<table>
<thead>
<tr>
<th>MULTICS MENU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATION</td>
</tr>
<tr>
<td>FACILITIES</td>
</tr>
</tbody>
</table>
SUBJECT

Description of the Multics Menu Creation Facilities

SPECIAL INSTRUCTIONS


Throughout the document change bars are used to indicate technical changes and additions; asterisks denote deletions.

Refer to the Preface for "Significant Changes."

SOFTWARE SUPPORTED

Multics Software Release 11.0

ORDER NUMBER

CP51-02

DATE

February 1985
PREFACE

The publication is intended for application programmers who are building menu interfaces to existing software. The Multics menu system consists of several commands and subroutines which can be used to create and manage menus.

The major topics presented are:

- A description of the terminal-management software that provides a means of dividing the terminal screen into different regions and of performing real-time editing. The terminal-management software is referred to in text as the "video system."

- A description of the Multics commands and subroutines provided for creation and manipulation of video screens and creation and display of menus.

- A description of the Multics I/O modules that support terminal-management functions.

There are some manuals that are prerequisites to this one in that they describe tools that the application writer uses. The writer must be familiar with Multics I/O processing, commands, and subroutines. The manuals describing these are as follows:

(Order No. AG91)  Programmer's Reference

Multics Commands and Active Functions  
(Order No. AG92)  Commands

Multics Subroutines and I/O Modules  
(Order No. AG93)  Subroutines

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.
The Programmer's Reference manual describes I/O processing and contains specific details for the use of screen terminals. The Commands manual contains the descriptions of commands that are referenced in text, such as exec_com. The Subroutines manual contains Multics subroutine descriptions.

In addition, this publication assumes the programmer is familiar with PL/I, FORTRAN, or COBOL, and the Multics exec_com facility. (The exec_com examples in this document use Version 2 of exec_com.) The PL/I language is described in the PLII Language Specification, (Order No. AG94); the FORTRAN Language is described in the FORTRAN Reference Manual, (Order No. AT58); the COBOL Language is described in the COBOL Reference Manual, (Order No. AS44).

Significant Changes in CP51-02

The video system now supports windows which do not extend across the full width of the screen. See Section 2 for details.

The video system editor now accepts either upper or lower case letters when you use default escape sequences. See Section 4 for details.

The "suppress_redisplay" field has been added to the line_editor_info structure. See Section 4 for details.

The "window_call" command now accepts the "--width NC (--wid NC)" control argument which specifies the width of a region for a request. See the "window_call" command in Section 5.

The "change_window" and "create_window" arguments to the "window_call" command now accept the "--column C" and "--width NC" control argument. See Section 5.

A "--line_speed (--ls)" control argument has been added to "window_call invoke". This allows you to specify the speed of your connection to Multics when you use the video system. If no "--line_speed" is specified, the current line_speed is used. See Section 5 for details.

The "window_$edit_line" entry which allows applications to preload the video editor input buffer with a string, has been added to the window_ subroutine. See Section 6 for details.

The "window_$write_raw_text" entry in the window_ subroutine now causes the cursor position to become undefined and sets the screen_invalid window status flag. See Section 6.
Support for the "set_term_type" control order has been added to the tc_io_ I/O module. This control order or the set_tty command allows you to change the terminal type in a video session.

Two new control arguments have been added to window_io_ switch. The "-first_column COL_NO" (control argument) is the column number on the screen where the window is to begin. The "-width N COLS" (control argument) is the number of columns in the window. See Section 7.

The "set_output_conversion" and "get_output_conversion" control orders have been added to the window_io_ I/O module. The "get_output_conversion" control order obtains the current contents of the specified table. The "set_output_conversion" control order provides a table to use in formatting output to identify certain kinds of special characters. See Section 7 for details.

The "get_special" and "set_special" control orders have also been added to window_io_. The "get_special" control order obtains the contents of the special_chars table currently in use. The "set_special" control order provides a table that specifies sequences to be substituted for certain output characters. See Section 7.

A "get_editor_key_bindings" control order, which returns a pointer to the line_editor_key_binding structure describing the key bindings, has been added to window_io_. The "set_editor_key_bindings" control order has been changed. New fields have been added to the line_editor_key_binding structure. The control arguments "-name STR", "-description STR", and "-info_pathname PATH" have been added to the io_call support set_editor_key_bindings control order.

A new mode, "edited, ^edited" suppresses printing of characters for which there is not defined Multics equivalent on the device referenced. See Section 7 for details.
## CONTENTS

### Section 1
Introduction to the Menu System .......................... 1-1
  What is a Menu ....................................... 1-1
  The Connection Between the Menu System and the Video System .......................... 1-5

### Section 2
Introduction to the Video System .......................... 2-1
  What is a Window ..................................... 2-1
    Window Capabilities .................................. 2-3
    Positioning the Cursor ............................... 2-3
    Selective Erasure ................................... 2-4
    Scrolling ............................................. 2-4
    Selective Alteration .................................. 2-4
    Miscellaneous ....................................... 2-4
  Window/Video Commands and Subroutines .................. 2-5
  Attaching the Video System ............................. 2-5
  Detaching the Video System ............................. 2-7
  Design Requirements for Windows ......................... 2-8
  Window Operations ..................................... 2-9
    Create Window Operation .............................. 2-9
    Important Window Requests ............................. 2-9
    Change Window Operation .............................. 2-12
    Destroy Window Operation .............................. 2-12
    Clear Window Operation ............................... 2-13
    Other Useful Operations ............................... 2-14

### Section 3
Menu Applications ......................................... 3-1
  Guidelines for Function Keys ............................ 3-1
  The exec_com Example ................................... 3-2
  The PL/I Example ....................................... 3-4

### Section 4
Video System Details ...................................... 4-1
  Real-Time Editing ...................................... 4-1
    The Erase Character .................................. 4-2
    The Kill Character .................................... 4-2
    The Line Editor ...................................... 4-2
      Moving the Cursor ................................... 4-2
      Deleting Characters and Words ....................... 4-2
      Retrieving Deleted Text ............................. 4-3
    Other Editor Requests ................................ 4-4
    Writing Editor Extensions ............................. 4-5
    Line Editor Routines ................................ 4-5
    Window Editor Utilities ............................... 4-8
    End-Of-Window Processing ............................. 4-10
  More Processing ....................................... 4-10
  Output Buffering ...................................... 4-11
  Subroutine Interface ................................... 4-11
Command Line Interface ........................................................................ 4-11

Section 5 Commands ........................................................................... 5-1
  menu_create ................................................................................... 5-2
  menu_delete ................................................................................... 5-4
  menu_describe ................................................................................. 5-5
  menu_display ................................................................................ 5-6
  menu_get_choice ........................................................................ 5-7
  menu_list ....................................................................................... 5-9
  window_call ................................................................................... 5-10
  Argument Descriptions .................................................................. 5-11

Section 6 PL/I Subroutine Interface .................................................. 6-1
  menu_ ......................................................................................... 6-2
  video_data_ .................................................................................. 6-12
  video_utils_ ................................................................................ 6-13
  window_ ....................................................................................... 6-15

Section 7 I/O Modules ....................................................................... 7-1
  tc_io_ .......................................................................................... 7-2
  window_io_ ................................................................................... 7-5

Section 8 Fortran Interface ................................................................ 8-1
  ft_menu_ ....................................................................................... 8-2
  ft_menu_$init1 ............................................................................. 8-8
  ft_menu_$init2 ............................................................................. 8-8
  ft_window_ ................................................................................... 8-14

Section 9 Cobol Interface .................................................................. 9-1
  cb_menu_ ....................................................................................... 9-2
  cb_window_ ................................................................................... 9-14

Appendix A I/O Switch Attachments .................................................. A-1

Appendix B Error Code Handling ...................................................... B-1

Index ................................................................................................. i-1

Illustrations

Figure 2-1. Windows on a Screen ....................................................... 2-3
Figure A-1. Standard Attachments .................................................... A-3
Figure A-2. Attachments After the Invocation of Video .................... A-6
Figure A-3. Attachments After Execution of the doc_sys.ec exec_com A-7
SECTION 1

INTRODUCTION TO THE MENU SYSTEM

Multics is a large-scale, interactive system with a rich repertoire of commands, subroutines, tools and many interrelated subsystems. There are over 2,000 commands and subroutines alone. Effective use of the power this system has to offer is a specialty not every user masters. In many cases there is no reason to master it. The majority of users have specific tasks to accomplish online and require only a small subset of the available tools. What they do need to know is what tools exist to get their job done most efficiently. The easier it is to figure that out, the better. Since many of these people are not trained in computer use, the system should be made easy to understand, provide flexibility and keep training at a minimum. The Multics menu system provides a means of accomplishing this.

WHAT IS A MENU

A menu is a list of options presented to the user on a video terminal. By typing a single key, designating an option choice, an action is performed. The most important feature the menu system has to offer is permitting the user who knows very little about the system to interact with the computer. No knowledge of commands is required since the system calls the command to do the job once the user indicates an option. All the actions required for a specific task are displayed on the terminal, selections are made, and the user is ushered through a given task by being prompted. You can design menus for all different levels of expertise and for any number of tasks. The easiest way to explain the menu system and to provide application ideas is to give examples of menus. The menu "Games" is shown here.

<table>
<thead>
<tr>
<th>Type a number and the corresponding action will be performed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAMES</td>
</tr>
<tr>
<td>1) Print a maze</td>
</tr>
<tr>
<td>2) Print a large maze</td>
</tr>
<tr>
<td>3) Play Football</td>
</tr>
<tr>
<td>4) Play Baseball</td>
</tr>
<tr>
<td>5) Do a Simulated Parachute Jump</td>
</tr>
<tr>
<td>6) Play Star Trek</td>
</tr>
<tr>
<td>7) Play Adventure</td>
</tr>
<tr>
<td>8) Guess the Animal</td>
</tr>
<tr>
<td>9) Play Backgammon</td>
</tr>
<tr>
<td>==============================================================</td>
</tr>
</tbody>
</table>
Imagine that the boxes in all the examples in this section are on terminal screens. This entire display is defined by the menu application. The screen is divided into two sections with the top part of the screen for menu display and the bottom part of the screen for user input/output.

The user of this menu selects one of the options and the screen changes from the list of menu options to the description of a specific game. The screen is no longer divided into two parts and the user input/output section of the screen is expanded to full size. For example, if Option 5 is selected, the transactions appear as follows. In this example, user-typed input is preceded by an exclamation point.

```
Welcome to "splat"--the game that simulates a parachute jump. Try to open your chute at the last possible moment without going splat.
Select your own altitude? !yes
What altitude (ft)? !5000
```

When the user is finished playing this game, the screen goes back to the original menu display. Another option is selected or the user exits this particular menu.

The next example is a menu for Tess True-Heart, a new terminal operator. Other than knowing how to login, Tess is a Multics novice. She needs to learn a little bit about the Multics system, i.e., how a command works, how to read her mail, and what manuals to read for details. Tess is at an advantage because the word processing system she worked on in her previous job also used menus, so she understands the concept. This is an important advantage for the application writer too. Since menus are used widely throughout the industry, people who use your menus will not find the concept a foreign one. As illustrated below, the menu system quite effectively "fences off" the Multics system into an understandable set of tools for personal use. The following example was written for Tess as an introduction to Multics.
As an example of the material in some of these options, Option 2 might discuss the list, help, and dprint commands. All the commands Tess is likely to use in her daily work are candidates for this option.

The commands in Option 3 would be more sophisticated and might include exec_com and the absentee commands.

Another example of a menu user is Percival C. Monday. Percy has no former experience with a computer and it is peripheral to his job. He uses it essentially as a filing system. This application is not unlike one intended for ticket agents at an airline counter, who use a computer strictly for one set of tasks. Percy must be able to read orders received, process orders, file the orders, check the budget allocation/expenditures, charge a department, maintain an inventory and change the inventory as orders are filled and shipments arrive. A menu to accomplish these tasks might look like this:

```
<<MANUAL ORDERS>>

Enter the number corresponding to the function to be performed.

1) List orders to be processed
2) List orders processed this month
3) List budget information
4) Enter billing information
5) Update inventory

============================================================
```
In the previous examples, the user went from a menu to the game "parachute jump" or to explanations of commands. In this example, Percy is going from the first menu to other menus. If Percy selects Option 5, the screen might look like this:

<table>
<thead>
<tr>
<th>1) List of parts available</th>
</tr>
</thead>
<tbody>
<tr>
<td>2) Additions made to inventory list</td>
</tr>
<tr>
<td>3) Deletions made to inventory list</td>
</tr>
</tbody>
</table>

Percy selects one of the options in this list and performs the appropriate action.

The first three examples are for inexperienced users and the advantages for such persons are obvious. However, the menu system can be tailored for the experienced user as well. The next example is a manager's application. The manager is Gloria VanDerMint, who has five people working for her in the research and development department. In addition to her development work, she has a number of tasks that must be performed routinely, so you can incorporate them all into one menu. You can set up a number of data bases containing information such as weekly status reports from her unit, and from these she writes the unit status report or performance appraisals, and updates schedules. You may also include the memo command to remind her when performance appraisals and status reports are due. Another convenient command to incorporate is calendar, which reminds her of meetings and trips. Gloria's menu is given below.

<<<The Good, The Bad, and The Boring>>>  
1) Read memos 6) Personal Schedules  
2) Send memos 7) Produce Schedules  
3) Calendar 8) Performance Appraisal Form  
4) Modify calendar  
5) Unit Status reports  

============================================
1-4 CP51-02
An additional menu for more complex tasks is one that offers a choice of programming procedures. This is helpful for people who have programmed on other systems but not on Multics and discusses the languages and editors available, tells them about formatting programs explain compiling on Multics and discusses debugging tools. It might contain the following:

<table>
<thead>
<tr>
<th>&lt;&lt;&lt;&lt;Programming Procedures&gt;&gt;&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Name Your Language</td>
</tr>
<tr>
<td>2) Enter New Program</td>
</tr>
<tr>
<td>3) Update Existing Program</td>
</tr>
</tbody>
</table>

THE CONNECTION BETWEEN THE MENU SYSTEM AND THE VIDEO SYSTEM

In the next section, the video system is introduced. A more detailed discussion is presented in Section 4. Menu application programs use the video system to manage the display on the terminal screen. As noted above, all these examples have the screen divided into portions which have different uses. Since we cannot physically divide the screen, we must do it logically. This is the job of the video system, and this terminal management is required to support the menu software.
INTRODUCTION TO THE VIDEO SYSTEM

The advent of comparatively low cost video terminals has brought a new dimension to the computing industry. Today's video terminals have many more capabilities than hard copy ones. Real-time editing, and higher speed communication are available and the display can change easily and quickly with varied functions.

The video system is a terminal-independent presentation interface. This means that an application can run on any supported terminal and produce essentially the same display. The video system enables the application writer to divide the terminal screen into "windows" to partition the display. The menu writer must have a thorough understanding of windows; how to invoke the video system (the first step in the process of creating windows), how to revoke the video system, and how to create, destroy, change and clear windows. This section discusses design considerations involved in using windows and covers the video material most important to the menu application writer. For a more detailed description of the features of the video system, see Section 4 of this manual.

WHAT IS A WINDOW

A window is an area of the screen whose contents can be manipulated without affecting the rest of the display. For example, the user may scroll the contents of a segment in one window without moving the contents of the segment displayed on any other part of the screen.

Each window behaves like an individual video terminal. Many possible operations may be performed on a window. These include displaying characters, moving the cursor, erasing lines, inserting lines, and others. Characters are normally sent to a window via the Multics I/O system and the iox_ subroutine (see the Multics I/O and Subroutines manual, Order No. AG93). Additional operations specific to the capabilities of video terminals are performed by the window_ subroutine (described in Section 6), which is analogous to iox_.

A window is a rectangular region of the screen. The screen can be divided into several windows that can be viewed simultaneously but the windows may not overlap. The number of line and columns in each window can vary. A window can be one column wide or it can extend across the full width of the screen.

The size of a window is specified at the time the window is created. Character positions are identified by line and column with the origin (or home) located at the upper left hand corner of the window. Each window has its own home, line 1, column 1, and character positions are always with respect to the home of the specific window.
If you want to create a window from command level, use the window_call command with the create_window argument. You can also use the window_create entry to the window_subroutine to create a window on the terminal screen. (Refer to the description of window_subroutine later in this manual.)

The command syntax for creating windows from command level is:

```
window_call create_window -io_switch WINDOW
{ -line L -column C -height NL -width NC}
```

window_call, wdc

this command provides a command interface to the video system

create_window, crwd

this argument creates a new window on the screen

- io_switch WINDOW 1 - io_switch
  WINDOW 1 specifies the window associated with the given I/O switch.

- line L
  specifies the line number on the screen where the window is to begin. To create a
  window beginning on the third line, use -line 3. If -line is not specified, the default is
  line 1.

- column C, col C
  specifies the column number on the screen where the window is to begin. To create a
  window beginning in the third column, use -column 3. If -column is not specified, the
  default is column 1.

height NL, - hgt NL
  specifies the height of the window. To create a window 10 lines high, use -height 10. If
  -height is not specified, the default is the remainder of the screen.

- width NC, - wid NC
  specifies the width of the window. To create a window 20 columns wide, use -width 20.
  If -width is not specified, the default is the remainder of the screen.

Figure 2-1 is an example of three different types of windows that you can create on a
screen. You can create a window called WINDOW_1 that is 20 columns wide and 10 lines
high. This window begins on the third line and the second column. To create WINDOW_1
(shown in Figure 2-1), type the following command line:

```
wdc crwd -is WINDOW_1 -line 3 -column 2 -height 10 -width 20
```

You can also create a second window on the same screen called WINDOW_2. This
window is 3 columns wide and begins in column 25. Since line and height are not specified,
the window begins in line one and fills the remainder of the screen. To create WINDOW_2
(shown in Figure 2-1), type:

```
wdc crwd -is WINDOW_2 -column 25 -width 3
```
You can create a third window on the screen called WINDOW_3. This window is 14
columns wide and 7 lines high. This window begins at line 17 and column 32. To create
WINDOW_3 (shown in Figure 2-1), type:

    wdc crwd -is WINDOW_3 -line 17 -column 32 -height 7 -width 14

Refer to Section 5 for more information on these commands and other commands used
by the menu and video software.

Window Capabilities

The capabilities defined for a window are grouped into five categories: positioning the
cursor, selective erasure, scrolling, selective alteration, and miscellaneous. Window operations
may be performed with the window_call command or by a call to the window_subroutine.
These are described in Sections 5 and 6 respectively.
POSITIONING THE CURSOR

Each window has its own logical cursor. This cursor exists even when the terminal’s cursor is performing operations in another window. The position of this cursor may be explicitly changed in a variety of ways. The cursor can be positioned absolutely or relatively. Absolute positioning can be to the home position or to an arbitrary line and column. Relative positioning can be up, down, left, or right any number of positions. The cursor also moves as characters are displayed in the window.

SELECTIVE ERASURE

Selective Erasure (or clearing) means changing some region of the display so that no visible characters appear in that region, without changing any other area of the window. Most video terminals are capable of at least some selective erase operations. Where possible, the video system uses any special terminal features present to clear regions. When the terminal has no useful feature for clearing the specified region, regions are cleared by overwriting them with spaces. This can be a rather slow operation.

A region is a rectangle contained within a window. Like a window, it has an extent (height and width) and a position. All erasure operations pertain to regions. The definition of the region may be explicit (position and extent supplied in the call) or implicit (the region begins at the current cursor location, or at home). The cursor is always left at the origin of the region.

A window may be cleared: entirely, from the home to the end of the window; from the current cursor position to the end of the current line in the window; from the current cursor position to the end of the window. An arbitrary region may also be cleared.

SCROLLING

A window may be scrolled up or down by a given number of lines. Scrolling up means moving lines up from the bottom of the window – deleting lines at the top, and adding new, blank lines at the bottom. Scrolling down means moving lines from the top of the window down, deleting at the bottom and adding at the top. Scrolling is usually done automatically by the video system when output fills the window, but it can also be requested explicitly.

SELECTIVE ALTERATION

Selective alteration means adding or deleting characters or lines in the middle of the window. When characters (or lines) are added, adjoining characters (or lines) move over to make room for the new ones. When characters (or lines) are deleted, characters (or lines) move in to fill up the gap. This differs from selective erasure, which only affects the characters erased.

MISCELLANEOUS

Among other things, entries are provided in the window subroutine and the window_call command to sound an audible alarm, to obtain the current cursor position, and to output an arbitrary character sequence.
Window/Video Commands and Subroutines

The command supporting windows is introduced here but is explained in detail in Section 5 of this manual.

window_call
is the command interface to the video system. This is used in exec_com applications while the window subroutine is used in PL/I, ft_window_ is used in FORTRAN, and cb_window_ is used in COBOL applications.

The subroutines supporting windows and the video system are described in detail in Section 6 of this manual and are as follows:

video_utils_
activates and deactivates the video system.

video_data_
is a data segment containing information about the video system.

window_
is the subroutine interface to the video system. It is the corresponding subroutine to the window_call command.

Attaching the Video System

You must check whether or not the users of the proposed menu have the video system turned on. It is not likely that novice users would do this initially but it might be included in a project start_up. If it is on, it is important that you leave it alone. Do not turn it on again or you will get an error message. If you have determined that the video system is turned on, you should then have your application use the space allocated to the user_input/output window instead of the whole screen. Thus, if the user creates a separate window for interactive messages, an application should not use that space. Using the space allocated to the user_io window respects the user’s explicit wishes and prevents violation of the restriction against using two overlapping windows at the same time.

When the video system is invoked, the entire screen is covered by a window associated with the user_i/o I/O switch. Determine how much of the screen you have and divide up that amount for use by your application. Since terminals vary in the length of the screen, and some users already may have some lines devoted to their own video display, you are probably dealing with less than 20 lines, so design with that in mind. As long as there are eight or ten lines available for user input/output that should be sufficient.

The first step then is determining whether or not the video system is turned on and, if not, turn it on. This should be included at the beginning of all menu applications. The following is the exec_com example. The lines are numbered only for the purpose of explanation and should not be included in your exec_com.
$\text{set already_video $[io attached user_terminal]\ ]}$

\$\text{if not (already_video)\ ]}$

\$\text{then window_call invoke}$

\$\text{set first_line $[window_call get_first_line] \ n\_lines}$

\$\text{[window_call get_window_height]}$

where:

1. determines whether or not the video system is attached to the user's terminal.
2. turns it on if it isn't already on.
3. invokes window_call initiating the window environment.
4. sets the lines for the window. This is part of the first step because when you revoke the video system at the end of the exec_com, you must set the screen to the size it was originally.

The following is the PL/I example that does the same thing. Declare statements are included in the example. Again, the lines are numbered for the purpose of explanation and the numbers should not be included in your program.

dcl (addr, null) builtin;

dcl iox_$control entry (ptr, char (*), ptr, fixed bin (35));
dcl com_err_ entry () options (variable);
dcl iox_$user_io ptr ext static;
dcl video_utils$turn_on_login_channel entry
    (fixed bin (35), char (*));
dcl video_data$terminal_iocb ext static ptr;
dcl ME char (32) init ("test_program") static options (constant);
dcl code fixed bin (35);
dcl already_video bit (1);
dcl reason char (128);

1  %include window_control info;
2  dcl 1 my_window_info like window_position_info;
3  my_window_info.version = window_position_info_version_1;
4  if video_data$terminal_iocb = null () then do;
5     call video_utils$turn_on_login_channel (code, reason);
6     if code ^= 0 then do;
6       call com_err_ (code, ME, "^a", reason);
7       return;
8     end;
9     already_video = "0"b;
10    end;
11 else already_video = "1"b;
12  call iox_$control (iox_$user_io, "get_window_info",
13       addr (my_window_info), code);
14  if code ^= 0 then do;
15     call com_err_ (code, ME, "get_window_info");
16     return;
17    end;
where:

1. includes appropriate structure declarations
2. declares an automatic copy of window info
3. sets the version number of window info
4. determines if the video system is not activated then does 4 through 6
5. turns on the video system and
6. if there is an error, reports it to the caller and quits
7. makes a note to the effect that video was invoked by this program
8. goes to here if the video system is already activated (video was not activated by this program)
9. gets the current size and location (beginning line number) of the user_i/o window
10. prints error message

Detaching the Video System

At the end of the exec_com, you have to turn off video and leave things as you found them. First, is the exec_com example for revoking the video system. The lines are numbered only for the purpose of explanation and these numbers should not be included in your exec_com.

```plaintext
1 &if &{(already_video)
2 &then window_call change_window -line &{(first_line)
   -height &{(n_lines)
3 &else window_call revoke
```

where:

1. determines whether or not video was activated by this exec_com.
2. if video was activated by another exec_com, then user_i/o window is returned to previous size and it is cleared.
3. otherwise, the window interface to the video system is deactivated and the user_i/o window goes to full screen.

The PL/I example:

```plaintext
1   if already_video then do;
2      call video_utils_${turn_off_login_channel}(code);
      if code ^=0 then do;
      .
      .
      .
      end;
end;
3   else do; call iox_$control(iox_$user_io, "set_window_info",
      addr(my_window_info), code);
      if code ^=0 then do;
      .
      .
      .
      end;
end;
```
Design Requirements for Windows

In Section 1, all of the examples used two windows: the top window which displayed the menu itself and the bottom window which was for user input/output. As part of the menu design process, you decide ahead of time how the display will look and from that determine the number of windows that will be advantageous.

As an example, you may have the screen divided into three windows. The top window could display the status of the user with the user name, a description of what he's doing and a clock. The middle window could contain various menus and could grow or shrink depending on the selection made. The bottom window could be for unformatted output and for typing in input.

The number of windows technically permitted is quite large and probably more than you will need. Knowing how many functions are to be performed, you should carefully select the number of windows to be used by an application. It is possible on a 24 line terminal to have 24 windows but rarely, if ever, would that be useful. Each window would be too small and the screen would be too cluttered. Practically, there should not be more than five. In the examples in Section 1, there are lines marking the division between top and bottom windows. This is a trailer line specified in the exec_com or program. It is not necessary, but does make the delineation obvious and aids readability. Windows may not overlap. Each window has its own extent (height and width) and location (the position of its home on the screen). Windows can change their extent and location as long as they never overlap. The initial extent and location of a window is determined in the attach description of the window.

Window Operations

The rest of this section discusses the operations of window_call and window that are most essential. These include: create, change, destroy, and clear. Specific examples are given for exec_com and PL/I applications.
**CREATE WINDOW OPERATION**

Now you need to define windows and this is done with arguments to window_call or with the entry points of the window_subroutine. The first action discussed is create_window. Part of the creation process is the naming of windows. Windows are associated with iox_ I/O switches. The "name of the window" is just the name of the switch, or as it is sometimes called, the iocb name. Since many Multics commands and subroutines make use of the standard switches user_io, user_input, error_output, and user_output, it is usually necessary to have these switches connected to some window. This is done by window_call invoke or video_utils$turn_on_login_channel. By convention, the bottom window of the screen is used for user_i/o.

**Important Window Requests**

Before a window can be created you must decide on its starting line number as discussed above in "Attaching Video" and its length (in number of lines). As mentioned earlier, it is customary to get space for a new window from the user_i/o window and to position the new window at the top of the user_i/o window. Therefore, one of the first things to do is find out where the user_i/o window is. Once this is known you must determine just how high, in lines, the new window must be and shrink the user_i/o window by that amount. It is a good idea to always check to make sure there is enough space left in the user_i/o window to allow meaningful communication once it has been shrunk. In our examples we will insist on at least a five line user_i/o window.
To do all that has been discussed so far in an exec_com, we would have the following:

δ- stored in the default value segment as the_menu.

\[ \text{δset } \text{io}_\text{start } \& [\text{window_call get_first_line}] \]
\[ \text{δset } \text{io}_\text{height } \& [\text{window_call get_window_height}] \]
\[ \text{δset } \text{menu}_\text{height } \& [\text{menu_describe the_menu -height}] \]

δ- Now calculate the new positions of both windows.

\[ \text{δset } \text{menu}_\text{start } \& (\text{io}_\text{start}) \]
\[ \text{δset } \text{io}_\text{start } \& [\text{plus } \& (\text{io}_\text{start}) \& (\text{menu}_\text{height})] \]
\[ \text{δset } \text{io}_\text{height } \& [\text{minus } \& (\text{io}_\text{height}) \& (\text{menu}_\text{height})] \]

δ- The label referenced below would, of course, need to be δ- defined and would include an appropriate error message.

\[ \text{δif } \& [\text{nless } \& (\text{io}_\text{height}) \& 5] \]
\[ \& \text{then } \& \text{goto USER_I/O_TOO_SMALL} \]

δ- Now shrink user_i/o

\[ \text{window_call change_window -line } \& (\text{io}_\text{start}) -\text{height } \& (\text{io}_\text{height}) \]

δ- And define the new window, called able

\[ \text{window_call create_window -io_switch able -line } \& (\text{menu}_\text{start}) -\text{height } \& (\text{menu}_\text{height}) \]

The real work of creating the new window above was done by the window_call command with the create_window argument. This command created the necessary iox_I/O switch attachments to make "able" an I/O switch which describes a video system window that occupies the first "menu_height" lines of what was user_i/o.
THE PL/I EXAMPLE

A window can be created either at command level or from a PL/I subroutine. To do the same thing in PL/I you would use the following code fragment:

/* Get the variables initialized. We assume the menu has */
/* been created and the requirements for the menu are */
/* stored in the menu_needs structure (see menu_ for dcl) */

%include window_control_info;
 dcl 1 io_window_info like (window_position_info);
 dcl 1 menu_window_info like (window_position_info);

 io_window_info.version = window_position_info_version_1;
 call iox_$control (iox_$user_io, "get_window_info", addr
   (io_window_info), code);
   if code ^= 0 then do;
     process the error
   end;

 menu_window_info = io_window_info;

/* Now calculate the new positions of both windows. */

 menu_window_info.origin.line = io_window_info.origin.line;
 io_window_info.origin.line = io_window_info.origin.line
   -menu_window_info.extent.height;
 io_window_info.extent.height = io_window_info.extent.height
   -menu_window_info.extent.height;
 if io_window_info.extent.height < 5
 then do;
   if code ^= 0 then do;
     process the error
   end;
 end;

/* Now shrink user_i/o */

 call iox_$control (iox_$user_io, "set_window_info", addr
   (io_window_info), code);
   if code ^= 0 then do;
     process the error
   end;

/* And define the new window */

 call window_$create (video_data_$terminal_iocb, addr (menu_window_info),
   menu_window_iocbp, code);
CHANGE WINDOW OPERATION

In the above examples we have seen that it was necessary to change or shrink the user_i/o window in order to create a new window. When we discuss destroying windows below we will see a need to expand the user_i/o window to recover the space freed by the destruction of a window.

Command level changes are done with the window_call keyword change_window. In PL/I the changes are made by the set_window_info control order. In general this will be preceded by a get_window_info control order and some calculations.

DESTROY WINDOW OPERATION

Once a window is no longer needed it must be destroyed, i.e., the I/O switch must be closed and detached thus freeing up the space on the screen that was occupied by the window. In addition, this space should be returned to some active window so that it can be used. If the freed space is adjacent to the user_i/o window it should be consumed by that window, but it can be added to any adjacent window. In our examples we will add it back to user_i/o.

To reverse the effects of the exec_com window creation example above we would have:

\[
\begin{align*}
\&- & \text{destroy the able window} \\
\&- & \text{and let user_i/o have the space back} \\
\&\text{set io_start } \& (\text{menu-start}) \\
\&\text{set io_height } \& [\text{plus } \& (\text{menu_height}) \& (\text{io_height})] \\
\&\text{set menu_start 0 menu_height 0} \\
\text{window_call change_window -line } \& (\text{io_start}) -height \& (\text{io_height})
\end{align*}
\]

In PL/I we would have:

\[
\begin{align*}
/* & \text{destroy the able window */} \\
\text{call window\$_destroy (\ldots);} \\
\text{if code } \& = 0 \text{ then do; } \\
\text{. process the error} \\
\text{end; }
\end{align*}
\]
/* and let user_i/o have the space back */

io_window_info.origin.line = menu_window_info.origin.line;
io_window_info.extent.height = menu_window_info.extent.height
    + io_window_info.extent.height;

call iox$_control (iox$_user_io, "set_window_info",
    addr (io_window_info), code);
    if code ^= 0 then do;

    process the error

end;

CLEAR WINDOW OPERATION

Another very useful operation is the clear_window operation. This clears the entire
window to all spaces and leaves the cursor positioned at the upper left hand corner of the
window. There are other clearing operations, but this one is the simplest and most useful.

From command level we can clear the user_i/o window by:

    window_call clear_window

If we had wanted to clear, say the able window, we would have included the -io_switch
control argument specifying able as the window to operate on.

This same effect, clearing the able window of our examples, can be accomplished from
PL/I by:

call window$_clear_window (menu_window_iocbp, code);
    if code ^= 0 then do;

    process the error

end;

The clear_window operation is useful when an application wants to start with a clean
slate in the user_i/o window. For example, before printing out a description of some menu
option it might be desirable to clear the user_i/o window.
OTHER USEFUL OPERATIONS

Once window status is set, any operation performed on that window (except for a create or destroy operation) returns the status code video_et$_$window_status_pending until the status is reset. To reset the status, perform a get_window_status control order on the window switch. Refer to "Control Operations" for window_io_ later in this manual.

There are many other operations that can be performed on windows using the video system. These are all described in the window_call command in Section 5 or in the window_ subroutine description in Section 6 or the control orders or modes of the window_io_ I/O module in Section 7.
This section discusses the use of function keys and the building of a menu application. It includes a sample exec_com, and PL/I programs. FORTRAN and COBOL programmers refer to Section 8 and Section 9, respectively.

GUIDELINES FOR FUNCTION KEYS

A set of keys that are integral to the menu system are the function keys. These are used to get information, move from one menu to another, or to exit from a menu and return to Multics command level. The reason that the function keys are used at all is to reserve the numbers for the options and also to eliminate the need to include these functions in every list of options in every menu. Ease of use is enhanced when the function keys are doing the same thing from application to application. The following example shows the definitions of the function keys in the "Games" menu.

Press F1 - Gives definitions of the function keys
Press F2 - Returns to the first menu
Press F3 - Goes to the previous menu
Press F4 - Returns to Multics command level

If there are no function keys on the terminal, then the user could type specially assigned keys in sequence. In the following example the escape key has been chosen in conjunction with a letter that is related to the action performed. The selection would then be:

ESC d - Gives definitions of the function keys
ESC f - Returns to the first menu
ESC p - Goes to the previous menu
ESC r - Returns to Multics command level

Since not all terminals have function keys, you must include a call to ttt_info_$function_key_data (described in Multics Subroutines and I/O Modules, Order No. AG93) in your program, which will return information about the terminal being used. This information covers whether or not there are function keys and how many there are.

For those terminals without function keys, or which do not have enough, you must designate keys to be used in their place. It is helpful to the end user if the first of these keys is a "special" key such as the escape key. This should be followed by a regular key that is somewhat related to the action to be performed. You can use a single key, but the advantage of two in sequence is that it does not interfere with the option numbers or letters that have been used. The sequence can also be more than two keys, but the longer it is the greater the chance of typing errors.
Summary of function key recommendations:

- Assign the same meaning to specific keys for every menu.
- Include a call to ttt_info_$function_key_data in your program.
- There is no command level interface to ttt_info_$function_key_data so this cannot be done with exec_com.

If function keys are not available, follow the above suggestions plus:

- Use a combination of characters such that the first character is not the same as any menu option character. A suggestion is using a special key (not @ or #) such as <ESC> in conjunction with a character related to the action performed. For example, <ESC> p for previous menu, or <ESC> r for returning to command level.
- Do not use numbers or single letters as they are reserved for options.
- Do not use two digit numbers because only the first digit is "heard" and an option would therefore be selected. In other words, if you have a function numbered 12 only the first digit is processed so option 1 would be selected.

THE EXEC_COM EXAMPLE

There are four ways in which menu applications may be built: one using exec_com and written in the Multics command language; the others using PL/I, FORTRAN or COBOL programs. The exec_coms provide a quick and easy way to implement very simple menu applications whereas PL/I, FORTRAN or COBOL programs provide for more powerful and robust ones. The Multics menu system provides commands and subroutines to facilitate either type of implementation.

Below is an example of an exec_com interface to the menu system. It is a very simple application and it illustrates how you can begin. The example is a document menu for everyday office use. It is called "Document System". The user will be able to enter, edit, display, print, list or delete documents. The last option available is to exit the document system. So, there are seven options in all and they will be displayed in the top window. Since you will probably want them displayed in the fewest number of lines possible, make space in this window for 6 lines allowing for the headers, the trailers, and the list of menu options printed in two columns. The area from line seven to the end of the screen is the user_i/o window. To see how the standard I/O switch attachments change when you use an exec_com to create a menu, refer to Appendix A, especially Figure A-3. Line numbers are used in this example to indicate new lines, e.g., line 18 is all one line in the exec_com and a new line does not occur till the number 19 appears. Line numbers should not be included in your exec_com.
&version 2
&trace off

&- First we will see if the video system is enabled
&- in the users process. This is done by checking
&- to see if the I/O switch user_terminal is
&- attached. If it isn't we invoke the video
&- system. We need to do this so that we can later
&- return the user to his/her normal environment.

&set already_video &[io attached user_terminal]
&if &[not &[already_video]]
&then window_call invoke

&- Now we will create our demonstration menu. In
&- real applications this menu would most likely be
&- saved in some value segment containing other menus.

menu_create main -option "enter new document"
-option "edit old document" -option "print document on
terminal" -option "print document on printer" -option
"list documents" -option "delete document" -columns 2
-header "<< DOCUMENT SYSTEM >>" -center_headers
-trailer "-" -trailer "USE FUNCTION KEY 1 TO EXIT"
-trailer "-" -center_trailers -pad "-"

&- Here we determine where the windows will go.
&- What we will attempt to do is split the user_i/o
&- window into two windows. The top window is named
&- using a unique name to avoid conflict with other
&- I/O switch names in the process and will contain
&- the menu. The bottom window will be user_i/o.
&- This split of user_i/o is done to allow this
&- application to run while other video applications
&- windows exist on the screen.

&set menu_start &[window_call get_first_line]
&set menu_height &[menu_describe main -height]
&set io_start &[plus &[menu_start] &[menu_height]]
&set io_height &[minus [window_call get_window_height] &[menu_height]]

&- We must have at least 5 lines left in user_i/o.
&- This is an arbitrary limit that this exec_com
&- will enforce.

&if &[nless &[io_height] 5]
&then &do
&print There is not enough room on the screen to run.
&quit
&end

&- Now establish the window to be used to display
&- the menu. It takes its space on the screen from
&- user_i/o, so first shrink user_i/o. The menu
&- window is given a name using the unique active
&- function to avoid conflicts with I/O switch names
&- already in existence.

16  window_call change_window -line &(io_start) -height &(io_height)
17  &set menu_switch &[unique].menu
18  window_call create_window -io_switch &(menu_switch) -line
  &(menu_start) -height &(menu_height)

&- We are now ready to display the menu and get a
&- choice. We must display the menu each time
&- through the loop due to the fact that
&- menu_get_choice will modify the menu display in
&- the window. We will set a local exec_com
&- variable to the choice made just in case we want
&- it in the future (in this example we don't, but
&- its a good idea anyway).

19  &label GET-CHOICE
20  menu_display main -io_switch &(menu_switch)
21  &set choice &[menu_get_choice main -io_switch &(menu_switch)]

&- Now that we have either (1) a valid menu choice
&- in the form of a decimal integer, or (2) a
&- function key selection in the form "F" followed
&- by the function key number, let's perform the
&- requested action.

22  &goto CHOICE-&(choice)

&- This choice is "enter a new document." It will
&- first create the new document and then enter ted
&- to allow entry of the text. Before doing
&- anything, this action, like all others, will
&- clear the user_i/o window. This gives a feeling
&- of starting some new action that we want at this
&- point (this is done for all actions).

23  &label CHOICE-1
24  window_call clear_window
25  io control user_i/o reset_more
26  do "create &1;ted -pn &1" [response "new document name:"
27  &goto GET-CHOICE

&- This choice is "edit an old document." It will
&- enter ted for editing of the requested document.

28  &label CHOICE-2
29  window_call clear_window
30  io control user_i/o reset_more
31  ted -pn [response "old document name:"]
32  &goto GET-CHOICE
&- This choice is "print document on terminal." It
&- will just print the specified document in the
&- user_i/o window.

33 &label CHOICE-3

34 window_call clear_window
35 io control user_i/o reset_more
36 print [response "document name:"] 1
37 &goto GET-CHOICE

&- This action is "print document on printer." It
&- will simply enter a dprint request of the
&- specified document.

38 &label CHOICE-4

39 window_call clear_window
40 io control user_i/o reset_more
41 dprint [response "document name:"]
42 &goto GET-CHOICE

&- This is the "list documents" action. It will
&- simply list the names of all of the documents
&- defined. Our convention for document naming is
&- simple - any single component segment name will
&- do.

43 &label CHOICE-5

44 window_call clear_window
45 io control user_i/o reset_more
46 list * -name -primary
47 &goto GET-CHOICE

&- This is the "delete document" action. It deletes
&- the document specified by the user.

48 &label CHOICE-6

49 window_call clear_window
50 io control user_i/o reset_more
51 delete [response "document name:"]
52 &goto GET-CHOICE

&- This is the action for function key #1. This
&- action exits the document subsystem. At this
&- point we will destroy the menu window and either:
&- (1) return the user_i/o window to its former
&- state, or (2) revoke the window system entirely.
&- This choice is based on whether the video system
&- was in effect when we started this exec_com.

53 &label CHOICE-F1
&if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &if &i...
/* Based */

dcl fkey_data like function_key_data based (function_key_data_ptr);
/* Builtin */

dcl (addr, empty, length, null) builtin;
/* Conditions */

dcl cleanup condition;
/* Entries */

dcl com_err_entry () options (variable);
dcl ioa_entry () options (variable);
dcl ttt_info$function_key_data entry
  (char (), ptr, ptr, fixed bin (35));
dcl unique_chars_entry (bit ()) returns (char (15));
dcl user_info$terminal_data entry (char (), char (), char (), fixed bin, char ());
dcl video_utils$turn_off_login_channel entry (fixed bin (35));
dcl video_utils$turn_on_login_channel entry (char (), fixed bin (35));
/* External */

dcl video_data$terminal_iocb pointer external;
/* Static */

dcl ALTERNATE_F1_STRING char (2) static options (constant) init ("u");
dcl ME char (3) static options (constant) init ("md1");
dcl MIN_USER_I0_HEIGHT fixed bin static options (constant) init (5);
dcl USER_I0 char (8) static options (constant) init ("user_i/o");

  video_was_already_on = (video_data$terminal_iocb ^= null);
  on cleanup call terminate_sys ();
/* Set up the menu. */
/* Invoke the window system if it's not already invoked. */

if ^video_was_already_on then do;
  call video_utils$turn_on_login_channel (code, reason);
  if code ^= 0 then
    call quit (code, reason);
end;

  call window$clear_window (iox$user_io, code);
  if code ^= 0 then
    call quit (code, USER_I0);
/* Create the menu. */
choices (1) = "enter new document";
choices (2) = "edit old document";
choices (3) = "print document on terminal";
choices (4) = "print document on printer";
choices (5) = "list documents";
choices (6) = "delete document";

headers (1) = "<<< DOCUMENT SYSTEM >>>";
trailers (1) = "USE FUNCTION KEY 1 TO EXIT";
trailers (2) = "-";

call user_info $terminal_data (""", term_type, ",", (0), (""""));
call ttt_info $function_key_data (term_type, addr (my_area),
    function_key_data_ptr, (code));

if code ^= 0 then
call quit (code, "Unable to determine terminal type")

/* See if we have to use an escape sequence for F1 */
if fkey_data.highest < 1 then do;
    trailers (1) = "USE ESC-q TO EXIT";
    free fkey_data in (my_area);
    function_key_data.highest = 1;
    allocate fkey_data in (my_area) set (function_key_data_ptr);
    fkey_data.version = function_key_data_version_1;
    fkey_data.seq_ptr = addr (ALTERNATE_F1_STRING);
    fkey_data.seq_len = length (ALTERNATE_F1_STRING);
    do key_shift_idx = 0 to 3;
        fkey_data.home.sequence_length
            (key_shift_idx) = 0;
        fkey_data.left.sequence_length
            (key_shift_idx) = 0;
        fkey_data.up.sequence_length
            (key_shift_idx) = 0;
        fkey_data.right.sequence_length
            (key_shift_idx) = 0;
        fkey_data.down.sequence_length
            (key_shift_idx) = 0;
        fkey_data.function_keys.sequence_length
            (0, key_shift_idx) = 0;
        fkey_data.function_keys.sequence_length
            (1, key_shift_idx) = 0;
    end;
    fkey_data.function_keys.sequence_index (1, KEY_PLAIN) = 1;
    fkey_data.function_keys.sequence_length (1, KEY_PLAIN) =
        length (ALTERNATE_F1_STRING);
end;

my_menu_format.version = menu_format_version_1;
my_menu_format.max_width = 80;
my_menu_format.max_height = 6;
my_menu_format.n_columns = 2;
my_menu_format.center_headers = "|"b;
my_menu_format.center_trailers = "|"b;

3-8 CP51-02
my_menu_format.pad = "O"b;
my_menu_format.pad_char = "_";

my_menu_requirements.version = menu_requirements_version_1;

/* Now we can create the menu. */
call menu_Screate (choices, headers, trailers, addr
    (my_menu_format),
    MENU_OPTION_KEYS, addr (my_area), addr (my_menu_requirements),
    menu_ptr, code);
if code ^= 0 then
    call quit (code, "Unable to create menu.");

/* Now carve the menu I/O window out of the user i/o window.
This program insists that the user i/o window must be at
least 5 lines high after this is done. The menu I/O window
is given a unique name so that this program can be invoked
recursively. */
user_io_window_info.version = window_position_info_version_1;
call iox_Scontrol (iox_Suser_io, "get_window_info",
    addr (user_io_window_info), code);
if code ^= 0 then
    call quit (code, USER_I0);
if user_io_window_info.height
    < my_menu_requirements.lines_needed + MIN_USER_I0_HEIGHT then
    call quit (0, "Window ""user_i/o"" is too small.");

new_window_info.version = window_position_info_version_1;
new_window_info.line =
    user_io_window_info.line + my_menu_requirements.lines_needed;
new_window_info.width = user_io_window_info.width;
new_window_info.height =
    user_io_window_info.height - my_menu_requirements.lines_needed;
call iox_Scontrol (iox_Suser_io, "set_window_info",
    addr (new_window_info), code);
if code ^= 0 then
    call quit (code, "Unable to shrink window ""user_i/o"".");
menu_io_switch_name = "menu_i/o" || unique_chars_ ("0"b);
call iox_SFind_iocb (menu_io_switch_name, menu_io, code);
if code ^= 0 then
    call quit (code, "Unable to get IOCB pointer for menu window.");

new_window_info.line = user_io_window_info.line;
new_window_info.height = my_menu_requirements.lines_needed;

call window_Screate (video_data_terminal_iocb,
    addr (new_window_info), menu_io, code);
if code ^= 0 then
    call quit (code, "Unable to create the menu_i/o window.");
/* Now that we have the window system all set up we can go ahead and display the menu and start processing. */
call menu_$display (menu_io, menu_ptr, code);
if code ^= 0 then
call quit (code, "Unable to display menu.");

/* Now start processing input from the user. */
do while ("1"b);

/* Get an option number or function key value from the user. */
call menu_$get_choice (menu_io, menu_ptr, function_key_data_ptr, fkey, choice, code);

/* Perform an action depending on the user's selection. */
if code ^= 0 then
call quit (code, "Unable to get choice.");
if fkey then
  if choice = 1 then do;
    call terminate_sys ();
    if video_was_alread_on then
call window_$clear_window (iox_$user_io, (0));
  goto EXIT;
end;
else call window_$bell (menu_io, (0));
else do;
  if choice = 1 then
    call terminate_sys ();
  else if choice = 2 then
    call edit_document ();
  else if choice = 3 then
    call display_document ();
  else if choice = 4 then
    call print_document ();
  else if choice = 5 then
    call list_documents ();
  else if choice = 6 then
    call delete_document ();
  else call window_$bell (menu_io, (0));
end;
end; /* do while */
EXIT:
  return;

/* procedures for options */
create_document:
  proc ()
    call ioa_ ("To be provided.");
end create_document;

edit_document:
  proc ()
  call ioa_ ("To be provided.");
end edit_document;

display_document:
  proc ()
  call ioa_ ("To be provided.");
end display_document;

print_document:
  proc ()
  call ioa_ ("To be provided.");
end print_document;

list_documents:
  proc ()
  call ioa_ ("To be provided.");
end list_documents;

delete_document:
  proc ()
  call ioa_ ("To be provided.");
end delete_document;

/* internal procedures */

/* This procedure is called whenever we must leave the subsystem we have set up (if an error occurs or the user wants to leave). It rearranges things back to the way they were before. */

terminate_sys:
  proc ()
  if menu_io ^= null () then
    call window_destroy (menu_io, (0));
  if video_was_already_on then
    call iox$control (iox$user io, "set_window_info", 3-11
addr (user_io_window_info), (0));
else call video_utils_turn_off_login_channel ((0));
end terminate_sys;
quit:
proc (code, explanation);
dcl code fixed bin (35);
dcl explanation char (*);
call terminate_sys();
call com_err_ (code, ME, explanation);
go to EXIT;
end quit;
%include iox_dcls;
%page;
%include window_dcls;
%page;
%include function_key_data;
%page;
%include menu_dcls;
%page;
%include window_control_info;
end mdl;
This section describes the Multics Video System. The Multics Video System is an upwards compatible extension to the Multics Communications System. The basic features of the Multics Video System are:

- Dividing the user's terminal into one or more windows. Windows are described in detail in Section 2 of this manual.

- A powerful real-time editor for input lines. The erase and kill characters take effect as soon as they are typed. Additional characters allow the user to delete words and to retrieve deleted text.

- Flexible control over output. When a window is full of output it can scroll (removing lines from the top of the window, adding new ones to the bottom), or wrap (output begins at the top of the window, optionally clearing the window first).

- MORE Processing. The video system pauses when a window is full of output until the user indicates that the window has been read. This is an extension to End Of Page processing. The user can also choose to discard unseen all pending output.

REAL-TIME EDITING

Real-time editing is markedly different from usual Multics editing. All editing requests take effect immediately. The screen changes to show the effect of the characters or lines deleted. In addition, the set of editing characters expands to include several control characters.

Control characters are characters entered using the control key. The control key is a key that acts like the shift key. By itself it generates no characters; it is used to change the meaning of some other key. When the key "A" is typed while the control key is held down, the character sent by the terminal is control A, which is written as ^A. The control characters are the first 32 ASCII characters, 000 through 037 octal.

Alphabetic characters are given in capitals, but either an upper or lower case letter (as for N or n) can be used with default escape sequences. If an upper case letter is used with a user-defined sequence, both the upper and lower case keys must be bound in order for both keys to work. The letters ESC represent the escape key. For ESC F, you would press the escape key, release it and type an f or F.
Although most Multics users keep the system default erase (#) and kill (@) symbols, the video system recognizes and then assumes the values of any erase and kill characters that may have been set via the set_tty command.

The Erase Character

The erase character removes the character to the left of the cursor. The cursor moves to the left, and exactly one character is deleted. This is different from usual Multics editing where an erase character typed after white space deletes all whitespace, and otherwise deletes all characters from a column position. The erase character is settable for each window. In addition, the DEL character (\177) and the backspace character (\010) are always erase characters.

The Kill Character

The kill character deletes the entire line to the left of the cursor. The cursor then goes back to the beginning of the line. Again, this happens immediately. The deleted line is saved, and can be recovered. See "Retrieving Deleted Text" below. The kill character is settable per-window.

The Line Editor

Additional editing is possible using sequences of one and two characters. The two-character sequences all begin with the ASCII ESC character, (^[, octal 033), which is not the same as the Multics input escape character ("\").

MOVING THE CURSOR

The line editor can move the cursor forward or backward within the current line while repositioning the cursor either a character at a time or a word at a time. A word is an unbroken string of uppercase and lowercase alphabetics, numerals, underscores, backspace characters, and hyphens. (This is the default definition of a word, which can be changed with the set_token_delimiters order, described in the window_io_ writeup.) The cursor can also move explicitly to the beginning or the end of the current line. The requests that perform these actions are listed under "Other Editor Requests" below.

DELETING CHARACTERS AND WORDS

The line editor can delete a single character or an entire word at a time. Various editing requests described below can delete the character or word immediately to either the left or the right of the cursor. The deleted text (only words, not characters) is saved and can be retrieved. For example, typing ESC DEL (or ESC followed by the current erase character) deletes the word to the left of the cursor. The word is saved on the kill ring (see below).
RETRIEVING DELETED TEXT

Text deleted by the word and line kill characters is saved, and can be restored. The text is saved on a kill ring. A kill ring is a set of kill slots. Each slot holds deleted text. Successive word kills share one kill slot, so if several words are deleted one after another, all of them will be retrieved by a single retrieve command.

Deleted text is saved with previously deleted text if two delete characters are typed in succession. If intervening characters are typed, the kill ring is rotated: a new slot is selected to hold saved text.

Text is entered when the user types text followed by a carriage return. Each input line is added to the kill ring. This provides editing of the previous input line.

The following control characters are used to retrieve deleted text:

\[^Y\] (or yank) retrieves deleted text from the kill ring. This is the only way to recover from an erroneous kill character.

\[\text{ESC } Y\] can be typed only after either \[^Y\] or \[\text{ESC } Y\]. It deletes the text just retrieved, without saving it on the kill ring, rotates the ring (to the next most recently killed text) and retrieves the text from the new top slot.

The following example is given in triplets. The first line shows what the user types, the second line shows what one line of the display looks like afterwards, and the third line (or lines) shows the kill ring. The top item on the kill ring is at the top of the column.

User Types: This is a sentence  
Display is: This is a sentence  
Kill Ring: <empty>

NOTE: The kill ring is empty because the user has just invoked the video system.

User Types: \[\text{ESC } \text{DEL}\]  
Display is: This is a  
Kill Ring: sentence

One word is deleted, and it begins the kill ring.

User Types: \[\text{ESC } \text{DEL}\]  
Display is: This is  
Kill Ring: a sentence
Another word is deleted; it is merged into the same kill slot.

User Types: an example sofa

Display is: This is an example sofa
Kill Ring: a sentence

User Types: ESC DEL
Display is: This is an example
Kill Ring: sofa
     a sentence

This deleted word is not merged, because there has been typing since the last kill command.
There are now two slots on the kill ring.

User Types: of ^Y
Display is: This is an example of sofa
Kill Ring: sofa
     a sentence

The top kill slot is yanked back.

User Types: ESC Y
Display is: This is an example of a sentence
Kill Ring: a sentence
     sofa

The kill ring is rotated; the previously yanked contents are deleted from the line, and the
new top item from the ring is yanked to replace it.

If a carriage return were typed at the end of "This is an example of a sentence", the
kill ring would then contain a new slot containing the entire input line.

Other Editor Requests

Alphabetic characters are given in capitals, but either an upper or lower case letter
(ESC F or ESC f) can be used. The following control characters are also recognized by the
line editor:

^L                   Clears the window and redisplays the input line.
^Q                   "quotes" the next character, causing it to have no special
                     meaning. This is useful for entering control characters. It serves
                     some of the same purposes as the input escape character (^).
^F                   moves the cursor forward one character.
^B                   moves the cursor backward one character.
ESC F                moves the cursor forward one word.
ESC B                moves the cursor backward one word.
A moves the cursor to the beginning of the current line.

E moves the cursor to the end of the current line.

D deletes the current character (deletes forward).

DEL, # deletes the character to the left of the cursor (deletes backward).

ESC D deletes the current word (deletes forward).

ESC DEL, ESC # deletes the word to the left of the cursor (deletes backward).

ESC C capitalize initial word.

ESC U capitalize word.

ESC L lower case word.

ESC T twiddle word.

By default, no other control characters have meaning. If any are typed, the only action they cause is an audible alarm. You can create additional editor requests by writing PL/I programs that conform to a standard calling sequence (see "Writing Editor Extensions").

The set of characters used to define a word for control characters such as ESC F can be changed via the set_token_characters control order. See the description in the window_io_ module later in this manual.

Writing Editor Extensions

The video system provides a full input line editor, including the ability to edit in the middle of the line. Of course, there are many potential editor functions that people might like to use (see the Emacs Text Editor User's Guide Order No. CH27), and not all of these are provided. Rather than attempt to anticipate every possible editor request, the video system allows users who are familiar with PL/I to write their own editor requests and associate sequences of keystrokes (key bindings) with these requests.

The key binding mechanism can be used for a wide variety of applications. Since editor requests are executed immediately by single or multiple keystroke sequences, highly interactive facilities can be built into the input line editor.

LINE EDITOR ROUTINES

Editor request routines are PL/I programs that conform to a standard calling sequence. The request procedure is given complete control of the input buffer and can add or delete characters or modify the current contents of the buffer. The video system editor's redisplay facility manages all display updates; the individual editor routines need no knowledge of the video environment or the screen contents.
A library of editor utility routines is provided (see "Editor Utilities"). These can be called by user-written editor routines to perform such actions as insertion and deletion of text from the buffer, manipulation of the kill ring, and manipulation of words within the input buffer.

A line editor routine is declared as follows:

**USAGE**

dcl twiddle_words entry (pointer, fixed bin(35));
call twiddle_words (line_editor_info_ptr, code);

**ARGUMENTS**

line_editor_info_ptr
   is a pointer to the line_editor_info data structure (described below).

code
   is a standard status code. (Output) If the status code returned by the editor routine is error_table$action_not_performed, the editor will ring the terminal bell to indicate that the editor routine was used improperly. Any other code will reported in a more drastic manner, via the sub_err_ mechanism.

The line_editor_info structure (declared in window_line_editor.incl.pl1) is declared as follows:

dcl line_editor_info aligned based (line_editor_info_ptr),
   2 version char(8),
   2 iocb_ptr pointer, /* to current window */
   2 repetition_count fixed bin,
   2 flags,
      3 return_from_editor bit(1) unaligned,
      3 merge_next_kill bit(1) unaligned,
      3 old_merge_next_kill bit(1) unaligned,
      3 last_kill_direction bit(1) unaligned,
      3 numarg_given bit(1) unaligned,
      3 suppress_redisplay bit(1) unaligned
   3 pad bit(30) unaligned,
   2 user_data_ptr pointer, /* for user state info */
   2 cursor_index fixed bin(21),
   2 line_length fixed bin(21),
   2 input_buffer character(1024) unaligned;
   2 key_sequence character(128);

dcl line_editor_input_line char(line_editor_info.line_length) based (addr (line_editor_info.input_buffer));

dcl line_editor_info_version_2 char(8) static options (constant) init ("lei00002");
**STRUCTURE ELEMENTS**

version
is string for this structure. (Input) The current version string, "lei00002", is the value of the variable line_editor_info_version_2, declared in the same include file.

iocb_ptr
is the pointer to the current window. (Input)

repetition_count
is the value of the numeric argument specified by the user, and is undefined if no numeric argument was specified (i.e., numarg_given flag = "0"b). (Input)

return_from_editor
is a flag which is set by the editor routine if the editor invocation is to be terminated and the input line returned to the caller. The input buffer is redisplayed before the buffer is returned to the caller, unless overridden by the line_editor_info.suppress_redisplay flag.

merge_next_kill
is a flag which should be set when text is deleted and added to the kill ring if subsequent deletions are to be added to the same kill ring element. (Input/Output) This flag is managed by the editor utility routines. If the editor utility routines are used for all input buffer modifications, the user-written editor routine need never set this flag.

old_merge_next_kill
is an internal editor state flag and should not be modified. (not used)

last_kill_direction
direction of last kill (forward or backward).

numarg_given
is "1"b (i.e. true) if a numeric argument was supplied by the user via ESC-NNN or ^U.

suppress_redisplay
is a flag that stops the redisplay of the input buffer when line_editor_info.return_from_editor is set.

pad
reserved for future use.

user_data_ptr
points to a user data structure which the video system ignores, other than passing this pointer to requests that follow.

cursor_index
is the index of the character in the input buffer on which the cursor is currently located. (Input/Output) This index must be updated if characters are added or deleted before the cursor, or the cursor is moved by the editor routine. The cursor index must be no larger than one greater than the input_line_length. If the editor utility routines are used for all input buffer manipulations, the cursor_index will be updated appropriately.
line_length
is a count of the number of characters in the current input line. (Input/Output) This
variable must be updated if any characters are inserted or deleted from the input buffer.
The value of the line_length variable must always be non-negative, and must never be
larger than the length of the input buffer. If the line editor utility routines are used for
all input buffer manipulations, the line_length variable will be updated automatically.

input_buffer
is a character string containing the current input line. (Input/Output) Any manipulations
may be performed on this string by the editor routine. It is recommended that the
editor utility routines be used for all insertions and deletions to ensure that the various
state variables and flags remain consistent. The line_editor_input_line variable can be
used to address the valid part of the input buffer as a string.

key_sequence
A character string that contains the sequence of key strokes that invoked this editor
routine.

Window Editor Utilities
As was mentioned above, a library of editor utility routines is provided for the benefit
of user-written editor routines. Some operations can be performed simply by a user-written
editor routine. For example, to position the cursor to the end of the line, simply set the
cursor_index variable to one greater than the value of the line_length variable. However,
most actions are more complex than this and it is recommended that the editor utility
routines be used to perform most operations. The following is a description of these routines.
In all cases, line_editor_info_ptr is the pointer to the editor data structure that is supplied as
an argument to user-written editor routines.

dcl window_editor_utils_$insert_text entry (ptr, char(*), code);
call window_editor_utils_$insert_text (line_editor_info_ptr, "text", code);

Inserts the supplied character string into the input buffer at the current cursor
location. If the string is too large to fit in the remaining buffer space, the code
error_table_$action_not_performed is returned. This routine updates the line_length
field of the line_editor_info structure, and the cursor_index if necessary.

dcl window_editor_utils_$delete_text entry (ptr, fixed bin, code);
call window_editor_utils_$delete_text (line_editor_info_ptr, count, code);

Deletes a specified number of characters (supplied by the variable count) from the
input buffer at the current cursor location. If there are not enough characters
remaining between the cursor and the end of the line, error_table_$action_not_performed
is returned and no characters are deleted. The line_length component of the
line_editor_info_structure is updated, and the cursor_index if necessary.
dcl window_editor_utils_$delete_text_save entry (ptr, fixed bin, bit(1), code);
call window_editor_utils_$delete_text_save (line_editor_info_ptr, count, kill_direction, code);

This entrypoint is identical to delete_text, but the deleted text is added to the kill ring. The kill_direction flag is used during kill merging to decide whether the killed text will be concatenated onto the beginning or end of the current kill ring element. "1"b is used to specify a forward kill (e.g. FORWARD_DELETE_WORD), "0" a backward kill.

dcl window_editor_utils_$move_forward entry (ptr, fixed bin, code);
call window_editor_utils_$move_forward (line_editor_info_ptr, count, code);

Advances the cursor forward a specified number of characters (supplied by the variable "count") in the input line. If there are not enough characters between the cursor and the end of the line, error_table_$action_not_performed is returned.

dcl window_editor_utils_$move_backward entry (ptr, fixed bin, code);
call window_editor_utils_$move_backward (line_editor_info_ptr, count, code);

Moves the cursor backward a specified number of characters (supplied by the variable "count") in the input line. If there are not enough characters between the cursor and the end of the line, error_table_$action_not_performed is returned.

dcl window_editor_utils_$move_forward_word entry (ptr, code);
call window_editor_utils_$move_forward_word (line_editor_info_ptr, code);

Updates the cursor_index to a position after the next word (or token) in the input line. A word is defined via the editor's set of token delimiters, set via the set_token_delimiters control order.

dcl window_editor_utils_$move_backward_word entry (ptr, code);
call window_editor_utils_$move_backward_word (line_editor_info_ptr, code);

Updates the cursor_index to a position before the preceding word (or token) in the input line. A word is defined via the editor's set of token delimiters, set via the set_token_delimiters control order.

dcl window_editor_utils_$get_top_kill_ring_element entry (ptr, char(*), fixed bin 35);
call window_editor_utils_$get_top_kill_ring_element (line_editor_info_ptr, text code),

Returns the top kill ring element.
Rotates the kill ring.

End-Of-Window Processing

When output has filled a window, old lines must be removed to make way for new ones. This is usually done by scrolling old lines off the top of the window. But for windows that cannot be scrolled (usually because the terminal cannot scroll) it is possible to move the cursor back to home, and output new lines overwriting the old ones. This is known as wrapped output. A variation on wrapped output is to clear the window after moving the cursor home. The action taken when a window is full is controlled on a per-window basis by any one or the following more_mode modes:

- clear the window is cleared, and output starts at the home position.
- fold output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response when it is about to overwrite the first line written since the last read or MORE break.
- scroll lines are scrolled off the top of the window, and new lines are printed in the space that is cleared at the bottom of the screen. This is the default for all terminals capable of scrolling (i.e., those terminals that have the capability to insert and delete lines).
- wrap output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response at the bottom of every window of output. This is the default for terminals incapable of scrolling.

For more information, refer to window_io_ in Section 7.

MORE PROCESSING

As lines are displayed in the window, old lines are scrolled off the top of the window or otherwise removed. When output would cause a line to be removed that has been displayed since the most recent input, it is assumed that the user may not have had a chance to read it, and MORE processing occurs. The question "MORE?" appears on the screen, and no further output occurs until the user indicates that pending output is to be either displayed or discarded. MORE processing is controlled by the "more" mode, which is enabled by default.

Output resumes if the user strikes CR, and is discarded if the user strikes DEL. The characters used can be set by a control order. Type ahead characters are not seen by MORE processing. The response to MORE must be typed after the prompt appears. All other characters are buffered to be returned later.
When output is discarded, the video system simply ignores output until a get_line or get_chars call is made, a "reset_more" control order call is made, or the window is cleared, or the cursor is moved to home. WARNING: a prompt sent just before a get_line call will not be printed if output is discarded, unless the prompting program first issues a "reset_more" control order (or otherwise resets more processing).

OUTPUT BUFFERING

The video system sometimes buffers output internally, sending it to the terminal when certain internal conditions are satisfied. All buffered output is sent to the terminal whenever an input call is made (e.g., window_$get_echoed_chars). This ensures that all output, including prompts, is seen by the user before input is read. An application program that calls window_entrypoints directly should take this buffering into account to perform correctly. If it is necessary to send output to the terminal when no read request is to be done (e.g., displaying an incremental message during a long computation), the application should call window_$sync on the I/O switch after the output has been requested (e.g., via a call to window_$overwrite_text). See the description of window_$sync in the window_subroutine description later in this manual.

SUBROUTINE INTERFACE

The video system provides a standard set of operations for windows available through the window_subroutine. Some terminals are not capable of supporting all of these operations. In addition, the standard iox_operations of get_line, get_chars, and put_chars are provided. Some manipulations on windows are made via iox_control orders (the window_io_description in Section 7). Some of these are compatible with existing tty_control orders. The iox_ and tty_subroutines are both described in the Multics Subroutines and I/O Modules manual, Order No. AG93. Other manipulations control features that are specific to the window environment.

The iox_operations are defined in terms of the more primitive window_operations. For example, the window_primitive, window$_overwrite_text, can only display a string of characters that fit on a terminal line. The iox$_put_chars wraps long strings onto multiple lines, and displays control characters with the conventional octal representation. For this reason special care must be taken when using window_applications on a window when iox_operations are in use as well. For more details see the description of the reset_more control order in the window_io description in Section 7.

COMMAND LINE INTERFACE

The command level interface to the video system is the window_call command. This command can perform most of the operations on a window supported by window_directly from command level. The window_call command is described in Section 5.
This section contains descriptions of the commands used by the menu and video software, presented in alphabetical order.
menu_create

Name: menu_create

SYNTAX AS A COMMAND

menu_create menu_name {-control_args}

FUNCTION

The menu_create command creates a menu description, assigns it a specified name, and stores the description in a segment. The menu description may be used with other menu commands, active functions, and subroutines.

ARGUMENTS

menu_name

is the name assigned to the menu when it is stored.

CONTROL ARGUMENTS

-pathname PATH, -pn PATH

is the pathname of the segment in which the menu is stored. Menus are stored in value segments. If the specified segment does not exist, the user is asked argument). The value suffix is assumed. If this control argument is omitted, the user's default value segment (>udd>Project_id>Person_id>Person_id.value) is used to store the menu.

-brief, -bf

means that if the segment specified by the -pathname control argument does not exist, it is to be created without querying the user.

-option STR, -opt STR

specifies a menu option. The options appear in the menu in the order given. At least one option must be supplied. If STR contains blanks or special characters, it must be quoted.

-header STR, -he STR

specifies a line of header. All header lines specified appear in the menu in the order given. If STR contains blanks or special characters, it must be quoted.

-trailer STR, -tr STR

specifies a trailer line. All trailers appear in the menu in the order given. If STR contains blanks or special characters, it must be quoted.

The remaining control arguments control the format of the menu. All are optional.

-columns N, -col N

where N is a positive decimal integer, sets the number of columns in the menu to N. The default is one column.

-center_headers, -ceh

does nothing.
-no_center_headers, -nceh
   causes header lines to be flush left. This is the default.

-center_trailers, -cet
   causes all trailer lines to be centered.

-no_center_trailers, -ncet
   causes trailer lines to be flush left. This is the default.

-option_keys STR, -keys STR
   specifies the keystrokes to be associated with each option. Each character in STR is
   associated with the corresponding option, so that if it is typed, the corresponding option
   is selected. There must be at least as many characters in STR as there are options. If
   this control argument is not given, the string "123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" is used.

-pad C
   where C is one character, specifies the padding character for centering. The default is
   the space character.

-line_length N, -ll N
   where N is a positive decimal integer, specifies the line length for the menu. If not
   supplied, the line length will be the line length of the user's terminal at the time the
   command is invoked.

ACCESS REQUIRED
The user must have r and w access on the value segment.

EXAMPLES
The following example sets up a small menu named compile.

menu_create compile -pn [pd]>temp -pad = -he "SAMPLE MENU" -tr = -ceh -cet
   -columns 2 -ll 78 -opt "Compile with No Options"
   -opt "Symbol Table" -opt "Profile Info"

Creates a menu that looks like this:

=================================SAMPLE MENU==================================
| (1) Compile with No Options | (3) Profile Info  |
| (2) Symbol Table             |

==============================================================================
Name: menu_delete

SYNTAX AS A COMMAND

menu_delete menu_name {-control_arg}

FUNCTION

The menu_delete command deletes a menu description from a specified value segment.

ARGUMENTS

menu_name

is the name that was assigned to the menu when it was stored.

CONTROL ARGUMENTS

-pathname PATH, -pn PATH

is the pathname of the value segment in which the menu is stored. If this control argument is not given, the user's default value segment is searched for the menu. The value suffix is assumed.
Name: menu_describe

SYNTAX AS A COMMAND

menu_describe menu_name {-control_args}

SYNTAX AS AN ACTIVE FUNCTION

[menu_describe menu_name -control_args]

FUNCTION

The menu_describe command prints or returns information about a menu.

ARGUMENTS

menu_name

is the name that was assigned to the menu when it was stored.

CONTROL ARGUMENTS

-count, -ct
returns the number of options defined in the menu.

-height
returns the height of the menu.

-pathname PATH, -pn PATH
is the name of the value segment in which the menu has been stored. The value suffix is assumed. If this control argument is omitted, the user's default value segment is searched for the menu.

-width
returns the width of the menu.

NOTES

When used as an active function, exactly one of -count, -height, or -width must be given.

As a command, any number are allowed. If none are given, all attributes are displayed.
Name: menu_display

SYNTAX AS A COMMAND

menu_display menu_name {-control_args}

FUNCTION

The menu_display command displays a menu in a window.

ARGUMENTS

menu_name
    is the name that was assigned to the menu when it was stored.

CONTROL ARGUMENTS

-io_switch STR, -is STR
    specifies the name of an I/O switch for a window. The default is user_output.

-pathname PATH, -pn PATH
    is the name of the value segment in which the menu has been stored. The value suffix is assumed. If this control argument is omitted, the user's default value segment is searched for the menu.
Name: menu_get_choice

SYNTAX AS A COMMAND

menu_get_choice menu_name {-control_args}

SYNTAX AS AN ACTIVE FUNCTION

[menu_get_choice menu_name {-control_args}]

FUNCTION

The menu_get_choice command, given the menu called menu_name on display in a window, gets a menu choice from the user and prints or returns it.

ARGUMENTS

menu_name

is the name that was assigned to the menu when it was stored.

CONTROL ARGUMENTS

-pathname PATH, -pn PATH

is the name of the value segment in which the menu has been stored. The value suffix is assumed. If this control argument is omitted, the user's default value segment is searched for the menu.

-io_switch STR, -is STR

where STR is the name of an I/O switch for a window. The default is user_i/o.

-default_fkeys STR, -fkeys STR

specifies the keys to be used if the terminal does not have function keys or if the terminal does not have the proper set of function keys. See "Notes on Function Keys" below.

-function_keys STR, -fkeys STR

specifies the keys to be used to simulate function keys. This control overrides any function key definitions already established for the terminal. See "Notes on Function Keys" below.
NOTES ON FUNCTION KEYS

Many terminals have function keys. On many of these terminals (such as the Honeywell VIP7801) they are labelled "F1", "F2", etc. If you type one of these function keys, menu_get_choice returns the string "F*", where * is a one or two digit number signifying which function key was pressed. It is possible to specify your own set of keystrokes to be used in lieu of the terminal's function keys, or to specify a set of keystrokes to be used if the terminal does not have enough function keys. These are done by using the -fkeys and -dfkeys control arguments. Each of these control arguments is followed by a string. Each character in this string is used to simulate a function key. The first character is used to simulate function key 0, the next to simulate function key 1, etc. To simulate a given function key, type esc-C, where C is the character corresponding to the function key. Thus if the string is "0123456789", typing esc-2 will return F2.

The -fkeys control argument is used to specify keystrokes to be used instead of any which might be defined for the terminal. If this control argument is given, then the simulation of function keys always takes place.

The -dfkeys control argument is used if you want to use the terminal defined function keys if possible, but wish to specify key sequences to be used to simulate function keys if necessary. Each character in the string following -dfkeys corresponds to one function key. If the character is a space, it means it makes no difference if the terminal has a function key corresponding to that position. If the character is not a space, that character will be used to simulate a function key if necessary. If the terminal does not have a function key for every non-space character in the string, then the -dfkeys string is used to simulate function keys. Thus, the string " lp q" means that you do not care whether the terminal has a function key 0 or a function key 3, but you wish to use function keys 1,2, and 4. If any of these 3 function keys is not present on the terminal, then esc-? will substitute for F1, esc-p will substitute for F2, and esc-q will substitute for F4.
Name: menu_list

SYNTAX AS A COMMAND

menu_list {menu_starname} {-control_arg}

SYNTAX AS AN ACTIVE FUNCTION

[menu_list {menu_starname} {-control_arg}]

FUNCTION

The menu_list command lists the names of the menu descriptions stored in a value segment.

ARGUMENTS

menu_starname

is a starname that is used to search for menu descriptions. If it is omitted, the default

is **.

CONTROL ARGUMENTS

-pathname PATH, -pn PATH

is the pathname of the value segment in which the menu has been stored. The value

suffix is assumed. If this control argument is not given, the user's default value segment

(udd>Project_id>Person_id>Person_id.value) is searched for the menu.
Name: window_call

SYNTAX AS A COMMAND

window_call arguments {-control_args}

SYNTAX AS AN ACTIVE FUNCTION

[window_call arguments {-control_args}]

FUNCTION

The window_call command provides a command interface to the video system.

ARGUMENTS

are listed below. A detailed description follows the control arguments section.

- bell
- change_window, chgwd
- clear_region, clrnn
- clear_to_end_of_line, cleol
- clear_to_end_of_window, cleowd
- clear_window, clwd
- create_window, crwrd
- delete_chars, dlch
- delete_window, dlwd
- get_echoed_chars, gech
- get_first_line, gfl
- get_one_unechoed_char, gouch
- get_position, gpos

- get_terminal_height, gtmhgt
- get_terminal_width, gtmwid
- get_unechoed_chars, guch
- insert_text, itx
- invoke
- overwrite_text, otx
- revoke
- scroll_region, scrgn
- set_position, spos
- set_position_re1, sposrel
- supported_terminal
- sync
- video_invoked
- write_sync_read, wsr

CONTROL ARGUMENTS

- -column C, -col C
  specifies a column on the screen. The leftmost column is 1. If -column is not specified, the default is the remainder of the screen.

- -count N, -ct N
  specifies a count. See the specific requests for details.

- -height NL, -hgt NL
  specifies the height of a region or a window for a request. If -height is not specified, the default is the remainder of the screen.
-io_switch WINDOW, -is WINDOW
where WINDOW is an I/O switch. The operation is performed on the window associated
with the named I/O switch.

-line L
specifies a line on the screen. The top line is line 1.

-line_speed N, -ls N
specifies the speed of the terminal's connection to Multics. N is in characters per
second.

-terminal_type STR, -ttp STR
where STR is a terminal type. Information on accepted terminal types can be obtained
with the print_terminal_types (ptt) command.

-string TEXT, -str TEXT
specifies a text string for display. If TEXT contains blanks or other special command
processor characters it must be enclosed in quotes.

-width NC, -wid NC
specifies the width of a region for a request. If -width is not specified, the default is
the remainder of the screen.

Argument Descriptions

bell

SYNTAX AS A COMMAND

wdc bell { -io_switch WINDOW }

FUNCTION

activates the terminal bell. On some terminals, this may produce a visual indication instead of
an audible tone. The cursor position must be defined. The cursor is positioned to the current
position of the specified window, if it is elsewhere on the screen. If -io_switch is not
specified, user_i/o is assumed.
change_window, chgwd

SYNTAX AS A COMMAND

wdc chgwd {-line L} {-column C} {-height NL} {-width NC} {-io_switch WINDOW}

FUNCTION

changes the origin or size of the specified window. At least one of -line, -column, -height, or -width is required. -line L specifies the line number or the screen where the window is to begin. If -line is not supplied, the default is column 1. If only the -line control argument is given (changing the top line of the window), the window length is automatically adjusted. That is, if the -line control argument increases the value of the top line number (moving the window down), the window length shrinks accordingly. However, if the -line control argument decreases the top line number (moving the window up), the length remains the same. -column C specifies the column number on the screen where the window is to begin. If -column is not supplied, the default is column 1. If only the -column control argument is given (changing the first column on the left), the window width is automatically adjusted. -height NL specifies the height of the window. If height is not specified, the default is the remainder of the screen. If only the -height control argument is given (changing the window length), the origin line remains the same. -width NC specifies the width of the window. If width is not specified, the default is the remainder of the screen. If only the -width control argument is given (changing the window width), the origin column remains the same. If -io_switch is not specified, user_i/o is assumed. See Section 2 for more information on the use of this command.

clear_region, clrgn

SYNTAX AS A COMMAND

wdc clrgn -line N -column N -height N -width N {-io_switch WINDOW}

FUNCTION

clears the specified rectangular region of the window to blanks. The region may be part or all of the window. If -io_switch is not specified, user_i/o is assumed. See Section 2 for more information on the use of this command.

clear_to_end_of_line, cleol

SYNTAX AS A COMMAND

wdc cleol {-io_switch WINDOW}
clears the line from the current cursor position to the end of the line to blanks. The current cursor position must be defined. If -io_switch is not specified, user_i/o is assumed.

clear_to_end_of_window, cleowd

SYNTAX AS A COMMAND

wdc cleowd {-io_switch WINDOW}

FUNCTION

clears the window from the current cursor position to the end of the window to blanks. The current cursor position must be defined. If -io_switch is not specified, user_i/o is assumed.

clear_window, clwd

SYNTAX AS A COMMAND

wdc clwd {-io_switch WINDOW}

FUNCTION

clears the specified window so that its content becomes entirely blank. The current cursor position is defined to be at Line 1, Column 1 of the specified window. If -io_switch is not specified, user_i/o is assumed. See Section 2 for more information on the use of this command.

create_window, crwd

SYNTAX AS A COMMAND

wdc crwd -io_switch WINDOW {-line L -column C -height NL -width NC}

FUNCTION

creates a new window on the screen with name (and I/O switch) WINDOW. -line L specifies the line number on the screen where the window is to begin. If -line is not supplied, the default is line 1. -column C specifies the column number on the screen where the window is to begin. If -column is not supplied, the default is column 1. -height NL specifies the height of the window. If -height is not specified, the default is the remainder of the screen. -width NC specifies the width of the window. If -width is not specified, the default is the remainder of the screen. The window is blank when created, and the cursor position is Line 1, Column 1 of the new window. See Section 2 for more information on the use of this command.
delete_chars, dlch

SYNTAX AS A COMMAND

wdc dlch -count N {-io_switch WINDOW}

FUNCTION

deletes N characters to the right of the current cursor position on the current line. The cursor remains stationary; characters to the right of the deleted characters move to the left to fill the vacated space. The current cursor position must be defined. If -io_switch is not specified, user_i/o is assumed.

delete_window, dlwd

SYNTAX AS A COMMAND

wdc dlwd -io_switch WINDOW

FUNCTION

destroys the specified window. The I/O switch is closed and detached. See Section 2 for more information on the use of this command.

get_echoed_chars, gecho

SYNTAX AS A COMMAND

wdc gecho -count N {-io_switch WINDOW}

FUNCTION

reads characters from the terminal until either N characters or a break character is read. All characters except the break are echoed on the screen in the current window. For information on break characters, see the break_table control order in the description of window_io_ in Section 7. The current cursor position must be defined. If -io_switch is not specified, user_i/o is assumed.
window_call

ACTIVE FUNCTION USAGE

two strings are returned. The first contains any nonbreak characters read, and the second contains the break character, if any.

get_first_line, gfl

SYNTAX AS A COMMAND

wdc gfl {-io_switch WINDOW}

FUNCTION

prints the line on the screen where the specified window begins. If -io_switch is not specified, user_i/o is assumed.

ACTIVE FUNCTION USAGE

returns the line on the screen where the specified window begins. If -io_switch is not specified, user_i/o is assumed.

get_one_unechoed_char, gouch

SYNTAX AS A COMMAND

wdc gouch {-io_switch WINDOW}

FUNCTION

reads a single unechoed character from the terminal. If -io_switch is not specified, user_i/o is assumed.

ACTIVE FUNCTION USAGE

returns a single unechoed character from the terminal.

get_position, gpos

SYNTAX AS A COMMAND

wdc gpos {-io_switch WINDOW}

FUNCTION

prints the current line and column position of the cursor.
*ACTIVE FUNCTION USAGE*

returns the line and column position as a pair of integers separated by a space. If `-io_switch` is not specified, `user_i/o` is assumed.

get__terminal__height, gtmhgt

*SYNTAX AS A COMMAND*

`wdc gtmhgt`

*FUNCTION*

prints the total number of lines on the user's terminal.

*ACTIVE FUNCTION USAGE*

returns the total number of lines on the user's terminal.

get__terminal__width, gtmwid

*SYNTAX AS A COMMAND*

`wdc gtmwid`

*FUNCTION*

prints the total number of columns on the user's terminal.

*ACTIVE FUNCTION USAGE*

returns the total number of columns on the user's terminal.

get__unechoed__chars, guch

*SYNTAX AS A COMMAND*

`wdc guch -count N [-io_switch WINDOW]`

*FUNCTION*

reads characters from the terminal until either N characters or a break character are read. The current cursor position must be defined. If `-io_switch` is not specified, `user_i/o` is assumed.
ACTIVE FUNCTION USAGE

returns two strings. The first contains any nonbreak characters read, and the second contains the break character, if any.

get_window_height, gwdhgt

SYNTAX AS A COMMAND

wdc gwdhgt {-io_switch WINDOW}

FUNCTION

prints the height of the specified window.

insert_text, itx

SYNTAX AS A COMMAND

wdc itx -string window {-io_switch WINDOW}

FUNCTION

displays the text string window at the current cursor position. If there are any characters to the right of the current position on the current line, they are moved to the right to accommodate the new string. There is no wraparound feature; if text goes off the screen it is dropped. The text string window may contain only printable ASCII characters. Use the io_call put_chars command to display nonprintable characters in a readable form. If -io_switch is not specified, user_i/o is assumed.

invoke

SYNTAX AS A COMMAND

wdc invoke {-line_speed N, -ls N}

FUNCTION

activates the video system on the user's terminal. If no line speed is specified, the current line speed is used. The user's terminal must be attached with the tty I/O module. If graphics or auditing are in use they must be removed before this command is given. The settings of the following tty_ modes are copied when the video system is invoked: vertsp, can, erkl, esc, red, and ctl_char. In addition, if ^pl is set on video system invocation, ^more will be set in the video system. (For more details on modes, see the window_io_ I/O module in Section 7.) Similarly, the settings of the current erase and kill characters are copied when the video system invoked. (See "Real-Time Editing" in Section 4 for details.) See Section 2 for more information on the use of this command.
overwrite_text, otx

**SYNTAX AS A COMMAND**

wdc otx -string STR {-io_switch STR}

**FUNCTION**

displays the text string STR at the current cursor position in the window. If there is any text to the right of the current position in the window, it is overwritten with the supplied string. The text string STR may contain only printable ASCII characters. Use the io_call put_chars command to display nonprintable characters in a readable form. If -io_switch is not specified, user_i/o is assumed.

revoke

**SYNTAX AS A COMMAND**

wdc revoke

**FUNCTION**

removes the video system from the user's terminal. The standard tty_attachment is restored. The settings of the following modes are copied when the video system is revoked: vertsp, can, erkl, esc, red, and ctl_char. If ^more is set while in the video system, ^pl mode will be set after revoking the video system. (For more details on modes, see the window_io I/O module in Section 7.) Similarly, the settings of the current erase and kill characters are copied when the video system is revoked. (See "Real-Time Editing" in Section 4 for details.) See Section 2 for more information on the use of this command.

scroll_region, scrgn

**SYNTAX AS A COMMAND**

wdc scrgn -count N {-line L -height C -io_switch WINDOW}

**FUNCTION**

scrolls the specified region N lines as specified by -count. The specified region is the whole width of the screen. It can be a whole window or part of a window. If -count N is negative the window is scrolled down, and if it is positive the window is scrolled up. If lines are scrolled off the screen they are dropped. If -line is not supplied, the default is 1. If -height is not supplied, the remainder of the window is scrolled. If -io_switch is not specified, user_i/o is assumed.
window_call

set_position, spos

SYNTAX AS A COMMAND

wdc spos -line L -column C {-io_switch WINDOW}

FUNCTION

positions the cursor to the specified line and column of the specific window. If -io_switch is not specified, user_i/o is assumed.

set_position_rel, sposrel

SYNTAX AS A COMMAND

wdc sposrel -line L -column C {-io_switch WINDOW}

FUNCTION

changes the cursor position by N lines and N columns. If -io_switch is not specified, user_i/o is assumed. The current cursor position must be defined. One of the control_args must be specified and both may be specified. Whichever control_arg is not specified defaults to zero.

supported_terminal

SYNTAX AS A COMMAND

wdc supported_terminal {-ttp terminal_type}

FUNCTION

returns "true" if the video system can be invoked on the specified terminal type. If no terminal type is specified, the current terminal type is used.

sync

SYNTAX AS A COMMAND

wdc sync {-io_switch WINDOW}

FUNCTION

waits for the last operation performed on the window to be completed. Over certain networks it may not be possible to actually wait for delivery of the characters to the terminals. If -io_switch is not specified, user_i/o is assumed.
window_call

video_invoked

SYNTAX AS A COMMAND

wdc video_invoked

FUNCTION

returns "true" if the video system is in use in the user's process.

write_sync_read, wsr

SYNTAX AS A COMMAND

| wdc wsr -string STR -count N {-io_switch WINDOW}

FUNCTION

displays a prompting string STR at the current cursor position in the window, and then reads input typed in response to the prompt. Characters are read unechoed, until either N characters or a break character is read. If -io_switch is not specified, user_i/o is assumed.

ACTIVE FUNCTION USAGE

prints a prompting string and returns the characters read.
This section contains descriptions of the PL/I subroutines used by the menu and video software, presented in alphabetical order.
Name: menu_

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 other than comparison with the null pointer or 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.pll described below in "Data Structures".

Entry: menu_Screate

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

USAGE

declare menu_Screate entry ((* char (* varying, (* char (* varying, (* char (* varying, ptr, (* char (* varying, ptr, ptr, ptr,
                fixed bin (35)));

call menu_Screate (choices, headers, trailers, format_ptr, keys, area_ptr,
                needs_ptr, menu, code);

ARGUMENTS

choices
    is an array of the names of the options. (Input) If the maximum number of choices is exceeded, the code menu_et_$too_many_options is returned. The current maximum is 61.

headers
    is an array of headers. (Input) 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 is invalid PL/I.

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

format_ptr
    points to a structure, menu_format, that controls formatting of the menu. (Input) This structure is described below in "Data Structures".
keys
    is an array specifying the keystroke for each option. (Input) 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. If the valid keys (the keys for which there are choices) are either all upper case or all lower case, menu_$get_choice will treat upper and lower case letters identically.

area_ptr
    is a pointer to an area where the menu description is allocated. (Input) If the area is not large enough, the area condition is signalled. If this pointer is null, the system free area is used.

needs_ptr
    points to the menu_requirements structure giving requirements to display the menu. (Input) The structure is described below in "Data Structures". The caller supplies this structure and fills in the version number menu_requirements_version_1, the remaining members are output from this entry.

menu
    is a newly created menu object. (Output)

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

Entry: menu_$delete

This entry deletes a menu object from a specified value segment.

USAGE

declare menu_$delete entry (char (*), char (*), char (*), fixed bin (35));
call menu_$delete (dirname, entryname, menu_name, code);

ARGUMENTS

dirname
    is the pathname of the containing directory. (Input)

entryname
    is the entryname of the segment. (Input) It must have the value suffix.

menu_name
    is the name that was assigned to the menu when it was stored (see the description of menu_$store). (Input)

code
    is a standard system error code. (Output)
Entry: menu_$describe

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**

```plaintext
declare menu_$describe entry (ptr, ptr, fixed bin (35));
call menu_$describe (menu, needs_ptr, code);
```

**ARGUMENTS**

- **menu** is the menu object to describe. (Input)
- **needs_ptr** points to a structure declared like menu_requirements described in "Data Structures" below. (Input) The caller fills in the version to be menu_requirements_version_1, and the remaining members are filled in by this entry.
- **code** is a standard system error code. (Output)

Entry: menu_$destroy

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 or on stored menus.

**USAGE**

```plaintext
declare menu_$destroy entry (ptr, fixed bin (35));
call menu_$destroy (menu, code);
```

**ARGUMENTS**

- **menu** is the menu object to destroy. (Input)
- **code** is a standard system error code. (Output)
Entry: menu_$display

This entry displays a menu object on a supplied window.

**USAGE**

```plaintext
declare menu_$display entry (ptr, ptr, fixed bin (35));
call menu_$display (window, menu, code);
```

**ARGUMENTS**

- `window` is a pointer to an IOCB for an I/O switch attached through window_io_. (Input) This window must be large enough to hold the menu. A menu window should be used ONLY for menu I/O, if redisplay optimizations are desired.

- `menu` is the menu object to be displayed. (Input)

- `code` is a standard system error code. (Output)

Entry: menu_$get_choice

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

**USAGE**

```plaintext
declare menu_$get_choice entry (ptr, ptr, ptr, bit (1) aligned, fixed bin, fixed bin (35));
call menu_$get_choice (window, menu, function_key_info, fkey, selection, code);
```

**ARGUMENTS**

- `window` is a pointer to the IOCB for the I/O switch used to display the menu. (Input)

- `menu` is the menu object on display in the window. (Input)

- `function_key_info` is a pointer to a data structure describing the function keys available on the terminal. (Input) 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
returns a value of "1"b if a function key was hit instead of a menu selection. (Output)

selection
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
is a standard system error code. (Output)

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_$list

This entry lists the menu objects stored in a specified value segment.

USAGE

declare menu_$list entry (char (*), char (*), char (*), ptr, fixed bin, ptr,
                   fixed bin (35));
call menu_$list (dirname, entryname, menu_starname, area_ptr,
                   menu_list_info_version, menu_list_info_ptr, code);

ARGUMENTS

dirname
is the pathname of the containing directory. (Input)

entryname
is the entryname of the segment. (Input) It must have the value suffix.

menu_starname
is matched against the names of the menus stored in the segment. (Input) Only names
that match menu_starname are returned. (see the description of menu_$store).

area_ptr
is a pointer to an area in which to allocate the structure containing the menu names.
(Input) If it is null, the system free area is used.
menu_list_info_version
is the version of the menu_list_info structure that the caller expects. (Input) It must be
a supported menu_list_info structure version. The only supported version is
menu_list_info_version_1.

menu_list_info_ptr
is a pointer to the menu_list_info structure, described below under "Data Structures". (Output)

code
is a standard system error code. (Output)

Entry: menu_$retrieve

This entry retrieves a menu from a specified segment. The segment must be a value segment.
The menu data structure is allocated in a caller-supplied area. The menu information is
copied from the menu object stored in the segment into the newly allocated structure.

USAGE

declare menu_$retrieve entry (char (*), char (*), char (*), ptr, ptr,
fixed bin (35));
call menu_$retrieve (dirname, entryname, menu_name, area_ptr, menu_ptr, code);

ARGUMENTS

dirname
is the pathname of the containing directory. (Input)

entryname
is the entryname of the segment. (Input) It must have the value suffix.

menu_name
is the name that was assigned to the menu when it was stored (see the description of
menu_$store). (Input)

area_ptr
is a pointer to an area where the menu object is allocated. (Input) If this argument is
null, the system free area is used. If the area is not large enough, the area condition is
signalled.

menu_ptr
is a pointer to the menu object that is retrieved from the segment. (Output)

code
is a standard system error code. (Output)
Entry: menu_$store

This entry stores a menu object in a specified segment. The specified segment must be a value segment.

USAGE

```
declare menu_$store entry (char (*), char (*), char (*), bit(1) aligned, ptr,
    fixed bin (35));
call menu_$store (dirname, entryname, menu_name, create_sw, menu_ptr, code);
```

ARGUMENTS

- **dirname**
  - is the pathname of the containing directory. (Input)

- **entryname**
  - is the entryname of the segment. (Input) It must have the value suffix.

- **menu_name**
  - is a name to be assigned to the menu. (Input)

- **create_sw**
  - determines whether or not the segment is created if it does not already exist. If the segment does not exist, a value of "1"b will cause it to be created. (Input)

- **menu_ptr**
  - is a pointer to the menu object that is to be stored in the segment. (Input)

- **code**
  - is a standard system error code. (Output)

DATA STRUCTURES

A menu is described by the "menu_format" structure. It is declared in menu_dcls.incl.pl1.

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

menu_format
specifies the format for menu display. (Input) 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. (Input)

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

max_height
is the maximum height of the window, in lines. (Input)

n_columns
is the number of columns to use in displaying options. (Input)

center_headers
if set, header lines will be centered using the window width supplied above. (Input) If not set, they are flush with the left edge of the window.

center_trailers
same as center_headers, but for trailers. (Input)

pad
must be "0"b. (Input)

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

**THE MENU_LIST_INFO STRUCTURE**

This entry returns information in the menu_list_info structure, found in the include file menu_list_info.incl.pll, shown below:

```markdown
DCL 1 menu_list_info
  2 version fixed bin,
  2 n_names fixed bin,
  2 name_string_length fixed bin (21),
  2 names (menu_list_n_names refer
    (menu_list_info.n_names) aligned,
    3 position fixed bin (21),
    3 length fixed bin (21),
    2 name_string character (menu_list_name_string_length
      refer (menu_list_info.name_string_length)) unaligned;
```
STRUCTURE ELEMENTS

version
   is the version of this structure, menu_list_info_version_1. (Output)

n_names
   is the number of menu object names that matched the supplied starname. (Output)

name_string_length
   is the total length of all the names that matched the supplied starname, concatenated
   together. (Output)
	names
   is an array of information with one entry for each name that matched the specified
   starname. (Output)

position
   is the position in the string menu_list_info.name_string of this menu name. (Output)

length
   is the length of this menu name in the string menu_list_info.name_string. (Output)

name_string
   contains all the returned names, concatenated together. (Output) The PL/I "defined"
   attribute can be used to advantage to refer to individual names. For example, we wish to
   print the menu name indexed by name_index.

begin;
   declare this_name character (menu_list_info.length (name_index))
        defined (menu_list_info.name_string)
        position (menu_list_info.position (name_index));

   call ioa_ ("The ^d'th menu name is: ^a", name_index, this_name);
end;
THE MENU_REQUIREMENTS STRUCTURE

The requirements for a menu are specified by the menu_requirements structure. It is declared in menu_dcls.incl.pll.

```
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;
```

**STRUCTURE ELEMENTS**

**version**

is set by the caller, and must be menu_requirements_version_1. (Input)

**lines_needed**

is the number of lines required. (Output) If the window does not have this number of lines, menu display will fail.

**width_needed**

is the number of columns needed. (Output)

**n_options**

is the number of options defined. (Output)

The include file, menu_dcls.incl.pll, 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 entry. Its name is MENU_OPTION_KEYS.
The video_data_ subroutine is a data segment containing information about the video system.

Entry: video_data_ $terminal_iocb

This is the terminal control switch IOC pointer. If the video system is activated for the user's terminal, this pointer is nonnull, and points to the IOC for the switch user_terminal_.

**USAGE**

```c
fnt typ declare video_data_ $terminal_iocb pointer external static;
```

**NOTES**

User programs may use this pointer for two purposes:

1. Inquiring as to whether the video system is activated, by checking to see if the pointer is null.

2. Determining the physical characteristics and capabilities of the terminal. This may be accomplished with the get_capabilities control order, described under the window_io_ I/O module. The height and width returned will be that of the physical terminal screen.

   No other manipulations of this switch are permitted.
This subroutine provides interfaces for activating and de-activating the video system.

Entry: video_utils__$turn_on_login_channel

This entry removes the existing attachment of the user's terminal, replacing it with the video system. When this entry returns successfully, the switch user_terminal_ is attached through tc_io_ to the user's terminal. The switch user_i/o is attached through window_io_ to a window covering the entire screen. invoked: vertsp, can, erkl, esc, red, and ctl_char. In addition, if ^p is set on video system invocation, ^more will be set in the video system. (For more details on modes, see the window_io_ I/O module.) Similarly, the settings of the current erase and kill characters are copied when the video system is invoked. (See "Real-Time Editing" for details.) To see how the standard I/O switch attachments change when you activate the video system on your terminal, refer to Figure A-2 in Appendix A.

USAGE

declare video_utils__$turn_on_login_channel entry (fixed bin (35), char (#));
call video_utils__$turn_on_login_channel (code, reason);

ARGUMENTS

code
isis a standard system error code. (Output)

reason
contains information about the error, if there is one. (Output) (128 characters are enough to hold any message that may be returned in reason.)

NOTES

If the video system is already in service on the user's terminal, the status code video_et__$swsys_invoked is returned, and the value of reason is not defined.

If the activation of the video system fails, the original attachment of the terminal (through tty_) is restored, and information is returned in reason and code.

In particular, if the switch user_i/o is not currently attached through tty_, the code video_et__$switch_not_attached_with_tty_ is returned. This may indicate that the user has auditing or the graphic system in place. The message returned in reason advises the user to remove graphics or auditing and try again.
Entry: video_utils$_$turn_off_login_channel

This entry reverses the actions of video_utils$_$turn_on_login_channel. That is, it removes the window attachment of user_i/o, detaches terminal control from the user's terminal, and attaches user_i/o to the user's terminal via tty_. The settings of the following modes are copied when the video system is revoked: vertsp, can, erkl, esc, red, and ctl_char. If ^more is set while in the video system, ^pl mode will be set after revoking the video system. (For more details on modes, see the window_io_ I/O module.) Similarly, the settings of the current erase and kill characters are copied when the video system is revoked. (See "Real-Time Editing" for details.) It is the user's responsibility to detach any windows other than user_io before calling this entry point.

USAGE

declare video_utilities$_$turn_off_login_channel entry (fixed bin (35));
call video_utilities$_$turn_off_login_channel (code);

ARGUMENTS

code

is a standard system error code. (Output) It is nonzero if and only if the video system can not be removed from the user's terminal.
Name: window_

The window_ subroutine provides a terminal independent interface to video terminal operations. More specifically, it controls and performs I/O to a window.

The window_ subroutine is used in conjunction with the iox_ subroutine call entry points in the window_io_ I/O module. The window_ and video_utils_ subroutines together perform the same functions as the window_call command.

The virtual terminal implemented by window_ corresponds closely to common video terminals. The features of the terminal are defined implicitly by the entries below. Not all entries can be supported on all terminals. The result of calling an unsupported feature is the error code video_et_$capability_lacking. Programs can determine whether the device in question supports a given operation by using a get_capabilities control order, described under the window_io_ I/O module.

Additional terminals may be supported by defining their video attributes in the Terminal Type File (TTF). The TTF is described in the Multics Programmer's Reference Manual, Order No. AG91.

Some entry points require that the current cursor position be defined when they are called. The current position is defined unless a call is made to the write_raw_text entry point, or an asynchronous event changes the window contents. If the current position is not defined, these entry points will return the status code video_et_$cursor_position_undefined.

If an asynchronous event changes the state of the window, status will be set for the window. Once window status is set, all calls to window_ on that window will return the status code video_et_$window_status_pending until a get_window_status control order is used to pick up the status.

The calling sequences for all the entry points are in the include file window_dcls.incl.pl1.

Entry: window_$bell

This entry activates the terminal alarm. For most terminals, this will be the audible bell. For some it will be a visible signal.

USAGE
declare window_$bell entry (ptr, fixed bin (35));
call window_$bell (iocb_ptr, code);

ARGUMENTS

iocb_ptr
    is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
    is a standard system error code. (Output)
NOTES

The current cursor position must be defined for this call. If the cursor is in some other window on the screen when this call is made, it is moved to the current position in this window.

Entry: window_\_change_column

This entry moves the cursor to a different column on the current line, without changing the line.

**USAGE**

declare window_\_change_column entry (ptr, fixed bin, fixed bin (35));
call window_\_change_column (iocb_ptr, new_column, code);

**ARGUMENTS**

iocb_ptr
   is a pointer to an IOCB for a switch attached with window_io_. (Input)

new_column
   is the new column. (Input)

code
   is a standard system error code. (Output)

NOTES
The current cursor position must be defined.

Entry: window_\_change_line

This entry moves the cursor to a new line without changing the column.

**USAGE**

declare window_\_change_line entry (ptr, fixed bin, fixed bin (35));
call window_\_change_line (iocb_ptr, new_line, code);

**ARGUMENTS**

iocb_ptr
   is a pointer to an IOCB for a switch attached with window_io_. (Input)

new_line
   is the new line. (Input)
code is a standard system error code. (Output)

Entry: window_$clear_region

This entry replaces the contents of the region specified with spaces, and leaves the cursor at the upper left-hand corner of the region. The region is defined by giving the upper left-hand corner (line and column), and the width and height of the region.

USAGE

declare window_$clear_region entry (ptr, fixed bin, fixed bin, fixed bin,
fixed bin, fixed bin (35));
call window_$clear_region (iocb_ptr, start_line, start_col, n_lines, n_cols,
code);

ARGUMENTS

iocb_ptr is a pointer to an IOCB for a switch attached with window_io_. (Input)

start_line is the number of the line where clearing will begin. (Input)

start_col is the number of the column where clearing will begin. (Input)

n_lines is the number of lines which will be cleared. (Input)

n_cols is the number of columns which will be cleared. (Input)

code is a standard system error code. (Output)

NOTES

The rectangular region described in cleared. The cursor position defined at (start_line, start_col).
Entry: window$clear_to_end_of_line

This entry clears everything to the right of the cursor on the current line to spaces. Positions to the left of the cursor are not affected. The cursor is not moved.

USAGE

declare window$clear_to_end_of_line entry (ptr, fixed bin (35));
call window$clear_to_end_of_line (iocb_ptr, code);

ARGUMENTS

iocb_ptr
  is a pointer to an IOCB for a switch attached with window io_. (Input)

code
  is a standard system error code. (Output)

NOTES

The cursor position must be defined.

Entry: window$clear_to_end_of_window

This entry clears all of the window between the cursor and the end of the window. This includes everything to the right of the cursor on the current line, and all lines below the cursor. The cursor is not moved.

USAGE

declare window$clear_to_end_of_window entry (ptr, fixed bin (35));
call window$clear_to_end_of_window (iocb_ptr, code);

ARGUMENTS

iocb_ptr
  is a pointer to an IOCB for a switch attached with window io_. (Input)

code
  is a standard system error code. (Output)

NOTES

The current cursor position must be defined.
Entry: window_$clear_window
This entry clears the entire window to spaces, and leaves the cursor at home.

USAGE
declare window_$clear_window entry (ptr, fixed bin (35));
call window_$clear_window (iocb_ptr, code);

ARGUMENTS

iocb_ptr
    is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
    is a standard system error code. (Output)

NOTES
The cursor position is defined to be at line 1, column 1 after the screen is cleared.

Entry: window_$create
This entry creates a new window on the terminal screen.

USAGE
declare window_$create entry (ptr, ptr, ptr, fixed bin (35));
call window_$create (terminal_iocb_ptr, window_info_ptr, window_iocb_ptr, code);

ARGUMENTS

terminal_iocb_ptr
    is a pointer to an IOCB for the terminal control switch. (Input) Normally this should be video_data_$terminal_iocb.

window_info_ptr
    is a pointer to a standard window_position_info structure, as declared in windowcontrol_info.incl.pl. (Input)

window_iocb_ptr
    is a pointer to a detached IOCB pointer. (Input) It may be obtained with iox_$find_iocb which must be done before the call to window_$create. For example:

        call iox_$find_iocb ("top_window", window_iocb_ptr, code);

        where the value returned for window_iocb_ptr is used in the call to window_$create.
window_

code
is a standard system error code. (Output)

NOTES
The window_info_ptr must point to a window_position_info structure, as declared in window_control_info.incl.pll. If window_position_info.width is set to zero, the window will occupy the full width of the screen. Currently windows must occupy the full width of the screen. If tc_io_.*.in window_position_info.height is set to zero, the remainder of the screen is used. The iocb_ptr is an input argument, iox_$find_iocb may be used to obtain an iocb_ptr for a new switch.

Entry: window_$delete_chars

This entry deletes characters on the current line. Characters to the right of the cursor are moved to the left. Character positions opened up on the right margin are filled with spaces. It is an error to call this entry point if the terminal does not support the delete chars operation.

USAGE
declare window_$delete_chars entry (ptr, fixed bin, fixed bin (35));
call window_$delete_chars (iocb_ptr, n_chars, code);

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

n_chars
is the number of characters (starting at the current cursor position) that will be removed from the screen. (Input) If n_chars is zero, no action is taken.

code
is a standard system error code. (Output)

NOTES
The current cursor position must be defined. The number of characters specified by n_chars are deleted, and the remaining characters on the line, if any, move leftward to occupy the space.
Entry: window_$destroy

This entry destroys an existing window, leaving its IOCB in a detached state.

 USAGE

declare window_$destroy entry (ptr, fixed bin (35));
call window_$destroy (window_iocb_ptr, code);

 ARGUMENTS

window_iocb_ptr
  is a pointer to an IOCB attached with window_$create. (Input)

code
  is a standard system error code. (Output)

Entry: window_$edit_line

This entry allows applications to preload the video editor input buffer with a string.

 USAGE

declare window_$edit_line entry (pointer, pointer, pointer, fixed bin (21),
  fixed bin (21), fixed bin (35));
call window_$edit_line (iocb_ptr, window_edit_line_info_ptr, buffer_ptr,
  buffer_len, n_returned, code);

 ARGUMENTS

window_iocb_ptr
  is a pointer to an IOCB for a switch attached with window_io. (Input)

window_edit_line_info_ptr
  is a pointer to a window_edit_line_info structure, as declared in window_control_info.incl.pl1
  (described below). (Input)

version
  is the version number of the structure. (Input) This is currently window_edit_line_version_1.

line_ptr
  is a pointer to the initial text string to be loaded into the input buffer before editing begins. (Input)

line_length
  is the length of the string pointed to by line_ptr. (Input)
buffer_ptr
  is a pointer to a buffer where the users input will be put. (Input)

buffer_len
  is the size of the input buffer. (Input)

n_returned
  is the number of characters in the final output line. (Output)

code
  is a standard system error code. (Output)

Entry: window__$get_cursor_position

This entry is used to return the current position of the cursor. If the last operation done to
the terminal was in some other window, this will not be the actual position of the cursor on
the screen.

USAGE

declare window__$get_cursor_position entry (ptr, fixed bin, fixed bin, fixed
  bin (35));
call window__$get_cursor_position (iocb_ptr, line, col, code);

ARGUMENTS

ioucb_ptr
  is a pointer to an IOCB for a switch attached with window_io_. (Input)

line
  is the line number. (Output)

col
  is the column position. (Output)

code
  is a standard system error code. (Output)

NOTES

The current cursor position must be defined.
Entry: window$_$get_echoed_chars

This entry accepts input from the typist, echoing the characters as typed, until either a specified number of characters are read, or a break character is encountered. By default, the break characters are the control characters plus DEL (177 octal).

Usage

declare window$_$get_echoed_chars entry (ptr, fixed bin (21), char (n), fixed bin (21), char (1) varying, fixed bin (35));

call window$_$get_echoed_chars (iocb_ptr, n_to_get, buffer, n_got, break, code);

Arguments

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

n_to_get
is the number of columns (N) between the cursor and the end of the line. (Input) At most N characters will be returned.

buffer
is the caller-supplied buffer that holds characters returned. (Input)

n_got
is the number of characters returned. (Output) Each character is echoed.

break
is the character that causes the echoing to stop. (Output) This character is not echoed.

code
is a standard system error code. (Output)

Notes

This entry point returns no more than n_to_get characters in buffer. It reads and echoes characters until either (1) it has read n_to_get characters, or (2) it has read a break character. If it stops due to a break character, the break character is returned in break, otherwise break is equal to '"'.

6-23 CP51-02
Entry: window$_$get_one_unechoed_char

This entry reads a single character, unechoed, from the terminal. Optionally, it can return instead of waiting if there are no characters available.

**USAGE**

```pl
declare window$_$get_one_unechoed_char entry (ptr, char (1) varying, bit (1) aligned, fixed bin (35));

call window$_$get_one_unechoed_char (iocb_ptr, char_read, block_flag, code);
```

**ARGUMENTS**

* `iocb_ptr` is a pointer to an IOCB for a switch attached with window_io_. (Input)

* `char_read` is the read character. (Output) If block_flag is "0"b, and no input is typed ahead, then this will be a zero length character string.

* `block_flag` if this flag is "1"b, input from the terminal is awaited if none is available. (Input) If it is "0"b, and no input is available, then this entry returns immediately, and sets char_read to "".

* `code` is a standard system error code. (Output)

**NOTES**

Beware of the PL/I language definition of character string comparisons when using this entry with a block flag of "0"b. In PL/I, both of the following comparisons are true:

```pl
("" = " ")
("" = " ")
```

That is, a zero length varying string compares equally to a single space. To test if char_read is nonempty, use an expression like:

```pl
(length (char_read) > 0)
```
Entry: window__$get__unechoed__chars

This entry accepts input from the typist, leaving it unechoed, until either a specified number of characters are read, or a break character is encountered.

**USAGE**

```plaintext
declare window__$get__unechoed__chars entry (ptr, fixed bin (21), char (*), fixed bin (21), char (1) varying, fixed bin (35));

call window__$get__unechoed__chars (iocb_ptr, n_to_get, buffer, n_got, break, code);
```

**ARGUMENTS**

- `iocb_ptr`  
  is a pointer to an IOCB for a switch attached with window_io_. (Input)

- `n_to_get`  
  is the number of columns (N) between the cursor and the end of the line. (Input) At most N characters will be returned.

- `buffer`  
  is the caller-supplied buffer that holds characters returned. (Input)

- `n_got`  
  is the number of characters returned. (Output) Each character is echoed.

- `break`  
  is the character that causes the echoing to stop. (Output) This character is not echoed.

- `code`  
  is a standard system error code. (Output)

**NOTES**

This entry point will read no more than `n_to_get` characters from the terminal, without echoing them to the typist. The characters are returned in the buffer. Characters are read until either (1) `n_to_get` characters are read, or (2) a break character is read. If reading stops due to a break character, then the break character is returned in `break`. Otherwise `break` is "".

6-25 CP51-02
Entry: window_$insert_text

This entry inserts text at the current cursor position. Text at the cursor or to the right of
the cursor is shifted to the right, to accommodate the new text. It is an error to call this
entry if the terminal does not support the insertion of text.

 USAGE

declare window_$insert_text entry (ptr, char (*), fixed bin (35));
call window_$insert_text (iocb_ptr, text, code);

 ARGUMENTS

 iocb_ptr
   is a pointer to an IOCB for a switch attached with window_io_. (Input)

text
   is the character string to be written. (Input) When converted to output, each character in
   this string must occupy exactly one print position. The length of this string must be such
   that characters moved to the right will remain on the current line in the window. If
   these conditions are not met, the result is undefined. The cursor is left after the last
   character inserted.

code
   is a standard system error code. (Output)

 NOTES

The current cursor position must be defined. The string "text" must contain only printable
ASCII graphics. If it contains any other characters, the status code video_et_$string_not_printable
is returned.

Entry: window_$overwrite_text

This entry writes text on the window in the current cursor location. If there is any text at
or to the right of the current cursor position in the window, it is overwritten with the
supplied string.

 USAGE

declare window_$overwrite_text entry (ptr, char (*), fixed bin (35));
call window_$overwrite_text (iocb_ptr, text, code);

 ARGUMENTS

 iocb_ptr
   is a pointer to an IOCB for a switch attached with window_io_. (Input)
text
is the character string to be written. (Input) This string should consist of only printable ASCII graphics (octal codes 040 through 176 inclusive), and may not be longer than the space remaining on the current line.

code
is a standard system error code. (Output)

NOTES
The cursor position must be defined. The string "text" may contain only printable ASCII graphics. If it contains anything else the status code video_et_$string_not_printable is returned.

Entry: window_$position_cursor
This entry moves the cursor to any requested position in the window. It defines the current cursor position if it is undefined.

USAGE
declare window_$position_cursor entry (ptr, fixed bin, fixed bin, fixed bin (35));
call window_$position_cursor (iocb_ptr, line, col, code);

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input) line is the line number. (Input)

col
is the column position. (Input)

code
is a standard system error code. (Output)

Entry: window_$position_cursor_rel
The entry moves the cursor relative to the current location.

USAGE
declare window_$position_cursor_rel entry (ptr, fixed bin, fixed bin, fixed bin (35));
call window_$position_cursor_rel (iocb_ptr, line_inc, col_inc, code);
ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

line_inc
is the change in line number. (Input) If line_inc is a positive number, the cursor is moved down. If it is a negative number, the cursor is moved up. If it is zero, the cursor's line number is not changed.

col_inc
is the change in column position. (Input) If col_inc is a positive number, the cursor is moved to the right. If it is a negative number, the cursor is moved to the left. If it is zero, the cursor's column position is not changed.

code
is a standard system error code. (Output)

Entry: window__scroll_region

This entry scrolls a region up or down a given number of lines. A positive scroll count scrolls the window up, deleting lines from the top of the window and adding new blank lines to the bottom. The cursor's new position is at the beginning of the first new blank line. A negative count scrolls the window down, deleting lines from the bottom and adding lines to the top. The cursor is left at home. If this entry is called and the terminal does not support either scrolling or insert and delete lines, the result is an error status, video_et__capabilities_lacking.

USAGE

declare window__scroll_region entry (ptr, fixed bin, fixed bin, fixed bin, fixed bin (35));
call window__scroll_region (iocb_ptr, start_line, n_lines, scroll_distance, code);

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

start_line
is the number of the first line of the region. (Input)

n_lines
is the number of lines that compose the region. (Input)

scroll_distance
is the distance in lines by which the region will be scrolled. (Input)
code

is a standard system error code. (Output)

NOTES

The cursor position is defined to be column one on first_line. The region from first_line for n_lines is scrolled scroll_distance lines, which may be negative.

Entry: window_$sync

This entry synchronizes the process with the typist by writing any pending output to the terminal.

USAGE

declare window_$sync entry (ptr, fixed bin (35));
call window_$sync (iocb_ptr, code);

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
is a standard system error code. (Output)

NOTES

The calling process is made to wait until the typist types something after the last text output has been transmitted to the terminal.

Entry: window_$write_raw_text

This entry is used to output a terminal dependent sequence. The current cursor position becomes undefined after this call is made. This entry should not be used to output sequences that put graphics onto the terminal screen, as the video system's internal screen image will become inconsistent. This entry is used for terminal-specific features that cannot be accessed via the video system.

USAGE

declare window_$write_raw_text entry (ptr, char (x), fixed bin (35));
call window_$write_raw_text (iocb_ptr, text, code);

6-29 CP51-02
ARGUMENTS

iocb_ptr
  is a pointer to an IOCB for a switch attached with window_io_. (Input)

text
  is any string of printable ASCII characters to be transmitted to the terminal. (Input)

code
  is a standard system error code. (Output)

NOTES

Any call to window_$write_raw_text causes the cursor position to become undefined and sets the screen_invalid window status flag. Subsequent calls to write_raw_text will ignore this flag, but all other window_ entrypoints will return the status code video_et_$window_status_pending until the status flag is cleared. It is the responsibility of the application performing the raw output call to perform a get_window_status control order to clear the status flag.

Entry: window__$write__sync__read

This entry writes a prompt, synchronizes input to the output of the prompt, and reads a response. This entry is useful for queries where it is important to avoid interpreting type-ahead as a response to a question.

USAGE

declare window__$write_sync_read entry (ptr, char (*), fixed bin (21),
  char (*), fixed bin (21), char (1) varying, fixed bin (35));

call window__$write_sync_read (iocb_ptr, prompt, n_to_get, buffer, n_got, break, code);

ARGUMENTS

iocb_ptr
  is a pointer to an IOCB for a switch attached with window_io_. (Input)

prompt
  is a string of printable ASCII characters which must fit on the current line. (Input)

n_to_get
  is the number of columns (N) between the cursor and the end of the line. (Input) At most N characters will be returned.
buffer
    is the caller-supplied buffer that holds characters returned. (Input)

default
    is the number of characters returned. (Output) Each character is echoed.

break
    is the character that causes the echoing to stop. (Output) This character is not echoed.

code
    is a standard system error code. (Output)

NOTES

The current cursor position must be defined. This entry overwrites the text string "prompt" at the current cursor position. It then reads characters typed after the prompt has been transmitted to the terminal. The characters are read in the same fashion as the get_unechoed_chars entry point. Any characters read before the prompt is transmitted, are buffered and returned to get_echoed_chars or subsequent get_unechoed_chars calls.
This section contains descriptions of the I/O modules used by the menu and video software, presented in alphabetical order. For details on I/O processing, see the Multics Programmer's Reference Manual, Order No. AG91.
Name: tc_io_

The tc_io_ I/O module supports terminal independent I/O to the screen of a video terminal.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system interfaces iox_.

**ATTACH DESCRIPTION**

tc_io_ {device} {-control_args}

**ARGUMENTS**

device
* is the channel name of the device to be attached. If a device is not given, the 
  -login_channel control argument must be given.

**CONTROL ARGUMENTS**

-login_channel
  specifies attachment to the user's primary login channel. If a device is not specified, 
  then the user's login channel is used. This control argument flags this switch for 
  reconnection by the process disconnection facility. If the user's login device should hang 
  up, this switch will be automatically closed, detached, attached, and opened on the user's 
  new login channel when the user reconnects, if permission to use this facility is specified 
  in the SAT and PDT for the user.

-destination DESTINATION
  specifies that the attached device is to be called using the address DESTINATION. In 
  the case of telephone auto_call lines, DESTINATION is the telephone number to be 
  dialed. See the dial_manager_ subroutine in the *Multics Subroutines and I/O Modules* 
  manual, Order No. AG93, for more details.

-no_block
  specifies that the device is to be managed asynchronously. The tty_ subroutine will not 
  block to wait for input to be available or output space to be available. This control 
  argument should not be used on the login channel, because it will cause the command 
  listener to loop calling get_chars.

-no_hangup_onDetach
  prevents the detach entry point from hanging up the device. This is not meaningful for 
  the login channel.

-hangup_onDetach
  causes the detach entry point to hang up the device automatically. This is not meaningful 
  for the login channel.
OPEN OPERATION

Opens the module for stream_input_output.

GET LINE OPERATION

The get_line operation is not supported.

CONTROL OPERATION

The following control orders are supported:

clear_screen
    clears the entire terminal screen. The info_ptr is null. It is intended for use when the
    screen image may have been damaged due to communications problems, for example.

get_capabilities
    returns information about the capabilities of the terminal. The info structure is described
    in the description of the "get_capabilities" control order in the window_io_module.

get_break_table
    returns the current break table. The info pointer should point to a break table, declared
    as follows (window_control_info.incl.pll):

    dcl 1 break_table_info aligned based (break_table_ptr),
        2 version fixed bin,
        2 breaks (0:127) bit (1) unaligned;

STRUCTURE ELEMENTS

version
    must be set by the caller to break_table_infc_version_I. (Input)

breaks
    has a "1"b for each character that is a break character. (Output)

set_break_table
    sets the break table. The info pointer should point to a break table as defined by the
    get_break_table order, above. By default, the break table has "1"b for all nonprintable
    characters, and "0"b elsewhere. Applications that set the break table must be careful to
    reset it afterwards, and establish an appropriate cleanup handler.

set_line_speed
    sets the speed of the terminal's connection to Multics. The info_ptr should point to a
    fixed binary number representing the line speed in characters per second. Negative line
    speeds are not allowed.
set_term_type
changes the terminal type. The info pointer should point to a set_term_type_info structure, described below. This sets window_status_pending for all windows and sets the ttp_change field in the window_status structure along with the screen_invalid. This operation re-initializes all the terminal specific video system information such as the video sequences, length and width of the screen, and capabilities. It is equivalent to doing "window_call revoke; stty -ttp new_terminal_type; window_call invoke", except no windows are destroyed. The set_term_type_info structure is declared in set_term_type_info.incl.pl1:

```
dcl 1 set_term_type_info aligned based (sttip)
  2 version fixed bin
  2 name char (32) unaligned
  2 flags unaligned
    3 send_initial_string bit (1)
    3 set_modes bit (1)
    3 ignore_line_type bit (1)
    3 mbz bit (33);
```

**STRUCTURE ELEMENTS**

version
is the version of this structure. (Input) It must be stti_version_1.

name
is the name of the new terminal type. (Input)

**NOTES**

The send_initial_string, set_modes and ignore_line_type flags are all ignored by the video system. The initial string will always be sent.

reconnection
determines the new terminal type (which may or may not be the same as before the disconnection). Performs a set_term_type control order to inform the rest of the system of the change in terminal type. If the set_term_type fails then the video_utils_$turn_off_login_channel is invoked in an attempt to re-attach tty_. Reconnection (a field in window_status) is set to indicate to an application doing get_window_status that a reconnection has occurred.

The window_status_info structure is declared in window_status.incl.pl1.
The window_io_ I/O module supports I/O to a window. In addition to the usual iox_ entries, the module provides terminal independent access to special video terminal features, such as a moveable cursor, selective erasure, and scrolling of regions. The module provides a real-time input line editor and performs output conversion and "MORE" processing.

Entry points in this module are not called directly by users; rather, this module is accessed through the I/O system interfaces iox_ and window_.

**ATTACH DESCRIPTION**

```
window_io_switch {-control_args}
```

**ARGUMENTS**

`switch` is the name of an I/O switch attached to a terminal via the tc_io_ I/O module. The window created by this attach operation will be mapped onto the screen of that terminal. Use window_$create to attach and open, and use window_$destroy to detach and close windows on the login terminal.

**CONTROL ARGUMENTS**

- `first_line LINE_NO`  
  LINE_NO is the line number on the screen where the window is to begin. If omitted, the window starts on the topmost line of the screen (line 1).

- `height N_LINES, -n_lines N_LINES`  
  N_LINES is the number of lines in the window. The default is to use all lines to the end of the screen.

- `first_column COL_NO`  
  COL_NO is the column number on the screen where the window is to begin. If omitted, the window starts on the leftmost column of the screen (column 1).

- `width N_COLS, -n_columns N_COLS`  
  N_COLS is the number of the columns in the window. The default is all columns to the end of the screen.

**NOTES**

The attach description control arguments must specify a region which lies within the terminal screen. If not, the attachment is not made, and the error code video_et_$out_of_terminal_bounds is returned.

When the window is attached, it is cleared and the cursor is left at home.
**OPEN OPERATION**

The following opening mode is supported: stream_input_output.

**PUR CHARs OPERATION**

This operation is used to output a character string to the window. If rawo mode (see below) is disabled, the characters are processed according to the output conversions defined for the terminal. If necessary, the string is continued on subsequent lines of the window. If output passes the last line of the window, the placement of additional lines is controlled by the setting of the more_mode mode (see below). If an output line must be erased from the window to make room for this new output, and there has been no intervening input in this window, and more_mode (see below) is enabled, the user is queried for the disposition of this new output. (See MORE processing in Section 4.)

In rawo mode, the characters are written directly to the terminal, without any of the above processing.

**GET CHARs OPERATION**

This operation returns exactly one character, unechoed, regardless of the size of the caller's buffer. The line editor is not invoked by this call.

**GET LINE OPERATION**

The get_line operation invokes the real-time input line editor, and returns a complete line typed by the user. A description of the typing conventions is given in Section 4. The put_chars and get_line operations retrieve and reset any statuses that they encounter, so that applications that make these calls need not be changed to check for video_et_$window_status_pending.

**CONTROL OPERATION**

The control operations below are supported. Note that many of the control operations can be issued at command level via io_call commands; these include any control orders that do not require an info structure, and those described below. The following relations must hold when changing windows (set_window_info). These relations are always true when obtaining information about a window (get_window_info):

- \(0 < \text{column} + \text{width} \leq \text{screen width}\)
- \(0 < \text{line} + \text{height} \leq \text{screen height}\)
get_window_info
returns information about the position and extent of the window. The info ptr points to
the following structure (declared in window_control_info.incl.pl1):

```plaintext
dcl 1 window_position_info based (window_position_info_ptr),
    2 version fixed bin,
    2 origin,
        3 column fixed bin,
        3 line fixed bin,
    2 extent,
        3 width fixed bin,
        3 height fixed bin;
```

**STRUCTURE ELEMENTS**

**version**
- is the version number of this structure. (Input) It must be
  window_position_info_version_2.

**column**
- is the column of the upper left-hand corner of the window. (Output) If the column
  of the upper left-hand corner is zero, then the first column will be used, to allow
  old programs written when this was a mbz field to run without modification.

**line**
- is the line of the upper left-hand corner of the window. (Output)

**width**
- is the width of the window (columns). (Output)

**height**
- is the height of the window (lines). (Output)

set_window_info
causes the window to be relocated or to change size (or both). The info ptr points to
the same structure used in the "get_window_info" control order. The values have the
same meaning, but are the new values for the window when setting (Input), and are
returned by get_window_info (Output).
get_window_status, set_window_status

window status is used to inform the application that some asynchronous event has disturbed the contents of the window. When window status is set for a window, all calls to window_ will return video_et_window_status_pending until the status is reset. To reset the status, make a get_window_status control order on the switch. The info pointer should point to the following structure (declared in window_control_info.inc.pll):

```

dcl 1 window_status_info aligned based (window_status_info_ptr),
    2 version fixed bin,
    2 status_string bit (36) aligned;
```

**STRUCTURE ELEMENTS**

**version**

is the version of this structure. (Input) It must be window_status_version_1.

**status_string**

is the window status information. (Input) To interpret the actual status_string, use the include file window_status.inc.pll:

```
dcl 1 window_status_info aligned based (window_status_info_ptr),
    2 screen_invalid bit (1) unaligned,
    2 async_change bit (1) unaligned,
    2 ttp_change bit (1) unaligned,
    2 reconnection bit (1) unaligned,
    2 pad bit (32) unaligned;
```

**STRUCTURE ELEMENTS**

**screen_invalid**

indicates that the contents of the window have become undefined. (Input for set, Output for get) This will happen, for example, in the event of a disconnection/reconnection of the terminal.

**async_change**

indicates that a timer or event call procedure has made a modification to the window. (Input for set, Output for get)

**ttp_change**

indicates that the terminal type has changed. (Input for set, Output for get) This re-initializes all the terminal specific video system information such as the video sequences, length and width of the screen, and capabilities.
reconnection
determines the new terminal type (which may or may not be the same as before the
disconnection). (Input for set, Output for get) Performs a set_term_type control
order to inform the rest of the system of the change in terminal type.

pad
reserved for future expansion and must be "0"h.

NOTES
The get_window_status and get_window_status control orders are available from command level
and as active functions with the following io_call commands:

io_call control window_switch get_window_status status_key_1
    {status_key_2} N
io_call control window_switch set_window_status status_key_1
    {status_key_N}

where status_key_N is either screen_invalid, asynchronous_change, ttp_change, or reconnection.

get_capabilities
returns information about the generic capabilities of the terminal. These are the "raw"
physical characteristics of the terminal. The video system may simulate those that are
lacking. For example, the system simulates insert and delete characters, but does not
simulate insert and delete lines. The info ptr should point to the following structure
(declared in terminal_capabilities.incl.pl1):

dcl 1 capabilities_info aligned based(capabilities_info_ptr),
    2 version fixed bin,
    2 screensize,
        3 columns fixed bin,
        3 rows fixed bin,
    2 flags,
        3 scroll_region bit (1) unal,
        3 insert_chars bit (1) unal,
        3 insert_mode bit (1) unal,
        3 delete_chars bit (1) unal,
        3 overprint bit (1) unal,
        3 pad bit (28) unal,
    2 line_speed fixed bin,

STRUCTURE ELEMENTS

version
is the version number of this structure. (Input) It must be capabilities_info_version_1,
also declared in the include file.
columns
is the number of columns on the terminal. (Output)

rows
is the number of rows (lines) on the terminal. (Output)

scroll_region
is true if the terminal is capable of scrolling, with insert and delete lines. (Output)

insert_chars
is true if the insert_chars function is supported. (Output)

insert_mode
is true if the terminal is capable of going into and out of insert mode. (Output)

delete_chars
is true if the delete chars function is supported. (Output)

overprint
is true if the terminal is capable of printing overstrike characters. (Output) It is currently always set to "0"b (false).

pad
reserved for future expansion and must be "0"b.

line_speed
is the speed of the communications channel to the terminal, in characters per second. (Output)

reset_more
causes MORE Processing to be reset. All lines on the window may be freely discarded without querying the user.

get_editing_chars
is identical to the operation supported by the tty_ I/O module.

set_editing_chars
is identical to the operation supported by the tty_ I/O module.
NOTES

The get_editing_chars and set_editing_chars control orders are available from command level and as active functions with the following io_call commands:

io_call window_switch get_editing_chars
io_call control window_switch set_editing_chars erase_kill_characters

get_more_responses
returns information about the acceptable responses to MORE processing. The info pointer should point to the following structure (declared in window_control_info.incl.pl1):

dcl more_responses_info
2 version fixed bin,
2 n_yeses fixed bin,
2 n_noes fixed bin,
2 yeses char (32) unaligned,
2 noes char (32) unaligned;

STRUCTURE ELEMENTS

version
is the version number of this structure and must be set to more_responses_info_version_1, also declared in the include file. (Input)

n_yeses
is the number of different affirmative responses, from zero to 32. (Output)

n_noes
is the number of different negative responses. (Output)

yeses
is the concatenation of all the affirmative responses. (Output) Each response is one character. Only the first "n_yeses" are valid.

noes
is the concatenation of all negative responses. (Output) Each response is one character. Only the first "n_noes" are valid.

set_more_responses
sets the responses. The data structure is the same as the one used for the "get_more_responses" order except that all fields are Input. At most, 32 yeses and 32 noes may be supplied. It is highly recommended that there be at least one yes, so that output may continue. The "yes" and "no" characters must be distinct. If they are not, the error code video_et$overlapping_more_responses is returned, and the responses are not changed.
NOTES

The `get_more_response` and `set_more_response` control orders are available from command level and as active functions with the following `io_call` command:

```
io_call control window_switch get_more_responses
io_call control window_switch set_more_responses yes_responses
     no_responses
```

where the `yes_responses` and `no_responses` will be used as arguments to the `get_more_responses` control order. If either of the response strings contains blanks or special characters, it must be quoted.

`get_more_prompt set_more_prompt`

sets the prompt displayed when a more break occurs. The current more responses can be displayed as part of the more prompt, by including the proper `ioa_` control codes as part of the prompt string. For example the default video system more prompt string is "More? (^a for more; ^a to discard output)." With the default more responses of carriage return for more and the delete for discard, the final string displayed is "More (RETURN for more; DEL to discard output)." The info pointer should point to the following structure (declared in `window_control_info.incl.pll`):

```
dcl more_prompt_info aligned based (more_prompt_info_ptr),
   2 version char (8),
   2 more_prompt char (80);
```

**STRUCTURE ELEMENTS**

- `version`
  is the version number of this structure (currently more_prompt_info_version_1). (Input)

- `more_prompt`
  is the `ioa_` control string to serve as the more prompt. (Input for set, Output for get)
The get_more_prompt and set_more_prompt control orders are available from command level and as active functions with the following io_call command:

io_call control window_switch get_more_prompt
io_call control window_switch set_more_prompt prompt_string

where window_switch is a valid window_io_switch and prompt_string is the ioa_control_string described above.

get_more_handler set_more_handler

Sets the handler for video system more breaks to the specified routine. The info pointer should point to the following structure (declared in window_control_io.inc1.pll):

dcl more_handler_info aligned based (more_handler_info_ptr),
  2 version fixed bin,
  2 flags unaligned,
    3 old_handler_valid bit(1),
    3 pad bit(35),
  2 more_handler entry (pointer, bit(1) aligned),
  2 old_more_handler entry (pointer, bit(1) aligned);

dcl (more_handler_info_version_3);
  fixed bin internal static options (constant) init (3);

STRUCTURE ELEMENTS

version
  is the version number of this structure, and must be set to more_handler_info_version_3
  (also declared in the include file). (Input)

more_handler
  is the entry to be called at a more break. (Input for set) (Output for get) It will
  be passed two arguments, described below.

old_handler_valid
  is a flag specifying whether some other user-supplied more handler was in effect
  when the order call was made. (Output) (This can only be used with get.)

old_more_handler
  is the user supplied entry that was acting as more handler before the order call was
  made. (Output) Its value is only defined if the old_handler_valid flag is on. (This
  can only be used with get.)

The more handler routine is called with two arguments. The first is a pointer to a
structure containing information of interest to a more handler (see below), and the
second is a flag which the more handler sets to indicate whether or not output should
be flushed ("1"b to continue output, "0"b to flush output).
The structure can be found in the include file window_more_handler.incl.pl1, and is
declared as follows:

```
dcl 1 more_info aligned base (more_info_ptr),
    version fixed bin,
    more_mode fixed bin, /* which flavor */
    window_iocb_ptr pointer, /* for window that MORE'd */
    more_prompt character (80), /* MORE? */
    more_responses,
      3 n_yeses fixed bin,
      3 n_noes fixed bin,
      3 more_yeses character (32) unaligned,
        /* at most 32 yeses */
      3 more_noes character (32) unaligned;
```

**STRUCTURE ELEMENTS**

*version*  
is the version number of the structure (declared as more_handler_info_version_2 in
the include file). (Input)

*window_iocb_ptr*  
is a pointer to the iocb for the window in which the more break occurred. (Input)
Prompt output should be written to this switch, and responses should be read from
it.

*more_mode*  
is the current more mode. (Input) Constants for the different more modes are
declared in the include file window_io_attach_data.incl.pl1.

*more_prompt*  
is the current more prompt. (Input) This is the string "More? (^a for more; ^a to
discard output)" and is user-settable.

*more_responses*  
is the current set of more responses, and is declared similarly to the more_responses_info
structure in the get_more_responses order description above. (Input)
NOTES

The get_more_handler and set_more_handler control orders are available from command level and as active functions with the following io_call command:

```c
io_call window_switch get_more_handler
io_call window_switch set_more_handler more_handler
```

where more_handler is the entryname of the routine to be used as the more handler routine. The name is converted to an entry using the user's search rules and is then used as described in the set_more_handler control order.

```c
dcl break_table_info aligned based (break_table_ptr),
   2 versions fixed bin,
   2 breaks (0:127) bit (1) unaligned;
```

**STRUCTURE ELEMENTS**

versions

must be set by the caller to break_table_info_version_1. (Input)

breaks

has a "1"b for each character that is a break character. (Input/Output)

reset_more_handler

cancels the last user-defined more_handler. The reset_more_handler control order is available from command level with the following io_call command:

```c
io_call control window_switch reset_more_handler
```
get_output_conversion

This order is used to obtain the current contents of the specified table. The info_ptr points to a structure like the one described for the corresponding "set" order below, which is filled in as a result of the call (except for the version number, which must be supplied by the caller). If the specified table does not exist (no translation or conversion is required), the status code error_table_$no_table is returned.

set_output_conversion

Provides a table to be used in formatting output to identify certain kinds of special characters. The info_ptr points to the following structure (declared in tty_convert.incl.pl1). If the info_ptr is null, no transaction is to be done.

```
dcl 1 cv_trans_struc aligned
  2 version fixed bin,
  2 default fixed bin,
  2 cv_trans aligned
  3 value (0:255) fixed bin (8) unaligned
```

**STRUCTURE ELEMENTS**

- **version**
  - is the version number of the structure. It must be 2 and declared in tty_convert.incl.pl1.

- **default**
  - indicates, if nonzero, that the table is the one that was in effect before video was invoked.

- **values**
  - are the elements of the table. This table is indexed by the value of a typed input character, and the corresponding entry contains the ASCII character resulting from the translation.

get_special

is used to obtain the contents of the special_chars table currently in use. The info_ptr points to the following structure (defined in tty_convert.incl.pl1):

```
dcl 1 get_special_info_struc aligned
  2 area_ptr ptr,
  2 table_ptr ptr;
```
STRUCTURE ELEMENTS

area_ptr
    points to an area in which a copy of the current special_chars table is returned. (Input)

table_ptr
    is set to the address of the returned copy of the table. (Output)

set_special
    provides a table that specifies sequences to be substituted for certain output characters, and characters that are to be interpreted as parts of escape sequences on input. Output sequences are of the following form (defined in tty_convert.incl.pl1):

        dcl 1 c_chars based aligned,
                2 count fixed bin (8) unaligned,
                2 chars (3) char (1) unaligned;

STRUCTURE ELEMENTS

count
    is the actual length of the sequence in characters (0 <= count <= 3). If count is zero, there is no sequence.
chars

are the characters that make up the sequence. The info_ptr points to a structure of the following form (defined in tty_convert.incl.p1):

dcl 1 special_chars_struc aligned based,
    2 version fixed bin,
    2 default fixed bin,
    2 special_chars
    3 nl_seq aligned like c_chars,
    3 cr_seq aligned like c_chars,
    3 bs_seq aligned like c_chars,
    3 tab_seq aligned like c_chars,
    3 vt_seq aligned like c_chars,
    3 ff_seq aligned like c_chars,
    3 printer_on aligned like c_chars,
    3 printer_off aligned like c_chars,
    3 red_ribbon_shift aligned like c_chars,
    3 black_ribbon_shift aligned like c_chars,
    3 end_of_page aligned like c_chars,
    3 escape_length fixed bin,
    3 not_edited_escapes (sc_escape_len refer
       (special_chars.escape_length))
       like c_chars,
    3 edited_escapes (sc_escape_len refer
       (special_chars.escape_length))
       like c_chars,
    3 input_escapes aligned,
       4 len fixed bin(8) unaligned,
       4 str char (sc_input_escape_len refer
          (special_chars.input_escapes.len))
          unaligned,
    3 input_results aligned,
       4 pad bit(9) unaligned,
       4 str char (sc_input_escape_len refer
          (special_chars.input_escapes.len))
          unaligned;

NOTES

Video ignores cr_seq, bs_seq, tab_seq, vt_seq, ff_seq, printer_on, printer_off, end_of_page, input_escapes, and input results.
**STRUCTURE ELEMENTS**

- **version**
  - is the version number of this structure. It must be 1.

- **default**
  - indicates, if nonzero, that the default values for the current terminal type and baud rate are to be used and that the remainder of the structure is to be ignored.

- **nl_seq**
  - is the output character sequence to be substituted for a newline character. The nl_seq.count generally should be nonzero.

- **cr_seq**
  - is the output character sequence to be substituted for a carriage-return character. If count is zero, the appropriate number of backspaces is substituted. However, either cr_seq.count or bs_seq.count should be nonzero (i.e., both should not be zero).

- **bs_seq**
  - is the output character sequence to be substituted for a backspace character. If count is zero, a carriage return and the appropriate number of spaces are substituted. However, either bs_seq.count or cr_seq.count, should be nonzero (i.e., both should not be zero).

- **tab_seq**
  - is the output character sequence to be substituted for a horizontal tab. If count is zero, the appropriate number of spaces is substituted.

- **vt_seq**
  - is the output character sequence to be substituted for a vertical tab. If count is zero, no characters are substituted.

- **ff_seq**
  - is the output character sequence to be substituted for a formfeed. If count is zero, no characters are substituted.

- **printer_on**
  - is the character sequence to be used to implement the printer_on control operation. If count is zero, the function is not performed.

- **printer_off**
  - is the character sequence to be used to implement the printer_off control operation. If count is zero, the function is not performed.

- **red_ribbon_shift**
  - is the character sequence to be substituted for a red-ribbon-shift character. If count is zero, no characters are substituted.
black_ribbon_shift
is the character sequence to be substituted for a black_ribbon_shift character. If
count is zero, no characters are substituted.

end_of_page
is the character sequence to be printed to indicate that a page of output is full. If
count is zero, no additional characters are printed, and the cursor is left at the end
of the last line.

escape_length
is the number of output escape sequences in each of the two escape arrays.

not_edited_escapes
is an array of escape sequences to be substituted for particular characters if the
terminal is in "^edited" mode. This array is indexed according to the indicator found
in the corresponding output conversion table (see the description of the
set_output_conversion order above).

edited_escapes
is an array of escape sequences to be used in edited mode. It is indexed in the
same fashion as not_edited_escapes.

input_escapes
is a string of characters each of which forms an escape sequence when preceded by
an escape character.

input_results
is a string of characters each of which is to replace the escape sequence consisting
of an escape character and the character occupying the corresponding position in
input_escapes.

get_token_characters, set_token_characters
changes the set of characters that are used by the video system input line editor to
define a word for such requests as ESC DEL. The set of characters supplied in the
structure replace the existing set of characters. The info_ptr points to the following
structure (declared in window_control_info.incl.pl1):

dcl 1 token_characters_info aligned based
     (token_characters_info_ptr),
       2 version char (8),
       2 token_characters_count fixed bin,
       2 token_characters char (128) unaligned;

STRUCTURE ELEMENTS

version
is the version string for this structure. (Input) Its current value is
token_characters_info_version_l, also declared in the include file.
token_characters_count
is the number of characters in the token_characters string. (Input)

token_characters
is a character string containing the new set of token characters. (Input)

NOTES
The set_token_characters and get_token_characters control orders are available from command_level and as active functions with the following io_call commands:

```plaintext
io_call control window_switch get_token_characters
io_call control window_switch set_token_characters token_char_string
```

where token_char_string is a character string containing the new set of token characters.
get_token_character returns its result as a string if it was invoked as an active function, otherwise it prints out the token characters.

get_editor_key_bindings
returns a pointer to the line_editor_key_binding structure describing the key bindings.

io_call support points out the pathname of each editor routine, listing only the names of builtin requests in capital letters, with the word "builtin" in parentheses. The control order prints or returns current information about the key bindings. Use the set_editor_key_bindings control order to change the bindings. This control order prints or returns current information about the key_bindings. Use the set_editor_key_bindings control order to change the bindings.

The info_ptr points to the following structure (declared in window_control_info.incl.pl1):

```plaintext
dcl 1 get_editor_key_bindings_info aligned based (get_editor_key_binding_info_ptr),
  2 version char(8),
  2 flags,
  3 entire_state bit (1) unaligned,
  3 mbx bit (35) unaligned,
  2 key_binding_info_ptr ptr,
  2 entire_state_ptr ptr;
```

STRUCTURE ELEMENTS

version
is get_editor_key_binding_info_version_1. (Input)

entire_state
is "1"b if the entire state is desired, "0"b if only information about certain keybindings is desired. (Input)
key_binding_info_ptr
if entire_state = "0"b, then this points to a line_editor_key_binding_structure. (Input)
The bindings component of this structure is then filled in based upon the value of each key_sequence supplied.

entire_state_ptr
is set to point to the "state" of the key bindings, if entire_state = "1"b. (Output)
This is suitable input to the set_editor_key_bindings control order.

NOTES
The get_editor_key_bindings control order is available from command level and as an active function with following io_call command:

io_call control window_switch get_editor_key_bindings

The get_editor_key_bindings control order prints or returns information about a key binding. When you use it as an active function the information is returned in a form suitable as arguments to the set_editor_key_bindings control order.

set_editor_key_bindings
A line editor routine is bound to a sequence of keystrokes via the set_editor_key bindings control order. The sequence of characters that triggers an editor request may be of any length, with multiple-key sequences working like the Emacs prefix characters. This allows the use of terminal function keys (which often send three or more character sequences) to invoke line editor requests. More than one binding can be set in one invocation of this control order.

The info_ptr points to the following structure (declared in window_control_info.inc.pl1):

dcl 1 set_editor_key_bindings_info aligned based
     (set_editor_key_bindings_info_ptr),
     2 version char (8),
     2 flags,
     3 replace bit (1) unaligned,
     3 update bit (1) unaligned,
     3 pad bit (34) unaligned.
     2 key_binding_info_ptr;

STRUCTURE ELEMENTS

version
is the version of the structure. (Input) It must be set_editor_key_bindings_info_version_1.

replace
if "1"b then key_binding_info is considered to be returned by a previous get_editor_key_bindings operation with entire_state = "1"b and will be used to replace the keybinding state of the editor. (Input)
update
if "l"b then key_binding_info_ptr is considered a pointer to a line_editor_key_binding_info structure, which will be used to update the keybinding state of the editor. (Input)

Note: only one of replace and update may be true, but at least one of them must be true.

key_binding_info_ptr
is a pointer received from get_editor_key_bindings operation or a pointer to a line_editor_key_binding_info structure, depending on the value of the replace and update flags. (Input)

Notes on freeing: The video system's internal data structures are freed at the following times: video system revocation and when a set_editor_key_bindings control order with replace = "l"b is done.

NOTES

The set_editor_key_bindings control order is available from command level and as an active function with the following io_call command:

    io_call control window_switch set_editor_key_bindings key_sequence
    {user_routine1} {control_args1} ... key_sequenceN
    {user_routineN} {control_args1} {control_argsN}

where user_routine is the name of a user-written editor request.

control args are:

    -external user_routine
    -builtin builtin_request_name
    -numarg_action numarg_action_name

The line_editor_key_bindings_info structure is described in Section 7.

At least one user_routine or one of -external/-builtin must be specified for each key sequence, with the rightmost editor request specifier taking precedence (for example, io control window_switch set Editor_key_bindings foo -builtin FORWARD_word) will bind control -a to the forward word builtin, not the user routine foo.

numarg_action_name
the type of automatic numeric argument to be taken when the editor routine is invoked, must be one of the following and can only be given for external editor routines

REPEAT
(the default is PASS). This can be entered in upper or lower case. Call the user routine n times, where n is the numeric argument supplied by the user.
REJECT
ring the terminal bell and don't call the user routine if a numeric argument is given.

PASS
pass any numeric argument to the user routine, without any other action.

IGNORE
same as PASS but implies the user routine will not make use of the numeric argument.

-mode STR
specifies the name of the editor command being assigned to the key. If this is the null string, then a default name is used (for builtins this is the name of the builtin, otherwise it is segname$entrypoint). STR must be quoted if it contains whitespace.

description STR
specifies a description string to be associated with the key binding. If this is the null string, a default description is used. The defaults can be found in the include file window_editor_values.incl.pll. STR must be quoted if it contains whitespace.

-info_pathname PATH
specifies an info segment pathname to be associated with this key binding. This info segment is expected to have more information about the editor_routine. If this is not specified, it defaults to >doc>info>video_editing.gi.info if -builtin, otherwise no info segment is associated with the key. The info suffix is assumed on PATH.

MODES OPERATION

The modes operation is supported by window_io_. The recognized modes are listed below. Some modes have a complement indicated by the circumflex character (^) that turns the mode off (e.g. ^more). For these modes, the complement is displayed with that mode. Some modes specify a parameter that can take on a value (e.g. more_mode). These modes are specified as MODE=VALUE, where MODE is the name of the mode and VALUE is the value it is to be set to. Parameterized modes are indicated by the notation (P) in the following description:

more, ^more
Turns MORE processing on. Default is on. If ^pl is set before you invoke the video system, ^more will be set when you invoke the video system.

more_mode = STR
controls behavior when the window is filled. The value for STR may be one of the following:

  clear
  the window is cleared, and output starts at the home position.

  fold
  output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response when it is about to overwrite the first line written since the last read or MORE break.
lines are scrolled off the top of the window, and new lines are printed in the space that is cleared at the bottom of the screen. This is the default for full width windows on all terminals capable of scrolling.

wrap
output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response at the bottom of every window of output. This is the default for all terminals that are incapable of scrolling or when using partial width windows.

vertsp, ^vertsp
is only effective when more mode is on. When vertsp mode is on, output of a FF or VT will cause an immediate MORE query. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is ^vertsp.

rawo, ^rawo
causes characters to be output with no processing whatsoever. The result of output in this mode is undefined.

can, ^can
causes input lines to be canonicalized before they are returned. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is on.

ctl_char, ^ctl_char
specifies that ASCII control characters that do not cause newline or linefeed motion are to be accepted as input except for the NUL character. If the mode is off all such characters are discarded. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is off.

edited, ^edited
suppresses printing of characters for which there is no defined Multics equivalent on the device referenced. If edited mode is off, the 9-bit octal representation of the character is printed. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is off.

erkl, ^erkl
controls the editing functions of get_line. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is on, which allows erase and kill processing and the additional line editor functions.
esc, ^esc
controls input escape processing. When you invoke the video system, it copies the
current setting of this mode before attaching the window_io_ module. The default is
on.

rawi, ^rawi
acts as a master control for can, erkl, and esc. If this mode is on, none of the
input conventions are provided. The default is on.

ll = STR
is the width of the window, in characters, and it can only be changed with the
set_window_info control operation.

pl = STR
is the height of the window (i.e., number of lines), and it can only be changed with
the set_window_info control operation.

red, ^red
controls interpretation of red shift and black shift characters on output. When you
invoke the video system, it copies the current setting of this mode before attaching
the window_io_ module. The default is ^red, which ignores them. In red mode, the
character sequence given in the TTF is output. The effect is undefined and
terminal-specific. In some cases, "red shifted" output appears in inverse video, but
this is not guaranteed.

CONTROL OPERATIONS FROM COMMAND LEVEL

Those control operations which require no info_ptr and those additional orders described
above may be performed from command level using the io_call command, as follows:

 io_call control switch_name control_order

ARGUMENTS

switch_name
is the name of the I/O switch.

control_order
can be any control order described above under "Control Operation" that can accept a
null info_ptr.
This section contains descriptions of the FORTRAN subroutine interface to the menu and video software. Two sample FORTRAN programs are provided that illustrate menu creation using automatic window management, and the FORTRAN video interface capabilities.

In the FORTRAN environment, window management can be performed automatically. By using arguments to the FORTRAN window management functions ft_menu_$init1, ft_menu_$init2, and ft_menu_$terminate, applications that do not require sophisticated window management can employ automatic window management. When using automatic window management, your application works in two-window mode: the window in which the menu is displayed and the user input/output window.

If your application requires greater window management capabilities, the menu interface capability lets you build menu applications using the ft_window_$create, ft_window_$destroy, ft_window_$clear, and ft_window_$change capabilities.

Of course, FORTRAN applications can still use command or PL/1 video management capabilities.

Note that it is not possible to call the ft_menu_ routines with both ANSI77 and ANSI66 character strings. Currently, only ANSI77 character strings are allowed.
The ft_menu_ subroutine allows a FORTRAN program to use the Multics menu facility (menu_). Through ft_menu_ a FORTRAN program may create a menu object, display the menu, and get a user-entered selection from a menu. Once a menu object has been created, the FORTRAN program can use this menu object by referencing it via a menu-id returned to the caller when the menu object was created or when a stored menu object was retrieved.

The functionality available is provided through the various entry points defined below. Also refer to the FORTRAN include file at the end of this section.

**Entry: ft_menu_**

Utilized to create a menu object. It returns a menu identifier (menu_id) which is subsequently used to reference the menu object.

**USAGE**

declarations:

```fortran
character*n1 choices(m1)
character*n2 headers(m2)
character*n3 trailers(m3)
character*l keys(m4)
character*l pad_char
integer menu_format(6)
integer menu_needs(3)
integer menu_id
integer code
```

call ft_menu_$create (choices, headers, trailers, pad_char, menu_format,
key, menu_needs, menu_id, code)

**STRUCTURE ELEMENTS**

**choices**

is an array of character variables which are the text of the options that the user wishes to display in the menu. (Input) n1 is the length, in characters, of the longest character string comprising the text of an option. m1 is the extent of the array, i.e., the number of options in the menu being described. This array must be at least of extent 1.
headers
is an array of character variables to be displayed at the top of the menu. (Input) n2 is the length, in characters, of the longest header specified. m2 is the extent of the array, i.e., the number of headers (lines) desired. At least one header must be specified (if the first variable is set to blanks, no headers will be used).

trailers
is an array of trailers (displayed immediately below the menu). (Input) n3, m3, are analogous to n2, m2 respectively.

menu_format
is an array, which specifies the format of the menu being created. (Input) Prior to calling this entry point, the FORTRAN programmer is responsible for setting the following variables:

```fortran
menu_format(menu_version) = version number of menu_
(currently, only version 1 is defined).
menu_format(max_width) = maximum width of the window
on which the menu will be displayed.
menu_format(max_height) = maximum height of window
on which menu is to be displayed.
menu_format(no_of_columns) = number of columns to be used
by the menu manager to display the options.
menu_format(center_headers) = 0 or 1; 0 = no, 1 = yes.
menu_format(center_trailers) = 0 or 1; 0 = no, 1 = yes.
```

pad_char
is the character that the menu facility will display at the right and left of a centered header or trailer to fill out the line. (Input)

keys
is an array (maximum value of m4 is 61) that identifies the keystroke to be associated with each choice. (Input) This array must be at least as long as the number of choices in the menu. Each element in the array must be unique.

menu_needs
an array that contains menu related information on successful execution of call. (Output)

Returned information:

```fortran
menu_needs(lines_needed) the number of lines required
to display the menu.
menu_needs(width_needed) the number of columns required
to display the menu.
menu_needs(no_of_options) the number of options defined
in the menu.
```
menu_id
  the menu identifier (i.e., the menu object "identifier"). (Output) It must not be altered
  in any way by the application program.

code
  return code. (Output) (See Appendix B.)

Entry: ft_menu_$delete

Deletes a menu object from a given value segment. (See ft_menu_$store.)

USAGE

declarations:

  character*168  dir_name
  character*32   entry_name
  character*32   menu_name
  integer        code

call ft_menu_$delete (dir_name, entry_name, menu_name, code)

STRUCTURE ELEMENTS

dir_name
  pathname of the directory containing the menu object. (Input)

entry_name
  entry name of value segment containing the menu object. (Input) The suffix "value" need
  not be specified.

menu_name
  name used to identify the menu object when the menu object was stored. (Input)

code
  return code. (Output) (See Appendix B.)

Entry: ft_menu_$describe

Returns information about a menu object. It returns the number of options in the menu, the
number of lines and number of columns required to display the menu. It is primarily used
to determine if the menu can be displayed in a given window.
ft_menu_

USAGE
declarations:

    integer menu_id
    integer menu_needs(3)
    integer code

    call ft_menu_$describe (menu_id, menu_needs, code)

STRUCTURE ELEMENTS

menu_id
    the menu identifier returned by ft_menu_$create or ft_menu_$retrieve. (Input)

menu_needs
    an array into which menu related information is returned. (Output)

    Returned information:
    menu_needs(lines_needed) the number of lines required
to display the menu.

    menu_needs(width_needed) the number of columns needed
to display the menu.

    menu_needs(no_of_options) the number of options defined
in the menu.

code
    return code. (Output) (See Appendix B.)

Entry: ft_menu_$destroy

Invoked to delete a menu object from storage. (Not to be confused with ft_menu_$delete,
which deletes the menu object from a value segment.) Deleting the menu object has no
effect on the screen contents.

USAGE
declarations:

    integer menu_id
    integer code

    call ft_menu_$destroy (menu_id, code);
**ft_menu_**

**STRUCTURE ELEMENTS**

**menu_id**
  menu identifier returned by ft_menu_$create or ft_menu_$retrieve. (Input/Output) Set to an invalid value on return to prevent the old menu_id from being accidentally used.

**code**
  return code. (Output) (See Appendix B.)

**Entry: ft_menu_$display**

Invoked to display a menu in a given window.

**USAGE**

**declarations:**

```
integer window_id
integer menu_id
integer code
```

call ft_menu_$display (window_id, menu_id, code)

**STRUCTURE ELEMENTS**

**window_id**
  a window identifier returned by ft_window_$create. (Input) If usage_mode = 0 this argument will be ignored (see ft_menu_$init2).

**menu_id**
  menu identifier returned when the menu object was created or retrieved. (Input)

**code**
  return code. (Output) (See Appendix B.)

**Entry: ft_menu_$get_choice**

Returns the choice made by the user, i.e., an integer representing either the menu item chosen or the function key (or its equivalent escape sequence) entered.
USAGE

declarations:

character*n1  function_key_info
integer      window_id
integer      menu_id
integer      fkeys
integer      selection
integer      code

call ft_menu_$get_choice (window_id, menu_id, function_key_info, fkeys,
                         selection, code)

STRUCTURE ELEMENTS

window_id
a window identifier returned by ft_window_$create. (Input) If usage_mode = 0 this
argument will be ignored. (see ft_menu_$init2)

menu_id
menu identifier returned by ft_menu_$create or ft_menu_$retrieve. (Input)

function_key_info
a character variable (n1 as required) used to specify the role of function keys (if they
exist for the terminal being used) or an equivalent set of escape sequences if the
terminal does not have function keys or not the function keys required by the
application. (Input) The objective is to let the application use the terminal's function
keys if possible, else specify key sequences to be used to simulate function keys. Each
character in the string corresponds to one function key. If the character is a space, then
it is not relevant if the corresponding function key exists or not. If the character is not
a space, that character will be used to simulate a function key if the terminal does not
have function keys. If the terminal does not have a function key for every non-space
character in the string, then function keys will be simulated. Thus, the string " ?p q"
means that the caller does not care whether the terminal has function key 0 or 3, but
the caller does wish to use function keys 1,2, and 4. If any of these 3 function keys is
not present on the terminal, then esc-? will substitute for F1, esc-p will substitute for
F2, and esc-q will substitute for F4.

fkeys
if fkeys = 1 user entered a function key or escape sequence if fkeys = 0 user selected
an option (Output)

selection
is an integer representing the choice made by the user. (Output) If the user has chosen
an option, it is a number between 1 and the highest defined option. If the user has
entered a function key, or escape sequence simulating a function key, it is the number
associated with the function key.
code
   return code.  (Output)  (See Appendix B.)

Entries:  ft_menu_init1,  ft_menu_init2

These must be the first calls made to the menu manager. They set up the necessary
environment for the menu application and return information concerning the user i/o window.

USAGE

declarations:
   integer code
   integer usage_mode

   call ft_menu_init1()

   call ft_menu_init2 (usage_mode, user_window_lines, user_window_columns,
          user_window_id, code)

STRUCTURE ELEMENTS

usage_mode
   usage_mode = 0 means that the caller does not wish to do any window management at
   all.  (Input) When he/she wishes to display a menu, the window required will be
   automatically created. This means that the application will operate in a two window
   mode, the window containing the menu, and the user_io window. Both windows will be
   managed automatically for the user. If the user specifies this mode, all calls to the
   ft_window subroutine will be ignored and will return an appropriate error code. See
   Error Code Handling (Appendix B), below. All calls to the ft_menu subroutine that
   require a window identifier will ignore the user provided window_id.

   usage_mode = 1 means that the user wishes to define the number and characteristics of
   the windows to be used in the application. Thus, calls to ft_window will be supported
   and, for the entry points of ft_menu that require a window identifier, the caller must
   use a legal window_id (returned by ft_window_create).

user_window_lines
   the number of lines (rows) in the user i/o window at the time the user invokes
   ft_menu_init (which must be the first call to the menu manager in the application).
   (Output) Undefined if usage_mode = 0.

user_window_columns
   the number of columns of the user i/o window when ft_menu_init invoked.  (Output)
   Undefined if usage_mode = 0.

user_window_id
   window identifier of the user i/o window.  (Output) Undefined if usage_mode = 0.

code
   return code (See Appendix B.)  (Output)
Entry: *ft_menu_Slist*

Used to list the menu object(s) stored in value segment. The names selected are those that match a user provided string.

**USAGE**

declarations:

- character*168 dir_name
- character*32 names_array(ml)
- character*32 entry_name
- character*32 match_string
- integer no_of_matches
- integer code

```call ft_menu_Slist (dir_name, entry_name, match_string, no_of_matches, names_array, code)```

**STRUCTURE ELEMENTS**

- **dir_name**
  - pathname of directory containing the menu object. (Input)

- **entry_name**
  - entry name of value segment containing the menu object. (Input) The suffix "value" need not be specified.

- **match_string**
  - a character variable that is to be used as the selection criteria to determine what menu object, if any, is contained in the specified value segment that match (or contain) this string. (Input) If set to space(s), all names returned.

- **no_of_matches**
  - the number of matches found. (Output) If none, then is is 0.

- **names_array**
  - an array, of extent ml. (Output) The user should insure that ml is sufficiently large to contain all matches that may be found. Contains the names of all menu objects, in the specified value segment, that match the character string match_string.

- **code**
  - return code. (Output) (See Appendix B.)
Entry: ft_menu_$retrieve

Used to retrieve a menu object previously stored via the ft_menu_$store. Once retrieved, the user can reference the menu object via the menu identifier (menu_id).

**USAGE**

declarations:
- character*168 dir_name
- character*32 entry_name
- character*32 menu_name
- integer menu_id
- integer code

    call ft_menu_$retrieve (dir_name, entry_name, menu_name, menu_id, code)

**STRUCTURE ELEMENTS**

dir_name
- pathname of the directory containing the menu object. (Input)

t_entry_name
- entry name of value segment containing menu object. (Input) The suffix "value" need not be specified.

menu_name
- name of the menu object used when the object was stored. (Input)

menu_id
- is the menu id returned by the call. (Output) It is used as the menu object identifier.

code
- return code. (Output) (See Appendix B.)

Entry: ft_menu_$store

Used to store a menu object in a specified value segment.

**USAGE**

declarations:
- character*168 dir_name
- character*32 entry_name
- character*32 menu_name
- integer create_seg
- integer menu_id
- integer code
call ft_menu_$store (dir_name, entry_name, menu_name, create_seg, menu_id, code)

**STRUCTURE ELEMENTS**

**dir_name**
- pathname of directory into which the menu object is to be placed. (Input)

**entry_name**
- entry name of value segment into which the menu object is to be placed. (Input) The suffix "value" need not be specified.

**menu_name**
- it is the name to be assigned to the stored menu object. (Input)

**create_seg**
- create_seg = 0 means do not store if value segment identified by entry_name does not already exist. (Input)
- create_seg = 1 means create value segment, if it does not already exist, and store menu object in it.

**menu_id**
- it is the menu object identifier returned when ft_menu_$create or ft_menu_$retrieve was called. (Input)

**code**
- return code. (Output) (See Appendix B.)

**Entry: ft_menu_$terminate**

Must be the last call to the menu manager in the menu application. It will remove the special environment created by ft_menu_$init1 and ft_menu_$init2.

**USAGE**

**declarations:** none

```plaintext
call ft_menu_$terminate ()
```
FORTRAN INCLUDE FILE

This include file contains the following declarations:

```fortran
external ft_menu_$create (descriptors)
external ft_menu_$delete (descriptors)
external ft_menu_$describe (descriptors)
external ft_menu_$destroy (descriptors)
external ft_menu_$display (descriptors)
external ft_menu_$get_choice (descriptors)
external ft_menu_$init1 (descriptors)
external ft_menu_$init2 (descriptors)
external ft_menu_$list (descriptors)
external ft_menu_$retrieve (descriptors)
external ft_menu_$store (descriptors)
external ft_window_$change (descriptors)
external ft_window_$create (descriptors)
external ft_window_$destroy (descriptors)

integer menu_version
integer max_width
integer max_height
integer no_of_columns
integer lines_needed
integer width_needed
integer no_of_options
integer center_headers
integer center_trailers
integer user_window_id
integer user_window_lines
integer user_window_columns

parameter (menu_version = 1)
parameter (max_width = 2)
parameter (max_height = 3)
parameter (no_of_columns = 4)
parameter (center_headers = 5)
parameter (center_trailers = 6)
parameter (lines_needed = 1)
parameter (width_needed = 2)
parameter (no_of_options = 3)
```
Name: ft_window_

This is the basic video interface subroutine to be used by FORTRAN to create/destroy/change windows. (This subroutine should not be called if usage_mode = 0 (see ft_menu_$init2)).

Its facilities are available through the following entry points.

Entry: ft_window__$change

This entry point is used to change the size of an existing window. The size of a window can always be "shrunk", however it can be increased only if it does not overlap with another defined window. (This entry point should not be called if usage_mode = 0 (see ft_menu_$init2)).

USAGE declarations:

integer window_id
integer first_line
integer height
integer code

call ft_window__$change (window_id, first_line, height, code)

STRUCTURE ELEMENTS

window_id
window identifier returned by ft_window__$create (or by ft_menu_$init in the case of the user i/o window). (Input)

first_line
new first line number for the window being changed. (Input) Positive integer.

height
new height for the window being changed. (Input) Positive integer.

code
return code. (Output) (See Appendix B.)
Entry: ft_window_$clear_window

Used to clear a specified window.

 USAGE

 declarations:
 integer window_id
 integer code

call ft_window_$clear_window (window_id, code)

 STRUCTURE ELEMENTS

 window_id
 The window identifier (returned by ft_window_$create) of the window to be cleared. (Input)

code
 return code. (Output) (See Appendix B.)

Entry: ft_window_$create

Used to create a new window on the terminal screen. (This entry point should not be called if usage_mode \neq 0.) (see ft_menu_$init2)

 USAGE

 declarations:
 character*32 switch_name
 integer window_id
 integer first_line
 integer height
 integer code

call ft_window_$create (first_line, height, switch_name, window_id, code)

 STRUCTURE ELEMENTS

 first_line
 is the line number where the window is to start. (Input)

 height
 the number of lines used by the window, i.e., its height. (Input)
switch_name
the name that the caller wishes to associate with the switch. (Input) (The caller may use
the switch name, for example, in the FORTRAN "open" statement.)

window_id
the returned id of the window just created. (Output) It must not be altered in any way
by the application program.

code
return code. (Output) (See Appendix B.)

Entry: ft_window_$destroy

Used to destroy a previously created window. (This entry point should not be called if
usage_mode = 0 (see ft_menu_$init2).)

USAGE
declarations:
    integer window_id
    integer code

call ft_window_$destroy (window_id, code)

STRUCTURE ELEMENTS

window_id
    window identifier (returned by the ft_window_$create). (Input/Output) It is reset to an
    illegal value by this call.

code
    return code. (Output) (See Appendix B.)

FORTRAN MENU APPLICATION EXAMPLES

In the following two FORTRAN examples, a "Message" menu application is created that
allows you to display, print, discard, or forward messages. Example 1 is a simple FORTRAN
program that interfaces with the Multics menu manager via the ft_menu_ routine. Note in
Example 1 that window management functions are called automatically through arguments in
the ft_menu_$init2 subroutine.

Example 2 is a FORTRAN program that interfaces with the Multics menu manager through
ft_menu_routine; in example 2, however, window management functions are performed by the
ft_window_ routine.
**EXAMPLE 1:**

In this example, all window management is done automatically.

```fortran
subroutine testcase1()

%include ft_menu_dcls

external ft_menu$sinit$(descriptors)
external ft-menu$init2 (descriptors)
character*15 choices (6)
character*12 headers (1)
character*27 trailers (1)
character*1 keys (6)
character*168 dir_name
character*32 entry_name
character*32 menu_name
character*12 function_key_info
character*32 switch_name
character*9 ME
integer create_seg
integer no_of_matches
integer window_id
integer fkeys
integer selection
integer usage_mode
integer menu_format (6)
integer menu_needs (3)
integer menu_id
integer code
integer zero

external com_err_(descriptors)

integer too_few_keys
integer bad_arg
integer keys_not_unique

ME = "testcase1"
zero = 0

choices(1) = "Display Message"
choices(2) = "Print Message"
choices(3) = "Discard Message"
choices(4) = "Forward Message"
choices(5) = "Reply Message"
choices(6) = "List Messages"
headers(1) = "READ MAIL"
trailers(1) = "Press F1 (or esc-q) to quit"
keys(1) = "1"
keys(2) = "2"
```
keys(3) = "3"
keys(4) = "4"
keys(5) = "5"
keys(6) = "6"
pad_char = "-"

menu_format(menu_version) = 1
menu_format(max_width) = 79
menu_format(max_height) = 10
menu_format(no_of_columns) = 2
menu_format(center_headers) = 1
menu_format(center_trailers) = 1

code = 0
usage_mode = 0 ! Window management will be done automatically
! by the system, i.e., usage_mode is set to 0.
! by the system, i.e., usage_mode is set to 0.
call ft_menu$init1()
call ft_menu$init2(usage_mode,user_window_lines,user_window_columns,
user_window_id,code)
! Calling ft_menu$init MUST
! be the first call to ft_menu_ in the program.

if (code .eq. zero) go to 5
call com_err_ (code, ME, "(calling ft_menu$init2)"
print, "Unable to set up the appropriate environment for the application."
go to 999

c The following calls to cv_error$name are used retrieve and store
the error codes associated with certain errors of interest returned
by calls to the menu manager or the system.

5     call cv_error$name ("error_table$bad_arg", bad_arg, code)
if (code .eq. zero) go to 10
call com_err_ (code, ME, "error_table$bad_arg")
go to 999

10    call cv_error$name ("menu_et$too_few_keys",too_few_keys,code)
if (code .eq. zero) go to 20
call com_err_ (code, ME, "menu_et$too_few_keys")
go to 999

20    call cv_error$name ("menu_et$keys_not_unique", keys_not_unique, code)
if (code .eq. zero) go to 40
call com_err_ (code, ME, "menu_et$keys_not_unique")
go to 999

40    call ft_menu$create (choices,headers,trailers,pad_char,menu_format,
& keys,menu_needs,menu_id,code)

c This call creates the menu object and returns the menu object identifier,
c "menu_id".
if (code .eq. zero) go to 45
    call com_err_ (code, ME, " (calling ft_menu_$create)"
    print, "The menu could not be created."
    go to 999

45    dir_name = ">udd>r"       ! pathname of directory
        entry_name = "menus_seg"       ! entry name of "value" segment
        menu_name = "ft_read_mail_menu" ! name of menu
        create_seg = 1           ! create "value" seg if it does not already exist.
    call ft_menu_$store (dir_name, entry_name, menu_name, create_seg, menu_id, code)
    if (code .eq. zero) go to 50
    call com_err_ (code, ME, " (calling ft_menu_$store)"
    print, "The menu could not be stored."
    go to 999

50    window_id = 0
    call ft_menu_$display (window_id, menu_id, code) ! This call displays
        ! the menu in its own window at top of screen. Since the usage_mode
        ! was set to 0, the program does not have to create the window
        ! before calling ft_menu_$display. The window_id argument is ignored.
    if (code .eq. zero) go to 60
    call com_err_ (code, ME, " (calling ft_menu_$display)"
    print, "The menu could not be displayed."
    go to 999

60    function_key_info = "q" ! Defines the function key requirements, i.e,
        ! if the terminal has function key 1 (F1) then F1 will be used
        ! to "quit", otherwise "esc_q" will be used to "quit".

61    call ft_menu_$get_choice (window_id, menu_id, function_key_info, fkeys, 
        & selection, code)

    This call accepts the user input from the menu. On return, the variable
    "selection" will contain a number (1, 2, 3, or 4) representing the option
    chosen by user.
    Note: if the user entered anything other than 1 or 2 or 3 or 4
    the terminal "beeped", and the user input was ignored.
    Since usage_mode is 0, the window_id argument is ignored.

    if (code .eq. zero) go to 90
    if (code .ne. too_few_keys) go to 70
    call com_err_ (0, ME, "Number of keys less than number of options."
    go to 999

70    if (code .ne. keys_not_unique) go to 80
    call com_err_ (0, ME, "Option keys not unique."
    go to 999

8-18       CP51-02
80    call com_err_ (code, ME, "(calling ft_menu_$get_choice).
           An internal programming error has occurred.")
           go to 999
90    if (fkeys .eq. zero) go to 110
    if (fkeys .eq. 1) go to 100
    print, "An internal program error has occurred. Quitting."
           go to 999
100   if (selection .ne. 1) go to 61
    print, "You entered '"Fl'" or '"esc qa"'. Quitting."
           go to 999
110   print 103, selection
103   format ("You selected option 'il)"
           go to 50
999   call ft_menu_$terminate()
           return
end

EXAMPLE 2:

In this example, FORTRAN interfaces with the Multics menu manager and the Multics
window manager via the ft_menu_ and ft_window_ subroutines.

subroutine testcase2 ()
    include ft_menu_dcls
    external ft_menu_$init1(descriptors)
    external ft_menu_$init2(descriptors)
    external ft_window_$clear_window (descriptors)
    character*9   choices_one(2)
    character*21  choices_three(4)
    character*21  headers(1)
    character*49  trailers(1)
    character*1   keys(6)
    character*168 dir_name
    character*32  entry_name
    character*32  menu_name
    character*12  function_key_info
    character*32  match_string
    character*32  names_array(10)
    character*32  switch_name
    character*9   ME
    integer       create_seg
    integer       no_of_matches
    integer       window_id1
    integer       window_id2
    integer       fkeys
    integer       selection
    integer       usage_mode

```c
integer menu_format (6)
integer menu_needs_one (3)
integer menu_needs_two (3)
integer menu_needs_three (3)
integer curr_window_id
integer menu_id1
integer menu_id2
integer menu_id3
integer code
integer zero

external com_err_(descriptors)

integer bad_window_id
integer nonexistent_window
integer insuff_room_for_window

ME = "testcase2"
zero = 0

choices_one (1) = "Read Mail"
choices_one (2) = "Send Mail"
choices_three (1) = "Send New Message"
choices_three (2) = "Send Deferred Message"
choices_three (3) = "Print Sent Message"
choices_three (4) = "Save Sent Message"
trailer (1) = "F1 (or esc-q) = quit ; F2 (or esc-f) = first menu"
keys (1) = "1"
keys (2) = "2"
keys (3) = "3"
keys (4) = "4"
keys (5) = "5"
keys (6) = "6"

pad_char = "="

menu_format(menu_version) = 1
menu_format(max_width) = 79
menu_format(max_height) = 8
menu_format(no_of_columns) = 2
menu_format(center_headers) = 1
menu_format(center_trailers) = 1

code = 0
call ft_menu_$init1 ()
usage_mode = 1 Window management will be done by user
call ft_menu_$init2 (usage_mode, user_window_lines, user_window_columns,

& user_window_id, code) Calling ft_menu_$init MUST be the first call to ft_menu_ in the program.
```
if (code .eq. 0) go to 5
  call com_err_ (code, ME, " (calling ft_menu_$init)"
  print, "Unable to set up the appropriate environment for the
  application."
  go to 999

c The following calls to cv_error_$name are used retrieve and store
the error codes associated with certain errors of interest returned
by calls to the menu manager or the system.

    call cv_error_$name ("video_et$_bad_window_id", bad_window_id, code)
    if (code .eq. zero) go to 10
    call com_err_ (code, ME, "video_et$_bad_window_id")
    go to 999

    call cv_error_$name ("video_et$_nonexistent_window", nonexistent_window, code)
    if (code .eq. zero) go to 20
    call com_err_ (code, ME, "video_et$_nonexistent_window")
    go to 999

    call cv_error_$name ("video_et$_insuff_room_for_window", insuff_room_for_window, code)
    if (code .eq. zero) go to 40
    call com_err_ (code, ME, "video_et$_insuff_room_for_window")
    go to 999

c Create first menu

    headers(1) = "MULTICS MAIL"
    call ft_menu_$create (choices_one,headers,trailers,pad_char,menu_format,
     &     keys,menu_needs_one,menu_id1,code)

    This call creates the menu object and returns the menu object identifier.
    This menu is referenced by menu_id1.

    if (code .eq. 0) go to 41
    call com_err_ (code, ME, " (calling ft_menu_$create)"
    print, "The first menu could not be created."
    go to 999

c For the second menu use a menu object which was stored in a "value" seg.
c First determine if menu object exists.

    dir_name = ">udd>m>ri"
    entry_name = "menus_seg"
    match_string = "ft_read_mail_menu"
    call ft_menu_$list (dir_name,entry_name,match_string,no_of_matches,
     &     names_array,code)
    if (code .eq. zero) go to 42
    call com_err_ (code, ME, " (calling ft_menu_$_list)"
    go to 999

    if (no_of_matches .eq. zero) then
print, "Stored menu not found."
go to 999
else
if (no_of_matches .eq. 1) go to 43
print, "Internal error. Quitting."
go to 999
end if
c Retrieve stored menu.
43    menu_name = "ft_read_mail_menu"
call ft_menu_$retrieve (dir_name,entry_name,menu_name,menu_id2,code)
if (code .eq. zero) go to 44
    call com_err_ (code, ME, " (calling ft_menu_$retrieve)"
    go to 999
c Get attributes of retrieved menu.
44    call ft_menu_$describe (menu_id2,menu_needs_two,code)
if (code .eq. zero) go to 45
    call com_err_ (code, ME, " (calling ft_menu_$describe)"
    go to 999
c Create third menu
45    headers(l) = "SEND MAIL"
call ft_menu_$create (choices_three,headers,trailers,pad_char,
& menu_format,keys,menu_needs_three,menu_id3,code)
if (code .eq. 0) go to 50
    call com_err_ (code, ME, " (calling ft_menu_$create)"
    print, "The third menu could not be created."
go to 999
50    curr_window_id = -1  "-1" indicates that there is no current menu
being displayed; otherwise, curr_window_id contains the menu window id
52    call change_menu (user_window_id,curr_window_id,menu_id1,menu_needs_one,
& user_window_lines,window_idl,code)
if (code) = 51,53,51
51    call com_err_ (code,"change_menu","Internal error while changing menus.")
go to 999
53    call ft_window$_clear_window (user_window_id, code)
60    call get_choice (menu_id1,window_id1,fkeys,selection,code)
c This call accepts the user input from the menu. On return, the variable
c "selection" will contain a number (0, 1, 2) representing the option or
c the function key (or its equivalent escape sequence) entered by the user.
c If fkeys = 1 then the user entered F1 or F2 (or esc-q or esc-f):
if Fl (or esc-q) was entered, then selection = 0
if F2 (or esc-f) was entered, then selection = 1
If fkeys = 0 then the user selected option:
if first option was chosen, then selection = 1
if second option was chosen, then selection = 2
Note: if the user entered anything other than Fl or F2 or 1 or 2
the terminal "beeped", and the user input was ignored.

if (code .eq. zero) go to 70
call com_err_ (0, "get_choice", "Internal program error
while getting user choice")
go to 999
70 if (fkeys .eq. zero) go to 90 user selected an option
if (fkeys .eq. 1) then
    go to 80 user entered function key
else something is wrong
    print, "An internal program error has occurred. Quitting."
go to 999
end if
80 go to (81,82), selection
call com_err_ (code, ME, "An internal program has occurred. Quitting.")
go to 999
81 print, "Exiting" (user has entered F1 or esc-q. Wants to exit)
go to 999
82 print, "You already are in the first menu." User want to go to
    first menu
    go to 60
90 go to (100,170), selection Display either "Read Mail" or "Send Mail"
    menu
call com_err_ (code, ME, "Internal program error. Quitting.")
go to 999
100 call change_menu (user_window_id,window_id,menu_id2,menu needs two,
    & user_window_lines, window_id2, code)
    if (code .eq. zero) go to 110
call com_err_ (code, "change_menu", "Internal error occurred
    while switching menus")
go to 999
110 call get_choice (menu_id2, window_id2, fkeys, selection, code)
    if (code .ne. zero) then
call com_err_ (code, "get_choice", "Internal error
    while getting user choice").
go to 999
    end if
    go to (160,150), fkeys + 1
call com_err_ (code, ME, "Internal program error. Quitting.")
go to 999
150 go to (151,152), selection user entered function key
    go to 110
151 print, "Exiting at your request"
go to 999
152 curr_window_id = window_id2

8-23 CP51-02
c User chose "Send Mail" option

170 call change_menu (user_window_id, window_id1, menu_id3, menu_needs_three, &
                    user_window_lines, window_id2, code)
    if (code) 171, 180, 171
171 call com_err_ (code, "change_menu", "Internal error
                    while changing menus")
    go to 999
180 call get_choice (menu_id3, window_id2, fkeys, selection, code)
    if (code) 181, 190, 181
181 call com_err_ (code, "get_choice", "Internal error
                    while getting user choice")
    go to 999
190 go to (210, 200), fkeys + 1
    print, "Internal error. Quitting"
    go to 999
200 go to (201, 202), selection
    go to 180
201 print, "Exiting at your request."
    go to 888
202 curr_window_id = window_id2
    go to 52
210 print 301, selection
301 format ("You selected option "il)
    go to 180
    c Delete second menu from the value seg.
888 call ft_menu_delete (dir_name, entry_name, menu_name, code)
    if (code .eq. zero) go to 999
    print, "Menu could not be deleted from value segment."
999 call ft_menu_terminate() return
end
subroutine get_choice (menu_id, window_id, fkeys, selection, code)

external ft_menu_$get_choice (descriptors)

character*2 function_key_info
integer fkeys
integer selection
integer menu_id
integer window_id
integer code

code = 0

function_key_info = "qf"  Defines the function key requirements, i.e.,
if the terminal has function keys 1 and 2 (F1 and F2) then F1
will be used to "quit" and F2 to switch to the first menu,
otherwise "esc_q" will be used to "quit" and "esc-f" to switch
to the first menu

call ft_menu_$get_choice (window_id, menu_id, function_key_info, fkeys, &
                             selection, code)

return
end

subroutine change_menu (user_window_id, curr_window_id, menu_id, menu_needs,
                        user_window_lines, window_id, code)

external ft_window_$change (descriptors)
external ft_window_$create (descriptors)
external ft_window_$destroy (descriptors)
external ft_menu_$display (descriptors)
external com_err_ (descriptors)

character*32 switch_name

integer menu_needs(3)
integer user_window_id
integer user_window_columns
integer user_window_lines
integer curr_window_id
integer menu_id
integer window_id
integer code
integer first_line
integer height

parameter (lines_needed = 1)
c Destroy the current menu-window
   if (curr_window_id + 1) 90,100,90
      call ft_window$destroy (curr_window_id,code)
      if (code) 999,100,999
   c Change the size of the user i/o window to accomodate the new menu-window
100   first_line = 1 + menu_needs(lines_needed)
      height = user_window_lines - menu_needs(lines_needed)
      call ft_window$change (user_window_id,first_line,height,code)
      if (code) 999,110,999
   c Create window for new menu
110   switch_name = "menu_window"
      call ft_window$create (1,menu_needs(lines_needed),switch_name,window_id,
         & code)
      if (code) 999,120,999
   c Display the menu in the menu-window
120   call ft_menu$display (window_id,menu_id,code)
999   return
end
This section contains descriptions of the COBOL interface to the menu and video software. Two sample COBOL programs are provided that illustrate menu creation using automatic window management, and the COBOL video interface capabilities.

In the COBOL environment, window management can be performed automatically. By using the COBOL window management functions cb_menu_$init1, cb_menu_$init2, and cb_menu_$terminate, applications that do not require sophisticated window management can employ automatic window management activity. When using automatic window management, your application works in two-window mode: the window in which the menu is displayed and the user_I/O window.

If your application requires greater window management capabilities, the menu interface capability lets you build viable menu applications using the cb_window_$create, cb_window_$destroy, and cb_window_$change capabilities.

Of course, COBOL applications can still use command or PL/1 video management capabilities.
Name: cb_menu_

The cb_menu_ subroutine allows a COBOL program to use the Multics menu facility (menu_). Through cb_menu_ a COBOL program may create a menu object, display the menu, and get a user-entered selection from a menu. Once a menu object has been created, the COBOL program can use this menu object by referencing it via a menu-id returned to the caller when the menu object was created or when a stored menu object was retrieved.

The functionality available is provided through the various entry points described below.

Entry: cb_menu_$_create

Utilized to create a menu-object. Returns a menu-id which may be subsequently used by other entry points.

USAGE

declarations:

01 choices-table.
   02 choices PIC X(n1) OCCURS (m1) TIMES.
01 headers-table.
   02 headers PIC X(n2) OCCURS (m2) TIMES.
01 trailers-table.
   02 trailers PIC X(n3) OCCURS (m3) TIMES.
01 keys-table.
   02 keys PIC X(1) OCCURS (m4) TIMES.

01 menu-format.
   02 menu_version USAGE IS COMP-6
   02 constraints USAGE IS COMP-6
      03 max-width.
      03 max-height.
   02 no-of-columns USAGE IS COMP-6.
   02 flags.
      03 center-headers PIC 9(1).
      03 center-trailers PIC 9(1).
   02 pad-char PIC X(1).

01 menu-needs USAGE IS COMP-6.
   02 lines-needed.
   02 width-needed.
   02 no-of-options.

77 menu-id USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.
CALL "cb_menu_$create" USING choices-table, headers-table, trailers-table, menu-format, keys-table, menu-needs, menu-id, ret-code.

**STRUCTURE ELEMENTS**

- **choices-table**
  - is a table of elementary data items which are the text of the options that the user wishes to display in the menu. \( n1 \) is the length, in characters, of the longest character string comprising the text of an option. \( m1 \) is the extent of the table, i.e., the number of options in the menu being described. This table must be at least of extent 1.

- **headers-table**
  - is a table of elementary data items to be displayed at the top of the menu. (Input) \( n2 \) is the length, in characters, of the longest header specified. \( m2 \) is the extent of the table, i.e., the number of headers (lines) desired. At least one header must be specified (if the first header is set to space(s), no headers will be used).

- **trailers-table**
  - is a table of trailers (displayed immediately below the menu). (Input) \( n3, m3 \) are analogous to \( n2, m2 \) respectively.

- **menu-format**
  - is a group item defining the format of the menu being created. (Input)
In the COBOL program the caller is responsible for setting the following elementary data items:

- **menu-version**: the version number of the menu facility. (Only version 1 is currently defined)
- **max-width**: maximum width of the window on which the menu is to be displayed.
- **max-height**: maximum height of window on which the menu is to be displayed.
- **no-of-columns**: number of columns to be used to display the options.
- **center-headers**: 0 or 1; 0 = no, 1 = yes.
- **center-trailers**: 0 or 1 (same as center-headers)

**keys-table**

is a table (maximum value of m4 is 61) that identifies the keystroke to be associated with each choice. (Input) This table must be at least as long as the number of choices in the menu. Each element in the table must be unique.

**menu-needs**

a group item that contains menu related information on successful execution of call. (Output)

Returned information:

- **lines-needed**: the number of lines required to display the menu.
- **width-needed**: the number of columns needed to display the menu.
- **no-of-options**: the number of options defined in the menu.

**menu-id**

the menu-object identifier (i.e., it is the menu object "pointer"). (Output) It must not be altered in any way by the application program.

**ret-code**

return code. (Output) (See Appendix B.)
Entry: cb_menu_$delete

Deletes a menu object from a given value segment.

**USAGE**

declarations:

```
77 dir-name PIC X(168).
77 entry-name PIC X(32).
77 name-of-menu PIC X(32).
77 ret-code USAGE IS COMP-6.
```

call "cb_menu_$delete" USING dir-name, entry-name, name-of-menu, ret-code.

**STRUCTURE ELEMENTS**

dir-name
   pathname of the directory containing the menu object. (Input)

entry-name
   entry name of value segment containing the menu object. (Input) The suffix "value" need not be specified.

name-of-menu
   name used to identify the menu object when the menu object was stored. (Input)

ret-code
   return code. (Output) (See Appendix B.)

Entry: cb_menu_$describe

Returns information about a menu object. It returns the number of options in the menu, the number of lines and number of columns required to display the menu. It is primarily used to determine if the menu can be displayed in a given window.
cb_menu_

**USAGE**

declarations:

```assembler
01 menu-needs USAGE IS COMP-6.
  02 lines-needed.
  02 width-needed.
  02 no-of-options.

77 menu-id USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.

  call "cb_menu_$describe" USING menu-id, menu-needs, ret-code.
```

**STRUCTURE ELEMENTS**

**menu-id**
- the menu identifier returned by `cb_menu_$create` (or `cb_menu_$retrieve` in cases where the menu object has been stored). (Input)

**menu-needs**
- a group item that contains menu-related information on successful execution of call. (Output)

  Returned information:
  ```
  lines-needed the number of lines needed to display the menu.
  width-needed the number of columns needed to display the menu.
  no-of-option the number of options defined in the menu.
  ```

**ret-code**
- return code. (Output) (See Appendix B.)

**Entry: cb_menu_$destroy**

Used to free storage of a menu (not to be confused with `cb_menu_$delete`, which is used to delete the menu object from a value segment). Destroying the menu has no effect on the screen contents.
cb_menu_

**USAGE**

declarations:

77 menu-id USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.

call "cb_menu_$destroy" USING menu-id, ret-code.

**STRUCTURE ELEMENTS**

**menu-id**
- menu identifier returned by cb_menu_$create or cb_menu_$retrieve. (Input/Output) (If usage-mode is 0 (see cb_menu_$init2) this operand will be ignored.) Set to an invalid value on return to prevent the old menu-id from being accidentally used.

**ret-code**
- return code. (Output) (See Appendix B.)

**Entry: cb_menu__$display**

Invoked to display a menu in a given window.

**USAGE**

declarations:

77 window-id USAGE IS COMP-6.
77 menu-id USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.

call "cb_menu__$display" USING window-id, menu-id, ret-code.

**STRUCTURE ELEMENTS**

**window-id**
- a window identifier returned by cb_window__$create entry point. (Input) If usage-mode = 0 this operand will be ignored (see cb_menu__$init2).

**menu-id**
- menu identifier returned when the menu object was created or retrieved. (Input)

**ret-code**
- return code. (Output) (See Appendix B.)
Entry: cb_menu_$get_choice

Returns the choice made by the user, i.e., a number representing either the menu item chosen or the function key (or its equivalent escape sequence) entered.

**USAGE declarations:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>77 function-key-info</td>
<td>PIC X(n1).</td>
</tr>
<tr>
<td>77 window-id</td>
<td>USAGE IS COMP-6.</td>
</tr>
<tr>
<td>77 menu-id</td>
<td>USAGE IS COMP-6.</td>
</tr>
<tr>
<td>77 fkeys</td>
<td>USAGE IS COMP-6.</td>
</tr>
<tr>
<td>77 selection</td>
<td>USAGE IS COMP-6.</td>
</tr>
<tr>
<td>77 ret-code</td>
<td>USAGE IS COMP-6.</td>
</tr>
</tbody>
</table>

```
call "cb_menu_$get_choice" USING window-id, menu-id, function-key-info, fkeys, selection, ret-code.
```

**STRUCTURE ELEMENTS**

**window-id**

A window identifier returned by the cb_window_$create entry point. (Input) If usage-mode = 0 this operand will be ignored (see cb_menu_$init2).

**menu-id**

A menu identifier returned by cb_menu_$create or cb_menu_$retrive. (Input)

**function-key-info**

A character elementary data item (n1 as required) used to specify the role of function keys (if they exist for the terminal being used) or an equivalent set of escape sequences if the terminal does not have function keys or not the function keys required by the application. (Input) The objective is to let the application use the terminal's function keys if possible, else specify key sequences to be used to simulate function keys. Each character in the string corresponds to one function key. If the character is a space, then it is not relevant if the corresponding function key exists or not. If the character is not a space, that character will be used to simulate a function key if the terminal does not have function keys. If the terminal does not have a function key for every non-space character in the string, then function keys will be simulated. Thus, the string " ?p q" means that the caller does not care whether the terminal has function key 0 or 3, but the caller does wish to use function keys 1, 2, and 4. If any of these 3 function keys is not present on the terminal, then esc-? will substitute for F1, esc-p will substitute for F2, and esc-q will substitute for F4.

**fkeys**

fkeys = 1 user entered a function key or escape sequence fkeys = 0 user selected an option (Output)
selection
is a number representing the choice made by the user. (Output) If the user has chosen an option, it is a number between 1 and the highest defined option. If the user has entered a function key, or escape sequence simulating a function key, it is the number associated with the function key.

ret-code
return code. (Output) (See Appendix B.)

Entries: cb_menu_sinit1, cb_menu_sinit2

These must be the first calls made to the menu manager. They set up the necessary environment for the menu application and return information concerning the user I/O window.

USAGE

declarations:

```
integer usage-mode

call cb_menu_sinit1

call cb_menu_sinit2 (usage-mode, user-window-lines, user-window-columns,
   user-window-id, ret-code)
```

STRUCTURE ELEMENTS

usage-mode
usage-mode = 0 means that the caller does not wish to do any explicit window management. (Input) When he/she wishes to display a menu, the window required will be automatically created. This means that the application will operate in a two window mode, the window containing the menu, and the user_io window. Both windows will be managed automatically for the user. If the user specifies this mode, all calls to the cb_window_ subroutine will be ignored and will return an appropriate error code. See Error Code Handling, below. All calls to the cb_menu_ subroutine that require a window identifier will ignore the user provided window-id.

usage-mode = 1 means that the user wishes to define the number and characteristics of the windows to be used in the application. Thus, calls to cb_window_ will be supported and, for the entry points of cb_menu_ that require a window identifier, the caller must use a legal window-id (returned by cb_window_$create).

user-window-lines
the number of physical lines (rows) of the user i/o window when cb_menu_sinit is called (which must be the first cb_menu_ call in the application.) Undefined if usage-mode = 0. (Output)
user-window-columns
the number of columns of the user i/o window at time that cb_menu_$init is called (see
immediately above). (Output) Undefined if usage-mode = 0.

user-window-id
window identifier of the user i/o window. (Output) Undefined if usage-mode = 0.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$list

Used to list the menu object(s), stored in value segment. The menu objects selected are those
that match the string input by the caller.

USAGE
declarations:

01 matched-names.
  02 no-of-matches USAGE IS COMP-6.
  02 menu-names PIC X(32) OCCURS (ml) TIMES.

77 dir-name PIC X(168).
77 entry-name PIC X(32).
77 match-string PIC X(32).
77 ret-code USAGE IS COMP-6.

call "cb_menu_$list" USING dir-name, entry-name, match-string,
matched-names, ret-code.

STRUCTURE ELEMENTS

dir-name
  pathname of directory containing the menu object. (Input)

entry-name
  entry name of value segment containing the menu object. (Input) The suffix "value" need
  not be specified.

match-string
  a character elementary data item that is to be used as the selection criteria for
determining what menu object, if any, is contained in the specified value segment that
match (or contain) this string. (Input)

no-of-matches
  the number of matches found. (Output) If none, then it is 0.
menu-names
On return, contains the names of all menu objects, in the specified value segment, that match the character string match-string. (Output) Note, if m1 is not large enough to contain all the names, only m1 names will be returned.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$retrieve
Used to retrieve a menu object previously stored via the cb_menu_$store subroutine.

USAGE
declarations:

77 dir-name     PIC X(168).
77 entry-name   PIC X(32).
77 name-of-menu PIC X(32).
77 menu-id      USAGE IS COMP-6.
77 ret-code     USAGE IS COMP-6.

call "cb_menu_$retrieve" USING dir-name, entry-name, name-of-menu, menu-id, ret-code.

STRUCTURE ELEMENTS

dir-name
pathname of the directory containing the menu object. (Input)

entry-name
entry name of value segment containing menu object. (Input) The suffix "value" need not be specified.

name-of-menu
name of the menu object used when the object was stored. (Input)

menu-id
is the menu id returned by the call. (Output)

ret-code
return code. (Output) (See Appendix B.)
Entry: cb_menu_$store

Used to store a menu object in a specified value segment.

**USAGE declarations:**

```plaintext
77 dir-name PIC X(168).
77 entry-name PIC X(32).
77 name-of-menu PIC X(32).
77 create-seg USAGE IS COMP-6.
77 menu-id USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.
```

```plaintext
call "cb_menu_$store" USING dir-name, entry-name, name-of-menu,
    create-seg, menu-id, ret-code.
```

**STRUCTURE ELEMENTS**

dir-name
- pathname of directory into which the menu object is to be placed. (Input)

tentry-name
- entry name of value segment into which menu object is to be placed. (Input) The suffix "value" need not be specified.

name-of-menu
- is the name to be assigned to the stored menu object. (Input)

create-seg
- create-seg = 0 means do not store if value segment identified by entry-name does not already exist. (Input) create-seg = 1 means create value segment, if it does not already exist, and store menu object in it.

menu-id
- is the menu object identifier returned by cb_menu_$create or cb_menu_$retrieve. (Input)

ret-code
- return code. (Output) (See Appendix B.)
Entry: cb_menu_Sterminate

Must be the last call to the menu manager in the menu application.

USAGE

declarations: none

    call "cb_menu_Sterminate".

STRUCTURE ELEMENTS

    There are no arguments.
**Name: cb_window__**

This is the basic video interface subroutine to be used by COBOL to create/destroy/change windows. (If usage-mode = 0 (see cb_menu_$init2) this subroutine should not be called.)

Its facilities are available through the following entry points.

**Entry: cb_window__$change**

This entry points provides a facility for changing the size of an existing window. The size of a window can always be "shrunk", however it can be increased only if it does not overlap with another defined window. (If usage-mode = 0 (see cb_menu_$init2) this entry point should not be called.)

**USAGE**

declarations:

```
77 window-id   USAGE IS COMP-6.
77 first-line USAGE IS COMP-6.
77 height     USAGE IS COMP-6.
77 ret-code   USAGE IS COMP-6.
```

```
call "cb_window__$change" USING window-id, first-line, height, ret-code.
```

**STRUCTURE ELEMENTS**

- **window-id**
  - window identifier returned by cb_window__$create. (Input)

- **first-line**
  - new first line number for the window being changed. (Input) A positive value.

- **height**
  - new height for the window being changed. (Input) A positive value.

- **ret-code**
  - return code. (Output) (See Appendix B.)
Entry: cb_window_$clear_window

Used to clear a specified window.

**USAGE**

**declarations:**

```
77 window-id  USAGE IS COMP-6.
77 ret-code   USAGE IS COMP-6.
```

call "cb_window_$clear_window" USING window-id, ret-code.

**STRUCTURE ELEMENTS**

- **window-id**
  - the window identifier (returned by cb_window_$create) of the window to be cleared. (Input)

- **ret-code**
  - return code. (Output) (See Appendix B.)

Entry: cb_window_$create

This entry is used to create a new window on the terminal screen. (If usage-mode = 0 (see cb_menu_$init2) this entry point should not be called.)

**USAGE**

**declarations:**

```
77 switch-name  PIC X(32).
77 first-line   USAGE IS COMP-6.
77 height       USAGE IS COMP-6.
77 window-id    USAGE IS COMP-6.
77 ret-code     USAGE IS COMP-6.
```

call "cb_window_$create" USING first-line, height, switch-name, window-id, ret-code.

**STRUCTURE ELEMENTS**

- **first-line**
  - is the line number where the window is to start. (Input)

- **height**
  - the number of lines used by the window, i.e., its height. (Input)
switch-name
  the name that the caller wishes to associate with the switch. (Input)

window-id
  the returned id of the window just created. (Output) It must not be altered in any way by the application program.

ret-code
  return code. (Output) (See Appendix B.)

Entry: cb_window_$destroy

Used to destroy a previously created window. (If usage-mode = 0 (see cb_menu_$init2) this entry point should not be called.)

USAGE

declarations:

77 window-id USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.

call "cb_window_$destroy" USING window-id, ret-code.

STRUCTURE ELEMENTS

window-id
  window identifier (returned by the cb_window_$create). (Input/Output) It is reset to an illegal value by this call.

ret-code
  return code. (Output) (See Appendix B.)

COBOL MENU APPLICATION EXAMPLES

In the following two COBOL examples, a "Message" menu application is created that allows you to display, print, discard, or forward messages. Example 1 is a simple COBOL program that interfaces with the Multics menu manager via the cb_menu_ routine. Note in example 1 that window management functions are called automatically through arguments in the ft_menu_$init2 subroutine.

Example 2 is a COBOL program that interfaces with the Multics menu manager through the cb_menu_routine; in example 2, however, window management functions are performed by the cb_window_ routine.
EXAMPLE 1:

In this example, all window management is done automatically.

```
/**********************************************
  * A simple COBOL program interfacing with the Multics *
  * menu manager via the cb_menu_routine.   *
  **********************************************
CONTROL DIVISION.
DEFAULT GENERATE AGGREGATE DESCRIPTORS.
IDENTIFICATION DIVISION.

PROGRAM-ID.
  cbtestl.

AUTHOR.
  R. I.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
  Multics.

OBJECT-COMPUTER.
  Multics.

DATA DIVISION.
WORKING-STORAGE SECTION.

  01 choices-table.
  02 choices PIC X(15) OCCURS 6 TIMES.

  01 headers-table.
  02 headers PIC X(14) OCCURS 1 TIMES.

  01 trailers-table.
  02 trailers PIC X(32) OCCURS 1 TIMES.

  01 keys-table.
  02 keys PIC X(1) OCCURS 6 TIMES.

  01 menu-format.
  02 menu-version USAGE IS COMP-6 VALUE 1.
  02 constraints USAGE IS COMP-6.
  03 max-width VALUE 79.
  03 max-height VALUE 10.
  02 no-of-columns USAGE IS COMP-6 VALUE 2.
  02 flags.
  03 center-headers PIC 9(1) VALUE 1.
  03 center-trailer PIC 9(1) VALUE 1.
```
PROCEDURE DIVISION.
* The call to the cv_error_$name are used to collect the code for
* certain error messages that are of interest this application.
* Once these codes are retrieved the occurrence of that error can
* be easily tested for.

START-IT.
CALL "cb_menu_$init1".
CALL "cb_menu_$init2" USING easy-mode, user-window-lines,
- user-window-columns, user-window-id, ret-code.

* The calls to cb_menu_$init1 & 2 MUST be the first calls to cb_menu_.
* They set up the appropriate environment for the menu application.

IF ret-code EQUAL TO zero GO TO NEXT-ERR-CODE.
CALL "com_err_" USING ret-code, me, "Internal error. Could not set up appropriate environment.".
GO TO STOP-IT.

CALL "cv_error_$name" USING "menu_et_Skeys_not_unique",
- keys-not-unique, ret-code.
call "ioa_" USING "Error code for keys-not-unique = "d", keys-not-unique.
IF ret-code EQUAL TO zero GO TO NEXT-ERR-CODE.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.

NEXT-ERR-CODE.
CALL "cv_error_$name" USING "error_table_$bad_arg", bad-arg, ret-code.
IF ret-code EQUAL TO zero GO TO LAST-ERR-CODE.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.

LAST-ERR-CODE.
CALL "cv_error_$name" USING "menu_et_$too_few_keys", too-few-keys,
   ret-code.
IF ret-code EQUAL TO zero GO TO SET-UP.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.

SET-UP.
   MOVE 1 TO menu-version.
   MOVE "Display Message" TO choices(1).
   MOVE "Print Message" TO choices(2).
   MOVE "Discard Message" TO choices(3).
       MOVE "Forward Message" TO choices(4).
   MOVE "Reply Message" TO choices(5).

   MOVE "List Messages" TO choices(6).
   MOVE "MULTICS MAIL" TO headers(1).
   MOVE "Press F1 or enter esc-q to quit" TO trailers(1).
   MOVE "1" TO keys(1).
       MOVE "2" TO keys(2).
   MOVE "3" TO keys(3).
       MOVE "4" TO keys(4).
   MOVE "5" TO keys(5).
   MOVE "6" TO keys(6).

MENU-CREATE.
   DISPLAY choices-table.
   DISPLAY menu-version.
   CALL "cb_menu_$create" USING choices-table, headers-table,
   trailers-table, menu-format, keys-table, menu-needs,
   menu-id, ret-code.

   * This call creates a menu object and return the menu object
   * identifier. This menu object is referenced as "menu-id".
   IF ret-code EQUAL TO zero GO TO STORE-MENU.
   CALL "com_err_" USING ret-code, me, " (calling cb_menu_$create)".
GO TO STOP-IT.

STORE-MENU.
   MOVE ">udd>m>ri" TO dir-name.
   MOVE "menus_seg" TO entry-name.
   MOVE "cb_read_mail_menu" TO menu-name.
       MOVE 1 TO create-seg.
CALL "cb_menu_$store" USING dir-name, entry-name, menu-name,
    - create-seg, menu-id, ret-code.
IF ret-code EQUAL TO zero GO TO DISPLAY-MENU.
CALL "com_err" USING ret-code, me, "(calling cb_menu_$store)".
GO TO STOP-IT.
DISPLAY-MENU.
CALL "cb_menu_$display" USING window-id, menu-id, ret-code.

* This call displays the menu in its own window at top of screen.
* Since the usage-mode was set to 0, the program does not have to
  * create the window before calling cb_menu_$display.
* The window-id argument is ignored.
IF ret-code EQUAL TO zero GO TO GET-CHOICE.
CALL "com_err" USING ret-code, me, "Internal error.
    Menu could not be displayed."
GO TO STOP-IT.
GET-CHOICE.
    *
    * Defines the function key requirements, i.e.,
    * if the terminal has function key 1 (F1) then F1 will be used
    * to "quit", otherwise "esc q" will be used to "quit".
CALL "cb_menu_$get_choice" USING window-id, menu-id,
    - function-key-info, fkeys, option, ret-code.
IF ret-code EQUAL TO zero GO TO TEST-FKEY.
CALL "com_err" USING ret-code, me, "Internal error. While getting
    user's choice."
GO TO STOP-IT.
TEST-FKEY.
IF fkeys EQUAL TO 1
    CALL "ioa_1" USING "Exiting at your request."
    GO TO STOP-IT
ELSE
    CALL "ioa_1" USING "You chose option ^d.", option
    GO TO GET-CHOICE.
STOP-IT.
CALL "cb_menu_$terminate".
    * cb_menu_$terminate MUST be the last call to cb_menu_ in the
    * application. It terminates the environment set up cb_menu_$init.
    EXIT PROGRAM.
EXAMPLE 2:

In this example, COBOL interfaces with the Multics menu manager and the Multics window manager via the cb_menu_ and cb_window_ subroutines.

****************************************************************************
* A simple COBOL program interfacing with the Multics *
* menu manager and window manager via the cb_menu_ and *
* cb_window_ routines, respectively. *
****************************************************************************

CONTROL DIVISION.
DEFAULT GENERATE AGGREGATE DESCRIPTORS.
IDENTIFICATION DIVISION.

PROGRAM-ID.
cbtest2.

AUTHOR.
R. I.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
Multics.
OBJECT-COMPUTER.
Multics.

****************************************************************************

DATA DIVISION.

WORKING-STORAGE SECTION.

01 choices-table1.
   02 choices1 PIC X(9) OCCURS 2 TIMES.
01 choices-table2.
   02 choices2 PIC X(15) OCCURS 6 TIMES.
01 choices-table3.
   02 choices3 PIC X(21) OCCURS 4 TIMES.
01 headers-table.
   02 headers PIC X(23) OCCURS 1 TIMES.
01 trailers-table.
   02 trailers PIC X(52) OCCURS 1 TIMES.
01 keys-table.
   02 keys PIC X(1) OCCURS 6 TIMES.

01 menu-format.
   02 menu-version USAGE IS COMP-6 VALUE 1.
   02 constraints USAGE IS COMP-6.
   03 max-width VALUE 80.
03 max-height VALUE 10.
02 no-of-columns USAGE IS COMP-6 VALUE 2.
02 flags.
  03 center-headers PIC 9(1) VALUE 1.
  03 center-trailer PIC 9(1) VALUE 1.
  02 padder PIC X(1) VALUE ".".

  01 menu-needs1 USAGE IS COMP-6.
  02 lines-needed1.
  02 width-needed1.
  02 no-of-options1.

  01 menu-needs2 USAGE IS COMP-6.
  02 lines-needed2.
  02 width-needed2.
  02 no-of-options2.

  01 menu-needs3 USAGE IS COMP-6.
  02 lines-needed3.
  02 width-needed3.
  02 no-of-options3.

77 dir-name PIC X(168).
77 entry-name PIC X(32).
77 menu-name PIC X(32).
77 function-key-info PIC X(2) VALUE "qf".
77 me PIC X(7) VALUE "cbtest2".
77 switch-name PIC X(32).

77 lines-needed USAGE IS COMP-6.
77 first-line USAGE IS COMP-6.

77 height USAGE IS COMP-6.
77 menu-id USAGE IS COMP-6.
77 menu-id1 USAGE IS COMP-6.
77 menu-id2 USAGE IS COMP-6.
77 menu-id3 USAGE IS COMP-6.
77 ret-code USAGE IS COMP-6.
77 curr-window-id USAGE IS COMP-6.
77 window-id USAGE IS COMP-6.
77 window-id1 USAGE IS COMP-6.
77 window-id2 USAGE IS COMP-6.
77 fkeys USAGE IS COMP-6.
77 option USAGE IS COMP-6.
77 do-it-yourself USAGE IS COMP-6 VALUE 1.
77 user-window-lines USAGE IS COMP-6.
77 user-window-columns USAGE IS COMP-6.
77 user-window-id USAGE IS COMP-6.
77 create-seg USAGE IS COMP-6.
77 bad-window-id USAGE IS COMP-6.
PROCEDURE DIVISION.

* The call to the cv_error_$name are used to collect the code for
* certain error messages that are of interest this application.
* Once these codes are retrieved the occurrence of that error can
* be easily tested for.

START-IT.
CALL "cv_error_$name" USING "video_et_$bad_window_id",
    - bad-window-id, ret-code.
IF ret-code EQUAL TO zero GO TO NEXT-ERR-CODE.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.

NEXT-ERR-CODE.
CALL "cv_error_$name" USING "video_et_$nonexistent_window",
    - nonexistent-window, ret-code.
IF ret-code EQUAL TO zero GO TO LAST-ERR-CODE.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.

LAST-ERR-CODE.
CALL "cv_error_$name" USING "video_et_$insuff_room_for_window",
    - insuff-room-for-window, ret-code.
IF ret-code EQUAL TO zero GO TO SET-UP.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.

SET-UP.
MOVE "Read Mail" TO choices1(1).
MOVE "Send Mail" TO choices1(2).
MOVE "Display Message" TO choices2(1).
    MOVE "Print Message" TO choices2(2).
MOVE "Discard Message" TO choices2(3).
MOVE "Forward Message" TO choices2(4).
    MOVE "Reply Message" TO choices2(5).
    MOVE "List Messages" TO choices2(6).
MOVE "Send New Message" TO choices3(1).
MOVE "Send Deferred Message" TO choices3(2).
    MOVE "Print Sent Message" TO choices3(3).
    MOVE "Save Sent Message" TO choices3(4).

MOVE "1" TO keys(1).
    MOVE "2" TO keys(2).
MOVE "3" TO keys(3).
    MOVE "4" TO keys(4).
cb_window_ cb_window_

MOVE "5" TO keys(5).
MOVE "6" TO keys(6).

CALL "cb_menu_Sinit1".
CALL "cb_menu_Sinit2" USING do-it-yourself, user-window-lines,
   user-window-columns, user-window-id, ret-code.

* The call to cb_menu_Sinit1 & 2 MUST be the first call to cb_menu_.
* It sets up the appropriate environment for the menu application.
* The application must do the window management, since
* "do-it-yourself" is set to 1.

IF ret-code EQUAL TO zero GO TO CREATE-FIRST-MENU.
CALL "com_err_" USING ret-code, me, "Internal error. Could not set up
   appropriate environment."
GO TO STOP-IT.

CREATE-FIRST-MENU.

* Create first menu object.

MOVE "F1 (or esc-q) = quit" TO trailers(1).
MOVE "MULTICS MAIL" TO headers(1).
CALL "cb_menu_Screate" USING choices-table1, headers-table,
   trailers-table, menu-format, keys-table, menu-needs1,
   menu-id1, ret-code.

IF ret-code EQUAL TO zero GO TO CREATE-SECOND-MENU.
CALL "com_err_" USING ret-code, me, " (calling cb_menu_Screate)".
GO TO STOP-IT.

CREATE-SECOND-MENU.

* Create second menu object.

MOVE "F1 (or esc-q) = quit; F2 (or esc-f) = first menu" TO trailers(1).
MOVE "READ MAIL" TO headers(1).
CALL "cb_menu_Screate" USING choices-table2, headers-table,
   trailers-table, menu-format, keys-table, menu-needs2,
   menu-id2, ret-code.
IF ret-code EQUAL TO zero GO TO CREATE-THIRD-MENU.
CALL "com_err_" USING ret-code, me, " (calling cb_menu_Screate)".
GO TO STOP-IT.

CREATE-THIRD-MENU.

* Create third menu object.

MOVE "SEND MAIL" TO headers(1).
CALL "cb_menu_Screate" USING choices-table3, headers-table,
   trailers-table, menu-format, keys-table, menu-needs3,
   menu-id3, ret-code.
IF ret-code EQUAL TO zero GO TO STORE-MENU.
CALL "com_err_" USING ret-code, me, "(calling cb_menu_$create)".
GO TO STOP-IT.

STORE-MENU.
MOVE ">udd>m>ri" TO dir-name.
MOVE "menu_seg" TO entry-name.
MOVE "cb_test_menu_" TO menu-name.
MOVE 1 TO create-seg.
CALL "cb_menu_$store" USING dir-name, entry-name, menu-name,
  - create-seg, menu-idl, ret-code.
IF ret-code EQUAL TO zero GO TO DISPLAY-IT.
CALL "com_err_" USING ret-code, me, "(calling cb_menu_$store)".
GO TO STOP-IT.

DISPLAY-IT.
MOVE -1 TO curr-window-id.
  * Setting curr-window-id to "-1" means that there is no current window
  * defined.
MOVE menu-idl TO menu-id.
MOVE lines-neededl TO lines-needed.

DISPLAY-FIRST-MENU.
PERFORM CHANGE-MENU THRU GOBACK.
  * The user i/o window has been "shrunk", the window for the first menu
  * has been created, and the first menu has been displayed.
  MOVE window-id TO window-idl.
IF ret-code EQUAL TO zero GO TO GET-IT.
CALL "com_err_" USING ret-code, me, "Internal error.
  Menu could not be displayed.".
GO TO STOP-IT.

GET-IT.
PERFORM GET-CHOICE.
  * Get the user input. Two values are returned. (1) fkey. If fkey = 1,
  * then the user entered a function key (or its equivalent escape
  * sequence). If fkey = 0 then the user has selected an option. (2) option.
  * If fkey = 1 then option is the function key number entered. (F1 = 1,
  * F2 = 2, etc.). If fkey = 0, then option is the option number selected,
  * option = 1 means option 1 selected, etc.
IF ret-code EQUAL TO zero GO TO TEST-FKEY.
CALL "com_err_" USING ret-code, me, "Internal error.
  While getting user's choice.".
GO TO STOP-IT.

TEST-FKEY.
IF fkeys EQUAL TO 1
  IF option EQUAL TO 1
    CALL "ioa_" USING "Exiting at your request."
    GO TO STOP-IT
  ELSE
    GO TO GET-IT

9-25 CP51-02
ELSE
  IF option EQUAL TO 1
    MOVE menu-id2 TO menu-id
    MOVE lines-needed2 TO lines-needed
    PERFORM CHANGE-MENU THRU GOBACK
  ELSE
    MOVE menu-id3 TO menu-id
    MOVE lines-needed3 TO lines-needed
    PERFORM CHANGE-MENU THRU GOBACK.
  IF ret-code NOT EQUAL TO zero
    CALL "com_err_" USING ret-code, me, "Internal error.
    While trying to display menu."
    GO TO STOP-IT
  ELSE
    MOVE window-id TO window-id2.
    NEXT-GET-IT.
    PERFORM GET-CHOICE.
    IF fkeys EQUAL TO zero GO TO CHOSE-OPTION.
    IF option EQUAL TO 1
      CALL "ioa_" USING "Exiting at your request."
      GO TO STOP-IT
    ELSE
      IF option GREATER 2
        GO TO NEXT-GET-IT
      ELSE
        MOVE menu-id1 TO menu-id
        MOVE lines-needed1 TO lines-needed
        GO TO DISPLAY-FIRST-MENU.
        CHOSE-OPTION.
        CALL "ioa_" USING "You chose option ^d.", option.
        GO TO NEXT-GET-IT.
        GET-CHOICE.
        CALL "cb_menu$_get_choice" USING window-id, menu-id,
          function-key-info, fkeys, option, ret-code.
    END-IF
    CHANGE-MENU.
    * Destroy the current menu window.
    IF (curr-window-id ) EQUAL TO -1 GO TO CHANGE-USER-WIND.
    CALL "cb_window$_destroy" USING curr-window-id, ret-code.
    IF ret-code EQUAL TO zero GO TO CHANGE-USER-WIND.
    GO TO GOBACK.
  END-ELSE
  CHANGE-USER-WIND.
  COMPUTE first-line = lines-needed + 1.
  COMPUTE height = user-window-lines - lines-needed.
  CALL "cb_window$_change" USING user-window-id, first-line, height,
    ret-code.
  IF ret-code EQUAL TO zero GO TO CREATE-NEW-WIND
  ELSE GO TO GOBACK.
  CREATE-NEW-WIND.
  MOVE "menu-window" TO switch-name.
MOVE 1 TO first-line.
CALL "cb_window_$create" USING first-line, lines-needed,
  switch-name, window-id, ret-code.
IF ret-code EQUAL TO zero GO TO DISPLAY-MENU
ELSE GO TO GOBACK.
  DISPLAY-MENU.
  MOVE window-id TO curr-window-id.
CALL "cb_menu_$display" USING window-id, menu-id, ret-code.
  CALL "cb_window_$clear_window" USING user-window-id, ret-code.
GOBACK.
EXIT.

STOP-IT.
CALL "cb_menu_$terminate".
  * cb_menu_$terminate MUST be the last call to cb_menu_ in the
  * application. It terminates the environment set up cb_menu_$init.
  
  EXIT PROGRAM.
This appendix reviews the standard I/O switch attachments, then describes how these attachments change when you activate the video system on your terminal and create a menu.

There are four standard switches which are attached when your process is created. These switches are as follows:

1. user_i/o: this switch acts as a common collecting point for all terminal I/O. It's attached to your terminal through the I/O module tty_, and is opened for stream input and output.

2. user_input: this switch controls command and data input at your terminal. It's attached to user_i/o through the I/O module syn_, and through that to your terminal. It's opened for stream input.

3. user_output: this switch controls command and data output at your terminal. It's attached to user_i/o through the I/O module syn_, and through that to your terminal. It's opened for stream output.

4. error_output: this switch controls output of error messages at your terminal. It's attached to user_i/o through the I/O module syn_, and through that to your terminal. It's opened for stream output.

To get information about I/O switch attachments, you can use the print_attach_table (pat) command. If you type "pat" on your terminal right after you log in, the system will print the following:

```
error_output       syn_user_i/o -inh close get_line get_chars
user_input         syn_user_i/o -inh close put_chars
user_i/o           tty_ -login_channel
                 stream_input_output
user_output        syn_user_i/o -inh close get_line get_chars
```
You can see from this that user_input, user_output, and error_output are all attached via syn_ to user_i/o, which in turn is attached via tty_ to your terminal. Figure A-1 illustrates these standard I/O switch attachments.

When you activate the video system, by issuing a call to video_utils_$turn_on_login_channel or by executing the window_call invoke command, the existing tty_ attachment of your terminal is removed and replaced with video system attachments. The I/O switch user_i/o is now attached through the I/O module window_io_ to a new I/O switch, user_terminal_. User_terminal_ is attached through the I/O module tc_io_ to your terminal.
Figure A-1. Standard Attachments
If you type "pat" on your terminal after invoking video, the system will print the following:

```
user_terminal  tc_io_ -login_channel
  stream_input_output
error_output  syn_ user_i/o -inh close get_line get_chars
user_input    syn_ user_i/o -inh close put_chars
user_i/o      window_io_ user_terminal_ -first_line 1 -n_lines 24
  stream_input_output Video
user_output   syn_ user_i/o -inh close get_line get_chars
```

You can see from this that user_input, user_output, and error_output are still attached via syn_ to user_i/o, but that user_i/o is now attached via window_io_ to user_terminal_, which is in turn attached via tc_io_ to your terminal. User_i/o is now a window as well as a switch. It begins on line 1 of your screen and is 24 lines long. On a VIP7801 terminal, this means that the window covers the entire screen. Figure A-2 illustrates these changes to the standard I/O switch attachments.

When you execute an exec_com to create a menu, the necessary attachments for your menu are built on top of those already set up by your activation of the video system. If you run the exec_com discussed in Section 3 (doc_sys.ec), then type "pat" on your terminal, the system will print the following:

```
user_terminal  tc_io_ -login_channel
  stream_input_output
error_output  syn_ user_i/o -inh close get_line get_chars
user_input    syn_ user_i/o -inh close put_chars
user_i/o      window_io_ user_terminal_ -first_line 8 -n_lines 17
  stream_input_output Video
user_output   syn_ user_i/o -inh close get_line get_chars
  !BBBJLXdqDbMNnn.menu
  window_io_ user_terminal_ -first_line 1 -n_lines 7
  stream_input_output Video
  811007144650.613707.exec_com
  ec_input_ ">`udd>Project>Person>doc_sys.ec" stream_input
```
You can see from this that user_input, user_output, and error_output are still attached via syn_ to user_i/o, that user_i/o is still attached via window_io_ to user_terminal_, and that user_terminal_ is still attached via tc_ic_ to your terminal. But in addition, the !BBBJLXDqDbMNnn.menu is now attached through window_io_ to user_terminal_ also. (The unique character string "!BBBJLXDqDbMNnn" is generated by using the unique active function, as in the construction [unique].menu, used in doc_sys.ec.) The user_i/o window still begins on line 1, but now it is only 7 lines long. The !BBBJLXDqDbMNnn.menu, which, like user_i/o, is a window as well as a switch, begins on line 8, and is 17 lines long.

The last two lines printed above provide information about attachments made to support the execution of the exec_com. They are of no concern to you in this discussion. Figure A-3 illustrates I/O switch attachments after the video system has been activated and an exec_com creating a menu has been run.

For more information on the print_attach_table command and the unique active function, refer to the Multics Commands and Active Functions manual, Order No. AG92.
Figure A-2. Attachments After the Invocation of Video
Figure A-3. Attachments After Execution of doc_sys.ec exec_com
The subroutine cv_error_$name is provided in order to allow the FORTRAN and COBOL programmer to test return codes in a way similar to that provided by PL/I.

It provides a means to associate an error "name", e.g., "menu_et$_too_few_keys" with the numeric value of the returned code. Once this is done the programmer can test for a given error code by using the name associated with it.

**SYNTAX IN FORTRAN**

call cv_error_$name (error_name, converted_code, code)

**SYNTAX IN COBOL**

CALL "cv_error_$name" USING error-name, converted-code, ret-code.

**ARGUMENTS**

- error_name (error-name)
  - a quoted string, e.g., "menu_et$_too_many_options", which is name of the error. (Input)

- converted_string (converted-string)
  - an integer (USAGE IS COMP-6 in COBOL) variable where the returned numeric value of the code is to be stored. (Output)

- code (FORTRAN)
  - 0 if call was successful, nonzero otherwise. (integer) (Output)

- ret-code (COBOL)
  - 0 if call was successful, nonzero otherwise. (USAGE IS COMP-6) (Output)

**NOTES**

"code" must be declared as "integer" in a FORTRAN program and "ret-code" as USAGE IS COMP-6 in a COBOL program. In every call, to any entry point defined in this document, a return code of zero always means that the call was executed successfully.)
Error codes of particular interest are:

**menu_et$_too_many_options**
(A menu can contain at most 61 choices.)

**menu_et$_too_few_keys**
(There are fewer keys than choices.)

**menu_et$_keys_not_unique**
(Each key must be unique.)

**menu_et$_higher_than_max**
(The menu will not fit within the specified maximum height.)

**video_et$_bad_window_id**
(The supplied window id was not valid.)

**video_et$_overlapping_windows**
(Two windows may not overlap on the screen.)

**video_et$_window_too_big** (The screen is too small to accommodate a window of the requested size.)

**video_et$_insuff_room_for_window**
(Insufficient room to create window.)

**video_et$_window_too_small**
(Tried to adjust window past minimum size.)

**video_et$_negative_screen_size**
(Negative screen size specified.)

**video_et$_negative_window_size**
(Negative window size specified.)

**video_et$_nonexistent_window**
(Specified window does not exist.)

**video_et$_overlaps_other_window**
(Specified window overlaps other windows.)

**video_et$_unable_to_create_window**
(Unable to create window.)

**video_et$_unable_to_dest_window**
(Unable to destroy window.)

**video_et$_switch_not_window**
(The specified switch is not attached as a window.)

**error_table$_no_operation**
(Cannot call this entry point in your current mode. Requested operation could not be performed.)
<table>
<thead>
<tr>
<th>MISCELLANEOUS</th>
<th>arguments for window_call</th>
</tr>
</thead>
<tbody>
<tr>
<td>^A 4-5</td>
<td>delete_chars 5-14</td>
</tr>
<tr>
<td>^B 4-4</td>
<td>delete_window 2-12, 5-14</td>
</tr>
<tr>
<td>^D 4-5</td>
<td>get_first_line 5-15</td>
</tr>
<tr>
<td>^E 4-5</td>
<td>get_one_unechoed_char 5-15</td>
</tr>
<tr>
<td>^F 4-4</td>
<td>get_position 5-15</td>
</tr>
<tr>
<td>^L 4-4</td>
<td>get_terminal_height 5-16</td>
</tr>
<tr>
<td>^Q 4-4</td>
<td>get_terminal_width 5-16</td>
</tr>
<tr>
<td>^Y 4-3</td>
<td>get_unechoed_chars 5-16</td>
</tr>
<tr>
<td># 4-5</td>
<td>get_window_height 5-17</td>
</tr>
<tr>
<td>A</td>
<td>insert_text 5-17</td>
</tr>
<tr>
<td></td>
<td>invoke 2-9, 5-17</td>
</tr>
<tr>
<td></td>
<td>overwrite_text 5-18</td>
</tr>
<tr>
<td></td>
<td>revoke 5-18</td>
</tr>
<tr>
<td></td>
<td>scroll_region 5-18</td>
</tr>
<tr>
<td></td>
<td>set_position 5-19</td>
</tr>
<tr>
<td></td>
<td>set_position_rel 5-19</td>
</tr>
<tr>
<td></td>
<td>supported_terminal 5-19</td>
</tr>
<tr>
<td></td>
<td>sync 5-19</td>
</tr>
<tr>
<td></td>
<td>video_invoked 5-20</td>
</tr>
<tr>
<td></td>
<td>write_sync_read 5-20</td>
</tr>
</tbody>
</table>

arguments for window_call (cont)

attaching video system 2-5

attachments

review of

see also video attachments

see standard attachments

i-1

CP51-02
backspace key 4-2

backward character
    ^B 4-4

backward word
    ESC B 4-5

beginning line
    ^A 4-5

capitalize initial word
    ESC C 4-5

capitalize word
    ESC U 4-5

clear and redisplay
    ESC ^L 4-4

delete character
    ^D 4-5

delete word
    ESC D 4-5

end of line
    ^E 4-5

erase 4-2
    backspace key 4-2

DEL, # 4-5

delete word
    ESC DEL, ESC # 4-5

forward character
    ^F 4-4

forward word
    ESC F 4-4

kill 4-2

lower case word
    ESC L 4-5

quoting character
    ^Q 4-4

real-time editor 4-1

redisplay
    ^L 4-4

retrieving deleted text
    ESC Y 4-3

^Y 4-3

twiddle word
    ESC T 4-5

two characters 4-2

deleting words 4-2

retrieving deleted text 4-3

clear window example 2-13

create window example 2-10
data structures
  menu_format 6-8
  menu_list_info 6-9
  menu_requirements 6-11
DEL 4-5
delete character
  ^D 4-5
delete word
  ESC D 4-5
delete_window example 2-12
deleting words 4-2
detaching video system 2-7

del
end of line
  ^E 4-5
end of window processing 4-10
erase character 4-2
  # 4-5
  backspace key 4-2
  DEL 4-5
erase word
  ESC DEL, ESC # 4-5
error_output switch A-1
ESC # 4-5
ESC B 4-5
ESC C 4-5
ESC D 4-5
ESC DEL 4-5
ESC F 4-4
ESC L 4-4, 4-5
ESC T 4-5
ESC U 4-5
ESC Y 4-3

examples
exec_com
  attaching video 2-5
  clear_window 2-13
  create_window 2-10
  delete_window 2-12
  detaching video 2-7
  document system 3-2
  function keys 3-1
  function keys alternative 3-1
pl1
  attaching video 2-6
  detaching video 2-7
  document system 3-6
  window$clear_window 2-13
  window$create 2-11
  window$destroy 2-12

extensions
writing editor 4-5

f
forward character
  ^F 4-4
forward word
  ESC F 4-4
ft_menu_ 8-2
  ft_menu$create 8-2
  ft_menu$delete 8-4
  ft_menu$describe 8-4
  ft_menu$destroy 8-5

i-3
ft_menu_ (cont)

ft_menu_$display 8-6
ft_menu_$get_choice 8-6
ft_menu_$init1 8-8
ft_menu_$init2 8-8
ft_menu_$retrieve 8-10
ft_menu_$store 8-10
ft_menu_$terminate 8-11

ft_window_ 8-13
ft_window_$change 8-13
ft_window_$clear_window 8-14
ft_window_$create 8-14
ft_window_$destroy 8-15

function keys
alternatives 3-2
guidelines for 3-1
recommendations 3-2

I/O modules
tc_io_ 7-2
window_io_ 7-5

kill character 4-2
kill ring 4-3

lower case word
ESC L 4-5
menu_list_info data structure 6-9

menu_requirements data structure 6-11

miscellaneous capabilities in windows 2-4

MORE processing 4-1, 4-10

operations on windows 2-8
  change_window 2-12
  clear_window 2-13
  create_window 2-9
  set_window_info 2-12

output buffering 4-11

output control 4-1

overlap rule for windows 2-8

positioning the cursor in windows 2-3

print_attach_table description
  see the Commands manual

quoting character
  ^Q 4-4

real-time editor 4-1

control characters 4-1

deleting words 4-2

erase and kill values 4-1

erase character 4-2

kill character 4-2

retrieving deleted text 4-3

redisplay
  ^L 4-4

requirements for windows 2-8

retrieving deleted text 4-3

ESC Y 4-3

^Y 4-3

routines
  line editor 4-6

scrolling in windows 2-4

selective alteration in windows 2-4

selective erasure in windows 2-4

standard attachments A-1
  after invoking video A-4
  illustration of A-3
  via syn_ A-2
  via tty_ A-2

standard 1/O switch A-1

standard 1/O switch attachments
  see also standard attachments
standard switch names
error_output A-1
user_input A-1
user_io switch A-1
user_output A-1

switch attachments
see also video attachments
see standard attachments

syn_
see the Subroutines manual

T
tc_io_ 7-2
attach description 7-2
control operations
clear_screen 7-3
get_break_table 7-3
get_capabilities 7-3
reconnection 7-4
set_break_table 7-3
set_line_speed 7-3
set_term_type 7-4
get_line_operation 7-3
open_operation 7-3

trailer lines 2-8

tty_
see the Subroutines manual

twiddle word
ESC T 4-5

U
unique active function
see the Commands manual

user_input switch A-1
user_io switch A-1

user_io window 2-5, 2-9, 2-12, 2-13, 3-2
size of 2-5

user_output switch A-1

utilities
window editor 4-8
video subroutines (cont)
  window $clear_window example 2-13
  window $create 6-19
  window $create example 2-11
  window $delete_chars 6-20
  window $destroy 6-21
  window $destroy example 2-12
  window $edit_line 6-21
  window $get_cursor_position 6-22
  window $get_echoed_chars 6-23
  window $get_unechoed_chars 6-25
  window $insert_text 6-26
  window $overwrite_text 6-26
  window $position_cursor 6-27
  window $position_cursor_rel 6-27
  window $scroll_region 6-28
  window $sync 6-29
  window $write_raw_text 6-29

video system
  attaching 2-5
  command interface 4-11
  detaching 2-7
  features 4-1
  end of window processing 4-10
  MORE processing 4-1, 4-10
  output control 4-1
  real-time editing 4-1
  windows 2-1
  subroutine interface 4-11

window
  window editor_utils $ get_top_killer_ring_element 4-9
  insert_text 4-8
  move_backward_word 4-9
  move_forward 4-9
  move_forward_word 4-9
  rotate_killer_ring 4-10

windows
  definition 2-1
  height of 2-9
  miscellaneous capabilities 2-4
  naming of 2-9
  number permitted 2-8
  operations 2-1, 2-8
  change_window 2-12
  clear_window 2-13
  create_window 2-9
  set_window_info 2-12
  overlap rule 2-8
  positioning cursor 2-3
  requirements 2-8
  scrolling 2-4
  selective alteration 2-4
  selective erasure 2-4
  trailer lines 2-8
  width of 2-8

window_ 4-11, 6-15
  clear_to_end_of_window 6-18
  get_one_unechoed_char 6-24
  window $bell 6-15
  window $change_column 6-16
  window $change_line 6-16
  window $clear_region 6-17
  window $clear_to_end_of_line 6-18
  window $clear_window 6-19
  example 2-13
  window $create 6-19
  example 2-11
  window $delete_chars 6-20
  window $destroy 6-21
window_ (cont)
  window_$destroy
    example 2-12
  window_$edit_line 6-21
  window_$get_cursor_position 6-22
  window_$get_echoed_chars 6-23
  window_$get_unechoed_chars 6-25
  window_$insert_text 6-26
  window_$overwrite_text 6-26
  window_$position_cursor 6-27
  window_$position_cursor_rel 6-27
  window_$scroll_region 6-28
  window_$sync 6-29
  window_$write_raw_text 6-29
  window_$write_sync_read 6-30

window_call 4-11
  window_call arguments
    see arguments for
      window_call

window_call command 5-10

window_io_ 7-5
  attach description 7-5
  control operations
    get_break_table 7-15
    get_capabilities 7-9
    get_editing_chars 7-10
    get_more_responses 7-11
    get_output_conversion 7-16
    get_special 7-16
    get_token_characters 7-20
    get_window_info 7-7
    get_window_status 7-8
    reset_more 7-10
    set_break_table 7-15
    set_editing_chars 7-10
    set_more_responses 7-11
    set_output_conversion 7-16

  control operations
    set_special 7-17
    set_token_characters 7-20
    set_window_info 7-7
    set_window_status 7-8

  control operations from
    command level 7-26
  get_chars_operation 7-6
  get_line_operation 7-6

  modes operations
    can, ^can 7-25
    ctl_char, ^ctl_char 7-25
    erkl, ^erkl 7-25
    esc, ^esc 7-25
    il 7-26
    more, ^more 7-24
    more_mode 7-24
    pl 7-26
    rawi, ^rawi 7-26
    rawo, ^rawo 7-25
    red, ^red 7-26
    vertsp, ^vertsp 7-25
  open_operation 7-6
  put_chars_operation 7-6
TITLE: MULTICS MENU CREATION FACILITIES

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here.

FROM: NAME

TITLE

COMPANY

ADDRESS

DATE

ORDER NO.: CP51-02

DATED: FEBRUARY 1985
PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154
POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486
Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 456, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW9 0DH Italy: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. Japan: 2-2 Kanda Jimbo-cho, Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 S.E. Asia: Mandarin Plaza, Tsimshatsui East, H.K.
42356, 7.5C385, Printed in U.S.A.

CP51-02