant string with an arbitrary number of parameters. Parameter replacement in the constant string is signified by "%x", where x is one of c (character), s (character string), o (octal), d (decimal), or l (unsigned decimal). The starting position of the printing is determined by the position of the "cursor". The cursor is moved by the action of printing each character, and can be set explicitely by the routine set_cursor. (The position of the cursor is remembered internally -- it is not displayed.)

The screen contains 32 character lines of 64 spaces each. Character Ø of line Ø is in the lower left corner of the screen.

Calling Sequence : printf (format, p1, p2, p3, ...)

1. format Pointer to the constant format string.
2. p1, p2, p3, ... Parameters to the format string.

---

read

Description : Handles reading from an arbitrary device.

Calling Sequence : read (device_id, buffer_pointer, buffer_length, return_status)

1. device_id Internal identifier of the device to be read from.
2. buffer_pointer Pointer to the buffer where the input is to be placed.
3. buffer_length Size of the buffer -- the maximum number of characters to be read.
4. return_status Pointer to a return status word. Used to indicate the success or type of failure to the calling procedure.

---

screen_clear

Description : Erases all the dots on the screen.

Calling Sequence : screen_clear ()

set_cursor

Description : Positions the cursor on the screen. Used in connection with printf.

Calling Sequence : set_cursor (x, y)

1. x — Horizontal co-ordinate of the new cursor position. Each line has 64 spaces, with space 0 being on the left.
2. y — Vertical co-ordinate of the new cursor position. The screen has 32 lines. with line 0 being on the bottom.

---

str_lth

Description : Determines the length of a (null-terminated) character string.

Calling Sequence : length = str_lth (string)

0. length — Length of the string.
1. string — Pointer to the character string.

---

table

Description : Displays up to 3 data items in a tabular format. The table is centered on the screen. The items are presented in columns, separated by line. The columns are labeled at the top with the item name, and on the left side of the table with the element names. Every third line is underlined with a dotted line for readability.

Calling Sequence : table (item_names, number_items, element_names, item_size, item_values)

1. item_names — Array of pointers to the names of the items to be displayed. Used to label columns.
2. number_items — Number of items to be displayed.
3. element_names — Array of pointers to the names of the elements in these items. Used to label rows.
4. item_size — Number of elements in the items.
5. item_values — Array of pointers to the values for the items to be displayed.

terminal

Description : Simulates an ordinary ASCII terminal. Reads from both the keyboard and the phone line. Input from the keyboard is shipped down the phone line and echoed on the screen. Input from the phone is printed on the screen. This routine handles character and line delete and prevents characters from the phone from being interspursed with text from the keyboard.

Calling Sequence : system_status = terminal (phone id address)

0. system_status Return value is 1 if the host system is available, and 0 otherwise.
1. phone_id_address Pointer to the internal identifier of the phone line.

---

write

Description : Handles writing to an arbitrary device.

Calling Sequence : write (device_id, buffer_pointer, buffer_length, return_status)

1. device_id Internal identifier of the device to be written to.
2. buffer_pointer Pointer to the beginning of the character string to be written.
3. buffer_length Number of characters to be written, including the terminating null.
4. return_status Pointer to a return status word. Used to indicate the success or type of failure to the calling procedure.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CAC Document Number 186
CCTC-WAD Document Number 6502

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Humanizing Data Management Systems:
An Intelligent Terminal Approach

5. TYPE OF REPORT & PERIOD COVERED: Masters Thesis

6. PERFORMING ORG. REPORT NUMBER: CAC #186

7. AUTHOR(s): Deborah Sue Brown

8. CONTRACT OR GRANT NUMBER(s): USDefense DCA 100-76-C-0088

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Command and Control Technical Center
WWMCCS ADP Directorate, 11440 Isaac Newton Sq., N.
Reston, VA 22090

12. REPORT DATE: January 1976

13. NUMBER OF PAGES: 89

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Copies may be obtained from the

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restriction on distribution

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Intelligent terminals

Touch input

Data management systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis discusses the implementation of an intelligent terminal with touch input to provide an improved user interface to an existing data management system. General design goals and the generic solution used are presented. A specific terminal system is discussed in detail.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUC-CAC-DN-186	2.	3. Recipient's Accession No.

4. Title and Subtitle

Humanizing Data Management Systems:

An Intelligent Terminal Approach

5. Report Date: January 1976

6.

7. Author(s)

Deborah Sue Brown

8. Performing Organization Rept. No. CAC #186

9. Performing Organization Name and Address

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

USDefense
DCA 100-76-C-0088

12. Sponsoring Organization Name and Address

Command and Control Technical Center
WWMCCS ADP Directorate, 11440 Isaac Newton Sq., N.
Reston, VA 22090

13. Type of Report & Period Covered

Masters Thesis

14.

15. Supplementary Notes

16. Abstracts

This thesis discusses the implementation of an intelligent terminal with touch input to provide an improved user interface to an existing data management system. General design goals and the generic solution used are presented. A specific terminal system is discussed in detail.

17. Key Words and Document Analysis. 17a. Descriptors

Intelligent terminals

Touch input

Data management systems

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

No restriction on distribution
Available from the Center for Advanced Computation
Univ. of Illinois at Urbana-Chanpaign, Urbana, IL61801

19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 89
20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (REV. 3-72)

USCOMM-DC 14952-P72