DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES

Apple II Console and Keyboard Tools (8/85)

DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES
DEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTESDEVELOPERNOTES

7

Table of Contents

Foreword

About This Document

Console Driver

The Console Driver is a version of the Apple III Console Driver, adapted to the Apple IIe and IIc. The Console Driver supplies a simple and consistent interface to a set of display format and control procedures in a relatively small and fast package. Both display and control commands are sent to the driver in the same way, allowing developers to build a set of data structures that contain both display and control information. The Console Driver is described in Chapter 1.

Standard User Input Routine

Apple Computer has published several documents encouraging standard design, including how an Apple II input routine should look and behave. To help software developers to create programs that are consistent in terms of user interface, Apple is making available a Standard User Input Routine (UIR). It incorporates the standards adopted by Apple and is available for three environments:

- Apple II Pascal
- Applesoft BASIC
- Apple II Assembler

The User Input Routine is described in Chapter 2.

About the Disks

This package contains three disks, one for each of the languages. Each disk contains the Console Driver and the Standard User Input Routine.

Note: Because the Console Driver makes the User Input Routine more efficient, Apple recommends that the User Input Routine and the Console Driver be used together. The Assembler and BASIC versions of the UIR can be used without the Console Driver—special versions of the UIRs can be ordered from Apple Technical Support. Unlike the UIR-Console Driver combinations, these standalone UIRs work with both 4∅- and 8∅-column displays.

Each of the disks contains a demonstration program that runs when the disk is started up. It lets you specify several parameters, then runs the User Input Routine. Except for field width, which has no default value, you can just press RETURN instead of specifying your own values. While the demonstration program is running, you can press ESC to restart the demonstration.

Pascal

If your application program is written in Pascal (Pascal 1.3 or the 128K version of Pascal 1.2), use the Pascal version of the User Input Routine and Console Driver. The Pascal disk (volume name /PAS.CON) contains eleven files:

```
    SYSTEM.LIBRARY
    SYSTEM.MISCINFO
    SYSTEM.ATTACH
    SYSTEM.APPLE
    SYSTEM.PASCAL
    SYSTEM.STAR.LIB
    ATTACH.DRIVERS
    ATTACH.DATA
    INPUT.INFO.TEXT (text of Information Block)
    DEMO.TEXT
    SYSTEM.STARTUP (the startup demonstration program)
```

Assembly Language

The routines on these disks can be used in assembly-language application programs or called from BASIC programs. The disk, volume name /ASMCON, contains ten files, including both a relocatable and an absolute version.

```
PRODOS
BASIC.SYSTEM
CONUIR.REL (relocatable version of Console Driver/UIR)
CONUIR.OBJ (absolute version)
RELO.DOC.TEXT (tells how to use RELOCATOR)
RELO.OBJ
RELOCATOR (creates CONUIR.OBJ from CONUIR.REL at address you choose)
ASSEM.INTER
ASSEM.INTER.Ø
STARTUP (demonstration program)
```

To use RELOCATOR, run RELOCATOR. Then, in response to prompts, answer

- CONUIR.REL
- $address
- CONUIR.OBJ

BASIC

The BASIC version of the UIR is relocatable. The disk, volume name /BASCON, contains eight files:

```
PRODOS
BASIC.SYSTEM
CONUIR.REL (User Input Routine plus Console Driver)
CONDAMP.REL (the ampersand package)
RBOOT
RLOAD
RELEASE
STARTUP (the demonstration program)
```

Hardware Requirements

To use this product, you need either

- an Apple IIc, or
- an Apple IIe with an 8Ø-column card.

Chapter 1

The Console Driver

Overview

The Console Driver described here is a version of the Apple III Console Driver, adapted to the Apple IIe and IIc. The Console Driver can serve as a low level tool for the implementation of different styles of human interface. Once the Console Driver is used, all subsequent screen output should come from the Console Driver.

Unlike with the typical programming interface, you don't have to make a sequence of calls to set up for text to be displayed--it can be done with one call to the Console Driver. This simplifies the programming of the human interface. Information used to format the text can be imbedded in the text itself.

The driver supports a form of screen structure known as a viewport, a rectangular portion of the screen where all console functions take place. Once a viewport is established, any future text display is within the viewport. All text outside the viewport is protected.

The Screen

The screen consists of

- 8Ø columns of text, numbered (left to right) Ø to 79

- 24 lines of text, numbered (top to bottom) Ø to 23

The upper left corner is column Ø, line Ø (abbreviated Ø,Ø).

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 Console Driver maintains an invisible cursor, which represents the current location at which a displayable character will be placed. The position of this cursor is specified by the two variables CH and CV, described later in this chapter. The default is Ø,Ø (upper-left corner).

When the Console Driver is first used, the viewport defaults to the whole screen. You can set the viewport by a special control and four parameter bytes which specify the upper-left and lower-right corners of the viewport. All console functions then take place within the new viewport.

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

Viewport Specification

Six variables specify the top, bottom, left, and right edge of the viewport, as well as its width (in columns) and its length (in lines). The default viewport is the entire screen. The variables, together with their default values, are

Variable	Definition	Default
WNDTOP	top line	Ø
WNDBOT	bottom line	23
WNDLFT	left column	Ø
WNDRGT	right column	79
WNDWTH	width in columns	8Ø
WNDLEN	length in lines	24

The Cursor

This section describes how the cursor's position and movement are specified.

Cursor Position

The current cursor position is maintained in two variables:

- CH (current horizontal position)

- CV (current vertical position)

When the Console Driver is first used, both values are set to zero, signifying the upper-left corner of the screen.

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

Cursor Movement

Five flags direct the Console Driver how to move the cursor within the viewport. In all five cases, the default value is 1 (TRUE). If set to zero, they are FALSE.

CONLFD (Line Feed)

When CONLFD is true, the Console Driver automatically performs a line feed (control code 1Ø decimal, $ØA hex) after every carriage return (13 decimal or $ØD hex). When it is false, no automatic line feed is performed. You can force a line feed by sending a line feed character.

CONADV (Advance)

When CONADV is true, the cursor advances one space to the right after each display character is placed on the screen. When it is false, the cursor does not advance after each character, but remains in the same position. In this case, you must explicitly move the cursor by sending a Move Cursor Right control (Ø9 decimal or $Ø9 hex).

CONWRAP (Wrap)

When CONWRAP is true, an attempt to move the cursor beyond the right (or left) edge of the viewport causes the cursor to be placed at the opposite edge of the next (or previous) line of the viewport. When it is false, the cursor remains at the edge of the viewport on the current line. To move the cursor to the next line, send a Move Cursor Down (1Ø decimal, $ØA hex). To move the cursor to the previous line, send a Move Cursor Up (11 decimal, $ØB hex). Follow these by a Return Cursor (13 decimal, $ØD hex) to move the cursor to the beginning of a line.

CONSCRL (Scroll)

When CONSCRL is true, an attempt to move the cursor beyond the top or bottom line of the viewport causes the contents of the viewport to be scrolled either down or up. The cursor moves to the beginning of the new top or bottom line. If it is false, the cursor remains at the top or bottom of the viewport.

DLEFLAG (Space Expansion)

When DLEFLAG is true, DLEs (16 decimal, $1Ø hex) are interpreted as space expansion controls with a following parameter byte. (See Screen Control Codes, later in this chapter.) If it is false, they are ignored. This is used to support Apple II Pascal text files. For other uses, set this flag to Ø (false).

Text Modes

You can determine the fill character, whether MouseText is used, and whether text is displayed in normal or inverse mode.

Fill Character

The fill character is the character used to clear the contents of the viewport. The default value is a space (32 decimal, $2Ø hex). Its value is in the status block variable CONFILL. Due to the Apple II character mapping, the actual binary value of the fill character is

- $ØAØ hex (16Ø decimal) for a normal space character, or

- $2Ø hex (32 decimal) for an inverse space character.

MouseText

The MOUSE flag specifies whether the Console Driver displays MouseText characters. The default is FALSE. If MOUSE is true, characters in the range $4Ø to $5F (64 to 95 decimal) are mapped into the MouseText character set. Control codes are processed as is.

Normal and Inverse

The CONVID flag determines whether text is displayed in normal or inverse mode. CONVID is set via two control codes (Set Normal Text and Set Inverse Text), described later in this chapter. If CONVID is $8Ø (128 decimal), text is normal. If CONVID is Ø, text is inverse. The default value is NORMAL.

Screen Control Codes

This section summarizes the 29 screen control codes. These control codes are numbered $ØØ through $1F (ØØ through 31 decimal), except that control codes $Ø5, $Ø6, and $Ø9 are undefined and, if used, are ignored. The table on the next page lists them in numerical order. The detailed descriptions that follow the table are in functional sequence, rather than numerical order.

hex	decimal	Control Code
$ØØ	ØØ	no operation
$Ø1	Ø1	save and reset viewport
$Ø2	Ø2	set viewport
$Ø3	Ø3	clear from beginning of line
$Ø4	Ø4	restore viewport
$Ø7	Ø7	sound the bell
$Ø8	Ø8	move cursor left
$ØA	1Ø	move cursor down (line feed)
$ØB	11	clear to end of viewport
$ØC	12	clear viewport
$ØD	13	return cursor (carriage return)
$ØE	14	normal text
$ØF	15	inverse text
$1Ø	16	space expansion
$11	17	horizontal shift
$12	18	vertical position
$13	19	clear from beginning of viewport
$14	2Ø	horizontal position
$15	21	cursor movement
$16	22	scroll down
$17	23	scroll up
$18	24	MouseText off
$19	25	home cursor
$1A	26	clear line
$1B	27	MouseText on
$1C	28	move cursor right
$1D	29	clear to end of line
$1E	3Ø	absolute position
$1F	31	move cursor up

No Operation

control code: $ØØ (decimal ØØ)

This control code has no effect and is ignored.

Set Viewport

control code: $Ø2 (decimal Ø2)

Sets the boundaries of the viewport. It requires all four of its parameter bytes (if any is missing, the control code is ignored). The four parameters specify the absolute coordinates for the upper-left and lower-right corners of the viewport, and must appear in this order:

1. upper-left corner X (or column) value

2. upper-left corner Y (or line) value
3. lower-right corner X (or column) value
4. lower-right corner Y (or line) value

This control does not affect cursor movement, normal/inverse text mode, nor the MouseText setting. It does not save the current viewport (see Save and Reset Viewport). The cursor is placed in the upper-left corner of the new viewport.

Validity Checking

The parameters are checked for validity before the viewport values are set. The rules are

- Any parameter byte greater than 127 is negative (because bit 7 is set), causing this command to be ignored.
- If the X coordinate of the upper-left or lower-right corner is greater than 79, it is set to 79.
- If the Y coordinate of the upper-left or lower-right corner is greater than 23, it is set to 23.
- The X-coordinate of the upper-left corner is used for WNDLFT.
- The Y-coordinate of the upper-left corner is used for WNDTOP.
- The X-coordinate of the lower-right corner, if greater than WNDLFT, is used for WNDRGT, else this command is ignored.
- The Y-coordinate of the lower-right corner, if greater than WNDTOP, is used for WNDBOT, else this command is ignored.
- If for any reason the command is ignored, it does not change the current viewport settings.

Save and Reset Viewport

control code: $Ø1 (decimal Ø1)

Saves the current settings of the viewport: its coordinates, cursor position, cursor motion controls, mousetext, and normal/inverse setting. The viewport is then set to the default values of the full screen. Only one level of save is allowed--saving a second viewport erases any information for a previously-saved viewport.

Restore Viewport

control code: $Ø4 (decimal Ø4)

Restores the viewport to the values of the most recently saved viewport. If no viewport has been saved, the values are set to the default values for the whole screen. (See Save and Reset Viewport.)

Clear Viewport

control code: $ØC (decimal 12)

Moves the cursor to the upper-left corner of the viewport and then clears the viewport by setting the contents to the current fill character.

Clear from Beginning of Viewport

control code: $13 (decimal 19)

Clears the viewport from position Ø, Ø through the cursor. The cursor does not move.

Clear to End of Viewport

control code: $ØB (decimal 11)

Clears the contents of the viewport, from the current cursor position to the end of the cursor line, and all lines below the cursor. The cursor does not move.

Clear Line

control code: $1A (decimal 26)

Moves the cursor to the beginning of the current line and clears the entire line.

Clear from Beginning of Line

control code: $Ø3 (decimal Ø3)

Clears the current line, from the beginning of the line through the current cursor position in that line.

Clear to End of Line

control code: $1D (decimal 29)

Clears the current line, starting from and including the current cursor position in the line. The cursor does not move.

Cursor Movement

control code: $15 (decimal 21)

This control code and its parameter set the cursor movement controls as specified by the parameter. The parameter is a single-byte value, with only the lower five bits significant. The upper three bits are to be set to zero. A zero resets the control; a one sets it. If the parameter does not exist, or the upper three bits are non-zero, the command is ignored. (See also The Cursor, earlier in this chapter.)

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

Home Cursor

control code: $19 (decimal 25)

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.

Move Cursor Left

control code: $Ø8 (decimal Ø8)

Moves the cursor left one position. Wrapping around and scrolling are determined by the cursor controls. (See Cursor Controls, earlier in this chapter.)

Move Cursor Right

control code: $1C (decimal 28)

Moves the cursor right one position. Wrapping and scrolling are controlled by the cursor controls.

Move Cursor Up

control code: $1F (decimal 31)

Moves the cursor up one line. Scrolling is controlled by the cursor controls.

Move Cursor Down (Line Feed)

control code: $ØA (decimal 1Ø)

Moves the cursor down one line. Scrolling is performed by the cursor controls. (See Cursor Control, earlier in this chapter.)

Return Cursor (Carriage Return)

control code: $ØD (decimal 13)

Moves the cursor to the beginning of the current line (the left edge of the viewport). A line feed may also be issued automatically, depending on the setting of the cursor controls. Scrolling may also take place. (See Cursor Control, earlier in this chapter.)

Scroll Up

control code: $17 (decimal 23)

Causes the contents of the viewport to scroll up, leaving a blank line at the bottom of the viewport. The cursor does not move.

Scroll Down

control code: $16 (decimal 22)

Causes the contents of the viewport to scroll down, leaving a blank line at the top of the viewport. The cursor does not move.

Horizontal Position

control code: $14 (decimal 2Ø)

Moves the cursor horizontally to the relative column number passed in a single-byte parameter (Ø to 79). A parameter of 1Ø means to move to the tenth column in the viewport, not to column 1Ø of the whole screen. A parameter of Ø moves the cursor to the leftmost column. To determine the correct relative column, add the parameter to the value of WNDLFT (see Viewport Specifications). If the sum is greater than 127 (negative), the cursor moves to the left column.

If the parameter is missing, this control is ignored. This control has no effect on the vertical position of the cursor.

Vertical Position

control code: $12 (decimal 18)

Moves the cursor vertically to the relative line number passed in a single-byte parameter (Ø through 23). A parameter of 1Ø means to move to the tenth line in the viewport, not to line 1Ø of the whole screen. A parameter of Ø moves the cursor to the top line.

To determine the correct relative line, add the parameter to the value of WNDTOP (see Viewport Specifications, earlier in this chapter). If the resulting value is greater than the value of WNDBOT (the bottom line of the viewport), the cursor is placed in the bottom line of the viewport. If the parameter is missing, this control is ignored. This control has no effect on the horizontal position of the cursor.

Absolute Position

control code: $1E (decimal 3Ø)

This control code combines the actions of the Horizontal Position and Vertical Position control codes. 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.

Normal Text

control code: $ØE (decimal 14)

Causes all subsequent characters to be displayed as light characters on a dark background. It does not affect any characters already on the screen. This control code sets the CONVID flag to $8Ø (128 decimal).

Inverse Text

control code: $ØF (decimal 15)

Causes all subsequent characters to be displayed as dark characters on a light background. It does not affect any characters already on the screen. This control code sets the CONVID flag to Ø. See also Normal Text.

MouseText On

control code: $1B (decimal 27)

Turns on the display of MouseText characters. All displayable characters in the range $4Ø through $5F (64 through 95 decimal) are mapped into the MouseText characters for display.

MouseText Off

control code: $18 (decimal 24)

Turns off the display of MouseText characters.

Horizontal Shift

control code: $11 (decimal 17)

Causes 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 Ø, the control has 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 are 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 are 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, the viewport is cleared.

The shifted characters are moved directly to their destination. 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.

Space Expansion

control code: $1Ø (decimal 16)

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, the Console Driver ignores this control. DLE expansion can be turned off using the mode value of 4 or 12 in the UNITWRITE call to the driver (see Pascal Interface, later in this chapter). It can also be turned on or off with the Cursor Control. The default is ON.

Sound the Bell

control code: $Ø7 (decimal Ø7)

This control code, when used once, sounds the ProDOS-recommended beep. It has no effect on the screen. Repeated control codes produce a longer sound.

Displayable Characters

The Console Driver displays the Apple II's Alternate character set. It assumes however, that all characters passed to it are in the standard ASCII character set (range $ØØ to $7F, Ø to 127 decimal). These characters are mapped into the appropriate character set (normal or inverse, MouseText) for display purposes.

Characters passed to the Console Driver in the range $8Ø to $FF (128 to 255 decimal) are a special case. The characters are displayed after the seventh bit is reset, resulting in this mapping:

$8Ø - $9F (decimal 128 - 159)	mapped to inverse uppercase letters
$AØ - $BF (decimal 16Ø - 191)	mapped to inverse special characters
$CØ - $DF (decimal 192 - 223)	mapped to MouseText characters
$EØ - $FF (decimal 224 - 255)	mapped to inverse lowercase letters

This is independent of the settings for normal/inverse and MouseText in the driver. Reference manuals for specific computers contain details on the character sets.

Characters in the range $ØØ to $1F (Ø to 31 decimal) are defined as control codes that invoke the operations listed earlier in this chapter.

Characters in the range $2Ø to $7F (32 to 127 decimal) are defined as displayable characters and are displayed according to the settings of the Console Driver.

MouseText

To use MouseText, you must send the MouseText-on control code to the Console Driver. Characters in the range $4Ø to $5F (64 to 95 decimal) are then mapped into the appropriate MouseText character. For example, to display the file-folder icon instead of the letters X and Y:

27	MouseText-on control code
"X"	first part of picture of file folder
"Y"	second part of file folder
24	MouseText-off control code

At the end of a sequence of MouseText characters, be sure to turn off MouseText with the MouseText-off control code. Any characters not in the MouseText range will be displayed according to the settings of the Console Driver.

Language Interfaces

The Console Driver can be used with Apple II Pascal, Applesoft BASIC, and 65Ø2 Assembler Language.

Pascal

The version of the Console Driver that is used with Pascal accepts five calls, each described in this section. This section also describes the Pascal data interface and how the Console Driver is called.

Data Interface

Both control codes and text to be displayed are passed to the Console Driver as a contiguous array of data. For example, to print "Hello" on line 1Ø, column 15, in inverse, then to home the cursor and to return to normal text, you would create the following array of data (all numbers are decimal):

3Ø	absolute position
15	parameter (column 15)
1Ø	parameter (line 1Ø)
15	inverse text
72	"H"
1Ø1	"e"
1Ø8	"l"
1Ø8	"l"
111	"o"
25	home cursor
14	normal text

This array is not a string in the Pascal sense of the word: the first byte is data rather than the length of the array (as in a string). The Console Driver can accept an array of up to 32,767 bytes (Pascal's limit on integers).

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

Calling the Console Driver

The Console Driver is a Pascal Attach driver. Its unit number is #13Ø. For information on Pascal Attach drivers, see the Apple II Pascal 1.2 Device and Interrupt Support Tools manual.

To transfer data to the Console Driver to be displayed, use a UNITWRITE call from a Pascal program. UNITWRITE's format is

```
    UNITWRITE(13Ø, ARRAY_ADDR, LENGTH_ARRAY, MODE)
```

where

13Ø is the Console Driver's unit number

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). MODE can have four values:

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

When passing a string to the driver, always reference the string as

```
    STRING_VAR[1]
```

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

Status Calls

The Console Driver accepts only one status call. It returns a data structure that describes the Driver's status. This call instructs the Console Driver to copy its values into this record, where you can inspect it. The variables are described earlier in this chapter.

Here is the form of the UNITSTATUS call:

```
    UNITSTATUS(13Ø, CON_STAT_BLK, Ø)
```

where

13Ø is the unit number of the driver

CON_STAT_BLK is a record with the format:

```
    TYPE BYTE = Ø..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;
```

Control Calls

The driver accepts four control calls, which allow you to

- get the current location of the cursor and the text character at the current cursor location
- save and restore the contents of the current viewport.

You must supply the buffer in which this data is stored, as it is not in the Console Driver. It is recommended that you allocate some space on the heap for this buffer, allowing this space to be reclaimed as needed. If your program does not require this function, this space can be saved. To calculate the amount of space required for a viewport, multiply its width (WNDWTH) by its length (WNDLEN).

Getting the Current Cursor Position

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

```
    UNITSTATUS(13∅, LOCATION, 2);
```

where LOCATION is a record of the form

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

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

Getting the Current Text Screen Character

Make a UNITSTATUS call of the form

```
    UNITSTATUS(13∅, CHARACTER, 8194);
```

where CHARACTER is a byte (∅..255) variable. The driver will return the current binary value of the character found at the current cursor location. You must map this value in the proper ASCII interpretation.

Saving the Viewport

To save the contents of the viewport, make a UNITSTATUS call of the form

```
    UNITSTATUS(13∅, VWPORT_BUF, 16386);
```

where

> 130 is the Console Driver's unit number
>
> VWPORT_BUF is a buffer to hold the contents of the viewport.

Restoring the Viewport

To restore the contents of the viewport, make a UNITSTATUS call of the form

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

where

> 130 is the Console Driver's unit number
>
> VWPORT_BUF is a buffer to hold the contents of the screen.

You must keep track of which viewport has been saved in which buffer. Before restoring a viewport, you must set the required viewport before making the restore call.

BASIC

The version of the Console Driver that is used with BASIC programs supports twelve functions, all of them ampersand (&) routines. Each is described below.

- 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

Console Driver Functions

Calling the Console Driver

Calls to the Console Driver are made with the "ampersand hook." BASIC statements used to call the Console Driver have the form

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

Specific formats for the calls are described below.

Output Data to the Console

There are two Console Driver calls to output data to the display. The first has the form

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

where S$ is a string. This call outputs the contents of S$ to the display. S$ can include both control codes and ASCII characters.

The second has the form

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

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

This call outputs the contents of the string array SA$, beginning with the string selected by the index I1% and ending with the string indexed by I2%. These strings can contain both control codes and ASCII characters.

Save the Current Viewport Contents

Before saving the contents of the viewport, first allocate a buffer via a call to the special function "Get memory," which has the form

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

P% is an integer that specifies the number of pages (256 bytes) of memory to allocate A% is the address of that memory. Here is a formula for calculating the number of pages required:

```
    (WNDWTH * WNDLEN) / 256
```

rounding up to the nearest integer.

For example, to store the contents of the whole screen requires an allocation of eight pages. If not enough pages are available, BASIC's OUT OF MEMORY error will occur.

Once the &GTMEM call has been made, you can make the call to save the contents of the viewport. It has the form

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

where A% is the address returned from the &GTMEM call.

Restore the Current Viewport Contents

To restore the viewport contents, make a call with the form

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

where A% is the address used in the &SVVP call. This restores the previously saved contents to the viewport. Be sure that the contents you restore are the same size as the current viewport.

Get the Status of the Console Driver

To get the status of the console driver, make this call:

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

where CI% is a 16-element array, for example

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

This returns the contents of the status block to the array CI%. 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%(1Ø) = CONADV
    CI%(11) = CONLFD
```

```
CI%(12) = CONSCRL
CI%(13) = CONVID
CI%(14) = DLEFLAG
CI%(15) = CONFILL
CI%(16) = MOUSE
```

Get the Current Cursor Position

To get the current absolute coordinates of the cursor, make this call:

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

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

Get the Current Text Screen Character

To get the binary value of the text character at the current cursor position, make this call:

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

where C% is the character returned.

Initialize the Console Driver

To initialize the Console Driver to its default environment, make this call:

```
&INITCD
```

Release Console Driver

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

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

where C% is 8Ø.

Get Console Driver Version and Copyright

To get the version number of the Console Driver, make the call:

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

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

To get the Console Driver's copyright notice, make the call:

```
    &CDCPYRT(CM$)
```

where CM$ is the copyright notice returned.

Setting the Console Driver Address

Before the ampersand package can use the Console Driver, it must have the location of the Console Driver. Do this with the call:

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

where A% is the starting address (also of the entry-point) of the Console Driver. This call must be made before any other calls to the ampersand package.

Loading Ampersand Package and Console Driver

This BASIC routine loads the ampersand package and Console Driver:

```
    1Ø PRINT CHR$(4);"brun release":  REM release memory buffers
    2Ø PRINT CHR$(4);"pr#3"
    3Ø REM load & initialize Console Driver & UIR
    4Ø A1 = Ø :  A2 = Ø
    5Ø PRINT CHR$(4);"brun rboot"
    6Ø A1 = USR(Ø),"conuir.rel":  REM load Console Driver & UIR
    7Ø A2 = USR(Ø),"condamp.rel":  REM load ampersand interface
    8Ø CALL A2
    9Ø &STCDADR(A1)
```

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 Console Driver. This example uses the Console Driver (after it and the ampersand package have been loaded) to place the string "Hello there" on the screen:

```
    1Ø DIM AB$(3)
    2Ø DIM ST$(11)
    3Ø AB$(1) = CHR$(3Ø):  REM ABSOLUTE POSITION
    4Ø AB$(2) = CHR$(1Ø):  REM X COORDINATE
    5Ø AB$(3) = CHR$(15):  REM Y COORDINATE
    6Ø ST$ = "Hello there"
    7Ø &WRTSTR(AB$)
    8Ø &WRTSTR(ST$)
```

Relocating the Console Driver

The Console Driver is a REL (relocatable) file produced by the EDASM Editor/Assembler. It must be relocated in memory before it can be used. Follow the instructions in either the ProDOS Assembler Tools manual or 65Ø2 Assembler/DOS Tool Kit manual, and use RBOOT and RLOAD to perform the relocation.

Assembly Language

The version of the Console Driver that is used with assembly language programs supports the following seven 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

Console Driver Functions

The Console Driver has a single entry point. Calling the driver is done in much the same way as ProDOS MLI calls. See the ProDOS Technical Reference Manual for details.

Calling the Console Driver

The driver has only one entry point, located at the beginning. Once the driver has been relocated, 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
```

where the label PCONSOLE is the starting address of the driver. You determine this when deciding where to relocate the driver in memory. In the calling program, there should be a statement 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 varies according to the Console Driver function. The specific formats are described below.

The driver returns to the caller with the carry flag clear if no error occurred, 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 which error occurred.

Output Data to the Console

This call outputs 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     3Ø       ;absolute position
             DFB     1Ø       ;x position
             DFB     15       ;y position
             ASC     "Hello there!!"

    LENGTH1 EQU      16       ;length of DATA1
```

calling format:

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

parameter list format:

```
    OUTPUTDATA      DW      DATA1
                    DW      LENGTH1
```

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

Save the Current Viewport Contents

This call saves 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, the buffer is large enough to hold the contents of the entire screen (8Ø columns by 24 lines).

calling format:

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

parameter list format:

```
    BUFFERSIZE       EQU      192Ø     ;full screen

    SAVEBUFFER       DS       BUFFERSIZE
```

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

Restore the Current Viewport Contents

This call restores the contents of the current viewport from the buffer pointed to in the call--in this case SAVEBUFFER. Be sure 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 to it. This lets you move a viewport and its contents around the screen.

calling format:

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

parameter list format:

```
    SAVEBUFFER       DS       BUFFERSIZE
```

This call returns no errors. The A-register is Ø and the carry flag is cleared.

Get the Status of the Console Driver

This call returns the current status of the Console Driver in the status block pointed to in the call--in this case STATUSBLK. Be sure the status block used matches this description exactly--data may be destroyed if the status block is smaller than the one described.

calling format:

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

parameter list format:

```
    STATUSBLK        EQU      *

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

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

Get the Current Cursor Position

This call returns 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 earlier).

calling format:

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

parameter list format:

```
    CURSORPOS        EQU      *

    XPOS      DFB     Ø
    YPOS      DFB     Ø
```

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

Get the Current Text Screen Character

This call returns the binary value of the text character located at the current cursor position. This value reflects whether the character is inverse, normal, or MouseText. The calling program must decipher the value.

calling format:

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

parameter list format:

```
    TEXTCHAR        DFB     Ø
```

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

Initialize the Console Driver

This call sets the Console Driver back to its default state. No parameter list is required.

calling format:

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

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

Using the Console Driver With Your Program

This section describes how the Console Driver uses soft switches and the Zero Page, and introduces the need for relocating the Console Driver.

Console Driver Zero Page Use

The console driver uses zero page locations $2Ø to $4Ø. The contents of these locations are saved when the driver is called, and restored upon exit.

Console Driver Soft Switch Use

The console driver uses soft switches to control its use of the display memory:

8ØCOL ($CØØD)	- turn on 8Ø-column card
8ØSTORE ($CØØ1)	- use auxiliary memory for display
PAGE2 ($CØ55, $CØ54)	- to switch between even and odd locations on the 8Ø-column card
ALTCHARSET ($CØØF)	- to use alternate character set

When the Console Driver is called, these switches are set to their appropriate values. Since the Console Driver is intended to be the sole manager of the console display, these switches are not reset when the driver returns to the calling program. It is up to the program to reset to the normal environment.

Relocating the Console Driver

The Console Driver is a REL (relocatable) file produced by the EDASM Editor/Assembler. It must be relocated in memory before it can be used. Follow the instructions in either the ProDOS Assembler Tools manual or 65Ø2 Assembler/DOS Tool Kit manual, and use RBOOT and RLOAD to perform the relocation.

Chapter 2

Standard User Input Routine

Overview

Why Standardization is Needed

Most application programs at some point ask the user to enter data at the keyboard. Unfortunately, there has been little standardization in the way that programs interface with the user. Pascal and BASIC have such different conventions that a user has to completely relearn how to interact with one language after using the other. Many application programs use the input conventions built into the language being used, while others use more sophisticated and user-friendly ones. The user of several application programs may have just as many different interfaces to contend with.

Overview of the User Input Routine

The User Input Routine (UIR) described here follows the standards published in Apple II Human Interface Guidelines and is a superset of the standards used in the popular AppleWorks program.

The UIR displays a field on the screen. This field consists of the default string, followed by a series of fill characters. To the right of the default string, a cursor is visible. Initially, this is the Insert Cursor described in the Guidelines. Pressing CONTROL-E toggles between the Insert Cursor and the Replace (or overstrike) Cursor.

When the Insert Cursor is present, typing any printing character inserts that character into the field at the current cursor position. All characters in the field to the right of the cursor are shifted right.

When the Replace Cursor is present, typing any printing character places that character in the field replacing the character under the cursor.

The user can edit the field by adding or replacing characters or by using the editing commands described below. When satisfied with the string in the field, the user presses the RETURN key. This terminates the UIR and returns control to the application program. The user's response will be in the string variable specified when the UIR was called (replacing the default).

If the application program specifies a string variable that can contain more characters than the width of the field, the UIR retains characters that are hidden beyond the right edge of the field. These characters return to view if characters in the field are deleted.

The UIR supports these editing commands:

- The left- and right-arrow keys move the cursor left and right within the field.

- DELETE and CONTROL-D both delete the character to the left of the cursor. The characters to the right of the cursor are shifted left.

- CONTROL-F deletes the character under the cursor (Forward Delete).

- CONTROL-E toggles between the Insert and Replace cursors.

- CONTROL-X deletes all characters in the field.

- CONTROL-Y deletes all characters from the cursor position to the end of the field (including those characters saved by insert).

- CONTROL-Z restores the default string.

Customization and Advanced Uses

In general, the UIR behaves as described in the Overview. It can, however, be customized to the needs of a particular application program. A structure called the Information Block lets the application program tell the UIR how to react to the user's keystrokes, and lets the UIR tell the application program about its status.

If a viewport has been defined, the UIR respects it, with one restriction: the last two positions in the viewport can not be included in the input field. A field as large as 254 characters can be specified.

Terminating the UIR

Normally, when the RETURN key or the ESCAPE key is pressed, the UIR terminates with the input string set to the characters currently in the field on the screen (fill characters excepted).

Other characters can be used to terminate (or interrupt) the UIR. Up to 2Ø characters can be specified as termination characters. For each termination character, the application program can specify whether the Open Apple or Solid Apple key must be pressed with the character.

For each termination character, the application program can also specify whether to completely terminate the UIR or just to interrupt it temporarily. After the UIR is interrupted, then called again, it remains as it was when it was interrupted (unless the application program has changed parameters in the Information Block). One way to use this feature is to let a help character (perhaps Open Apple-?) interrupt the UIR during editing to display a help message.

Immediate Mode

Immediate mode, an advanced use of the UIR, allows the application program to constantly monitor the input process. This feature can be used by the application program to update a clock display, provide animated sequences, or run in demonstration mode.

Information Block

The Information Block is divided into three logical sections:

- General Information,
- Termination Information, and
- Internal Information.

Format of the Information Block

```
max_terms      equ    2Ø      ;Maximum number of terminators

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


;                              Termination Information
;                              -----------------------
exit_type      db     Ø       ;Indicates which termination condition
                              ;  occurred
                              ;Ø-not terminated yet
                              ;not Ø-index into terminating char list
last_event     db     Ø       ;last event type (not used)
last_ch        db     Ø       ;character user keyed in
last_mod       db     Ø       ;keypress modifier
n_chars        db     Ø       ;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"
                                   ;Ø-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"
                                               ;Ø-terminate input
                                               ;1-interrupt input

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

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

Information Block Default Values

The default values of the Information Block are:

```
        width=254;
        fill_char=' ';
        mouse_fill=Ø;
        cursor=Ø;
        control=Ø;
        beep=1;
        immediate=Ø;
        entry_type=Ø;
        exit_type=Ø;
        bord_ch=' ';
        last_event=Ø;
        last_ch=Ø;
        last_mod=Ø;
        n_chars=2;
        char_list[1]=chr(13);     {RETURN}
        char_list[2]=chr(27);     {ESCAPE}
        mod_list[1]=Ø;
        mod_list[2]=Ø;
        exit_list[1]=Ø;
        exit_list[2]=Ø;
        origin_x= ,
        origin_y= ,         Curr relative cursor coordinate in viewport
        cursor_x= ,         defined by Console Driver
        cursor_y= ,
        cursor_pos=Ø;
        input_length=Ø;
        slow_blink= ,       Values necessary to blink cursor
        fast_blink= ,       8Ø times per minute
```

General Information Section

width

This parameter tells the UIR how wide to make the field on the screen. When the UIR is called, it displays the input string's default value at the cursor position. If there is any room left in the field, fill characters are displayed (the number of fill characters equals width minus length of input string). The parameter named fill_char contains the fill character. 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 viewport minus two, the UIR reduces width accordingly.

fill_char

This is the fill character that is used in the field. It is initially the blank character. Mouse_fill (see below) overrides fill_char.

mouse_fill

If mouse_fill is 1, the MouseText ghost underline is used as the fill character. If mouse_fill is Ø, the character in fill_char is used as described above. Mouse_fill is initially Ø.

Before using this option, the application program must determine whether MouseText is available in ROM. If memory location $FBB3 contains $Ø6 <u>and</u> memory location $FBCØ does not contain $EA, then MouseText is available.

cursor

This represents the cursor being used. It is Ø when the insert cursor is in use, and 1 when the replace cursor is in use. CONTROL-E toggles between the two. The initial value is normally Ø, but the application program can force the UIR to start with the replace cursor by setting this parameter to 1 before calling the routine.

control

This parameter is initially Ø, meaning that control characters are not allowed as input (typing a control character causes a beep).

If this parameter is set to 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 Open Apple key, the CONTROL key, and one other key. This lets the user type, for example, CONTROL-X without deleting the input field.

The actual value inserted in the string is the ASCII value + 128, which appears on the screen as the inverse of the corresponding character. For example, to insert the carriage return character (ASCII 13), the user presses Open Apple, CONTROL, and M (ASCII 77). The screen shows an inverse M, and the string will contain the value 2Ø5 (77+128). To extract the control character, subtract 192.

ASCII Code	CONTROL Character	Value in String	Value on Screen
13	CONTROL-M	2Ø5	inverse M

Note that editing characters and termination characters are not affected by the setting of control.

beep

If this parameter is 1 (the initial value), any illegal keypresses cause the UIR to beep. If it is Ø, there is no beep.

immediate

If this parameter is 1, the UIR returns to the application program after each keypress check. When the application program next calls the UIR, it will be considered an "immediate" re-entry.

If this parameter is Ø (the initial value), the UIR returns to the application program only after a termination character is pressed.

During "immediate" processing, the application program can tell whether a key has been pressed, by checking the last_ch parameter. If it is not Ø, a key has been pressed and last_ch contains the ASCII value of that keypress (its corresponding keypress modifier is in last_mod). When the UIR is re-entered, it checks last_ch and last_mod. If there is a keystroke, the UIR processes it; otherwise it looks for the next keystroke. The application program can therefore "process" the keystroke before the UIR does. At this point, the application program can leave the keystroke intact and re-enter the UIR, which will also "process" the keystroke. Alternatively, the application program can set last_ch and last_mod to Ø, which causes the UIR to ignore the keystroke.

Application programs that use immediate mode must keep the cursor blinking at the correct rate. See the description of slow_blink and fast_blink.

entry_type

Tells the UIR what type of entry is being made. If entry_type is Ø, it 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 program. This parameter is managed by the UIR and normally does not need to be changed by the application program.

bord_ch

Normally, the cursor blinks by alternating between the cursor character and the space character. Sometimes, for example when the field is filled and the cursor resides one character beyond the field, bord_ch (border character) is used instead of space.

Termination Information Section

exit_type

When the UIR terminates, this parameter contains the number (1 through 2Ø) of the termination character that caused the termination. If exit_type is Ø, this indicates that the UIR has not terminated yet (that is, immediate mode is in effect).

last_event

Not currently used.

last_ch

Contains the ASCII value of the last keypress, if the last keypress check sensed a keystroke. It contains Ø if no keypress was sensed. It is useful for application programs using the UIR's immediate mode.

last_mod

Contains the keystroke modifier if a keystroke was sensed by the last keypress check. Otherwise it is Ø. Possible values are:

Ø - no modifier was pressed
1 - Open Apple key was pressed together with another key
2 - Solid Apple key was pressed together with another key
3 - both Apple keys were pressed together with another key

n_chars

The number of termination characters that have been configured. This is initially 2 (for RETURN and ESCAPE).

char_list

This is a 2Ø-byte table containing the ASCII values of the configured termination characters. For the alphabetic characters A through Z, only the uppercase ASCII values need be in the table.

The UIR looks only at the first n_chars bytes. The first two bytes in this list are initially 13 and 27, the ASCII codes for RETURN and ESCAPE.

mod_list

This is a 2Ø-byte table that specifies what keystroke modifiers are needed for each termination character to be recognized.

Ø	- no modifiers can be pressed
1	- Open Apple must be pressed together with termination character
2	- Solid Apple key must be pressed
3	- either the Open Apple or Solid Apple key must be pressed

term_list

This is a 2Ø-byte table that specifies the termination type of each termination character. A value of Ø 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.

Internal Information Section

origin_x and origin_y

These contain the relative coordinates of the start of the field within the current viewport. When the UIR 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.

cursor_x and cursor_y

These contain the relative coordinates of the <u>cursor</u> within the current viewport. When the UIR 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.

cursor_pos

This contains the relative position of the cursor in the field (not in the viewport). The value of cursor_pos ranges from 1 to width.

input_length

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 UIR uses the invisible part of the input string to save characters that were pushed out of the field by insertions. Thus, input_length may be greater than width. However, in this case, the length of the input string actually returned to the user still ranges from 1 to width. The returned length of the input string is contained in the first byte of the input string.

slow_blink and fast_blink

These are the count-down timers used to get the correct blinking frequency for the cursor. This is a concern only in immediate mode, when the program no longer has control over the rate.

An important part of the Human Interface Guidelines is that the cursor blinks 8Ø times a minute, with one phase taking twice as long as the other. That is, if the insert cursor is active and under a character in the field, the character should be visible twice as long as the underline. And if the replace cursor is active, the inverse character should be visible twice as long as the normal character.

The initial values of slow_blink and fast_blink 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 program will get control in the middle of the blink loop. The application program must change slow_blink and fast_blink so that the cursor will again blink at the correct rate.

Language Interfaces

The User Input Routine can be used with Apple II Pascal, Applesoft BASIC, and 65Ø2 Assembly Language.

Pascal

The Apple II Pascal version of the UIR is part of the Console Driver (see Chapter 2) and therefore requires that the Pascal environment be loaded with the correct Attach files. The Console Driver is configured as unit number 13Ø. (Pascal 1.3 or the 128K version of Pascal 1.2 is required.)

To access the UIR, 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 UIR is in the form of a unitread. These calls are described later in this section.

Format of the Information Block

The following is the Pascal equivalent of the Information Block. The text of this data structure is in the file INPUT.INFO.TEXT on the /PASCON disk.

```
const max_terms=2Ø;           {Maximum number of terminators}
type  byte=Ø..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;        {Ø-use "fill_char" as fill character
                               1-use MouseText ghost underline}
      cursor:byte             {current cursor being used
                                 Ø-insert cursor
                                 1-replacement cursor}
      control:byte;           {Ø-Control chars will be ignored
                               1-Control chars allowed as input}
      beep:byte;              {Ø-errors will not be beeped
                               1-errors will be beeped}
      immediate:byte;         {Ø-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
                                 Ø-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
```

```
                               Ø-not terminated yet
                               not Ø-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"
                                Ø-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"
                                Ø-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};
```

Console Driver Calls

Initializing Input Information

To set the UIR Information Block to its default values, call the procedure

```
       init_mode:=24577;        {Console Driver command $6ØØ1}
       unitstatus(13Ø,Input_Info,init_mode);
```

or if the console driver is also to be initialized, use

```
        unitclear(13Ø);
```

By the Way: The variable Input Info in the unitstatus above call is not actually used by the UIR. It is needed in the unitstatus call because of its parameter structure.

The Pascal system performs an automatic unitclear when it is started up.

Retrieving Input Information

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

```
        get_info:=16385;       {Console Driver command $4ØØ1}
        unitstatus(13Ø,Input_Info,get_info);
```

where Input_Info is a record with the format specified in "Format of the Information Block." The include file INPUT.INFO.TEXT can be used to define the variable Input_Info.

Setting Input Information

To change the data in the UIR Information Block, call the procedure

```
        set_info:=8193;        {Console Driver command $2ØØ1}
        unitstatus(13Ø,Input_Info,set_info);
```

where Input_Info is a record with the format specified in "Format of the Information Block." If this call is never made, the UIR uses the default values.

Changing any parameters in the record will have no effect until the unitstatus call is made.

Calling the User Input Routine

To call the UIR, call the procedure:

```
        unitread(13Ø,Input_Str,max_length);
```

where Input_Str is a string, supplied by the calling routine, where the UIR will store the user's keystrokes. Max_length specifies the maximum number of characters that will fit in the string (usually 8Ø unless Input_Str is defined as an extended string).

If the input string has an initial value, the UIR assumes that it is a default value and displays it.

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

Pascal Examples

The program named Demo can be used to try out many of the UIR's features.

In the simplest use of the UIR, the application program uses the Console Driver to display a question on the screen and then calls the UIR for the answer. Here is a program segment that illustrates this:

```
VAR
  question,answer:string;
  ...
  ...
  question:='What is your name ? ';
  answer:='';
  unitwrite(13Ø,question[1],length(question));
  unitread(13Ø,answer,8Ø);
```

If the application program is to provide a default name:

```
VAR
  question,answer:string;
  ...
  ...
  question:='What is your name ? ';
  answer:='Fred';
  unitwrite(13Ø,question[1],length(question));
  unitread(13Ø,answer,8Ø);
```

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

```
CONST
  get_info=16385;         {Console Driver command $4ØØ1}
  set_info=8193;          {Console Driver command $2ØØ1}
VAR
  question,answer:string;
  Input_Info:packed record
           {use record structure in 5.1.2}
           end;
```

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

unitstatus(13Ø,Input_Info,get_info);

{Change the desired parameters}

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

{Set the updated Information Block}

unitstatus(13Ø,Input_Info,set_info);

{The rest of the logic is the same}

question:='What is your name ? ';
answer:='Fred';
unitwrite(13Ø,question[1],length(question));
unitread(13Ø,answer,8Ø);
```

BASIC

The Applesoft BASIC version of the User Input Routine consists 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. Five such commands are available:

&INITINPUT	- initialize Information Block
&GETINFO(IB%)	- get Information Block
&SETINFO(IB%)	- set Information Block
&INPUT(IS$)	- call UIR
&EXITINPUT	- remove package from ampersand hooks

where IB% represents any legal integer array and IS% any integer string name.

If these are to co-exist with other ampersand calls, CONDAMP.REL must be loaded last.

The BASIC disk contains the UIR in a relocatable file named CONUIR.REL. The RLOAD facility (one of the ProDOS Assembler Tools) must be used to load CONUIR.REL from within the application program.

&INITINPUT

This call initializes the Information Block to its default values. The default values are defined earlier in this chapter.

&GETINFO(IB%)

This call retrieves the current Information Block and stores it in the integer array named IB%. The array IB% should be dimensioned for at least 22+3*max_terms integers, where max_terms is currently 2Ø.

Here is the content of each integer in IB%:

```
     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%(1Ø) = 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_list
     IB%(75) = origin_x
     IB%(76) = origin_y
     IB%(77) = cursor_x
     IB%(78) = cursor_y
     IB%(79) = cursor_pos
     IB%(8Ø) = input_length
     IB%(81) = slow_blink
     IB%(82) = fast_blink
```

&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.

&INPUT(IS$)

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

&EXITINPUT

This call terminates the UIR and disconnects the ampersand package.

BASIC Examples

The program named STARTUP on the BASCON disk can be used to try out many of the UIR's features.

In the simplest use of the UIR, the application program displays a question on the screen and then calls the UIR for the answer.

The three programs that follow are for illustration only; they won't run as is.

Here is a program segment that illustrates:

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

If the application program is to provide a default name:

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

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

```
    PRINT CHR$(4);"brun release":  REM release memory buffers
    PRINT "pr#3"
    REM load & initialize Console Driver & UIR
    A1 = Ø :  A2 = Ø
    PRINT CHR$(4);"brun rboot"
    A1 = USR(Ø),"conuir.rel":  REM load Console Driver & UIR
    A2 = USR(Ø),"condamp.rel":  REM load ampersand interface
    CALL A2
    &STCDADR(A1)

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

Assembly Language

The Assembly-Language version of the User Input Routine provides a set of calls similar to ProDOS MLI calls. They provide four functions:

- Initializing Input Information

- Retrieving Input Information

- Setting Input Information

- Calling the User Input Routine

The ASMCON disk contains an absolute binary file named CONUIR.OBJ and a relocatable file named CONUIR.REL.

CONUIR.OBJ was generated from CONUIR.REL, with the starting address $4ØØØ. If this starting address is not satisfactory for the application program, use the RELOCATOR program to generate a new absolute file that starts at the desired location.

Format of the Information Block

Here is the Assembler equivalent of the Information Block:

```
maxterms        equ     2Ø      ;Maximum number of terminators
```

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

;                               Termination Information
;                               -----------------------
exittype        db      Ø       ;Indicates which termination condition
                                ;  occurred
                                ;Ø-not terminated yet
                                ;not Ø-index into terminating char list
lastevent       db      Ø       ;last event type (not used)
lastch          db      Ø       ;character user keyed in
lastmod         db      Ø       ;keypress modifier
nchars          db      Ø       ;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"
                                   ;Ø-none
                                   ;1-Open Apple
                                   ;2-Solid Apple
                                   ;3-Either Open or Solid Apple
termlist        ds      maxterms ;Termination types for each char in
                                 ;  "charlist"
                                   ;Ø-terminate input
                                   ;1-interrupt input
```

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

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

Format of Calls

The UIR 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 UIR. The programmer will determine this location when the routine is relocated in memory. In the application program, there should be a statement of the form:

```
        INPUT    EQU      nnnn
```

where nnnn is the starting address of the UIR.

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

When the UIR 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 UIR 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.

Initializing Input Information

This call initializes the Information Block to its default values (defined earlier in this chapter). Its format is:

```
          JSR      INPUT
          DB       1Ø             ;command number for Initialize
          DW       Ø
```

Retrieving Input Information

This call retrieves the current contents of the Input Information Block. Its format 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. The format of the Information Block is defined earlier in this chapter.

Setting Input Information

This call sets the Input Information Block to values in the specified buffer. Its format is:

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

where INPUTINFO is the address of the buffer.

Calling the User Input Routine

This call performs the actual input. Its format:

```
          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.

Assembly-Language Examples

The demonstration program (STARTUP) on the assembly-language disk can be used to try out many of the UIR's features.

In the simplest use of the UIR, the application program displays a question on the screen and then calls the UIR for the answer. Here is a program segment that illustrates this:

```
QUESTION STR "What is your name ? "
ANSWER   STR ""
         DS  81-*+ANSWER
MAXLEN   EQU *-ANSWER-1
PARAM    DW  ANSWER
         DB  MAXLEN
         ...
         ...
;
;        display question
;
         LDY #Ø
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 program is 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 #Ø
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 program is 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 #8Ø
         STA INPUTINFO        ;width
         LDA #"."
         STA INPUTINFO+1      ;fillchar
;
;        set Information Block
;
         JSR INPUT
         DS  12
         DW  INPUTINFO
;
;        display question
;
         LDY #Ø
LOOP     LDA QUESTION+1,Y
         JSR $FDED            ;display the char
         INY
         CPY QUESTION
         BCC LOOP
;
;        get answer
;
         JSR INPUT
```

```
        DB  13
        DW  PARAM
```