To: Distribution

From: James R. Davis

Date: 11/14/80

Subject: Menu Software in MR 9

This MTB gives plans for the MR 9 Menu Software. This software satisifies PFS item 4.5.2. In addition, the Executive Mail Application (PFS 4.5.3) will be implemented using this software.

This MTB gives a brief description of the user interface provided by the menu system, an informal menu application programmers guide, a description of the commands comprising the menu system, and discusses the defects of the current system, and what might be done to correct them.

Send comments to:

The System M continuum >udd>m>jrd>mtgs>menus

by Multics Mail on System M or MIT: JRDavis.Multics

by phone: (617) 492-9382
HVN 261-9382

1. CONTEXT

The MR 9 menu system is a re-implementation of the prototype menu system. The prototype menu system is described in MTB 446. The chief novelty of the MR9 menu system is that it does not provide its own video support, but instead relies on the Multics Video System.

The menu system consists of three related programs.

menu_manager
: displays menus to users and executes command lines based on the response. The menu_manager command interprets a menu data segment that describes the menus and actions.

cv_sd
: creates menu data segments from menu source segments.

display_sd
: displays menu data segments, for verification of their contents. (This is an auxiliary function, not required for normal operation.)

2. USER INTERFACE

The menu system provides a mechanism whereby users can be presented a "menu" or a list of alternative actions to be taken. Users are restricted to a choice from the menu. Each option in the menu is described by a short "label", and the details of implementation are hidden from the user.

The menu system is used only on video terminals. When in use, the screen is divided into two regions or "windows", one above the other, each occupying the full width of the terminal. The top window (the menu window) contains a menu, and the bottom contains output. Occasionally the application will ask a question in the bottom window, and the user types in a response.

A menu appears to be a list of one or more options, arranged in one or two columns. Each option has a title or label, describing the action to be done if the option is chosen, and a name, which is a single digit or letter, enclosed in parenthesis, to the left of the title. Above the list of options are the header lines, which identify the menu (and application), below the list of options are the trailer lines, which separate the menu from the bottom window. Header and trailer lines are optional.

Normally the cursor (whose appearance is terminal dependent, but usually a blinking box or underline) is in the menu window, indicating that the menu system is ready for the user to make a selection from the menu.

The user picks from the menu by typing the name (the letter or number) of the option, and some action is performed. The action may invoke a command or exec_com, or a new menu may be displayed, with further choices. As actions are performed, they may ask the user for input by prompting in the lower window. Then, the cursor appears in the lower window, and the user types in the desired information. Typing in the lower window is through the real-time input line editor of the Multics Video System - thus the erase and kill characters take immediate effect.

When an action displays a new menu of choices, the old menu is remembered. The user can return the the previous set of choices, or to the first menu. In effect, the user is traversing a tree structure, similar to the Multics directory structure. (The structure of choices may be more complex than a tree, but that can be deferred for now.)

Other than selecting options, the user may strike:

- Function key 1 (F1).
  to get help. Typing F1 followed by the number or letter associated with a menu option will cause help information about that option to be displayed in the bottom window. If Function key 1 is pressed twice consecutively, a file giving general help information about the particular subsystem being used is printed. Help is only available if the definer of the application provided it.

- Function key 2 (F2).
  to return to the first menu.

- Function key 3 (F3).
  to return to the previous menu.

- Function key 4 (F4).
  To exit the menu manager. This will cause the menu manager to return to its caller. This option may be disabled, to provide a closed subsystem.

- Function key 5 (F5).
  To escape to full Multics command level in the bottom window. This option may be disabled, to provide a closed subsystem.

- CLEAR/RESET (F0).
  To redisplay the menu. This option is useful if the screen was messed up due to transmission line or software problems.

Any character not in the above list is rejected and ignored (the terminal beeps).

Terminals Without Function Keys

If a terminal does not have function keys, or does not have enough function keys, the following sequences may be used instead of the function keys:

ESC-h	(HELP)
ESC-f	(First menu)
ESC-p	(Previous menu)
ESC-a	(Abort. Quit the menu manager)
ESC-q	(Quit. Same as Abort)
ESC-e	(Escape to Multics)
ESC-m	(Menu Redisplay)

Quit Processing

If the break key is hit while the cursor is in the menu portion of the screen, no action is taken. If the cursor is in the bottom window when the break key is hit, whatever action was being performed is aborted and the cursor is moved into the menu portion of the screen.

3. APPLICATION WRITERS GUIDE

A Menu Application consists of a menu or menus (in a menu data segment), and usually some programs (in exec_com or some other language) and help files. The remainder of this section describes the language used to define menus.

Menus are defined in a simple, keyword based language. Every statement in the language begins with keyword and ends with a semi-colon. Most keywords are followed by a colon and some value. Often the value is a quoted string.

Menu definitions consist of global statements, format statements, and screen definitions, freely mixed, in any order. Global statements pertain to the entire menu segment. Format statements control the appearance of subsequent screens. Screen definitions describe individual menus and their actions. Comments may also appear, bracketed with "/*" and "*/" as in PL/I.

3.1. Global Statements

The two global statements are the "Start_up" statement and the "General_info" statement.

The Start_up statement specifies a command line that is executed when the menu_manager is first invoked on the menu.

example:
```
     Start_up: "format_line ""Welcome to the EXL mail  system""";
     Start_up: "ec create_dir_if_needed  user name ";
```

The General_info statement supplies information about the entire application being defined. This is the information printed when the HELP key is struck twice in succession.

If the string begins with either ">" or " rd >", it is interpreted as a pathname and the segment indicated is printed. (rd is an abbreviation for the directory including the screen object segment).

example:
```
     General_info: " rd >generic_info";
     General_info: "This is a terse info.";
```

3.2. Format Statements

Format statements control the appearance of subsequently defined menus. Format statements control the number of columns used in the list of menu items, and the appearance and number of header and trailer lines.

The Columns keyword is followed by the number of columns. There may be one or two columns, and the default is one.

example:
```
     Columns: 1;
```

The menu can contain header and trailer lines. These lines appear above and below the menu, respectively. A header is commonly used to identify the menu to the user, and a trailer to separate the menu from the bottom window.

Up to ten headers and ten trailers may be displayed. A header or trailer line must fit on a single line of the terminal. One of the limitations of the window system is that there is no way to know the terminal width when the menu is defined.

Headers are specified by a Header statement, and trailers by a Trailer statement. These statements have the same syntax:

```
    Header  (N) : STRING;
    Trailer  (N) : STRING;
```

where

N
: is an integer from 1 to 10, and specifies the number of the header of trailer. If it is omitted it defaults to 1, and the parenthesis must also be omitted. Lines are displayed in numeric order.

STRING
: is a quoted string. A null string ("") will cause that line to appear as a blank line except that if all headers or trailers after a given one are blank, none of these are displayed. That is, the last non-null header or trailer is the last one to be displayed.

Header or Trailer statements cannot appear between a "screen" statement and the corresponding "end" statement. A header or trailer definition applies to all subsequent screens unless changed by another subsequent statement.

Header or Trailer statements may contain active function references, i.e., strings that are bracketed by " ... " sequences. Such active functions are expanded at screen display time, in such a way that any white space in the line is turned into a single space.

example:

```
    Header (1): "-------- MAIL SYSTEM --------";
    Header (2): "--- reading:  valf mbx-name  ";
    Trailer:    "-----------------------------";
```

3.3. Screen Definitions

The screen statement is used to define a menu. The menu will consist of the currently defined headers followed by one or more columns of options followed by the trailers.

A screen definition begins with a "screen" statement, and ends with an "end" statement. Everything in between these two statements defines the screen.

Each screen definition must have at least one option statement. The option statement specifies the text of the string to be placed in the menu (the label). The maximum length of this string depends upon the width of the terminal and the number of columns.

example:

```
      screen: trial_screen;
       option:  "Instruct the Jury.";
         .
         .
         .
       option: "Perjure.";
         .
         .
     end;
```

Each option must have at least one action. The action of an option can be a transfer to another screen (specified by a "next_screen" statement) or a command line to be executed (specified by a "handler" statement). If the action transfers to another screen it may also have a screen_start_up statement. Options may also have a help statement.

The handler Statement

The handler statement specifies the command line to execute if the corresponding menu option is selected.

example:

```
     handler: "ls -a -sort name";
     handler: "discard_output dprint  segs  *.info ";
```

The next_screen Statement

This statement is used to indicate to the menu_manager that a new menu is to be displayed whenever this option is selected. The keyword "next_screen" must be followed with the name of another screen defined in the menu definition segment.

example:
```
     next_screen: dir_screen;
```

The screen_start_up Statement

The screen_start_up statement is used to specify a command line to be executed before the new menu is displayed. Often several options will use a common next_screen. This statement is used to customize the environment before the next screen is entered. For example, it can be used to set variables which are effectively input parameters to the handler. Another use of the screen_start_up is to display information in the bottom window that can be left there after the new menu is displayed.

example:
```
     screen_start_up: "setf mbx-name  user name .mbx"
```

The help Statement

The help statement specifies the text of the message to print if the help function key (or equivalent) is input while the cursor is in the menu. As in the "General_info" statement, the value of this keyword may be a literal string or the pathname of a file.

example:
```
     help: "lists all files containing named string";
     help: " rd >string-search-list.info";
```

3.4. Returning Responses

One of the most powerful features of the menu system is the ability of the menu writer to specify strings to be returned to the request loop of an interactive subsystem. This feature must be carefully used, since the writer must supply full and correct request strings for a (possibly changing) subsystem. There is no provision for error analysis or recovery.

A subsystem invoked in this manner must either not prompt, or there must be a way to shut the prompt off, otherwise the subsystem prompt will appear in the lower window.

It is extremely desirable that the subsystem have an "execute" request allowing active functions to be evaluated from the request loop, with their values returned into the request loop, so that the string returned can contain references to variables and user supplied values.

When a subsystem is used in this way, the menu manager is said to be in "menu_input" mode. In this mode, the I/O switch user_input is attached to a special I/O module which returns the strings defined in the menu. These strings are defined by the "response" keyword.

The "response" statement specifies the string to be returned to the subsystem. The value after the keyword is a quoted string, which is returned to the subsystem. It can contain anything meaningful to the subsystem. The example below is taken from a mail reading menu.

example:
```
     response: "list /|| e response ""string to search for:"" /";
```

When an option has a next_screen, menu_manager examines the screen specified. If it contains any "response" keywords, then the "menu_input" mode is entered. It is assumed that the handler of the selected option is about to invoke an interactive subsystem. The subsystem must obtain its input only by iox_get_line calls on user_input.

Unlike commands, subsystems usually only return when some explicit "quit" request is given. This can be a problem, because a user of a menu application does not know that a subsystem is being used, and cannot be expected to provide a quit request when finished with the subsystem. The menu user expects to exit with one of the function keys (F2, F3, or F4, or the equivalent). Therefore, the writer must supply a string which, when returned to the subsystem, will cause it to return to the command processor. This is done by the "quit" keyword. When the user signals his or her desire to exit the menu, this string is returned to the subsystem, causing it to return to the handler which originally invoked it. If this keyword is not supplied, the subsystem is exited by a non-local goto, which may or may not have desirable effects.

example:

```
    quit: "quit -force";
```

3.5. Example

```
  Start_up: "format_line ""Sample  Application.""";
  General_info: "This is a terse info.";

  Columns: 1;
  Header (1): "--- Sample:  date  ---";
  Trailer:    "----------------------";

  screen: top;
    option: "List Files";
     handler: "list -sort -name";
     help:    "shows list of all files, in alphabetic order";
   option:  "Delete File";
     handler: "delete  response ""What File to Delete:"" ";
     help:    "asks you for a file name, then deletes it.";
   option:  "Edit a file using TED";
     handler: "ted -pn  response ""File to Edit:"" ";
   option:  "Read Your Mail.";
     handler: "read_mail -no_prompt";
     next_screen: read_mail_menu;
     cleanup: "quit -force";
     screen_start_up: "";
  end;

  Header: "--- READ MAIL ---";
  screen: read_mail_menu;
   option: "List Contents.";
     response: "list all";
     help: "Prints a list of all messages in your mailbox.";
   option: "Print a Message";
     response:  "e  mm$bottom_window_input;  e  setf   msgno    e
response Msg: ; e mm$menu_input; print  e valf msgno ";
  end;
```

4. ASSISTANCE FOR APPLICATION PROGRAMS

Programs will run without modification under the menu system, but a more pleasing user interface can be created by taking advantage of the menu system. bThe first principle of a menu program is that all its I/O takes place in the bottom window. The bottom window has limited size, and is cleared before each action is performed.

There are several entry points into the menu_manager command which may be called from application programs (exec_coms or PL/I procedure) to regulate use of the screen. These entry points take no arguments, so they may be called as commands or as subroutines. The entry points are listed in the description of the menu_manager command.

5. RESTRICTIONS AND DEFECTS

This section gives some of the restrictions and defects in the MR 9 Menu System, and what might be done to remove or correct them.

The menu_manager command can only use function keys on Honeywell VIP 7801 or 7200 terminals. On all other terminals, the escape sequences given above must be used. Software has been designed to correct this problem, but is not planned for MR9.

The menu manager cannot use the auxilliary port on any terminal other than the Honeywell VIP 7801 or 7200. At this time, there are no plans to add terminal independent support of auxilliary printer ports to Multics.

The menu system is not fully terminal independent in that there are restrictions on maximum length of strings (in titles, in headers and trailers) that depend on line length, and the writer of a menu has to know the line length when writing the menu. So a menu written for a terminal whose line length is 80 will fail on a terminal of lesser line length. Solutions to this problem are under investigation.

Since the bulk of the work of an application is usually done by exec_coms, the usual liabilities of exec_com apply. As it exists today, exec_com has no variables, limited control flow primitives, no debugging facilities, and no facility for error analysis or recovery. Some of these problems are being addressed by MR 9 version II ec.

The worst flaws of the menu system pertain to its use in "menu_input" mode. This mode can only work when the writer provides complete and correct strings to be returned to a subsystem. A subsystem request language is not intended to be a programming language. The writer must anticipate every response the user might need to give. There is no provision for orderly error analysis or correction. This problem is being considered, and a solution will be offered in a future MTB.

The error messages given by cv_sd are too terse.

Another problem is that Multics users have the habit of striking RETURN after their input to Multics. The menu system does not need the RETURN, since it operates in breakall. If the action selected prompts the user for some value (and this is common), the RETURN typed by the user will terminate the request. This problem cannot be fixed without drastic changes to Multics Communications Management.

6. POTENTIAL EXTENSIONS

As the menu system has been used, more and more requests have come in for additional features. These are listed here. None are planned for MR 9. Discussion of these features continues in the continuum meeting cited above.

> The ability to transfer to a menu from any other menu (rather than in a strict top-down tree traversal). Users don't like the delay in travelling down the tree structure. The issue here is how the user can specify the desired menu. The most common proposal is that the user supply the "pathname" (in the form of the names of the intermediate menu selections). This clearly won't do, since the user must then remember strings of digits with no inherent meaning, and also makes any change to the menu possibly "incompatible". Another proposal is that the user supply the name of the screen. This suffers because the screen names are not chosen (by the writer) for mnemonic value. In addition, there may be several ways to get to a given menu (that is, there may be several leaves with the same name).
>
> Menu data segments should be full Multics object segments. This would allow use of commands such as "date_time_compiled", would make it possible to find menus (as "entrypoints") in other segments.

Menu segments should be found using a search list.

Help files should be found using a search list (possibly the info search list). There is no reason to restrict the writer to absolute pathnames.

The menu_manager command should be able to interpret menu definition language, because it is often easier to debug interpreted languages than compiled ones.

The menu definition language needs some additional features. One is a "screen_wrap_up" keyword, which would give a command line to be executed when a screen returned. This might close a data base, or unlock a lock. The "quit" string is not sufficient, since it is only available for a subsystem in "menu_input" mode, and is executed by a subsystem, before the subsystem returns.

The question mark character should be accepted as a help character.

It has been suggested that a form of menu_manager be available that works as the "answer" command does. This command could trap command_query_, and display answers appropriate for that question.

It would be very desirable to make menu selection available to all commands and exec_coms. This could be done by a command/subroutine/AF that took as argument a menu description and returned a description of the item chosen. For command usage, the "response" string might be returned.

One limitation of the menu system is that menus have fixed contents. A command or subroutine interface for menu selection would be more valuable if it were easy to build menus "on the fly". As is, the options are embedded in the quoted strings in the source language.

7. DOCUMENTATION

MPM - style documentation follows. (This documentation describes the Menu System as planned, not as it is today.)

Name: menu_manager, mm

The menu_manager command interprets menu data segments. It requires the Multics Video System in order to run. This command may not be invoked recursively.

Usage

```
     menu_manager path  -control_args
```

where

path
: is the pathname of the screen object segment (created by cv_sd) to be used.

control_args
: are chosen from the list:

-disable STR
: prevents the user from using a specified feature of the menu manager. STR may be "escape" (or "e") or "abort" ("a"). This option may be used more than once in a command line.

Utility Entry Points

There are several utility entry points to menu_manager that enable the user of menu_manager (i.e., the subsystem designer and implementer) to control some of the actions taken. These are described briefly below. These entry points may only be called while there is an invocation of menu_manager active.

Entry: mm_dirty_screen

This entry informs the menu_manager that the contents of the screen have been changed in a way that menu_manager has no control about. The menu_manager program will therefore redisplay the upper window the next time it gains control. This entry is useful with programs such as Emacs which manage the entire screen.

Entry: mm_clear_bottom_window

This entry clears the bottom window.

Entry: mm_defer_clear_of_bottom_window

This entry prevents the contents of the bottom window from being erased for one redisplay of the menu. Usually, the bottom window is cleared whenever the menu changes. This entry point is useful when some information has been displayed that will be useful in a decision to be made by the user. For example, the start_up for a mail reading application can display a summary of mail, call this entry point, and cause it to be preserved so the user may see the full list while choosing which message to read.

Entry: mm_clear_screen_for_use

This entry expands the bottom window to fill all but the top line of the screen. The expanded bottom window is available for output. When the handler returns, menu_manager pauses (by prompting for a NEW-LINE), then restores the original window sizes, redisplaying the original menu, and clearing the bottom window.

Entry: mm_bottom_window_input

This entry changes the mm's input mode to conventional input. It is used to escape from menu_input mode. If the application itself wishes to query the user, it must first ask the menu_manager to exit menu_input mode, otherwise the application will also read from the menu. This can be done by calling the entry point mm_bottom_window_input. For example, in a mail reading application, the user might be prompted for the number of a message to delete:

example:
```
response: "e mm_bottom_window_input; delete   e response  ""What
     message?"" "
```

Entry: mm_menu_input

This entry changes mm's input mode to menu input. It is used only to undo the effect of a call to mm_bottom_window_input. In the example above, mm_menu_input should be called after the response active function, so that the subsystem will continue to read "from the menu".

Entry: mm_enable_local_printer

This entry is called to enable a "local printer" that may be attached to the video terminal being used. The local printer may be used for output only. The printer should be attached to the "auxilliary" port of the terminal, so that it may be controlled from Multics.

The call to enable the local printer should be made prior to establishing the terminal characteristics for the terminal (such as line length, tabbing, etc.). When the output is completed, mm_disable_local_printer should be called to return the terminal to its original state. If any terminal characteristics were changed to do the local printing, they should be reset after calling mm_disable_local_printer. This entry may only be called if the terminal used is a Honeywell VIP7801 or VIP7200 with a local printer.

Entry: mm_disable_local_printer

This entry is used to reestablish the video terminal as the primary input and output device of a multi-device workstation. See the description of mm_enable_local_printer for a description of its use.

Name: cv_sd

The cv_sd command compiles a screen definition source segment creating a screen data segment if no errors are found in the source.

Usage

```
     cv_sd path
```

where path is the name of the screen definition segment to be used as input. The suffix "sd" is assumed but need not be specified. The output segment has the same name as the input segment, with the suffix removed.

Name: display_sd

The display_sd command is used to generate a listing of the contents of a screen definition data segment (created by the cv_sd command). The output of this command is <u>not</u> suitable for input to the cv_sd compiler.

Usage

```
     display_sd path  -control_args
```

where

path
: is the pathname of the screen data segment.

-control_args
: can be chosen from the follwing:

  -brief, -bf
  : produces an output that lists the name of each screen and the labels defined in each screen. This is the default.

  -long, -lg
  : produces output which resembles the appearance of the menu as displayed. This option produces substantially more output than -brief.