To: Distribution

From: James R. Davis

Date: 03/02/81

Subject: Whither Menus

1. INTRODUCTION

The original design for MR 9 Menu software (MTB-476) has drastic technical problems (MTR-170), and must be discarded. This document presents new plans - specifically, menu subroutines. These subroutines allow PL/I programs to present menus to users and get choices back. The functions here are at a lower level than those in the Menu Manager, thus they are more flexible and general. It is likely that these primitives would be combined with others to make it easier to write menu applications.

Send comments to:

By continuum on System M:
>udd>m>jrd>mtgs>menus

By Extended Mail Facility, on MIT or M:
JRDavis.Multics

By Telephone:
HVN 261-9382 or (617)-492-9382

2. OVERVIEW

2.1. What are the Requirements?

Item 4.5.2 of the MR9 Product Functional Specification (1) calls for a menu system. The specification in the PFS is very brief. It reads:

> MR9 will provide software that can be used to establish menu interfaces for existing software. This software will consist of a translator to translate menu descriptions into a form useful; for interpretation by the menu management software.

Since the PFS was written with the Menu Manager in mind, it may be a requirement that all functions of the Menu Manager also be provided in some form. Since I'm proposing completely new interfaces, a full and more detailed specification is of the utmost importance. For purposes of discussion, I see the following as the features of the Prototype Menu System.

> Applications programs are organized in a tree structure. The user starts at the root, and can travel down, up, return to the root, or quit. The Prototype Menu System used menu selection to travel down, and provided function keys to return to the root (First menu), to return to the parent node (Previous menu), and to quit.
>
> It can print help files (using function key) if provided by the applications writer.
>
> It manages an attached printer (two types are supported), in a terminal-dependent way, through an exec_com.
>
> The writer defines menus before the application runs. Headers and trailers are slightly dynamic in that they may contain references to active functions which are evaluated at display time.
>
> It sets up the window system, creating two windows. The upper window is always used for menu display, and the lower for interaction. The upper window is kept at a size just large enough for the menu on display, with the rest for use by the application.

---

(1) My copy is dated 28 May 1980

The application can request the full screen for large displays.

The reader should remember that many of the features were unpopular, and most have important limits - specifically terminal dependence.

2.2. Outcome of Technical Review

The design review of the Menu Manager was actually more concerned with how applications should be written - specifically, applications should not be written in command language, and interactive subsystems should not be treated as programming langauges (i.e. the "response" feature of the Menu Manager should not be provided in any form). This means that applications will be written in PL/I, and thus PL/I callable subroutines are needed.

2.3. Compatibility with the Prototype

I don't consider compatibility with any Prototype Menu System to be a requirement. The Prototype was just that, a prototype. It was an experiment into certain kinds of interfaces, it was not subject to the usual scrutiny of the Multics design process. Its popularity shows that Honeywell (like the rest of the Industry) recognized the significance and usefulness of menu interfaces, and indicates only that Multics should have some form of menu presentation, not the form it should take.

The above shouldn't need to be stated again, but it does.

The Marketing organization uses the Prototype for demonstration purposes. They could convert the Prototype to use the menu subroutines without major effort. No Multics product should be based on a Menu Manager, but there is no harm in running demos, so long as one is careful.

The menu descriptions (embedded in cv_sd statements) could be converted to the new forms given below by editor macros.

Nothing in the interface proposed here precludes use of exec_coms, indeed, MTB-494 proposes a command interface to these subroutines.

3. MENU DISPLAY

Several PL/I callable subroutines will be supplied. A menu will be represented by a "menu object", a pointer to an internal data structure, a representation of a menu (given below). Subroutines will be provided to create a menu object given a description, to display a menu (in a Video System window), to get a menu choice, and to delete menu objects.

Menu display is separated from menu choice because an application will commonly display one menu, and want several sucessive choices from it. When an option is selected, the cursor is placed at the option letter. To do this echoing, requires information about menu placement (e.g. where the option letters are). To avoid internal static, the caller will be required to keep pointers to this data, and pass them to the menu subroutines. This in turn requires the subroutines to create and destroy menu objects.

This interface is much more powerful than that of the Menu Manager, since it allows for menus to be defined as needed, rather than in advance (dynamic menus) and allows for multiple menus to be on display (in separate windows) at the same time.

MPM style documentation appears at the end of this document.

These interfaces will take four to seven person weeks to design, document, and debug.

4. CONTROL FLOW

It will be the application writers responsibility to provide any desired control flow. This is much easier in PL/I than in command language. The menu subroutines provide no flow control, which has the real advantage of not locking the programmer into a fixed, limited model. A skilled programmer could always emulate the heirarchical flow of the menu manager.

5. FUNCTION KEYS

If Multics software is to use function keys, it should be done in an terminal independent manner. A design for this was presented in MCR 4671, which has been approved. That MCR proposed storing function key descriptions in the Terminal Type File, and provided a method for accessing them. Although the design is in hand, the actual work has yet to be done, and would probably take two to four person-weeks to do.

Function keys are treated as a menu choice for which there is no displayed label. Function keys are not interrupts.

A perceived limitation of the Menu Manager is that function keys are not effective when typing in the lower window (i.e. in response to prompts). This design has the same limitation, and for good reasons. If need be, I can explain them, and suggest ways around the restriction, but I'd rather not do it here.

6. LOCAL PRINTER

Local printer support has nothing to do with menu presentation. The Prototype Menu Manager did have local printer support, but it is hard to use and quite terminal dependent.

Local printer support should provide logically separate paths (I/O switches) to the printer and to the screen, multiplexing the communications line to the terminal. Local printer support is a case of workstation support - where many physical devices (or components of the same physical device) are addressed over a shared line for the benefit of one user. We should have workstation support, but it is not further addressed here.

It will be addressed in a future MTB.

7. COMMAND INTERFACE, AF, ECS

A command/AF interface for menu selection is desirable. A forthcoming MTB proposes one.

8. MENU APPLICATION UTILITIES

Writing robust applications is not easy, menu oriented or not. There are many things that could be made easier (e.g. any_other handlers, querying the user) and many things completely missing (subroutine interfaces for per-user profile values, to lister, etc.) One can imagine a video-oriented tool for creating menu descriptions. Utilities could also be written to make the menu subroutines act more like Menu Manager (e.g. heirarchical control flow). We should remember that menu presentation is just one part of Office Automation, and that our finite resources may have better uses.

One important utility would manage windows (Desk Management, in the terms of the Window System (MTB-458)). No one knows how to define general Desk Management, so it is appropriate to implement some small utility to gain experience, then perhaps extend the Window System to do this itself.

9. SUBROUTINE DESCRIPTIONS

MPM-Style descriptions appear on the following pages. These subroutines, if approved, should be documented in CP51, tenatively titled "Menu Application Writer's Guide".

The menu_ subroutine provides menu display and selection services. It can display a menu in a window and get a selection from the user. The entries work with menu objects. A menu object is a pointer to an internal description of a menu. The caller is expected to preserve the pointer, and to perform no operation on it or through it other than comparison for nullity or equality with another menu object, except through the menu_ subroutine. Declarations for the entries and the associated structures are in the include file menu_dcls.incl.pl1

Entry: menu_$create_menu

This entry creates a menu object given its description. The menu data structure is allocated in a caller supplied area, and may be saved accross processes. A pointer to the new menu is returned, also with the minimum size of a window to hold the menu.

usage

```
dcl menu_$create_menu entry ((*) char (*) varying,
      (*) char (*) varying, (*) char (*) varying, pointer,
      (*) char (1) unal, pointer, pointer, pointer,
      fixed bin (35));

call menu_$create_menu (choices, headers, trailers,
      format_ptr, keys, area_ptr, needs_ptr, menu, code);
```

where:

choices (input)
: is an array of the names of the options. If the maximum number of choices is exceeded, the code menu_et_$too_many_options is returned. The current maximum is 35.

headers (input)
: is an array of headers. If the length of the first header is zero, then no headers are used. This allows the caller to specify no headers, without resorting to a zero-extent array, which would be invalid PL/I.

trailers (input)
: is an array of trailers. As for headers, a zero-length first trailer means that no trailers are

displayed.

format_ptr (input)
points to a structure that controls formatting of the menu. The structure is described below.

keys (input)
is an array specifying the keystroke for each option. The array must have at least as many elements as the array of option names. If not, the error code menu_et_$too_few_keys is returned. It may have more keys than choices. Each item of the array must be unique, or menu_et_$keys_not_unique is returned.

area_ptr (input)
is a pointer to an area where the menu description is allocated. If the area is not large enough, the area condition is signalled.

needs_ptr (input/output)
points to a structure giving requirements to display the menu. The structure is described below. The caller supplies this structure and fills in the version number, the remaining members are output from this entry.

menu (output)
is a newly created menu object.

code (output)
is a standard system error code, or an error code from menu_et_.

Entry: menu_$display_menu

This entry displays a menu object on a supplied window.

usage

```
dcl menu_$display_menu entry (pointer, pointer, pointer,
      fixed bin (35));

call menu_$display_menu (window, menu, code);
```

where:

window (input)
is a pointer to an IOCB for an I/O switch attached through the Window Management level of the Video System (i.e. crt_). This window must be large enough to hold the menu. A window used for menu display should be used ONLY for menu display, if redisplay optimizations are desired.

menu (input)
is the menu object to be displayed.

code (output)
is a standard system error code.

Entry: menu_$get_choice

This entry returns a choice from a menu. The menu is assumed to be already displayed on the window.

usage

```
dcl menu_$get_choice entry (pointer, pointer, pointer,
      bit (1) aligned, fixed bin, fixed bin (35));

call menu_$get_choice (window, menu, function_key_info,
      fkey, selection, code);
```

where:

window (input)
is a pointer to an IOCB for an I/O switch for a window.

menu (input)
is the menu object on display in the window.

function_key_info (input)
is a pointer to a data structure describing the function keys available on the terminal. This data structure is obtained by the caller from the ttt_info_$function_key_data subroutine. If this pointer is null, no function keys are used.

fkey (output)
tells whether if a function key was hit instead of a menu selection.

selection (output)
gives the option number or function key number chosen by the user. For an option, it is a number between 1 and the highest defined option, inclusive. For a function key, it is the number of the function key.

code (output)
is a standard system error code.

Notes

If a terminal has no function keys, the caller can define input escape sequences for function keys. These may be chosen to have mnemonic value to the end user. For example, if Function Key 1 is used to print a help file, the input sequence ESC H could replace it. In some applications, this will be easier for the end user to remember than an unlabelled function key. The caller can define these keys by allocating and filling in the same function key structure normally returned by the ttt_info_ subroutine.

If a key is hit that is not one of the option keys and is not a function key, then the terminal bell is rung.

Entry: menu_$describe_menu

This entry fills in a caller supplied data structure describing some of the aspects of a menu object. The caller can use this to ensure a window is sufficiently large to hold a menu.

usage

```
dcl menu_$describe_menu entry (pointer, pointer,
      fixed bin (35);

call menu_$describe_menu (menu, needs_ptr, code);
```

where:

menu (input)
is the menu object to describe.

needs_ptr (input)
points to a structure declared like menu_requirements. The caller fills in the version

to be menu_requirements_version_1, and the remaining members are filled in by this entry.

code (output)
is a standard system error code.

Entry: menu_$destroy_menu

This entry is used to delete a menu object. The caller uses this to free storage of a menu, since the representation of a menu is not known outside the menu_ subroutine. This entry has no effect on screen contents.

usage

```
dcl menu_$destroy menu entry (pointer, fixed bin (35));

call menu_$destroy menu (menu, code);
```

where:

menu (input)
is the menu object to destroy.

code (output)
is a standard system error code.

DATA STRUCTURES

A menu is described by the structure "menu_format".

```
dcl 1 menu_format          aligned based
(menu_format_ptr),
      2 version            fixed bin,
      2 constraints,
        3 max_width        fixed bin,
        3 max_height       fixed bin,
      2 n_columns          fixed bin,
      2 flags,
        3 center_headers   bit (1) unal,
        3 center_trailers  bit (1) unal,
        3 pad              bit (34) unal,
      2 pad_char           char (1);
```

where:

menu_format
: specifies the format for menu display. It gives limits for number of lines, and characters per line, specifies the number of columns (of options), and controls centering of headers and trailers.

version
: must be menu_format_version_1

max_width
: is the width of the window the menu will be displayed on. This value is used for centering headers and aligning columns.

max_height
: is the maximum height of the window, in lines.

n_columns
: is the number of columns to use in displaying options.

center headers
: if set, header lines will be centered using the window width supplied above. If not set, they are flush with the left edge of the window.

center_trailers
: Same as center_headers, but for trailers.

pad
must be "0"b.

pad_char
is the character used for centering headers and/or trailers.

The requirements for a menu are specified by the structure "menu_requirements".

```
dcl 1 menu_requirements    aligned based
(menu_requirements_ptr),
      2 version            fixed bin,
      2 lines_needed       fixed bin,
      2 width_needed       fixed bin,
      2 n_options          fixed bin;
```

where:

version
is set by the caller, and must be menu_requirements_version_1.

lines_needed
is the number of lines required. If the window does not have this many lines, menu display will fail.

width needed
is the number of columns needed.

n_options
is the number of options defined.

The include file also provides an array of key characters that may be used in the menu to select options. This array can be used by the caller as input to the menu_$create_menu entry. Its name is MENU_OPTION KEYS.

The menu_$display_menu subroutine uses another subroutine, window_display_ to display menus with as few changes to the actual screen as possible. This is called redisplay optimization, and is an attempt to make display of menus happen as quickly as possible. Redisplay should be done by the video system itself, but there are reasons it cannot be part of the video system at this time. The window_display_ subroutine is an attempt to provide at least some redisplay. Applications programmers do not use the window_display_ entry point itself, but will probably need to call the window_display_$window_changed entry point at times if they alter the window directly.

Documentation for this subroutine follows:

The window display subroutine provides a very simple, limited redisplay for a window. The caller provides a screen image, which is an array of unaligned characters strings. The screen image represents the intended contents of the window. The caller must perform all conversion - each character in the array must occupy exactly one character position on the terminal. In practise, this means that only printing characters may be used. The caller must also ensure that the window is large enough to hold the image.

Entry: window_display_

This entry displays a screen image on a given window.

usage

```
dcl window_display_ entry (pointer, (*) char (*) unal,
      fixed bin (35));

call window display  (window, image, code);
```

where:

window (input)
is the IOCB for a window.

image (input)
is the image to display.

code (output)
is a standard system error code.

Entry: window_display_$window_changed

This entry is called to inform the window_display_ package that the screen contents of the window are no longer certain. This implies that no redisplay optimization is possible, since knowledge of previous contents is unreliable. With a few exceptions, it must be called after any operation is performed on a window other than a call to window_display_. The list of exceptions is given below.

usage

```
dcl window display $window_changed entry (pointer,
      fixed bin (35));

call window_display_$window_changed (window, code);
```

where:

window (input)
is the window that has changed.

code (output)
is a standard system error code. No errors are possible.

Exceptions

This entry must be called after any operation that affects screen contents. Operations that do not effect screen contents are:

all modes operations
all cursor motion
iox_$get_chars
window_$bell

In addition, it need not be called after a change in window size if the size change affected only the only the end of the window was changed. That is, if the window grew, the new lines added were added to the end; or if it shrank, the lines removed were takenm from the end. This will be true of the top window on the screen only, as a rule.

Entry: window_display_$discard

This entry is called to inform window_display_ that a screen image is no longer needed. It is useful to reclaim the storage used for the screen image.

usage

```
dcl window_display_$discard entry (pointer, fixed bin (35));
```

```
call window_display $discard (window, code);
```

where:

window (input)
is the window that is no longer to have a screen image.

code (output)
is a standard system error code. No errors are possible.

10. INTERNAL REPRESENTATION

The internal representation of a menu is given here for review purposes. It is, of course, internal, and subject to change without notice.

```
dcl 1 menu              aligned based (menu_ptr),
      2 version         char (8) init (MENU_VERSION_ONE)
      2 window_requirements,              /* size of menu */
        3 height        fixed bin,
        3 width         fixed bin,
      2 n_options       fixed bin,        /* N. valid options */
      2 option_info     (35),
        3 key           char (1) unal,    /* key to hit */
        3 pad           bit (27) unal,
        3 line          fixed bin,        /* where to echo */
        3 col           fixed bin,
      2 screen_image    (lines_alloc refer (menu.height)) unal
                        char (chars_alloc refer (menu.width));

dcl menu_ptr pointer;

dcl (lines_alloc, chars_alloc) fixed bin (21);
```

(The version is a character string, rather than a number, so that this structure may be distinguished from all other structures that have a version in their first word, and whose version is also 1.)