TECHNOTE: Understanding the DMA Serial Driver

by Craig Prouse & Godfrey DiGiorgi **(ramarren@apple.com)**

Apple Developer Technical Support (DTS)

This Technote describes the Macintosh DMA Serial Driver, also known as SerialDMA 2.0. The SerialDMA driver is a replacement for the standard set of Macintosh serial device drivers often referred to by their driver names: .AIn, .AOut, .BIn, and .BOut. SerialDMA is applicable only to those Macintosh models which incorporate DMA channels servicing the standard Z8530 Serial Communications Controller.

Every attempt has been made to ensure compatibility with the classic Macintosh serial driver which does not support DMA operation. While some minor behavioral differences exist, the primary differences are:

- significantly enhanced serial throughput capabilities
- longstanding bugs fixed, and
- a few additional features.

Since the SerialDMA driver has undergone a major architectural revision during its lifetime, it is important to understand how it has evolved. This Technote primarily describes the second-generation SerialDMA driver, while the first-generation driver is described largely for context.

You should be familiar with Macintosh device drivers in general, and with the classic Macintosh serial driver in particular. For reference, see *Inside Macintosh: Devices*, chapters 1 and 7.

About the SerialDMA Driver

The SerialDMA driver which shipped in the system software of each DMA-capable Macintosh from the Macintosh Centris 660AV through the Power Macintosh 8100/110 (1993-1994) was Apple's first-generation DMA serial driver. This driver was cursed by some architectural flaws. The flaws exhibited themselves primarily as latency problems, which affected software handshaking and certain kinds of read-write operation.

The original version of SerialDMA is identifiable as Serial Driver version 8. As Apple and several third-parties have discovered, version 8 may simply be unsuitable for some categories of applications.

In response, Apple is now providing a second-generation SerialDMA driver (version 9), which corrects the design flaws in the first version, increases compatibility with the classic serial driver to the greatest extent possible, and optimizes performance to realize the full potential of the available DMA hardware. The effects of this re-architecture include

- a much smaller driver footprint (over 30% smaller) in memory for the 68K version
- native performance in the PowerPC version
- significantly higher reliablility
- better potential throughput, and
- minimal latency in response to read requests and XOn/XOff handshaking characters.

Three overriding goals guided the development of the second-generation SerialDMA driver. Certain of these goals were inherited from the original SerialDMA development effort, but customer feedback warranted a complete reassessment of the weight assigned to each of these goals and their priorities for subsequent releases.

Goal #1: Compatibility

It should be virtually impossible to distinguish SerialDMA 2.0 from the classic serial driver except that certain bugs are fixed and throughput is far superior. The SCC itself is programmed slightly differently when data is transferred by a DMA controller and, as a result, the standard transmit and receive interrupt handlers are rarely invoked. The classic SCC interrupt handler API is fully supported, however. Software which depends on replacing the serial drivers' SCC interrupt handlers while the driver is open will probably not function as expected. The DMA interrupt handler interface is private to Apple Computer, Inc.

On the other hand, completion routines are now called with interrupts masked rather than at deferred task time, which eliminates one source of compatibility problems with ill-behaved client software.

During an asynchronous I/O request, driver clients cannot depend on the parameter block's `ioActCount` field to increment because the processor does not intervene in the transfer.

Goal #2: Responsiveness

The greatest flaw in the first-generation driver was that the DMA controller was not programmed to interrupt at the precise moment when an XOn/XOff character arrived, nor at the precise moment when a read request became satisfied. Instead, there was a semi-customizable timeout latency before responding to such events. The result was very high interrupt latency tolerance at the expense of very poor performance for certain clients more interested in response time than throughput.

The second-generation driver implements sophisticated DMA channel management to arrive at a better compromise between the demands of responsiveness and latency tolerance. Responsiveness should now be indistinguishable from the classic serial driver. Latency tolerance should still provide ample margins.

Goal #3: Performance

Transmission throughput was already nearly optimal, but may have been improved slightly in some implementations as a result of more sophisticated DMA channel management.

Receive bandwidth has received special attention. Receive DMA channel availalability is maintained at the highest possible level consistent with responsive behavior toward driver clients. While exact channel availability characteristics depend on the specifics of the DMA controller, the possibility is very small that the DMA controller will exhaust its transfer count and allow the SCC to overrun.

The bandwidth of memory and typical system interrupt latencies provide for support of the maximum 230.4K bps data stream, provided that exceptional events do not deplete the available DMA resources and provided the client manipulates the driver in an efficient manner. New driver `Control` codes make 115.2K and 230.4K bps modes easily accessible to client software without difficult or risky hacks and workarounds.

Working with the SerialDMA Driver

When looking for a certain function in the serial driver, it is usually correct to simply make the appropriate `Control` or `Status` call and check the result. If a function is not supported, the call returns `controlErr` or `statusErr`. It is only appropriate to make decisions based on the driver version when general driver characteristics are known for a specific driver version in advance. This is difficult since serial drivers can and will be revised without changing the version number. However, Apple guarantees that the first-generation SerialDMA driver is obsolete and will not be maintained further.

For development purposes, the easiest and quickest way to check the serial driver version installed on a system is with the MacsBug `drvr` dcmd. This dcmd displays all the drivers installed on a system with their respective version numbers (if specified).

New Capabilities

Thanks to the high-performance DMA controllers available on the machines supported by SerialDMA, higher serial data rates are possible than ever before. With conscientious use of the enhanced serial driver API and an understanding of issues that may stress the drivers' performance, it is feasible to maintain connections in excess of the traditional 57,600 bps limitation. It should even be possible to sustain connections with 28,800 bps V.34 modems using a 230.4K bps DTE rate.

Please read the section on Additional Details for an explanation of how the client may affect the performance and the potential throughput of the SerialDMA driver.

Identifying the First-Generation Driver

It may be useful to identify the first-generation SerialDMA driver programmatically in applications where this driver is unsuitable due to bugs or unacceptable response times. Note that it is not the DMA nature of the driver which should be implicated but the implementation of the driver itself, so logically the test is not whether the driver uses DMA, but whether it is the first-generation DMA driver.

This driver may be identified in one of two ways. The client may make a `Status` call to the serial driver with a csCode of 9 to retrieve the driver's version. The first-generation driver returns the value 8 in the first byte of csParam. Alternatively, the client may issue a `Control` call with a csCode of 17987 and inspect the result; no other version of the serial driver implements this csCode and therefore only this version returns a result of `noErr` (other versions return `controlErr` by default). This special `Control` call, which was unique to the first-generation SerialDMA driver, allowed customization of the DMA timeout latency between one and 65,535 ticks. The latency must be specified in the first 16-bit csParam word when issuing the `Control` call; a value of 1 is safe and provides best performance although it is somewhat inefficient.

Note
In some cases, applications sensitive to serial driver response time may be made to work with the first-generation driver through the use of this special csCode. Because the second-generation driver is now available, however, this hack is best avoided. ◆

Identifying the Second-Generation Driver

Ideally, of course, all versions of the serial driver should be compatible, which means virtually indistinguishable. It is the stated compatibility goal of SerialDMA 2.0 that no one should know the difference between it and the classic non-DMA Serial Driver or first-generation DMA Serial Driver. But if by chance it is desired to identify specifically the second-generation SerialDMA driver, the only direct method is through examination of the driver version number. This is available to the client as the result of a `Status` call with csCode 9 as desribed above. The second-generation driver returns a value of 9 as its version.

The second-generation SerialDMA driver is the first serial driver to support csCodes to invoke 115.2K bps mode and 230.4K bps mode, so in some sense the driver version could be detected that way, but in the future, other driver versions may also support these calls. Therefore, if these functions are desired, make the `Control` calls without regard to the driver version.

IMPORTANT
Do not make any assumptions based on the numerical sequence of serial driver versions. Each version number is essentially an ID for a driver architecture as opposed to a chronological version number. Version 10, for example, may be a completely different design than SerialDMA, it may or may not support DMA at all, and so forth. ▲

SerialDMA Driver Reference

This section documents control codes new to the SerialDMA driver, relative to the classic non-DMA driver.

Low-Level Routines

The SerialDMA driver supports csCode 15, which was designed to help support MIDI externally clocked data rates. This csCode has been implemented previously in the IOP serial driver of the Macintosh IIfx, Quadra 900, and Quadra 950. It has not been officially documented because a large majority of the installed base does not implement the call.

The SerialDMA driver supports two new csCodes, 115 and 230, by which its `Control` routine can switch the driver to high-speed modes. These csCodes support 115.2K baud and 230.4K baud rates. The correct time to make these calls is after a normal `SerReset` call using some other (lower) baud rate. The reason for this is `SerReset` performs a number of configuration tasks but assumes that the baud rate is a function of the SCC baud rate generator. In order to achieve these two higher speeds, the baud rate generator must be bypassed, using the standard 3.672 MHz SCC clock and one of a very limited number of rate divisors. These csCodes effect the task of bypassing the baud rate generator and setting the clock divisor to achieve the specified rate.

Note
If the SCC is externally clocked on the GPi line rather than using the internal RTxC clock, these two control calls will have the effect of selecting a baud rate as a fraction of the GPi clock rate, where the divisor is 32 for csCode 115 and the divisor is 16 for csCode 230. ◆

Set MIDI Clocking[control code 15]

`csCode` = 15   `csParam` = byte

This call is designed to place the serial driver into a quasi-MIDI mode. It is similar to a `SerReset`, but it always leaves the serial driver in a mode of eight data bits and one stop bit. Hardware handshaking is disabled. Clocking is required externally at the CTS pin. The parameter byte represents the factor by which the external clock frequency exceeds the data rate according to the following table. The rate multiplier is encoded in the two most significant bits of the parameter byte, while all other bits are reserved and should be zero.

Table 1 csCode 15 rate multiplier encodings

Encoding	Rate multiplier	Example
0x00	× 1	
0x40	× 16	250K baud 8× MIDI rate @ 4 MHz
0x80	× 32	125K baud 4x MIDI rate @ 4 MHz
0xC0	× 64	31.25K baud standard MIDI rate @ 2 MHz

Set 115.2K Baud Rate[control code 115]

`csCode` = 115

This call is designed for high-speed modems. It typically requires DMA hardware on the receive channel to be successful. It is similar to the clock selection function available through csCode = 16, but it instead forces the serial driver to take its baud rate clock directly from the internal 3.672 MHz RTxC clock source with a rate multiplier of 32. The result is to force transmit and receive baud rates of nominally 115.2K baud. Other configuration parameters are not affected.

Set 230.4K Baud Rate[control code 230]

`csCode` = 230

This call is designed for high-speed modems. It typically requires DMA hardware on the receive channel to be successful. It is similar to the clock selection function available through csCode = 16, but it instead forces the serial driver to take its baud rate clock directly from the internal 3.672 MHz RTxC clock source with a rate multiplier of 16. The result is to force transmit and receive baud rates of nominally 230.4K baud. Other configuration parameters are not affected.

IMPORTANT
The highest baud rates supported by this driver may be quite demanding on the client software. Efficiency at the client layer is critical to take advantage of the benefits of serial DMA reception. Awkward or inefficient use of the serial driver API may result in poor system performance and failure to sustain the desired data rate. Please read the performance considerations in the following section. ◆

Additional Details

The basis of the second-generation SerialDMA driver is a rewrite of the classic serial driver (which was written in 68K assembly language) using the C language. There is a minimal amount of assembly language glue code (Mixed Mode glue in the case of the native Power Macintosh version) to bind to the 68K Device Manager. The SCC generally is no longer responsible for generating transmit or receive interrupts, although it still notifies the driver of external/status changes and special receive conditions. All normal data transfer is effected by DMA, and DMA interrupts keep the system synchronized with transfer progress.

No attempt has been made to abstract the Z8530 Serial Communications Controller hardware. The serial driver is intimately tied to this piece of legacy hardware. However, the bulk of the driver is relatively independent of the details of any specific DMA controller. A handful of primitive vectors are installed when the driver is opened and all DMA operations are handled in a device-independent manner by the main part of the SerialDMA driver.

Just to give an idea of the flexibility of the driver with respect to DMA models, the DMA controller in the Quadra 840AV requires a pair of user-defined, linear DMA buffers of arbitrary size and automatically ping-pongs between them when the transfer count on each buffer goes to zero. The Power Macintosh 8100 contains a single system-defined, circular DMA buffer which interrupts when the transfer count goes to zero and then optionally continues transferring characters even while the interrupt awaits processing. The Power Macintosh 9500 uses a semi-intelligent DMA command processor which supports an arbitrary number of buffers. The SerialDMA driver supports DMA models with ease, using only eight brief, abstract primitives and three or four interrupt handlers unique to each DMA controller.

PowerPC Native Implementation

For Power Macintosh, the SerialDMA driver is compiled native. This is not a Marconi/Copland CFM driver. It is a traditional Macintosh DRVR compiled in the native PowerPC instruction set. In order to bind the driver to the 68K architecture Device Manager, the driver header offsets reference Mixed Mode routine descriptors. All SCC and DMA interrupt handlers are native, and pointers to routine descriptors are installed in applicable interrupt dispatch tables. The driver calls 68K routines such as IODone and the LAP Manager port B arbitration routines with `CallUniversalProc` and appropriate procedure info constants.

It is understood that the Mixed Mode switches in the native PowerPC SerialDMA driver incur costly overhead, but it is thought that native mode execution of certain critical code sequences when DMA is suspended, or when delivering large packets of data with interrupts disabled, provides an overall win for the driver under the most challenging performance conditions.

The native version of the driver is packaged as a native code resource of type `'nsrd'` with Mixed Mode calling conventions identical to that of the `'SERD'` resource. The `'SERD'` resource is for 68K machines only. The `'nsrd'`resource is for PowerPC machines only.

Interrupts

The traditional Macintosh serial driver functions by responding to a system interrupt each and every time a character arrives at the SCC receiver, leaves the SCC transmitter, or when the state of the CTS input changes or a break condition is detected. This provides an exceptional level of responsiveness because the driver responds to every event occurring at the SCC with a delay equivalent only to the system interrupt latency period. On the other hand, this is also the greatest limitation of the traditional serial driver because the typical system interrupt latency places an upper bound on the data throughput of the driver. For example, at the rate of 230.4K bps, a new character may arrive at the SCC every 43.4µs, or 1085 machine cycles (25 MHz CPU). Since a round trip through the system interrupt handler and the driver’s interrupt service routine may easily take several hundred machine cycles, it is apparent that this data rate would consume a very large percentage of the CPU bandwidth resulting in poor system performance and perhaps exceed it, resulting in dropped data.

The DMA engine offloads responsibility from the CPU for moving data between the SCC and memory. In the best case, data throughput is limited not by system interrupt latency but by the bandwidth of the memory system, which is usually much higher than the rates achievable by common serial I/O hardware. The DMA hardware generates interrupts only after a previously specified number of characters have been transferred. Comparing this to the non-DMA model, it is intuitive that the benefit of DMA transfer increases approximately linearly with the average size of the DMA transfer count. Every character transferred without processor intervention saves valuable processor time and improves response time for other interrupt-driven processes.

One factor which reduces the benefit of DMA is the increased complexity of the DMA interrupt handler relative to the SCC single-character I/O interrupt handler. It is important to overcome this brake on performance by taking advantage of the DMA benefits described earlier. In general, it should only be necessary to average a few characters per DMA block to overcome the increased overhead of a generalized DMA block handler versus a single-character handler. However, it should be understood that it is possible to operate the DMA serial driver in ways that do not take advantage of DMA and incur even greater overhead than the traditional interrupt-driven serial driver. Every attempt should be made to avoid such inefficiencies when performance is critical or data throughput is high by traditional Macintosh serial I/O standards.

As previously stated, the DMA serial driver causes the hardware to generate interrupts upon the completion of a DMA block transfer. It also responds to status interrupts in exactly the same manner as the traditional Macintosh serial driver, so each time the state of the CTS input changes or a break condition is detected, an interrupt must be serviced. Furthermore, various serial driver API calls invoke the drivers interrupt handler in order to synchronize the DMA engine with pending I/O requests, handshaking thresholds, and so on. These implicit interrupts which occur as a result of API calls are required to approximate the responsiveness to certain events which are supported by the Macintosh serial driver API.

Performance Considerations

A primary concern, based on customer feedback, was how to duplicate the responsiveness of an interrupt-per-character serial driver in a DMA serial driver which, by definition, interrupts only upon the completion of a block transfer. The solution requires relatively sophisticated DMA receive channel management and implicit interrupts in response to a number of driver API calls. Each time a DMA interrupt occurs, and in various other circumstances, a new DMA count must be calculated in order to generate the next interrupt at the optimal moment based on the selected handshaking mode, read request, and buffer size. The exact details of this calculation are beyond the scope of this document, but extreme care has been taken to minimize the calculation time, during which the DMA receive channel must remain inactive. However, it bears mentioning the effects that the client's use of the serial driver can have upon the driver's DMA channel management.

XOn/XOff output handshaking is extraordinarily expensive to support within a DMA serial driver. The reason for this is that in order to guarantee acceptable response times to the reception of XOn and XOff characters, the driver must suspend most of the benefits of DMA and interrupt on every single received character, just like the non-DMA serial driver. Only it is worse because the DMA interrupt handler is more complex and time consuming than a standard receive character interrupt handler. Nevertheless, since the DMA driver cannot support old-fashioned pollprocs for immunity to interrupt latency, use of the DMA interrupt handler on every character is the only viable recourse. As a result, XOn/XOff handshaking is not recommended at high data rates (as a rule of thumb, 57,600 bps is probably too fast for efficient software handshaking).

It is not quite as punishing, but any type of input handshaking may put a limit on the DMA transfer count (and therefore available DMA resources) because it is necessary to generate an interrupt and assert flow control when the buffer threshold is reached.

Regardless of the initially programmed DMA transfer count, special receive conditions must terminate active DMA transfers (again, limiting the available DMA resources) in order to support extraction of corrupted characters from the data stream in accordance with the serial driver specification. Reception of break sequences may also temporarily limit DMA resources, or even suspend receive channel DMA during the break assertion.

Because DMA transfer counts are dependent upon the handshaking mode, buffer size, read request size, and other factors, numerous `Control` and `Status` calls require that DMA be stopped temporarily and restarted with new parameters. This involves some overhead which can be avoided frequently through more sophisticated use of the serial driver API. For example, rather than polling `SerGetBuf` frequently, it is much more efficient to make a single asynchronous `Read` request for some expected amount of data, and time out the request with a `KillIO` sometime later if it is not forthcoming. There is no overhead whatsoever to leave a pending read request while no data are being transferred by the SCC, but there is a great deal of overhead polling `SerGetBuf` in a loop. This is true of both DMA and non-DMA serial drivers.

Another performance-killer is the commonly-used small, chained read algorithm. The purpose of DMA is to stream relatively large quantities of data at high speed, and posing frequent, one-character read requests is very counterproductive. Each time a read completes, an interrupt must occur. Also, each time a read is issued, an implicit interrupt results to synchronize the DMA engine with the clients request count. The key to sustaining high data rates is in limiting the number of system interrupts and reaping some economy of scale in block data processing.

Hints for Optimizing Performance

So some hints for optimizing performance are:

- Avoid the use of XOn/XOff handshaking to control driver data output at high baud rates. This makes DMA worthless on the receive channel and poses a worst-case performance scenario.
- Avoid repeatedly polling the serial driver for status or buffer information. This approach, while simple, is fully synchronous and particularly inefficient for the DMA serial driver.
- Use aynchronous requests whenever possible, for the full amount of data expected in a transaction. Process the transaction all at once. You can and should always time out a transaction with `KillIO` if things do not go according to plan. On the other hand, it is *never* necessary to issue a `KillIO` command unless it is possible that an asynchronous request has been left pending.
- Avoid requiring the serial driver to use its own buffer (its default or the one set by `SerSetBuf`) when input data are expected. Keep an asynchronous read pending instead. A client buffer is more efficient because handshaking threshold checks are not required on client buffers.
- If your completion routine performs complex processing (i.e., more than setting or clearing a semaphore) or chained I/O, consider executing that code as a deferred task in order to return control to the drivers' interrupt handler as soon as possible, freeing the system to process other interrupts. The driver would prefer to call serial I/O completion routines as deferred tasks in the first place, but this causes compatibility problems with certain third-parties' software which call the driver synchronously from VBLs, Time Manager tasks, etc.

Acknowledgments

Special thanks to the following individuals for their support and making SerialDMA 2.0 possible: Rich Collyer, Mark Baumwell, Kevin McCoy, Bob Crane, John Banner, and Werner Klemperer.