New Technical Notes

Macintosh

Developer Support

Space Aliens Ate My Mouse (ADB–The Untold Story)

Hardware **M.HW.ADB**

Revised by: Rich Kubota October 1991
Written by: Cameron Birse August 1988

This Technical Note explains how the Apple Desktop Bus (ADB) works on the Macintosh. This Note covers the boot process, driver installation, ADB Manager run-time behavior, use of ADB Manager calls, and answers commonly asked questions.
Changes since February 1990: Added description of the boot process to include detail on how the ADBS resource gets called by the System, added detail to 2 of the answers in the Q&A section, and added sample completion routines for the ADBOp function.

Boot Process

During the boot process, the ADB Manager finds all the devices on the bus and resolves any address conflicts. An address conflict is defined as two or more devices with the same original (default) address. A good example of this conflict is a mouse and a graphics tablet that are both at address 3 (relative device). The ADB Manager resolves these address conflicts as described in Appendix B of the ADB Specification (Apple Drawing #062-0267-E) and the Q & A section of this document.

After the address resolution, the devices which have been “moved” due to address conflicts are addressed, starting from the highest unused soft address and working down. The system now loads and executes all the resources of type `'ADBS'` that match the devices on the bus (by original address).

Once all the ADB service routines are installed, the ADB transceiver (microcontroller) chip starts polling the active device. The active device is defined as the last device to send data. Since the mouse (pointing device) is the most likely device to have data ready at any given time, it defaults as the active device after startup.

The transceiver polls the active device (approximately every 10-16 milliseconds, do **not** depend on this interval), with a Talk R0 command. If the active device has new data, it can respond with it, and if it does not, it just times out. If any other devices have data to send, they can assert SRQ (refer to Figure 5 of the ADB Specification) at the end of the Talk R0 command. When the host detects an SRQ, it begins polling all addresses with a Talk R0

command until one returns data. That device then becomes the active device.

Devices have no way of knowing if they are the “active device”. The algorithm for a device with data ready to send is as follows:

- Wait for a Talk R0 command.
- If the Talk R0 is for you, then return the data.
- If the Talk R0 is not for you, wait for the end of the command, and assert SRQ.
- If the Talk R0 is addressed to you, then respond with your data.

Now that a device has been polled, the host retrieves the data from the bus and calls the service routine installed for that device (service routines are installed by calling `_SetADBInfo` and are maintained by the ADB Manager). The system passes pointers to the service routine itself, its data area, and the data received from the device, as well as the ADB command byte that caused the routine to be called.

Normally, the service routine does not need to use the `_ADBOp` call to retrieve data. The ADB “philosophy” assumes that register zero of a device is the main data transmission register. Since register zero is automatically polled by the system, there should be no need to call `_ADBOp` from the service routine. Typically, `_ADBOp` is used to set modes of a device, or to interrogate the device for status—the sort of things that should not need to be done more than once or twice during normal operation.

It is important to note that ADB service routines are called at interrupt time, which means that they must follow all the rules regarding code that executes at interrupt time. (See *Inside Macintosh* references to VBL tasks and Device Manager I/O completion routines.)

Installing an ADB Service Routine and Optional Data Area

At boot time the system searches for `'ADBS'` resources in the System file. The system matches desktop bus devices by their original address to an `'ADBS'` resource (i.e., if the machine has a device that responds at address 4, the system looks for an `'ADBS'` resource with ID=4). The limitation of this method is that there can only be one `'ADBS'` resource for each address on the bus.

When the system finds these resources, it loads, detaches, and executes them. The System loads in each ADBS driver with a `_GetResource` call. If successful, the System calls _DetachResource on the handle to the ADBS driver. The registers are set up as described below, and a JSR (A0) call is made to execute the resource. It is the responsibility of the driver to dispose of itself if a failure occurs.

If there is insufficient memory in the System heap to load the resource, the ADBS will not be executed, and the System continues on to the next ADBS resource. For this reason, the size of the ADBS resource should be kept as small as possible. This condition should only occur under System 6.0.x and earlier.

A typical 'ADBS' resource allocates space in the system heap for its service routine and, optional, data area. Next, it moves the service routine into the allocated space and initializes the data area, if necessary. This code should also install an _ADBReInit preprocessing routine to deallocate the memory used by the service routine (*Inside Macintosh* V-367).

When the system loads and executes an 'ADBS' resource, it passes the following parameters:

A0 = Address of 'ADBS' resource in memory.

D0 = ADB device address (0-15). This address may be different than the "original address," since it occurs after address resolution.
D1 = ADB Device Type (same as the handler ID)

With this information, the 'ADBS' code can call _SetADBInfo to install the service routine and data area. The installer should make sure the handler ID (Device Type) is the one it expects.

Note: Previous versions of this note advised using an 'INIT' resource as an alternative method for installing ADB service routine. Apple no longer advises this method. ADB service routines should only be installed by an 'ADBS' resource located in the System file (see MacDTS Sample Code #17, TbltDrvr for an example). 'INIT' resources and application-based installation methods do not work on the Macintosh Portable, because the bus and ADB Manager may be re-initialized after waking up. Part of the re-initialization process loads and executes the 'ADBS' resources associated with the devices present on the bus. If a service routine is installed using the 'INIT' or application method, it does not get re-installed when the Macintosh Portable wakes up.

General ADB Manager Run-Time Behavior

Since the implementation of the ADB Manager on Macintosh CPUs has varied slightly, it's useful to know what behavior to expect, and what not to depend upon. System Tools disks after 6.0.4 make the ADB Manager consistent on all Macintosh models.

Address Resolution

It is important that devices implement the collision detection and address moveability to prevent possible conflicts between devices that have the same default address.

Auto-Polling

All devices on the bus should expect, and be able to handle, being auto-polled. If they do not have a reason to respond, they should simply ignore the poll (time out). If the ADB Manager auto-polls a device which has no service routine installed, it simply throws away any data it may have gotten from the device. The Macintosh SE, SE/30, II, IIx, and IIcx implementations of the ADB Manager only auto-polls devices that have previously responded to an auto-poll, or that have requested service (by asserting SRQ). In addition, the Macintosh IIci and Portable implementations also auto-poll the last device addressed by an _ADBOp command—regardless of whether they have a service routine installed (via the _SetADBInfo call).

System Tools disks later than 6.0.4 patch the ADB Manager so that if the device does not have a service routine installed (with _SetADBInfo), it should not get auto-polled. In the

unlikely case that SRQ is active and none of the devices with routines installed respond, the ADB Manager polls all devices (every address) trying to clear the SRQ. This case is why all devices on the bus should expect, and be able to handle, being auto-polled.

Note: `_ADBOp` commands always have priority over auto-polling and SRQ polling. Whenever there are pending `_ADBOp` commands in the command queue, they are executed before the host resumes auto-polling. Therefore, applications should not issue `_ADBOp` commands repeatedly, keeping the command queue

full. Doing so results in effectively "locking up" the mouse and keyboard, which rely upon the auto-polling and SRQ polling to provide user input.

SRQ Polling

Since the ADB Manager may be polling any device on the bus when an SRQ happens, an application should not rely upon the sequence in which it polls devices. Instead, simply remember that the ADB Manager polls all devices, in turn, on the bus until SRQ is no longer asserted. After SRQ has been satisfied, the ADB Manager begins auto-polling the last device from which it got data.

ADB Manager Bugs 'n Fixes

_ADBOp Talk Command

Through System Software 6.0.4, there is a bug in the Macintosh SE, SE/30, II, IIx, and IIcx implementations of the ADB Manager where the count byte returned by an `_ADBOp` Talk command that timed out is not set to zero to reflect that no bytes were transferred. In addition, the two bytes following it are both $FF (these should be ignored). On the Macintosh IIci and Portable implementations, the count byte for a time out is zero, and any bytes which follow it should, of course, be ignored. This bug is fixed in System Tools disks after 6.0.4.

_ADBOp Listen Command

There is also a bug in the Macintosh SE, SE/30, II, IIx, and IIcx implementations where the number of bytes transferred was one off from the supplied count byte. On the Macintosh IIci and Macintosh Portable implementations the number of bytes transferred is what the count byte specifies. This bug is fixed in System Tools disks after 6.0.4.

There is a bug in the Macintosh Portable where all Listen commands with a data count greater than six send garbage in the seventh and eighth bytes. This bug is fixed in System Tools disks after 6.0.4.

_ADBOp Completion Routines

There is yet another bug in the Macintosh SE, SE/30, II, IIx, and IIcx implementations of the ADB Manager where the completion routines passed to the `_ADBOp` routine are not always called. This bug is not present in the Macintosh IIci and Macintosh Portable implementations and is fixed in System Tools disks after 6.0.4.

_ADBReInit

Inside Macintosh, Volume V states that `_ADBReInit` should be called with a device is

added to the bus while the system is running. This statement is misleading. Do **not** attach devices of **any** kind to a Macintosh while the power is on. If there is a device that can be added to the bus via software (i.e., a device is already attached, and an additional "virtual" device can be added under software control), then it may be useful to call `_ADBReInit`, but it is not absolutely necessary. Devices can be added by simply installing a service routine for the appropriate address using the `_SetADBInfo` call.

However, if you do plan on using `_ADBReInit`, then you should know about the following bug with keyboard layouts (`'KCHR'` resources) other than the standard U.S. layout (`ID =`

0). Most international systems use alternate 'KCHR' resources and may permit switching between them. On these systems, when _ADBReInit is called, it does not reinstall the current 'KCHR' resource, but instead reinstalls the default U.S. 'KCHR' resource (ID = 0). This problem is evident on the Macintosh Portable, since it may call 7/30/24 when it wakes up. This bug is fixed in Systems Tools disks after 6.0.4.

Users can fix this problem by toggling the keyboard mapping selection in the Control Panel. From an application, one could install an _ADBReInit post-processing routine (in the low-memory variable jADBProc, see *Inside Macintosh*, Volume V, The Apple Desktop Bus, pp. 367-368), which reinstalls the correct 'KCHR' resource using the Script Manager _GetEnvirons and _KeyScript calls (see Technical Note #160, Key Mapping) after a call to _ADBReInit.

```
KeyScript(INTEGER(GetEnvirons(smKeyScript)));
```

This code makes a _KeyScript call with the current keyboard script (as described in the 'itlb' system resource). The 'KCHR' and 'SICN' IDs for that script are already setup in the 'itlb' resource and in the appropriate script's local variables. For an example of a jADBProc, see MacDTS Sample Code #17, TbltDrvr.

Answers to Commonly-Asked ADB Questions

Question: I need information on developing an Apple Desktop Bus product. (Hey, that's not a question!)

Answer: Apple's Desktop Bus and ADB Device Specifications are a licensable product available through Software Licensing. For more information, contact:

Apple Software Licensing
Apple Computer, Inc.,
20525 Mariani Avenue, M/S 38-I
Cupertino, CA, 95014
(408) 974-4667
AppleLink: Sw.License
Internet: Sw.License@AppleLink.Apple.com

Additional ADB references are as follows:

Macintosh

Inside Macintosh, Volume V, The Apple Desktop Bus
Macintosh Family Hardware Reference

Apple II

Apple IIGS Hardware Reference Manual

Desktop Bus

Apple IIGS Firmware Reference Manual

General

Baum, Peter.“Boarding the Bus,” *MacUser*, July 1987, p. 142.

“An Overview of Apple Desktop Bus,”

Call A.P.P.L.E., June 1987, p. 24.

Question: I would like to extend the keyboard cable for my Macintosh. How can I do this, and how can I make the extension?

Answer: The ADB specification states the maximum length of all cables on the Desktop Bus is five meters. If you wish to use longer cables than those supplied with the ADB device, Kensington MicroWare (800) 535-4242, Monster Cable (800) 331-3755, and Data Spec (800) 431-8124 all supply them.

Disclaimer: This listing for Kensington MicroWare, Monster Cable, and Data Spec neither implies nor constitutes an endorsement by Apple Computer, Inc. If your company supplies these cables and you would like to be listed, contact us at the address in Technical Note #0.

Question: How can I use the LEDs on the Apple Extended Keyboard?

Answer: Using the LEDs on the extended keyboard involves the `_ADBOp` call. Once you determine that you have an extended keyboard (with `_CountADBs` and `_GetIndADB`), then register 2 of the extended keyboard has the LED toggles in the low 3 bits of the second data byte.

Therefore, you would do a Talk to register 2 to have the device send you the contents of register 2, manipulate the low three bits to set the LEDs, and then pass the modified register 2 back to the device with a Listen to register 2 command.

The Apple Extended Keyboard has an ID of 02 and a device handler ID of 02, while the Apple Standard Keyboard has an ID of 02 and a device handler ID of 01.

Note: At this point it is not clear what Apple has in mind for these LEDs, so you are using them at your own risk.

Question: I am confused about the service routines and data areas passed in the `_ADBOp` call. What does it all mean?

Answer: That 's a good question.

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

`data` is a pointer to the "optional data area". This area is provided for the use of the service routine (if needed).

`compRout` is a pointer to the completion or service routine to be called when the `_ADBOp` command has been completed -i.e sent to the device. Since the ADBOp function is always called asynchronously, the completion routine can be used to flag call completion. Note that the function result of ADBOp indicates whether the call was successfully placed on the command queue - not whether the command has been sent to the device.

`buffer` is a pointer to a Pascal string, which includes a length byte followed by zero to eight bytes of information. These are the zero to eight bytes that a particular register of an ADB device is capable of sending and receiving.

commandNum is an integer that describes the command to be sent over the bus.

There is some confusion over the way that the completion routines are called from _ADBOp. This calling may be done in one of the following three ways:

- You do not wish to have a completion routine called, as in a Listen command. Pass a NIL pointer to _ADBOp.
- You wish to call the routine already in use by the system for that address (as installed by _SetADBInfo). Call _GetADBInfo before calling _ADBOp, and pass the routine pointer returned by _GetADBInfo to _ADBOp.
- You wish to provide your own completion routine and data area for the _ADBOp call. Note that the ADBOp call is always called asynchronously. In this case, simply pass your own pointers to the _ADBOp call.

The following Pascal code demonstrates a method to synchronously call ADBOp. This routine is useful for Talk commands, where the driver needs to wait for the device to return data. CallADBOp accepts the buffer and commandNum parameters and sets up a short word variable "done" as a flag variable. Initially, the flag is set to zero. CallADBOp calls ADBOp passing a pointer to the flag and to the completion routine, "CompRoutine", in addition to the buffer and commandNum parameters. The completion routine simply changes the value of the flag to -1. After calling ADBOp, the CallADBOp function enters a while loop waiting for the flag "done" to change to some non-zero value.

```
PROCEDURE SetA2;
    INLINE $34BC,$FFFF; { MOVE.W  #$FFFF,(A2)                          }
                        { A2 points to the variable - done - our compl flag.}
                        { Upon entry, the flag is set to zero.  Set value       }
                        { to non-zero, -1 used here, to indicate completion }

PROCEDURE CompRoutine;  { Sample ADBOp completion routine              }

BEGIN
    SetA2;              { Set 2 byte area pointed to by A2 to non-zero  }
END;

FUNCTION CallADBOp(buffer: Ptr; cmdNum: INTEGER): OSErr;
{ Modified version of the ADBOp function which takes the same arguments as     }
{ ADBOp except for completion routine ProcPtr.  Calls ADBOp asynchronously     }
{ then waits until the completion routine modifies "done" parameter.   }

VAR
    done:  INTEGER;
    temp:  LONGINT;
    err:   OSErr;

BEGIN
    done := 0;
    err := ADBOp(@done, @CompRoutine, @buffer, cmdNum);
    IF err = noErr THEN { request successfully queued                  }
       REPEAT
          {Delay(2, temp);}   { uncomment this line as noted below     }
```

```
            { For some time critical operations, the use of Delay procedure       }
            { has proven useful with Talk commands towards allowing the   }
            { device to complete the command.                             }
        UNTIL (done <> 0);
    CallADBOp := err;       { 0 command entered into command queue.       }
                            { -1   command queue full, unsuccessful completion.      }
END;
```

The following is the same example in C.

```
void SetA2(void)
    = {0x34BC,0xFFFF;       { MOVE.W                                    #$FFFF,(A2)
    }
                            { A2 points to the variable - done - our compl flag.}
                            { Upon entry, the flag is set to zero.  Set value       }
                            { to non-zero, -1 used here, to indicate completion }

void CompRoutine(void)  { Sample ADBOp completion routine               }

{
    SetA2;              { Set 2 byte area pointed to by A2 to non-zero  }
}

OSErr CallADBOp(Ptr buffer, short cmdNum)
{ Modified version of the ADBOp function which takes the same arguments as      }
{ ADBOp except for completion routine ProcPtr.  Calls ADBOp asynchronously      }
{ then waits until the completion routine modifies "done" parameter.    }

{
    short  done;
    long   temp;
    OSErr  err;

    done = 0;
    err = ADBOp(&done, CompRoutine, buffer, cmdNum);
    if (err == noErr)       { request successfully queued               }
    {
        do
            {Delay(2, temp);}   { uncomment this line as noted below      }
            { For some time critical operations, the use of Delay procedure       }
            { has proven useful with Talk commands towards allowing the   }
            { device to complete the command.                             }
        while (done != 0);
    }
    return (err};       { 0 command entered into command queue.         }
                        { -1   command queue full, unsuccessful completion.      }
}
```

Remember, there should rarely be a reason to call `_ADBOp`. Most cases are handled by the system's polling and service request mechanism. In the cases where it is necessary to call `_ADBOp`, it should not be done in a polling fashion, but as a mechanism of telling the device something (i.e., change modes, or in the case of our extended keyboard, turn on or off an LED).

Question: How can I make my Macintosh II or IIx power up automatically after a power outage?
Answer: The Macintosh II and IIx power can be turned on via the keyboard through the Apple Desktop Bus port (ADB) since the reset key is wired to pin two of the ADB connector. When you press this key, it pulls pin two to ground and initiates a power-on sequence. You can emulate this feature with a momentary switch connected to the ADB port. Note that the switch on the back panel of a Macintosh IIcx and later Macintosh II models, can be locked in the On position to automatically restart after a power outage

An idea for a power-on circuit would be to have a momentary (one-shot) relay powered by the same outlet that powers the machine and have the contacts close pin two of the ADB connector. (Without having tried this, I am concerned that you may need a delay before the relay fires to give the AC time to stabilize, etc.)

Question: I'm more than a little confused about the way ADB device address conflicts are resolved at boot time.
Answer: The method used by the host to separate and identify the devices at boot time is not well documented, so I'll try to describe it with some clarity.

The host issues a Talk R3 command to an address. Let's say there are two devices at that address. Both try to respond to the command, and when they try to put the random number (the address field of register 3) on the bus, one of them should detect a collision. The one that detects the collision backs off and marks itself (internally) as unmovable.

The device that did respond successfully is then told to move to a new address (the highest free address). By definition, moving to a new address means that it now responds only to commands addressed to this new address, and it ignores commands to the original address.

The host then issues another Talk R3 command to the original address. This time the second device responds without detecting a collision. When it successfully completes a Talk R3 response, it marks itself as movable. It then is told to move to a new address.

The host again issues a Talk R3 command to the original address. Since there are no more devices at that address, the bus times out, and the host moves the last device back to the original address.

At this point, the host moves up to the next address that has a device and begins the process all over.

Generally, when having trouble separating devices on the ADB, it is because the collision detection doesn't work well. In fact, this problem is evident on Apple keyboards. The bug is that the random number returned in R3 isn't really a random number. Since the microcontrollers on the keyboards are clocked with a crystal, they tend to generate the same "random" number, so when the system attempts to separate them with a Talk R3 command, they never detect the collision.

One possible solution is to use a low-tolerance capacitor on the reset line of the microcontroller, thereby forcing the time from power on to the time reset is negated to be fairly random. In this way, the microcontroller can start a count until it receives the first Talk R3 command, and hopefully it is a different number than another device at the same address on the bus.

If you find your device shows up at all addresses, it may be because it is responding to the move address command when it should be marked as unmovable.

Finally, if the device doesn't show up at all, it may be because it is unable to respond to the Talk R3 command at boot time (i.e., not able to initialize itself and start watching the bus in time).

Further Reference:

- *Inside Macintosh*, Volume V, Apple Desktop Bus
- *Inside Macintosh*, Volume V, The Script Manager
- *The Script Manager 2.0*, Interim Chapter (DTS)
- *Macintosh Family Hardware Reference*, Chapters 11 & 19
- Technical Note M.TB.KeyMapping — Key Mapping
- MacDTS Sample Code #17, TbltDrvr