
Program Documentation

by

Harry W. Morris III

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Science in Computer Science and Engineering
at the
Massachusetts Institute of Technology
May 1987

© Harry W. Morris III 1987

The author hereby grants to M.I.T. permission to reproduce and to
distribute
copies of this thesis document in whole or in part.

Signature of Author

Department of Electrical Engineering and Computer Science
May 18, 1987

Certified by

Michael J. Markow
Thesis Supervisor

Accepted by

Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses
Abstract:

Rational management decisions require the juggling of numerous variables and relationships. Searching for viable solutions is a time intensive process that requires expert attention. The Facility Management System (FMS) addresses these problems by providing a flexible simulation environment for life cycle cost analysis.
FACILITY MANAGEMENT SYSTEM

Program Documentation

Release 1.0

6-18-1987

Prepared by Harry W. Morris III

for the department of Civil Engineering
the Massachusetts Institute of Technology
Cambridge, MA. 02159
Dedication:

This work is dedicated to my mother, Bernhilt Sophie Merckelbach Morris.
CONTENTS

1.0 Introduction
2.0 DISPLAY
3.0 FORMS
4.0 UTILITIES
5.0 Facility Management System

Appendix A Computing Environment
FIGURES

2.0 A Screen Layout with Three Windows
2.1 Example Window Layout from FMS
5.1 FMS Control Structure
1.0 INTRODUCTION

Introduction to FMS:

FMS is a facility management system for civil works. It allows managers to simulate the aging of a facility under various maintenance policies and usage patterns. Future versions will support mathematical optimization routines to assist in the discovery of viable maintenance plans.

This document describes FMS from a programmer's point of view. The major program modules are described in enough detail to provide an understanding of the system without looking at the actual code. The intended audience includes system maintainers and programmers implementing the proposed extensions. Because of the system's powerful user interface libraries, other applications may be built using much of the existing code. These readers are also addressed.

FMS itself is built upon two major sub-systems. The sub-systems handle all input output functions. These functions must be understood in order to understand FMS. Because of this, the procedures making up FMS are described last.
Division Of Work:

This work is divided into four functional units. The low level input and output procedures are contained in DISPLAY. DISPLAY also contains the windowing mechanism. FORMS contains a flexible high level input mechanism. It is used to get raw data from the user and build the command structure of the program. FMS itself is an application that uses DISPLAY and FORMS. It implements a simulation of facility deterioration. UTIL provides simple support for all of the modules. This support includes string handling and memory management functions and support for the abstract data type SET.

FORMS, DISPLAY, and UTIL are independent of FMS. They are quite powerful and can be used for building any interactive application. For this reason they are described separately from one another. The only reference to FMS is in the form of examples.

Style:

In this document words in bold face are key words unique to the program. Key words include the names of commands, variables, files and program examples. They are to be interpreted literally with the exception of initial capitalization. Words in all capitals are either data types, major global data structures, or macros. Actual functions and files are never capitalized in the code, although they may be capitalized in this document if they appear at the beginning of a sentence. Function names are always
followed by an empty pair of parentheses. An underscore preceding a function name indicates that the function is internal to a particular module and should not be referenced outside that module. File names are always followed by a . (dot) and an extension of either: c which means that the file contains C source code; mac which means that the file contains macro definitions for the C pre-processor; equ which means that the file is an equates file containing C definitions of data structures; or exe which indicates that the file is an executable file which can be run from dos. Furthermore FORMS uses the extension .hlp for text files containing help messages, and FMS reserves the extensions .fac, .pol for data describing facilities and policies.

Fully capitalized words in plain text are used to refer to program concepts. These are not actual commands or variables, but ideas that give structure to the program. Examples include module names and menu choices.

Underlined words in plain text are used to draw attention to important concepts and underscore conceptual divisions.
2.0 DISPLAY

Introduction:

The DISPLAY module provides high level routines to facilitate input and output. With DISPLAY it is easy to build applications that interact with the user in logical and efficient ways. DISPLAY's key ability is to divide the screen into usable chunks. These chunks are called windows.

DISPLAY is a window manager. A window is a rectangular sub-region of the computer screen. DISPLAY commands facilitate creating windows, printing text in them, moving around in them, and destroying them.

It is useful to think of a window as a coordinate grid with its origin in the window's upper left hand corner. The X axis moves to the right and the Y axis moves down. Coordinates expressed relative to the window's origin are called window coordinates. A window's origin itself is specified in relation to the upper left hand corner of the screen. These coordinates are known as screen coordinates. Each position in the grid may contain a single character. Each window maintains a coordinate called the insertion point. The insertion point is the coordinate where the window's next character will appear. It can be moved by specifying a window and the desired X and Y coordinates (see figure 2.0).
Figure 2.0 A Screen Layout with three Windows

Commands similar to C's printf() allow the output of text to a particular window. Used in this way, windows act exactly like small...
versions of the computer's standard screen. The power of the window lies in the fact that many windows may coexist on the same screen. In this way the computer's physical screen is divided into many virtual screens.

In order to conserve screen space, it is highly desirable to overlap the windows. This allows windows containing information of lesser importance to be "buried" under windows that are currently relevant. The buried windows maintain their information and can be retrieved at any time.

Each window is completely independent of all other windows\(^1\). Each window can have its own characteristics and function. Their ability to be individually addressed.

In addition to relieving the size limitations of a physical screen, windows are a valuable aid in formatting output\(^2\). In a word processor, for example, a report in progress might be displayed in a central window, while footnotes are displayed in another window, and the current time is shown in a third window. Since each window is independently addressable, the functions of each window can be easily implemented. Even more importantly though, from the user's standpoint, the functionality of each window is distinct. Thus windows allow a

---

\(^1\) There is one exception, see "Getting Started" below.

\(^2\) As we will see in the next chapter, windows are also very valuable for formatting input.
programmer to easily present his data in a manner consistent with his program's objectives.

The remainder of this chapter will present the typical abilities and applications of windows, describe how to use them, and present a technical account of DISPLAY.

Using Windows: 

Almost every application will benefit from the use of windows. This is evidenced by the overwhelming number of window based programs now available for the PC. The reason for the popularity of window systems is clear. They provide the user with a coherent method for understanding his application program, and they allow the application programmer to generate the application code quickly, and reliably.

Essentially, the only output style available directly from 'C' is the line at a time technique held over from the . Hence, programs written strictly in 'C' and similar languages are restricted to the command line format. That is, the user requests an action, and the computer responds by typing out one or more lines. The user then examines the output, and responds. The user is not given the opportunity to review work after it has scrolled off the screen.

\[3\] doing windows, the gestalt of windows
A primitive interface like this has a serious problem. The problem is lack of control over the screen. As commands are given, the screen fills up with new data, pushing old data off the top. As a result it is often necessary to repeat commands in order to view the pertinent data. Consider how many times it is necessary to list a directory over and over, just to find a single file. With such an interface it is impossible to maintain context between commands. Since the screen display is controlled strictly by the commands most recently requested, the program (actually the programmer) has no way to maintain an appropriate context across user commands.

I will call this style of user interface a serial interface. It is serial in the sense that the user can see only the results of his last action. Execution proceeds event by event. On the other hand, a parallel interface allows arbitrary data chunks to be displayed simultaneously and independently of each other. Thus it is possible for parallel interfaces to display related data, regardless of how long ago it was requested by the user.

DISPLAY is such an interface. It uses windowing to addresses the failings of serial interfaces, as well as opening a broad spectrum of features that I will discuss latter. With windows, the application program is free to control its output. The best way to do this is to use the windows to reflect the logic of the program. In chapter 5 (FMS) we will see a typical example of the use of windows. Briefly, FMS uses two kinds of windows: command windows, and object windows. (see
figure 2.1) Command windows request input which will dramatically
effect the database or the display. A typical command window is a menu,
giving the user the option to edit, run, report, or quit. Object
windows represent the data itself. Each object window is associated
with a particular type of object in the data base. For example, a
window named FACILITY_WINDOW is used to display the gross physical data
associated with a particular facility. FACILITY_WINDOW presents the
user with data such as name, location, and size. In short, just the
information that an individual would recognize if he were to personally
inspect the facility. Associated with and available from
FACILITY_WINDOW is HIST_WINDOW. HIST_WINDOW contains the history of the
facility. Since the physical properties are essentially different from
the historical data, it makes sense to separate the information into
different parts of the screen. On the other hand, when a user asks to
see a facility, he will expect to be able to see the physical and
historical data at the same time. This is easily done by giving them
their own windows. More windows can be added to support commonly needed
functions like help messages, printing commands, and operating system
interfaces. With tools like these always at hand, it is easy for the
user to learn the software and use it effectively.
The Facility Window

The Traffic/Capacity Window

(It is created by the "edit traffic/capacity model" command in the facility window.)

facility name: DASHI
route: OHIO RIVER
mile marker: 967.70
district: ORP
division: ORD
state: OH
year open: 1929
length: 600
width: 110
height: 18
main facility: Yes
edit history:
edit deterioration model: 
edit traffic/capacity model: 

traffic growth rate: 3.00
growth years: 10.00
growth start: 4000.00
s rate zero: 2.00
s rate coefficient a: -8.00
coefficient b: -1.00
coefficient c: -1.00
s sigma zero: 0.10
average delay cost: 100.00
bottleneck proportion: 0.50
closure cost: 75.00

SELECT ONE:
- general help
- redefine keys
- exit form
- return to prev. form

The General Help Window

(This window is provided by FORMS (see chapt. 4). It is accessible from any form.)

Squares are command options.
(Pressing return when the cursor is over a square executes the command associated with that square. Note that some windows are both command and object windows.)

The Cursor

(The cursor is a flashing underline or block that highlights the active window's insertion point. The active window is the one currently accepting input.)

Figure 2.1 Example window layout from FMS.
Windows supply the user with parallel output, but how does the user supply parallel input? The answer is that he doesn't. Input is serial for two reasons. First of all, the keyboard is physically a one-at-a-time device. More importantly, input only makes sense in context. In human to human interaction, it is possible to address more than one listener as long as each listener maintains loosely the same context. Human to computer interaction is not so forgiving, since the computer requires specific input and interprets it in a strict context.

The formatting of input is left completely open. The input primitives currently supplied by DISPLAY do nothing more than read characters from the keyboard. It is up to the programmer to provide a meaningful and flexible input scheme. One such scheme, used by FMS, is supported by FORMS. FORMS is documented in the chapter 3.

**Getting Started:**

Programming with DISPLAY is simply a matter of initializing the main data structure, setting aside memory for the windows, and executing a make-write-kill cycle. The first step is to initialize DISPLAY's main data structure. This structure is used to keep track of where each window is located. The initialization is done with a call to `init_disp()`. `InitDisp()` expects an integer between 1 and 4. The argument specifies the number of pages that DISPLAY will support.
see when working with a PC. Up to four pages can be used, each holding an entire screen full of text (of course only one can be viewed at a time). Pages are valuable because it is possible, using `set_page()`, to instantly switch between pages. Such flexibility is ideal for displaying help screens, notes, graphs, etc. The cost is a modest amount of memory per page `Init_disp()` returns a pointer to the main data structure. The pointer is of type `DISPLAYP`. It is used to tell `kill_disp()` which display to kill when shutting down the program.

The second step in creating a `DISPLAY` is allocating memory for the windows. This is accomplished with the commands `declare_window()` and `declare_box()`. `Declare_window()` sets aside space for a simple window as described above. It expects three arguments: the address of a pointer which will point to the window; a maximum width for the window, and a maximum depth for the window. The address is filled with a pointer to the window's data structure (see below). In addition to allocating space for this data structure, `declare_window()` allocates space for the window's internal representation of its display. This is a contiguous chunk of memory width*(depth+1)+1 bytes long. For a full screen window the 80*41+1 = 3281 or approximately 3K-bytes is needed. The window can not be expanded beyond this size. `Declare_box()` is similar to `declare_window()` except that it requires the addresses of two pointers to windows. The first window is used as a normal text window. The second is used to provide a border around the first window. The

---

3 A proposed extension to `DISPLAY` would support the management of multiple displays.
second (border) window is internally connected to the first (main) window so that DISPLAY functions effecting the main window automatically propagate to the border window. The connection is recognized by all DISPLAY functions. In the current implementation, windows requiring border's (called boxes) require more than twice the memory of similar sized window's without borders. Where memory is a concern see "Memory Management" in chapter 5. DISPLAY is now initialized and control may return to the application program which will complete it's initialization and begin using windows!

Windows declared in the initialization step are merely pointers to blank memory. They do not appear on the screen, and have no interesting characteristics until they are brought to life with the commands make_win() or box_window(). These commands accept arguments such as the origin of the window and it's size. They fill the window's data structure with these parameters, update the page map, and take other steps necessary to put the (blank) window on the screen. It is important to note that these parameters are distinct from the memory allocated by the declarative commands. A call to make_win() or box_window() simply instantiates the window. Likewise a call to kill_win() or kill_box() invalidates the parameters and erases the window from the screen. It does not deallocate the memory. Thus it is quite possible to use the same window to edit text at one moment and to display a menu the next moment. Just remember: a window can appear on

---

4 In fact there is no mechanism provided to deallocate window memory. Again see "Memory Management" in chapter 5.
display a menu the next moment. Just remember: a window can appear on any page at any time, but no window can appear in more than one place at once - kill a window before re-creating it. This is the make-write-kill cycle.

Make_win() and box_window() require the following arguments: two integers specifying the upper left hand coordinate of the window - they are expressed in screen coordinates; an integer specifying the depth of the window in characters; an integer specifying the width of the window in characters; a boolean flag - if it is TRUE the window will scroll upward if it overflows (just like a standard terminal) if it is FALSE, an overflow will cause DISPLAY to bring up a small window with the word "MORE" flashing in it - as soon as the user presses a key, the MORE window will go away and the overflowing window will be cleared and written with the overflowing data; another boolean flag - if it is TRUE fill mode will be turned on - in fill mode, words that would run over a vertical window boundary are moved to the beginning of the next line rather than being split in the middle; an integer page number specifying the page on which the window should appear; and a pointer to the window. In addition, box_window() requires a pointer to the border window. Note that the boolean parameters can be updated at any time through the commands scroll(), no_scroll(), fill(), and no_fill().

Once a window has been instantiated, it exists independently of all other windows. With few exceptions, the only interaction between windows occurs when a window is created or revealed subsequently
obscuring or revealing the windows under it. This is known as overlaying and is automatically handled by DISPLAY.

The remaining commands manipulate individual windows. Each command requires a pointer to the window which it will effect. They can roughly be divided into the following categories: commands which output text; other commands which modify what the window displays; miscellaneous commands which effect windows; and commands to get input from the keyboard.

The commands \texttt{w\_ch()}, \texttt{w\_s()}, \texttt{w\_ch\_n()}, \texttt{w\_s\_a()} and \texttt{nl()} print text to a window. \texttt{w\_ch()} and \texttt{w\_ch\_n()} print single characters. They except the following arguments: an ascii character to print; a character variable indicating the text attribute to use; a pointer to the window; and a boolean flag known as the force flag - it is reserved for future use and should always be set to \texttt{FALSE} (see "Further Work" below). \texttt{w\_ch\_n()} is identical except that it allows printing of multiple copies of the character. \texttt{w\_ch\_n()} expects an integer specifying the number of characters to print. It is very quick. \texttt{nl()} is a special case of \texttt{w\_ch()}. It accepts a single argument - a pointer to a window - and outputs a carriage return/newline.

The character printing commands are used to build string printing commands. These commands are \texttt{w\_s()} and \texttt{w\_s\_a()}. \texttt{w\_s()} takes string, a window pointer and a force flag (again, always set it to \texttt{FALSE}) as arguments. The string is a standard C character string. The escape
characters '\n' (newline), and '\t' (tab) are supported. \texttt{w_s_a()} also requires an integer representing the text attribute to print the string in.

The final output command is \texttt{win\_border()}. \texttt{Win\_border()} draws a border around the specified window. A bordered window created with this command is not the equivalent of a box created with \texttt{box\_window()}. Since the border of a box are located in a special window, they are protected from corruption by the any output commands directed at the main window. Borders created with \texttt{win\_border()} are not protected. \texttt{Win\_border()} is useful for distinguishing a window which is guaranteed not to have data written to the character positions on its edges.

The remaining commands will be given only a cursory introduction. They provide control of individual windows and pages.

The first set includes miscellaneous commands for updating the text displayed on the screen. \texttt{Refresh\_all()} completely redraws a window. This is used to update the screen when a window's internal text image has changed. \texttt{Clear()} erases the internal and external representations of the window's text. \texttt{Scroll\_win()} moves the window's text up a number of lines, leaving blank lines in their place.

The commands to format the window: \texttt{scroll()}, \texttt{no\_scroll()}, \texttt{fill()}, and \texttt{no\_fill()} simply modify the parameters initialized by \texttt{make\_win()} and \texttt{box\_window()}. \texttt{Write\_enable()} and \texttt{write\_disable()} are also formatting
commands. By default windows are writeable. That is they will display text generated by the writing commands. Windows that with write disabled ignore these commands.

A set of miscellaneous functions is provided to aid in controlling windows. **First_window()** returns the bottom most window on a particular page. This window is defined by DISPLAY itself and is the size of the screen. **Crt_cls()** instantly clears the entire screen (the current physical page). It does not update the windows or the internal representation of the page, thus it is only useful when all the windows will be killed or redrawn anyway. **Mv_(cur())** moves a given window's insertion point to the specified \(x,y\) coordinate. The coordinates are specified in window space. **Home()** moves the cursor the specified window's origin. **Beep()** rings the computer's console bell.

**Key_test()** and **key_in()** are simple commands used to get keyboard input from the user. **Key_test()** returns **TRUE** if the user has pressed any keys which have not yet been processed by the program. It returns **FALSE** otherwise. **Key_in()** returns an integer which represents the key that the user pressed. The low byte of the integer is the ascii code of the key pressed. The high byte of the integer is a control code which is used to support non-ascii keys such as function and cursor control keys. The IBM Basic manual explains the interpretation of control codes.

Here is a simple window example:
```c
#include "disp.equ"; /* define the DISPLAY data types */

/* global window pointers */
WINWP HELLO_WINP, HELLO_BOXP;
WINWP PROMPT_WINP, PROMPT_BOXP;

/* a pointer to the display object: used in kill_disp() */
DISPLAYP THE_DISPLAY;

main()
{
  DISPLAYP disp_init();
  char s[80];

  /* the initialization section */
  THE_DISPLAY = disp_init(1); /* use only one page */

  /* now set aside memory for a 20 X 10 box */
  declare_box(&HELLO_WINP, &HELLO_BOXP, 20, 10);
  /* and a 30 X 3 box */
  declare_box(&PROMPT_WINP, &PROMPT_BOXP, 30, 3);

  /* put the PROMPT window up */
  box_window(10, 10, 25, 1, FALSE, FALSE, 0, TRUE,
             PROMPT_WINP, PROMPT_BOXP);

  /* write the prompt in INVERSE */
  w_s_a("YOUR NAME PLEASE ", INVERSE, PROMPT_WINP, FALSE);

  /* get the user's name */
  getstr(s);

  /* now show the HELLO box */
  box_window(30, 5, 20, 10, FALSE, FALSE, 0, TRUE,
             HELLO_WINP, HELLO_BOXP);

  /* now print the hello message */
  beep(HELLO_WINP);
  /* note the use of tabs! */
  w_s_a("\t\thi there ", NORMAL, HELLO_WINP, FALSE);
  w_s_a(s, BOLD, HELLO_WINP, FALSE);
  /* this line will cause the window to overflow */
  w_s("\n\n\ntpress a key\n\n\n\n\\n\\n\n\n", HELLO_WINP, FALSE);
  w_s("GOOD BYE!!!!!", HELLO_WINP, FALSE);
  beep(HELLO_WINP);

  /* hide the HELLO window */
  kill_box(HELLO_WINP);

  /* have some fun */

  clear(PROMPT_WINP);
  w_s("Now Watch!", PROMPT_WINP, FALSE);
  /* re-instantiate the hello window */
  box_window(30, 10, 10, 10, FALSE, FALSE, 0, TRUE,
```
DISPLAY

HELLO_WINP, HELLO_BOXP);

mv_cur(2, 4, HELLO_WINP);
w_s_a("I'M BACK!!!", INVERSE, HELLO_WINP, FALSE);

/* clean up */
kill_box(PROMPT_WINP);
kill_box(HELLO_WINP);
kil_disp(THE_DISPLAY);
}

Further Work:

DISPLAY is a very stable system with few surprises. Any changes will be directed toward either expanding features or increasing efficiency. There are two fairly important improvements that should be considered. First, the memory required to declare a window is rather high. This is a result of the character for character internal representation. If groups of spaces and other common characters could be represented by a single code, substantial memory savings would result. The problem with such a scheme is that interpreting the code would require extra processor time, further slowing the display process. This is a classic example of familiar time-space trade off. The problem is serious here because DISPLAY is already on the slow side. While a careful analysis might improve DISPLAY's speed, it would be one time only benefit, not effecting the efficiency of future additions to the system.

One such feature that really needs to be added is support of color. As color displays become the standard, more and more pressure
builds for serious display managers to support color. Color hasn't been implemented so far because I haven't seen any way to do it efficiently. Essentially, the codes used to remember color are identical must be thought of as characters. Therefore making a completely general color mechanism is could reduce display speed by as much as 50 percent while doubling the memory requirements. Clearly this is unacceptable. Implementing color will require a clever algorithm which takes advantage of the fact that color is almost always uniform over a rectangular region of the screen.
2.1 DISPLAY COMMAND REFERENCE

This section describes DISPLAY functions from the point of view of a programer using DISPLAY to implement an application. It is divided into two sections: a description of the functions used to build an application, and a description of the procedures used internally by DISPLAY. The former is split into the following functional categories:

- Initialization
- Window Creation
- Text Output
- Other Screen Update Functions
- Window Format
- Window Destruction
- Cleaning Up
- Miscellaneous Window Functions
- Input

The section on internal functions is intended only for reference - they are for DISPLAY's use only and should not be called by outside modules.

All of these functions are located in the file \src\disp.c.
Initialization:

disp_init(p)

arguments: int p The number of virtual pages to support.
\leq 4

effects: allocates a new display object with p virtual pages,
does other initialization necessary for DISPLAY,
modifies _DISPDAT.

returns: DISPLAYP: A pointer to the new display object

notes: Call this procedure before using DISPLAY. Since each
virtual page includes a full-sized window, memory is
saved by using minimum p. The returned value should
be saved for use by kill_disp().
declare_window(w, x, y)

arguments: WINDOWP *w  Address of empty pointer to window
int x    Maximum width this window will use
int y    Maximum depth this window will use

effects: allocates space for a window of maximum size x * y,
modifies *w

returns: void

notes: Call declare_window() for each window, specifying the
maximum size it will use. It should be called during
program initialization, before any calls to free() are
made. The purpose of declare_window is to avoid
memory fragmentation, and allow you to use the same
memory for a number of windows. As long as windows
don't appear at the same time, it is O.K. to use the
same window pointer. For example, say window A and
window B never appear together. Call:

declare_window(&GENERIC_WINDOWP, x, y)

with x and y bigger than windows A and B ever get.
Then use:

WINDOW_POINTER A = GENERIC_WINDOWP and
WINDOW_POINTER_B = GENERIC_WINDOWP.

to make the windows to use the same memory.

declare_box(w1, w2, x, y)

arguments: WINDOWP *w1  Address of empty ptr to main window
WINDOWP *w2  Address of empty ptr to border window
int x    Maximum width the main window will use
int y    Maximum depth the main window will use

effects: allocates space for a box of maximum size x * y not
including border, modifies *w

returns: void

notes: See declare_win(). Declare_box() works by calling
declare_win() once for the border window (which only
serves to display the box's border) and once for the
main window (which displays the text). Note that the
border window is a full fledged window of sized x + 2
* y + 2, so boxes are expensive in terms of memory.
Window Creation:

\texttt{make\_win}(ox, oy, w, d, s, f, pn, it)

\textbf{arguments:}\quad int \quad ox \quad \text{screen coord. of left edge.}  
int \quad oy \quad \text{screen coord. of top edge.}  
int \quad w \quad \text{width in characters} 
int \quad d \quad \text{height in characters} 
BOOL \quad s \quad \text{scroll flag} 
BOOL \quad f \quad \text{fill flag} 
int \quad pn \quad \text{page number} 
WINDOWP \quad it \quad \text{window pointer to use}

\textbf{effects:}\quad \text{Updates the page map, sets window parameters, clears text memory, displays the (blank) window.}

\textbf{returns:}\quad WINDOWP: NULL if error, else it

\textbf{notes:}\quad \text{Use make\_win() to display the windows declared with declare\_win(). Calling make\_win() installs the window in the page map and on the screen. Each call to make\_win() must be matched by a corresponding call to kill\_window().}
box_window(ox, oy, w, d, s, f, pn, down, it1, it2)

arguments:
int ox screen coord. of left edge.
int oy screen coord. of top edge
int w width in characters
int d height in characters
BOOL s scroll flag
BOOL f fill flag
int pn page number
BOOL down dir to try if size conflict <see below>
WINiOWP it1 window pointer for text
WINiOWP it2 window pointer for border

effects:
Updates the page map, sets window parameters, clears
text memory, displays the (blank) box.

returns:
WINiOWP: NULL if error, else it1

notes:
Use box_window() to display the boxes that you
declared with declare_box(). Calling box_window() installs the box in the page map and on the screen.
Each call to box_window() must be matched by a corresponding call to kill_box(). box_window() places
a link between it1 and it2, so that commands directed
at it1 can effect it2. Since box_window() results
in the creation of a border window with width w + 2
and depth d + 2, it is possible to get an error from
box_window() where make_win() would work. (eg. imagine
the call to box_window() that results in a main window
at the very top edge of the screen. This does not
leave room for the border window which is one
character bigger in each direction.) Use down to
solve this problem. When box_window() gets an error,
it checks down. Setting down to TRUE tells
box_window() to increment oy and try again. It will
keep doing this until oy is greater than the depth of
the screen. Setting down to FALSE works the same way
from the bottom of the screen. Unfortunately there is
no support for the similar problem along the x axis.
Text Output:

w_ch(c, a, o, f)

arguments:  char  c  the character to print
             char  a  the text attribute to use
             OUTP  o  the output object
             BOOL  f  the force flag

effects:    Prints characters to output o, updating o's text image
            and possibly updating the screen. If o is a WINDOW
            with scroll disabled, w_ch() may cause a text overflow
            resulting in the creation of the MORE window, and a
            pause for user acknowledgement.

returns:    BOOL:  TRUE if text was output, FALSE on error
            condition

notes:      W_ch() will write single characters to an output. Mainly
            of interest for building higher level text manipulation.
            The force flag forces screen output even when there is an
            overlaying window. The force flag is not currently used.
            It should always be FALSE.

w_ch_n(c, n, a, o, f)

arguments:  char  c  the character to print
             int   n  the number of characters to print
             char  a  the text attribute to use
             OUTP  o  the output object
             BOOL  f  the force flag

effects:    Prints characters to output o, updating o's text image
            and possibly updating the screen. If o is a WINDOW
            with scroll disabled, w_ch_n() may cause a text
            overflow resulting in the creation of the MORE window,
            and a pause for user acknowledgement.

returns:    BOOL:  TRUE if text was output, FALSE on error
            condition

notes:      W_ch_n() will write single characters or multiple
            copies of a single character to an output. Used
            mainly for quickly erasing a large area. f is not
            used, and should be set to FALSE. Filling is handled
            by this procedure.
w_s(s, o, f)

arguments: char *s the string to write
OUTP o the output object
BOOL f the force flag

effects: Prints characters to output o, updating o's text image and possibly updating the screen. If o is a WINDOW with scroll disabled, w_s() may cause a text overflow resulting in the creation of the MORE window, and a pause for user acknowledgement.

returns: BOOL: TRUE if text was output, FALSE on error condition

notes: Essentially w_s() calls w_ch() for every character in s. There are two important differences though. First, w_s() supports filling, second it automatically uses w_ch_n() for any repeated characters, so it is very fast. The force flag is not used, f should always be FALSE. w_s() is DISPLAY's most common output primitive.

w_s_a(s, a, o, f)

arguments: char *s the string to write
int a the text attribute to use
OUTP o the output object
BOOL f the force flag

effects: Prints characters to output o, updating o's text image and possibly updating the screen. If o is a WINDOW with scroll disabled, w_s_a() may cause a text overflow resulting in the creation of the MORE window, and a pause for user acknowledgement.

returns: BOOL: TRUE if text was output, FALSE on error condition

notes: Essentially w_s_a() calls w_ch() for every character in s. There are two important differences though. First, w_s() supports filling, second it automatically uses w_ch_n() for any repeated characters, so it is very fast. The force flag is not used, it should always be FALSE. w_s_a is the same as w_s, accept that you may specify text attributes such as BOLD, INVERSE, FLASH and NORMAL.
nl(o)

arguments: OUTP o the output object
effects: Prints characters to output o, updating o's text image
and possibly updating the screen. If o is a WINDOW
with scroll disabled, nl() may cause a text overflow
resulting in the creation of the MORE window, and a
pause for user acknowledgement.
returns: void
notes: Writes a newline ('\n') to o, using text attribute
NORMAL.

win_border(o, a, ul, h, ur, v, ll, lr)

arguments: OUTP o the output object
int a the text attribute
char ul the upper-left-hand character
char h the horizontal character
char ur the upper-right hand character
char v the vertical character
char ll the lower-left-hand character
char lr the lower-right-hand character
effects: outputs to o, possibly updateing the screen.
returns: void
notes: win_border() is usually used by box_window() to set up
a box. If any user interaction will go on in the
window, use of box_window() is strongly recommend,
since it generates borders that are safe from over-
write by the user. Win_border() borders take less
memory, but result in borders with no more safety than
ordinary characters.

Other Screen Update Functions:

refresh_all(w)

arguments: WINDOWP w the window
effects: updates screen
returns: BOOL: TRUE
notes: refresh_all() re-draws window w. It does not honor
overlaying windows, so to accurately refresh the
screen you must also refresh all windows above w, and
all windows above them, etc.
clear(o)

arguments: OUTP o    the output object
effects: updates screen and internal representations of output
         object o.
returns: BOOL:     TRUE if clearing is done, FALSE otherwise
notes:  If o is a window, clear() first clears the area of the
        screen which o occupied. In all cases, the internal
        memory of o is then flushed. Note that clear() does
        not currently honor overlaying windows. Use
        refresh_all() to re-draw them in bottom up order.

scroll_win(n, o, a)

arguments: char n    the number of lines to scroll
           OUTP o    the output object.
           char a    the text attribute for the blank lines
effects: updates screen and internal representations of output
         object o.
returns: BOOL:     TRUE.
notes:  Moves the window's text up n lines while keeping the
        window's position fixed. The lines moving off the top
        are disposed of, and the n lines exposed at the bottom
        contain blanks. It will also update the screen if o
        is a window. At the time of writing, scroll_win() is
        not functional.

Window format:

scroll(o)

arguments: WINDOWP o    the window to scroll
effects: modifies scroll flag of o
returns: BOOL:     TRUE if o is a window, FALSE otherwise.
notes:  Sets o's scroll flag to TRUE, indicating that the
        window scrolls rather than mores. Note that scrolling
        is currently unimplemented.

no_scroll(o)

arguments: WINDOWP o    the window to more
effects: modifies scroll flag of o
returns: BOOL:     TRUE if o is a window, FALSE otherwise.
notes:  Sets o's scroll flag to FALSE, indicating that the
        window mores rather than scrolls.
write_enable(o)

arguments: OUTP o the output object
effects: modifies write flag of o
returns: void
notes: Sets o's write flag to TRUE, indicating that modification of the text of this output object is allowed. This is the typical condition.

write_disable(o)

arguments: OUTP o the output object.
effects: modifies write flag of o
returns: void
notes: Sets o's write flag to FALSE, indicating that modification of the text of this output object is not allowed.

fill(o)

arguments: OUTP o the output object.
effects: modifies fill flag of o
returns: void
notes: Sets o's fill flag to TRUE, indicating that incoming text strings should be filled. This is useful for displaying large unformatted documents such as help messages.

no_fill(o)

arguments: OUTP o the output object.
effects: modifies fill flag of o
returns: void
notes: Sets o's fill flag to FALSE, indicating that incoming text should not be filled. This is the typical setting since it is faster.
Window Destruction:

\textbf{\texttt{kill\_window}}(w)

- \textbf{arguments}: \texttt{WINDOWP w} the window to kill
- \textbf{effects}: updates the screen, modifies \texttt{*w}
- \textbf{returns}: \texttt{void}
- \textbf{notes}: This procedure is the opposite of \texttt{make\_win()}. It clears \texttt{w} from the screen and removes it from the page map. Like \texttt{make\_win()}, \texttt{kill\_window()} does not effect the memory allocated to \texttt{w}. So \texttt{w} can be used again. Unlike \texttt{clear()}, \texttt{kill\_window()} automatically takes care to re-draw any windows overlaying \texttt{w}. Thus \texttt{kill\_window()} is the preferred method for erasing windows from the screen.

\textbf{\texttt{kill\_box}}(w)

- \textbf{arguments}: \texttt{WINDOWP w} the main window to kill
- \textbf{effects}: updates the screen, modifies \texttt{*w}
- \textbf{returns}: \texttt{void}
- \textbf{notes}: \texttt{kill\_box()} calls \texttt{kill\_window()} for \texttt{w} (the main window of the box) and the border window associated with \texttt{w}. It is the preferred way to erase a box.

Cleaning Up:

\textbf{\texttt{kill\_disp}}(d)

- \textbf{arguments}: \texttt{DISPLAYP d} the display pointer to kill
- \textbf{effects}: clears all pages and kills associated windows, thereby updating the screen. \texttt{free's d}
- \textbf{returns}: \texttt{void}
- \textbf{notes}: Before exiting, it is good form to call \texttt{kill\_disp()}. It is not necessary though. This function may be of more use when virtual memory and multi-tasking becomes available.
Miscellaneous Window Functions:

**first_window(pn)**

- **arguments:** int pn a page number
- **effects:** none
- **returns:** WINDOWP: a pointer to the bottom most window on page p
- **notes:** This is the standard way to find out which window is on the bottom of the page. Once the bottom window is found, it is possible to reference the other windows.

**crt_cls()**

- **arguments:** none
- **effects:** updates the screen
- **returns:** void
- **notes:** crt_cls() is a call to the lowest level clearing routine. It instantly clears the entire screen, erasing windows without effecting their internal representations or page maps. It is a quick and dirty command used only in special cases.

**mv_cur(x, y, o)**

- **arguments:** char x new horizontal coordinate
  char y new vertical coordinate
  OUTP o the output object
- **effects:** updates cursor and insertio point position
- **returns:** BOOL: TRUE if coordinates are valid, FALSE otherwise
- **notes:** Moves the cursor and insertion point to the x,y position specified. x and y are specified relative to the output object's upper left hand corner. They are characters rather than integers for compatibility with the low level cursor routine in BIOS. Output object o becomes the active window.

**home(o)**

- **arguments:** OUTP o the output object
- **effects:** updates cursor position
- **returns:** void
- **notes:** Moves the cursor to the upper left hand corner of the output object.
**set_page(p)**

**arguments:** char p the page number to change to  
**effects:** changes virtual pages  
**returns:** BOOL: TRUE if virtual page p exists, FALSE otherwise  
**notes:** Switches to virtual page p.

**beep(o)**

**arguments:** OUTP o the output object  
**effects:** beeps the console speaker, or writes cntrl-G to an output file  
**returns:** void  
**notes:** If o is a window the console speaker is beeped. If o is a file, cntrl-G is written (deferring the beep until the file is printed). If o is a memory buffer, nothing is done (since the user shouldn't be alerted to internal work). beep() will also beep the console if given NULL as an argument. This is handy when a beep is necessary but there is no convenient output object to use.

**Input:**

**key_test()**

**arguments:** none  
**effects:** none  
**returns:** BOOL: TRUE if there is unread keyboard input, FALSE otherwise.  
**notes:** The type ahead buffer will store approximately 20 characters of keyboard input that has not been read by the application program. key_test() returns true if there is any such input. Use this procedure to strobe the input until a key is present, then call key_in().
key_in()

arguments: none

effects: reads one character from the type ahead buffer

returns: int: The low byte is the ascii character code, the
high byte is a special code used to signal function
and cursor keys.

notes: This procedure will read the first character from the
type ahead buffer returning an integer representing an
ascii code and a control code. I do not know what it
will do if called when there are no characters in the
buffer, so call it after key_test() returns TRUE.
These commands are used to build higher level input
procedures (see FORMS). The prototypical calling
sequence is:

```
while (key_test() == FALSE)
    /* wait for input */
    input = key_in();
    input_character = (char)(input % 256);
    input_code = (char)(input / 256);
    /* do something with the new input */
```

Control codes are interpreted in the IBM Basic Manual

Internal Functions:

NOTE: Internal functions are for use by DISPLAY only. They
should not be called by outside modules.

_init_page(p)

arguments: PAGEP p the page to initialize

effects: allocates memory for the pages, page map, and base
window

returns: BOOL: TRUE if it was possible to make the window,
FALSE otherwise.

notes: _init_page() declares and makes the window which
becomes the bottom window of page p. This is always
the window returned by first_window(p).
_init_map(m)

arguments: LINE_MAPP m the map to initialize
effects: clears the line map
returns: void
notes: _init_map sets all elements of m to NULL, thus indicating that there are no windows on the page.

_write_char(c, n, a, o, f)

arguments: char c the character to print
int n the number of characters to print
char a the attribute to use
OUTP o the output object
BOOL f the force flag
effects: prints characters to output o, updating o's text image and possibly updating the screen.
returns: int: the number of characters actually written
notes: _write_char() is the lowest level writing procedure included in DISPLAY proper. It is responsible for correlating the screen image and the internal text image. _write_char() thinks of newline as a single character.
_base_make_win(ox, oy, w, d, s, f, pn, clr, it)

arguments: int ox screen coord. of left edge.
          int oy screen coord. of top edge.
          int w width in characters
          int d height in characters
          BOOL s scroll flag
          BOOL f fill flag
          int pn page number
          BOOL clr clear on display
          WINDOWP it window pointer to use

effects: Updates the page map, sets window parameters, clears
text memory. Displays the (blank) window if clr
equals TRUE.

returns: WINDOWP: NULL if error, else it

notes: _base_make_win() does the work of make_win(). In
addition it supports the clr argument. If clr is
TRUE, as is always the case with calls to make_win(),
the window will be immediately drawn on the screen.
Otherwise the window will be fully created with the
exception of the clearing which precedes screen
update. This facility is used to make boxes. First
the border window is made with a regular make_win()
call, clearing the area in which the box will appear.
Then the main window is made with a call to
_base_make_win() with clr set to FALSE. There is no
need to clear the area under the main window since it
has already been cleared (it is directly over the
border window). The result is a smooth box creation
with no redundant clearing.

_overlay_number(o)

arguments: int *o ptr to an int which is really an output
          obj

effects: none

returns: int: the overlay value of the output object

notes: _overlay_number() is used by kill_window() to order a
set of window pointers into a bottom up list. This
list is then used to refresh the display while
honoring overlays.
_refresh_part(w, x1, y1, x2, y2)

arguments: WINDOWP w the window to re-draw
           int x1 the rightmost column to redraw
           int x2 the leftmost column to redraw
           int y1 the line to start redrawing from
           int y2 the line to redraw to

effects: updates the screen

returns: BOOL: TRUE if w is a window, FALSE otherwise

notes: This procedure converts l1 and l2 from screen
       coordinates to l1' and l2' in the coordinates of w. w
       is then re-drawn from line l1' to l2'. The
       specification of l1 and l2 minimizes the amount of w
       that must be redrawn. This is for efficiency.

_refresh_window(w)

arguments: int *w ptr to an integer which is really a
           window

effects: updates the screen

returns: void

notes: _refresh_window() converts w to a pointer to a window,
       then calls refresh part. The effect is to refresh the
       part of w that lies under the window pointed to by the
       global variable _ex_win.

_weed_windows(s)

arguments: SETP s set of ptrs to windows in ascending
           order

effects: none

returns: void

notes: _weed_windows() scans s and removes any window
       pointers whose windows do not overlap other windows.
       _weed_windows() is used by _base_kill_window().

_base_kill_window(w, c)

arguments: WINDOWP w the window to kill
           BOOL c clear on kill

effects: clears text memory, may update the screen

returns: BOOL: TRUE if w is a window, FALSE otherwise

notes: _base_kill_window() does the work of kill_window().
       In addition it provides the option to kill the window
       without erasing it from the screen. This option is
       invoked by setting c to FALSE. It is used by
       kill_box() to provide smooth boxes. See
       _base_make_win() for explanation.
_kill_page(p)

arguments: PAGEP p a pointer to the virtual page to kill
effects: clears all pages and their page maps, and kills
associated windows, thereby updating the screen.
returns: void
notes: _kill_page(p) is called only by kill_disp(). It kills
p's windows and initializes p's page map.
2.2 BIOS

BIOS is IBM's Basic Input Output System. It supplies the most basic screen I/O functions through direct interrupts to the micro processor. The functions described here simply set up the proper registers and call BIOS. They serve to insulate DISPLAY from the nastiness of low level I/O and BIOS. Every C compiler has its own syntax for interfacing with BIOS. The result is that these functions must be recoded for each compiler.

For Microsoft C version 4.0 the relevant include files are dos.h and conio.h. The DISPLAY file bios.equ contains equates for the relevant interrupts. The code itself is in \src\bios.c.

When designing a screen interface for the PC family there are three candidates for low level I/O. The ANSI device driver supplied with PC-DOS is the most standard. It implements the ANSI standard terminal protocol (DEC VT100 I think). The advantage of using this driver is compatibility with many other terminal types. Unfortunately, because of its generality, ANSI is the slowest of the candidates. The fastest method of screen I/O is to write characters directly to the memory used by the graphics adaptor. This method is used by most commercial word processors. The problem with it is that the program
must be able to determine where the graphics memory starts, and how it
is used to represent strings of characters. The result is that each
graphics adaptor (there are 5 popular ones and innumerable custom jobs)
requires its own I/O program. Furthermore the application must be able
to determine which graphics adaptor is currently in use. These
limitations are too severe for the purposes of DISPLAY. BIOS is a
compromise in which the standard IBM screen protocol is used. It is not
as fast as direct memory access, but it is much more flexible and will
work with all IBM PC compatible computers now and in the future.

_b_set_cur(x, y, p)

arguments:  char x the X screen coordinate
char y the Y screen coordinate
char p the page number.
effects: Moves the cursor to (x,y) on page p.
returns: void
notes: none

_b_write_char(c, n, a, p)

arguments:  char c the character to print
int n the number of characters to
write
char a the attribute to use
char p the page number
effects: Writes n c's to page p starting at p's cursor. The
caracters are written in attribute a. p's cursor is
not moved.
returns: void
notes: none
_b_scroll_win(o, ux, uy, lx, ly, a)

arguments:  char  n  the number of lines to scroll up.
            char  ux  the left column
            char  uy  the top row
            char  lx  the right column
            char  ly  the bottom row
            char  a  the attribute for the blank lines

effects:    Defines a rectangle with screen coordinates
            (ux,uy) (lx,ly) and scrolls the text in that rectangle
            up by n lines. The space vacated at the bottom of the
            region is filled with blank lines of attribute a.

returns:    void

notes:      none

_b_set_page(p)

arguments:  char  p  the page to display

effects:    gives physical page p the control of the computer's
            physical screen.

returns:    void

notes:      none

_b_key_in()

arguments:  none

effects:    Removes one character from the type-ahead buffer.
            the input character (see below)

returns:    int

notes:      This function returns the first character in the type-
            ahead buffer. The low byte is the ascii character,
            the high byte is the control code (see IBM's Basic
            manual for control code interpretations). It is not
            actually implemented as a bios call. Instead it calls
            MSC's getch() function.

_b_key_test()

arguments:  none

effects:    none

returns:    BOOL returns TRUE if there are characters waiting
            in the keyboard type-ahead buffer. FALSE otherwise.

notes:      This function is not actually implemented as a bios
            call. Instead it calls MSC's kbhit() function.
_b_read_char(p, c, a)

**arguments:**
- char  p       the page to use
- char  *c     a pointer to a character variable
- char  *a     a pointer to a character variable

**effects:** none
**returns:** void
**notes:** This function looks up the value and the text attribute of the character under p's cursor. It sets *c to the ascii value and *a to the attribute value.

_b_read_mode(m, c, p)

**arguments:**
- int  *m     a pointer to an integer variable
- int  *c     a pointer to an integer variable
- int  *p     a pointer to an integer variable

**effects:** modifies *m, *c, and *p
**returns:** void
**notes:** This function looks up the current display mode, number of columns, and active page number. It sets *m to the mode number, *c to the column number, and *p to the page number. The mode number is interpreted as follows:

- 0 = 40 column by 25 row black and white mode
- 1 = 40 column by 25 row color mode
- 2 = 80 column by 25 row black and white mode
- 3 = 80 column by 25 row color mode

_reg_zero(r)

**arguments:** REG r a pointer to the register to clear
**effects:** Sets all fields of the register to zero.
**returns:** void
**notes:** This is a macro. See dos.h (included in MSC) for definition of REG. It is used by all BIOS commands to clear the memory used to directly communicate with the computer's microprocessor.
2.3 DISPLAY GLOSSARY

Active window
The last window updated is the active window. Updates include printing and moving the insertion point. All input goes to the active window. It is usually the top most window.

Attribute
The text attribute is a one byte value describing the color a character will be displayed in. The attribute is calculated according to the following tables:

Table 2.1 Monochrome Display Attributes
(global equates in parentheses)

| Black characters on black background | (BLANK) | 0 |
| White characters on black background | (NORMAL) | 7 |
| Bright characters on black background | (BOLD) | 15 |
| Black characters on white background | (INVERSE) | 112 |
| Flashing NORMAL characters | (BLINKNORMAL) | 135 |
| Flashing BOLD characters | (BLINKBOLD) | 143 |
| Flashing INVERSE characters | (BLINKINVERSE) | 240 |

Table 2.2 CGA Display Attributes

<table>
<thead>
<tr>
<th>Foreground Colors</th>
<th>Background Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>Black</td>
</tr>
<tr>
<td>Gray</td>
<td>Blue</td>
</tr>
<tr>
<td>Blue</td>
<td>Green</td>
</tr>
<tr>
<td>Light Blue</td>
<td>Cyan</td>
</tr>
<tr>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>Light Green</td>
<td>Magenta</td>
</tr>
<tr>
<td>Cyan</td>
<td>Brown</td>
</tr>
<tr>
<td>Light Cyan</td>
<td>White</td>
</tr>
<tr>
<td>Red</td>
<td></td>
</tr>
<tr>
<td>Light Red</td>
<td></td>
</tr>
<tr>
<td>Magenta</td>
<td></td>
</tr>
<tr>
<td>Light Magenta</td>
<td></td>
</tr>
<tr>
<td>Brown</td>
<td></td>
</tr>
<tr>
<td>Yellow</td>
<td></td>
</tr>
<tr>
<td>White</td>
<td></td>
</tr>
<tr>
<td>Bright White</td>
<td></td>
</tr>
</tbody>
</table>
To calculate the values of screen attributes for a CGA display, follow this procedure:

- Choose a foreground color and a background color from table 2.2.
- Add the color's values. If the text is to flash, add 128.

**Base Window**
Each virtual page automatically contains a base window. This is a normal window of maximum size. All other windows overlay the base window.

**BIOS**
BIOS is the IBM Basic Input Output System. It is composed of a set of ROM traps which accomplish primitive functions like writing a character to the screen.

**Border Window**
A border window is a dummy window used to implement the borders around boxes. They should not be modified by the user. See Main Window.

**Box**
A box is a window with a border around it. Most output windows are boxes.

**Command Window**
A window used to get commands (verbs) from the user. See Object Window.

**Cursor**
The cursor indicates which window is active, and where new text will be printed. The cursor is usually a blinking underline or block.

**Insertion Point**
Each window has an insertion point which indicates where the next character of text will be printed.

**Internal Representation**
In order to remember the contents of a window which has been overlain, the window's text is stored internally. These internal representations are managed by DISPLAY.

**Main Window**
A main window is the window used by boxes to display text. It is a normal window with the exception that it is linked to a border window. When referring to a box,
use a pointer to its main window. See Border Window.

More
If a window's more flag is TRUE, and text is sent to it that would cause its insertion point to move beyond the lower right hand corner, a small window will appear in the upper left hand corner of the screen. The window contains the word "MORE". Nothing happens until the user presses a key. The window will then erase its text, reposition its insertion point to the upper left hand corner, and continue displaying text. See Scroll.

Object Window
A window used display and manipulate data. See Command Window.

Overlay
When one window partially obscures a second window, the first window is said to overlay the second. Maintaining the data in the overlain window is one of the most difficult aspects of DISPLAY.

Page Map
A page map is a DISPLAY data structure used to record the positions of windows on a virtual page. It is mainly used to handle the overlaying of windows.

Physical Page
A physical page is an area of memory automatically allocated for use by the screen. The text displayed on the screen is contained in one physical page. The CGA adaptor has 4 physical pages. Thus it can hold 4 screenfuls of text, even though it can only display one at a time. See Virtual Page.

Screen:
The screen is the computer's interactive output device. The monitor. They are typically 25 rows deep by 80 columns wide.

Scroll
If a window's more flag is FALSE, and text is sent to it that would cause its insertion point to move beyond the lower right hand corner, the window's text is moved up enough lines to accommodate the incoming text, and the text is displayed. Text moving off the top of the window is lost. See More.
Type-Ahead-Buffer: Even when the computer is doing calculations, it reads the keyboard. The first 20 or 25 keys pressed during long calculations are stored in the type-ahead-buffer. When the computer's attention returns to the user, keys stored in the type-ahead-buffer are acted on before the any further keystrokes.

Virtual Page: A virtual page is a DISPLAY data structure associated with a particular physical page. A virtual page keeps track of the windows currently defined within the page. See Physical Page.

Window: A window is the basic display element. It is a rectangular region on the screen, in which text is manipulated.
3.0 FORMS

Introduction:

Forms templates for the structuring of information. The FORMS module implements templates. Input templates built with this system allow the programmer to define the control structure of his application in a easy, modular way. In addition, FORMS gives the application's users a powerful, uniform interface with their data.

Basically a form\(^1\) is a data structure that describes how another data structure will be displayed and manipulated. It is best to think of a form as a transparent overlay which gives individual meaning to the raw data of an internal binary representation. FORMS allows a form to combine with such internal data, giving the user access to the data.

A form itself is a data object. It contains information on the size and origin of the window it will use for communication with the user. More importantly, it maintains a list of form-lines. Each form-line describes one entry in the form. A typical entry will contain a string of descriptive text, and a pointer to the internal data to be formatted. Each entry may also contain fields specifying default values.

---

\(^1\) The words "form" and "template" have the same meaning in this discussion.
for the given data, upper and lower bounds that the data may assume, and an entry in a help file.

Fill_object() is the main procedure of FORMS. When fill_object() is called with a form, an internal data structure, and a window, it displays the data structure in the format specified by the form. (see figure 2.1 for an example of two actual forms.) The user is then free to examine the data and make changes directly to it. The changes he makes are automatically reflected in the internal representation.

Keyboard Input:

Fill_object() presents the user with a window full of entries. Here is an example form with only two entries:

<table>
<thead>
<tr>
<th>Facility Name:</th>
<th>Dashields</th>
</tr>
</thead>
<tbody>
<tr>
<td>The value of Pie:</td>
<td>3.14</td>
</tr>
</tbody>
</table>

The words "Dashields" and "3.14" are textual representations of internally stored data. The internal data is almost certainly not in a human readable format, but the form-line tells fill_object() how to format the data. On the computer screen, the currently active textual representation (the one with the cursor in it - here we represent the cursor with an underline) appears in inverse video (black on white). This indicates that the user may modify it by simply typing at the keyboard.
For simple text input it is sufficient to use the alpha numeric keys. In addition, the following special keys facilitate editing. By default they conform to the key bindings of the Emacs text editor. They can be easily changed from within the program by using the general help menu (see below).

The keys recognized are as follows:

**to get exit the form:**
- ESC exit this form
- cntrl Q exit this form

**to move between lines (switch active fields):**
- cntrl N go to next form-line
- cntrl J go to previous form-line

**to move around in the active field:**
- cntrl F move one character forward
- cntrl B move one character backward
- cntrl A move to the beginning of the input field
- cntrl E move to the end of the input field

**to edit the active field:**
- cntrl U delete the entire line
- cntrl K delete the line to the right of the cursor
- cntrl D delete the character under the cursor
- BACKSPACE move the cursor one character to the left and delete it

**miscellaneous:**
- ENTER execute this field (only if it is a command)
- cntrl L redraw screen
- cntrl Y get help on this input field (if it exists)
- cntrl X display the general help menu

There are two special types of form-line which bear mentioning.
First there is the Boolean data field. The internal representation of this field is a single character. If its value is TRUE, then it assumes boolean truth. Likewise if its value is FALSE, it evaluates to a
boolean falsehood². The textual representations used by FORMS are the strings "YES" and "NO" corresponding to boolean truth and falsehood respectively. Pressing a key when the active field is boolean simply toggles between "YES" and "NO". This makes sense since boolean values may only assume two states.

The other interesting form-line is a "command" or procedure "form-line". At present there is no textual representation of these lines. When one becomes active a single square appears in inverse video. Pressing any key other than return does nothing. If return is pressed however, an associated function is called. This is very convenient, for it means that a form can do more than display data. It can cause action! For example, forms can be used to make a menu, or a form containing data might also have procedure form-lines that allow the user to access other forms.

Functions which form-lines can call must obey the following format. They must accept four arguments: a pointer to a character — this is the character which resulted in a call to the function (thus it is always a return) — the value of this variable is that the function can modify it, thereby making FORMS think that the user pressed a different key; a pointer to the form which called it; a pointer to the internal data which the calling form was formatting; and a pointer to the calling form's window. If these values are not necessary, they can

² TRUE and FALSE are global equates. The are defined as the appropriate values to cause execution in an "if then" statement. Their actual value is highly machine dependent.
be ignored. All other input must come from global variables set up beforehand. Functions to be called by a form-line must also be define to return void.

**Command Line Format:**

FORMS uses the standard syntax of printf()\(^3\) to format its data fields. In summary, the valid data formats are:

- %b: a boolean
- %p: a procedure
- %c: a single character
- %s: a string
- %d: an integer
- %u: an unsigned integer
- %o: an integer in octal notation
- %x: an integer in hexadecimal notation
- %f: a floating point number
- %e: a floating point number with exponent
- %g: use %e or %f whichever is shorter

The first two: %b and %p are unique to FORMS. They are used to tell FORMS that the internal data is a boolean value or a pointer to a procedure.

**Defining Forms — The Form Description Language:**

While it is possible to manually allocate the space for a form, it is much easier and more efficient to use the Form Description Language

---

\(^3\) Check your favorite C reference manual.
(FDL). FDL allows the definition of forms in an external text file. Load_form() reads the form in from the file, automatically doing all allocation and offset calculation. FDL is more efficient because in order to manually create a form, the program must contain statements that fill the form's fields with the appropriate data. This results in two copies of the fields - one in the form itself and one in the program which creates the form - plus the code to actually fill the form's fields. Thus memory requirements for manually created forms are more than twice as great as those for forms created by FDL. FDL is also more flexible since forms can be changed without necessitating reccompilation. This results in reduced debugging time, and allows the user to easily customize his forms. FDL describes forms as follows:

- The first two lines are numbers declaring the (X, Y) coordinate of the preferred upper left hand corner of the window which the form will use. This value is expressed in screen coordinates. It is up to the application programmer to make use of these values when creating the window.

- The second pair of numbers declares the minimum size of the form's window. The first number is the window's width, the second is its depth. They are in characters. The sum of the X coordinate and the width should be less than the width of the screen. Likewise the total depth should be less than the screen's depth. When doing these
calculations, don't forget that if the form is to be displayed in a box, the border will require an extra character on each side. As with the X,Y coordinates, the application programmer is responsible for giving fill_object() a window with at least enough room. Since all four values described so far are dependent on the size of the form's data, it is best to define the rest of the form first and come back to these values after their values are better know.

* The remainder of the form's description is made up of form-line descriptions. They are each started by a start command (double asterisks - **) in the leftmost column. The text of the form then follows with one field on each line. Since many form-lines contain blank fields, the following commands are provided to skip to a field further down.

* skip this field only
*LOW skip to low bound
*HIGH skip to high bound
*HELP skip to help field

Note that once a field has been skipped, there is no way to go back to it.
• Three more FDL commands handle advanced uses:

• An obscure bug (in MSC?) causes a small percentage of the fields in a form line to become mis-aligned. The effect is a trashed display beginning at the line where the bug occurs. The solution is to place the *BUGFIX command just before the offending line.

• Load_form() returns the offset's final value. When designing a form which will display a large data object, it is good practice to break it into parts. This way one form can call the other ones. In order to do this it is necessary to find the offset value of the beginning of the sub-form from the beginning of the data object. This is accomplished by giving the FDL command *RETOFF at the point where the new form should start. If this is done, the value of the offset when *RETOFF is encountered is the one returned.

• Sometimes it is not desirable to display a section of a data object. In order to accomplish this, the offset must be incremented by the number of bytes of the data. The command *SKIPAHEAD accomplishes this task. It causes load_form() to read another line of the form description file. This line must contain a valid data format in the leftmost position. This data format is converted into a number equivalent to the number of bytes a data object of that type would require. The offset
is then incremented by that many bytes. For example, to leave a "hidden" float in your data, simply include the lines:

```
*skipahead
f
```

Just where the data's form-line would have been defined.

- Finally, any line with a semi-colon (;) on the leftmost side is interpreted as a comment and ignored.

When using FDL with command forms, it is important to note that C does not permit data files to reference pointers to functions. For this reason, it is still necessary to manually load the form-line with the function's pointer. It is still useful to use FDL for the bulk of the form. To load the function pointer simply write a small procedure using find_line().

For the moment the same caveat is true when using the bounds checking procedures described in section 3.1. This limitation will disappear in future implementations however.

Here is an example form description:

```
; example form 1
; it starts one character in from the top left hand corner of the ; screen. (It may appear in a box.)
1
1
; it is 57 characters wide and 17 characters deep
57
17
; here are its form-lines
```
Further Work:

As a whole, FORMS is a quite stable module. There are however a few improvements that could be made. They are enumerated below:

- Each form-line contains an offset which is added to the pointer to the internal data object. The resulting value is then converted into a pointer to the appropriate data type as defined by the form-line's command string.

Although this process sounds simple, it is perhaps the
most difficult aspect of FMS. Work needs to be done to find a better way to accomplish this goal.

- Presently, the cursor begins on the leftmost edge of the active field. Typing a character either overwrites the character under the cursor, or pushes the characters to the right of the cursor over one space. The insert flag controls which behavior is used. This is is natural for text input, but not for numeric input. Numeric input should proceed from the right to the left. This should be a fairly simple change.

- As noted above, the presence of a procedure datatype is not indicated until it becomes the active field. The programmer should be allowed to define a special prompt for use by each procedure.
3.1 FORMS COMMAND REFERENCE

This section describes FORMS functions from the point of view of a programmer using FORMS to implement an application. It is divided into two sections: a description of the functions used to build an application, and a description of the procedures used internally by FORMS. The former is split into the following functional categories:

- Initialization
- Form Creation
- Form Use
- Relations

The section on internal functions is intended only for reference - they are for Form's use only and should not be called by outside modules.

All of these functions are located in the file `\src\forms.c`.

Initialization:

`init_temp()`

- **arguments:** none
- **effects:** allocates and initializes all key bindings, menus and windows used by FORMS.
- **returns:** void
- **notes:** `Init_temp()` must be called before any other FORMS commands. It preforms general initialization. Of note, two windows: `_MENU_WINDOWP` and `_HELP_ERROR_WINDOWP` and a form: `_mentemp` are declared.
Form Creation:

load_form(ob, file_name, off)

**arguments:**
- OBJ_TYPEP ob the form to read in
- char *file_name the file to read from
- int off the initial offset to start the form from

**effects:** modifies ob and allocates new form-lines

**returns:**
- int : if the file contains the string
  - "*RETOFF", the offset in effect at the time "*RETOFF"
  - is encountered is returned, otherwise the final offset
  - is returned

**notes:** Load_form() opens the specified file and reads in the
- form description contained therein. See the
- discussion of the form description language above.

Form Use:

fill_object(temp, obj, out)

**arguments:**
- OBJ_TYPEP temp the form to use for formatting
- char *obj a pointer to the object to fill
- OUTP out the window to use

**effects:** *obj is filled with user input

**returns:**
- int: _QUIT if user quits _ESCAPE if the user escapes,
  - -1 on error.

**notes:** Fill_object() is the command used to access FORM's
- capabilities. It displays the information in *obj in
  - the specified window formatted according to temp. Obj
  - is a pointer declared as a character because there is
  - no way to know exactly what high level data type it
  - is.

find_line(o, s, n)

**arguments:**
- OBJ_TYPEP o the form to search through
- char *s the string to search for
- int n the depth to search

**effects:** none

**returns:**
- OBJ_LINEP: a pointer to the line. Failed searches
  - return NULL

**notes:** find_line() preforms a simple linear search through
- o's form-lines looking for one with a command line
  - equal to s. A pointer to the form-line containing nth
  - match is returned. If no match occurs, or n is
  - greater than the number of matches, NULL is returned.
Relations:

**NOTE:** Relations provide a means to bound the value a user may input. There are 13 relation functions as enumerated below. They all have the syntax given for \( gL() \).

\( gL(x, l, h) \)

- **arguments:**
  - double \( x \): a number to be bounded
  - double \( l \): the lower bound
  - double \( h \): the upper bound

- **effects:**
  - none

- **returns:**
  - double: the bounded value of \( x \)

- **notes:**
  - \( gL() \) bounds \( x \) to be greater than \( l \). If \( x \) is less than or equal to \( l \), \( x+1 \) is returned, otherwise \( x \) is returned. If bounding occurs, the console bell is beeped.

The other relations are:

- \( gL() \):
  - \( x > l \)
- \( geL() \):
  - \( x \geq l \)
- \( LH() \):
  - \( x < h \)
- \( leH() \):
  - \( x \leq h \)
- \( LBH() \):
  - \( 1 < x < h \)
- \( ebH() \):
  - \( 1 \leq x < h \)
- \( Lbe() \):
  - \( 1 < x \leq h \)
- \( ebe() \):
  - \( 1 \leq x \leq h \)
- \( neL() \):
  - \( x \neq 1 \)
- \( LoH() \):
  - \( x < 1 \) or \( x > h \)
- \( eoH() \):
  - \( x \leq 1 \) or \( x > h \)
- \( Loe() \):
  - \( x < 1 \) or \( x \geq h \)
- \( eoe() \):
  - \( x \leq 1 \) or \( x \geq h \)

The names of these functions are interpreted as follows:

- \( L = \) lower bound
- \( H = \) upper bound
- \( l = \) less than
- \( g = \) greater than
- \( e = \) equals
- \( b = \) between
- \( o = \) outside (not between)
Internal Functions:

NOTE: Internal functions are for use by FORMS only. They should not be called by outside modules.

_hello_handler(file, offset, w)

arguments:   char *file the help file to use
             long offset the beginning of the particular help record
             OUTP w the window originating the help call

effects:     Switches to and modifies the page 2 base window

returns:     BOOL: TRUE if help message successfully displayed

otes:       Help_handler() opens the specified file and scans offset characters into it. It then switches to page 2 and prints the file to the page 2 base window. It continues reading and printing until the string "#END_HELP" appears in the file. It then pauses until the user hits a key, and switches back to the originating page.

_men_help(k, l, o, w)

arguments:   the arguments are ignored

effects:     sets _key_pass to 0

returns:     void

otes:       _men_help() uses help_handler() to display the help file "\help\general.hlp". This file contains notes such as key bindings that are useful in all applications of FORMS.

_men_redefine(k, l, o, w)

arguments:   the arguments are ignored

effects:     sets _key_pass to 0

returns:     void

otes:       This procedure allows the user to redefine the key bindings recognized by FORMS. It is not currently implemented.
_men_exit(k, l, o, w)

arguments:  the arguments are ignored
effects:    sets _key_pass to _ESCAPE
returns:    void
notes:      _men_exit() exits the current form and the form which
            called it (if there is one).

_signal_quit(k, l, o, w)

arguments:  the arguments are ignored
effects:    none
returns:    void
notes:      _signal_quit() exits the current form.

_print_template_line(l, o, out)

arguments:  OBJ_LINEP      l  the form-line to use for
            char *o      a pointer to the object to fill
            OUTP     out the window to use
effects:    none
returns:    BOOL: FALSE on errcr, TRUE otherwise
notes:      _print_template_line() prints the text associated with
            line l, then reads the data pointed to by o+l->offset
            and interprets it according to the format command in
            l. The result is printed in the window out.

_write_template_val(s, v, a, out)

arguments:  char *s      the command string
            char *v      a pointer to the data to be interpreted
            char a       the text attribute for printing
            OUTP     out the window to use
effects:    none
returns:    BOOL: FALSE on errcr, TRUE otherwise
notes:      _write_template_val() is used by print_template_line() to
            format and print the form-line's data. The output
            goes to in the window out.
_in_template_line(l, o, w)

arguments: OBJ_LINEP l the form-line to use for formatting
           char *o the object to fill
           WINDOWP w the window to use for input

effects: modifies o

returns: int: returns the terminating character

notes: _in_template_line() handles the basic input operations of FORMS. It scans the keyboard, interpreting the keystrokes in context of the format of the form-line l. Results are displayed in w, and o+l->offset is updated accordingly. It returns when the user presses the key bound to _ESCAPE or _QUIT. All user interaction with FORMS goes through this procedure.

_copy_defaults(t, o)

arguments: OBJ_TYPEP t the form to model o after
           char *o a pointer to the object to fill

effects: modifies o

returns: void

notes: _copy_defaults() fills the fields of *o with the values specified in the def fields of t's form-lines. Thus if t is a form describing a typical, _copy_defaults() will make *o a typical martian. It is up to the caller to insure that *o is large enough to hold the data.

_str_to_num_util(s1, s2, c, l, o)

arguments: char *s1 a number as input by the user
           char *s2 a formatting string
           char c the formatting command
           OBJ_LINEP l the form-line to use for formatting
           char *o a pointer to the object to fill

effects: none

returns: void

notes: _str_to_num_util() takes s1 (a textual representation of a number) and converts it to an int or a float in C's internal format. The int or float is stored in o at o+l->offset. The number is constrained by l's relation procedure. Note: c and s2 are redundant, the redundancy increases efficiency.
_rdform(s, l, f)

arguments: char *s a buffer to fill from the file
           int l the size of the buffer
           FILE *f the file to read from

effects: none

returns: int : the number of characters actually read
         before a line-feed or end-of-file was encountered in f.

notes: _rdform() reads fills s with characters read from f. Reading stops when a new-line or end-of-file is encountered. _rdform() skips lines whose first non-white-space character is a semi-colon. These lines are considered comments. The procedure also converts the two character sequence "\n" to the single character whose ascii code is 10. This allows new-lines to be included without terminating the read.

_parseform(s)

arguments: char *s a string to interpret

effects: none

returns: int: the parse value of the string

notes: _parseform() determines if s is one of the form description language commands. If so an integer corresponding the the particular command is returned, otherwise _parseform() returns 8.

_calc_offset(off, c)

arguments: int *off a pointer to an integer offset
           char c a command character

effects: modifies *off

returns: void

notes: _calc_offset() interprets c as a command character a la printf (d == int, f == float, etc). The size of this datum is added to the value pointed to by off.
### 3.2 FORMS GLOSSARY

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>A data structure describing the formatting of another data structure.</td>
</tr>
<tr>
<td>Form Description</td>
<td>A means of defining a form without recourse to manually implementing it.</td>
</tr>
<tr>
<td>Language</td>
<td></td>
</tr>
<tr>
<td>Form-line</td>
<td>An entry a form. A form-line describes one line of a form.</td>
</tr>
<tr>
<td>Offset</td>
<td>A number that, when added to a pointer to an internal data structure, results in the address of a particular data field to be formatted.</td>
</tr>
<tr>
<td>Relations</td>
<td>Functions which implement the bounding of a form-line's value.</td>
</tr>
<tr>
<td>Template</td>
<td>A template is a Form.</td>
</tr>
</tbody>
</table>
4.0 UTILITIES

Introduction:

Utilities are miscellaneous functions used throughout the program. They consist of memory managers, math functions, and string manipulators. Of special note is the implementation of the SET data type. Sets are groups of elements, each of which is a pointer to an integer. In practice, the elements are typically used to things other than integers - windows perhaps. Within a set, each element is unique. Sets can be sorted, iterated over and searched.
4.1 UTILITIES COMMAND REFERENCE

This section describes the miscellaneous utility functions used to support DISPLAY, FORMS, and FMS. It is divided into the following functional categories:

- String functions
- Memory functions
- User input functions
- Math functions

These functions are located in the file `\src\util.c` and known collectively as the UTILITY module.

String Functions:

`sub_copy(s1, s2, r1, r2, r3)`

**Arguments:**
- `char *s1` the string to copy from
- `char *s2` the string to copy to
- `int r1` the first position in `s1`
- `int r2` the last position in `s1`
- `int r3` the first position in `s2`

**Effects:** modifies `s2`  
**Returns:** `void`

**Notes:** `Sub_copy()` copies the part of `s1` defined by the range `r1` through `r2` into `s2`, starting at `r3`. The following rules must be enforced by the programmer: `r1 ≤ r2`, `r1 ≥ 0`, `r2 ≤ strlen(s1)`, and `r2 - r1 ≤ strlen(s2).`
**wstrip(s1, s2)**

**arguments:**
- `char *s1` the source string
- `char *s2` the destination string

**effects:** modifies s2

**returns:** void

**notes:** Wstrip() removes white space from the head and tail of s1, copying it into s2. S2 must be large enough to hold the new string.

**strcenter(s1, s2, l)**

**arguments:**
- `char *s1` the source string
- `char *s2` the destination string
- `int l` the length s2 will assume

**effects:** modifies s2

**returns:** void

**notes:** Strcenter() centers s1 in a field of length l. The result is copied into s2. If s1 is bigger than l, s1 is truncated.

**hrarifyfgets(s, l, f)**

**arguments:**
- `char *s` a buffer to read into
- `int l` the length of the buffer
- `FILE *f` the file to read from

**effects:** modifies s

**returns:** `char*`: NULL on error or end-of-file, s otherwise

**notes:** Harryfgets() is a special version of the standard C function fgets(). It reads from f until l characters have been read, or a new-line or end-of-file is encountered. The input is copied into s. S is then scanned to its end and if there is a terminating new-line, it is removed. This behavior permits data fields to be stored on disk and separated by new-lines. The new-lines are only separators, and are not taken literally. Typically a special sequence such as "\n" is used to indicate literal new-lines.
Memory Functions:

init_bytetal()  
arguments: none  
effects: modifies _BYTETOTAL  
returns: void  
notes: Init_bytetal() initializes the variable _BYTETOTAL to zero. _BYTETOTAL is used to maintain a count of the number of bytes allocated by harryalloc(). Note that since _BYTETOTAL is an unsigned integer, it can only keep track of the bytes allocated modulus $2^{16} = 65536$ bytes.

harryalloc(a, b)  
arguments: unsigned a number of elements  
unsigned b size in bytes of each element  
effects: allocates a*b bytes, modifies _BYTETOTAL  
returns: char* a pointer to the memory allocated.  
notes: Harryalloc() is a special version of the standard C function calloc(). If calloc() is successful in allocating a*b bytes of memory, harryalloc() increments _BYTETOTAL by a*b, and returns a pointer to the memory. The pointer must be cast into a pointer to the appropriate datatype. If there is not enough memory for calloc() to allocate, harryalloc() beeps three times and prints the message "fatal ERROR -- out of memory ... please report to vendor". In either case, if the global variable DEBUG is 1, harryalloc() prints a summary of the amount of memory thus far allocated. Note that init_bytetal() should be called before harryalloc() is first used.

dtof(d, f)  
arguments: double d a number in the double format  
float *f a pointer to a float  
effects: modifies *f  
returns: BOOL TRUE if conversion is successful, FALSE otherwise  
notes:Dtof() converts d into a float and stores the float in the location pointed to by f.
**DTOI(d, i)**

**arguments:**
- double *d* a number in the double format
- int *i* a pointer to an integer

**effects:** modifies *i

**returns:**
- BOOL TRUE if conversion is successful, FALSE otherwise

**notes:** Dtoi() converts d into an integer (dropping all decimal places) and stores the integer in the location pointed to by i.

**ITOD(i, d)**

**arguments:**
- int *i* a number in the integer format
- double *d* a pointer to a double

**effects:** modifies *d

**returns:**
- BOOL TRUE if conversion is successful, FALSE otherwise

**notes:** Itod() converts i into a double (padding with 0's after the decimal point) and stores the double in the location pointed to by d.

**PINM(f, d)**

**arguments:**
- char *f* a ptr to a memory location containing a ptr
- char *d* a pointer to a storage location

**effects:** modifies *d

**returns:** void

**notes:** Pinm() moves the pointer located at *f* to the location pointed to by *d*. This is very useful for storing pointers in memory declared to contain other primitive C data types. For example: if f is a function returning an integer, and p is a variable declared as int (*p)() (ie. a pointer to a function returning an integer), and n is a double, the commands:

```c
p = f;
pinm(&p, &d);
```

will store the function's address in the double. Note that the assignment of c to p is essential since C does not recognize statements like &f. Furthermore, it is critical that the location pointed to by d is large enough to hold a pointer. The programmer must insure this.
**iswap(x, y)**

*arguments:* int *x* a pointer to an integer
int *y* a pointer to an integer

*effects:* modifies x and y

*returns:* void

*notes:* Iswap() makes x point to the position y originally pointed to, and makes y point to the position x originally pointed to.

**swap_ptrrs(px, py)**

*arguments:* int **px** a pointer to a pointer
int **py** a pointer to a pointer

*effects:* modifies px and py

*returns:* void

*notes:* Swap_ptrrs() makes px point to the position py originally pointed to and makes py point to the position px originally pointed to. Although px and py are declared as pointers to pointers to integers, the procedure is intended to work with pointers to pointers of any type.

**User Input Functions:**

**pause()**

*arguments:* none

*effects:* clears the type-ahead buffer and reads one keystroke

*returns:* void

*notes:* Pause() simply waits for a key to be pressed. If the key is ctrl-c, the program is immediately exited. It is typically used to allow the user to read a partial message before the remainder is displayed.

**flash_pause(s, w)**

*arguments:* char *s* a prompt to display
WINDOWP w the prompt window

*effects:* clears the type-ahead buffer and reads one keystroke

*returns:* void

*notes:* Flash_pause() flashes s in window w until a key stroke occurs. If the key is ctrl-c, the program is immediately exited.
Math Functions:

**round(x)**

- **arguments:** float x the number to round off
- **effects:** none
- **returns:** int: x rounded off
- **notes:** Round() returns x rounded to the nearest integer.

**lround(x)**

- **arguments:** float x the number to rounded off
- **effects:** none
- **returns:** long: x rounded off
- **notes:** Lround() returns x rounded to the nearest long integer. It is identical to round() except that it returns a long integer as opposed to a regular integer. This is a convenience feature.

**ipr(x, p)**

- **arguments:** double x the base int p the power
- **effects:** none
- **returns:** double: x to the p\textsuperscript{th} power
- **notes:** Ipr() simply raises x to the p. It does so recursively.

**max(x, y)**

- **arguments:** any x a number any y a number
- **effects:** none
- **returns:** any
- **notes:** Max() returns x or y whichever is greater. It is defined as a macro so it should not be declared in a program. x and y can be numeric variables of any type.
\textbf{max3}(x, y, z)

\begin{itemize}
\item \textbf{arguments:} float x any number
\item float y any number
\item float z any number
\item \textbf{effects:} none
\item \textbf{returns:} double: see below
\item \textbf{notes:} Max3() returns the maximum of x, y, and z.
\end{itemize}

\textbf{min}(x, y)

\begin{itemize}
\item \textbf{arguments:} any x a number
\item any y a number
\item \textbf{effects:} none
\item \textbf{returns:} any
\item \textbf{notes:} Min() returns x or y whichever is lesser. It is defined as a macro so it should not be declared in a program. x and y can be numeric variables of any type.
4.2 SET COMMAND REFERENCE

This section describes the procedures used to implement sets. They are grouped into the file \src\set.c which is known as the SET module. The discussion is divided into two sections: a description of the set functions and a description of the functions internal to SET. The former consists of the following functional categories:

- Initialization
- Manipulation

The section on internal functions is intended only for reference - they are for SET's use only and should not be called by outside modules.

Initialization:

init_set(s,l)

arguments:  SETP    s
            int     l

effects:  modifies *s, may allocate memory (see below)

returns:  BOOL:    TRUE if allocation done, FALSE otherwise

notes:  Init_set() must be called before set s is used.
         Init_set() initializes s as a set of size l with no elements. If s is NULL space is allocated for it. If s has already been allocated, to a size equal or larger than l, it is resized to l. If s has already been allocated to a size less than l, the old memory is deallocated, and new memory of size l is allocated.
Manipulation:

add_set(s, p)

arguments: SETP s the set to add to int *p the element to add
effects: modifies *s
returns: BOOL: FALSE if there is no room left in s, TRUE otherwise
notes: Add_set() attempts to add p to s. If p already exists in s, there is no effect.

remove_set(s, p)

arguments: SETP s the set to remove from int *p the element to remove
effects: modifies *s
returns: BOOL: FALSE if p is not in s, TRUE otherwise
notes: Remove_set() scans s looking for p. If p is found, it is removed from s. Otherwise there is no effect.

sort_set(s, i)

arguments: SETP s the set to sort int (*i)() a ptr to the indexing function (see below)
effects: modifies s
returns: void
notes: Sort_set() sorts s based on the indexing function i. The indexing function is a function with one argument. The argument is a pointer to an integer (i.e. an element of the set). The function is applied to each element of the set and returns an integer. The value returned is used to determine the ordering of the set. The set is ordered in decreasing order.
search_set(s, p, i, ok)

arguments: SETP s the set to search
int p the index to search for
int (*i)() the indexing function
BOOL *ok

effects: modifies *ok
returns: int* a pointer to the first element of index p or NULL

notes: Search_set() applies the indexing procedure i to each element of s until a match is found. If a match is found, *ok is set to TRUE and a pointer to the matching element is returned. If a match is not found, ok is set to FALSE and NULL is returned. The indexing function must accept a pointer to an integer as an argument and return an integer index.

match_sets(key, acc, extr, in, out, arr, size)

arguments: char *key the key word to match against
int acc the number of characters that must match
int (*extr)() the indexing function
SETP in the input set
SETP out the output set
int **arr a buffer to put in into
int size the size of in

effects: modifies *arr and *out
returns: BOOL:

notes: Match_sets() is similar to search_set() in that it extracts certain elements from the input set. The elements extracted are put in the output set for further work. Match_sets() iterates over the elements of in, applying extr() to each one in turn. Extr() must except two arguments: an element and a pointer to a character buffer. It converts the element to an index which is placed in the buffer. Match_sets() then compares the buffer to key. If they match in the first acc positions, the element is accepted an added to out.

iter_set(s, a)

arguments: SETP s the set to iterate over
void (*a)() a pointer to a function (see below)

effects: none
returns: void

notes: Iter_set() applies the procedure a to every element of s. The procedure is a function with one argument. The argument is a pointer to an integer (i.e. an element of the set). The procedure must return void.
set_to_array(s, array, size)

arguments:  SETP    s  the set to convert
            int     **array the memory buffer to use
            int     size the buffer's maximum size

effects:    modifies *array
returns:    void
notes:      Set_to_array() fills array with the elements of s.
            Array is an array of pointers to integers. It is
            terminated with NULL. This representation is often
            more convenient to use than that of a set. Note that
            s is not affected by this translation.

set_size(s)

arguments:  SETP    s  a set

effects:    none
returns:    int: the number of elements in the set
notes:      Set_size() returns the number of elements currently in
            the set.

clear_set(s)

arguments:  SETP    s  the set to clear

effects:    modifies *s
returns:    void
notes:      Clear_set() resets s so that it contains no elements.
            The size of s remains the same.

kill_set(s)

arguments:  SETP    s  the set to kill

effects:    modifies *s
returns:    void
notes:      Kill_set() frees the memory associated with the set. This
            is done only when a set is no longer needed.
Internal Functions:

**NOTE**: Internal functions are for use by SET only. They should not be called by outside modules.

---

**_quick_sort_indirect_**

**arguments:**
- int **s** a pointer to the element list of a set
- int n the number of elements under consideration
- int (*i)() a ptr to the indexing function

**effects:**
- modifies *s

**returns:**
- void

**notes:**
This procedure is used by sort_set(). It implements the standard recursive quick sort algorithm given on page 229 of *A Book on C* by Al Kelly and Ira Pohl.

---

**_find_pivot_**(s, n, pivot_ptr, i)

**arguments:**
- int **s** a pointer to the element list of a set
- int n the number of elements under consideration
- int *pivot_ptr* a pointer to the pivot position
- int (*i)() a ptr to the indexing function

**effects:**
- modifies *s

**returns:**
- BOOL: FALSE if s is already sorted, TRUE otherwise

**notes:**
This procedure is used by quick_sort_indirect().

---

**_partition_**(s, n, pivot, i)

**arguments:**
- int **s** a pointer to the element list of a set
- int n the number of elements under consideration
- int pivot an index to the pivot point
- int (*i)() a ptr to the indexing function

**effects:**
- modifies *s

**returns:**
- int: the number of elements to consider next

**notes:**
This procedure is used by quick_sort_indirect().
4.3 UTILITIES GLOSSARY

**Screen:** The screen is the computer's interactive output device.

**Physical Page:** A physical page is one instance of a frame buffer within a screen. For example, the IBM CGA monitor supports four physical pages. That is, four screenfuls of text can be stored in the CGA's memory. Of course only one physical page can be displayed at a time. The actively displayed physical page can be changed quickly and easily.
5.0 Facility Management System

Introduction:

The goal of the Facility Management System is to provide the managers of civil works with a powerful environment in which to study the effects of various maintenance policies. This is a particularly important goal in light of recent funding cutbacks. FMS allows managers to try different maintenance policies and compare them in terms of facility condition, agency cost, user cost, and scheduled and unscheduled delays. When it is completed, the optimization option will allow FMS to do numeric and/or heuristic analysis of the engineering conditions and recommend a best solution subject to specified constraints.

In order that the system can be used by all management levels, every feature completely supports defaults. A novice never needs to modify the defaults. Once a facility and a policy are defined, the simulation will run without further specification. All that is necessary for the advanced user to change policies is to modify the values of numerical coefficients. Such changes are made directly from FMS without using text editor.
The structure of the system itself is completely general so that most facility types can be modeled with relatively minor changes to the program. Test implementations of FMS have demonstrated its use with lock and highway facilities.

The Facility Management System is an application built upon DISPLAY and FORMS. As such its code is divided into three distinct categories. The first category defines the control structure. It uses FORMS to query the user for commands and data. The second category consists of the data manipulators which embody the simulation's functionality. The final category Initializes the system. Disk input/output is considered part of this last category.

Since FMS is an application, it will be treated here slightly differently than in the modules of previous chapters. Rather than discussing individual procedures, I will introduce the system's organization as a whole, then discuss the particular ramifications of the design. FMS should serve as a working example of the use of FORMS and DISPLAY, and the programming methodology of a large system.

ORGANIZATION:

As discussed above, FMS is divided in three - initialization, control structure, and data manipulation. Upon invocation, FMS does all of its initialization. This topic is covered below in the section on Memory Management.
The control structure divides the user input into Command windows and object windows as described in chapter 2. FMS opens with a horizontal command window at the bottom of the screen. The window provides the following options: **edit**, **run**, **report**, and **quit**. (Of course since FMS is built on top of FORMS, the general help menu is always available through the help key\(^1\).) These commands comprise all the fundamental abilities of FMS. Moving the cursor over any one of them and pressing return executes the related command. The commands continue in this manner, eventually prompting for the name of a facility or policy. The selected command is then applied to this object. In the case of **edit**, the object is displayed in a window and its data is available for user modification. If the object is a facility, further command options appended to the object allow the user to view and modify the facility's history record, deterioration model, and traffic model (see figure 2.1). Similarly, commands imbedded in a policy object allow the user access to the policy's routine maintenance fields. Figure 5.0 summarizes the FMS command structure.

---

\(^1\) The default help key is control X.
Internally, FMS is tied together by a global data structure named \( S \). \( S \) is a variable of type \texttt{STATE}. It provides hooks to all the relevant internal data. In particular, the state maintains a pointer to the window manager's data structure, memory set aside to hold the most recently activated facility and policy, and temporary storage used by
the prediction and optimization modules. Other global variables allow all FMS procedures to access the required windows and forms.

Memory Management:

FMS is written in C. Unfortunately, the C standard does not mention memory management. Primitive C functions such as alloc() and calloc() allow blocks of memory to be reserved for use by the program, but there is no way of deallocating memory once it has been allocated. This is a major disadvantage when working with data that changes size. For the present, there is only one solution. All memory blocks are allocated to the largest size they will ever assume. This solution presents problems since it forces the programmer to determine the block size before compilation. As a result, the user is incapable of exceeding certain limits. For example, the prediction model assumes that the maximum number of years of prediction is the global equate MAXHORIZON\(^2\). The user has no way of predicting over more than MAXHORIZON years.

It is highly recommended that any future work conforms to the FMS standard of doing all allocation before any other part of the system is called. This is easily accomplished with a simple initialization routine called from main() and results in easier debugging.

---

\(^2\) MAXHORIZON is currently set at 50 years, although it may be upped to 100 years once the systems memory needs are under control.
An associated problem, although less serious, is the IBM PC's use of stack space. The stack is of fixed length which again must be set before compilation. The main effect of this problem is to limit the depth to which procedures can be nested. This is particularly important when working with recursive procedures. The best way to avoid the problem is to limit the number of arguments procedures take, and the number of automatic variables they declare. In particular it is important not to use strings declared as automatic, since they use stack space proportional to their length. Instead declare them as static local variables\(^3\). This places them in the system heap rather than on the stack.

**File Structure:**

Due to memory limitations, FMS only has room to contain the data describing one facility and one policy. These are the facility and policy that were most recently asked for by the user. All other facilities and policies are stored on disk.

The data is stored in a simple sequential text format. Each field of the internal data structure appears on a line of text in the file. Ascii character 13 (new-line) separates the fields. The data is stored in text format rather than in the compacted machine readable format for

---

\(^3\) This is easily accomplished. Simply place the key word *static* in front of their normal automatic definition. Their value will be maintained between calls to the procedure, so be sure to initialize them within the procedure.
two reasons. First, text format is human readable, thus facilitating off line inspection and editing of the data. Second, the machine format is different for every computer, thus data stored in such a manner is non-portable.

Any line whose first character is a semi-colon is treated as a comment and ignored.

**Further Work:**

Of all of the files discussed here, FMS is the most incomplete. This is not surprising since it depends on the reliability of the other modules, and it promises to do the most. Rather than discuss all of the additions we have planned, I will simply enumerate some of the major ones.

- The ability to add new facilities to a network of established facilities must be added. Included in this is a strict definition of what a network is and how it will operate under FMS.

- A means to update a number of facilities on the same network (for example changing the scheduled down time of all facilities in a certain district) is essential.
• The optimization model has not yet been implemented.

• The report generator must be improved. Improvements include more flexibility, the use of graphics, the ability to produce hard copy, and compatibility with popular data base programs.

• The ability for the user to more flexibly view data is desirable. For example, a user might want to simultaneously view the deterioration models of two facilities. This not possible in the current implementation.
5.1 FMS COMMAND REFERENCE

This section describes the various modules that make up FMS. It is divided into the following functional groups and files:

**initialization:**
- FMS
- TEMPL
- SAVELOAD
  - initialization functions
  - form initialization
  - disk i/o

**commands:**
- EDIT
- MOD
- ADD
- RUN
- PREDICT
- OPT
- REPORT
- QUIT
  - edit a facility or policy
  - view and update a facility or policy
  - add a facility or policy
  - execute the pmodel or omodel
  - set up for prediction model
  - set up for optimization model
  - print results
  - save work and exit

**simulation models:**
- PMODEL
- OMODEL
  - the prediction model itself
  - the optimization model itself

All of these files are located in the `src` directory.
5.1.1 FMS COMMANDS

main()

arguments: none
effects: Does allocation and modifies most global variables.
returns: void
notes: Main() is the top level procedure of FMS. It calls all of the initialization functions, displays the title page, and calls the first menu.

make_facility()

arguments: none
effects: does allocation
returns: FACILITYP: a pointer to a blank facility
notes: Make_facility() simply allocates the memory needed to store a facility, and returns a pointer to the beginning of that memory.

edit_facility_history(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Edit_facility_det() first creates a box to display the current facility's history data in, then calls fill_object to display the data.

edit_facility_det(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Edit_facility_det() first creates a box to display the current facility's deterioration data in, then calls fill_object to display the data.
edit_facility_cap()

arguments: all arguments are ignored
effects: none
returns: void
notes: Edit_facility_cap() first creates a box to display the current facility's capacity data in, then calls fill_object to display the data.

init_input_line()

arguments: INPUT_LINEP l an input line to initialize int len the input line's maximum length
effects: allocates memory for an input line of length len.
         Modifies *l.
returns: void
notes: Init_input_line() initializes l for use as an instance of the INPUT_LINE data type. INPUT_LINE is used by a form for getting textual input from the user.

make_policy()

arguments: none
effects: does allocation
returns: POLICY: a pointer to a blank policy
notes: Make_policy() simply allocates the memory needed to store a policy, and returns a pointer to the beginning of that memory.

edit_sub_system_one routine(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Edit_g_routine() first creates a box to display the current policy's gate routine maintenance policy in, then calls fill_object to display the data.

edit_sub_system_two routine(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Edit_w_routine() first creates a box to display the current policy's wall routine maintenance policy in, then calls fill_object to display the data. Note: this routine is specific to FMS:canal. It is included here as an example only.
init_pred_dat()

arguments: none
effects: none
returns: void
notes: This procedure is reserved to initialize the prediction model database. It is not currently used.

init_run_dat()

arguments: none
effects: modifies *S.run_p
returns: void
notes: Init_run_dat() initializes the parameters controlling runs of pmodel and omodel. Things like the default planning horizon and base year are set here.

5.1.2 TEMPL COMMANDS

init_forms_system()

arguments: none
effects: all the forms and windows in the system
returns: void
notes: Init_forms_system() initializes the template system, loads in all form descriptions, calls the fix procedures (below), and declares all windows.

fix_facility_temp(o)

arguments: OBJ_TYPEP o the facility form
effects: *o
returns: void
notes: This function loads the facility form with procedure pointers.
fix_policy_temp(o)

arguments: OBJ_TYPEP o the policy form
effects: *o
returns: void
notes: This function loads the policy form with procedure pointers.

fix_top_menu(o)

arguments: OBJ_TYPEP o the form for the top menu
effects: *o
returns: void
notes: This function loads the top menu with procedure pointers.

fix_quit_menu(o)

arguments: OBJ_TYPEP o the form for the quit menu
effects: *o
returns: void
notes: This function loads the quit menu with procedure pointers.

fix_add_mod_menu(o)

arguments: OBJ_TYPEP o the form for the add_mod menu
effects: *o
returns: void
notes: This function loads the add_mod menu with procedure pointers.

fix_what_menu(o)

arguments: OBJ_TYPEP o the form for the what menu
effects: *o
returns: void
notes: This function loads the what menu with procedure pointers.
fix_pred_opt_menu(o)

arguments: OBJ_TYPEP o the form for the pred_opt menu
effects: *o
returns: void
notes: This function loads the pred_opt menu with procedure points.

5.1.3 SAVELOAD COMMANDS

find_facility(s)

arguments: char *s the name of the file containing the facility
effects: modifies S.net->thefac, S.net->facdown and S.net->facup
returns: FACILITYP: a pointer to the facility, or NULL on error
notes: Find_facility() opens the file named s. If s does not contain path information, the path \fac\ on the default drive is used. The facility is loaded into S.net->thefac, and the next and previous facilities are loaded into S.net->facdown and S.net->facup respectively.

find_policy(s)

arguments: char *s the name of the file containing the policy
returns: POLICYP: a pointer to the policy or NULL on error
notes: Find_policy() opens the the file named s. If s does not contain path information, the path \pol\ on the default drive is used. The policy is loaded into S.net->thepolicy.
load_facility(l, f)

arguments: FACILITYP l a pointer to the facility to load
            FILE *f the file to read from
effects: modifies S.net->thefac, S.net->facdown, S.net->facup,
         S.net->the_fac_file and S.net->dirtyfac
returns: BOOL: always returns TRUE
notes: Load_facility() is used by find_facility(). It takes care of reading facility information in from a file.

load_policy(m, f)

arguments: POLICYP m a pointer to the policy to load
            FILE *f the file to read from
effects: modifies S.net->thepolicy, S.net->the_pol_file and
         S.net->dirtypol
returns: BOOL: always returns TRUE
notes: Load_policy() is used by find_policy(). It takes care of reading policy information in from a file.

loadroutine_policy(r, f)

arguments: ROUTPOLICYP r a pointer to the routine maint to load
            FILE *f a pointer to the file to read from
effects: modifies S.net->thepolicy, S.net->the_pol_file and
         S.net->dirtypol
returns: BOOL: always returns TRUE
notes: Load_routine_policy() is used by load_policy(). It takes care of reading policy information pertaining to routine maintenance in from a file.

save_mods(k, l, o, w)

arguments: all arguments are ignored
effects: modifies S.net->dirtyfac, S.net->dirtypol, S.net->
         the_pol_file, and S.net->the_fac_file
returns: void
notes: Save_mods() is used to flush the contents of S.net-
         >thefac, S.net->facdown, S.net->facup, and S.net-
         >thepolicy back to their files of origin. It is called from quit_proc(). It is not currently implemented.
5.1.4 EDIT COMMANDS

\texttt{edit(k, l, o, w)}

\begin{itemize}
  \item \textbf{arguments:} all arguments are ignored
  \item \textbf{effects:} none
  \item \textbf{returns:} void
  \item \textbf{notes:} This function displays the add_mod menu.
\end{itemize}

5.1.5 MOD COMMANDS

\texttt{mod(k, l, o, w)}

\begin{itemize}
  \item \textbf{arguments:} all arguments are ignored
  \item \textbf{effects:} none
  \item \textbf{returns:} void
  \item \textbf{notes:} This function calls the what menu to determine if a facility or a policy is desired, then again to find out which facility or policy in particular. \texttt{Find_facility()} or \texttt{Find_policy()} is then called, and if it returns successfully, \texttt{fac_temp or pol_temp} is used to display the newly loaded data.
\end{itemize}

\texttt{select\_facility\_obj(k, l, o, w)}

\begin{itemize}
  \item \textbf{arguments:} all arguments are ignored
  \item \textbf{effects:} none
  \item \textbf{returns:} void
  \item \textbf{notes:} This procedure is used by the what menu to indicate that a facility has been chosen.
\end{itemize}

\texttt{select\_policy\_obj()}

\begin{itemize}
  \item \textbf{arguments:} all arguments are ignored
  \item \textbf{effects:} none
  \item \textbf{returns:} void
  \item \textbf{notes:} This procedure is used by the what menu to indicate that a policy has been chosen.
5.1.6 ADD COMMANDS

add(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: This procedure will allow facilities to be added to the network. It is currently unimplemented.

stub(s, w)

arguments: char *s a warning message
WINDOWP w the stub window
effects: *w
returns: void
notes: The only function of this procedure is to display the warning message and return. It creates w and displays s in it. It prompts the user for a key press upon which it returns. The warning message usually has something to do with the fact that a function that would normally execute has not yet been defined.

5.1.7 RUN COMMANDS

run(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: This function displays the pred_opt menu.

set_run_params(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: This function displays the run_temp menu.
5.1.8 PREDICT COMMANDS

pred(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: This function is used to invoke the prediction model. It calls the what menu once to determine which facility desired, then again to determine which policy is desired. Find_facility() and Find_policy() are then called, and if they return successfully, prediction_model() is called to simulate the facility's aging.

5.1.9 OPT COMMANDS

opt(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Opt() is used to invoke the optimization routines. They are not currently implemented.
5.1.10 REPORT COMMANDS

rep_head(w, p)

arguments: WINDOW w the window to print in
           int p the page number
effects: none
returns: void
notes: Rep_head() prints the report header. p is the page
       number of the report and not the page number of the
       display screen (see DISPLAY). w is currently ignored.

report(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Report() calls the various functions which present the
       user with a report of the latest prediction or
       optimization results.

cond_rep(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Prints the condition report.

serv_rep(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Prints the service report.

repair_rep(k, l, o, w)

arguments: all arguments are ignored
effects: none
returns: void
notes: Prints the repair report.
total_repl(k, l, o, w)

  arguments: all arguments are ignored
  effects: none
  returns: void
  notes: Prints the total cost summary.

gain(x)

  arguments: float x any number
  effects: none
  returns: float: see below
  notes: Gain() calculates the function:

\[
\frac{x \cdot \text{discount rate} \cdot (1 + \text{discount rate})^{\text{horizon}}}{(1 + \text{discount rate})^{\text{horizon}} - 1}
\]

5.1.11 QUIT COMMANDS

quit_proc(k, l, o, w)

  arguments: all arguments are ignored
  effects: none
  returns: void
  notes: This function displays the quit_temp menu.

getout(k, l, o, w)

  arguments: all arguments are ignored
  effects: see below
  returns: void
  notes: Getout() clears the screen and exits to DOS. The exit code is 0.
5.1.12 PMODEL COMMANDS

**prediction_model(ld, p, w)**

**arguments:**
- FACILITYP ld the facility to use
- POLICYP p the policy to use
- WINDOWP w the window to use

**effects:** modifies *w, S.pred...

**returns:** void

**notes:** This is the main procedure which calls all of the other prediction procedures in order. After calling each procedure a period is printed to the window to facilitate user feedback.

**traffic_pred(ld)**

**arguments:**
- FACILITYP ld the facility to use

**effects:** modifies S.pred...

**returns:** void

**notes:** Traffic_pred() calculates the traffic flow over the facility.

**efficiency(ef)**

**arguments:**
- float *ef an array of efficiency values

**effects:** modifies S.pred...

**returns:** void

**notes:** Efficiency() calculates the efficiency rating over the facility.

**condition_index(ld, p, sw, ef)**

**arguments:**
- FACILITYP ld the facility to use
- POLICYP p the policy to use
- int sw the sub-system switch
- float *ef an array of efficiency ratings

**effects:** modifies S.pred...

**returns:** void

**notes:** Condition_index() predicts the condition of a sub-system. The sub-system to use is selected with the sw switch. The switch values will be different for different facilities. This procedure also calculates the time and intensity of any maintenance activities.
prob_failure(p, eff)

arguments: POLICYYP p the policy to use
            float *eff an array of efficiency ratings
effects: modifies S.pred...
returns: void
notes: Prob_failure() calculates the probability that a
        facility will fail.

service_rate(ld)

arguments: FACILITYP ld
effects: modifies S.pred...
returns: void
notes: Service_rate() calculates a facilities service rate.

rout_policy(p)

arguments: POLICYYP p the policy to update
effects: modifies S.pred...
returns: void
notes: Rout_policy() converts the routine maintenance
       schedule as input by the user into a representation
       for use by prediction_model().

sigma(ld, sw)

arguments: FACILITYP ld the facility to use
            int sw the sub-system switch
effects: modifies S.pred...
returns: void
notes: Sigma() calculates sigma for the sub-system specified
       by sw.

s_sigma()

arguments: none
effects: modifies S.pred...
returns: void
notes: S_sigma() calculates the s_sigma function for the
       facility in question.
**sub_system_one_sched_policy**(ld, eff)

*arguments:* FACILITYP ld the facility to use  
float *eff an array of efficiency ratings

*effects:* modifies S.pred...
*returns:* void
*notes:* This procedure calculates the scheduled maintenance policy for ld sub system 1.

**sub_system_two_sched_policy()**

*arguments:* FACILITYP ld the facility to use  
float *eff an array of efficiency ratings

*effects:* modifies S.pred...
*returns:* void
*notes:* This procedure calculates the scheduled maintenance policy for ld sub system 2.

**sub_system_three_sched_policy_cost()**

*arguments:* FACILITYP ld the facility to use  
float *eff an array of efficiency ratings

*effects:* modifies S.pred...
*returns:* void
*notes:* This procedure calculates the scheduled maintenance policy for ld sub system 3.

**sub_system_one_unsched_policy_cost(ld)**

*arguments:* FACILITYP ld the facility to use
*effects:* modifies S.pred...
*returns:* void
*notes:* This procedure calculates the necessary unscheduled maintenance level for ld sub system 1.

**sub_system_two_unsched_policy_cost(ld)**

*arguments:* FACILITYP ld the facility to use
*effects:* modifies S.pred...
*returns:* void
*notes:* This procedure calculates the necessary unscheduled maintenance level for ld sub system 2.
sub_system_three_unsched_policy_cost(ld)

arguments: FACILITYP ld the facility to use
effects: modifies S.pred...
returns: void
notes: This procedure calculates the necessary unscheduled maintenance level for ld sub system 3.

routine_policy_cost(p)

arguments: POLICYP p the policy to use
effects: modifies S.pred...
returns: void
notes: This procedure calculates the cost of routine maintenance for policy p.

operating_cost(ld)

arguments: FACILITYP ld the facility to use
effects: modifies S.pred...
returns: void
notes: This procedure calculates the cost of operating facility ld.

down_time_cost(ld)

arguments: FACILITYP ld the facility to use
effects: modifies S.pred...
returns: void
notes: This procedure calculates the down time cost of facility ld.

time_delay_cost(ld)

arguments: FACILITYP ld the facility to use
effects: modifies S.pred...
returns: void
notes: This procedure calculates the delay time cost to users of facility ld.

agency_user_cost(ld)

arguments: FACILITYP ld the facility to use
effects: modifies S.pred...
returns: void
notes: This procedure calculates the sum costs to the management agency and the users of facility ld.
5.1.13 OMODEL COMMANDS

(The Optimization Model is not yet implemented.)
APPENDIX A: COMPUTING ENVIRONMENT

(DRAFT)

This section is intended to guide the programer who will update and maintain the Facility Management System and any of its descendents. As such it is directed at experienced programers, who are willing to bridge the gap between assembly language and C. FMS is a very difficult project to describe, since it encompasses at least three major headings.

Goals:

The goals in creating FMS are as follows:

- Programer: Generality, extensibility, maintainability
- User: Robust, Fast, continuity, help available, functional

Program History:

FMS was developed between January 1986 and June 1987. From the outset it was designed to support both FMS-ROADS and FMS-LOCKS, for the Federal Highway Commission and the Army Corp of Engineers respectively. The technical foundation for the actual Civil Engineering part was
developed separately by Mr. Markow and his associates. While the technical aspects of it were researched, I designed DISPLAY. Once its prototype was completed, work began on FORMS (which assumed that control over the screen is already present). FORMS required quite a bit of memory cheating, and therefore quite a bit of debugging. Once it was roughed out, and its interface to DISPLAY was finalized, work began on the command structure of FMS-LOCKS. By this time the spring demo for CERL had occurred, and suggestions for faster more distinct windows were heard. Work continued, slowly fleshing out the details of the locks. Actual prediction code was available no sooner than two weeks before the summer demo. It was luckily one of the easiest procedures. Since then most work has gone into converting form C86 to Microsoft C v4.0 for memory considerations.

Programming Methodology:

An attempt has been made to divide the program according to functionality into modules. Primarily, DISPLAY and FORMS are well contained. Most programs wishing to use them must simply include their equates files. Almost every program must also use the main equates file which provides information of general importance.

(and so on)

Discussion Of C:
FMS is implemented in C. For better or worse. C is glorified assembly language (PDP-11 assembly language to be exact). This is good for speed, but provides many programming headaches. Especially, the retarded type checking facilities make intelligent programming difficult by letting mistakes slip by while disallowing many logical conversions. Memory management (or the lack thereof) is the absolute worst part. The lack of a paging apparatus or at least a means of reclaiming deallocated memory results in even minor applications encountering memory limits. C is relatively fast though, and provides significant data type advantages over fortran. C also beats Pascal on the basis of speed and modularity. This program would be impossible in Pascal. Perhaps Modula 2 is the answer, but it is not currently a well understood language. C is well supported C is also good for portability.

Programming Environment:

This program was developed for distribution to various government agencies employing general office computers. Thus the IBM PC's 8086 standard was adopted for its generality and common use. It is certainly a completely backward computer, employing an ancient operating system. Due to DOS, the entire memory available is restricted to a maximum of 640k. This is a severe limit when you consider that approximately 100K is reserved for the operating system and 200K is used for the program itself.

For running the program disk must be set up as follows:
The following discussion applies to the development computer (since it is likely the system you are using). We programmed the system on a Compaq 286 portable. It is a convenient size for limited portage (for demos and the like) and supports a 20 meg hard disk, 640 whopping K bytes of memory and a reasonably fast 8 MHz Processor 80287. The operating system is MicroSoft DOS v3.1, and the compiler is MicroSoft C v4.0. This development kit also includes an overriding linker and a symbolic debugger. The debugger will be your savior, learn it well. I also use a word processor called Final Word. It conforms to emacs key bindings, allows scribe like formatting commands (no this document wasn't prepared with it) and includes a wonderful spelling checker.

The file system of the Compaq looks like this:

<diagram of file system>

explain the batch files here