

## **Devices**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> Devices		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		March 28, 2025	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>Devices</b>	<b>1</b>
1.1	Amiga® RKM Devices: 4 Console Device . . . . .	1
1.2	4 Console Device / Console Device Commands and Functions . . . . .	2
1.3	4 Console Device / Device Interface . . . . .	3
1.4	4 / Device Interface / Console Device Units . . . . .	4
1.5	4 / Device Interface / Opening The Console Device . . . . .	4
1.6	4 / Device Interface / Closing The Console Device . . . . .	6
1.7	4 Console Device / About Console I/O . . . . .	6
1.8	4 / About Console I/O / Exec Functions And The Console Device . . . . .	7
1.9	4 / About Console I/O / General Console Screen Output . . . . .	7
1.10	4 / About Console I/O / Console Keyboard Input . . . . .	7
1.11	4 Console Device / Writing to the Console Device . . . . .	8
1.12	4 / Writing to the Console Device / Hints For Writing Text . . . . .	8
1.13	4 / Writing to the Console Device / Control Sequences For Window Output . . . . .	8
1.14	4 / / Set Graphic Rendition Implementation Notes . . . . .	12
1.15	4 / Writing to the Console Device / Example Console Control Sequences . . . . .	15
1.16	4 Console Device / Reading from the Console Device . . . . .	15
1.17	4 / Reading from the Console Device / Information About The Input Stream . . . . .	16
1.18	4 / Reading from the Console Device / Cursor Position Report . . . . .	17
1.19	4 / Reading from the Console Device / Window Bounds Report . . . . .	17
1.20	4 Console Device / Copy and Paste Support . . . . .	18
1.21	4 Console Device / Selecting Raw Input Events . . . . .	19
1.22	4 Console Device / Input Event Reports . . . . .	20
1.23	4 Console Device / Using the Console Device Without a Window . . . . .	22
1.24	4 Console Device / Where Is All the Keymap Information? . . . . .	23
1.25	4 Console Device / Console Device Caveats . . . . .	24
1.26	4 Console Device / Additional Information on the Console Device . . . . .	24

---

# Chapter 1

## Devices

### 1.1 Amiga® RKM Devices: 4 Console Device

The console device provides the text-oriented interface for Intuition windows. It acts like an enhanced ASCII terminal obeying many of the standard ANSI sequences as well as special sequences unique to the Amiga. The console device also provides a copy-and-paste facility and an internal character map to redraw a window when it is resized.

#### NEW CONSOLE FEATURES FOR VERSION 2.0

Feature	Description
-----	-----
CONU_LIBRARY	New #define
CONU_STANDARD	New #define
CONU_CHARMAP	Console Unit
CONU_SNIPMAP	Console Unit
CONFLAG_DEFAULT	Console Flag
CONFLAG_NODRAW_ON_NEWSIZE	Console Flag

#### Compatibility Warning:

-----  
The new features for the 2.0 console device are not backwards compatible.

Console Device Commands and Functions  
Device Interface  
About Console I/O  
Writing to the Console Device  
Reading from the Console Device  
Copy and Paste Support  
Selecting Raw Input Events  
Input Event Reports  
Using the Console Device Without a Window  
Where Is All the Keymap Information?  
Console Device Caveats  
Console Device Example Code  
Additional Information on the Console Device

## 1.2 4 Console Device / Console Device Commands and Functions

Command -----	Operation -----
CD_ASKDEFAULTKEYMAP	Get the current default keymap.
CD_ASKKEYMAP	Get the current key map structure for this console.
CD_SETDEFAULTKEYMAP	Set the current default keymap.
CD_SETKEYMAP	Set the current key map structure for this console.
CMD_CLEAR	Remove any reports waiting to satisfy read requests from the console input buffer.
CMD_READ	Read the next input, generally from the keyboard. The form of this input is as an ANSI byte stream.
CMD_WRITE	Write a text record to the display interpreting any ANSI control characters in the record.

### Console Device Function -----

CDInputHandler()	Handle an input event for the console device.
RawKeyConvert()	Decode raw input classes and convert input events of type IECLASS_RAWKEY to ANSI bytes based on the keymap in use.

### Exec Functions as Used in This Chapter -----

AbortIO()	Abort an I/O request to the console device.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the console device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
GetMsg()	Get the next message from the reply port.
OpenDevice()	Obtain use of the console device. You specify the type of unit and its characteristics in the call to OpenDevice().
OpenLibrary()	Gain access to a library.
OpenWindow()	Open an intuition window.
SendIO()	Initiate a command and return immediately (asynchronous request).

---

Wait()	Wait for one or more signals.
WaitIO()	Wait for completion of an I/O request and remove it from the reply port.
WaitPort()	Wait for the reply port to be non-empty. Does not remove the message from port.

#### Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure for use in communicating with the console device.
CreatePort()	Create a message port for reply messages from the console device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete the extended I/O request structure created by CreateExtIO().
DeletePort()	Delete the message port created by CreatePort().

## 1.3 4 Console Device / Device Interface

The console device operates like the other Amiga devices. To use it, you must first open the console device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the console device is called IOStdReq.

```
struct IOStdReq
{
    struct Message io_Message;
    struct Device *io_Device; /* device node pointer */
    struct Unit *io_Unit; /* unit (driver private) */
    UWORD io_Command; /* device command */
    UBYTE io_Flags;
    BYTE io_Error; /* error or warning num */
    ULONG io_Actual; /* actual number of bytes transferred */
    ULONG io_Length; /* requested number bytes transferred */
    APTR io_Data; /* points to data area */
    ULONG io_Offset; /* offset for block structured devices */
};
```

See the include file exec/io.h for the complete structure definition.

Console Device Units  
 Opening The Console Device  
 Closing The Console Device

## 1.4 4 / Device Interface / Console Device Units

The console device provides four units, three that require a console window and one that does not. The unit type is specified when you open the device. See the Opening the Console Device section below for more details.

The CONU\_STANDARD unit (0) is generally used with a SMART\_REFRESH window. This unit has the least amount of overhead (e.g., memory usage and rendering time), and is highly compatible with all versions of the operating system.

As of V36, a character mapped console device was introduced. There are two variations of character mapped console units. Both must be used with SIMPLE\_REFRESH windows and both have the ability to automatically redraw a console window when resized or revealed.

A character mapped console can be opened which allows the user to drag-select text with the mouse and COPY the highlighted area. The copied text can then be PASTED into other console windows or other windows which support reading data from the clipboard device.

Character mapped console units have more overhead than standard consoles (e.g., rendering times and memory usage).

The CONU\_LIBRARY unit (-1) does not require a console window. It is designed to be primarily used with the console device functions and also with the console device commands that do not require a console window.

The Amiga uses the ECMA-94 Latin1 International 8-bit character set. See Appendix A (page 397) for a table of character codes.

## 1.5 4 / Device Interface / Opening The Console Device

Four primary steps are required to open the console device:

- \* Create a message port using `CreatePort()`. Reply messages from the device must be directed to a message port.
- \* Create an I/O request structure of type `IOStdReq`. The `IOStdReq` structure is created by the `CreateExtIO()` function. `CreateExtIO` will initialize your I/O request to point to your reply port.
- \* Open an Intuition window and set a pointer to it in the `io_Data` field of the `IOStdReq` and the size of the window in the `io_Length` field. This is the window to which the console will be attached. The window must be `SIMPLE_REFRESH` for use with the `CONU_CHARMAP` and `CONU_SNIPMAP` units.
- \* Open the console device. Call `OpenDevice()` passing it the I/O request and the type of console unit set in the `unit` and `flags` fields. Console unit types and flag values are listed below.

Console device units:

---

- \* CONU\_LIBRARY - Return the device library vector pointer used for calling console device functions. No console is opened.
- \* CONU\_STANDARD - Open a standard console.
- \* CONU\_CHARMAP - Open a console with a character map.
- \* CONU\_SNIPMAP - Open a console with a character map and copy-and-paste support.

See the include file `devices/conunit.h` for the unit definitions and the Amiga ROM Kernel Reference Manual: Includes and Autodocs for an explanation of each unit.

#### No Changes Required

CONU\_STANDARD has a numeric value of zero to insure compatibility with pre-V36 applications. CONU\_LIBRARY has a numeric value of negative one and is also compatible with pre-V36 applications.

#### Console device flags:

- \* CONFLAG\_DEFAULT - The console device will redraw the window when it is resized.
- \* CONFLAG\_NODRAW\_ON\_NEWSIZE - The console device will not redraw the window when it is resized

The character map units, CONU\_CHARMAP and CONU\_SNIPMAP, are the only units which use the flags parameter to set how the character map is used. CONU\_STANDARD units ignore the flags parameter.

See the include file `devices/conunit.h` for the flag definitions and the Amiga ROM Kernel Reference Manual: Includes and Autodocs for an explanation of the flags.

```
struct MsgPort *ConsoleMP;    /* Message port pointer */
struct IOStdReq *ConsIO;      /* I/O structure pointer */
struct Window *win = NULL;    /* Window pointer */

struct NewWindow nw =
{
    10, 10,                    /* starting position (left,top) */
    620,180,                  /* width, height */
    -1,-1,                    /* detailpen, blockpen */
    CLOSEWINDOW,              /* flags for idcmp */
    WINDOWDEPTH|WINDOWSIZING|
    WINDOWDRAG|WINDOWCLOSE|
    SIMPLE_REFRESH|ACTIVATE,  /* window flags */
    NULL,                      /* no user gadgets */
    NULL,                      /* no user checkmark */
    "Console Test",           /* title */
    NULL,                      /* pointer to window screen */
    NULL,                      /* pointer to super bitmap */
    100,45,                    /* min width, height */
    640,200,                   /* max width, height */
}
```



```

    WBMCHSCREEN                      /* open on workbench screen */
};

    /* Create reply port console */
if (!(ConsoleMP = CreatePort("RKM.Console",0)))
    cleanexit("Can't create write port\n",RETURN_FAIL);

    /* Create message block for device I/O */
if (!(ConsIO = CreateExtIO(ConsoleMP,sizeof(struct IOStdReq))))
    cleanexit("Can't create IORequest\n",RETURN_FAIL);

    /* Open a window - we assume intuition.library is already open */
if (!(win = OpenWindow(&nw)))
    cleanexit("Can't open window\n",RETURN_FAIL);

    /* Set window pointer and size in I/O request */
ConsIO->io_Data = (APTR) win;
ConsIO->io_Length = sizeof(struct Window);

    /* Open the console device */
if (error = OpenDevice("console.device",CONU_CHARMAP,ConsIO,
                      CONFLAG_DEFAULT))
    cleanexit("Can't open console.device\n",RETURN_FAIL);

```

## 1.6 4 / Device Interface / Closing The Console Device

Each `OpenDevice()` must eventually be matched by a call to `CloseDevice()`.

All I/O requests must be complete before `CloseDevice()`. If any requests are still pending, abort them with `AbortIO()`.

```

if (!(CheckIO(ConsIO)))
    AbortIO(ConsIO);    /* Ask device to abort any pending requests */

WaitIO(ConsIO);        /* Wait for abort, then clean up */
CloseDevice(ConsIO);   /* Close console device */

```

## 1.7 4 Console Device / About Console I/O

The console device may be thought of as a kind of terminal. You send character streams to the console device; you also receive them from the console device. These streams may be characters, control sequences or a combination of the two.

Console I/O is closely associated with the Amiga Intuition interface; a console must be tied to a window that is already opened. From the Window data structure, the console device determines how many characters it can display on a line and how many lines of text it can display in a window without clipping at any edge.

You can open the console device many times, if you wish. The result of each open call is a new console unit. AmigaDOS and Intuition see to it

that only one window is currently active and its console, if any, is the only one (with a few exceptions) that receives notification of input events, such as keystrokes. Later in this chapter you will see that other Intuition events can be sensed by the console device as well.

Introducing...

-----

For this entire chapter the characters "<CSI>" represent the control sequence introducer. For output you may use either the two-character sequence <Esc>[ (0x1B 0x5B) or the one-byte value 0x9B. For input you will receive 0x9B unless the sequence has been typed by the user.

Exec Functions And The Console Device  
General Console Screen Output  
Console Keyboard Input

## 1.8 4 / About Console I/O / Exec Functions And The Console Device

The various Exec functions such as DoIO(), SendIO(), AbortIO() and CheckIO() operate normally. The only caveats are that CMD\_WRITE may cause your application to wait internally, even with SendIO(), and a task using CMD\_READ to wait on a response from a console is at the user's mercy. If the user never reselects that window, and the console response provides the only wake-up call, that task will sleep forever.

## 1.9 4 / About Console I/O / General Console Screen Output

Console character screen output (as compared to console command sequence transmission) outputs all standard printable characters (character values hex 20 through 7E and A0 through FF) normally.

Many control characters such as BACKSPACE (0x8) and RETURN (0x13) are translated into their exact ANSI equivalent actions. The LINEFEED character (0xA) is a bit different in that it can be translated into a RETURN/LINEFEED action. The net effect is that the cursor moves to the first column of the next line whenever an <LF> is displayed. This option is set via the mode control sequences discussed under Control Sequences for Window Output.

## 1.10 4 / About Console I/O / Console Keyboard Input

If you read from the console device, the keyboard inputs are preprocessed for you and you will get ASCII characters, such as "B." Most normal text-gathering programs will read from the console device in this manner. Some programs may also ask to receive raw events in their console stream. Keypresses are converted to ASCII characters or CSI sequences via the keymap associated with the unit.

---

## 1.11 4 Console Device / Writing to the Console Device

You write to the console device by passing an I/O request to the device with a pointer to the write buffer set in `io_Data`, the number of bytes in the buffer set in `io_Length` and `CMD_WRITE` set in `io_Command`.

```
UBYTE *outstring= "Make it so.";

ConsIO->io_Data = outstring;
ConsIO->io_Length = strlen(outstring);
ConsIO->io_Command = CMD_WRITE;
DoIO(ConsIO);
```

You may also send NULL-terminated strings to the console device in the same manner except that `io_Length` must be set to -1.

```
ConsIO->io_Data = "\033[3mOh boy.";
ConsIO->io_Length = -1;
ConsIO->io_Command = CMD_WRITE;
DoIO(ConsIO);
```

The fragment above will output the string "Oh boy." in italics. Keep in mind that setting the text rendition to italics will remain in effect until you specifically instruct the console device to change it to another text style.

Hints For Writing Text  
Control Sequences For Window Output  
Example Console Control Sequences

## 1.12 4 / Writing to the Console Device / Hints For Writing Text

256 Is A Nice Round Number

-----

You must keep in mind that the console device locks all layers while writing text. To avoid, problems with this, it is best to send smaller rather larger numbers of character to be written. We recommend no more than 256 bytes per write as the optimum size

Turn Off The Cursor

-----

If your console is attached to a V1.2/1.3 SuperBitmap window, you will not see a cursor rendered. For output speed and compatibility with future OS versions which may visibly render the cursor, you should send the cursor-off sequence (ESC[0 p) whenever you open or reset (ESCc) a SuperBitmap window's console.

## 1.13 4 / Writing to the Console Device / Control Sequences For Window Output

The following table lists functions that the console device supports, along with the character stream that you must send to the console to produce the effect. For more information on the control sequences,

consult the console.doc of the Amiga ROM Kernel Reference Manual: Includes and Autodocs. The table uses the second form of <CSI>, that is, the hex value 0x9B, to minimize the number of characters to be transmitted to produce a function.

A couple of notes about the table. If an item is enclosed in square brackets, it is optional and may be omitted. For example, for INSERT [N] CHARACTERS the value for N is shown as optional. The console device responds to such optional items by treating the value of N as 1 if it is not specified. The value of N or M is always a decimal number, having one or more ASCII digits to express its value.

#### ANSI CONSOLE CONTROL SEQUENCES

Console Command -----	Sequence of Characters (in Hexadecimal Form) -----
BELL (Flash the display ie; do an Intuition DisplayBeep())	07
BACKSPACE (move left one column)	08
HORIZONTAL TAB (move right one tab stop)	09
LINEFEED (move down one text line as specified by the mode function)	0A
VERTICAL TAB (move up one text line)	0B
FORMFEED (clear the console's window)	0C
CARRIAGE RETURN (move to first column)	0D
SHIFT IN (undo SHIFT OUT)	0E
SHIFT OUT (set MSB of each character before displaying)	0F
ESC (escape; can be part of the control sequence introducer)	1B
INDEX (move the active position down one line)	84
NEXT LINE (go to the beginning of the next	85

---

---

line)	
HORIZONTAL TABULATION SET (Set a tab at the active cursor position)	88
REVERSE INDEX (move the active position up one line)	8D
CSI (control sequence introducer)	9B
RESET TO INITIAL STATE 1B	63
INSERT [N] CHARACTERS 9B [N] (insert one or more spaces, shifting the remainder of the line to the right)	40
CURSOR UP [N] CHARACTER POSITIONS (default = 1)	9B [N] 41
CURSOR DOWN [N] CHARACTER POSITIONS (default = 1)	9B [N] 42
CURSOR FORWARD [N] CHARACTER POSITIONS (default = 1)	9B [N] 43
CURSOR BACKWARD [N] CHARACTER (default = 1)	9B [N] 44
CURSOR NEXT LINE [N] (to column 1)	9B [N] 45
CURSOR PRECEDING LINE [N] (to column 1)	9B [N] 46
CURSOR POSITION (where N is row, M is column, and semicolon (hex 3B) must be present as a separator, or if row is left out, so the console device can tell that the number after the semicolon actually represents the column number)	9B [N] [3B M] 48
CURSOR HORIZONTAL TABULATION (move cursor forward to Nth tab position)	9B [N] 49
ERASE IN DISPLAY (only to end of display)	9B 4A
ERASE IN LINE (only to end of line)	9B 4B
INSERT LINE	9B 4C

---

(above the line containing the cursor)

DELETE LINE (remove current line, move all lines up one position to fill gap, blank bottom line)	9B 4D
DELETE CHARACTER [N] (that cursor is sitting on and to the right if [N] is specified)	9B [N] 50
SCROLL UP [N] LINES (Remove line(s) from top of window, move all other lines up, blanks [N] bottom lines)	9B [N] 53
SCROLL DOWN [N] LINES (Remove line(s) from bottom of window, move all other lines down, blanks [N] top lines)	9B [N] 54
CURSOR TABULATION CONTROL (where N = 0 set tab, 2 = clear tab, 5 = clear all tabs.)	9B [N] 57
CURSOR BACKWARD TABULATION (move cursor backward to Nth tab position.)	9B [N] 5A
SET LINEFEED MODE (cause LINEFEED to respond as RETURN-LINEFEED)	9B 32 30 68
RESET NEWLINE MODE (cause LINEFEED to respond only as LINEFEED)	9B 32 30 6C
DEVICE STATUS REPORT (cause console device to insert a CURSOR POSITION REPORT into your read stream; see "Reading from the Console Device" for more information)	9B 36 6E
SELECT GRAPHIC RENDITION (select text style, character color, character cell color, background color)	9B N 3B 3N 3B 4N 3B >N 6D (See note below).

For SELECT GRAPHIC RENDITION, any number of parameters, in any order, are valid. They are separated by semicolons.

The parameters follow:

<text style> =

0	Plain text	8	Concealed mode
1	Boldface	22	Normal color, not bold (V36)
2	faint (secondary color)	23	Italic off (V36)
3	Italic	24	Underscore off (V36)
4	Underscore	27	Reversed off (V36)
7	Reversed character/cell colors	28	Concealed off (V36)

<character color> =

30-37 System colors 0-7 for character color.  
 39 Reset to default character color  
 Transmitted as two ASCII characters.

<character cell color> =

40-47 System colors 0-7 for character cell color.  
 39 Reset to default character color  
 Transmitted as two ASCII characters.

<background color> =

>0-7System colors 0-7 for background color.(V36)  
 You must specify the ">" in order for this to  
 be recognized and it must be the last parameter.

For example, to select bold face, with color 3 as the character color, and color 0 as the character cell color and the background color, send the hex sequence:

```
9B 31 3B 33 33 3B 34 30 3B 3E 30 6D
```

representing the ASCII sequence:

```
<CSI>1;33;40;>0m
```

where <CSI> is the control sequence introducer, here used as the single character value 0x9B.

```
Go Easy On The Eyes.
```

```
-----  
In most cases, the character cell color and the background color  
should be the same.
```

Set Graphic Rendition Implementation Notes

## 1.14 4 // Set Graphic Rendition Implementation Notes

Previous versions of the operating system did not support the global background color sequence as is listed above. Instead, the background color was set by setting the character cell color and then clearing the screen (e.g., a FORMFEED).

In fact, vacated areas of windows (vacated because of an ERASE or SCROLL) were filled in with the character cell color. This is no longer the case. Now, when an area is vacated, it is filled in with the global background

color.

SMART\_REFRESH windows are a special case:

Under V33-V34:

The cell color had to be set and a FORMFEED (clear window) needed to be sent on resize or immediately to clear the window and set the background color.

For example, if you took a CLI window and sent the sequence to set the cell color to something other than the default, the background color would not be changed immediately (contrary to what was expected).

If you then sent a FORMFEED, the background color would change, but if you resized the window larger, you would note that the newly revealed areas were filled in with PEN 0.

Under V36-V37 (non-character mapped):

You need to set the global background color and do a FormFeed. The background color will then be used to fill the window, but like V33-V34, if you make the window larger, the vacated areas will be filled in with PEN 0.

Under V36-V37 (character mapped):

You need to set the global background color, the window is redrawn immediately (because we have the character map) and will be correctly redrawn with the global background color on subsequent resizes.

The sequences in the next table are not ANSI standard sequences, they are private Amiga sequences. In these command descriptions, length, width, and offset are comprised of one or more ASCII digits, defining a decimal value.

#### AMIGA CONSOLE CONTROL SEQUENCES

Console Command	Sequence of Characters (in Hex Format)
-----	-----
ENABLE SCROLL (default)	9B 3E 31 68
DISABLE SCROLL	9B 3E 31 6C
AUTOWRAP ON (default)	9B 3F 37 68
AUTOWRAP OFF	9B 3F 37 6C
SET PAGE LENGTH (in character raster lines, causes console to recalculate, using current font, how many text lines will fit on the page)	9B <length> 74
SET LINE LENGTH (in character positions, using current font, how many characters should be placed on each line)	9B <width> 75
SET LEFT OFFSET (in raster columns, how far from	9B <offset> 78



the left of the window should  
the text begin)

SET TOP OFFSET 9B <offset> 79  
(in raster lines, how far  
from the top of the window's  
RastPort should the topmost  
line of the character begin)

SET RAW EVENTS 9B <events> 7B  
(set the raw input events that  
will trigger an INPUT EVENT  
REPORT. see the  
"Selecting Raw Input Events"  
section below for more details.)

INPUT EVENT REPORT 9B <parameters> 7C  
(returned by the console device  
in response to a raw event  
set by the SET RAW EVENT sequence.  
See the "Input Event Reports"  
section below for more details.)

RESET RAW EVENTS 9B <events> 7D  
(reset the raw events set by  
the SET RAW EVENT sequence. See the  
"Selecting Raw Input Events"  
section below.)

SPECIAL KEY REPORT 9B <keyvalue> 7E  
(returned by the console device  
whenever HELP, or one of the  
function keys or arrow keys is  
pressed. Some sequences do not end  
with 7E)

SET CURSOR RENDITION 9B N 20 70  
(make the cursor visible or invisible:  
Note-turning off the cursor increases  
text output speed)

Invisible: 9B 30 20 70  
Visible: 9B 20 70

WINDOW STATUS REQUEST 9B 30 20 71  
(ask the console device to tell you  
the current bounds of the window,  
in upper and lower row and column  
character positions. User may have  
resized or repositioned it. See  
"Window Bounds Report" below.)

WINDOW BOUNDS REPORT 9B 31 3B 31 3B <bot margin>  
(returned by the console device in  
response to a WINDOW STATUS REQUEST  
sequence) 3B <right margin> 72

RIGHT AMIGA V PRESS 9B 30 20 76  
(returned by the console device when  
the user presses RIGHT-AMIGA-V. See  
the "Copy and Paste Support" section  
below for more details.)

Give Back What You Take.  
-----

The console device normally handles the SET PAGE LENGTH, SET LINE  
LENGTH, SET LEFT OFFSET, and SET TOP OFFSET functions automatically.  
To allow it to do so again after setting your own values, send the  
functions without a parameter.

1.15 4 / Writing to the Console Device / Example Console Control Sequences

	Character String Equivalent -----	Numeric (hex) Equivalent -----
Move cursor right by 1:	<CSI>C or <CSI>1C	9B 43 9B 31 43
Move cursor right by 20:	<CSI>20C	9B 32 30 43
Move cursor to upper-left corner (Home Position)	<CSI>H or <CSI>1;1H or <CSI>;1H or <CSI>1;H	9B 48 9B 31 3B 31 48 9B 3B 31 48 9B 31 3B 48
Move cursor to the fourth column of the first line of the window:	<CSI>1;4H or <CSI>;4H	9B 31 3B 34 48 9B 3B 34 48
Clear the window:	<FF> or CTRL-L or	0C
(home and clear to end of window)	<CSI>H<CSI>J	9B 48 9B 4A

1.16 4 Console Device / Reading from the Console Device

Reading input from the console device returns an ANSI 3.64 standard byte stream. This stream may contain normal characters and/or RAW input event information. You may also request other RAW input events using the SET RAW EVENTS and RESET RAW EVENTS control sequences discussed below. See "Selection of Raw Input Events."

Generally, console reads are performed asynchronously so that your program can respond to other events and other user input (such as menu selections) when the user is not typing on the keyboard. To perform asynchronous I/O, an I/O request is sent to the console using the SendIO() function (rather than a synchronous DoIO() which would wait until the read request returned with a character).

You read from the console device by passing an I/O request to the device

---

with a pointer to the read buffer set in `io_Data`, the number of bytes in the buffer set in `io_Length` and `CMD_READ` set in `io_Command`.

```
#define READ_BUFFER_SIZE 25
char ConsoleReadBuffer[READ_BUFFER_SIZE];

ConsIO->io_Data = (APTR)ConsoleReadBuffer;
ConsIO->io_Length = READ_BUFFER_SIZE;
ConsIO->io_Command = CMD_READ;
SendIO(ConsIO);
```

You May Get Less Than You Bargained For.

-----  
A request for more than one character may be satisfied by the receipt of only one character. If you request more than one character, you will have to examine the `io_Actual` field of the request when it returns to determine how many characters you have actually received.

After sending the read request, your program can wait on a combination of signal bits including that of the reply port you created. The following fragment demonstrates waiting on both a queued console read request, and Window IDCMP messages:

```
ULONG conreadsig = 1 << ConsoleMP->mp_SigBit;
ULONG windowsig = 1 << win->UserPort->mp_SigBit;

/* A character, or an IDCMP msg, or both will wake us up */
ULONG signals = Wait(conreadsig | windowsig);

if (signals & conreadsig)
{
    /* Then check for a character */
};

if (signals & windowsig)
{
    /* Then check window messages */
};
```

Information About The Input Stream

Cursor Position Report

Window Bounds Report

## 1.17 4 / Reading from the Console Device / Information About The Input Stream

For the most part, keys whose keycaps are labeled with ANSI-standard characters will ordinarily be translated into their ASCII-equivalent character by the console device through the use of its keymap. Keymap information can be found in the "Keymap Library" chapter of the Amiga ROM Kernel Reference Manual: Libraries.

For keys other than those with normal ASCII equivalents, an escape sequence is generated and inserted into your input stream. For example, in the default state (no raw input events selected) the function, arrow and special keys (reserved for 101 key keyboards) will cause the sequences

shown in the next table to be inserted in the input stream.

#### SPECIAL KEY REPORT SEQUENCES

Key	Unshifted Sends	Shifted Sends
---	-----	-----
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
F3	<CSI>2~	<CSI>12~
F4	<CSI>3~	<CSI>13~
F5	<CSI>4~	<CSI>14~
F6	<CSI>5~	<CSI>15~
F7	<CSI>6~	<CSI>16~
F8	<CSI>7~	<CSI>17~
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
F11	<CSI>20~	<CSI>30~ (101 key keyboard)
F12	<CSI>21~	<CSI>31~ (101 key keyboard)
HELP	<CSI>?~	<CSI>?~ (same sequence for both)
Insert	<CSI>40~	<CSI>50~ (101 key keyboard)
Page Up	<CSI>41~	<CSI>51~ (101 key keyboard)
Page Down	<CSI>42~	<CSI>52~ (101 key keyboard)
Pause/Break	<CSI>43~	<CSI>53~ (101 key keyboard)
Home	<CSI>44~	<CSI>54~ (101 key keyboard)
End	<CSI>45~	<CSI>55~ (101 key keyboard)
Arrow keys:		
Up	<CSI>A	<CSI>T
Down	<CSI>B	<CSI>S
Left	<CSI>D	<CSI>A (notice the space
Right	<CSI>C	<CSI>@ after <CSI>)

## 1.18 4 / Reading from the Console Device / Cursor Position Report

If you have sent the DEVICE STATUS REPORT command sequence, the console device returns a cursor position report into your input stream. It takes the form:

```
<CSI><row>;<column>R
```

For example, if the cursor is at column 40 and row 12, here are the ASCII values (in hex) you receive in a stream:

```
9B 34 30 3B 31 32 52
```

## 1.19 4 / Reading from the Console Device / Window Bounds Report

A user may have either moved or resized the window to which your console is bound. By issuing a WINDOW STATUS REPORT to the console, you can read the current position and size in the input stream. This window bounds report takes the following form:

```
<CSI>1;1;<bottom margin>;<right margin> r
```

The bottom and right margins give you the window row and column dimensions as well. For a window that holds 20 lines with 60 characters per line, you will receive the following in the input stream:

```
9B 31 3B 31 3B 32 30 3B 36 30 20 72
```

## 1.20 4 Console Device / Copy and Paste Support

As noted above, opening the console device with a unit of CONU\_SNIPMAP allows the user to drag-select text with the mouse and copy the selection with Right-Amiga-C.

Internally, the snip is copied to a private buffer managed by the console device where it can be copied to other console device windows by pressing Right-Amiga-V.

However, your application should assume that the user is running the Conclip" utility which is part of the standard Workbench 2.0 environment. Conclip copies snips from the console device to the clipboard device where they can be used by other applications which support reading from the clipboard.

When Conclip is running and the user presses Right-Amiga-V, the console device puts an escape sequence in your read stream - <CSI>0 v (Hex 9B 30 20 76) - which tells you that the user wants to paste text from the clipboard.

Upon receipt of this sequence, your application should read the contents of the clipboard device, make a copy of any text found there and then release the clipboard so that it can be used by other applications. See the "Clipboard Device" chapter for more information on reading data from it.

You paste what you read from the clipboard by using successive writes to the console. In order to avoid problems with excessively long data in the clipboard, you should limit the size of writes to something reasonable. (We define reasonable as no more than 1K per write with the ideal amount being 256 bytes.) You should also continue to monitor the console read stream for additional use input, paster requests and, possibly, RAW INPUT EVENTS while you are doing this.

You should not open a character mapped console unit with COPY capability if you are unable to support PASTE from the clipboard device. The user will reasonably expect to be able to PASTE into windows from which a COPY can be done.

Keep in mind that users do make mistakes, so an UNDO mechanism for aborting a PASTE is highly desirable - particularly if the user has just accidentally pasted text into an application like a terminal program which is sending data at a slow rate.

Use CON:; You'll Be Glad You Did.

-----

---

It is highly recommended that you consider using the console-handler (CON:) if you want a console window with COPY and PASTE capabilities. CON: provides you with free PASTE support and is considerably easier to open and use than using the console device directly.

## 1.21 4 Console Device / Selecting Raw Input Events

If the keyboard information - including "cooked" keystrokes - does not give you enough information about input events, you can request additional information from the console driver.

The command to SET RAW EVENTS is formatted as:

```
<CSI>[event-types-separated-by-semicolons]{
```

If, for example, you need to know when each key is pressed and released, you would request "RAW keyboard input." This is done by writing "<CSI>1{" to the console. In a single SET RAW EVENTS request, you can ask the console to set up for multiple event types at one time. You must send multiple numeric parameters, separating them by semicolons (;). For example, to ask for gadget pressed, gadget released, and close gadget events, write:

```
<CSI>7;8;11{
```

You can reset, that is, delete from reporting, one or more of the raw input event types by using the RESET RAW EVENTS command, in the same manner as the SET RAW EVENTS was used to establish them in the first place. This command stream is formatted as:

```
<CSI>[event-types-separated-by-semicolons]}
```

So, for example, you could reset all of the events set in the above example by transmitting the command sequence:

```
<CSI>7;8;11}
```

The Read Stream May Not Be Dry.

-----  
There could still be pending RAW INPUT EVENTS in your read stream after turning off one or more RAW INPUT EVENTS.

The following table lists the valid raw input event types.

RAW INPUT EVENT TYPES			
Request Number	Description	Number	Request Description
0	No-op (used internally)	11	Close Gadget
1	RAW keyboard input *	12	Window resized
2	RAW mouse input	13	Window refreshed
3	Private Console Event	14	Preferences changed
4	Pointer position	15	Disk removed
5	(unused)	16	Disk inserted

6	Timer	17	Active window
7	Gadget pressed	18	Inactive window
8	Gadget released	19	New pointer position (V36)
9	Requester activity	20	Menu help (V36)
10	Menu numbers	21	Window changed (V36) (zoom, move)

\* Note: Intuition swallows all except the select button.

The event types-requester, window refreshed, active window, inactive window, window resized and window changed-are dispatched to the console unit which owns the window from which the events are generated, even if it is not the active (selected) window at the time the event is generated. This ensures that the proper console unit is notified of those events. All other events are dispatched to the active console unit (if it has requested those events).

## 1.22 4 Console Device / Input Event Reports

If you select any of these events you will start to get information about the events in the following form:

```
<CSI><class>;<subclass>;<keycode>;<qualifiers>;<x>;<y>;<secs>;<microsecs>|
```

<CSI>

is a one-byte field. It is the "control sequence introducer," 0x9B in hex.

<class>

is the RAW input event type, from the above table.

<subclass>

is usually 0. If the mouse is moved to the right controller, this would be 1.

<keycode>

indicates which raw key number was pressed. This field can also be used for mouse information.

The Raw Key Might Be The Wrong Key.

-----  
National keyboards often have different keyboard arrangements. This means that a particular raw key number may represent different characters on different national keyboards. The normal console read stream (as opposed to raw events) will contain the proper ASCII character for the keypress as translated according to the user's keymap.

<qualifiers>

indicates the state of the keyboard and system.

The qualifiers are defined as follows:

INPUT EVENT QUALIFIERS

Bit	Mask	Key
---	----	---
0	0001	Left shift
1	0002	Right shift
2	0004	Caps Lock Associated keycode is special; see below.
3	0008	Ctrl
4	0010	Left Alt
5	0020	Right Alt
6	0040	Left Amiga key pressed
7	0080	Right Amiga key pressed
8	0100	Numeric pad
9	0200	Repeat
10	0400	Interrupt Not currently used.
11	0800	Multibroadcast This window (active one) or all windows.
12	1000	Middle mouse button (Not available on standard mouse)
13	2000	Right mouse button
14	4000	Left mouse button
15	8000	Relative mouse Mouse coordinates are relative, not absolute.

The Caps Lock key is handled in a special manner. It generates a keycode only when it is pressed, not when it is released. However, the up/down bit (80 hex) is still used and reported. If pressing the Caps Lock key causes the LED to light, keycode 62 (Caps Lock pressed) is sent. If pressing the Caps Lock key extinguishes the LED, keycode 190 (Caps Lock released) is sent. In effect, the keyboard reports this key as held down until it is struck again.

The <x> and <y> fields are filled by some classes with an Intuition address:  $x \ll 16 + y$ .

The <seconds> and <microseconds> fields contain the system time stamp taken at the time the event occurred. These values are stored as longwords by the system.

With RAW keyboard input selected, keys will no longer return a simple one-character "A" to "Z" but will instead return raw keycode reports of the form:

```
<CSI>1;0;<keycode>;<qualifiers>;<prev1>;<prev2>;<seconds>;<microseconds>|
```

For example, if the user pressed and released the A key with the left Shift and right Amiga keys also pressed, you might receive the following data:

```
<CSI>1;0;32;32769;14593;5889;421939940;316673|
```

```
<CSI>1;0;160;32769;0;0;421939991;816683|
```

The <keycode> field is an ASCII decimal value representing the key pressed or released. Adding 128 to the pressed key code will result in the released keycode.

The <prev1> and <prev2> fields are relevant for the interpretation of keys which are modifiable by dead-keys (see "Dead-Class Keys" section). The <prev1> field shows the previous key pressed. The lower byte shows the qualifier, the upper byte shows the key code. The <prev2> field shows the key pressed before the previous key. The lower byte shows the qualifier,



the upper byte shows the key code.

## 1.23 4 Console Device / Using the Console Device Without a Window

Most console device processing involves a window, but there are functions and special commands that may be used without a window. To use the console device without a window, you call `OpenDevice()` with the console unit `CONU_LIBRARY`.

The console device functions are `CDInputHandler()` and `RawKeyConvert()`; they may only be used with the `CONU_LIBRARY` console unit. The console device commands which do not require a window are `CD_ASKDEFAULTKEYMAP` and `CD_SETDEFAULTKEYMAP`; they be used with any console unit. The advantage of using the commands with the `CONU_LIBRARY` unit is the lack of overhead required for `CONU_LIBRARY` because it doesn't require a window.

To use the functions requires the following steps:

- \* Declare the console device base address variable `ConsoleDevice` in the global data area.
- \* Declare storage for an I/O request of type `IOStdReq`.
- \* Open the console device with `CONU_LIBRARY` set as the console unit.
- \* Set the console device base address variable to point to the device library vector which is returned in `io_Device`.
- \* Call the console device function(s).
- \* Close the console device when you are finished.

```
#include <devices/conunit.h>
struct ConsoleDevice *ConsoleDevice; /* declare device base address */

struct IOStdReq ConsIO= {0};          /* I/O request */

main()

    /* Open the device with CONU_LIBRARY for function use */
    if (0 == OpenDevice("console.device",CONU_LIBRARY,
        (struct IORequest *)&ConsIO,0) )
    {
        /* Set the base address variable to the device library vector */
        ConsoleDevice = (struct ConsoleDevice *)ConsIO.io_Device;

        .
        .      (console device functions would be called here)
        .

        CloseDevice(ConsIO);
    }
```

The code fragment shows only the steps outlined above, it is not complete in any sense of the word. For a complete example of using a console device function, see the `rawkey.c` code example in the "Intuition: Mouse and Keyboard" chapter of the Amiga ROM Kernel Reference Manual: Libraries. The example uses the `RawKeyConvert()` function.

To use the commands with the `CONU_LIBRARY` console unit, you follow the same steps that were outlined in the Opening the Console Device section of this chapter.

```
struct MsgPort *ConsoleMP;          /* pointer to our message port */
struct IOStdReq *ConsoleIO;         /* pointer to our I/O request */
struct KeyMap *keymap;              /* pointer to keymap */

/* Create the message port */
if (ConsoleMP=CreateMsgPort())
{
    /* Create the I/O request */
    if (ConsoleIO = CreateIORequest(ConsoleMP,sizeof(struct IOStdReq))
        {
            /* Open the Console device */
            if (OpenDevice("console.device",CONU_LIBRARY,
                          (struct IORequest *)ConsoleIO,0L))

                /* Inform user that it could not be opened */
                printf("Error: console.device did not open\n");
            else
            {
                /* Allocate memory for the keymap */
                if (keymap = (struct KeyMap *)
                    AllocMem(sizeof(struct KeyMap),MEMF_PUBLIC | MEMF_CLEAR))
                {
                    /* device opened, send CD_ASKKEYMAP command to it */
                    ConsoleIO->io_Length = sizeof(struct KeyMap);
                    ConsoleIO->io_Data = (APTR)keymap; /* where to put it */
                    ConsoleIO->io_Command = CD_ASKKEYMAP;
                    DoIO((struct IORequest *)ConsoleIO)
                }

                CloseDevice(ConsoleIO);
            }
        }
    }
}
```

Again, as in the previous code fragment, this is not complete (that's why it's a fragment!) and you should only use it as a guide.

## 1.24 4 Console Device / Where Is All the Keymap Information?

Unlike previous editions of this chapter, this one has a very small amount of keymap information. Keymap information is now contained, appropriately enough, in the "Keymap Library" chapter of the Amiga ROM Kernel Reference Manual: Libraries.

## 1.25 4 Console Device / Console Device Caveats

- \* Only one console unit can be attached per window. Sharing a console window must be done at a level higher than the device.
- \* Do not mix graphics.library calls with console rendering in the same areas of a window. It is permissible to send console sequences to adjust the area in which console renders, and use graphics.library calls to render outside of the area console is using.

For example, do not render text with console sequences and scroll using the graphics.library ScrollRaster() function.

- \* The character map feature is private and cannot be accessed by the programmer. Implementation details and behaviors of the character map may change in the future.
- \* Do not use an IDCMP with character mapped consoles. All Intuition messages should be obtained via RAW INPUT EVENTS from the console device.

## 1.26 4 Console Device / Additional Information on the Console Device

Additional programming information on the console device can be found in the include files and the Autodocs for the console device. Both are contained in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

Console Device Information	
-----	
INCLUDES	devices/console.h devices/console.i devices/conunit.h devices/conunit.h
AUTODOCS	console.doc