Devices

COLLABORATORS			
	TITLE : Devices		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Devices

1.1 Amiga® RKM Devices: 7 Keyboard Device

The keyboard device gives low-level access to the Amiga keyboard. When you send this device the command to read one or more keystrokes from the keyboard, for each keystroke (whether key-up or key-down) the keyboard device creates a data structure called an input event to describe what happened. The keyboard device also provides the ability to do operations within the system reset processing (Ctrl-Amiga-Amiga).

1.2 7 Keyboard Device / Keyboard Device Commands and Functions

Command	Operation
CMD_CLEAR	Clear the keyboard input buffer. Removes any key transitions from the input buffer.
KBD_ADDRESETHANDLER	Add a reset handler function to the list of functions called by the keyboard device to clean up before a hard reset.
KBD_REMRESETHANDLER	Remove a previously added reset handler from the list of functions called by the keyboard device to clean up before a hard reset.
KBD_RESETHANDLERDONE	Indicate that a handler has completed its job and reset could possibly occur now.
KBD_READMATRIX	Read the state of every key in the keyboard. Tells the up/down state of every key.

KBD_READEVENT	Read one (or more) raw key event from the keyboard device.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the keyboard device.
AllocMem()	Allocate a block of memory.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the keyboard device.
DoIO()	Initiate a command and wait for it to complete (synchronous request).
FreeMem()	Free a block of previously allocated memory.
OpenDevice()	Obtain use of the keyboard device.
SendIO()	Initiate a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete the message will be removed from reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure. This structure will be used to communicate commands to the keyboard device.
CreatePort()	Create a signal message port for reply messages from the keyboard device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an extended I/O request structure created by CreateExtIO().
DeletePort()	Delete the message port created by CreatePort().

1.3 7 Keyboard Device / Device Interface

The keyboard device operates like the other Amiga devices. To use it, you must first open the keyboard 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 keyboard 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.

1.4 7 / Device Interface / Opening The Keyboard Device

Three primary steps are required to open the keyboard device:

* Create a message port using the CreatePort() function.

* Create an extended I/O request structure using the CreateExtIO() function. CreateExtIO() will initialize the I/O request with your reply port.

* Open the keyboard device. Call OpenDevice(), passing the I/O request.

```
struct MsgPort  *KeyMP;         /* Pointer for Message Port */
struct IOStdReq *KeyIO;         /* Pointer for I/O request */

if (KeyMP=CreatePort(NULL,NULL))
  if (KeyIO=(struct IOStdReq *)CreateExtIO(KeyMP,sizeof(struct IOStdReq)))
    if (OpenDevice( "keyboard.device",NULL,(struct IORequest *)KeyIO,NULL))
        printf("keyboard.device did not open\n");
```

1.5 7 / Device Interface / Closing The Keyboard Device

An 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() and remove them with WaitIO().

```
   if (!(CheckIO(KeyIO)))
       {
       AbortIO(KeyIO);  /* Ask device to abort request, if pending */
       }
       WaitIO(KeyIO);   /* Wait for abort, then clean up */
   CloseDevice(KeyIO);
```

1.6 7 Keyboard Device / Reading the Keyboard Matrix

The KBD_READMATRIX command returns the current state of every key in the key matrix (up = 0, down = 1). You provide a data area that is at least large enough to hold one bit per key, approximately 16 bytes. The keyboard layout for the A500, A2000 and A3000 is shown in the figure below, indicating the raw numeric value that each key transmits when it is pressed. This value is the numeric position that the key occupies in the key matrix.

Figure 7-1 - Keyboard Layout

The following example will read the key matrix and display the up-down state of all of the elements in the matrix in a table. Reading the column header and then the row number as a hex number gives you the raw key code.

Read_Keyboard_Matrix.c

In addition to the matrix data returned in io_Data, io_Actual returns the number of bytes filled in io_Data with key matrix data, i.e., the minimum of the supplied length and the internal key matrix size.

Value of io_Length.

A value of 13 in the io_Length field will be sufficient for most keyboards; extended keyboards will require a larger number. However, you must always set this field to 13 for V34 and earlier versions of Kickstart.

To find the status of a particular key - for example, to find out if the F2 key is down - you find the bit that specifies the current state by dividing the key matrix value by 8. Since hex 51 = 81, this indicates that the bit is in byte number 10 of the matrix. Then take the same number (decimal 81) and use modulo 8 to determine which bit position within that byte represents the state of the key. This yields a value of 1. So, by reading bit position 1 of byte number 10, you determine the status of the function key F2.

1.7 7 Keyboard Device / Amiga Reset Handling

When a user presses the Ctrl key and both left- and right-Amiga keys simulataneously (the reset sequence), the keyboard device senses this and calls a prioritized chain of reset-handlers. These might be thought of as clean-up routines that "must" be performed before reset is allowed to occur. For example, if a disk write is in progress, the system should finish that before resetting the hardware so as not to corrupt the contents of the disk.

It is important to note that not all Amigas handle reset processing in the same way. On the A500, the reset key sequence sends a hardware reset signal and never goes through the reset handlers. Also some of the early A2000s (i.e., German keyboards with the function keys the same size as the Esc key) do not handle the reset via the reset handlers. It is thus

recommended that your application not rely on the reset handler abilities of the keyboard device.

Adding A Reset Handler (KBD_ADDRESETHANDLER)
Removing A Reset Handler (KBD_REMRESETHANDLER)
Ending A Reset Task (KBD_RESETHANDLERDONE)

1.8 7 / Amiga Reset Handling / Adding A Reset Handler (KBD_ADDRESETHANDLER)

The KBD_ADDRESETHANDLER command adds a custom routine to the chain of reset-handlers. Reset handlers are just like any other handler and are added to the handler list with an Interrupt structure. The priority field in the list node of the Interrupt structure establishes the sequence in which reset handlers are processed by the system. Keyboard reset handlers are currently limited to the priority values of a software interrupt, that is, values of -32, -16, 0, 16, and 32.

The io_Data field of the I/O request is filled in with a pointer to the Interrupt structure and the io_Command field is set to KBD_ADDRESETHANDLER. These are the only two fields you need to initialize to add a reset handler. Any return value from the command is ignored. All keyboard reset handlers are activated if time permits. Normally, a reset handler will just signal the requisite task and return. The task then does whatever processing it needs to do and notifies the system that it is done by using the KBD_RESETHANDLERDONE command described below.

Non-interference and speed are the keys to success.

If you add your own handler to the chain, you must ensure that your handler allows the rest of reset processing to occur. Reset must continue to function. Also, if you don't execute your reset code fast enough, the system will still reboot (about 10 seconds).

1.9 7 / Amiga Reset Handling / Removing A Reset Handler (KBD_REMRESETHANDLER)

This command is used to remove a keyboard reset handler from the system. You need to supply the same Interrupt structure to this command that you used with the KBD_ADDRESETHANDLER command.

1.10 7 / Amiga Reset Handling / Ending A Reset Task (KBD_RESETHANDLERDONE)

This command tells the system that your reset handling code has completed. If you are the last outstanding reset handler, the system will reset after this call.

Can't Stop, Got No Brakes.

After 10 seconds, the system will reboot, regardless of outstanding reset handlers.

Here is an example program that installs a reset handler and either waits for the reboot or for the user to close the window. If there was a reboot, the window will close and, if executed from the shell, it will display a few messages. If the user closes the window, the handler is removed and the program exits cleanly.

Key_Reset.c

1.11 7 Keyboard Device / Reading Keyboard Events

Reading keyboard events is normally not done through direct access to the keyboard device. (Higher level devices such as the input device and console device are available for this. See the chapter Input Device for more information on the intimate linkage between the input device and the keyboard device.) This section is provided primarily to show you the component parts of a keyboard input event.

The keyboard matrix figure shown at the beginning of this chapter gives the code value that each key places into the ie_Code field of the input event for a key-down event. For a key-up event, a value of hexadecimal 80 is or'ed with the value shown above. Additionally, if either shift key is down, or if the key is one of those in the numeric keypad, the qualifier field of the keyboard input event will be filled in accordingly. In V34 and earlier versions of Kickstart, the keyboard device does not set the numeric qualifier for the keypad keys "(", ")", "/", "*" and "+".

When you ask to read events from the keyboard, the call will not be satisfied until at least one keyboard event is available to be returned. The io_Length field must contain the number of bytes available in io_Data to insert events into. Thus, you should use a multiple of the number of bytes in an InputEvent (see example below).

Type-Ahead Processing.

The keyboard device can queue up several keystrokes without a task requesting a report of keyboard events. However, when the keyboard event buffer has been filled with no task interaction, additional keystrokes will be discarded.

Example Read Keyboard Event Program

1.12 7 Keyboard Device / Additional Information on the Keyboard Device

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

Keyboard Device Information

INCLUDES	devices/keyboard.h
	devices/keyboard.i

```
                devices/inputevent.h
                devices/inputevent.i

AUTODOCS        keyboard.doc
```