Technote 1119

Serial Port Apocrypha

By Quinn "The Eskimo!"
Apple Developer Technical Support
devsupport@apple.com

CONTENTS

This Technote describes a number of problems often encountered by developers when dealing with serial ports under Mac OS. Most of this information is available from other sources, but those sources are obscure and commonly overlooked.

Specifically, this Note describes the correct techniques for finding, opening, closing, and yielding serial ports under the classic serial API and the Open Transport serial API. In addition, this Note describes the theory and practice of the original and Open Transport serial port arbitrators.

This Technote is directed at all Mac OS developers who use serial ports.

Notes for Both APIs

Mac OS provides two APIs for accessing the serial port: the classic serial API based on Device Manager `'DRVR'`s, described in *Inside Macintosh:Devices*, and the Open Transport serial API, described in *Inside Macintosh: Open Transport*. This section contains notes which are relevant to both serial APIs.

Open and Close on Demand

A serial port is a non-sharable resource. If your application has the port open, no other application can open it. For this reason, you should always open and close the serial port on demand.

For example, if your application only uses the serial port as part of its registration process, you open the port when you commence the registration and close the port immediately after you are done.

Yielding

Yielding is the process by which a passive serial program can yield the serial port to an active serial program, and regain the serial port after the active serial program is done.

For example, if you set Apple Remote Access (version 2.1 and lower) to wait for an incoming call, you can still make outgoing PPP connections using FreePPP. This is because the passive serial program (ARA) yields the serial port to the active serial program (FreePPP). When FreePPP closes the serial port, ARA will resume ownership and continue waiting for an incoming call.

Just The Facts: Classic Serial

The classic serial architecture is based on Device Manager `'DRVR's`, as described in *Inside Macintosh:Devices*. This section describes the correct way to find, open, close, and yield serial ports under the classic serial architecture.

Finding All Serial Ports

The correct way to find all the serial ports under Mac OS is to use the Communications Resource Manager (CRM) routine `CRMSearch` (part of the Communications Toolbox). Unfortunately, the book that documents the Communications Resource Manager (*Inside the Macintosh Communications Toolbox*) is not available in electronic form, so it can be hard to find documentation for `CRMSearch`. The following sample is included to make up for this deficiency:

```
static void PrintInfoAboutAllSerialPorts(void)
    // Prints a list of all the serial ports on the
    // machine, along with their corresponding input
    // and output driver names, to stdout.  Typically
    // you would use a routine like this to populate a
    // popup menu of the available serial ports.
{
    CRMRec          commRecord;
    CRMRecPtr       thisCommRecord;
    CRMSerialPtr    serialPtr;


    (void) InitCRM();

    // First set up commRecord to specify that
    // we're interested in serial devices.

    commRecord.crmDeviceType = crmSerialDevice;
    commRecord.crmDeviceID   = 0;

    // Now repeatedly call CRMSearch to iterate
    // through all the serial ports.

    thisCommRecord = &commRecord;
    do {
        thisCommRecord = (CRMRecPtr) CRMSearch( (CRMRecPtr) thisCommRecord );
        if ( thisCommRecord != nil ) {

            // Once we a have a CRMRec for the serial port,
            // we must cast the crmAttributes field to
            // a CRMSerialPtr to access to serial-specific
            // fields about the port.

            serialPtr = (CRMSerialPtr) thisCommRecord->crmAttributes;

            // Print the information about the port.

            printf("We have a port called: "%#s"\n", *(serialPtr->name));
            printf("   input driver named: "%#s"\n", *(serialPtr->inputDriverName));
            printf("  output driver named: "%#s"\n", *(serialPtr->outputDriverName));
            printf("\n");

            // Now ensure that CRMSearch finds the next device.

            commRecord.crmDeviceID = thisCommRecord->crmDeviceID;
        }
    } while ( thisCommRecord != nil );
}
```

IMPORTANT:
Ports registered with the CRM are supposed to "work like" the standard built-in serial ports. However, in some cases (both Apple and third party), it's just not possible to implement the API of the built-in serial ports exactly. When dealing with CRM-registered ports, your application should handle cases where this emulation breaks down. For example, if your application uses the externally clocked quasi-MIDI mode (csCode 15), it should gracefully fail if a serial driver returns an error when asked to engage this mode.

Opening a Serial Port

The correct way to open a serial port has been documented for many years as part of the ARA API document, currently available on the Mac OS SDK Developer CDs. However, this source is somewhat obscure (and the enclosed sample code is somewhat out of date), so the information is repeated here for your convenience.

The process is very easy to describe in English:

> If a serial port arbitrator is installed, always call `OpenDriver` to open the serial port; otherwise, walk the unit table to determine whether the driver is already open, and open it only if it isn't.

This high-level algorithm is captured in the following routines for opening both the input and output serial drivers:

```
static OSErr OpenOneSerialDriver(ConstStr255Param driverName, short *refNum)
    // The one true way of opening a serial driver.  This routine
    // tests whether a serial port arbitrator exists.  If it does,
    // it relies on the SPA to do the right thing when OpenDriver is called.
    // If not, it uses the old mechanism, which is to walk the unit table
    // to see whether the driver is already in use by another program.
{
    OSErr err;

    if ( SerialArbitrationExists() ) {
        err = OpenDriver(driverName, refNum);
    } else {
        if ( DriverIsOpen(driverName) ) {
            err = portInUse;
        } else {
            err = OpenDriver(driverName, refNum);
        }
    }
    return err;
}

static OSErr OpenSerialDrivers(ConstStr255Param inName, ConstStr255Param outName,
                               SInt16 *inRefNum, SInt16 *outRefNum)
    // Opens both the input and output serial drivers, and returns their
    // refNums.  Both refNums come back as an illegal value (0) if we
    // can't open either of the drivers.
{
    OSErr err;

    err = OpenOneSerialDriver(outName, outRefNum);
    if (err == noErr) {
        err = OpenOneSerialDriver(inName, inRefNum);
        if (err != noErr) {
            (void) CloseDriver(*outRefNum);
        }
    }
    if (err != noErr) {
        *inRefNum = 0;
        *outRefNum = 0;
    }
    return err;
}
```

```
enum {
    gestaltSerialPortArbitratorAttr = 'arb ',

        gestaltSerialPortArbitratorExists = 0
};

static Boolean SerialArbitrationExists(void)
    // Test Gestalt to see if serial arbitration exists
    // on this machine.
{
    Boolean result;
    long    response;

    result = ( Gestalt(gestaltSerialPortArbitratorAttr, &response) == noErr &&
                (response & (1 << gestaltSerialPortArbitratorExists) != 0)  != 0)
                );
    return result;
}
```

The final part of the puzzle is the routine `DriverIsOpen`, which walks the unit table to see if the driver serial driver is present and open. Remember that this routine -- which is inherently evil because it accesses low memory globals -- is only used if a serial port arbitrator is not installed.

```
static Boolean DriverIsOpen(ConstStr255Param driverName)
    // Walks the unit table to determine whether the
    // given driver is marked as open in the table.
    // Returns false if the driver is closed, or does
    // not exist.
{
    Boolean     found;
    Boolean     isOpen;
    short       unit;
    DCtlHandle  dceHandle;
    StringPtr   namePtr;

    found = false;
    isOpen = false;

    unit = 0;
    while ( ! found && ( unit < LMGetUnitTableEntryCount() ) ) {

        // Get handle to a device control entry.  GetDCtlEntry
        // takes a driver refNum, but we can convert between
        // a unit number and a driver refNum using bitwise not.

        dceHandle = GetDCtlEntry( ~unit );

        if ( dceHandle != nil && (**dceHandle).dCtlDriver != nil ) {

            // If the driver is RAM based, dCtlDriver is a handle,
            // otherwise it's a pointer.  We have to do some fancy
            // casting to handle each case.  This would be so much
            // easier to read in Pascal )-:

            if ( ((**dceHandle).dCtlFlags & dRAMBasedMask) != 0 ) {
                namePtr = & (**((DRVRHeaderHandle) (**dceHandle).dCtlDriver)).drvrName[0
            } else {
                namePtr = & (*((DRVRHeaderPtr) (**dceHandle).dCtlDriver)).drvrName[0];
            }

            // Now that we have a pointer to the driver name, compare
            // it to the name we're looking for.  If we find it,
            // then we can test the flags to see whether it's open or
            // not.

            if ( EqualString(driverName, namePtr, false, true) ) {
                found = true;
                isOpen = ((**dceHandle).dCtlFlags & dOpenedMask) != 0;
            }
        }
        unit += 1;
    }

    return isOpen;
}
```

NOTE:
The low memory accessor routine `LMGetUnitTableEntryCount` is defined in "LowMem.h" but is not exported by `InterfaceLib`. If you call this routine from CFM code built with Universal Interfaces 2.x, you will get a link error. To work around this, either write your own version of the function which accesses low memory directly, or upgrade to Universal Interfaces 3.x, which defines a C macro to cover this case.

Closing a Serial Port

If you successfully open a serial port, you should make sure to close it again when you're done. You should always use `CloseDriver` to close a serial port. Remember to close both the input and output drivers. The following code illustrates the correct way to close the serial driver:

```
static OSErr CloseSerialDrivers(SInt16 inRefNum, SInt16 outRefNum)
{
    OSErr err;

    (void) KillIO(inRefNum);
    err = CloseDriver(inRefNum);
    if (err == noErr) {
        (void) KillIO(outRefNum);
        (void) CloseDriver(outRefNum);
    }

    return err;
}
```

It's important that you close the serial driver, even if your application quits abnormally. If you fail to close the serial driver when you quit, it will be unavailable for other applications until the computer is restarted.

The following techniques are ways to ensure that you close the serial driver even if your application quits abnormally:

- If you're a CFM application, use your CFM fragment's terminate procedure. See *Inside Macintosh:PowerPC System Software* for details.
- If the Thread Manager is available, set a terminate procedure for your main thread using `SetThreadTerminator`.
- If neither of the above apply, patch `ExitToShell`.

NOTE:
Inside Macintosh II pp. 247-250 discusses the differences between the ROM and RAM serial drivers and the routines `RAMSDOpen` and `RAMSDClose` . This information is obsolete and should be ignored.

Yielding

The classic serial architecture has very limited support for yielding the serial port. Apple Remote Access does this using a private API exported by the Link Tool Manager (part of ARA). This API was never published by Apple, and is not available to third parties.

Open Transport provides a second API for serial on Mac OS, one that has much in common with the network APIs provided by OT. In the current implementation of OT (version 1.3 at the time of writing), the OT serial API is implemented as a shim layered on top of the classic serial drivers. This fact is important because the way you use the OT serial API affects the availability of serial ports to the classic API, and vice versa.

Inside Macintosh: Open Transport contains a lot of background material that you might find useful.

Finding All Serial Ports

If you are using the OT serial API, the correct way to find all the installed serial ports is to repeatedly call `OTGetIndexedPort` looking for all ports of type `kOTSerialDevice`. The following sample demonstrates this technique:

```
static OSStatus PrintSerialPortInfo(const OTPortRecord *portRecord)
    // Prints information about the port with the given portRecord.
{
    Str255 userVisibleName;

    // OTGetUserPortNameFromPortRef is a little known routine
    // from <OpenTptConfig.h> that allows you to get a user
    // visible name for an Open Transport port.

    OTGetUserPortNameFromPortRef(portRecord->fRef, userVisibleName);

    printf("Found a serial port with port reference $%08lx:\n", portRecord->fRef);
    printf("  User visible name is                        "%#s".\n", userVisibleName);
    printf("  String to pass to OTCreateConfiguration is "%s".\n",  portRecord->fPortNam
    printf("  Name of provider module is                  "%s".\n",  portRecord->fModuleN
    printf("\n");

    return kOTNoError;
}

static OSStatus OTFindSerialPorts(void)
    // Lists all of the serial ports on the machine using Open Transport.
{
    OSStatus err;
    Boolean portValid;
    SInt32 portIndex;
    OTPortRecord portRecord;
    UInt16 deviceType;

    // Start portIndex at 0 and call OTGetIndexedPort until
    // we find there are no more ports.

    portIndex = 0;
    err = kOTNoError;
    do {
        portValid = OTGetIndexedPort(&portRecord, portIndex);
        if (portValid) {

            // For each valid port, get the deviceType and, if
            // it's a serial port and not an alias, call PrintSerialPort
            // to dump out its information.  Note that you don't want
            // to include aliases to the serial ports in the list, otherwise
            // a standard machine will have 3 serial ports, "serialA", "serialB"
            // and "serial".

            deviceType = OTGetDeviceTypeFromPortRef(portRecord.fRef);
```

```
            if (deviceType == kOTSerialDevice &&
                        (portRecord.fInfoFlags & kOTPortIsAlias) == 0) {
                err = PrintSerialPortInfo(&portRecord);
            }
        }
        portIndex += 1;
    } while ( portValid && err == kOTNoError);

    return err;
}
```

IMPORTANT:
The `OTGetUserPortNameFromPortRef` routine is not available to 68K programs running on PowerPC computers. See Technical Q&A NW 48 "68K Open Transport Code on Power Macintoshes" for details.

NOTE:
The routine `OTGetUserPortNameFromPortRef` is defined in the"OpenTptConfig.h" header file. This file is not in Universal Interfaces, but it is included in the full OT SDK. Look for it in the "Open Tpt Protocol Developer" folder.

OT 1.1.1 (and later) will automatically register any CRM serial port as an OT serial port, so this technique will see built-in and third party serial ports. You can determine the currently installed version of OT using Gestalt, as described in Q&A NW 41 "Gestalt Selectors for Mac Networking."

Opening the Serial Port

Once you know which serial port to use, you can call `OTOpenEndpoint` to create an endpoint to that serial port. However, OT does not actually open the underlying serial driver until you use that endpoint to make an active or passive connection.

You make an active connection by first calling `OTBind` with a `qlen` of 0 and then calling `OTConnect`. The serial port is not actually opened until you do the `OTConnect`.

You make a passive connection by calling `OTBind` with a `qlen` of 1. The serial port is opened as part of the binding process. Your notifier will receive a `T_LISTEN` event when the first characters arrive at the serial port.

Your program should be sure to register itself as an OT client (using `OTRegisterAsClient`) so that it receives important notifications about the serial port it is using. Specifically, you should be prepared to handle the `kOTYieldPortRequest` notification, as described in the section on yielding. Also, your endpoint's notifier should be prepared to handle the `kOTProviderIsDisconnected` and `kOTProviderIsReconnected` notifications.

Closing the Serial Port

Once you're done with the serial port, you should be sure to close it. The exact point at which the underlying serial port is closed depends on how you opened it.

If you made an active connection, you can close the serial port by disconnecting (taking the endpoint to state `T_IDLE`), typically using the `OTSndDisconnect` routine.

If you made a passive connection, you can close the serial port by using `OTUnbind` to unbind the endpoint.

Of course, if you close the endpoint (using `OTCloseProvider`), the serial port will always be closed.

Unlike classic serial, Open Transport does keep track of which applications are using which serial ports. If your application unexpectedly quits, OT will automatically close all of its endpoints and thereby close any serial ports it had open. However, non-application code (like code resources and shared libraries) must take care to always call `CloseOpenTransport` before they are unloaded from memory.

Yielding

IMPORTANT:
This section describes how the OT serial port yielding process *should* work. Open Transport (version 1.3 at the time of writing) has bugs which prevent this from working in practice. From the application perspective, these bugs result in `OTYieldPortRequest` always returning `kEBUSYErr` even if the passive program allows the request.

Unlike classic serial, Open Transport does have a public API for yielding serial ports. The basic sequence of events runs:

1. The passive program opens the serial port by binding with a `qlen` of 1. The passive program will receive any incoming connections on the serial port.
2. The active program tries to open the serial port by calling `OTConnect`. Because the passive program already has the serial port open, `OTConnect` fails with an error, `kEBUSYErr`.
3. The active program notices this error and calls `OTYieldPortRequest` for that port.
4. OT sends a `kOTYieldPortRequest` notification to any program which has is registered with OT (using `OTRegisterAsClient`) and has an endpoint using that port.
5. The passive program's notification can choose to yield the port, or return an error saying why the request was denied.
6. If any registered client denies the request, the `OTYieldPortRequest` function returns an error and the active program cannot use the serial port. The `OTYieldPortRequest` also returns a list of clients that refused the request and the reasons why. The active program can use this information in its "port in use" error dialog.
7. If all registered clients agree to yield the port, the port is handed over to the active program. The active program has a short period of time (approximately 10 seconds) to open the port (by binding with a `qlen` of 1 or by calling `OTConnect`) before the port reverts back to the original passive program.
8. When the active program opens the port, the passive program receives a `kOTProviderIsDisconnected` notification.
9. When the active program is done with the serial port and closes it, the passive program receives a `kOTProviderIsReconnected`.

This somewhat convoluted process is described in more detail in *Inside Macintosh: Open Transport*.

A Tale of Two Arbitrators

The Serial Port Arbitrator is one of the least understood components of the Mac OS, partly because it is installed by Apple Remote Access and is not a core component of the system. This section explains why serial port arbitration is necessary, and the features of the two serial port arbitrators.

The Original Problem

The original Mac OS Device Manager architecture has an interesting 'quirk' in that, once a driver is opened, any further calls to `OpenDriver` just return the driver's reference number without calling the driver at all. For most types of drivers (e.g. the floppy disk driver) this is a non-issue, but serial drivers can only support one client at a time and serial port ownership is an important user-level concept.

On pre-MultiFinder Macintoshes, this was never a problem because only one program could be running at a time, and presumably it had control of the serial ports. However, with the advent of MultiFinder, multiple applications could be running simultaneously, and so the serial port ownership became an issue.

The Original Solution

The original solution was fairly easy: if the serial port is already open, it must be in use by another application, and hence you should not try to use it. While this requires serial applications to poke around in the unit table, it was a perfectly serviceable solution.

The New Problem

The new problem arose with the advent of Apple Remote Access. ARA has a mode in which it will

passively sit in the background waiting for calls. However, users were annoyed by the fact that ARA was permanently using their serial port (and, more specifically, their modem), so they could not make outgoing calls without first turning off ARA's answer mode.

This problem was hard to get around because of the original solution. A well-behaved application looked in the unit table, noticed that the serial driver was in use, and did not even attempt to call `OpenDriver`. So there was no way that ARA could shut down its answer mode when another application wanted the serial port.

The New Solution

The solution to this new problem was twofold. First, the rules were changed for developers. The new rule is the one described above: if a serial port arbitrator is installed, applications should ignore the unit table and always call `OpenDriver` when they want the serial port.

Second, ARA shipped with the Serial Port Arbitrator . The Serial Port Arbitrator patches `_Open` and `_Close` looking for applications opening and closing the serial port. When an application opened the serial port, Serial Port Arbitrator would tag the serial port as belonging to that application. If another application attempts to open the same serial port, the Serial Port Arbitrator would fail the second open request with a `portInUse` error.

NOTE:
The `portInUse` error returned by the Serial Port Arbitrator is the same error code that the built-in serial driver returns when the serial hardware is being used by some other type of driver, for example by the LocalTalk driver. Although it's the same error code, it's not exactly the same error condition. Serial Port Arbitrator returns `portInUse` when the serial port is being used for serial by some other process. The serial driver returns `portInUse` when the serial hardware is being used by some other driver.

The original Serial Port Arbitrator shipped as part of ARA 1.0. Its operation was intimately tied with the ARA Link Tool Manager. The Link Tool Manager API, which ARA uses to open a serial port in passive mode, was never publically documented.

The Newer Solution

Unfortunately, in computers, stability is death, and this is as true for ARA as it is for any other part of Mac OS. Part of the plan for ARA 3.0 was to get rid of the Link Tool Manager, and its associated Serial Port Arbitrator. However, by the time ARA 3.0 became a reality, developers were used to the Serial Port Arbitrator and were happily calling `OpenDriver` to open the serial port. If the ARA Serial Port Arbitrator went away, they would no longer receive an error when other application were using the serial port, with potentially disastrous consequences.

So ARA 3.0 includes a new serial port arbitrator, the OpenTpt Serial Arbitrator, which includes the serial port arbitration functionality of the original Serial Port Arbitrator. Like the original Serial Port Arbitrator, the OpenTpt Serial Arbitrator patches `_Open` and `_Close` and remembers which applications opened which serial ports. So the rule for how to open serial ports still stands.

NOTE:
Actually, the OpenTpt Serial Arbitrator first made its appearance with OT/PPP 1.0. However, OT/PPP is really just a cut down version of ARA 3.0, so I will consider them the same in this discussion.

NOTE:
What happens when both serial port arbitrators are installed? It's easy to get into this situation with the standard Mac OS 8.0 installer, by installing both the OT/PPP and ARA Client 2.1 software. The answer is that the Serial Port Arbitrator takes precedence over the OpenTpt Serial Arbitrator.

NOTE:
Early versions of the Serial Port Arbitrator and OpenTpt Serial Arbitrator would call the Process Manager from their `_Open` patch without first checking whether the Process Manager was available. The upshot of this is that system extensions that attempted to open the serial driver at startup would crash the system with an Unimplement Trap system error. This bug has been fixed in the latest version of both products.

The Latest Problems

Alas, Mac OS has still to achieve serial port arbitration nirvana. A number of serious deficiencies remain in the OpenTpt Serial Arbitrator:

- Serial port ownership is tagged by `ProcessSerialNumber` (see *Inside Macintosh: Processes* for details) -- This is a problem if you write a serial program that is not an application. For example, say you have a patch on `SystemTask` that opens the serial port, uses it for a few minutes, and then closes it. When you open it, you might be running in application A's context, but when you close it, you might be running in application B's context. This confuses the serial port arbitrator and is generally a problem for system extension authors.
- There is no tie in between classic serial port arbitration (the Serial Port Arbitrator and the OpenTpt Serial Arbitrator) and the OT serial port arbitration API -- This means that if an OT program opens a passive serial connection and a classic client attempts to use the serial port (ie calls `OpenDriver`), the OT program will not be notified to yield the port.
- For the above reason, if you have ARA 3.0 waiting for an incoming call, you can not use Z-Term to make an outgoing connection to a dial-up service.
- The OT arbitration API is currently broken. The good news is that calling the API is safe, so OT developers can still use the API on the assumption that it will eventually be fixed.

This note will be revised as these problems are addressed.

Summary

The Mac OS serial port is a shared resource, and the true owner of this resource -- the user -- gets upset when their serial programs do not play well together. By following the guidelines outlined in this note, your program will correctly find all the serial ports on the machine, use those serial ports in the most co-operative way, and be adored by Macintosh users around the world!

Further References

- *Inside the Macintosh Communications Toolbox*, Apple Computer, Inc., Addison-Wesley, 1991, ISBN 0201577755
- *Inside Macintosh: Devices*
- *Inside Macintosh: Networking with Open Transport*
- *ARA API*, available on the Mac OS SDK Developer CD at the path "MacOS SDK-1:Development Kits (Disc 1):Apple Remote Access API:Documentation:ARA API"
- Technote 1018 "Understanding the SerialDMA Driver"
- DV 25 - Setting Port Speed on a Modem Port
- DV 555 - Serial Driver Q&A
- HW 28 - PowerBook Miscellanea (Cold Serial in the Morning)
- SerDemo sample code
- Open Transport web page

Downloadables

 Acrobat version of this Note (44K)

Acknowledgments

Thanks to Brian Bechtel, Peter Gontier, Bo3b Johnson, Matt Mora, Roger Pantos, Craig Prouse, and George Warner.

Send feedback to devsupport@apple.com
Updated: 15-April-98