This chapter describes the ADB Manager, the part of the Macintosh Operating System that allows you to get information about and communicate with hardware devices attached to the Apple Desktop Bus (ADB). On most Macintosh computers, the ADB is used to communicate with the keyboard, the mouse, and other user-input devices.

The Macintosh Operating System contains standard keyboard and mouse handling routines that automatically take care of all required ADB access operations. Applications typically receive keyboard and mouse input by calling the Event Manager, not by calling the ADB Manager. For complete information about receiving and interpreting keyboard and mouse input, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter begins with an overview of the Apple Desktop Bus and the ADB Manager. It also shows how to

- get information about devices attached to the ADB
- communicate with devices on the ADB at a very low level
- write a device handler for a new user-input device that connects to the ADB

For detailed information about the ADB hardware, see *Guide to the Macintosh Family Hardware*, second edition.

About the Apple Desktop Bus

The ***Apple Desktop Bus*** is a low-speed serial bus that connects input devices, such as keyboards, mouse devices, and graphics tablets, to a Macintosh computer or to other hardware equipment. For information on the number of devices that you can connect to the ADB, see *Guide to the Macintosh Family Hardware*, second edition. Macintosh computers come equipped with one or two ADB connectors. Although a particular model might include two ADB connectors, all models come with only one Apple Desktop Bus.

The ADB is Apple Computer's standard interface for input devices such as keyboards and mouse devices. Apple provides a mouse with each Macintosh computer, except for models equipped with a trackball. Additionally, Apple provides various ADB keyboard options, such as the Apple Standard keyboard, the Apple Extended keyboard, and the Apple Adjustable keyboard.

Characteristics of ADB Devices

An ***ADB device*** is any input device that can connect to the ADB and meets the design requirements described in the *Apple Desktop Bus Specification*.

IMPORTANT

Apple Computer, Inc. owns patents on the Apple Desktop Bus (ADB). If you want to manufacture a device that works with the ADB software, you must obtain a license and device handler ID from Apple Computer, Inc. Write to this address:

Apple Software Licensing
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014

A license includes a copy of the *Apple Desktop Bus Specification*. ▲

An ADB device generally communicates with the Macintosh Operating System through a ***device handler*** —a set of low level routines designed to interact with a specific ADB device. The Macintosh Operating System already includes device handlers for Apple-supplied keyboards and mouse devices. You need to write your own device handler and the code that installs it only if you are designing your own ADB device. For more information on writing and installing an ADB device handler, see "Writing an ADB Device Handler" on page 5-29.

A properly designed ADB device has the following features:

- the memory in which to store data
- a default ADB device address and device handler ID
- the ability to detect and respond to bus collisions
- the ability to assert a service request signal

Each ADB device may contain up to four device registers, which you can read from or write to using certain ADB commands. One of these device registers stores the device's default ADB device address and device handler ID, both provided by Apple Software Licensing.

Each ADB device has a default address and initially responds to all ADB commands at that address. A ***default ADB device address*** is a 4-bit bus address that uniquely identifies the general type of device (such as a mouse or keyboard). An ***ADB device handler ID*** (or device handler identification) is an 8-bit value that identifies a more specific classification of the device type (such as the Apple Extended keyboard) or specific mode of operation (such as whether the keyboard differentiates between the Right and Left Shift keys). For more information on both these items, see "Default ADB Device Address and Device Handler Identification" on page 5-11.

To avoid collision of multiple ADB devices over the bus, an ADB device must be able to detect when another ADB device is transmitting data at the same time. For more information on collision detection, see "Address Resolution," beginning on page page 5-15.

An ADB device cannot initiate a data transaction. It must therefore be able to inform the ADB Manager that it needs to transmit new data by asserting a service request signal. (In addition, the ADB Manager continually queries ADB devices to see if they have data to

send.) For more details on service request signals, see "ADB Communication," beginning on page page 5-17.

About the ADB Manager

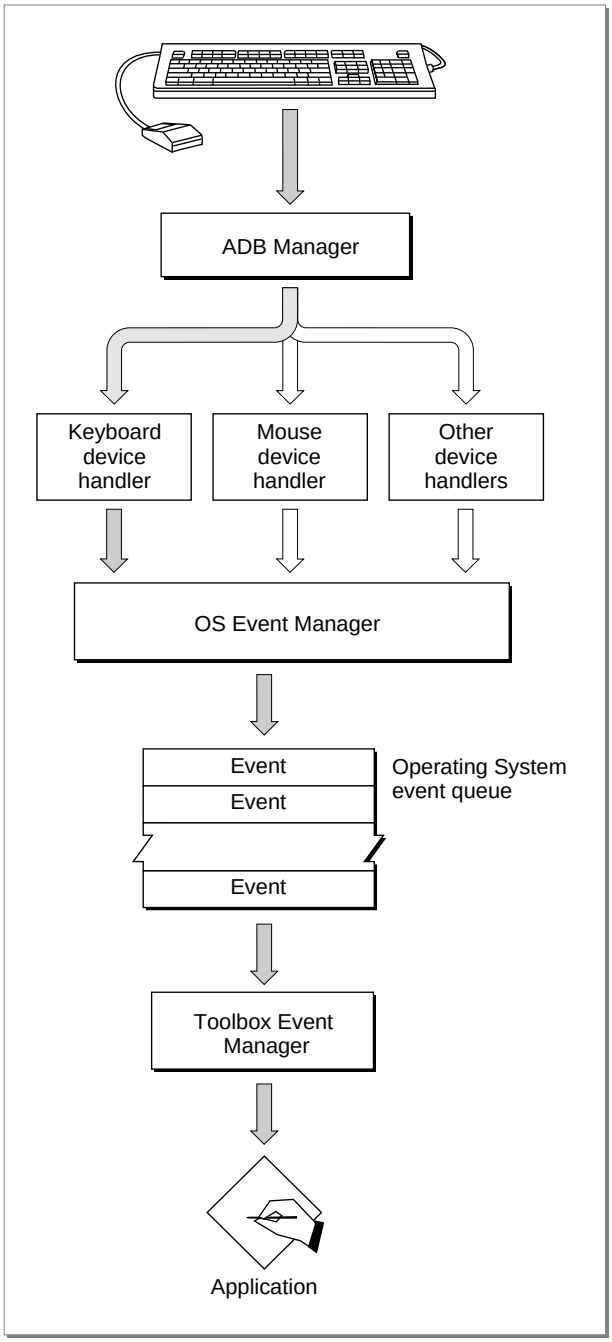
The ADB Manager is the part of the Macintosh Operating System that allows you to get information about and communicate with hardware devices attached to the Apple Desktop Bus. Most applications never need to interact with the ADB Manager, but can instead call the appropriate Event Manager routines for information about user actions on ADB devices such as keyboards or mouse devices. Also note that the ADB Manager does not interact with the Device Manager, but handles all ADB devices and ADB device handlers itself.

The ADB Manager handles three main tasks. First, at system startup, the ADB Manager builds the ***ADB device table*** , which contains the default ADB device address, device handler ID, and other identifying information for each ADB device. Whenever the ADB is reinitialized, the ADB Manager reinitializes the ADB device table. Second, if two or more ADB devices share the same default ADB device address when the ADB is building the ADB device table or when the ADB is reinitialized, the ADB Manager assigns each device a new ADB device address until no address conflict exists. This process is known as ***address resolution*** (see page 5-15 for more information). Third, the ADB Manager retrieves new data from the ADB devices and sends it to the appropriate device handler.

In general, ADB devices communicate with the Operating System only through the ADB Manager. The ADB Manager, in turn, calls a device handler to process data from the device. The device handler interprets data transmitted by the ADB device, and in some cases, passes this information to the Event Manager. A single device handler can manage more than one device of the same type (for example, the device handler for the Apple Extended keyboard can manage several keyboards). A single device handler can also manage more than one device type if the different device type emulates the device type associated with the particular device handler (for example, a mouse device handler can manage both a mouse and a graphics tablet emulating a mouse).

A device handler receives all data from its associated ADB device through the ADB Manager. The ADB Manager continually checks to see if ADB devices have new data to send. When the ADB Manager receives new data from an ADB device, it sends the data to the appropriate device handler. The device handler interprets the data and, if appropriate, places an event into the event queue using the `PostEvent` function. (For more information on `PostEvent`, see *Inside Macintosh: Macintosh Toolbox Essentials*.) For example, if the user types a key on the keyboard, the ADB Manager retrieves this data and sends the data to the device handler for the keyboard, which in turn places an event into the event queue. Figure 5-1 shows the relationship between the ADB Manager, device handlers, and the Event Manager.

Figure 5-1 The ADB Manager and device handlers



The ADB Manager retrieves data from an ADB device as a result of its normal polling process. The ADB Manager polls a device by sending it a command requesting it to return the contents of one of its registers. (Note that an ADB device should respond to the specific ADB command, Talk Register 0, only if the device has new data to send. See the next section, "ADB Commands," for more information.)

In general, the ADB Manager repeatedly polls the last ADB device that sent new data except under two circumstances: when it receives a service request signal, and when it builds the ADB device table. In these cases, the ADB Manager also polls other ADB devices. When responding to a service request signal, the ADB Manager polls all known addresses containing an ADB device until all pending data is transmitted and no device asserts a service request signal. When building the ADB device table, the ADB Manager polls each ADB device connected to the bus. For more information on the ADB device table, see "ADB Device Table" on page 5-13.

In general, only device handlers use the ADB Manager to communicate with devices. The normal polling of ADB devices performed by the ADB Manager retrieves data for the device handlers; your application should call the appropriate Event Manager routines for information about the user's input on ADB devices. If necessary, however, you can directly communicate with an ADB device using the `ADBOp` routine. You should use the `ADBOp` routine only for special purposes where you need to directly communicate with an ADB device (for example, to set the LED lights on an Apple Extended keyboard). Remember that in most circumstances, you do not need to call `ADBOp`.

ADB Commands

An ***ADB command*** is a 1-byte value that specifies the ADB device address of a device and encodes the desired action the target device should perform. In some cases, additional data may follow an ADB command. For example, the ADB Manager may transmit data to the device or the device may respond to a command by transmitting one or more bytes of data back to the ADB Manager. It's important to realize, however, that ADB devices never issue commands to the ADB Manager. At most, the device can assert a service request signal to request that the ADB Manager poll the bus for any devices wishing to transmit data. For more information on how ADB devices communicate with the ADB Manager, see "ADB Communication,"beginning on page page 5-17.

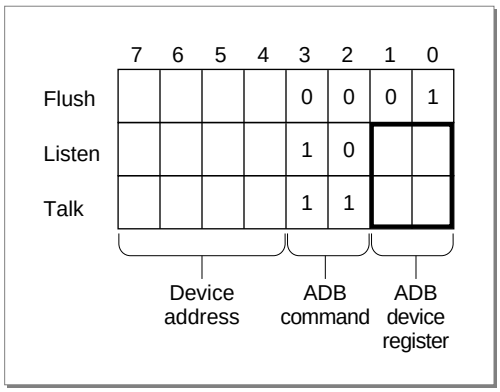
The ADB Manager can send any of four bus commands to an ADB device. Three of these commands, Talk, Listen, and Flush, are addressed to specific registers on a specific device. For more information on these registers, see "ADB Device Registers" on page 5-9. The fourth command, SendReset, applies to all ADB devices.

- ***Talk.*** The ADB Manager sends a Talk command to a device to fetch user input (or other data) from the device. The Talk command requests that a specified device send the contents of a specified device register across the bus. After the device sends the data from the specified register, the ADB Manager places the data into a buffer in RAM, which the ADB Manager makes available for use by device handlers or (in rare cases) applications. In the case of a Talk Register 0 command, the ADB device should respond to the ADB Manager only if it has new data to send.

- ***Listen.*** The ADB Manager sends a Listen command to a device to instruct it to prepare to receive additional data. The Listen command indicates which data register is to receive the data. After sending a Listen command, the ADB Manager then transfers data from a buffer in RAM to the device. The device must overwrite the existing contents of the specified register with the new data.
- ***Flush.*** The ADB Manager sends a Flush command to a device to force it to flush any existing user-input data from a specified device register. The device should prepare itself to receive any further input from the user.
- ***SendReset.*** The ADB Manager uses a SendReset command to force all devices on the bus to reset themselves to their startup states. Each device should clear any of its pending device actions and prepare to accept new ADB commands and user input data immediately. Note that the ADB device does not actually receive the SendReset command but recognizes that it should reset itself when the bus is driven low by the 3 millisecond reset pulse. Your application should never send the SendReset command.

Figure 5-2 shows the command formats for the Talk, Listen, and Flush commands.

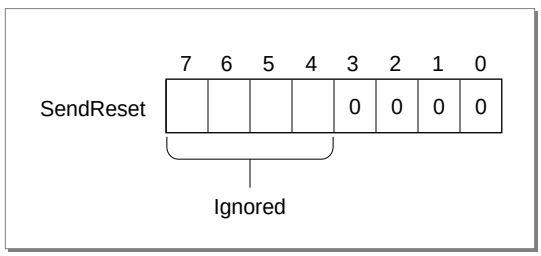
Figure 5-2 Command formats for Talk, Listen, and Flush



Bits 0 through 1 specify the ADB device register, bits 2 through 3 specify the command code, and bits 4 through 7 specify the device address.

Figure 5-3 shows the command format for the SendReset command.

Figure 5-3 Command format for SendReset



The first four bits of the SendReset command identify this command. Because the SendReset command applies to all ADB devices, bits 4 through 7 do not specify the address of a particular device. As previously described, an ADB device never receives a SendReset command; instead, the device resets itself in response to the 3 millisecond pulse.

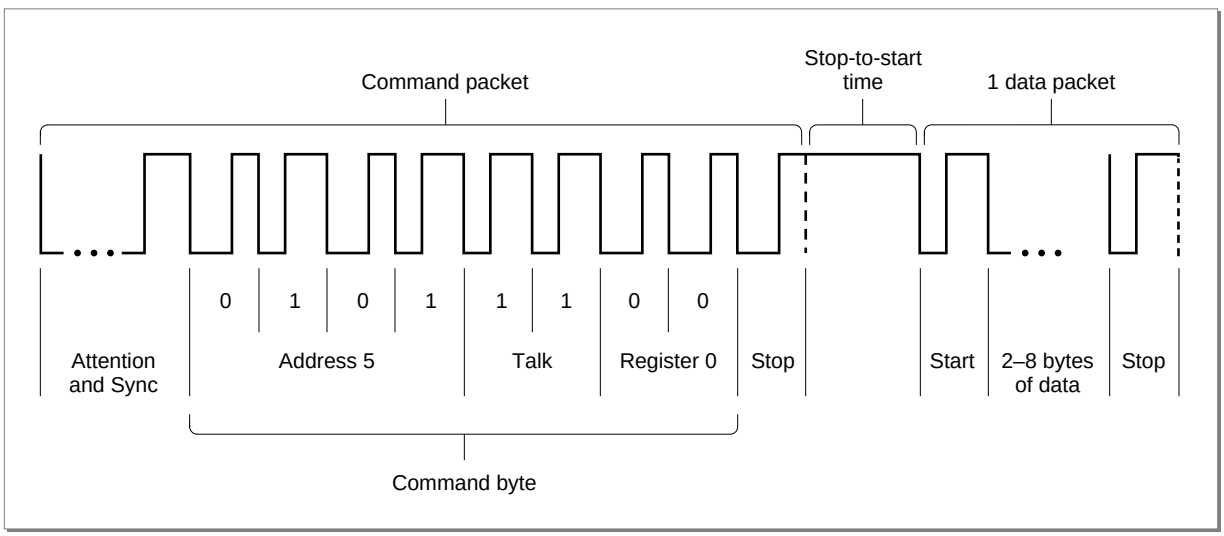
ADB Transactions

An ***ADB transaction*** is a bus communication between the computer and an ADB device. A transaction consists of a command sent by the computer, followed by a data packet of several bytes sent either by the computer or a device. An ADB command consists of four parts:

- an Attention signal
- a Sync signal
- one command byte
- one stop bit

Figure 5-4 shows a typical ADB transaction, consisting of a command followed by a data packet.

Figure 5-4 A typical ADB transaction



ADB Device Registers

Each device connected to the Apple Desktop Bus may provide up to four registers for storing data. These registers are referred to as ***ADB device registers*** . An ADB device can implement these registers as it chooses; that is, an ADB register does not have to correspond to an actual hardware register on the ADB device. An ADB device is accessed

over the ADB by reading from or writing to these registers. Each ADB device register may store between 2 and 8 bytes of data.

The ADB device registers are numbered 0 through 3. Register 0 and register 3 are defined according to the specifications described in the next two sections. Register 1 and register 2 are device-dependent and can be defined by a device for any purpose.

Register 0

For most devices, register 0 is used to hold data that needs to be fetched by the Macintosh Operating System. For example, register 0 of the Apple Standard keyboard contains information about the key pressed by the user.

The ADB Manager polls all ADB devices to determine which one asserted a service request signal by sending a Talk Register 0 command to each device in turn. A device should respond to a Talk Register 0 command only if it has new data to send. For more information about polling, see "ADB Communication," beginning on page 5-17.

Table 5-1 shows the bits of register 0 as defined by the Apple Standard keyboard. Note that these bits represent key transition codes (also called raw key codes). For examples of the bits of register 0 used for the Apple standard mouse and the Apple Extended keyboard, see *Guide to the Macintosh Family Hardware*, second edition.

Table 5-1 Register 0 in the Apple Standard keyboard

Bit	Description
15	Key status for first key; 0 = down
14–8	Key transition code for first key; a 7-bit ASCII value
7	Key status for second key; 0 = down
6–0	Key transition code for second key; a 7-bit ASCII value

Register 3

The bits in register 3 are defined by the ADB Manager. Figure 5-5 shows the defined bits for register 3, which include the default ADB device address, the device handler ID, a service request enable field, an exceptional event field, and several reserved bits.

Figure 5-5 Format of device register 3

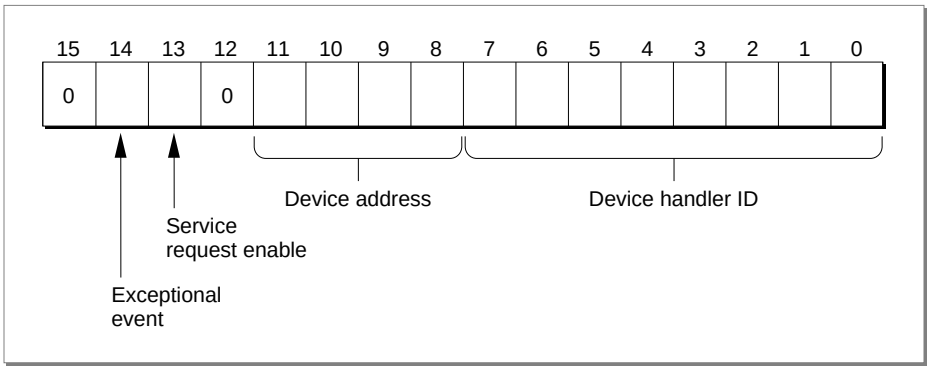


Table 5-2 provides a description of each bit in register 3.

Table 5-2 Bits in device register 3

Bit	Description
15	Reserved; must be 0
14	Exceptional event, device specific; always 1 if not used
13	Service request enable; 1 = enabled
12	Reserved; must be 0
11–8	ADB device address
7–0	Device handler ID

The functions of some of the bits in register 3 are discussed in detail in this chapter. For information on service request signals, "ADB Communication," beginning on page 5-17. For information on the default ADB device address and device handler ID, see the next section.

Default ADB Device Address and Device Handler Identification

As previously described, each ADB device has a default ADB device address and device handler identification (or device handler ID). Together, the default ADB device address and device handler ID identify the general type of device (such as a mouse or keyboard) as well as a more specific classification of the device type (such as the Apple Extended keyboard) or specific mode of operation (such as whether the keyboard differentiates between the Right and Left Shift keys).

A default ADB device address is a 4-bit bus address that uniquely identifies devices of the same type. The currently defined default ADB device addresses have values between 1 and 7. Table 5-3 shows the defined default ADB device addresses and their device type categories. Though it is not mandatory that an ADB device's default address define the

device type, doing so significantly reduces the possibility of multiple devices on the ADB sharing the same default address. Most device default addresses are movable addresses, which means that they can be replaced with a new address. If two ADB devices have the same default address, the ADB Manager must move one of the devices to a new address. An example of this process is described in detail in "Address Resolution," beginning on page 5-15.

Table 5-3 Defined default ADB device addresses

Default address	Device type	Example
$1	Protection devices	Software execution control devices
$2	Encoded devices	Keyboards
$3	Relative-position devices	Mouse devices
$4	Absolute-position devices	Tablets
$5	Data transfer devices	Low-speed ADB modems
$6	Any other	Reserved
$7	Any other	Appliances/miscellaneous

Note
The default address $0 is reserved for the Macintosh computer. Addresses $8 through $E are reserved by the ADB Manager for dynamically relocating devices to resolve address collision. ◆

The ADB device handler ID is an 8-bit value that further identifies the specific device type or its mode of operation. For example, an Apple Standard keyboard has a device handler ID of 1, while an Apple Extended keyboard has a device handler ID of 2.

An ADB device can support several device handler IDs and change its mode of operation according to its current device handler ID. The Apple Extended keyboard, for example, supports two device handler IDs: $02 and $03. The Apple Extended keyboard uses $02 as a device handler ID by default. When its device handler ID is changed to $03, the Apple Extended keyboard sends separate key codes for the Left and Right Shift keys. A device handler, application, or the ADB Manager can request a device to change its device handler ID by sending it a Listen Register 3 command. If a device accepts a new device handler ID, it sends that device handler ID in response to any subsequent Listen Register 3 command. An ADB device should respond to a request to change its device handler ID only if it recognizes the device handler ID; otherwise, it should ignore the request and continue to send its default device handler ID in response to a Listen Register 3 command. For example, if the Apple Extended keyboard is requested to change its device handler ID to $52, the keyboard ignores this request. When an ADB device handler changes its device handler ID anytime after the ADB Manager sets initial values for that device in the ADB device table (that is, after initial address resolution is complete), the ADB Manager does not update the device's entry in the ADB device table.

Apple reserves certain device handler IDs for special purposes, as shown in Table 5-4. ADB devices must recognize and respond appropriately to these special device handler IDs. When a device receives a Listen Register 3 command containing a special device handler ID, the device should immediately perform the specified action. Note, however, that the device should not change its device handler ID to the special device handler ID specified by the Listen Register 3 command.

Table 5-4 Special device handler IDs

ID value	Description
$FF	Instructs the device to initiate a self-test.
$FE	Instructs the device to change its ADB device address (as stored in bits 8–11 of register 3) to the new address set in the command if no collision has been detected.
$FD	Instructs the device to change its ADB device address (as stored in bits 8–11 of register 3) to the new address set in the command if the activator is pressed. (See *Guide to the Macintosh Family Hardware,* second edition, for complete details on activators.)
$00	Instructs the device to change its ADB device address (as stored in bits 8–11 of register 3) and enable bit (bit 13) to the new values set in the command.

Note
The special device handler ID $00 can also be returned by a device that fails a self-test. ◆

ADB Device Table

The ADB Manager creates the ADB device table and places it in the system heap during system startup. The ADB Manager also reinitializes the ADB device table whenever the ADB is reinitialized (as a result of a call to the `ADBReinit` procedure, for example). For each ADB device, the ADB device table contains an ***ADB device table entry*** . The device table entry specifies the device's handler ID, default ADB device address and current ADB address, as well as the address of the device handler and the address of the area in RAM used for global storage by the handler. For information on the address ADB device and device handler ID, see "Default ADB Device Address and Device Handler Identification" on page 5-11. For information on device handlers, see "Writing an ADB Device Handler" on page 5-29.

Once the ADB Manager has set the initial values for an ADB device in the ADB device table, thereafter it updates the device table entry only to reflect changes to a device's device handler routine and data area pointer. If an ADB device changes its device handler ID, the ADB Manager does not update the ADB device table to reflect this change. To find out the new device handler ID for a device, you must send the device a Talk Register 3 command.

The ADB device table is accessible only through the ADB Manager routines `GetIndADB`, `GetADBInfo`, and `SetADBInfo`. The `GetIndADB` and `GetADBInfo` routines return information from the device table in an ADB data block, defined by the `ADBDataBlock` data type. These routines are described in detail later in this chapter.

At system startup, the ADB Manager sends a Talk Register 3 command to each device to retrieve its default ADB device address and device handler ID. For an Apple ADB device, the ADB Manager immediately places in the device table the address of the appropriate device handler provided by Apple for that device. Each nonstandard device, however, requires its own handler installation code to place the address of its device handler in the table. For information on installing a device handler, see "Installing an ADB Device Handler,"beginning on page 5-30.

If more than one ADB device has the same default ADB device address, the ADB Manager performs address resolution. For more information, see "Address Resolution," beginning on page 5-15.

Table 5-5 shows an example of an ADB device table after all ADB devices have responded to polling and have been assigned unique ADB device addresses by the ADB Manager. This example shows just one way that address resolution might occur.

Table 5-5 Typical ADB device table at initialization

Index	Device handler ID	Current address	Default address	Address of device handler routine	Address of handler's data area
$1	$01	$2	$2 (keyboard)	$4080AB46	$5450
$2	$01	$3	$3 (mouse)	$4080AAE6	$0000
$3	$02	$E	$2 (keyboard)	$4080AB46	$548C
$4	$02	$D	$2 (keyboard)	$4080AB46	$548C
$5	$00	$0	$0	$0	$0
$6	$00	$0	$0	$0	$0
$F	$00	$0	$0	$0	$0

The leftmost column shows the device table index. In this example, four devices are connected to the ADB: three keyboards and a mouse. The keyboard at index $1 has a device handler ID of $01, specifying that it is an Apple Standard keyboard. The remaining two keyboards at index $3 and index $4 each have a device handler ID of $02, specifying that they are both Apple Extended keyboards. Because they are the same type of device, all three keyboards have a default ADB device address of $2. Each ADB device must have a unique ADB device address. The ADB Manager therefore performs address resolution by assigning each Apple Extended keyboard a new and unoccupied ADB address. See "Address Resolution," beginning on page 5-15, for complete details on address resolution.

Although the ADB Manager assigns each keyboard a unique current address, note that all three keyboards use the same device handler, which in this example is located at address $4080AB46. The device handler, however, stores data for the two keyboard types in different areas in RAM. In this example, the address of the data area for the two Apple Extended keyboards is at $548C, compared to the address of the data area for the Apple Standard keyboard located at $5450.

In contrast, the mouse at index $2 is the only ADB device of its type and therefore has the same default and current address. Also, the mouse uses a different device handler than the keyboards use, which in this example is located at address $4080AAE6. Finally, the mouse device handler does not need to use area in RAM for storage. As a result, the value for its data area is $0000.

Address Resolution

Each ADB device has a default ADB device address and initially responds to all ADB commands at that address. If two or more ADB devices respond to commands sent to a particular address, this is referred to as ***address collision***. Due to the design of ADB devices and the ability of the ADB Manager to perform address resolution, most address collision occurs only at initial startup or when you reset the ADB. Furthermore, once the ADB Manager reassigns those addresses in conflict, subsequent address collision is quite rare.

Collision detection is the ability of an ADB device to detect that another ADB device is transmitting data at the same time. An ADB device should be able to detect a bus collision if it is bringing the bus high when another device is bringing the bus low. Whenever an ADB device attempts to bring the bus high, it should verify whether the bus actually goes high. If the bus instead goes low, this indicates that another device is also trying to send data. The device detecting the collision must immediately stop transmitting and save the data it was sending. Because the detecting device is no longer transmitting data, the device driving the bus low is not able to detect the other device. As a result, only one of the two colliding devices—the device driving the bus high—actually detects the collision.

When the ADB Manager performs address resolution, it reassigns default ADB device addresses so that all devices have a unique address. The new address locations are always between $8 through $E. Because these locations are dynamic, there is no way to predict the order in which the ADB Manager assigns new addresses to ADB devices or the exact address that it assigns to a given device. For the ADB Manager to accomplish address resolution, an ADB device must meet two design requirements: first, it must have collision detection, and second, it must always respond to a Talk Register 3 command by returning a random device address in bits 8 through 11.

A random device address is a four-bit value; an ADB device must return a random device address to the ADB Manager in response to a Talk Register 3 command. An ADB device is designed to respond only to a Talk Register 3 command that is specifically addressed to it. Because the address of an ADB device is already confirmed by its ability to respond to the Talk Register 3 command, the device does not need to provide its ADB

device address to identify itself. The ability of devices to send random addresses plays a crucial role in collision detection.

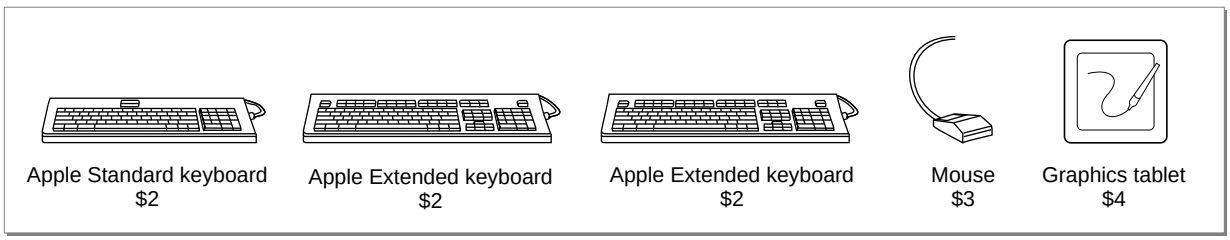
At system startup, the ADB Manager polls all ADB devices at each ADB address and begins the process of building the ADB device table by sending a Talk Register 3 command to each device. Each ADB device at a specific address attempts to respond by sending a random device address. If more than one ADB device shares an address, however, each device that detects a collision immediately stops transmitting data. The device that has not detected the collision completes sending its random address across the bus.

In response, the ADB Manager sends to the original address a Listen Register 3 command that contains a new ADB device address and a device handler ID of $FE. A new ADB device address is always a value between $8 and $E. A device handler ID of $FE instructs a device to change to the new device address only if it does not detect a collision. Any detecting devices will therefore ignore the next Listen Register 3 command containing a new ADB device address. As a result, only the device that did not detect the collision moves to the new address; the detecting devices remain at the original address. The ADB Manager now sends another Talk Register 3 command to the new address to verify that the device moved to that location. In response, the moved device must once again return a random address.

The ADB Manager repeats this process until it receives no response when it sends a Talk Register 3 command to the shared address. This indicates that no devices reside at the address and that it is an available address location for a device. The ADB Manager then moves the first device it relocated to a new address back to its original address.

Figure 5-6 shows three keyboards, a mouse, and a graphics tablet. In this example, assume these ADB devices are all connected to an ADB. This example describes one possible order and method that the ADB Manager might use to relocate ADB devices. Remember, however, that the specific implementation of address resolution is private to the ADB Manager.

Figure 5-6 Resolving address conflicts



In the example shown in Figure 5-6, all three keyboards are the same device type; thus, they share the same default ADB device address ($2). When the ADB Manager begins to build the device table by sending a Talk Register 3 command to address $2, all three

keyboards attempt to respond and address collision occurs. The ADB Manager then begins the process of address resolution.

In this particular example, the ADB Manager first sends a Listen Register 3 command that specifies a device handler ID of $FE and a new device address of $E to the ADB device at address $2. Only the keyboard that did not detect the collision responds to this command and moves to address $E. Next, the ADB Manager sends a Talk Register 3 command to address $E to confirm that the keyboard has relocated there. Once the relocated keyboard responds with a random address, the ADB Manager again sends a Talk Register 3 command to address $2. Because two keyboards still remain at address $2, address collision occurs again. The ADB Manager therefore sends a Listen Register 3 command that specifies a device handler ID of $FE and a new device address of $D to the ADB device at address $2. Only the keyboard that did not detect the collision moves to address $D. There is now only one keyboard remaining at address $2. When the ADB Manager sends another Talk Register 3 command to address $2, the single keyboard does not detect a collision. It therefore accepts the next Listen Register 3 command from the ADB Manager that tells it to move to a new address ($C). Once more, the ADB Manager sends a Talk Register 3 command to address $2. When it receives no response from any devices, the ADB Manager moves the keyboard relocated to address $E back to address $2.

In contrast, the mouse and the graphics tablet are the only devices of their type connected to the ADB. As a result, neither device shares a default address with another device; the mouse is located at address $3 and the graphics tablet is located at address $4. When the ADB Manager builds the device table, no address collision occurs for either device and they remain at their original addresses.

For more information on the ADB device table, see “ADB Device Table” on page 5-13.

ADB Communication

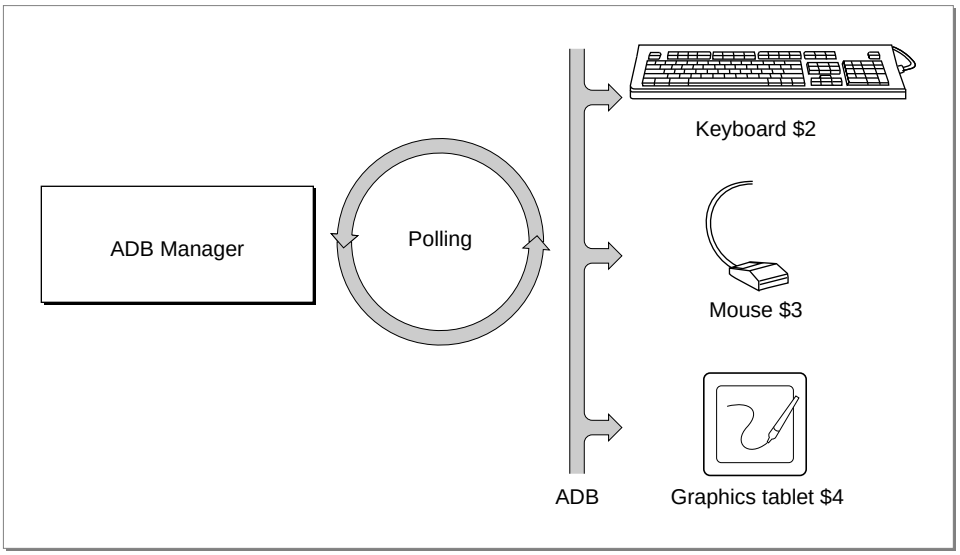
ADB devices cannot issue commands to the ADB Manager. Communication is accomplished in two ways. First, the ADB Manager performs polling of the ADB devices, and second, each ADB device can assert a service request signal to inform the ADB Manager that it has data to send. The ADB Manager passes the data sent by each ADB device to the associated device handler. In general, the ADB Manager continuously polls the ***active ADB device*** , which is the last device that sent new data after requesting service with a service request signal. The default active device is located at address $3, which is usually the mouse.

Polling (or autopolling) is accomplished by the ADB Manager repeatedly sending Talk Register 0 commands to an ADB device to see if it has new data to return. Register 0 is therefore the primary register for transferring data for all ADB devices. For an example of the register 0 contents for the Apple Standard keyboard, see Table 5-1 on page 5-10.

Note
If the data that is significant to the ADB device resides in an ADB register other than register 0, the device handler must directly retrieve the data from that register. For example, the Apple Extended keyboard contains data in both register 0 and register 2. The keyboard device handler must therefore directly retrieve the register 2 contents. ◆

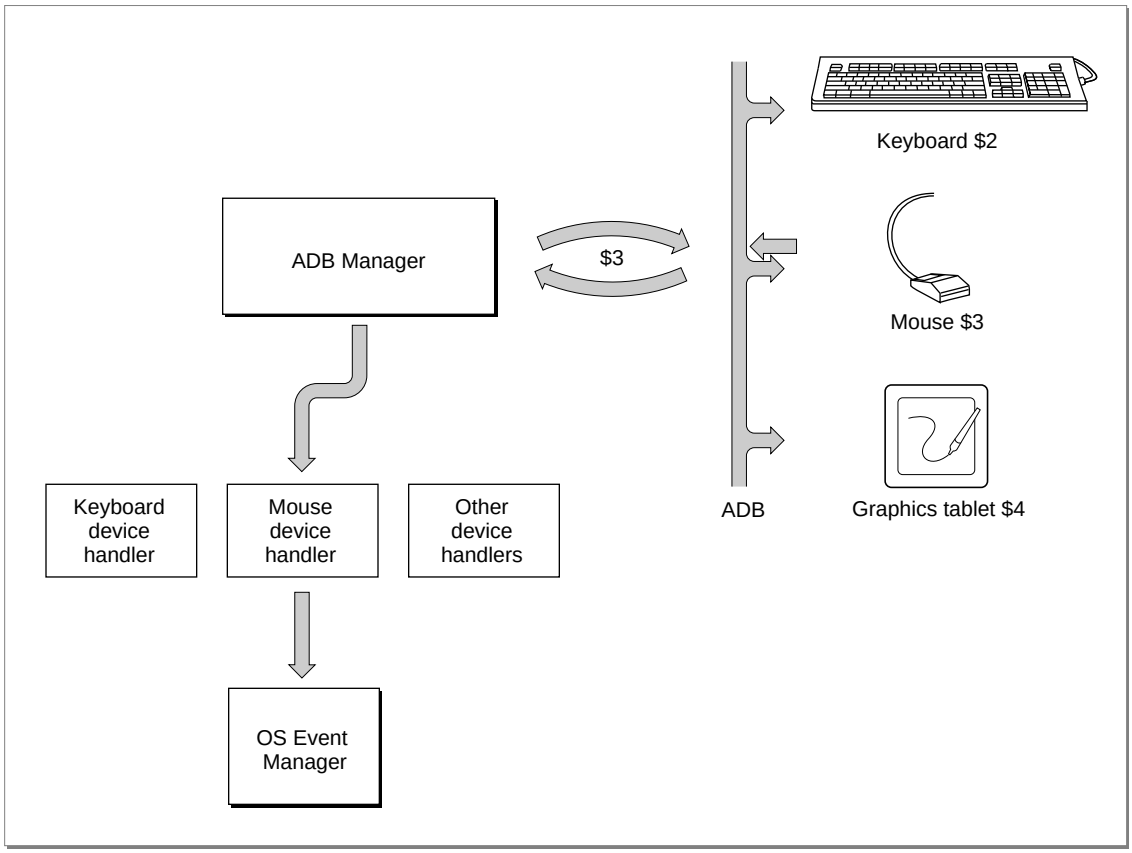
Figure 5-7 shows three ADB devices connected to the bus (a keyboard, a mouse, and a graphics tablet) and the ADB Manager performing polling.

Figure 5-7 Polling the ADB



An ADB device should respond to a Talk Register 0 command only if it has new data to send to the ADB Manager; that is, if the status of the device has changed since the last Talk Register 0 command. For example, Figure 5-8 shows a situation where the mouse is the active device. The ADB Manager polls the mouse, sending a Talk Register 0 command. If the mouse has new data to send, it should respond. Whenever the mouse responds with new data to a Talk Register 0 command, the ADB Manager sends this new data to the mouse handler, which uses the `PostEvent` function to place an event in the event queue.

Figure 5-8 How an ADB device responds to a polling request by the ADB Manager



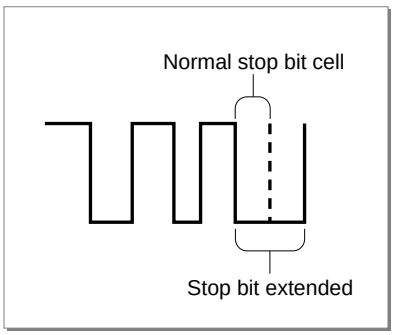
Note

Designing an ADB device to respond to a Talk Register 0 command only if it has new data to send can significantly optimize the performance of the Apple Desktop Bus. It reduces the effort required by the ADB Manager because it only has to call the device handler associated with a device when the device has actual data to send. It also avoids the endless polling cycles by the ADB Manager that can occur when an ADB device responds to a Talk Register 0 command with no new data. In an endless polling cycle, the ADB Manager continues to repeatedly poll the device not sending new data, rather than moving to the next ADB device that may have new data to send.

For further optimization, the ADB Manager automatically polls only those ADB devices that have an installed device handler. If an ADB device does not have a device handler installed, the ADB Manager skips that device during polling and instead polls an ADB device that has an installed device handler, even if the other device has not recently communicated with the ADB Manager. The ADB Manager may poll an ADB device that does not have an installed device handler, however, in response to a service request signal. ◆

If a Talk Register 0 command is completing, the ADB device should assert a special signal, known as a ***service request signal*** (or ***SRQ***), to inform the ADB Manager that it has data to send. As shown in Figure 5-9, an ADB device asserts an SRQ by holding the bus low during the low portion of the stop bit of any command or data transaction.

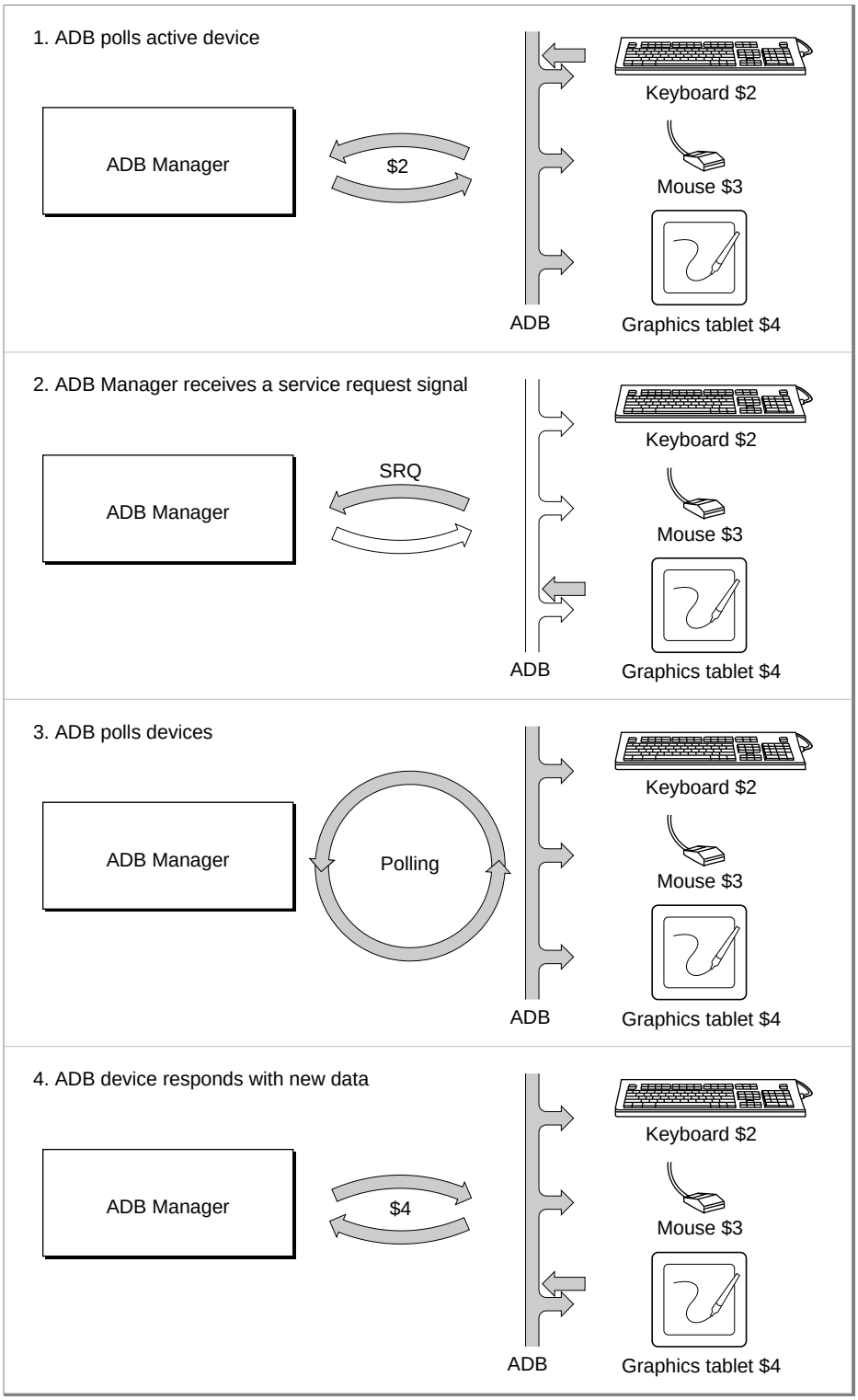
Figure 5-9 The ADB service request signal



For information on the timing parameters for ADB signals, see *Guide to the Macintosh Family Hardware*, second edition.

To identify which device asserted the SRQ, the ADB Manager polls each address known to contain an ADB device, beginning with the active ADB device. That is, if the first device polled by the ADB Manager does not respond to the Talk Register 0 command, it polls the next device. When the ADB Manager polls the device that asserted the SRQ, that device responds with new data. If another device asserts an SRQ, the ADB Manager continues polling until it finds that device. If no SRQ is asserted, this indicates that all pending data has been fetched and that the ADB Manager can return to polling the active device. For example, Figure 5-10 shows three ADB devices, with the ADB Manager polling the active ADB device. One of the three ADB devices, a graphics tablet, sends an SRQ to the ADB Manager. In this particular example, the ADB Manager responds by polling the active ADB device (in this case, the keyboard) and then polling the remaining ADB devices. After receiving a Talk Register 0 command from the ADB Manager, the graphics tablet can send its new data.

Figure 5-10 An ADB device asserts the service request signal



Using the ADB Manager

You can use the ADB Manager to communicate with and get information about devices attached to the Apple Desktop Bus. In general, applications interact with the ADB indirectly, by calling the Event Manager to retrieve information about user actions on the available input devices (keyboard, mouse, graphics tablet, and so forth). As a result, most applications do not need to know how to communicate directly with ADB devices, or even whether the ADB is present on the computer.

Some applications—such as diagnostic programs or other utilities—might want to report information about the ADB. Other software might even need to send commands directly to an ADB device (perhaps to query or modify device settings). This section shows how to

- determine whether the ADB Manager is present on the current computer
- get information about the devices attached to the ADB
- send commands to an ADB device in order to determine or modify device settings

For information on writing and installing ADB device handlers, see "Writing an ADB Device Handler" on page 5-29.

Checking for the ADB Manager

The Apple Desktop Bus was introduced on the Macintosh II and Macintosh SE computers. To test for the availability of the ADB Manager on your system, use the `NGetTrapAddress` function to see if the `_CountADBs` trap macro is available. See the chapter "Trap Manager" in *Inside Macintosh: Operating System Utilities* for information about the `NGetTrapAddress` function.

Getting Information About ADB Devices

You can use the ADB Manager to get several kinds of information about the ADB and about individual ADB devices on the bus. You can call `CountADBs` to determine how many devices are currently available on the Apple Desktop Bus. The `CountADBs` function simply counts the number of entries in the ADB device table.

You can call the `GetIndADB` function to get information about a device specified by its index in the ADB device table. The `GetIndADB` function returns as its function result the current ADB address of the device with the specified index and also returns additional information in a parameter block pointed to by one of its parameters. If you already know the address of an ADB device, you can call `GetADBInfo` to get that same information about the device.

Both `GetIndADB` and `GetADBInfo` return information about a particular device in an ADB data block, defined by the `ADBDataBlock` data type.

```
TYPE ADBDataBlock =
PACKED RECORD
   devType:          SignedByte;    {device handler ID}
   origADBAddr:      SignedByte;    {default ADB device address}
   dbServiceRtPtr:   Ptr;           {pointer to device handler}
   dbDataAreaAddr:   Ptr;           {pointer to data area}
END;
```

Note

The installation code for a device handler can set information (specifically the address of its device handler and optional data area) in its device's entry in the device table using the `SetADBInfo` function. ◆

You can examine the `devType` and `origADBAddr` fields of the `ADBData` block to determine what kind of ADB device is located at a particular ADB address. (Remember that once the ADB Manager has set the initial values for an ADB device in the ADB device table, it updates the device table entry for the device to reflect changes only to the address of the device handle routine and data area pointer. Thus, `GetIndADB` and `GetADBInfo` return the device's original device handler ID and original (default) ADB device address.) For example, the Apple Extended keyboard has a device handler ID of $02 and a default address of $2. Listing 5-1 shows one way to determine whether an ADB device is an Apple Extended keyboard.

Listing 5-1 Determining whether an ADB device is an Apple Extended keyboard

```
FUNCTION IsExtendedKeyboard (myAddress: ADBAddress): Boolean;
VAR
   myInfo:        ADBDataBlock;
   myCommand:     Integer;
   myErr:         OSErr;
CONST
   kExtKeyboardAddr = 2;
   kExtKeyboardOrigHandlerID = 2;
BEGIN
   myErr := GetADBInfo(myInfo, myAddress);
   IsExtendedKeyboard := (myInfo.origADBAddr = kExtKeyboardAddr)
                         AND (myInfo.devType = kExtKeyboardOrigHandlerID);
END;
```

The `IsExtendedKeyboard` function defined in Listing 5-1 is used later in this chapter, in Listing 5-5 on page 5-28.

Communicating With ADB Devices

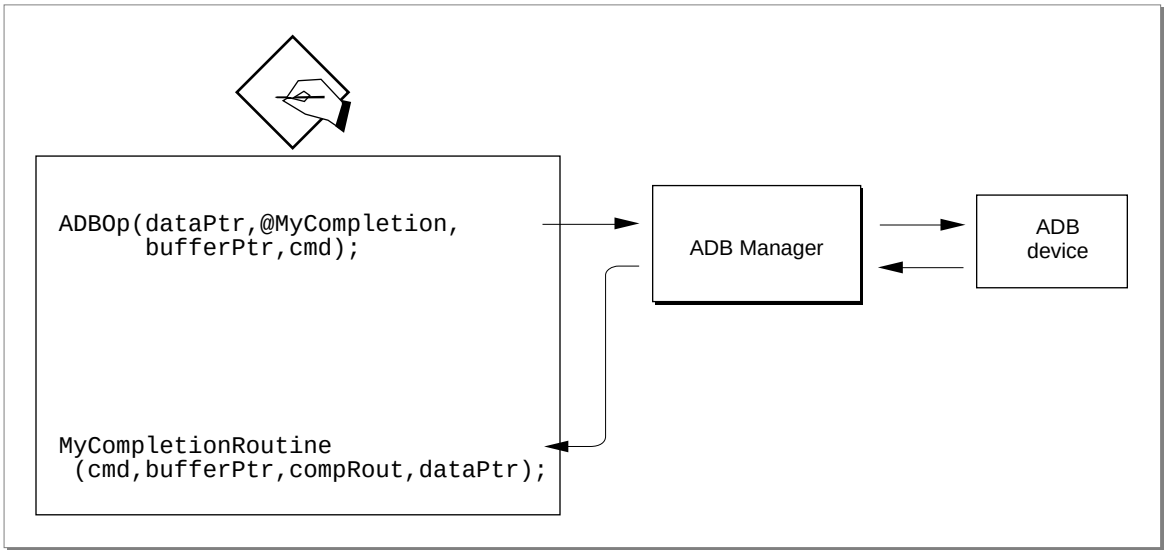
You can use the ADB Manager to communicate directly with ADB devices by sending ADB commands to those devices. In general, however, you don't need to do this, because the ADB Manager automatically polls for input from the connected ADB devices and passes any data received from a device to the device's device handler. Most applications should never interact directly with ADB devices, and even ADB device handlers need to do so only occasionally (for instance, to read or set device parameters stored in the device registers).

If you do need to send ADB commands directly to a device, you can do so using the `ADBOp` function. The `ADBOp` function transmits over the bus a command byte, whose structure is shown in Figure 5-2 on page 5-8 and Figure 5-3 on page 5-8. The command (Talk, Listen, Flush, and SendReset) and any register information are encoded into an integer that is passed to `ADBOp`. You also pass `ADBOp` three pointers:

- A pointer to the optional data area used by the completion routine.
- A pointer to a completion routine. This routine is executed once the command byte has been sent to the ADB device.
- A pointer to a Pascal string (maximum 8 bytes data preceded by one length byte). The first byte specifies the length of the string and the remaining bytes (if any) contain data to be sent to the device or provide storage for the data to be received from the device.

The `ADBOp` function is always executed asynchronously. If the bus is busy, the ADB command passed to `ADBOp` is held in a command queue until the bus is free. If your application requires synchronous behavior, you'll need to use a completion routine to determine when the ADB command itself has completed. Figure 5-11 shows the relationships between the `ADBOp` routine, the device to which it is directly communicating, the ADB Manager, and an ADB completion routine.

Figure 5-11 The ADBOp routine and an ADB completion routine



Listing 5-2 shows a way to send ADB commands synchronously.

Listing 5-2 Sending an ADB command synchronously

```
PROCEDURE MySetFlag;
{move a nonzero value into the word pointed to by register A2}
INLINE $34BC, $FFFF;       {MOVE.W #$FFFF, (A2)}

PROCEDURE MyCompletionRoutine;
BEGIN
   MySetFlag;              {set a flag to indicate done}
END;

FUNCTION MySendADBCommand (myBufferPtr: Ptr; myCommand: Integer): OSErr;
{send a command to an ADB device synchronously}
VAR
   myDone:     Integer;    {completion flag}
   myErr:      OSErr;
BEGIN
   myDone := 0;
   myErr := ADBOp(@myDone, @MyCompletionRoutine, myBufferPtr, myCommand);
   IF myErr = noErr THEN
      REPEAT
      UNTIL myDone <> 0;
```

```
   ELSE
      ; {ADB buffer overflowed -- retry command here}
   MySendADBCommand := myErr;
END;
```

The `MySendADBCommand` function sets the completion flag `myDone` to zero and then calls `ADBOp`, passing the address of that completion flag and the address of a completion routine along with the two parameters passed to `MySendADBCommand`. The completion routine simply calls an inline assembly routine that moves a nonzero value into the word pointed to by register A2. (When the completion routine is called, register A2 points to the optional data area, in this case, to the `myDone` variable.) The `MySendADBCommand` function waits until the value of the `myDone` variable changes, and then returns.

Rather than provide a completion routine to verify that a Talk command has completed, you can initialize the first byte of the data buffer to 0 before sending the command. The first byte of the data buffer contains the length of the buffer (in the same manner that the first byte of a Pascal string contains the length of the string). The data buffer can include from 0 to 8 bytes of information. After sending the command with `ADBOp`, you can then test the first byte of the data buffer to determine whether the command has completed. Once the first byte of information contains a nonzero value, then the command has completed, and the first byte of the buffer indicates the number of bytes returned by the ADB device.

Listing 5-3, Listing 5-4, and Listing 5-5 illustrate how to use the `MySendADBCommand` function (defined in Listing 5-2) to blink the LED lights on the Apple Extended keyboard. The Apple Extended keyboard maintains the current setting of the LED lights in the lower 3 bits of device register 2. You can read the current light setting by issuing a Talk command to the keyboard, as shown in Listing 5-3.

Listing 5-3 Reading the current state of the LED lights

```
VAR
   gRegisterData:    PACKED ARRAY[0..8] of Byte;   {buffer for register data}
CONST
   kListenMask       = 8;     {masks for ADB commands}
   kTalkMask         = 12;
   kLEDRegister      = 2;     {register containing LED settings}
   kLEDValueMask     = 7;     {mask for bits containing current LED setting}

FUNCTION MyGetLEDValue (myAddress: ADBAddress; VAR myLEDValue: Integer)
                       : OSErr;
VAR
   myCommand:  Integer;
   myErr:      OSErr;
```

```
BEGIN
   {initialize length of buffer; on return, the ADB device sets }
   gRegisterData[0] := Byte(0);  { this byte to the number of bytes returned}
   {get existing register contents with a Talk command}
   myCommand := (myAddress * 16) + kTalkMask + kLEDRegister;
   myErr := MySendADBCommand(@gRegisterData, myCommand);
   IF myErr = noErr THEN           {make sure completed successfuly}
      {gRegisterData now contains the existing data in device register 2; }
      { the lower 3 bits of byte 2 contain the LED value}
      myLEDValue := Integer(BAND(gRegisterData[2], kLEDValueMask))
   ELSE
      myLEDValue := 0;
   MyGetLEDValue := myErr;
END;
```

The `MyGetLEDValue` function constructs a Talk Register 2 command by adding the address value to command and register masks defined by the application. Then it calls the `MySendADBCommand` function to communicate with the device at the specified address. If `MySendADBCommand` completes successfully, then the `gRegisterData` variable contains (in array elements 1 and 2) the two-byte value in device register 2. Only the lower 3 bits of that value are used for the LED settings. If one of those bits is set, the corresponding light is off. Note that if `MyGetLedValue` returns an error, this generally indicates that the `ADBOp` buffer overflowed.

The `MySetLEDValue` function defined in Listing 5-4 sets the LED lights to a specific pattern.

Listing 5-4 Setting the current state of the LED lights

```
FUNCTION MySetLEDValue (myAddress: ADBAddress; myValue: Integer): OSErr;
VAR
   myCommand:  Integer;
   myByte:     Byte;               {existing byte 2 of device register 2}
   myErr:      OSErr;
BEGIN
   gRegisterData[0] := Byte(2);     {set length of buffer}
   {get existing register contents with a Talk command}
   myCommand := (myAddress * 16) + kTalkMask + kLEDRegister;
   myErr := MySendADBCommand(@gRegisterData, myCommand);
   MySetLEDValue := myErr;
   IF myErr <> noErr THEN          {make sure completed successfuly}
      EXIT(MySetLEDValue);
   {gRegisterData now contains the existing data in device register 2; }
   { reset the lower 3 bits of byte 2 to the desired value}
```

```
   myByte := gRegisterData[2];
   myByte := BAND(myByte, 255 - 7);          {mask off lower three bits}
   myByte := BOR(myByte, Byte(myValue));     {install desired value}
   gRegisterData[2] := myByte;
   myCommand := (myAddress * 16) + kListenMask + kLEDRegister;
   MySetLEDValue := MySendADBCommand(@gRegisterData, myCommand);
END;
```

Notice that the `MySetLEDValue` function first reads the current value in device register 2. This is necessary to preserve the bits in that register that do not encode the LED state. Register 2 contains sixteen bits; be sure to change only the three bits that represent the three LED lights.

Finally, the `MyCountWithLEDs` procedure shown in Listing 5-5 uses the `MyGetLEDValue` and `MySetLEDValue` routines to "count" in binary.

Listing 5-5 Counting in binary using a keyboard's LED lights

```
PROCEDURE MyCountWithLEDs;
VAR
   myValue:    Integer;
   myIndex:    Integer;
   myAddress:  ADBAddress;
   myOrigLED:  Integer;
   myInfo:     ADBDataBlock;        {needed for GetIndADB; ignored here}
   myDelay:    LongInt;             {needed for Delay; ignored here}
   myErr:      OSErr;
BEGIN
   FOR myIndex := 1 TO CountADBs DO
   BEGIN
      myAddress := GetIndADB(myInfo, myIndex);
      IF IsExtendedKeyboard(myAddress) THEN
      BEGIN
         {save original state of LED lights}
         myErr := MyGetLEDValue(myAddress, myOrigLED);
         myValue := 7;                     {turn all the lights OFF}
         WHILE myValue >= 0 DO
         BEGIN
            myErr := MySetLEDValue(myAddress, myValue);
            myValue := myValue - 1;
            Delay(30, myDelay);
         END;
         {restore original state of LED lights}
         myErr := MySetLEDValue(myAddress, myOrigLED);
```

```
      END; {IF}
   END; {FOR}
END;
```

The `MyCountWithLEDs` procedure looks for Apple Extended keyboards on the ADB and counts from 0 to 7, in binary, on the LED lights of any such keyboard it finds.

Note
The techniques shown in this section for reading and writing the LED state of an Apple Extended keyboard are provided for illustrative purposes only. Your application or other software should in general not modify the LED state of the user's keyboard. ◆

Writing an ADB Device Handler

The previous section, "Using the ADB Manager," illustrates how you can use the ADB Manager to communicate with and get information about devices attached to the ADB. This section describes how to write a device handler for an ADB device. You should write a device handler for a device only if you are the manufacturer of that device.

A device handler is a low-level routine that communicates with a particular ADB device. The device handler gathers data from an ADB device through the ADB Manager and interprets the data; depending on the device, the device handler might then post an event into the event queue using the `PostEvent` function.

A single device handler can manage more than one device; for example, the standard device handler for the Apple Extended keyboard can manage multiple extended keyboards. Also, in some cases the same handler can be used to manage two or more device types. For example, a relative-position graphics tablet could emulate a mouse, using the same default ADB device address and device handler ID as used by the mouse, and providing the same information in response to Talk commands. In this case, when both the mouse and tablet are connected to the ADB at the same time, the ADB Manager calls the mouse handler when either device requires it.

Each ADB device has a default ADB device address and default device handler ID. Some ADB devices support more than one device handler ID. In this case, the device handler manages the device based on the current device handler ID; this allows an ADB device to add or modify its performance or feature set. For more information about ADB addresses and device handler IDs, see "Default ADB Device Address and Device Handler Identification" on page 5-11.

In addition to writing a device handler for your device, you need to write the code that installs the device handler. The next few sections explain how to write a device handler and code to install the handler.

IMPORTANT
You need the information in this section only if you are writing a device handler for a new ADB device. The Macintosh Operating System includes device handlers for all Apple keyboards and Apple mouse devices. You do not need to write a device handler to receive input from these standard Apple devices; instead, your application should get information about mouse movements and key presses by calling the Event Manager. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information about how the Event Manager interacts with applications. ▲

Installing an ADB Device Handler

You install a device handler for an ADB device by placing the address of the device handler in the device's entry of the ADB device table. To do this, and to make your ADB device available to the user as soon as possible, Apple recommends that you provide users with a system extension that installs your device handler. Thus, your system extension should contain your device handler as well as the code that installs the device handler into the appropriate entry of the ADB device table. (See "ADB Device Table" on page 5-13 for a description of the structure of entries in the ADB device table.)

Your installation code should search the ADB device table for an entry whose default ADB device address and default device handler ID match the values assigned to your device. For example, if your ADB device has a default address of $7 and a default handler ID of 99, your installation code should search the ADB device table for entries matching these values. If your installation code finds any matching entries, it should install the address of your device handler into your device's entry in the ADB device table. The typical installation code for ADB devices other than a keyboard or mouse does this: calls the `CountADBs` function to determine the number of entries in the ADB device table; repeatedly calls the `GetIndADB` function to index through each device table entry and compares the default ADB device address and device handler ID with those of your device; for any matching entries, calls the `SetADBInfo` function to install the device handler for that device into the device's entry in the ADB device table. Note that before installing the address of your device handler into the ADB device table, your installation code must first allocate space in the system heap for your handler and copy your handler to this area; your installation code should also allocate space in the system heap for its optional data area.

If you provide a device handler for a mouse or keyboard you must consider whether your ADB device should use a standard device handler during initial startup (until your system extension has a chance to run and install the device's device handler) or whether your ADB device should use only its own device handler (which means your device will be unable to respond to the user until its handler is installed).

When the ADB Manager first builds the ADB device table, it associates with each device the device's default ADB address, the device's current ADB address, and the device's default device handler ID. In addition, for each device it initializes the field that contains the address of the device handler for the device and the field that contains a pointer to a data area used by the device handler. For Apple ADB devices, the ADB Manager installs

the appropriate device handler provided by Apple. Thus, the device handler for an Apple keyboard or Apple mouse is available almost immediately after initial startup. For all other ADB devices, the device's device handler must be specifically installed by the device's installation code. For example, the ADB Manager does not install the Apple device handler for a keyboard with a default ADB device address of $2 and a default device handler ID of 99; instead, the device's system extension must install the device's device handler.

If your ADB device is a keyboard or mouse and you want it to function as soon as possible in the startup process and before system extensions are run, you can design your ADB device to emulate an Apple keyboard or mouse and use that device's device handler until its own device handler is installed. In this case, your ADB device's default ADB address and default device handler ID initially matches that of an Apple device. This causes the ADB Manager to install the address of an Apple device handler for your device's entry in the ADB device table. To install the actual device handler for your device, you can provide a system extension that

- uses the `CountADBs` function to count the number of entries in the ADB device table.
- repeatedly uses the `GetIndADB` function to examine each entry in the ADB device table for an entry with a default ADB device address and default device handler ID that matches that of a standard device.
- upon finding a matching entry, uses the `ADBOp` function to send a Talk Register 3 command to the selected ADB device so that it sends its contents across the bus; uses the `ADBOp` function to send a Listen Register 3 command to the device to change its device handler ID from its default device handler ID to its actual device handler ID; and uses the `ADBOp` function to send another Talk Register 3 command to the device and examine the register contents to see whether the device returns the new device handler ID. If so, your extension has found the index entry for your device and can use the `SetADBInfo` function to install the appropriate device handler for your device. Note that when you request an ADB device to change to another device handler ID, the ADB Manager does not update the ADB device table entry to reflect the new device handler ID. You can find out the new handler ID for that device only by sending it a Talk Register 3 command.

Your installation code should also store a pointer to its reinitialization code in the system global variable `JADBProc` and should preserve the existing value of `JADBProc`, as illustrated in Listing 5-6 and Listing 5-7.

The next three listings, Listing 5-6, Listing 5-7, and Listing 5-8, show code that installs a device handler, handles reinitialization by appropriate use of the system global variable `JADBProc`, and performs the actual actions of the device handler.

Listing 5-6 shows an example of code that installs an ADB device handler. The code first defines some constants. It also defines a stack frame which includes storage for a variable called `myADBDB` that is used later as a parameter block for both `GetIndADB` and `SetADBInfo`. The installation code then jumps to the code starting at the label `MyInstallHandlers`; this code uses `CountADBs` and then `GetIndADB` to search all entries in the ADB device table for a matching default ADB device address and device handler ID. If it finds such an entry, it uses the code at the label `MySetDeviceInfo` to set up information for that device in the device's entry in the ADB device table.

Specifically, for each occurrence of a matching entry, the code at the label `MySetDeviceInfo` allocates space in the system heap for the data area used by the device handler for the ADB device at that address. (It does not need to allocate space for the handler itself at this time. This is because a resource containing the code shown in Listing 5-6 is marked to be loaded into the system heap; thus the system software loads the resource into the system heap when it executes this system extension.) The code then uses the `SetADBInfo` function to install into the ADB device table the address of the device's device handler as well as a pointer to the global data area used by the device handler.

Finally, the installation code stores in the `iNextProc` field the current value of the system global variable `JADBProc` and then sets `JADBProc` to contain a pointer to `myJADBProc`.

Listing 5-6 Installing an ADB device handler

```
;For installation to work, the resource containing this resource must be
;marked as sysHeap loaded. This way, you do not have to copy a version of it
;into the system heap prior to installing.
; MPW Build commands:
;   ASM 'ADBSample.a'
;   Link -t INIT -c WeSt -ra ADBSample=resSysHeap -rt INIT=128 -m MAIN -sg ∂
;     ADBSample 'ADBSample.a'.o -o ADBSample

myAddr          EQU     $xx             ;default ADB device address
myADBType       EQU     $xx             ;device handler ID definition

main            PROC    EXPORT

StackFrame      RECORD  {A6Link}, DECR  ;build a stack frame record
ParamBegin      EQU     *               ;start parameters after this point
ParamSize       EQU     ParamBegin-*    ;size of all the passed parameters
RetAddr         DS.L    1               ;place holder for return address
A6Link          DS.L    1               ;place holder for A6 link
myADBDB         DS      ADBDataBlock    ;local handle to our ADB data block
LocalSize       EQU     *               ;size of all the local variables
                ENDR

                WITH    StackFrame
                WITH    ADBDataBlock
                LINK    A6, #0          ;make a stack frame
                BSR     MyInstallHandlers ;install handlers for our devices
                TST.W   D0              ;D0 = number of old devices found
                BEQ.S   @exit           ;if none, exit
```

```
                LEA     main, A0        ;after installing, we need to
                _RecoverHandle, SYS     ; recover the handle and then
                MOVE.L  A0 -(SP)        ; detach this resource so it always
                _DetachResource         ; stays in memory

                LEA     iNextProc, A2   ;get pointer to old vector storage
                LEA     JADBProc, A3    ;make pointer to low memory vector
                MOVE.L  (A3), (A2)      ;save contents of vector for chaining

                LEA     myJADBProc, A2  ;get pointer to our jADBProc
                MOVE.L  A2, (A3)        ;install it in the low memory vector

@exit           UNLK    A6              ;dispose local variables
                RTS

;placeholder for MyADBHandler - see Listing 5-8 on page 5-37
;placeholder for myJADBProc - see Listing 5-7 on page 5-35

;MySetDeviceInfo routine (called by MyInstallHandlers)
; on entry: D0 = ADB address of our device
; does not preserve D4 or A1
MySetDeviceInfo
                LINK    A6, #LocalSize      ;make a stack frame
                LEA     myADBDB(A6), A1     ;pointer to stack-based param block
                LEA     MyADBHandler, A3    ;pointer to the handler routine
                MOVE.W  D0, D4              ;save the actual address
                MOVE.L  A3, (A1)            ;set up the handler address
                MOVE.L  #10, D0             ;data area for device is 10 bytes
                _NewPtr, SYS, CLEAR         ;allocate our data area
                TST.W   D0                  ;test for error
                BNE.S   @SDIExit            ;exit if error
                MOVE.L  A0, 4(A1)           ;put pointer to parameter data
                                            ; in data area
                MOVE.W  D4, D0              ;put actual address to set in D0
                MOVE.L  A1, A0              ;put parameter block pointer in A0
                _SetADBInfo                 ;set up info for this device
@SDIExit
                UNLK    A6                  ;dispose stack frame
                RTS                         ;exit this routine

iNextProc       DC.L    0                   ;store pointer to next jADBProc
```

```
;MyInstallHandlers routine (called by main)
; on exit: D0 = number of our device types found
; does not preserve D1, D2, D3, D4 or A1
MyInstallHandlers
              LINK     A6, #LocalSize    ;make a stack frame
              CLR.L    D3                ;clear device counter
              _CountADBs                 ;get number of ADB devices
              MOVE.W   D0, D2            ;save this number in D2
              BEQ.S    @return           ;exit if none
                                                ;put handler ID and
              MOVE.W   #(myADBType<<8)+myAddr, D1 ; default address into D1
@cntLoop
              MOVE.W   D2, D0            ;put device index in D0
              LEA      myADBDB(A6), A0   ;pointer to stack-based param block
              _GetINDADB                 ;get an ADB device table entry
              BMI.S    @nextRec          ;skip if invalid
              CMP.W    devType(A0), D1   ;is this one of our devices?
              BNE.S    @nextRec          ;skip if no match
              BSR.S    MySetDeviceInfo   ;set handler for this device
              ADDQ     #1, D3            ;found one of our devices, add to D3
@nextRec      SUBQ.W   #1, D2            ;try next index
              BNE.S    @cntLoop          ;loop if more
              MOVE.L   D3, D0            ;return number found in D0
@return       UNLK     A6
              RTS
              ENDP
              END
```

Note

In the past, Apple recommended that you install an ADB device handler by placing the ADB device handler in an `'ADBS'` resource in the System file. In this case, the `'ADBS'` resource ID corresponds to the ADB device's default address. At system startup, the ADB Manager searches the System file for `'ADBS'` resources for only those ADB devices that appear on the bus. The ADB Manager then loads these resources into memory and executes them. The ADB Manager also reads register 3 for each ADB device and places the device's default ADB device address and device handler ID into the ADB device table. This method, however, does not offer the same flexibility and scope as when you install a handler with an extension. For example, because `'ADBS'` resource IDs are indexed only by their default addresses, you cannot install ADB resources for two different devices at the same address using `'ADBS'` resources. Apple therefore recommends that you install all ADB device handlers using a system extension. ◆

Your installation code should set up the value of `JADBProc` (by chaining) to point to a routine that you provide which appropriately handles the case when the ADB is reinitialized. When the ADB is reinitialized, the ADB Manager calls the routine pointed to by the system global variable `JADBProc`; it calls this routine twice: once before reinitializing the ADB, and once after reinitializing the ADB. When this routine is called, D0 contains the value 0 for preprocessing and 1 for postprocessing. Your routine must restore the value of D0 and branch to the original value of `JADBProc` on exit.

For preprocessing, your reinstallation routine should deallocate any storage. It must also take action for postprocessing. Because the ADB (and ADB device table) is reinitialized during postprocessing, the ADB Manager might need to perform address resolution. As a result, you cannot assume that your ADB device still resides at its default address after postprocessing occurs. Therefore, for postprocessing your reinstallation routine should search the ADB bus for a matching device (just as in its installation code) and install its entry into the ADB device table. Finally, the code jumps to the routine stored in `iNextProc`, and thus chains to the next routine that needs to perform postprocessing. Listing 5-7 shows an example of this entire process.

Listing 5-7 Installing a routine pointer into `JADBProc`

```
;main goes here, see Listing 5-6 on page 5-32
;handler code goes here, see Listing 5-8 on page 5-37
;NOTE: This routine must be installed as part of the handler.
myJADBProc
               LINK     A6, #LocalSize    ;make a stack frame
               MOVEM.L  D0-D2/A1, -(SP)   ;save registers for next procedure
               TST.B    D0                ; D0 = 0 for pre-processing,
                                          ; D0 = 1 for post-processing
               BEQ.S    @preProc          ;if 0, pre-process data areas

@postProc
               BSR.S    MyInstallHandlers ;install handlers (Listing 5-6)
               BRA.S    @JADBExit
@prePost
               LEA      myADBDB(A6), A1   ;pointer to stack-based param block
               LEA      MyADBHandler, A2  ;address of handler for comparison
               _CountADBs                 ;get the number of ADB devices
               MOVE.W   D0, D2            ;save this value in D2
               BEQ.S    @JADBExit         ;exit if none
                                                ;put handler ID and
               MOVE.W   #(myADBType<<8)+myAddr, D1 ; default address into D1
@preLoop
               MOVE.W   D2, D0            ;current index
               MOVE.L   A1, A0            ;address of data block
               _GetIndADB                 ;get ADB device table entry
```

```
            BMI.S    @nextRec          ;skip if invalid
            CMP.W    devType(A0), D1   ;is this one of our devices?
            BNE.S    @nextRec          ;skip if no match
            CMPA.L   dbServiceRtPtr(A0), A2  ;compare with our handler ID
            BNE.S    @nextRec          ;if no match, don't delete pointer
            MOVE.L   dbDataAreaAddr(A0), A0  ;get the pointer to dispose
            _DisposePtr                ;if matches, it's ours, so dispose
@nextRec
            SUBQ.W   #1, D2            ;get next index
            BNE.S    @preLoop          ;loop if more
@JADBExit
            MOVEM.L  (SP)+, D0-D2/A1   ;restore registers
            UNLK     A6                ;dispose stack frame
            LEA      iNextProc, A0     ;get pointer to next procedure
            MOVE.L   (A0), A0
            JMP      (A0)              ;jump to next procedure
```

Creating an ADB Device Handler

A device handler communicates with a particular ADB device by gathering data about the device it manages from the ADB Manager, and then interpreting that data. For example, the device handler for a particular device might then post an event into the event queue using `PostEvent`.

Whenever an ADB device sends data (by responding to a Talk Register 0 command), the ADB Manager calls the associated device handler. The ADB Manager passes these parameters to the device handler:

- in register A0, a pointer to the ADB data sent by the ADB device
- in register A1, a pointer to the device handler routine
- in register A2, a pointer to the data area (if any) associated with the device handler
- in register D0, the ADB command that resulted in the handler being called

Note

ADB device handlers are always called at interrupt time; they must follow all rules for interrupt-level processing as described in *Inside Macintosh: Processes*. ◆

Listing 5-8 gives an example of a simple device handler that handles data from an ADB device. (Listing 5-6 on page 5-32 shows code that installs the address of this handler into the ADB device table.) This device handler simply saves the data sent by the ADB device into the device handler's global data area. Note that you must include with your device handler code that handles reinitialization of the ADB (see Listing 5-7 on page 5-35 for details of reinitialization).

Listing 5-8 A sample device handler

```
MyADBHandler
      ANDI.B   #$0F, D0          ;check command
      CMPI.B   #$0C, D0          ;was it a talk R0 command?
      BNE.S    @exit             ; no, exit (something is wrong)
      MOVE.B   (A0)+, D0         ;get the count
      CMPI.B   #2, D0            ;this device only sends 2 bytes
      BNE.S    @exit             ;bad count, exit
      MOVE. B  (A0)+, HndlrData(A2) ;grab the 1st byte, save in global area
      MOVE.B   (A0)+, MoreData(A2)  ;grab the 2nd byte, save in global area
@exit         RTS
      ;code from Listing 5-7 goes here
```

ADB Manager Reference

This section describes the data structures and routines provided by the ADB Manager. See "Using the ADB Manager," beginning on page 5-22, and "Writing an ADB Device Handler" on page 5-29, for detailed instructions on using these routines.

Data Structures

This section describes the ADB data block, ADB information block, and ADB operation block.

ADB Data Block

You can get information about an ADB device by calling the functions `GetIndADB` and `GetADBInfo`. These functions return information from the ADB device table in an ADB data block, defined by the `ADBDataBlock` data type.

```
TYPE ADBDataBlock =
PACKED RECORD
   devType:          SignedByte;    {device handler ID}
   origADBAddr:      SignedByte;    {original ADB address}
   dbServiceRtPtr:   Ptr;           {pointer to device handler}
   dbDataAreaAddr:   Ptr;           {pointer to data area}
END;
ADBDBlkPtr = ^ADBDataBlock;
```

Field descriptions

`devType` The device handler ID of the ADB device.

`origADBAddr` The device's default ADB address.

`dbServiceRtPtr`
A pointer to the device's device handler.

`dbDataAreaAddr`
A pointer to the device handler's optional data area.

ADB Information Block

You can set a device's device handler routine and data area by calling the `SetADBInfo` function. You pass `SetADBInfo` an ADB information block, defined by the `ADBSetInfoBlock` data type.

```
TYPE ADBSetInfoBlock =
RECORD
   siServiceRtPtr:   Ptr;           {pointer to device handler}
   siDataAreaAddr:   Ptr;           {pointer to data area}
END;
ADBSInfoPtr = ^ADBSetInfoBlock;
```

Field descriptions

`siServiceRtPtr`
A pointer to the device handler.

`siDataAreaAddr`
A pointer to the device handler's optional data area.

Remember that once the ADB Manager has set the initial values for an ADB device in the ADB device table, it updates the device table entry for the device to reflect changes only to the address of the device handler routine and data area pointer.

ADB Operation Block

You use the ADB operation block to pass information to the `ADBOp` function if you call the function from assembly language. The ADB operation block is defined by the `ADBOpBlock` data type.

```
TYPE ADBOpBlock =
   RECORD
      dataBuffPtr:      Ptr;          {address of data buffer}
      opServiceRtPtr:   Ptr;          {pointer to device handler}
      opDataAreaPtr:    Ptr;          {pointer to optional data
                                        area}
   END;
   ADBOpBPtr = ^ADBOpBlock;
```

Field descriptions

`dataBuffPtr` A pointer to a variable-length data buffer. The first byte of the buffer must contain the buffer's length.

`opServiceRtPtr` A pointer to a completion routine.

`opDataAreaPtr` A pointer to an optional data area.

ADB Manager Routines

The ADB Manager provides routines that you can use to initialize the ADB, communicate with ADB devices, and get or set ADB device information. In general, you need to use these routines only if you need to access devices on the ADB directly or communicate with a special device. You'll also need to use some of these routines to install an ADB device handler.

Initializing the ADB Manager

The ADB Manager provides the `ADBReInit` procedure to initialize the Apple Desktop Bus. As explained in the following paragraphs, however, you probably won't ever need to call `ADBReInit`.

ADBReInit

The Macintosh Operating System uses the `ADBReInit` procedure to reinitialize the Apple Desktop Bus.

```
PROCEDURE ADBReInit;
```

DESCRIPTION

The `ADBReInit` procedure reinitializes the Apple Desktop Bus to its original condition at system startup time. It clears the ADB device table and places a SendReset command on the bus to reset all devices to their original addresses. The ADB Manager resolves any address conflicts and rebuilds the device table.

IMPORTANT

In general, your application shouldn't call `ADBReInit`. If you need to assign a different device handler to a device, or activate a "virtual" device associated with some device that is already connected to the bus, you can use the `SetADBInfo` routine. ▲

The `ADBReInit` procedure also calls the routine pointed to by the system global variable `JADBProc` at the beginning and end of its execution. You can insert your own

preprocessing and postprocessing routine by changing the value of JADBProc. When this routine is called, D0 contains the value 0 for preprocessing and 1 for postprocessing. Your routine must restore the value of D0 and branch to the original value of JADBProc on exit. Because the ADB is reinitialized during postprocessing, the ADB Manager might need to perform address resolution. As a result, you cannot assume that your ADB device still resides at its default address after postprocessing occurs.

SPECIAL CONSIDERATIONS

Calling ADBReInit on computers running system software versions earlier than 6.0.4 can cause incorrect keyboard layouts to be loaded.

The ADBReInit procedure does not deallocate memory that has been allocated by the device handler installation code.

If you provide a device handler that is installed by a system extension, you must reinstall the entry for your ADB device in the ADB device table. See "Installing an ADB Device Handler," beginning on page 5-30 for more information.

Communicating Through the ADB

You can use the ADBOp function to communicate with devices on the ADB. In general, however, you shouldn't need to call ADBOp. Applications should get information about the user's input on ADB devices by calling the appropriate Event Manager routines. In addition, the ADB Manager automatically polls device register 0 (the register that contains the data to be transmitted from the device to the ADB device handler) as part of its normal bus polling and service request handling. As a result, device handlers should need to call ADBOp only occasionally for special purposes, such as setting device modes or obtaining device status.

ADBOp

You can use the ADBOp function to send a command to an ADB device.

```
FUNCTION ADBOp (data: Ptr; compRout: ProcPtr; buffer: Ptr;
                commandNum: Integer): OSErr;
```

`data` A pointer to an optional data area.

`compRout` A pointer to a completion routine.

`buffer` A pointer to a variable-length data buffer. The first byte of the buffer must contain the buffer's length.

`commandNum`

A command number. The command number is a 1-byte value that encodes the command to be performed, the register the command refers to, and the desired action the target device should perform.

DESCRIPTION

The `ADBOp` function transmits over the bus the command byte whose value is given by the `commandNum` parameter. See Figure 5-2 on page 5-8 for the structure of this command byte. For a Listen command, the ADB Manager also transmits the data pointed to by the `buffer` parameter. Upon completion of a Talk command, the area pointed to by the `buffer` parameter contains the data returned by the ADB device. The `ADBOp` function executes only when the ADB would otherwise be idle; if the bus is busy, the command byte is held in a command queue. If the command queue is full, the `ADBOp` function returns an error and the command is not placed in the queue.

The length of the data buffer pointed to by the `buffer` parameter must be contained in its first byte (in the same manner that the first byte of a Pascal string contains the length of the string). The data buffer can include from 0 to 8 bytes of information. For a Listen command, the data buffer should contain any data to be sent to the device. For a Talk command, the contents of the data buffer are valid only on completion of the command. To verify that the Talk command is completed, you should provide a completion routine when you send the command to an ADB device or simply test the value of the first byte of the data buffer (which contains the length of the buffer).

The optional data area to which the `data` parameter points is intended for storage used by the completion routine pointed to by the `compRout` parameter. When you use `ADBOp` to send a command, you can optionally supply a completion routine as a parameter. See "ADB Command Completion Routines," beginning on page 5-47 for details on completion routines.

SPECIAL CONSIDERATIONS

The `ADBOp` function is always executed asynchronously. The result code returned by `ADBOp` indicates only whether the ADB command was successfully placed into the ADB command queue, not whether the command itself was successful. A method for interacting with the ADB bus synchronously is illustrated in "Communicating With ADB Devices," beginning on page 5-24.

ASSEMBLY-LANGUAGE INFORMATION

The ADB operation block contains some of the information required by the `ADBOp` function. You'll need to set up an ADB operation block only if you call `ADBOp` from assembly-language. (In Pascal, the ADB operation block is defined by the `ADBOpBlock` data type.)

The registers on entry and exit for `ADBOp` are

Registers on entry

A0 Address of a parameter block of type `ADBOpBlock`

D0 A command number

Registers on exit

D0 Result code

The parameter block whose address is passed in register A0 has this structure

Parameter block:

→	`dataBuffPtr`	`Ptr`	A pointer to a variable-length data buffer. The first byte of the buffer must contain the buffer's length.
→	`opServiceRtPtr`	`Ptr`	A pointer to a completion routine.
→	`opDataAreaPtr`	`Ptr`	A pointer to an optional data area.

RESULT CODES

`noErr`	0	No error
`errADBOp`	–1	Command queue is full. Retry command.

SEE ALSO

See Listing 5-2 on page 5-25 for an example of using the `ADBOp` function.

Getting ADB Device Information

You can use the ADB Manager functions in this section to determine how many ADB devices are present and to get information about a specific ADB device, specified either by its ADB device address or by its index in the ADB device table.

CountADBs

You can use the `CountADBs` function to determine how many ADB devices are connected to the bus.

```
FUNCTION CountADBs: Integer;
```

DESCRIPTION

The `CountADBs` function returns a value representing the number of devices connected to the bus; it determines this information by counting the number of entries in the ADB device table.

GetIndADB

You can use the `GetIndADB` function to get information about an ADB device, specified by its index in the ADB device table.

```
FUNCTION GetIndADB (VAR info: ADBDataBlock;
                    devTableIndex: Integer): ADBAddress;
```

`info` An ADB data block. On exit, the fields of this parameter block are filled with information about the specified ADB device.

`devTableIndex`
An index into the ADB device table.

Parameter block

←	`devType`	`SignedByte`	The device handler ID.
←	`origADBAddr`	`SignedByte`	The device's default ADB address.
←	`dbServiceRtPtr`	`Ptr`	The address of the device's device handler routine.
←	`dbDataAreaAddr`	`Ptr`	The address of the device handler's data storage area.

DESCRIPTION

The `GetIndADB` function returns information from the ADB device table entry whose index number is specified by the `devTableIndex` parameter. The information is returned in an ADB data block, passed in the `info` parameter.

The `GetIndADB` function also returns the current ADB address of the specified device as its function result. If, however, `GetIndADB` is unable to find the specified entry in the ADB device table, it returns a negative value as its function result. In that case, the fields of the `info` data block are undefined.

SPECIAL CONSIDERATIONS

Once the ADB Manager has set the initial values for an ADB device in the ADB device table, it updates the device table entry for the device to reflect changes only to the address of the device handler routine and data area pointer.

GetADBInfo

You can use the `GetADBInfo` function to get information about an ADB device, specified by its ADB address.

```
FUNCTION GetADBInfo (VAR info: ADBDataBlock;
                     adbAddr: ADBAddress): OSErr;
```

`info` An ADB data block. On exit, the fields of this parameter block are filled with information about the specified ADB device.

`adbAddr` The ADB address of a device.

Parameter block

←	`devType`	`SignedByte`	The device handler ID.
←	`origADBAddr`	`SignedByte`	The device's default ADB address.
←	`dbServiceRtPtr`	`Ptr`	The address of the device's device handler.
←	`dbDataAreaAddr`	`Ptr`	The address of the device handler's data storage area.

DESCRIPTION

The `GetADBInfo` function returns, through the `info` parameter, information from the ADB device table entry of the device whose ADB address is specified by the `adbAddr` parameter.

SPECIAL CONSIDERATIONS

Once the ADB Manager has set the initial values for an ADB device in the ADB device table, it updates the device table entry for the device to reflect changes only to the address of the device handler routine and data area pointer.

RESULT CODES

`noErr`	0	No error

Setting ADB Device Information

You can call the ADB Manager function `SetADBInfo` to set or reset some of the information in the ADB device table about an ADB device.

SetADBInfo

You can use the `SetADBInfo` function to set the address of the device handler routine and data area address for a specified ADB device.

```
FUNCTION SetADBInfo (VAR info: ADBSetInfoBlock;
                     adbAddr: ADBAddress): OSErr;
```

`info` An ADB information block. On entry, the fields of this parameter block should contain the desired address of the device handler routine and data area.

`adbAddr` The ADB address of a device.

Parameter block

→	siServiceRtPtr	Ptr	The address of the device handler for this device.
→	siDataAreaAddr	Ptr	The address of the handler's data area for the device at the specified address.

DESCRIPTION

The `SetADBInfo` function sets the device handler address and the data area address in the ADB device table entry whose address is specified by the `adbAddr` parameter.

IMPORTANT

You should send a Flush command to the device after calling it with `SetADBInfo` to avoid sending old data to the new data area address. ▲

RESULT CODES

`noErr`	0	No error

SEE ALSO

See "ADB Information Block," beginning on page 5-38, for the structure of the ADB information block.

Application-Defined Routines

This section describes device handlers and ADB completion routines. A device handler is a low-level routine that communicates with a particular ADB device. An ADB completion routine is a routine that you can provide as a parameter to the `ADBOp` function.

ADB Device Handlers

The ADB Manager automatically polls for input from connected ADB devices and passes any data received from a device to the device's device handler. ADB device handlers are responsible for processing all input from ADB devices (except for commands sent to an ADB device using `ADBOp` or commands sent by the ADB Manager during address resolution).

MyDeviceHandler

Whenever an ADB device sends data (for example, in response to a Talk Register 0 command), the ADB Manager calls the device handler associated with that ADB device. You can provide a device handler to handle data from your ADB device.

```
PROCEDURE MyDeviceHandler;    {parameters passed in registers}
```

DESCRIPTION

When the ADB Manager calls a device handler, it passes parameters to the device handler in registers A0, A1, A2, and D0, as described next.

SPECIAL CONSIDERATIONS

ADB device handlers are always called at interrupt time; they must follow all rules for interrupt-level processing as described in *Inside Macintosh: Processes*.

ASSEMBLY-LANGUAGE INFORMATION

On entry to your device handler, the ADB Manager passes parameters in the following registers:

Register	Value
A0	A pointer to the data sent by the device. This area contains data stored as a Pascal string (maximum 8 bytes data preceded by one length byte).
A1	A pointer to the device handler routine.
A2	A pointer to the data area (if any) associated with the device handler.
D0	The ADB command number (byte) that resulted in the device handler being called.

A device handler should handle the data pointed to by register A0 in a manner appropriate to the device. For example, the mouse device handler interprets the data it receives in register A0 and posts an event to the event queue.

A device handler can use the area pointed to by register A2 to store global data as needed. (If a device handler needs a global data area, its installation code should allocate the needed memory at the same time it installs the device handler's address into the ADB device table.)

SEE ALSO

See "Writing an ADB Device Handler," beginning on page 5-29, for information on installing and creating an ADB device handler.

ADB Command Completion Routines

The `ADBOp` function is always executed asynchronously; if the bus is busy, the ADB command passed to `ADBOp` is held in a command queue until the bus is free. The result code returned by `ADBOp` indicates only whether the ADB command was successfully placed into the ADB command queue, not whether the command itself was successful.

Thus, when you use the `ADBOp` function to send a command to an ADB device, and your application requires synchronous behavior, you'll need to provide a completion routine to determine when the command has completed.

MyCompletionRoutine

When you use the `ADBOp` function to send a command to an ADB device, the ADB Manager calls your completion routine when the ADB device has completed the command.

```
PROCEDURE MyCompletionRoutine;   {parameters passed in registers}
```

DESCRIPTION

The ADB Manager passes parameters to a completion routine in registers A0, A1, A2, and D0, as described next.

ASSEMBLY-LANGUAGE INFORMATION

On entry to your completion routine, the ADB Manager sets the following registers:

Register	Value
A0	A pointer to the data area specified by the `buffer` parameter to the `ADBOp` function. This area contains data stored as a Pascal string (maximum 8 bytes of data preceded by one length byte). For example, data returned by an ADB device in response to a Talk command issued by a call to the `ADBOp` function can be accessed through this pointer.
A1	A pointer to the completion routine.
A2	A pointer to the data area that was specified by the `data` parameter to the `ADBOp` function. Your completion routine can use this area to store information; for example, to set a flag indicating that the command has completed.
D0	The ADB command number (byte) that resulted in the completion routine being called.

SEE ALSO

See Listing 5-2 on page 5-25 for an example of a completion routine.

Summary of the ADB Manager

Pascal Summary

Data Types

```
TYPE ADBDataBlock =
   PACKED RECORD
      devType:          SignedByte;    {device handler ID}
      origADBAddr:      SignedByte;    {default ADB address}
      dbServiceRtPtr:   Ptr;           {pointer to device handler}
      dbDataAreaAddr:   Ptr;           {pointer to data area}
   END;
   ADBDBlkPtr = ^ADBDataBlock;

TYPE ADBSetInfoBlock =
   RECORD
      siServiceRtPtr:   Ptr;           {pointer to device handler}
      siDataAreaAddr:   Ptr;           {pointer to data area}
   END;
   ADBSInfoPtr = ^ADBSetInfoBlock;

TYPE ADBOpBlock =
   RECORD
      dataBuffPtr:      Ptr;           {address of data buffer}
      opServiceRtPtr:   Ptr;           {pointer to device handler}
      opDataAreaPtr:    Ptr;           {pointer to optional data area}
   END;
   ADBOpBPtr = ^ADBOpBlock;

   ADBAddress = SignedByte;
```

ADB Manager Routines

Initializing the ADB Manager

```
PROCEDURE ADBReInit;
```

Communicating Through the ADB

```
FUNCTION ADBOp              (data: Ptr; compRout: ProcPtr; buffer: Ptr;
                             commandNum: Integer): OSErr;
```

Getting ADB Device Information

```
FUNCTION CountADBs:         Integer;
FUNCTION GetIndADB          (VAR info: ADBDataBlock;
                             devTableIndex: Integer): ADBAddress;
FUNCTION GetADBInfo         (VAR info: ADBDataBlock;
                             adbAddr: ADBAddress): OSErr;
```

Setting ADB Device Information

```
FUNCTION SetADBInfo         (VAR info: ADBSetInfoBlock;
                             adbAddr: ADBAddress): OSErr;
```

Application-Defined Routines

```
PROCEDURE MyDeviceHandler;

PROCEDURE MyCompletionRoutine;
```

C Summary

Data Types

```
typedef char ADBAddress;

struct ADBDataBlock {
   char        devType;               /*device type*/
   char        origADBAddr;           /*original ADB address*/
   Ptr         dbServiceRtPtr;        /*pointer to device handler*/
   Ptr         dbDataAreaAdd;         /*pointer to data area*/
};
typedef struct ADBDataBlock ADBDataBlock;
typedef ADBDataBlock *ADBDBlkPtr;

struct ADBSetInfoBlock {
   Ptr         siServiceRtPtr;        /*pointer to device handler*/
   Ptr         siDataAreaAddr;        /*pointer to data area*/
};
typedef struct ADBSetInfoBlock ADBSetInfoBlock;
```

```
typedef ADBSetInfoBlock *ADBSInfoPtr;

struct ADBOpBlock {
   Ptr         dataBuffPtr;            /*address of data buffer*/
   Ptr         opServiceRtPtr;         /*pointer to device handler*/
   Ptr         opDataAreaPtr;          /*pointer to optional data area*/
};
typedef struct ADBOpBlock ADBOpBlock;
typedef ADBOpBlock *ADBOpBPtr;
```

ADB Manager Functions

Initializing the ADB Manager

```
pascal void ADBReInit       (void);
```

Communicating Through the ADB

```
pascal OSErr ADBOp          (Ptr data, ProcPtr compRout, Ptr buffer,
                             short commandNum);
```

Getting ADB Device Information

```
pascal short CountADBs      (void);
pascal ADBAddress GetIndADB
                            (ADBDataBlock *info, short devTableIndex);
pascal OSErr GetADBInfo     (ADBDataBlock *info, ADBAddress adbAddr);
```

Setting ADB Device Information

```
pascal OSErr SetADBInfo     (ADBSetInfoBlock *info, ADBAddress adbAddr);
```

Application-Defined Functions

```
pascal void MyDeviceHandler (void);
pascal void MyCompletionRoutine (void);
```

Assembly-Language Summary

Data Structures

ADB Data Block

0	`devType`	byte	device type
1	`origADBAddr`	byte	original ADB address
2	`dbServiceRtPtr`	long	pointer to completion routine
6	`dbDataAreaAddr`	long	pointer to data area

ADB Information Block

0	`siServiceRtPtr`	long	pointer to completion routine
4	`siDataAreaAddr`	long	pointer to data area

ADB Operation Block

0	`dataBuffPtr`	long	address of data buffer
4	`opServiceRtPtr`	long	pointer to completion routine
8	`opDataAreaPtr`	long	pointer optional data area

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
`ADBReInit`	`_ADBReInit`
`ADBOp`	`_ADBOp`
`CountADBs`	`_CountADBs`
`GetIndADB`	`_GetIndADB`
`GetADBInfo`	`_GetADBInfo`
`SetADBInfo`	`_SetADBInfo`

Global Variables

`JADBProc`	long	Pointer to `ADBReInit` preprocessing/postprocessing routine.
`KbdLast`	byte	ADB address of the keyboard last used.
`KbdType`	byte	Keyboard type of the keyboard last used.

Result Codes

`noErr`	`0`	No error
`errADBop`	`-1`	Unsuccessful completion