User Input Routine
External Reference Specification

Lou Infeld

04/10/85

TABLE OF CONTENTS

1. Introduction

Most Applications at one point or another require that the user key in some textual information. In the past there has been little standardization in the way that an "Input Routine" interfaces with the user. Pascal and Basic each have different user input conventions. In fact they completely contradict each other; a user has to completely relearn how to interact with either language after using the other. Many applications use the "Input Routine" built into the language environment being used. Other applications use independently developed "Input Routines" which are more sophisticated and user friendly. However, the poor user of several applications has different interfaces to contend with, each with its own standards and idiosyncrasies.

To try to solve these problems, Apple Computer has published several documents encouraging "standard" design guidelines including how an "Input Routine" should look and behave. Now this "User Input Routine" is being made available to Apple // developers. It encorporates all the standards proposed by Apple Computer and is available for the following environments:

Apple // Assembler	(with or without Console Driver)
Apple // AppleSoft	(with or without Console Driver)
Apple // Pascal	(with Console Driver)

2. General Description

The "User Input Routine" attempts to fulfill the standards published in the "Apple //e Design Guidelines" manual (pp 24-37), Bruce Tognazzini's memo "UPDATE: Human Interface Design Guidelines" dated August 9, 1983, as well as de facto standards used in the popular AppleWorks program.

The "User Input Routine" is called by the application with a string variable containing a default (can be null) as well as the maximum number of characters that will fit in the string variable. A "string variable" is basically a buffer in which the first byte contains the "length" of the string. The following bytes are the actual characters in the string.

The "User Input Routine" will display a field on the screen consisting of the default string followed by a series of "fill" characters. A cursor will be visible to the right of the default string. The cursor is the "Insert Cursor" as described in the Guidelines. When this cursor is present, typing any printing character will place that character in the field at the current cursor position. All characters in the field to the right of the cursor are shifted one position. If the user presses the CONTROL key and "E" together, the "Replace Cursor" appears. When this cursor is present, typing any printing character will place that character in the field replacing the current character under the cursor. Pressing the CONTROL key and "E" again will return the "Insert Cursor".

The user can edit the field by adding or replacing characters or by using editing commands. When the user is satisfied with the string in the field, he presses the RETURN key. This will terminate the "User Input Routine" and return control back to the application. The user's response will be in the string variable specified when the "User Input Routine" was called.

If the application specifies a string variable that can contain more characters than the width of the field, the "User Input Routine" will retain characters that "fall off" the right edge of the field. These characters will "reappear" if characters in the field are deleted.

The following editing commands are supported:

LEFT-ARROW	Moves cursor left within field
RIGHT-ARROW	Moves cursor right within field
CONTROL-D	Deletes character to the left of the cursor
DELETE	Deletes character to the left of the cursor
CONTROL-F	Deletes character under the cursor (Forward Delete)
CONTROL-E	Toggle between insert and delete cursors
CONTROL-X	Deletes all characters in the field
CONTROL-Y	Deletes all characters from present cursor position to end of field (including characters saved by insert)
CONTROL-Z	Restores default string

3. Customization and Advanced Uses

In general, the "User Input Routine" will behave as described in section 2. However, the "User Input Routine" can be customized to the particular needs of the application. A structure called the Information Block is used as a conduit between the application and the "User Input Routine". The application tells the "User Input Routine" how to react to the user's keystrokes and conversely the "User Input Routine" tells the application all about its current status.

If a viewport (window) has been defined, the "User Input Routine" will respect it with the one restriction: the last two positions in the window can not be included in the input field. This restriction is necessary to eliminate scrolling and wrapping problems. A field as large as 254 characters can be specified.

Normally, when the RETURN key or the ESCAPE key is pressed, the "User Input Routine" will terminate with the Input String set to the characters currently in the field on the screen (without the fill characters). However, other terminating characters can be configured to cause termination instead or in addition to RETURN and ESCAPE. Also the "User Input Routine" can be interrupted rather than terminated. In this case, when the "User Input Routine" is called again, it continues in the state it was in when it was interrupted (assuming the application program has not changed any parameters in the Information Block). This feature is useful for a help facility. A help character (e.g. Open Apple-?) can be configured to interrupt the "User Input Routine" for a help message in the middle of editing.

Up to 20 characters can be specified as termination characters. For each termination character, the application can specify whether the Open Apple or Solid Apple key must be pressed with the character. Additionally, for each termination character, the application can specify whether to completely terminate the "User Input Routine" or just "interrupt" it temporarily.

An "immediate" mode is optionally available that allows the application to constantly gain control during the input process. This feature can be used by the application, for instance, to update a clock display, check for mouse movements or run in demonstration mode.

4. Information Block

4.1 Format

The Information Block is divided into three logical sections: General Information, Termination Information, and Internal Information.

```
max_terms      equ     20      ;Maximum number of terminators

Input_Info     equ     *
;                               General Information
;                               -------------------
width          db      0       ;Width of the field on the screen
fill_char      db      " "     ;Fill character
mouse_fill     db      0       ;0-use "fill_char" as fill character
                               ;1-use MouseText ghost underline
cursor         db      0       ;current cursor being used
                                 ;0-insert cursor
                                 ;1-replacement cursor
control        db      0       ;0-Control chars will be ignored
                               ;1-Control chars allowed as input
beep           db      0       ;0-errors will not be beeped
                               ;1-errors will be beeped
immediate      db      0       ;0-calling routine gets control after the
                               ;  complete input is keyed in by user
                               ;1-calling routine gets control after each
                               ;  keypress check
entry_type     db      0       ;Indicates type of entry into routine
                               ;0-initial entry
                               ;1-interrupt re-entry
                               ;2-immediate re-entry
bord_ch        db      0       ;char to blink outside of field

;                               Termination Information
;                               -----------------------
exit_type      db      0       ;Indicates which termination condition
                               ;  occurred
                               ;0-not terminated yet
                               ;not 0-index into terminating char list
last_event     db      0       ;last event type (not used)
last_ch        db      0       ;character user keyed in
last_mod       db      0       ;keypress modifier
n_chars        db      0       ;Number of terminator chars currently
                               ;  defined

                               ;The next 3 items define what keystrokes
                               ;  will terminate or interrupt the routine.

char_list      ds      max_terms ;Chars which will terminate input
mod_list       ds      max_terms ;Modifiers for each char in"char_list"
                                       ;0-none
                                       ;1-Open Apple
                                       ;2-Solid Apple
                                       ;3-Either Open or Solid Apple
term_list      ds      max_terms ;Termination types for each char in
```

```
                                           ;  "char_list"
                                              ;0-terminate input
                                              ;1-interrupt input

;                                        Internal Information
;                                        --------------------

origin_x        db      0       ;x coordinate of start of field
origin_y        db      0       ;y coordinate of start of field
cursor_x        db      0       ;x coordinate of cursor in field
cursor_y        db      0       ;y coordinate of cursor in field
cursor_pos      db      0       ;position     of cursor in field (1..width)
input_length    db      0       ;length of Input String (incl invisib part)
slow_blink      dw      0       ;slow blink rate
fast_blink      dw      0       ;fast blink rate
```

4.2 Description

4.2.1 General Information section

4.2.1.1 "width"

This parameter tells the "User Input Routine" how wide to make the field on the screen. When the "User Input Routine" is called, it displays the default value in the Input String on the screen at the current cursor position. If there is any room left in the field on the screen, fill characters are displayed. The parameter "fill_char" is used as the fill character. The number of fill characters displayed is "width" minus "length of Input String". "Width" is initially 254 characters.

If the value of "width" is greater than the number of character positions from the start of the field to the end of the window -2, the "User Input Routine" will reduce "width" accordingly.

4.2.1.2 "fill char"

This is the fill character that is used in the field. "Fill_char" is initially the blank character.

4.2.1.3 "mouse fill"

If this parameter is 1, the MouseText ghost underline is used as the fill character. If "mouse_fill" is 0 the character in "fill_char" will be used as described above. "Mouse_fill" is initially 0.

It is the application's responsibility to determine whether MouseText is available in ROM before using this option. The following algorithm can be used to determine whether MouseText is available:

```
If memory location $FBB3 contains $06
                AND
memory location $FBC0 does not contain $EA

then MouseText is available
```

4.2.1.4 "cursor"

This parameter represents the current cursor type being used. If it is 0, the "Insert Cursor" is in effect. If it is 1, the "Replace Cursor" is in effect. If the user presses CONTROL and "E", this parameter changes value. The application can force the "User Input Routine" to start with either of the cursor types by setting "cursor" accordingly before calling the routine. "Cursor" is initially 0.

4.2.1.5 "control"

If this parameter is 1, control characters (ASCII values less than 32) are allowed as input from the keyboard. To insert a control character, the user must press the CONTROL key, the Open Apple key and the corresponding alphabetic key. The alphabetic character is obtained by added 64 to the ASCII value of the control that is desired. The actual value inserted in the string is the ASCII value + 128 which will appear on the screen as the inverse of the corresponding character. For example, to insert the Carriage Return character (ASCII 13), the user presses CONTROL, Open Apple and "M" (ASCII 77). The screen will show an inverse "M" and the string will contain the value 205 (77+128). If "control" is 0, control characters will not be allowed and will result in a beep. Note that editing characters and termination characters are not affected by the setting of "control". "Control" is initially 0.

4.2.1.6 "beep"

If this parameter is 1, any illegal keypresses will cause the "User Input Routine" to beep. If this parameter is 0, there will be no beeps. "Beep" is initially 1.

4.2.1.7 "immediate"

If this parameter is 1, the "User Input Routine" will return to the application program after each keypress check. When the application next calls the "User Input Routine", it will be considered an "immediate" re-entry. If this parameter is 0, the "User Input Routine" will return to the application program only after a termination character is pressed.

During "immediate" processing, the application can tell whether a key has been pressed by checking the parameter "last_key". If it is not 0, a key has been pressed and its ASCII value is in that parameter (its corresponding keypress modifier is in "last_mod"). When the "User Input Routine" is re-entered, it will check "last_key" and "last_mod". If there is a keystroke, it will "process" it, otherwise it will look for the next keystroke. The application can therefore "process" the keystroke before the "User Input Routine". At this point, the application can leave the keystroke intact and re-enter the "User Input Routine" which will also "process" the keystroke. Alternatively, the application can set "last_key" and "last_mod" to 0 which will cause the keystroke to be ignored by the "User Input Routine".

Applications using "immediate" mode have the additional responsibility to keep the cursor blinking at the correct rate. See the description of "slow_blink" and "fast_blink" for the necessary considerations.

"Immediate" is initially 0.

4.2.1.8 "entry_type"

This parameter tells the "User Input Routine" what type of entry is being made. If the value of "entry_type" is 0, this is an initial entry and a new field is established. If the value is 1, the routine assumes it is being re-entered after an interrupt termination. If the value is 2, the routine assumes it is being re-entered after "immediate" processing by the application. This parameter is managed by the "User Input Routine" and normally does not need to be changed by the application.

4.2.1.9 "bord_ch"

This character will be used by the "User Input Routine" as the blink character when the cursor is outside of the field. This condition occurs when the field is completely filled in. "Bord_ch" is initially blank.

4.2.2 Termination Information section

4.2.2.1 "exit type"

When the "User Input Routine" terminates, this parameter contains the "type" of termination that occurred. Termination characters are numbered from 1 to 20. "Exit_type" will contain the number of the termination character that caused the termination. If "exit_type" is 0, this indicates that the "User Input Routine" has not terminated yet (i.e."immediate" mode is in effect).

4.2.2.2 "last event"

This parameter is not currently used.

4.2.2.3 "last ch"

This parameter contains an ASCII value if the last keypress check sensed a keystroke or 0. It is useful for applications using the "immediate" mode of the "User Input Routine".

4.2.2.4 "last mod"

This parameter contains the keystroke modifier if a keystroke was sensed by the last keypress check. Otherwise it is 0. The possible values of "last_mod" are:

0 - no modifier pressed
1 - Open Apple key pressed together with key
2 - Solid Apple key pressed together with key
3 - Either Apple keys pressed together with key

4.2.2.5 "n chars"

This parameter is the number of termination characters that have been configured. "N_chars" is initially 2 (for RETURN and ESCAPE).

4.2.2.6 "char list"

"Char_list" is a 20 byte table containing the ASCII values of the configured termination characters. For the alphabetic characters "A" to "Z", only the upper case ASCII values need be in the table.

Only the first "n_chars" bytes are looked at by the "User Input Routine". The first 2 bytes in this list are initially 13 and 27 respectively (these values are the ASCII codes for RETURN and ESCAPE).

4.2.2.7 "mod list"

"Mod_list" is a 20 byte table which specifies what keystroke modifiers are needed for each termination character to be recognized. A value of 0 indicates that no modifiers can be pressed. A value of 1 indicates that the Open Apple key must be pressed together with the termination character. A value of 2 indicates that the Solid Apple key must be pressed. A value of 3 indicates that either the Open Apple or Solid Apple keys must be pressed together with the termination character.

4.2.2.8 "term list"

"Term_list" is a 20 byte table which specifies the termination type of each termination character. A value of 0 indicates that a normal termination will occur when the termination character (along with any keystroke modifiers) is pressed. A value of 1 indicates that an "interrupt" termination will occur.

4.2.3 Internal Information section

4.2.3.1 "origin x" and "origin y"

These parameters contain the relative coordinates of the start of the field within the current window. When the "User Input Routine" is entered initially (not reentered after an "interrupt" termination or "immediate" termination), "origin_x" and "origin_y" are set to the current relative cursor position.

4.2.3.2 "cursor x" and "cursor y"

These parameters contain the relative coordinates of the cursor within the current window. When the "User Input Routine" is entered initially, the cursor is positioned after the default Input String in the field and "cursor_x" and "cursor_y" are set to that coordinate location.

4.2.3.3 "cursor pos"

This parameter contains the relative position of the cursor in the field (not in the window). The value of "cursor_pos" is in the range 1.."width".

4.2.3.4 "input length"

This parameter contains the current length of the Input String. If the maximum size of the Input String is larger than the width of the field on the screen, the "User Input Routine" uses the "invisible" part of the Input String to save characters that were "pushed" out of the field by insertions. Therefore, "input_length" may have a value greater than "width". However, in this case, the length of the Input String actually returned to the user is still in the range 1.."width". The returned length of the Input String is contained in the first byte of the Input String.

4.2.3.5 "slow blink" and "fast blink"

These parameters are the count-down timers used to get the correct blinking frequency for the cursor. The cursor should blink at 80 cycles per minute with one phase taking twice as long as the other. Assuming that the cursor is "under" a character in the field and the "insert" cursor is on, the character should be visible twice as long as the underline. If the "replace" cursor is on, the inverse character should be visible twice as long as the normal character. The initial values of "slow_blink" and "fast_blink" will cause the correct cursor blink rate. However, if "immediate" mode is turned on, the cursor will no longer blink at the correct rate because the application program will get control in the middle of the blink loop. It is up to the application program to change "slow_blink" and "fast_blink" so that the cursor will again blink at the correct rate.

4.3 Default values

The default values of the Input Information Block are:

```
        width=254;
        fill_char=' ';
        mouse_fill=0;
        cursor=0;
        control=0;
        beep=1;
        immediate=0;
        entry_type=0;
        exit_type=0;
        bord_ch=' ';
        last_event=0;
        last_ch=0;
        last_mod=0;
        n_chars=2;
        char_list[1]=chr(13);    {RETURN}
        char_list[2]=chr(27);    {ESCAPE}
        mod_list[1]=0;
        mod_list[2]=0;
        exit_list[1]=0;
        exit_list[2]=0;
        origin_x=  |
        origin_y=  |      Current relative cursor coordinate in window
        cursor_x=  |      defined by Console Driver
        cursor_y=  |
        cursor_pos=0;
        input_length=0;
        slow_blink=  |    Values necessary to blink cursor
        fast_blink=  |    80 times per minute
```

5. Interface Description

5.1 Apple // Pascal

5.1.1 General Description

The Apple // Pascal version of the "User Input Routine" is part of the Console Driver and therefore requires that the Pascal environment be loaded with the correct Attach files. The Console Driver is configured as unit number 130.

To access the "User Input Routine", a Pascal program must make calls to the Console Driver. Three "unitstatus" calls are provided to initialize, set and get the Information Block. The actual call to the "User Input Routine" is in the form of a "unitread".

Sections 5.1.3 to 5.1.6 will describe each of the Console Driver calls in detail.

5.1.2 Format of the Information Block

The following is the Pascal equivalent of the Information Block:

```
const max_terms=20;         {Maximum number of terminators}
type  byte=0..255;
var   Input_Info:packed record

                            {General Information}
                            {-------------------}

      width:byte;           {Width of the field on the screen}
      fill_char:char;       {Fill character}
      mouse_fill:byte;      {0-use "fill_char" as fill character
                             1-use MouseText ghost underline}
      cursor:byte           {current cursor being used
                               0-insert cursor
                               1-replacement cursor}
      control:byte;         {0-Control chars will be ignored
                             1-Control chars allowed as input}
      beep:byte;            {0-errors will not be beeped
                             1-errors will be beeped}
      immediate:byte;       {0-calling routine gets control after the
                                 complete input is keyed in by user
                             1-calling routine gets control after each
                                 printable character is input}
      entry_type:byte;      {Indicates type of entry into routine
                               0-initial entry
                               1-interrupt re-entry
                               2-immediate re-entry}
      bord_ch:char;         {char to blink outside of field}

                            {Termination Information}
                            {-----------------------}

      exit_type:byte;       {Indicates which termination condition occurred
                               0-not terminated yet
                               not 0-index into terminating char list}
      last_event:byte;      {last event type (not used)}
      last_ch:char;         {character user keyed in}
      last_mod:byte;        {keypress modifier}
      n_chars:byte;         {Number of termination chars defined}

                            {The next 3 items define what keystrokes will
                             terminate or interrupt the routine.  The case
                             of each character is ignored}

      char_list:packed array [1..max_terms] of char;
                            {Chars which will terminate input}
      mod_list :packed array [1..max_terms] of byte;
                            {Modifiers for each char in "char_list"
                               0-none
                               1-Open Apple
                               2-Solid Apple
```

```
                              3-Either Open or Solid Apple}
term_list:packed array [1..max_terms] of byte;
                           {Termination types for each char in "char_list"
                              0-terminate input
                              1-interrupt input}

                           {Internal Information}
                           {-------------------}


origin_x :    byte;        {x coordinate of start of field}
origin_y :    byte;        {y coordinate of start of field}
cursor_x:     byte;        {x coordinate of cursor in field}
cursor_y:     byte;        {y coordinate of cursor in field}
cursor_pos:   byte;        {position     of cursor in field (1..width)}
input_length:byte;         {length of Input String (incl invisible part)}
slow_blink:integer;        {slow blink rate}
fast_blink:integer;        {fast blink rate}

end {Input_Info};
```

The text of this data structure is in the file "INPUT.INFO.TEXT" on the release disk.

5.1.3 <u>Initializing Input Information</u>

To set the User Input Information Block to its default values, call the procedure:

```
        init_mode:=24577;       {Console Driver command $6001}
        unitstatus(130,Input_Info,init_mode);
```

OR if the console driver is also to be initialized use:

```
        unitclear(130);
```

Note: the variable "Input Info" in the unitstatus call above is not actually used by the "User Input Routine". It is needed in the "unitstatus" call because of its parameter structure.

An automatic "unitclear" is performed by the Pascal system when it is booted.

5.1.4 Retrieving Input Information

To get the current settings of all the Input Information parameters, call the procedure:

```
        get_info:=16385;       {Console Driver command $4001}
        unitstatus(130,Input_Info,get_info);
```

where "Input_Info" is a record with the format specified in 5.1.2.

5.1.5 Setting Input Information

To change the data in the User Input Information Block, call the procedure:

```
        set_info:=8193;        {Console Driver command $2001}
        unitstatus(130,Input_Info,set_info);
```

where "Input_Info" is a record with the format specified in 5.1.2. If this call is never made, the "User Input Routine" uses the default values.

Note that changing any parameters in the record will not have any effect until the "unitstatus" call is made.

5.1.6 Calling the User Input Routine

To call the "User Input Routine", call the procedure:

```
        unitread(130,Input_Str,max_length);
```

where "Input_Str" is a string supplied by the calling routine where the "User Input Routine" will store the user's keystrokes. "Max_length" specifies the maximum number of characters which will fit in the string (usually 80 unless "Input_Str" is defined as an extended string).

If the Input String has an initial value, the "User Input Routine" will assume that it is a default value and display it.

Upon return from "unitread", IORESULT will contain the "exit_type" value which is the index into the "char_list" of terminating characters.

5.1.7 Examples

The program "Demo" is a good example of the "User Input Routine" in action. It can be used to try out many of the features.

In the simplest use of the "User Input Routine", the application displays a question on the screen using the Console Driver and then calls the "User Input Routine" for the answer. The following program segment illustrates the above:

```
VAR
  question,answer:string;
  ...
  ...
  question:='What is your name ? ';
  answer:='';
  unitwrite(130,question[1],length(question));
  unitread(130,answer,80);
```

If the application wants to provide a default name:

```
VAR
  question,answer:string;
  ...
  ...
  question:='What is your name ? ';
  answer:='Fred';
  unitwrite(130,question[1],length(question));
  unitread(130,answer,80);
```

If the application wants to provide the user with a small visible field:

```
CONST
  get_info=16385;        {Console Driver command $4001}
  set_info=8193;         {Console Driver command $2001}
VAR
  question,answer:string;
  Input_Info:packed record
           {use record structure in 5.1.2}
```

```
          end;
...
...
{Get the current Information Block}

unitstatus(130,Input_Info,get_info);

{Change the desired parameters}

Input_Info.width:=12;
Input_Info.fill_char:='.';

{Set the updated Information Block}

unitstatus(130,Input_Info,set_info);

{The rest of the logic is the same}

question:='What is your name ? ';
answer:='Fred';
unitwrite(130,question[1],length(question));
unitread(130,answer,80);
```

5.2 Basic

The Basic version of the "User Input Routine" is in the form of several AMPERSAND ('&') calls. The AMPERSAND facility allows a machine-language program to be loaded from a Basic program and its functions called in the form of Basic commands. The following commands are available:

&INITINPUT	--- Initialize Information Block
&GETINFO(IB%)	--- Get Information Block
&SETINFO(IB%)	--- Set Information Block
&INPUT(IS$)	--- Call "User Input Routine"
&EXITINPUT	--- Removes package from Ampersand hooks

The release disk contains the "User Input Routine" in a "relocatable" file "INPUT.REL". The EdAsm RLOAD facility must be used to load "INPUT.REL" from within the application program.

5.2.1 &INITINPUT

This call will initialize the Information Block to its default values. See 4.3 for the default values associated with each parameter.

5.2.2 &GETINFO(IB%)

This call retrieves the current Information Block and stores it into the integer array IB%. The array IB% should be dimensioned for at least 22+3*max_terms integers where "max_terms" is currently 20. The contents of each integer in IB% is as follows:

IB%(1)	= width		IB%(2)	= fill_char		IB%(3)	= mouse_fill
IB%(4)	= cursor		IB%(5)	= control		IB%(6)	= beep
IB%(7)	= immediate		IB%(8)	= entry_type		IB%(9)	= bord_ch
IB%(10)	= exit_type		IB%(11)	= last_event		IB%(12)	= last_ch
IB%(13)	= last_mod		IB%(14)	= n_chars		IB%(15)	= char_list
IB%(35)	= mod_list		IB%(55)	= term_lis		IB%(75)	= origin_x
IB%(76)	= origin_y		IB%(77)	= cursor_x		IB%(78)	= cursor_y
IB%(79)	= cursor_pos		IB%(80)	= input_length		IB%(81)	= slow_blink
IB%(82)	= fast_blink						

5.2.3 &SETINFO(IB%)

This call moves the contents of the integer array IB% into the Input Information Block. The format of IB% is assumed to be the same as described above.

5.2.4 &INPUT(IS$)

This is the actual call to the "User Input Routine". The variable "IS$" is a string which contains the default Input String and will contain the result of the user's input.

5.2.5 &EXITINPUT

This call will terminate the "User Input Routine" and disconnect the ampersand package.

5.2.6 Examples

User Input Routine

The program "Demo" is a good example of the "User Input Routine" in action. It can be used to try out many of the features.

In the simplest use of the "User Input Routine", the application displays a question on the screen and then calls the "User Input Routine" for the answer. The following program segment illustrates the above:

```
PRINT CHR$(4);"BLOAD INPUT.OBJ"
PRINT "What is your name ? ";
&INPUT(IS$)
```

If the application wants to provide a default name:

```
PRINT CHR$(4);"BLOAD INPUT.OBJ"
PRINT "What is your name ? ";
IS$="Fred"
&INPUT(IS$)
```

If the application wants to provide the user with a small visible field:

```
DIM IB%(82)
PRINT CHR$(4);"BLOAD INPUT.OBJ"
&GETINFO(IB%)
IB%(1)=20:REM width
IB%(2)=".":REM fill_char
&SETINFO(IB%)
PRINT "What is your name ? ";
IS$="Fred"
&INPUT(IS$)
```

5.3 Assembler

The Assembler version of the "User Input Routine" provides a set of calls similar to ProDOS MLI calls which provide the following functions:.

Initializing Input Information
Retrieving Input Information
Setting Input Information
Calling the User Input Routine

The release disk contains an "absolute" binary file "INPUT.OBJ" and a "relocatable" file "INPUT.REL". "INPUT.OBJ" was generated from "INPUT.REL" with the starting address $4000. If this starting address is not satisfactory for the application, the program "RELOCATOR" must be used to generate a new "absolute" file which starts at the desired location.

5.3.1 Format of the Information Block

The following is the Assembler equivalent of the Information Block:

```
maxterms        equ     20      ;Maximum number of terminators

InputInfo       equ     *
;                                General Information
;                                -------------------
width           db      0       ;Width of the field on the screen
fillchar        db      " "     ;Fill character
mousefill       db      0       ;0-use "fillchar" as fill character
                                ;1-use MouseText ghost underline
cursor          db      0       ;current cursor being used
                                  ;0-insert cursor
                                  ;1-replacement cursor
control         db      0       ;0-Control chars will be ignored
                                ;1-Control chars allowed as input
beep            db      0       ;0-errors will not be beeped
                                ;1-errors will be beeped
immediate       db      0       ;0-calling routine gets control after the
                                ;  complete input is keyed in by user
                                ;1-calling routine gets control after each
                                ;  printable character is input
entrytype       db      0       ;Indicates type of entry into routine
                                ;0-initial entry
                                ;1-interrupt re-entry
                                ;2-immediate re-entry
bordch          db      0       ;char to blink outside of field

;                                Termination Information
;                                -----------------------
exittype        db      0       ;Indicates which termination condition
                                ;  occurred
                                ;0-not terminated yet
                                ;not 0-index into terminating char list
lastevent       db      0       ;last event type (not used)
lastch          db      0       ;character user keyed in
lastmod         db      0       ;keypress modifier
```

```
nchars          db      0       ;Number of terminator chars currently
                                ;  defined

                                ;The next 3 items define what keystrokes
                                ;  will terminate or interrupt the routine.

charlist        ds      maxterms ;Chars which will terminate input
modlist         ds      maxterms ;Modifiers for each char in"charlist"
                                   ;0-none
                                   ;1-Open Apple
                                   ;2-Solid Apple
                                   ;3-Either Open or Solid Apple
termlist        ds      maxterms ;Termination types for each char in
                                 ;  "charlist"
                                   ;0-terminate input
                                   ;1-interrupt input

;                                 Internal Information
;                                 --------------------

originx         db      0       ;x coordinate of start of field
originy         db      0       ;y coordinate of start of field
cursorx         db      0       ;x coordinate of cursor in field
cursory         db      0       ;y coordinate of cursor in field
cursorpos       db      0       ;position     of cursor in field (1..width)
inputlength     db      0       ;length of Input String (incl invisib part)
slowblink       dw      0       ;slow blink rate
fastblink       dw      0       ;fast blink rate
```

5.3.2 <u>Format of Calls</u>

The "User Input Routine" has only one entry for all the functions. It is located at the beginning of the code. A call is made as follows:

```
        JSR     INPUT
        DB      COMMAND
        DW      PARAMPTR
        BNE     ERROR
```

The label "INPUT" is the starting address of the "User Input Routine". The programmer will determine this location when he relocates the routine in memory. In the application, there should be a statement of the form:

```
        INPUT    EQU      nnnn
```

where "nnnn" is the starting address of the "User Input Routine".

"COMMAND" is a number which specifies which function is requested. "PARAMPTR" is a two byte pointer to a parameter list.

When the "User Input Routine" returns to the calling program, the carry flag will be set if an error has been detected. The only possible error that is detected by the "User Input Routine" is an illegal command error (3). This occurs if "COMMAND" is not one of the available function numbers.

The calling program should check the carry flag (as in the BNE instruction above) and report the appropriate error. The actual error type is passed to the calling program in the A-register.

5.3.3 Initializing Input Information

This call will initialize the Information Block to its default values. See 4.3 for the default values associated with each parameter. This call has the following format:

```
        JSR     INPUT
        DB      10          ;command number for Initialize
        DW      0
```

5.3.4 Retrieving Input Information

This call will retrieve the current contents of the Input Information Block. The format of the call is:

```
        JSR     INPUT
        DB      11          ;command number for Get Input Information
        DW      INPUTINFO
```

where "INPUTINFO" is the address of a buffer where the contents of the Information Block is to be moved. This buffer will have the format as described in 4.1.

5.3.5 Setting Input Information

This call will set the Input Information Block to values in the specified buffer. The format of the call is:

```
        JSR     INPUT
        DB      12          ;command number for Set Input Information
        DW      INPUTINFO
```

where "INPUTINFO" is the address of the buffer. This buffer must have the format as described in 4.1.

5.3.6 Calling the User Input Routine

This call will perform the actual input. The format of the call is:

```
        JSR     INPUT
        DB      13          ;command number for Input
        DW      PARAM
```

where the format of "PARAM" is:

```
PARAM   DW STRING
        DB maxlength
STRING  STR "This is the default"
```

Upon return from this call, the A register will contain the "exittype".

5.3.7 Examples

The program "Demo" is a good example of the "User Input Routine" in action. It can be used to try out many of the features.

In the simplest use of the "User Input Routine", the application displays a question on the screen and then calls the "User Input Routine" for the answer. The following program segment illustrates the above:

```
QUESTION STR "What is your name ? "
ANSWER   STR ""
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
         ...
         ...
;
;        display question
;
         LDY #0
LOOP     LDA QUESTION+1,Y
         JSR $FDED           ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
         JSR INPUT
         DB  13
         DW  PARAM
```

If the application wants to provide a default name:

```
QUESTION STR "What is your name ? "
ANSWER   STR "Fred"
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
         ...
         ...
;
;        display question
;
         LDY #0
LOOP     LDA QUESTION+1,Y
         JSR $FDED           ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
```

```
          JSR INPUT
          DB  13
          DW  PARAM
```

If the application wants to provide the user with a small visible field:

```
QUESTION STR "What is your name ? "
ANSWER   STR "Fred"
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
INPUTINFO DS 84
         ...
         ...
;
;        get current Information Block
;
         JSR INPUT
         DS  11
         DW  INPUTINFO
;
;        change values in Information Block
;
         LDA #80
         STA INPUTINFO       ;width
         LDA #"."
         STA INPUTINFO+1     ;fillchar
;
;        set Information Block
;
         JSR INPUT
         DS  12
         DW  INPUTINFO
;
;        display question
;
         LDY #0
LOOP     LDA QUESTION+1,Y
         JSR $FDED           ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
         JSR INPUT
         DB  13
         DW  PARAM
```

ConsoleStuff Library
External Reference Specification

11/02/84
Lou Infeld

BETA Release Version

TABLE OF CONTENTS

1. Introduction

The Console Driver provides a quick method for an application to display text on the screen. It provides many commands to quickly move around the screen, blank portions of the screen, divide the screen into windows and other useful facilities. To use these, a buffer containing text as well as console commands has to be sent to the Console Driver.

The ConsoleStuff Library is a Pascal unit that an application can use to build the necessary console buffer which can then be sent to the Apple // Console Driver. Many text formatting routines are in this library as well as other utility routines to write on the screen including Boxes, Message Areas and Help Screens.

Since this unit requires the Apple // Console Driver to be in the application's environment, the Console Driver must be loaded when the Pascal system is booted.

The ConsoleStuff Library contains a large (1024 bytes) Console Buffer which is used to send console data and commands to the Console Driver. Most of the routines in the library build up this buffer until it is full or until the calling routine explicitly requests that the buffer be sent to the Console Driver. Only when the Console Driver gets the buffer does anything happen on the screen. Since the Console Driver writes onto the screen very rapidly, a Pascal application gets machine language speed from its console output. This is in dramatic contrast to the built-in Pascal console interface.

In addition to formatting routines, this library also contains routines which display Boxes for Comments or Help Screens. These Boxes appear "on top" of the screen overwriting whatever was beneath. When the application is finished with these Boxes, they disappear and the text that was underneath reappears.

2. Description

2.1 Interface Constants and Data Structures

Constants and variables defined in the Interface section of the ConsoleStuff unit define mnemonics for the Console Driver commands and the Console Buffer and the Box Comment procedures.

2.1.1 Console Driver commands

Instead of using the numerical values of the Console Driver commands, the ConsoleStuff unit defines a character constant for each command which the application can use. The following character constants are defined as Console Driver commands:

```
c_Reset_View    {chr(01)},    c_Set_View      {chr(02)},
c_Beg_Line      {chr(03)},    c_Restore_View {chr(04)},
c_On_Cursor     {chr(05)},    c_Off_Cursor    {chr(06)},
c_Bell          {chr(07)},    c_L_Cursor      {chr(08)},
c_D_Cursor      {chr(10)},    c_Cl_To_End     {chr(11)},
c_Cl_View       {chr(12)},    c_Return_Cursor{chr(13)},
c_Normal        {chr(14)},    c_Inverse       {chr(15)},
c_DLE           {chr(16)},    c_Horiz_Shift   {chr(17)},
c_Vert_Pos      {chr(18)},    c_Cl_Beg_View   {chr(19)},
c_Horiz_Pos     {chr(20)},    c_Cursor_Move   {chr(21)},
c_D_Scroll      {chr(22)},    c_U_Scroll      {chr(23)},
c_Off_Mouse     {chr(24)},    c_Home_Cursor   {chr(25)},
c_Cl_Line       {chr(26)},    c_On_Mouse      {chr(27)},
c_Escape        {chr(27)},    c_R_Cursor      {chr(28)},
c_Cl_End_Line   {chr(29)},    c_Abs_Pos       {chr(30)},
c_U_Cursor      {chr(31)}
```

Note that the commands "c_On_Cursor" and "c_Off_Cursor" are ignored by the Apple // Console Driver since they affect the Pascal cursor. If these characters are sent to the Pascal console using a "write" command, they will have the desired effect.

2.1.2 MouseText characters

Additional character constants are defined for some useful MouseText characters. To use any of these characters, the "c_On_Mouse" command must first be sent to the Apple // Console Driver. These are the defined variables:

```
{Line drawing characters}

f_R_Side    {chr(95)}, f_L_Side    {chr(90)}, f_U_Side    {chr(95)},
f_D_Side    {chr(76)}, f_Horiz     {chr(83)}, f_Vert      {chr(124)},

{Arrows}

f_U_Arrow   {chr(75)}, f_D_Arrow   {chr(74)}, f_R_Arrow   {chr(85)},
f_L_Arrow   {chr(72)},

{Specials}
```

f_Open_Apple {chr(65)}, f_Closed_Apple{chr(64)}

2.1.3 Console Buffer data structure

The Console Buffer itself is available as an interface variable. Additionally, a set of variables are available which indicate the current status of the buffer:

```
CBuff:   CBuffType;                     {Console Buffer}

{CBuff data structure}

CBuff_Globals:  record
                  size: integer;        {# of characters in Console Buffer}
                  lines:integer;        {# of lines in Console Buffer}
                  width:integer;        {Max width of lines in Console Buffer}
                end;
```

2.1.4 Box Comment data structure

The "BoxComment" procedure displays a one line message on the screen. See 2.3 for a detailed discussion of the procedure. The configuration data structure associated with this procedure are:

```
{Box Comment data structure}

Box_Globals:  record
                Y:     integer;  {Y coordinate of Box}
                Stat:  integer;  {Status to be inserted instead of "&" in
                                  comment}
                Ch:    char;     {Character read if comment ended in "?"}
                Clear: boolean;  {If true, comment will be cleared from
                                  screen.  If false, comment remains on
                                  screen until next BoxComment call}
                Beep:  boolean;  {if true, a beep will be sounded with
                                  comment}
                Time:  integer;  {# of secs comment stays on screen if no
                                  keypress}
              end;
```

2.2 Console Buffer procedures

These procedures allow the application to prepare the Console Buffer for subsequent transmission to the Console Driver. Some provide formatting utilities similar to the Pascal "write" function. Some provide Window and Line drawing abilities.

Each procedure will add to the Console Buffer the necessary text and console commands to perform the requested function. When the "CWrite" procedure is called, the Console Buffer is sent to the Console Driver. The Console Driver will interpret each character in the Console Buffer and will either display the character or perform one of its console functions. The Console Buffer is emptied and can be again "filled" by the ConsoleStuff procedures.

Since the Console Driver supports the concept of a "window", all coordinate parameters should be specified relative to the window in effect. The one exception is the "Window" procedure which requires absolute coordinates since it establishes a new window relative to a screen coordinate systeem in which (0,0) indicates the upper left and (79,23) the lower right corners.

2.2.1 CWrite

This procedure sends the Console Buffer to the Console Driver and initializes the Console Buffer data structure to zeroes.

Example: CWrite;

2.2.2 CWriteCh

This procedure adds the specified character to the Console Buffer.

```
Examples:  CWriteCh('a');
           CWriteCh(c_Cl_View);
```

2.2.3 CWriteStr

This procedure adds the specified string to the Console Buffer. Any size string up to 255 characters is allowed.

```
Examples:  CWriteStr('This will be displayed');
           CWriteStr(strvar);
```

2.2.4 CWriteIStr

This procedure adds the specified string to the Console Buffer. However, the string will display in Inverse Mode.

```
Examples:  CWriteIStr('This will be in inverse');
           CWriteIStr(strvar);
```

2.2.5 CWriteln

This procedure is similar to "CWriteStr" except a Carriage Return is added to the Console Buffer after the string.

```
Examples:  CWriteln('This is a title');
           CWriteln('---------------');
```

2.2.6 CWriteNum

The specified integer is converted to an ascii string and added to the Console Buffer. The size of the field and the fill character can be specified. The integer will be right justified in the field unless the field size is 0.

```
Examples:  CWriteNum(10,5,' ');    {resulting field -- "   10"}
           i:=9;
           CWriteNum(i,2,'0');     {resulting field -- "09"}
           CWriteNum(100,0,' ');   {resulting field -- "100"}
```

2.2.7 CGotoxy

Calling this routine adds the Console Driver commands necessary to change the character position to the specified relative coordinates.

```
Example:  CGotoxy(10,10);          {char position changed to (10,10)}
```

2.2.8 CPlace

This routine combines the "CGotoxy" and "CWriteCh" procedures. It effectively puts the specified character into the specified position.

```
Example:  CPlace(10,10,'X');     {char "X" displayed at (10,10)}
```

2.2.9 Window

This procedure changes the Console Driver window to that specified by the given coordinates. The "absolute" coordinates of the upper left corner and the lower right corner must be specified. These coordinates are not checked for validity and illogical values will have strange effects.

```
Example:  Window(10,15,60,20);   {changes window so that upper left
                                  corner is at (10,15) and lower right
                                  corner is at (60,20)}
```

Note that this procedure is the only Console Buffer routine which uses absolute coordinates. All others use coordinates relative to the current window in effect.

2.2.10 Line

This procedure causes a line to be drawn with the specified character between the two sets of coordinates. Only vertical or horizontal lines can be drawn (others will be ignored). The coordinates specified are not checked for validity (other than defining a line) and illogical values will have strange effects.

```
Examples:  Line(5,10,5,20)       {result: vertical line between
                                  coordinates (5,10) and (5,20)}
           Line(10,5,60,5)       {result: horizontal line between
                                  coordinates (10,5) and (60,5)}
```

2.2.11 Box

This procedure draws a box (using MouseText fonts) at the specified coordinates. These coordinates are not checked for validity and illogical values will have strange effects.

```
Example:  Box(10,15,60,20);      {result: box with upper left corner at
                                  (10,15) and lower right corner at
                                  (60,20)}
```

2.3 Box Comment procedure

This procedure places the specified comment on the screen inside of a narrow box.

If the message is not a question (ends with a question mark), the box stays on the screen for a period of time or until any readable key is pressed.

If the message is a question, the box stays on the screen until a key is pressed. This key is assumed to be the answer to the question and is stored in the Box Comment data structure field "Box_Globals.Ch".

If an "&" is embedded in the comment, it is replaced with the ASCII equivalent of the integer in the "Stat" field of "Box_Globals".

Other fields of the Box Comment data structure can be set to configure the "BoxComment" procedure:

Y	-- Y coordinate of the Box (default is 21)
Clear	-- If TRUE (default), comment will be cleared from screen If FALSE, comment stays on screen until next call
Beep	-- If TRUE (default), a beep will be sounded with comment
Time	-- Number of secs comment stays on screen if no key pressed (default is 15)

Examples:
```
BoxComment('This is a comment');
Box_Globals.Stat:=10;
BoxComment('The status is "&"');
BoxComment('Do you want to continue (Y/N) ?');
```

2.4 Help procedures

Two procedures are available to aid in displaying Help Screens. The first opens up the Help Screen and the second closes it down and redisplays the original screen contents.

The calling routine first sets up the Console Buffer using the Console Buffer routines without calling the "CWrite" procedure. Next the "OpenHelp" routine is called. It puts a Box on the screen just large enough to contain the Help lines. When the "CloseHelp" procedure is called, the Help Box disappears and the screen environment is restored.

Examples:
```
CWriteln('This is the first line of the Help Screen');
CWriteln('This is the second line of the Help Screen');
CWriteln('This is the last line of the Help Screen');
OpenHelp;
read(keyboard,ch);
CloseHelp;
```

2.5 GetXY procedure

This routine returns the current relative coordinates of the character position within the current window as well as the window coordinates themselves. Note that this procedure is not a Console Buffer formatting procedure. Coordinates returned are those currently in effect.

Example:
```
GetXY(x,y,ulx,uly,lrx,lry);     {char position is (x,y) and
                                  upper left corner of window is
                                  (ulx,uly) and lower right
```

```
                                        corner of window is (lrx,lry)}
```

Console Driver/User Input Routine
Release 1.0B1 Notes

Lou Infeld

04/16/85

Version 1.0B1 is the first Beta release for the Console Driver and User Input Routine. Previous versions are cosidered Alpha releases. The following changes were made in the Console Driver and User Input Routines since the last release:

Console Driver

- o Documentation corrected -- Several of the control codes were incorrectly specified in the documentation.
- o Bug fixed -- Calling the Horizontal, Vertical or Absolute Position commands with values outside of the current window sometimes resulted in positioning the cursor to the top or left side of the window rather than the bottom or right side.
- o Bug fixed -- Clearing viewports that are two lines high caused Console Driver to hang.

User Input Routine

- o Bug fixed -- Sometimes cursor remnants remained on screen.
- o Bug fixed -- Control F didn't work in Pascal version. The fix was to disable all special Pascal control characters including Control @, Control S, Control Z, etc. as well as Control F.
- o Standard change -- Control R (restore) changed to Control Z (undo).
- o Enhancement -- Border character added to Information Block. This character will be blinked (rather than a Blank) whenever the field is filled and the cursor is forced outside.
- o Enhancement -- Upon initial entry in immediate mode, the application will get control before the cursor starts blinking. This will allow initial cursor repositioning without cursor remnants.
- o Enhancement -- Last event type parameter added to Information Block. This parameter is not currently used.

EXTERNAL REFERENCE SPECIFICATION

APPLE // CONSOLE DRIVER

Neal Johnson

April 10, 1985

BETA RELEASE VERSION

TABLE OF CONTENTS

1. Introduction

The Console Driver (henceforth known as "the driver") is an implementation of the Apple /// Console Driver, with special modifications, for the Apple // series of computers (//+, //e, and //c). The driver supplies a simple and consistent interface to a nearly complete set of display format and control procedures contained in a relatively small and fast package. Both display and control commands are sent to the driver in the same manner. This allows a programmer to build up a set of data structures that contain both display and control information. Presentation of the information to the driver can be made with one call. This simplifies the programming of the human interface for a program, in that the programmer does not have to make a sequence of calls to set up for text to be displayed. Instead, the format information can be imbedded in the text itself.

The driver supports a form of "window" known as a "viewport". The viewport is a rectangular portion of the screen where all console functions take place. This feature allows the programmer to define a portion of the screen where s/he wants text to be displayed. All text outside the viewport is protected. Any display of the text will occur within the bounds of the viewport.

The console driver can serve as a low level tool for the implementation of different styles of human interface. Much of the implementation for the various styles of human interface would be in the design of the data structures describing the format and text to be displayed.

NOTE: This release (1.0) of the Console Driver only supports an 80-column screen. Sections describing the 40-column screen should be ignored at this time.

2. Functional Description

2.1 Screen Map

2.1.1 40-Column Screen

The 40-Column screen consists of 40 columns of text in 24 lines. The upper left corner is column 0, line 0 (or simply 0,0.) Columns are number left to right, 0 to 39. Lines are numbered top to bottom, 0 to 23.

2.2.2 80-Column Screen

The 80-Column screen consists of 80 columns of text in 24 lines. The upper left corner is column 0, line 0 (or simply 0,0.) Columns are number left to right, 0 to 79. Lines are numbered top to bottom, 0 to 23.

2.2.3 The Viewport

The Viewport is a rectangular portion of the screen where all current text is displayed. Portions of the screen outside the viewport are not affected by either format or display commands.

The driver maintains a "cursor", which is not visible on the screen, that represents the current location that a displayable character will be placed. This cursor is specified by the value of the two variables CH and CV. (See Section 2.2.1 below)

When the console driver is first used, the viewport defaults to the whole screen (either 40 or 80 column display). The programmer can set the viewport by a special control and four parameter bytes which specify the upper left and the lower right corners of the viewport.From that point on, all console functions will take place within the bounds of the viewport.

The current viewport specifications can be saved and the viewport can then be set to the specifications of the previously saved viewport. The programmer can then return to the original viewport settings with another command.

2.2 Console Driver Environment Controls

2.2.1 Cursor Position

The current cursor position is maintained in two variables:

CH - current horizontal position

CV - current vertical postion

When the driver is first used, these values are set to zero signifying the upper-left corner of the screen.

The values of CH and CV always represent the absolute screen coordinates (actual column and line number) and are not relative to the current viewport.

2.2.2 Viewport Specification

The viewport is specified by six variables that specify the top, bottom, left, and right edge of the viewport and also its width (in columns) and its length (in lines).

WNDTOP - top line of viewport

WNDBOT - bottom line of viewport

WNDLFT - left column of viewport

WNDRGT - right column of viewport

WNDWTH - width of viewport in columns

WNDLEN - length of viewport in lines

2.2.3 Cursor Movement Controls

The cursor movement controls specify the rules for moving the cursor within a viewport. These controls are flags directing the driver how to move the cursor. If set to zero, they are false and if set to one, they are true. The four cursor movement controls are:

CONLFD (Line Feed) - If true, the console driver will automatically perform a line feed after every carriage return (control code 13 decimal or $0D hex.) When false, no automatic line feed is performed. The programmer can perform a line feed by explicitly sending a line feed character (10 decimal or $0A hex.) Scrolling is controlled by the other cursor movement control settings.

CONADV (Advance) - If true, the cursor will advance one space to the right after each display character is placed on the screen. When false, the cursor will not advance (it will remain in the same position) after each character. In this case the programmer would have to explicitly move the cursor by sending a Move Cursor Right control (09 decimal or $09 hex.) Wrapping and/or scrolling is controlled by the other cursor movement control settings.

CONWRAP (Wrap) - If true, an attempt to move the cursor beyond the right or left edge of the viewport will cause the cursor to be placed at the opposite edge of the next or previous line, respectively, of the viewport. If false, the cursor remains at the edge of the viewport on the current line. To move to either the next or previous line requires the programmer to send a Move Cursor Up (11 decimal or $0B hex) or a Move Cursor Down (10 decimal or $0A hex) character, followed by either a Return Cursor (13 decimal or $0D hex) to move the cursor to the beginning of the previous line or a Horizontal Position (24 decimal or $18 hex) with the appropriate parameter value to send the cursor to the end of line. Scrolling is controlled by the other cursor movement control.

CONSCRL (Scroll) - If true, an attempt to move the cursor beyond the top or bottom line of the viewport, will cause the contents of the viewport to be scrolled either down or up. The cursor will then be placed at the beginning of the new top or bottom line. If false, the cursor will remain at the top or bottom of the viewport.

DLEFLAG (Space Expansion) - If true, the DLE's ($10 hex or 16 decimal) will be interpreted as space expansion controls with a following parameter byte. (See section 2.3.17) If false, then they are ignored.

2.2.4 Fill Character

The fill character is the character used to clear the contents of the viewport. This value is a Space (32 decimal or $20 hex). Its value is in the variable CONFILL. Due to the Apple // character mapping the actual binary value of the fill character is $0A0 hex or 160 decimal for a normal Space character or $20 hex or 32 decimal for an inverse Space character.

2.2.5 Default Settings and Environment

2.2.5.1 The Default Viewport

The default viewport is the entire screen (either 40 or 80 columns).

Viewport Parameter	40-Col Value	80-Col Value
WNDTOP	0	0
WNDBOT	23	23
WNDLFT	0	0
WNDRGT	39	79
WNDWTH	40	80
WNDLEN	24	24

2.2.5.2 The Default Cursor Movement Controls

The default settings for the Cursor Movement Controls are:

CONLFD (Line Feed)	- TRUE
CONADV (Advance)	- TRUE
CONWRAP (Wrap)	- TRUE

CONSCRL (Scroll) - TRUE

DLEFLAG (Space Expansion) - TRUE

2.2.5.3 The Default Screen Environment

The default screen environment is the default viewport (See section 2.2.5.1), the text display mode is normal, the cursor is off, the fill character is space, and the initial position of the cursor is in the upper-left hand corner (0,0).

2.2.5.4 Mousetext

The flag MOUSE, specifies whether or not the driver will display mousetext characters. If MOUSE is true then character is the range $40 to 5F hex or 64 to 95 decimal will be mapped into the mousetext character set. If false, the mapping will not take place. Control codes will always be processed as is. The default is MOUSE false.

2.2.5.5 Normal and Inverse Text

The flag CONVID controls the display of text in either normal or inverse modes. If CONVID is $80 hex or 128 decimal, text is displayed in normal mode. If CONVID is 0, then text is displayed in inverse. The setting of CONVID is handled via two control codes described below (Set Normal Text or Set Inverse Text.)

2.3 Screen Control Codes

2.3.1 No Operation

CONTROL CODE: $00 (hex) or 00 (decimal)

OPERATION: No Operation

DESCRIPTION: This control code has no effect and is ignored.

2.3.2 Save and Reset Viewport

CONTROL CODE: $01 (hex) or 01 (decimal)

OPERATION: Save and Reset Viewport

DESCRIPTION: This control code saves the current settings of the viewport: its coordinates, cursor position, cursor motion controls, mousetext, and normal/inverse setting. The viewport will then be set to the default values of the full screen. (See section 2.3.5 Restore Viewport)

2.3.3 Set Viewport

CONTROL CODE: $02 (hex) or 02 (decimal)

OPERATION: Set Viewport

DESCRIPTION: This control code will set the viewport. It requires four parameter bytes which specify the absolute coordinates for the upper-left and lower-right corners of the viewport. The order of the parameters is:

> upper-left corner X (or column) value
>
> upper-left corner Y (or line) value
>
> lower-right corner X (or column) value
>
> lower-right corner Y (or line) value

If less than four parameters are passed, this control code will be ignored. This control simply sets the boundaries for the viewport. It does not affect the cursor motion controls, normal/inverse, or mousetext setting. It will not save the current viewport. The cursor will be placed in the upper-left corner of the new viewport.

The parameters are checked for validity prior to setting the viewport values. The rules for validity are as follows:

> If any paramter byte is > 127, i.e. minus value because bit 7 is set, this command will be ignored.
>
> For any X coordinate (UL corner or LR corner), if it is > 39 or 79 (depending on the screen size) then it will be set to 39 or 79.
>
> For any Y coordinate (UL corner or LR corner), if it is > 23 then it will be set to 23.
>
> UL corner X will be used for WNDLFT.
>
> UL corner Y will be used for WNDTOP.
>
> LR corner X, if greater than WNDLFT, will be used for WNDRGT, else this command will be ignored.
>
> LR corner Y, if greater than WNDTOP, will be used for WNDBOT, else this command will be ignored.

If for any reason the command is ignored, it will not change the current viewport settings.

2.3.4 Clear from Beginning of Line

CONTROL CODE: $03 (hex) or 03 (decimal)

OPERATION: Clear from Beginning of Line

DESCRIPTION: This control code will clear the current line from the beginning of the line to and including the current cursor position in that line.

2.3.5 Restore Viewport

CONTROL CODE: $04 (hex) or 04 (decimal)

OPERATION: Restore Viewport

DESCRIPTION: This control code will restore the viewport to the values of the last previously saved viewport. If no viewport has been saved, then the values will be set to the default values for the whole screen. (See section 2.3.2 Save and Reset Viewport)

2.3.6 Undefined

CONTROL CODE: $05 (hex) or 05 (decimal)

OPERATION: Undefined

DESCRIPTION: This control code is undefined and is ignored.

2.3.7 Undefined

CONTROL CODE: $06 (hex) or 06 (decimal)

OPERATION: Undefined

DESCRIPTION: This control code is undefined and is ignored.

2.3.8 Sound the Bell

CONTROL CODE: $07 (hex) or 07 (decimal)

OPERATION: Sound the Bell

DESCRIPTION: This control code will cause the ProDOS recommended "beep" to be sounded. It has no effect on the screen. Sequencial control codes will have the effect of producing a longer sound.

2.3.9 Move Cursor Left

CONTROL CODE: $08 (hex) or 08 (decimal)

OPERATION: Move Cursor Left

DESCRIPTION: This control code will move the cursor left one position. Wrapping around and scrolling are performed in accordance with the settings of the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.10 2.3.11 Move Cursor Down

CONTROL CODE: $0A (hex) or 10 (decimal)

OPERATION: Move Cursor Down (Line Feed)

DESCRIPTION: This control code moves the cursor down one line. Scrolling is performed in accordance with the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.12 Clear to End of Viewport

CONTROL CODE: $0B (hex) or 11 (decimal)

OPERATION: Clear to End of Viewport

DESCRIPTION: This control code will clear the contents of the viewport, starting from and including the current cursor position to the end of the line and all the lines below the cursor. The cursor is not moved.

2.3.13 Clear Viewport

CONTROL CODE: $0C (hex) or 12 (decimal)

OPERATION: Clear Viewport

DESCRIPTION: This control character will move the cursor to the upper-left corner of the viewport and then clear the viewport by setting the contents to space characters. The space characters will be either normal or inverse depending on the setting of this mode. (See sections 2.3.15 and 2.3.16)

2.3.14 Return Cursor

CONTROL CODE: $0D (hex) or 13 (decimal)

OPERATION: Return Cursor (Carriage Return)

DESCRIPTION: This control code moves the cursor to the beginning of the current line (the left edge of the viewport.) A line feed may also be issued automatically after the return depending on the setting of the cursor motion controls. Scrolling may also take place. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.15 Set Normal Text

CONTROL CODE: $0E (hex) or 14 (decimal)

OPERATION: Set Normal Text

DESCRIPTION: This control code specifies that all subsequent characters will be displayed as white characters on a black background. It does not affect any characters already on the screen. This control code will set the flag CONVID to $80 hex or 128 decimal. (See section 2.3.16 Set Inverse Text)

2.3.16 Set Inverse Text

CONTROL CODE: $0F (hex) or 15 (decimal)

OPERATION: Set Inverse Text

DESCRIPTION: This control code specifies that all subsequent characters will be displayed as black characters on a white background. It does not affect any characters already on the screen. This control code will set the flag CONVID to 0. (See section 2.3.15 Set Normal Text)

2.3.17 Space Expansion

CONTROL CODE: $10 (hex) or 16 (decimal)

OPERATION: Space Expansion

DESCRIPTION: This control code supports the DLE space expansion that exists in Pascal text files. It takes one parameter which represents the number of spaces to output plus 32. The driver subtracts 32 from the parameter to determine the number of spaces to output to the screen. If the parameter does not exist, then the driver will ignore this control. DLE expansion can be turned off using the mode value of 4 or 12 in the UNITWRITE call to the driver. (See section 3.1.2 below.) It can also be turned on or off with the Cursor Movement Control. (See Section 2.3.22 below) The default is on.

2.3.18 Horizontal Shift

CONTROL CODE: $11 (hex) or 17 (decimal)

OPERATION: Horizontal Shift

DESCRIPTION: This control code will cause the contents of the viewport to be shifted right or left the number of columns specified by the single byte parameter following the control code. If the parameter does not exist or is set to 0, the control will have no effect. The parameter is interpreted as an eight-bit two's complement number. If it is positive (less than 128 decimal or $7F hex) the contents will be shifted right the number of columns equal to the value of the number. If it is negative (greater than or equal to 128 decimal or $7F hex), the contents will be shifted left the number of columns equal to the negative value of the number. In both cases, if the value is greater than or equal to the width of the viewport, it will cause the viewport to be cleared.

The shifted characters are moved directly to their destination location. The space vacated by the shifted characters is set to blanks. Characters shifted out of the viewport are removed from the screen and are not recoverable.

2.3.19 Vertical Position

CONTROL CODE: $12 (hex) or 18 (decimal)

OPERATION: Vertical Position

DESCRIPTION: This control code will move the cursor vertically to the relative line number passed in a single byte parameter (0 to 23 for both 40-columns or 80-columns). A parameter whose value is 10 means to move to the tenth line in the viewport, not to line 10 of the whole screen. A parameter of 0 will move the cursor to the topmost line. To determine the correct relative line, the parameter is added to the value of WNDTOP (See Section 2.2.2 Viewport Specifications). This is an eight-bit add. If the resulting value is greater than the value of WNDBOT (the bottommost line of the viewport) but less than 127 then the cursor will be placed in the bottommost line of the viewport. If the sum is greater than 127 (negative) then the cursor will be placed in the topmost line. If the parameter is missing, this control will be ignored. This control has no effect on the horizontal position of the cursor.

2.3.20 Clear from Beginning of Viewport

CONTROL CODE: $13 (hex) or 19 (decimal)

OPERATION:Clear from Beginning of Viewport

DESCRIPTION: This control code will clear the viewport from its beginning (0, 0 or home position) to and including the cursor. The cursor is not moved.

2.3.21 Horizontal Position

CONTROL CODE: $14 (hex) or 20 (decimal)

OPERATION: Horizontal Position

DESCRIPTION: This control code will move the cursor horizontally to the relative column number passed in a single byte parameter (0 to 39 for 40-columns or 0 to 79 for 80-columns). A parameter whose value is 10 means to move to the tenth column in the viewport, not to column 10 of the whole screen. A parameter of 0 will move the cursor to the left-most column. To determine the correct relative column, the parameter is added to the value of WNDLFT (See Section 2.2.2 Viewport Specifications). This is an eight-bit add. If the resulting value is greater than the value of WNDRGT (the rightmost column of the viewport) but less than 127 then the cursor will be placed in the rightmost column of the viewport. If the sum is greater than 127 (negative) then the cursor will be placed in the leftmost column. If the parameter is missing, this control will be ignored. This control has no effect on the vertical position of the cursor.

2.3.22 Cursor Movement Controls

CONTROL CODE: $15 (hex) or 21 (decimal)

OPERATION: Cursor Movement Controls

DESCRIPTION: This control code and its parameter will set the cursor movement controls as specified by the parameter. The parameter is a single byte value, with only the lower five bits as significant. The upper four bits are to be set to zero. A zero will reset the control and a one will set it. If the parameter does not exist or the upper three bits are non-zero, the command is ignored. (See section 2.2.3 Cursor Movement Controls)

Bit	Control
Bit 0	Advance
Bit 1	Line Feed
Bit 2	Wrap
Bit 3	Scroll
Bit 4	DLE Space Expansion

2.3.23 Scroll Down

CONTROL CODE: $16 (hex) or 22 (decimal)

OPERATION: Scroll Down

DESCRIPTION: This control code will cause the contents of the viewport to scrolled down, leaving a blank line at the top of the viewport. The cursor position will remain the same after the scroll.

2.3.24 Scroll Up

CONTROL CODE: $17 (hex) or 23 (decimal)

OPERATION: Scroll Up

DESCRIPTION: This control code will cause the contents of the viewport to be scrolled up, leaving a blank line at the bottom of the viewport. The cursor position will remain the same after the scroll.

2.3.25 Turn Mousetext Off

CONTROL CODE: $18 (hex) or 24 (decimal)

OPERATION: Turn Mousetext Off

DESCRIPTION: This control code turns off the display of mousetext (See Section 2.3.28).

2.3.26 Home Cursor

CONTROL CODE: $19 (hex) or 25 (decimal)

OPERATION: Home Cursor

DESCRIPTION: This control code moves the cursor to the upper-left corner of the current viewport. It does not clear any portion of the viewport, nor does it change any of the viewport settings.

2.3.27 Clear Line

CONTROL CODE: $1A (hex) or 26 (decimal)

OPERATION: Clear Line

DESCRIPTION: This control code moves the cursor to the beginning of the current line and then clears the entire line.

2.3.28 Turn Mousetext On

CONTROL CODE: $1B (hex) or 27 (decimal)

OPERATION: Turn Mousetext On

DESCRIPTION: This control code turns on the display of mousetext characters. All displayable characters (See Section 2.4 Displayable Characters) in the range $40 - $5F hex or 64 - 95 decimal will be mapped into the mousetext characters for display. (See Section 2.3.25)

2.3.29 Move Cursor Right

CONTROL CODE: $1C (hex) or 28 (decimal)

OPERATION: Move Cursor Right

DESCRIPTION: This control code will move the cursor right one position. Wrapping around and scrolling are performed in accordance with the settings of the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.3.30 Clear to End Of Line

CONTROL CODE: $1D (hex) or 29 (decimal)

OPERATION: Clear to End of Line

DESCRIPTION: This control code clears the current line starting from and including the current cursor position in the line. The cursor is not moved.

2.3.31 Absolute Position

CONTROL CODE: $1E (hex) or 30 (decimal)

OPERATION: Absolute Position

DESCRIPTION: This control code combines the actions of the Horizontal Position and Vertical Position control codes. (See sections 2.3.25 and 2.3.26). It requires two single byte parameters. The first specifies the horizontal position and the second specifies the vertical position of the cursor. Placement of the cursor follows the rules given under both Horizontal and Vertical Position control codes. If both parameter bytes are missing, the command is ignored.

2.3.32 Move Cursor Up

CONTROL CODE: $1F (hex) or 31 (decimal)

OPERATION: Move Cusor Up (Vertical Tab)

DESCRIPTION: This control code moves the cursor up one line. Scrolling is performed in accordance with the cursor motion controls. (See sections 2.2.3 and 2.3.22 Cursor Movement Controls)

2.4 Displayable Characters

The Console Driver uses the Alternate Character set of the Apple // for the display of characters. It assumes however, that all characters passed to it are in the standard ASCII character set (range $00 to $7F hex or 0 to 127 decimal). These characters will be mapped into the appropriate character set for display purposes, e.g. normal or inverse or mousetext.

A special case is made for characters passed to the driver in the range $80 to $FF hex or 128 to 255 decimal. The characters are displayed after reseting the 7th bit. This results in the mapping shown in the chart below:

$80 - $9F	mapped to	Inverse upper case letters
$A0 - $BF	mapped to	Inverse special characters
$C0 - $DF	mapped to	Mousetext characters
$E0 - $FF	mapped to	Inverse lower case letters

This is independent of the settings for normal/inverse and mousetext in the driver. Refer to the Apple // Reference Manuals for more details on the character sets.

All characters in the range $00 to $1F hex or 0 to 31 decimal are defined as control codes which invoke the operations listed above in Section 2.3.

All characters in the range $20 to $7F hex or 32 to 127 decimal are defined as displayable characters and will be displayed given the various settings of the console driver on the screen.

The use of mousetext requires that the mousetext-on control code be sent to the console driver. Then any characters in the range $40 to $5F hex or 64 to 95 decimal will be mapped into the appropriate mousetext character. For example, to get the "running man" characters would require:

27 - mousetext-on control code
"F" - first part of "running man"
"G" - second part of "running man"

At the end of a sequence of mousetext characters, it is important to turn off mousetext with the mousetext-off control code. Any' characters not in the mousetext range will be displayed as is given the settings of the console driver.

3. Interface Description

3.1 Pascal

3.1.1 Data Interface

Both control codes and text to be displayed are passed to the driver as a contiguous array of data. For example, if the programmer wished to print "Hello" on line 10, column 15, in inverse, and then home the cursor and return back to normal text, s/he would create the following array of data (all numbers are decimal):

30	- absolute position
15	- parameter (column 15)
10	- parameter (line 10)
15	- inverse text
72	- "H"
101	- "e"
108	- "l"
108	- "l"
111	- "o"
25	- home cursor
14	- normal text

This array is not a string in the Pascal sense of the word, in that the first byte is data and not the length of the array (as in a string.) The console driver can accept an array up to 32767 bytes long (Pascal limit on integers).

The second required bit of data is an integer that denotes the length of the array to be processed by the driver. In the above example, the integer could either be a variable with the value 11 or the constant "11".

3.1.2 Calling the Console Driver

The driver is an "Attach" driver for Pascal. For information on Pascal Attach drivers, please refer to APPLE // PASCAL 1.2 DEVICE AND INTERRUPT SUPPORT TOOLS MANUAL. The unit number for the driver is #130.

To transfer data to the driver to be displayed on the screen, requires a UNITWRITE call from a Pascal program. The format for the call is shown below:

```
UNITWRITE(130, ARRAY_ADDR, LENGTH_ARRAY, MODE)
```

where 130 is the unit number for the driver

ARRAY_ADDR is a VAR parameter denoting the address of the array of data

LENGTH_ARRAY is the length of the array passed

MODE is the mode expression which is an integer. This can have four values:

value	DLE-expansion	Auto linefeed
0	TRUE	TRUE
2	FALSE	TRUE
8	TRUE	FALSE
12	FALSE	FALSE

When passing a string to the driver, it is important to always reference the string as:

```
STRING_VAR[1]
```

so as not to pass the length byte found in STRING_VAR[0].

3.1.3 Status Calls

The driver only accepts one status call that returns a data structure that describes the current state of the driver. (See section 2 for a description of these variables.) The form of the UNITSTATUS call is shown below:

```
UNITSTATUS(130, CON_STAT_BLK, 0)
```

where 130 is the unit number of the driver

CON_STAT_BLK is a record with the format:

```
TYPE BYTE = 0..255

VAR CON_STAT_BLK: PACKED RECORD OF
         CV:BYTE;
         CH:BYTE;
         WNDTOP:BYTE;
         WNDBOT:BYTE;
         WNDLFT:BYTE;
         WNDRGT:BYTE;
         WNDWTH:BYTE;
         WNDLEN:BYTE;
         CONWRAP:BYTE;
         CONADV:BYTE;
         CONLFD:BYTE;
         CONSCRL:BYTE;
         CONVID:BYTE;
         DLEFLAG:BYTE;
         CONFILL:BYTE;
         MOUSE:BYTE;
    END;
```

This call will instruct the driver to copy its values into this record so the programmer may inspect the current state of the driver.

3.1.4 Control Calls

The driver accepts four control calls. These calls allow the programmer to get the current location of the cursor, the text character at the current cursor location, or save and restore either the contents of the current viewport. The buffer in which this data is stored must be supplied by the programmer, it is not in the driver itself. For programs that do not require this function, this saves them space. It is recommended that the programmer allocate some space on the heap for this storage. This allows this space to be reclaimed as needed. To calculate the amount of space required for a viewport, multiply its width (WNDWTH) by its length (WNDLEN).

3.1.4.1 Getting the Current Cursor Position

To get the current location of the cursor on the text screen, the programmer can make a UNITSTATUS call of the form:

```
UNITSTATUS(130, LOCATION, 2);
```

where LOCATION is a record of the form:

```
LOCATION = RECORD
             HORIZONTAL: INTEGER;
             VERTICAL: INTEGER;
           END;
```

The driver will set these values equal to the screen coordinates, CH and CV. These are integer values. These values are not relative to the viewport but represent the actual column and line number.

3.1.4.2 Getting the Current Text Screen Character

By making a UNITSTATUS call of the form:

```
UNITSTATUS(130, CHARACTER, 8194);
```

where CHARACTER is a byte (0..255) variable,

the driver will return the current binary value of the character found at the current cursor location.

3.1.4.3 Saving and Restoring the Viewport

To save the contents of the viewport, requires a UNITSTATUS call of the form:

```
UNITSTATUS(130, VWPORT_BUF, 16386);
```

where 130 is the unit number for the driver

VWPORT_BUF is a buffer to hold the contents of the viewport.

To restore the contents of the viewport, requires a UNITSTATUS call of the form:

```
UNITSTATUS(130, SCREEN_BUF, 24578);
```

where 130 is the unit number for the driver

VWPORT_BUF is a buffer to hold the contents of the screen.

It is up to the programmer to keep track of which viewport has been saved in which buffer. When restoring a viewport, the programmer must have already set the required viewport prior to the restore call.

3.2 BASIC

The version of the console driver that is used with BASIC programs supports the following functions:

Output Data to the Console

Save the Current Viewport

Restore the Current Viewport

Get the Status of the Console Driver

Get the Current Cursor Position

Get the Current Text Screen Character

Initialize the Console Driver

Get A Segment of Memory

Get a Console Driver Error

Get the Console Driver Version

Get the Console Driver Copyright Notice

Release the Console Driver

The console driver functions are AMPERSAND ('&') routines.

3.2.1 Console Driver Functions

3.2.1.1 Calling the Console Driver

Calls the the Console Driver are done using the "Ampersand Hook". BASIC statements of the form:

```
&name(parameter list)
```

are used to call the Console Driver. Specific formats for the calls are described below.

3.2.1.2 Output Data to the Console

There are two calls to the driver to output data to the display. The first is of the form:

```
&WRTSTR(S$)
```

where S$ is a string

This call will output the contents of S$ to the display. S$ can include both control codes and ASCII characters.

The second form is:

```
&WRITE(I1%, I2%, SA%)
```

where SA$ is a one-dimensional string array and I1% is a starting index and I2% is an ending index

This call will output a sequence of strings contained in the string array SA$. The sequence begins with the string selected by the index I1% and will end with the string indexed by I2%. These strings can contain both control codes and ASCII characters.

3.2.1.3 Save the Current Viewport Contents

In order to save the contents of the viewport, a buffer must be allocated to store the contents. This is done through a call to the special function "Get memory" whose form is:

```
&GTMEM(P%, A%)
```

where P% is an integer specifies the number of pages (256 bytes) of memory to allocate and A% will be the address of that memory

This call allocates the number of pages required to store the viewport contents. The number of pages required can be calculated by

(WNDWTH * WNDLEN) / 256

rounding up to the nearest integer

For example to store the whole screen contents requires 8 pages to be allocated. The memory address of the memory allocated is returned in the variable A%. If the required number of pages is not available, then a BASIC "OUT OF MEMORY" error will occur.

Once a call to &GTMEM has been made, then a call to save the contents of the viewport can be made. The call is of the form:

```
&SVVP(A%)
```

where A% is the address returned from a call to &GTMEM

3.2.1.4 Restore the Current Viewport Contents

To restore the viewport contents, a call of the form:

```
&RSTRVP(A%)
```

where A% is the address used in the call to &SVVP

This will restore the previously saved contents to the viewport. The programmer must be careful to restore contents that are of the same size as the current viewport.

3.2.1.5 Get the Status of the Console Driver

To get the status of the console driver, a call of the form:

```
&CDINFO(CI%)
```

where CI% is a 16 element array, i.e.

```
DIM CI%(16)
```

This will return the contents of the status block to the array CI%. To inspect the contents, the following is a mapping of the array elements to the status block elements:

CI%(1)	=	CV
CI%(2)	=	CH
CI%(3)	=	WNDTOP
CI%(4)	=	WNDBOT
CI%(5)	=	WNDLFT
CI%(6)	=	WNDRGT
CI%(7)	=	WNDWTH
CI%(8)	=	WNDLEN
CI%(9)	=	CONWRAP
CI%(10)	=	CONADV
CI%(11)	=	CONLFD
CI%(12)	=	CONSCRL
CI%(13)	=	CONVID
CI%(14)	=	DLEFLAG
CI%(15)	=	CONFILL
CI%(16)	=	MOUSE

3.2.1.6 Get the Current Cursor Position

To get the current position of the cursor, a call of the form:

```
&GTCP(H%, V%)
```

where H% is the value of CH (x-position) and V% is the value of CV (y-position)

This call returns the absolute coordinates of the cursor.

3.2.1.7 Get the Current Text Screen Character

To get the value of the text character at the current cursor position, a call of the form:

```
&GTCHR(C%)
```

where C% is the character returned

This call returns the binary value of the text character at the current cursor position.

3.2.1.8 Initialize the Console Driver

To initialize the Console Driver to its default environment, a call of the form:

```
&INITCD
```

This call sets the driver environment to its default state described above.

3.2.1.9 Release Console Driver

To release the Console Driver Ampersand package and to restore the screen to a normal BASIC environment, a call of the form:

```
&STPCD(C%)
```

where C% is 40 to set up a normal 40-column display or 80 to set up a normal 80-column display

3.2.1.10 Console Driver Version and Copyright

To access the version number of the driver, a call of the form:

```
&CDVRSN(V%, R%)
```

where V% is the version number returned and R% is the revision number returned

To access the copyright notice of the driver, a call of the form:

```
&CDCPYRT(CM%)
```

where CM% is the copyright notice returned

3.2.1.11 Setting the Console Driver Address

Before the Ampersand package can use the Console Driver, it must have the location of the driver passed to it with the call:

```
&STCDADR(A%)
```

where A% is the starting address (which is also of the entry-point) of the console driver

This call must be made before any other calls to the Ampersand package.

3.2.2 Using the Console Driver with Your Program

A BASIC program using the console driver should do no console display through BASIC. All display should be done with the driver.

A sample use of the driver to place the string "Hello there" at position 10, 15 would be:

```
10 DIM ABS$(3)
```

```
20 DIM STR$(11)
30 ABS$(1) = CHR$(30): REM ABSOLUTE POSITION
40 ABS$(2) = CHR$(10): REM X COORDINATE
50 ABS$(3) = CHR$(15): REM Y COORDINATE
60 STR$ = "Hello there"
70 &WRTSTR(ABS$)
80 &WRTSTR(STR$)
```

3.2.3 Locating the Console Driver in Memory

The Console Driver is an EDASM produced REL file. This requires that it be relocated in memory before it can be used. Following the instructions in either the ProDOS or DOS Assembler Tools Manual, use RBOOT and RLOAD to perform the relocation.

3.3 Assembler

The version of the console driver that is used with assembly language programs supports the following functions:

Output Data to the Console

Save the Current Viewport

Restore the Current Viewport

Get the Status of the Console Driver

Get the Current Cursor Position

Get the Current Text Screen Character

Initialize the Console Driver

The console driver has a single entry point. Calling the driver is done in much the same way as ProDOS MLI calls.

3.3.1 Console Driver Functions

3.3.1.1 Calling the Console Driver

Calls to the console driver are done in much the same way as calls to the ProDOS MLI. The driver has only one entry point located at the beginning. Once the driver has been relocated in memory, its starting address is the entry point of the driver. A call is made as shown below:

```
JSR     PCONSOLE
DFB     COMMAND
DW      PARAMPTR
BNE     ERROR_HANDLER
```

The label PCONSOLE is the starting address of the driver. The programmer will determine this through deciding where to relocate the driver in memory. In the calling program there should be a ststement of the form:

```
PCONSOLE         EQU     nnnn
```

where nnnn is the starting address of the driver.

The JSR is followed by a byte that holds the command value which is a number that selects the appropriate console driver function. For specific values, see below.

Following the command value byte is a two byte pointer to a parameter list. The format for the parameter list verifies per console driver function. The specific formats are described below.

The driver will return to the caller with the carry flag clear if no error occured, or with the carry flag set if an error did occur. The calling program should check the carry flag (the BNE instruction shown above) and report an appropriate error. The actual error type is passed back to the caller in the A-register. The error handler can check this value to determine the specific error that occured.

3.3.1.2 Output Data to the Console

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     0               ;output to screen
        DW      OUTPUTDATA
```

PARAMETER LIST FORMAT:

```
OUTPUTDATA      DW      DATA1
                DW      LENGTH1
```

This call will output data (both text and control codes) to the console driver. The parameter list is a pointer to a data string followed by a length value.   For example, DATA1 would point to

```
DATA1   DFB     30      ;absolute position
        DFB     10      ;x position
        DFB     15      ;y position
        ASC     "Hello there!!"
```

```
LENGTH1 EQU     16      ;length of DATA1
```

This call returns no errors. The A-register value will be 0 and the carry flag will be clear.

3.3.1.3 Save the Current Viewport Contents

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     1               ;save viewport
        DW      SAVEBUFFER
```

PARAMETER LIST FORMAT:

```
BUFFERSIZE      EQU     1920    ;full screen

SAVEBUFFER      DS      BUFFERSIZE
```

This call will save the contents of the current viewport in the buffer pointed to in the call, in this case SAVEBUFFER. This buffer must be large enough to hold the entire contents of the viewport. The number of bytes required is equal to the width of the viewport (WNDWTH) times the length (WNDLEN). In the example shown above, the buffer is large enough to hold the contents of the entire screen (80 columns by 24 lines).

This call returns no errors. The A-register value will be 0 and the carry flag will be clear.

3.3.1.4 Restore the Current Viewport Contents

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     2               ;restore viewport
        DW      SAVEBUFFER
```

PARAMETER LIST FORMAT:

```
SAVEBUFFER      DS      BUFFERSIZE
```

This call will restore the contents of the current viewport from the buffer pointed to in the call, in this case SAVEBUFFER. The programmer should be careful that the viewport contents to be restored matches the size of the current viewport. A viewport can be defined, its contents saved, and then the viewport can be redefined as the same size but at a different location on the screen. Then the contents can be restored back to it. This gives the programmer the ability to move a viewport and its contents around the screen.

This call returns no errors, the A-register is 0 and the carry flag is cleared.

3.3.1.5 Get the Status of the Console Driver

CALLING FORMAT:

```
          JSR      PCONSOLE
          DFB      3             ;get status
          DW       STATUSBLK
```

PARAMETER LIST FORMAT:

```
STATUSBLK          EQU      *

CV        DFB      0
CH        DFB      0
WNDTOP    DFB      0
WNDBOT    DFB      0
WNDLFT    DFB      0
WNDRGT    DFB      0
WNDWTH    DFB      0
WNDLEN    DFB      0
CONWRAP   DFB      0
CONADV    DFB      0
CONLFD    DFB      0
CONSCRL   DFB      0
CONVID    DFB      0
DLEFLAG   DFB      0
CONFILL   DFB      0
MOUSE     DFB      0
```

This call will return the current status of the console driver in the status block pointed to in the call, in this case STATUSBLK. The programmer must insure that the status block used matches this description exactly or data may be destroyed if the status block is smaller than the one described.

This call returns no errors, the A-register will be 0 and the carry flag will be clear.

3.3.1.6 Get the Current Cursor Position

CALLING FORMAT:

```
          JSR      PCONSOLE
          DFB      4             ;get cursor position
          DW       CURSORPOS
```

PARAMETER LIST FORMAT:

```
CURSORPOS        EQU      *

XPOS     DFB     0
YPOS     DFB     0
```

This call will return the absolute screen coordinates of the current cursor position. XPOS is the column and YPOS is the line. These values correspond the values of CH and CV described above.

This call returns no errors, the A-register will be 0 and the carry flag will be clear.

3.3.1.7 Get the Current Text Screen Character

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     5             ;get text character
        DW      TEXTCHAR
```

PARAMETER LIST FORMAT:

```
TEXTCHAR        DFB     0
```

This call will return the binary value of the text character located at the current cursor position. This value will reflect whether or not the character is inverse, normal, or mousetext. It is up to the calling program to decipher the value.

This call returns no errors, the A-register will be 0 and the carry flag will be clear.

3.3.1.8 Initialize the Console Driver

CALLING FORMAT:

```
        JSR     PCONSOLE
        DFB     6             ;initialize
        DW      0
```

PARAMETER LIST FORMAT:

No parameter list required.

This call will set the console driver back to its default state. No parameter list is required.

This call returns no errors. The A-register will be 0 and the carry flag will be clear.

3.3.2 Using the Console Driver with Your Program

3.3.2.1 Console Driver Zero Page Usage

The console driver uses zero page locations $20 to $40. The contents of these locations are saved when the driver is called and restored upon exit.

3.3.2.2 Console Driver Softswitch Usage

The console driver uses certain softswitches to control its use of the display memory. They are:

80COL ($C00D) - turn on 80-column card

80STORE ($C001) - use auxilary memory for display

PAGE2 ($C055, $C054) - to switch between even and odd locations on the 80-column card

ALTCHARSET ($C00F) - to use alternate character set

When the console driver is called these switches are set to their appropriate value. Since the console driver is intended to be the SOLE means by which console display is managed, these switches are NOT reset when the driver returns to the calling program. It is up to the program to reset back to the normal environment.

3.3.2.3 Relocating the Console Driver

The Console Driver is an EDASM produced REL file. This requires that it be relocated in memory before it can be used. Following the instructions in either the ProDOS or DOS Assembler Tools Manual, use RBOOT and RLOAD to perform the relocation.

EXTERNAL REFFERENCE SPECIFICATION

APPLE // PASCAL "FILECARD" MENU SUPPORT UNIT

Neal Johnson

Novemeber 10, 1984

Final Release Version

TABLE OF CONTENTS

"Filecard" Menu Support Unit

1. Introduction

The Apple // Pascal "Filecard" Menu Support Unit (herein known as the unit) is an implementation of a simple "filecard" style human interface similar to that used in the products Appleworks and Access II. This unit allows a Pascal-based application to use a "filecard" style human interface without the programmer having to implement the details of such an interface.

The unit requires the Apple // Console Driver to be present in order to function (see the APPLE // CONSOLE DRIVER E.R.S.).

The unit is an intrinsic unit that can be either in SYSTEM.LIBRARY or in a program library (Pascal 1.2 128K system).

The unit supplies data structures to define and manage a "filecard" style, hierarchical menu structure as well as the routines necessary to perform the required functions of such a human interface.

This document describes the interface in detail, and explains how to go about designing an application that will use this style of interface. It then descibes the unit used to implement the interface in an application and a sample program that uses that uses the interface.

2. "Filecard" Menu Description

2.1 The Screen Layout

The screen layout for the "filecard" style interface is divided into three portions:

The Top

The "Filecard" Area

The Bottom

Each of these areas serves a primarily different function within the scope of the whole human interface. See Figure 2.1 for a picture of the layout.

There are routines in the unit to manage each of these portions, either as a whole or in parts.

2.2 The Top Portion

The top portion occupies lines 0 - 2 (three lines) on the screen. The top line (0) is divided into three portions known as the Left, the Middle, and the Right. Line 1 is left blank. Line 2 is a line of "_" dividing this portion of the screen from the center portion. See Figure 2.1 for an illustration.

The top portion is used in conjunction with the "filecard" area to display information about where the user is in the hierarchy of menus.

The Left hand side is primarily used for application specific information, such as the name of the application, the current disk drive selected, the file name of a selected file, etc. The contents for this side are up to the application.

The Middle section is used to display the title of the currently selected filecard. This section changes as other filecards are selected. The unit manages this area for you during the menu selection process (see below).

The Right hand side is used to display the "escape route" for the user. During the menu selection process, if a selection calls up a filecard (lower in the hierarchy), this section will display the name of the previous filecard. See Figures 2.4.1 to 2.4.4. When a filecard is displayed, typing an ESCAPE will revert back to the previous (or higher level) filecard. This section is also managed for you during the selection process.

2.3 The Bottom Portion

The Bottom portion occupies lines 21 - 23 (three lines). Line 21 is a line of "_" to divide this portion from the center section. Line 22 is used to display text. For the sample program described below, line 23 is left blank. See Figure 2.1 for illustration.

The Bottom portion is divided into two sections, the Right and the Left. This portion of the screen is used primarily to give the user instructions, such as what things to type during the selection process, or as an area for input. The Right section is used for these types of activities. The Left section is used for application specific information; its content being left up to the programmer. The unit supplies routines to manage this portion of the screen.

2.4 The "Filecard" Area

The "Filecard" area of the screen occupies the center portion, from line 3 to line 20 (18 lines). It is used for the display of filecards during the selection process. The unit allows up to four levels of filecards to be displayed at one time (each card overlays the previous card.) See Figures 2.4.1-4 for illustrations of the different levels and the interaction with the top portion of the screen.

The unit supplies the necessary routines to manage this area of the screen.

2.5 Error Boxes

The unit supplies a simple mechanism to put error messages on the screen, overlaying the current screen contents. An error box is placed on lines 12 - 16 and between columns 10 and 71. A viewport is defined within the error box that allows for up to three lines of text, up to 59 characters long. When an error box is displayed, the console will beep. There are two routines to support error boxes, one to display them and one to remove them from the display restoring the previous screen contents. See Figure 2.5 for an illustration.

3. "Filecards"

3.1 Introduction

The primary "pictorial" framework for a menu in this style of human interface is called a "filecard". This is not to suggest that a menu is like a filecard. The name is a result of the shape of the menu and not its function! A "filecard" is a rectangular box with a tab on the left top edge which holds the name of the menu displayed. The actual menu items are listed within the box.

It is suggested that the name of the "filecard" represent the generic relationship of the menu items. For example, a set of menu items dealing with files could have the name "File Activities".

Each menu item can be selected by the user of the program. The action taken may select another menu which causes another "filecard" to be displayed or it may result in performing some function which does not use the "filecard" interface. A menu item which selects another menu is called a "menu selector". A menu item that selects a function to be performed is called an "action selector". See Figure 3.1 for an illustration.

3.2 How to Design a Hierarchical Menu Structure

When designing an application, one of the most difficult problems is the design of the human interface. Using this unit makes designing the "look" of the interface quite simple. As an application developer, however, you are still faced with designing how you want to split the different activities that can be performed in your application into a series of menus and actions.

One approach that makes this type of designing manageable, is to conceptualize the actions in a "hierarchical" manner. For example, if your application supports a set of 4 major activities:

File Management

Printing

Configuration

Doing the Real Work

these become the menu items on the top most "filecard". Selecting any one of these would then display another "filecard" with items appropriate for that activity.

Selecting "File Management" would then display a "filecard" one level down with the following menu:

Create a File

Delete a File

Catalog a Disk

Rename a File

Selecting any of these items could either display another "filecard" or branch off to do the activity selected.

For every activity supported by your application, you would define a "path" to that activity moving from a general description such as "File Management" to a specific description such as "Create a File." This path moves through a hierarchy of "filecards" and menus. The idea behind this style of interface is that it helps lead the user to the activity they wish to perform. In some cases s/he may not know the actual "name" for the activity, but s/he knows the general type of activity it is. Using this style of human interface allows the application to present the range of activities in such a way that the user can find his/her way to the the desired goal.

As you specify the types of activities and the menus used to select them, it helps to draw a picture of the emerging "menu tree". Figure 3.2.3 shows such a drawing. It is from this drawing that you then design the initialization procedures for the actual "filecards" and menus in your program and the main body of your program where the selection process takes place.

3.3 Card Numbers

Each "filecard" is assigned a number that is used for identification. The number has no other meaning. When the "menu tree" is designed, a number can be assigned. See Figure 3.3 for an example. These numbers are used to refer to individual cards in a program. The numbers assigned are arbitrary, but they must be sequential starting from 0 to the highest numbered card. Card #0 is a special card that exists only as a placeholder for the "escape path" for card #1. For details see below.

3.4 Card Levels

The "menu tree" defines a set of levels where each card resides. More than one card can be at a particular level. The topmost card is level 1. There is only one card at this level. "Filecards" that are displayed as a result of selecting a menu item from the topmost card (level 1) are level 2 cards. "Filecards" displayed as a result of selecting an item at level 2 are level 3 cards, and so on. The unit only supports up to four levels of "filecards", i.e. only four "filecards" can be displayed at one time on the screen. See Figure

2.4.4 for an illustration. Since each "filecard" can have up to 9 menu items this allows for up to 6551 different "filecards" for one application!

3.5 Card Titles

Each "filecard" has a title which is displayed in the tab on the upper left corner of the card. This title can be up to 22 characters in length. The title should reflect the nature of the menu displayed in the card. The title is also displayed in the middle of the top portion of the screen. This represents "where" the user is in the "menu tree". See Figure 3.1 for an example.

3.6 The "Filecard" Data Structure

The unit supplies a data structure to represent each "filecard" used by your application. The format for the data structure is:

```
A_CARD = PACKED RECORD
           MENU_NUMBER: INTEGER;
           MENU_LEVEL: 0..4;
           P_CARD: INTEGER;
           MENU_TITLE: STR22;
         END;
```

The MENU_NUMBER is simply the number you have assigned to the card. The MENU_LEVEL is the level in the "menu tree" of the card. This can have a value between 1 and 4. The value of 0 is a special case. It represents the "top_most" card which is not displayed. It is used to specify the right hand side of the top display to show what happens when the user types ESCAPE at the level 1 card. The integer P_CARD is the card number of the previous card in the "menu tree". This value is used to update the top display Escape path. The MENU_TITLE is a string whose length is limited to 22 characters. This is the name of the "filecard" displayed in the tab on the left hand side.

In your application you should define an array of A_CARD's, one for each "filecard" in your "menu tree". Using your "menu tree" diagram, define a procedure in your program to initialize this data structure. The SAMPLE program provided as an appendix shows such a procedure (set_cards).

4. "Filecard" Menus

4.1 Introduction

Each "filecard" presents the user with a menu of items. The user then selects one of the items to perform. The user uses the UP or DOWN ARROW keys to move through the items, or they can type the item number displayed in the menu to choose an item. Once an item is chosen, they then type RETURN to select that item.

4.2 Menu Description

4.2.1 Menu Items that Select other Menus - "Menu Selectors"

Certain menu items will select another menu to be displayed on another "filecard". This new "filecard" will be one level lower than the "filecard" displaying the item selected. These menu items are called "menu selectors". They do not perform any other action than to specify the next "filecard" to be displayed.

4.2.2 Menu Items that Select Operations - "Action Selectors"

Other menu items select an action to be performed. These items are called "action selectors". In this case, the application branches off to perform some action that has been selected. Here the application may prompt the user for input, display new information for the user, or other such things. In most cases, this implies that the screen display of "filecards" will go away for the duration of that activity. When the activity is done, the application should return to the original "filecard" display shown prior to branching into the activity. Though the unit does not supply the means of performing the activities for your application, it does supply the means of selecting these activities and for coming back to the original "filecard" display. The SAMPLE program described below shows how this is done.

4.3 The Menu Item Data Structure

The unit supplies a data structure to define a single item in the menu. This is the MENU_ITEM data structure, whose format is:

```
MENU_ITEM = PACKED RECORD
              DO_POSITION: BYTE;
              XPOS: BYTE;
              YPOS: BYTE;
              STATE: BYTE;
              DSPLY_TEXT: STR60;
            END;
```

The first three bytes of the record contain control codes used by the Apple // Console Driver to do an Absolute Position. DO_POSITION holds the control code for absolute position, XPOS has the x-position value and YPOS has the y-position value. These are used to position the menu item in the "filecard" for display. The STATE value specifies whether or not the item is displayed in normal text or inverse text. This is used during the selection process. The DSPLY_TEXT is a string of up to 60 characters which is the actual text of the menu item that is displayed. The unit supplies an initialization routine to set up the values of DO_POSITION, XPOS, YPOS, and STATE (INIT_A_MENU).

This record holds all the information necessary to print it at a given location (XPOS, YPOS) in the current viewport, which is the inside of the current "filecard" displayed on the screen. The text of the item (DSPLY_TEXT) is printed either in normal or inverse depending on the control code in the variable STATE. For details on the control codes used (DO_POSITION and STATE) see the Apple // Console Driver ERS.

4.4 The Menu Data Structure

A set of menu items belonging in one menu are linked togther via another unit-supplied data structure, A_MENU. The format is shown below:

```
A_MENU = PACKED RECORD
           NUM_ITEMS: 1..9;
           CURRENT_ITEM: INTEGER;
           LIST: ARRAY[1..9] OF MENU_ITEM;
         END;
```

NUM_ITEMS specifies the number of menu items in this menu. This can be between 1 and 9. LIST is simply the list of menu items for this menu. The field CURRENT_ITEM is used to maintian the number of the most recently selected item in the menu. This is done so that when a user "escapes" back to a menu, the unit can display the item last selected as highlighted.

In your application you should define an array from 1 to the number of "filecards". For example,

```
MENU: ARRAY[1..9] OF A_MENU;
```

Each element of this array corresponds to one of the "filecards" you have defined. The index into this array is to match the number of the card. Thus MENU[3] specifies the menu to be displayed with the card whose number is 3.

Each element in MENU (MENU[1], MENU[2], ...) requires initialization. In your program you should set up a procedure to set up this array. If you used the INIT_A_MENU procedure, the only elements that require your input are:

```
MENU[n].NUM_ITEMS - gets the number of items for this menu
MENU[n].LIST[nn].DSPLY_TEXT - gets the text for the menu item
```

The SAMPLE program found in the appendix illustrates this procedure (P1_SET_MENU_TEXT and P2_SET_MENU_TEXT.)

5. The "Filecard" Menu Support Unit

5.1 Introduction

The "Filecard" Menu Support Unit supplies the necessary routines to support a simple "filecard" style human interface. In includes data structure definitions to help in setting up the necessary information about the "menu tree" and procedures to help initialize these data structures.

There are 10 low level routines that allow access to the console driver to support the display of text in conjuction with the unit.

The rest of the routines are designed to help set up the interface, display and remove "filecards" from the display, and get the user's selection from the menu.

Many of the details described below are based on the Apple // Console Driver E.R.S. You should be familiar with its contents before reading this section.

5.2 Unit Interface Data Structures

5.2.1 Console Driver Control Codes

The unit supplies as constants the set of console driver control codes. These can be used by the program to perform other console display activities. The list is:

NOOP	=	0
SAVEVP	=	1
SETVP	=	2
CLRBOL	=	3
RESTVP	=	4
BELL	=	7
CURLFT	=	8
CURDWN	=	10
CLREOV	=	11
CLRVP	=	12
CURRET	=	13
NORMAL	=	14
INVERSE	=	15
DLE	=	16
HORSHFT	=	17
VPOS	=	18
CLRBOV	=	19
HPOS	=	20
CMCONT	=	21
SCRDWN	=	22
SCRUP	=	23
MOFF	=	24
HOME	=	25
CLRLINE	=	26
MON	=	27
CURRGT	=	28
CLREOL	=	29
APOS	=	30
CURUP	=	31

See the Apple // Console Driver for details on these control codes.

5.2.2 Constant Declarations

The following constants are defined in the unit, though in most cases they are not needed in a program using the unit:

The is the console driver unit number:

```
CONSOLE = 130
```

The following are the required control mode values for UNITSTATUS calls to the console driver:

```
GET_STATUS = 0
GET_CURSOR = 2
SAVE_VP_CONTENTS = 16386
REST_VP_CONTENTS = 24578
```

See the Apple // Console Driver ERS for details on these control mode values.

5.2.3 Type Declarations

5.2.3.1 CON_STAT_BLK

The console driver has a status call which returns the current status of the driver. This record defines this status information:

```
CON_STAT_BLK = PACKED RECORD
                   CV: BYTE;
                   CH: BYTE;
                   WNDTOP: BYTE;
                   WNDBOT: BYTE;
                   WNDLFT: BYTE;
                   WNDRGT: BYTE;
                   WNDWTH: BYTE;
                   WNDLEN: BYTE;
                   CONWRAP: BYTE;
                   CONADV: BYTE;
                   CONLFD: BYTE;
                   CONSCRL: BYTE;
                   CONVID: BYTE;
                   DLEFLAG: BYTE;
                   CONFILL: BYTE;
                   MOUSE: BYTE;
               END;
```

See the Apple // Console Driver E.R.S. for a complete description of these fields.

The unit has a procedure (GET_CON_STATUS) which will perform the status call.

5.2.3.2 POSITION

This record defines the data structure by which the current cursor position can be read via a status call to the console driver:

```
POSITION = RECORD
             XPOS: INTEGER;
             YPOS: INTEGER;
           END;
```

where

XPOS is the absolute x-position of the cursor

YPOS is the absolute y-position of the cursor

5.2.3.3 MENU_ITEM

The record MENU_ITEM defines a single item in one menu:

```
MENU_ITEM = PACKED RECORD
              DO_POSITION: BYTE;
              XPOS: BYTE;
              YPOS: BYTE;
              STATE: BYTE;
              DSPLY_TEXT: STR60;
            END;
```

where

DO_POSITION is the control code for absolute position

XPOS is the x-position

YPOS is the y-position

STATE denotes whether the menu item is normal or inverse text (14 = normal, 15 = inverse)

DSPLY_TEXT is a string up to 60 characters in length that is the text for the menu item

5.2.3.4 A_MENU

This record defines a complete menu for a single "filecard". Its format is:

```
A_MENU = PACKED RECORD
```

```
          NUM_ITEMS: 1..9;
          CURRENT_ITEM: INTEGER;
          LIST: ARRAY[1..9] OF MENU_ITEM;
        END;
```

where

NUM_ITEMS is the number of menu items to be displayed (from 1 to 9)

CURRENT_ITEM is the number of the most recently selected item

LIST is the list of menu items (from 1 to 9) for this particular menu

5.2.3.5 A_CARD

This record defines a "filecard". Its format is:

```
A_CARD = PACKED RECORD
           MENU_NUMBER: INTEGER;
           MENU_LEVEL: 0..4;
           P_CARD: INTEGER;
           MENU_TITLE: STR22;
         END;
```

where

MENU_NUMBER is the number assigned to this "filecard" and its menu

MENU_LEVEL is the level assigned to this card (see above for a description of levels)

P_CARD is the menu_number of the previous card in the "menu tree"

MENU_TITLE is the title for this card, a string no greater than 22 characters in length

5.2.3.6 SCREEN_BUFFER

This type defines a buffer that can store one screen's worth of data. It is used to temporarily store the contents of the screen or a viewport when using a Save Viewport or Restore Viewport control call to the console driver.

```
SCREEN_BUFFER = PACKED ARRAY[1..1920] OF BYTE;
```

5.2.3.7 OUTPUT_BUFFER

This type defines an output buffer where data is placed prior to writing it out to the console driver.

```
OUTPUT_BUFFER = PACKED ARRAY[0..1023] OF BYTE;
```

5.2.3.8 ERROR_BUFFER

This type defines a smaller buffer where the screen contents behind an error box are stored so that the screen can be "re-painted" after an error box is removed from the screen.

```
ERROR_BUFFER = PACKED ARRAY[1..310] OF BYTE;
```

5.2.3.9 STR22 and STR60

For the strings used in some of the data strucutures defined in the unit, the following special string definitions are used:

```
STR22 = STRING[22]

STR60 = STRING[60]
```

5.2.4 Variable Declarations

5.2.4.1 SAVE_BUFFER

This is the buffer that is used by all saves and restores of the viewport . It is large enough to store a full screen (80 columns by 24 lines) of data.

```
SAVE_BUFFER: SCREEN_BUFFER;
```

5.2.4.2 ERR_BUFF

This is the buffer used by the errorbox routines to store the information overwritten by the errorbox on the screen so that it can be restored.

```
ERR_BUFF: ERROR_BUFFER;
```

5.2.4.3 BUFFER

This is the buffer used to collect data prior to writing it out to the console driver. All the routines in the unit that write to the console use this buffer.

```
BUFFER: OUTPUT_BUFFER;
```

5.2.4.4 BUFF_P

This is the global pointer into the output buffer, BUFFER. It is used as an index into the array. When data is placed into the array directly, this pointer must be incremented the appropriate number of times to reflect the number of bytes entered. There is a procedure, RESET_BUFFP, that will set it to 0, to reset the buffer for new input.

BUFF_P: INTEGER;

5.2.4.5 STATUS_BLK

This is a record to hold the status information for the console driver. Its format is described above and in the Apple // Console Driver E.R.S.

STATUS_BLK: CON_STAT_BLK;

5.2.4.6 MODE

All data output to the console driver is done via a UNITWRITE statement. This call requires a "mode expression" to control automatic DLE-expansion and/or automatic linefeeds. Normally, this value is 0 but it can be set for the procedure WRITE_BUFFER (described below) by setting this integer value, MODE. The values are their meanings are:

value	DLE-expansion	Auto linefeed
0	TRUE	TRUE
2	FALSE	TRUE
8	TRUE	FALSE
12	FALSE	FALSE

Any other values will result in undefined states. Changing this value while using the unit's functions can result in poor performance by the unit! If you change it for your own purposes, set it back to zero before calling the unit.

The unit supplies a procedure SET_MODE (5.2.5.9) that will properly set this value. The default setting used by the unit is DLE true and Auto-linefeed true.

5.2.5 Functions Available

5.2.5.1 PUT_CONTROL

CALL FORMAT:

PUT_CONTROL(CONTROL);

where CONTROL is an integer value that represents a control code for the console driver.

This procedure will place a console driver control code in the output buffer, BUFFER, and will increment BUFF_P. For example, to set up an absolute position control sequence, a program would have the following calls:

```
PUT_CONTROL(APOS);
PUT_CONTROL(NEW_X);
PUT_CONTROL(NEW_Y);
```

where NEW_X and NFW_Y are integer values corresponding to the x and y coordinates that the program wishes to move the cursor

5.2.5.2 PUT_STRING

CALL FORMAT:

PUT_STRING(A_STRING);

where A_STRING is a string (0 to 80 characters in length.

This procedure places a string in the output buffer, BUFFER, and increments the pointer BUFF_P the length of the string. For example,

```
PUT_STRING('This is a string to display!');
```

5.2.5.3 RFSET_BUFFP

CALL FORMAT:

RESET_BUFFP;

This procedure resets the value of BUFF_P to 0 which effectively clears the output buffer, BUFFER, of data. Before setting up a new buffer-full of data, this procedure should be called.

5.2.5.4 WRITE_BUFFER

CALL FORMAT:

WRITE_BUFFFR;

This procedure will write the current contents of the output buffer, BUFFER, to the console driver. The number of bytes written is equal to the current value of the pointer BUFF_P. After the buffer is written, BUFF_P is set to 0.

5.2.5.5 GET_CON_STATUS

CALL FORMAT:

```
GET_CON_STATUS;
```

This procedure will make a status call to the console driver and return the current status information in the data structure, STATUS_BLK, where the calling program can inspect it.

5.2.5.6 VP_SAVE

CALL FORMAT:

```
VP_SAVE(SCR_BUF);
```

where SCR_BUF is a byte array large enough to hold the number of characters contained in the current viewport.

This procedure will save off the contents of the current viewport into a buffer. It is critical that the buffer be large enough to hold the number of characters in the viewport. This number can be calculated via a GET_CON_STATUS call and then multiplying the values of WNDLEN and WNDWTH.

5.2.5.7 VP_RESTORE

CALL FORMAT:

```
VP_RESTORE(SCR_BUF);
```

where SCR_BUF is a byte array large enough to hold the number of characters contained in the current viewport.

This procedure will restore the contents of the current viewport from the buffer where they were previously saved via a VP_SAVE call. It is critical that the buffer have the same number of bytes of data as the size of the current viewport. It is not important that the viewport occupy the same absolute position on the screen.

5.2.5.8 GET_POSITION

CALL FORMAT:

```
GET_POSITION(CUR_POS);
```

where CUR_POS is a record of the type POSITION

This procedure will return the current absolute position of the cursor in the console driver.

5.2.5.9 SET_MODE

CALL FORMAT:

SET_MODE(DLE, LFD);

where DLE and LFD are Boolean values

This procedure will set the MODE variable to the appropriate value given the settings of the DLE and LFD parameters. If DLE is TRUE then DLE-expansion will be set, otherwise it will be reset. If LFD is TRUF than Auto-linefeed will be set, otherwise it will be reset.

5.2.5.10 MIDDLE_UPDATE

CALL FORMAT:

MIDDLE_UPDATE(STR);

where STR is a string

This procedure will update the middle portion of the top display. It is used in conjuction with RIGHT_UPDATE (see below) to update the top portion of the screen during the selection process as "filecards" are displayed. This procedure is used to place the title of the current "filecard" on the screen. This procedure will clear out the right portion of the top, requiring it to be updated also.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.11 RIGHT_UPDATE

CALL FORMAT:

RIGHT_UPDATE(STR);

where STR is a string

This procedure will update the right hand portion of the top display. It has no effect on the remaining part of the top display.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.12 MAKE_CARD

CALL FORMAT:

MAKE_CARD(CURRENT_CARD, PREVIOUS_CARD);

where CURRENT_CARD and PREVIOUS_CARD are A_CARD's

This procedure will display a "filecard" on the screen. The card displayed will be CURRENT_CARD. The level of this card will determine its placement on the screen. As well as displaying the "filecard" (only the outline and title, the menu is not displayed at this time) this procedure will update middle potion of the top with the title of the current card. Using the PREVIOUS_CARD record it will also update the right hand side of the top with the "escape path".

Upon exiting this procedure, the viewport will be set to the inside of the "filecard" on the screen.

5.2.5.13 REMOVE_CARD

CALL FORMAT:

REMOVE_CARD(LEVEL, PREVIOUS_CARD, PRE_ESCAPE_CARD);

where LEVEL is a value between 1..4, PREVIOUS_CARD and PRE_ESCAPE_CARD are A_CARD's

This procedure will remove the current card from the display, and then display the previous card (to the current card in the "menu tree") on the screen. LEVEL is the level of the current card. PREVIOUS_CARD is the card record of the card previous to the current card. PRE_ESCAPE_CARD is the card previous to the previous card! The description below of the sample program will make clear the use of this procedure. This procedure is used when the user types an escape during the selection process to go back to the previous "filecard".

Upon exiting this procedure, the viewport will be set to the inside of the new "filecard" on the screen.

5.2.5.14 MAKE_TOP

CALL FORMAT:

MAKE_TOP(LEFT, MIDDLE, RIGHT);

where LEFT, MIDDLF, RIGHT are strings

This procedure will put the top display on the screen, placing the LEFT string left-justified on the first line, centering the MIDDLE string, and right-justifying the RIGHT string. The second line is left blank, and a line of "_" is then drawn.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.15 MAKE_BOTTOM

CALL FORMAT:

MAKE_BOTTOM(LEFT, RIGHT);

where LEFT and RIGHT are strings

This procedure contructs the bottom portion of the display. The LEFT string is left-justified and the RIGHT string is right-jusitified on line 22. Line 21 is a line of "_" and line 23 is left blank.

This procedure makes a save-viewport control call to the console driver. The calling program should not assume that any viewports it has saved prior to this call will remain "restorable". Before exiting, this procedure will restore the previous viewport setting. The console driver only supports one level of save for viewports. Internal to the unit, this does not matter. Any program using the unit and is also making its own calls to the console driver (Save and Reset Viewport, Restore Viewport) should be aware of this.

5.2.5.16 CLEAR_SCREEN

CALL FORMAT:

CLEAR_SCREEN;

This is a general procedure to clear the entire screen. The viewport is left set to the entire screen. This is used primarily to clear the screen at the beginning of a program and at the end.

5.2.5.17 INIT_A_MENU

CALL FORMAT:

INIT_A_MENU(MENU_LIST);

where MENU_LIST is A_MENU

This procedure will set up the initial values of the following fields in a menu record. Those fields are:

DO_POSITION - set to control code for absolute position

XPOS - set to 1 (second column in viewport)

YPOS - set from 1 to 9 depending on the number of menu items in the list

STATE - for item 1 set to INVFRSF, for the other items set to NORMAL

The unit actually uses these data structures to paint the menu items in the "filecard". The absolute position control code and the XPOS and YPOS values determine where the text is placed. The STATE value determines whether or not the text is in INVERSE or normal text. This procedure defaults to displaying the menu as a single-spaced list in the "filecard". For example,

```
1. First menu item
2. Second menu item
3. Third menu item
.
.
.
9. Last menu item
```

A program can modify the values of XPOS and YPOS to control the positioning of the menu items in the "filecard".

The procedure also sets the field CURRENT_ITEM to 1.

5.2.5.18 GET_SELECTION

CALL FORMAT:

```
SELECTED := GET_SELECTION(MENU_LIST, SELECT_NUM,
              SHOW_MENU);
```

where MENU_LIST is A_MENU, SELECT_NUM is a VAR parameter to return the number of the item selected, and SELECTED is a program supplied BOOLEAN variable; SHOW_MENU is a BOOLEAN that specifies whether or not the menu display needs to be updated, if TRUE update the display, FALSE don't update the display

This is the main procedure to handle the complete selection process for a menu displayed in a "filecard". Once a card has been displayed via a MAKE_CARD call, GET_SELECTION is then called with the MENU_LIST for the current card. This call will paint the menu list in the "filecard" on the screen, with the first menu item in inverse. If the variable SHOW_MENU is FALSE, the menu will not be displayed. This assumes that the menu is already present on the screen.

At this point, the user can type one of the following things:

UP-ARROW - will move to the next item above in the list

DOWN-ARROW - will move to the next item below in the list

a number - typing a number will move to the item with that number in the list

RETURN - will return the number of the item currently in INVERSE (selected) in the variable SELECT_NUM and GET_SELECTION will return true

ESCAPE - will return 0 in SELECT_NUM and GET_SELECTION will return false

typing anything else (or a number not included in the list) will cause a beep

A menu item displayed in inverse is considered to be the "chosen" item. To select that item requires the user to type RETURN. Moving to an item either with the arrow-keys or typing a number constitutes choosing an item.

After GET_SELECTION returns it is up to the calling program to act on the choice. The description of the SAMPLE program below will illustrate how this is done.

5.2.5.19 ERROR_BOX

CALL FORMAT:

```
ERROR_BOX;
```

This procedure will place an error box on the screen, saving the screen contents behind the box. See section 2.5 for a description of an error box.

This procedure will do a save viewport control code. Any previously saved viewport specifiction will be lost. Internal to the unit this does not matter. Any viewport set by the calling program will have to be managed by that program.

It is up to the calling program to place any text in an error box. This has to be done with calls to the console driver (Pascal has no knowledge of the current screen state!) The current viewport is set to the inside of the error box, so all display will "automatically" take place there.

5.2.5.20 GO_AWAY_ERROR

CALL FORMAT:

```
GO_AWAY_ERROR;
```

This procedure must be used in conjunction with ERROR_BOX. Once an error box has been displayed, along with an error message (supplied by the calling program), the box can be removed from the screen by calling this procedure. It will restore the original screen contents and will reset the viewport to the values saved at the time of the call to ERROR_BOX.

Any text in an error box is removed by this call. The calling program does not need to remove it itself.

5.2.5.21 RESET_CARD_VP

CALL FORMAT:

```
RESET_CARD_VP(CARD_REC);
```

where CARD_REC is A_CARD

This procedure is used to set the viewport back to the inside of a "filecard" (CARD_REC) on the screen. It is used to update the screen during the selection process as cards are removed and replaced on the screen. See below for details on its use.

5.2.6 Using the Unit with Your Program

To use the unit requires a USES statement in your program of the form:

```
USES {$U library} FILECARD;
```

where library is the name of the library file where the unit is located.

5.3 A Sample Application that Uses the Unit

As an appendix, there is a listing of a sample program that utilizes the unit as the primary human interface code. This program has 9 different "filecards" arranged in the "menu tree" in Figure 3.3. Each menu has between two to five items, some of which point to other menus and others which branch off to "pseudo" activities. This program illustrates the type of data structures that are used to create the "menu tree", how to initialize the data, and how to organize the "main loop" in the program which controls the selection process.

5.3.1 Setting up the "Filecards"

The program defines an array of A_CARD which designates the "filecards" used by the program.

```
CARD: ARRAY[0..9] OF A_CARD;
```

The zeroth element of the array is used only to store a string (the menu_title) that is displayed for the topmost card's escape path. Elements 1 through 9 are the actual "filecards" that are displayed. For any program there should be an array, like that above, with 0..number_of_filecards.

The following fields in each A_CARD need to be initialized as follows:

MENU_NUMBER - the number assigned by the programmer

MENU_LEVEL - the level 1..4 in the "menu_tree"

P_CARD - the menu_number of the previous card in the menu_tree

MENU_TITLE - the title displayed in the upper left corner of the "filecard"

The procedure SET_CARDS shows such an initialization process. Most of the details used should have already been worked out in the initial design of the "menu tree".

5.3.2 Setting up the Menus

The program defines another array to hold the information about each of the menus associated with the "filecards".

```
MENU: ARRAY[1..9] OF A_MENU;
```

Each element 1 to 9 corresponds directly with each element 1 to 9 of the array CARD.

The first step is to initialize the fields that control the display of the menu items (DO_POSITION, XPOS, YPOS, and STATE). This is done through a call to the procedure INIT_A_MENU in the unit. Since there are nine menus to set up, a simple FOR-LOOP does the trick:

```
FOR I := 1 TO 9 DO INIT_A_MENU(MENU[I]);
```

This is found in the procedure INIT_THE_MENUS in the sample program.

The next step is to specify the number of items for each menu and the text to be displayed for each item in the menu. The procedures P1_SET_MENU_ITFMS and P2_SET_MENU_ITFMS show this process. (There are two procedures because of the size limitiations for the amount of code generated in a procedure!)

The standard set up used here will result in a single spaced list displayed left-justified in each "filecard". You can modify this by changing the values of XPOS and YPOS in each MENU_ITEM. When changing these values, remember that the x and y values are treated relative to the viewport coordinates.

5.3.3 The Main Body of the Program

Other than the calls to the initialization procedures, the main body of the program consists of a large REPEAT statement that contains a large CASE statement. This CASE statement controls the flow of action during the selection process.

5.3.3.1 The Selection Process

There are three types of action that occur during the selection process:

Display the next menu in the "menu tree" selected by a menu item.

Return to the previous "filecard" in the "menu tree".

Branch off to an activity that requires a different display than the "filecard" display.

The first occurs when a user selects a menu item using either the arrow keys or a number key and then types RETURN. If this menu item selects another menu, then the new "filecard" and its menu must be displayed.

The second happens when the user types ESCAPE while a "filecard" is displayed. For a "filecard" with a level greater than 1, this will result in "moving back" to the previous "filecard" in the "menu tree" (the card directly underneath the current "filecard" on the display.) For the topmost "filecard" (level 1) the result of typing escape is up to the program to determine. For the sample program, it terminates the execution of the program.

The third happens when a user selects an item as in the first case, but here the item selects an activity and not another menu. In this case, the "filecard" display will more than likely be removed from the screen and a different display will replace it. When the activity is completed, the display should return to where it was left (the "filecard" where the user selected the activity.)

The display of the menu (not the "filecard") and the handling of the selection process is done via a call to the function GET_SELECTION. The function will return FALSE if the user typed ESCAPE. It will return TRUE if the user selected an item and typed RETURN and it will return the number of the item selected.

5.3.3.2 Going through the Menu Tree

When the program begins, it needs to put up the first display. This consists of the top and bottom portions of the screen and the first "filecard". In the program, the procedure FIRST_SCREEN does this. At all times, there is a variable CURRENT_CARD which has the number of the currently selected "filecard". FIRST_SCREEN sets this to 1, the number of the topmost (level 1) card. This variable controls the flow of the display during the selection process. Another variable, OLD_CARD is used to store the number of the last selected "filecard". This value is used to determine whether or not the menu requires updating when the "filecard" is redisplayed. This occurs either because of an error box or when the program branches off to an activity.

Once the initial screen has been displayed, the selection process begins. This is found in the REPEAT loop in the main body of the sample program. The general structure of the REPEAT loop is:

```
REPEAT
  IF OLD_CARD <> CURRENT_CARD THEN
    {redisplay menu in "filecard"}
    SELECTED := GET_SELECTION(MENU[CURRENT_CARD],
                SEL_NUM, TRUE)
  ELSE
    {don't redisplay menu}
    SELECTED := GET_SELECTION(MENU[CURRENT_CARD],
                SEL_NUM, FALSE);
  CASE CURRENT_CARD OF
    1: {case for each "filecard"}
    2: {case for each "filecard"}
    .
    .
    .
    9: {case for each "filecard"}
  END;
UNTIL FALSE;
```

As CURRENT_CARD has been set to 1 as the program enters this loop, GET_SELECTION will display the menu for the "filecard" on the screen and wait for the user's input. Each case in the CASE statement corresponds to the menu_number of the CURRENT_CARD. Thus once something has been selected, the CASE statement will process the selection given the CURRENT_CARD.

Each case (of CURRENT_CARD) reacts to the type of selection the user has made. The basic format for each case is:

```
BEGIN
  IF NOT(SELECTED) THEN BACK_UP
  ELSE
    BEGIN
      CASE SEL_NUM OF
        1: {case for each menu item}
        2: {case for each menu item}
        .
        .
        .
        9: {case for each menu item}
      END;
      GO_FORWARD;
    END;
END;
```

The case statement here corresponds to the menu items in the menu displayed. For menu items that select another menu ("menu selectors") the entry in the case simply sets CURRENT_CARD to the value of the menu now selected. For example, if menu item 3 selects "filecard" 4 then in the case statement the entry for 3 would be:

```
3: CURRENT_CARD := 4;
```

When a "menu selector" has been selected, a new "filecard" must be displayed. The CASE statement will set CURRENT_CARD to the value of the new "filecard" number. At the bottom of the CASE statement there is a call to a procedure called GO_FORWARD. This procedure will display the new "filecard" selected. The sample program includes this procedure. Its content is shown below:

```
PROCEDURE GO_FORWARD;

VAR PREVIOUS_CARD, OLD_ITEM: INTEGER;

  BEGIN
    IF OLD_CARD <> CURRENT_CARD THEN
      BEGIN
        PREVIOUS_CARD := CARD[CURRENT_CARD].P_CARD;
        MAKE_CARD(CARD[CURRENT_CARD],
          CARD[PREVIOUS_CARD]);
        OLD_ITEM := MENU[CURRENT_CARD].CURRENT_ITEM;
        MENU[CURRENT_CARD].LIST[OLD_ITEM].STATE :=
          NORMAL;
        MENU[CURRENT_CARD].LIST[1].STATE := INVERSE;
        MENU[CURRENT_CARD].CURRENT_ITEM := 1;
      END;
  END;
```

The procedure determines the previous card to the new current card and then calls MAKE_CARD to display the new "filecard". It also sets the new menu values for CURRENT_ITEM.

This procedure only works when the program is moving to a new "filecard". When an activity is selected, the program will return to the original screen, which does not require that a new "filecard" be displayed. However, the case will fall through this procedure call. The test at the beginning handles this event.

If the user types ESCAPE, it is necessary to go back to the previous card in the display. This requires that the current card be removed from the display. The sample program has a procedure, BACK_UP which handles this. For each case in the case of CURRENT_CARD (except the first level 1 card) there is a test for SELECTED. If it is FALSE, the user has typed ESCAPE, so BACK_UP. The content of this procedure is shown below:

```
PROCEDURE BACK_UP;

VAR PREVIOUS_CARD,
    ESCAPE_CARD:     INTEGER;

  BEGIN
    PREVIOUS_CARD := CARD[CURRENT_CARD].P_CARD;
    ESCAPE_CARD := CARD[PREVIOUS_CARD].P_CARD;
    REMOVE_CARD(CARD[CURRENT_CARC].MENU_LEVEL,
                CARD[PREVIOUS_CARD],
                CARD[ESCAPE_CARD]);
    CURRENT_CARD := PREVIOUS_CARD;
  END;
```

The procedure determines the previous card (to back up to...) and the escape card to update the top of the display. It then calls REMOVE_CARD to clean up the display. Finally it sets CURRENT_CARD to PREVIOUS_CARD, thus the current card to be displayed is the "previous" card in the display.

5.3.3.3 Branching Off to an Operation

When the user selects a menu item that is an "action selector", the program must now branch off to perform that action. This requires in most cases a new display on the screen, removing either the top, bottom, or "filecard" area, or all three from the screen. Removing an area requires:

1. Setting the viewport to that area of the screen that is to be cleared.

2. Saving the contents of that portion of the screen so that it can be restored.

3. Clearing that area with a Clear Viewport command.

This will now set up that area to be used for activity-specific display.

The sample program has a procedure called SET_UP_ACTIVITY, which sets the viewport to the entire screen, saves the screen contents, and then clears the screen.

5.3.3.4 Coming Back from an Operation

When an activity is complete the user is to return to the "filecard" display at the point at which it was left. This means that the screen should be restored back to its original contents prior to branching off to the activity.

When returning, first set the viewport back to that area which was removed. Then restore the contents back to the screen. This will put the display back to its original form. Since, CURRENT_CARD has not changed, GO_FORWARD will restore the original "filecard" and not a new one. Falling through the end of the CASE statement will then bring us back to GET_SELECTION which will display the original menu. Thus the selection process begins anew at the place left when the user selected a "action selector".

The sample program has a procedure called RETURN_FROM_ACTIVITY which sets the viewport to the entire screen and then restores the screen contents, thus returning back to the "filecard" display at the point it was left.

5.3.3.5 Performing an Activity

In the case (of CURRENT_CARD), for each

card, there is a CASE statement corresponding to each item in the menu. For those items that are "activity selectors", the case has the form:

```
     CASE SEL_NUM OF
     .
     n: BEGIN {branch off to an activity}
          SET_UP_ACTIVITY;
          DO_ACTIVITY;
          RETURN_FROM_ACTIVITY;
        END;
     .
     END;
```

If SET_UP_ACTIVITY and RETURN_FROM_ACTIVITY are properly done (see above) the code for the activity itself does not have to worry about maintaining the integrity of the "filecard" display itself.

5.3.3.6 Reporting an Error

If there is an error to report, the error box procedures supplied by the unit facilitate the reporting process. The sample program has a procedure which presents a simple error, DO_AN_ERROR.

```
PROCEDURE DO_AN_ERROR;

  BEGIN
    ERROR_BOX;
    PUT_ERROR_MSG;
    PAUSE;
    GO_AWAY_ERROR;
  END;
```

The basic format is to display an error box and then to display a message in the error box. PAUSE waits for the user to read the message and then to type something to exit. GO_AWAY_ERROR then cleans the error box from the screen.