Symbian OS Internals/13. Peripheral Support

- Symbian OS Internals Table of Contents

by Peter Scobie

The nice thing about standards is that there are so many of them to choose from.
Andrew Tannenbaum

In this chapter, I begin by describing more of the services available to device drivers. Then I go on to describe a number of specific device driver implementations - among them media drivers, the MultiMediaCard driver and the USB driver.

The first service, one that device drivers frequently make use of, is direct memory access, or DMA.

DMA

DMA hardware

DMA hardware allows data to be transferred between a peripheral and system memory without the intervention of the processor. It is used to ease the burden on the processor of servicing peripherals that produce frequent interrupts. This applies equally to transfers involving data received by the phone (data taken from the peripheral and stored in a buffer) or data transmitted by the phone (data taken from a buffer and pushed to the peripheral).

A hardware DMA controller manages a set of DMA channels, each channel providing one direction of communication between the memory and the peripheral (either transmit or receive). Because of this, full duplex communication with a peripheral requires the use of two channels. Many controllers also support DMA transfer directly from one area of system memory to another.

Individual DMA transfer requests can be set up on each channel. These requests can involve a considerable amount of data - an amount that would otherwise involve the processor servicing a series of interrupts. But the use of DMA means that only one processor interrupt is generated, at the end of the transfer. For certain types of peripheral data transfer, such as bulk transfer over full-speed USB, the use of DMA becomes essential to avoid the CPU consuming an excessive proportion of its available bandwidth servicing interrupts.

Symbian OS phones normally provide one or more DMA channels dedicated to the LCD controller, for the transfer of pixel data between a frame buffer and the LCD interface. The LCD controller typically manages these DMA resources itself and they are not available to the main Symbian OS DMA framework.

In addition to these LCD channels, most phone platforms provide a limited number of general-purpose DMA channels. Platforms have differing strategies on how these channels are allocated. Some may fix the allocation of each DMA channel to a particular peripheral. Others allow *dynamic* allocation. This second method provides a pool of DMA channels that can be allocated to any peripheral on a first-come first-served basis. This could allow the platform to provide DMA services to more peripherals than there are DMA channels - assuming that device drivers free up DMA channels when they are not in use. With this scheme, however, there is a risk that a device driver might request a DMA channel but find that none are free. So the Symbian OS software framework normally fixes the allocation of DMA channels to particular devices, even when the hardware does not impose this.

Normally, DMA hardware reads and writes directly to physical address space, bypassing the MMU. Let's examine the simplest type of DMA request involving a one-shot transfer to or from a single, physically contiguous memory region. In this case, the DMA channel is supplied with information on the transfer source and destination (each being either a physical memory address or a peripheral identifier), together with the number of bytes to transfer. The request is initiated and the controller interrupts the processor once either the transfer completes successfully, or an error occurs.

DMA controllers often provide support for data to be transferred as a continuous stream as well as a one-shot transfer. This requires more than one set of transfer registers for each channel, allowing a double-buffer scheme to be employed. While the controller is transferring data using one set of registers, the software can simultaneously be programming the second set ready for the next transfer. As soon as the first transfer completes, the controller moves on to process the second one, without interrupting data flow. At the same time the controller signals the end of the first transfer with an interrupt, which signals the software to begin preparing the third - and so on.

Some DMA controllers support scatter-gather mode. This allows a DMA channel to transfer data to and from a number of memory locations that aren't contiguous, all as a single request. The control software has to supply information to the DMA controller, describing the transfer as a linked list of data structures, each specifying part of the transfer. These data structures are known as descriptors. (Not to be confused with the same term widely used in Symbian OS to refer to the family of `TDesC` derived classes!)

(C) Copyright Nokia 2014. All rights reserved.

The DMA controller acts on each descriptor in turn, and only interrupts the processor at the end of the descriptor chain. Some controllers allow the descriptor chain to be extended while transfer is in progress - another way to facilitate uninterrupted data transfer.

Figure 13.1 shows a setup for scatter-gather DMA transfer from two disjoint blocks of memory into a peripheral.

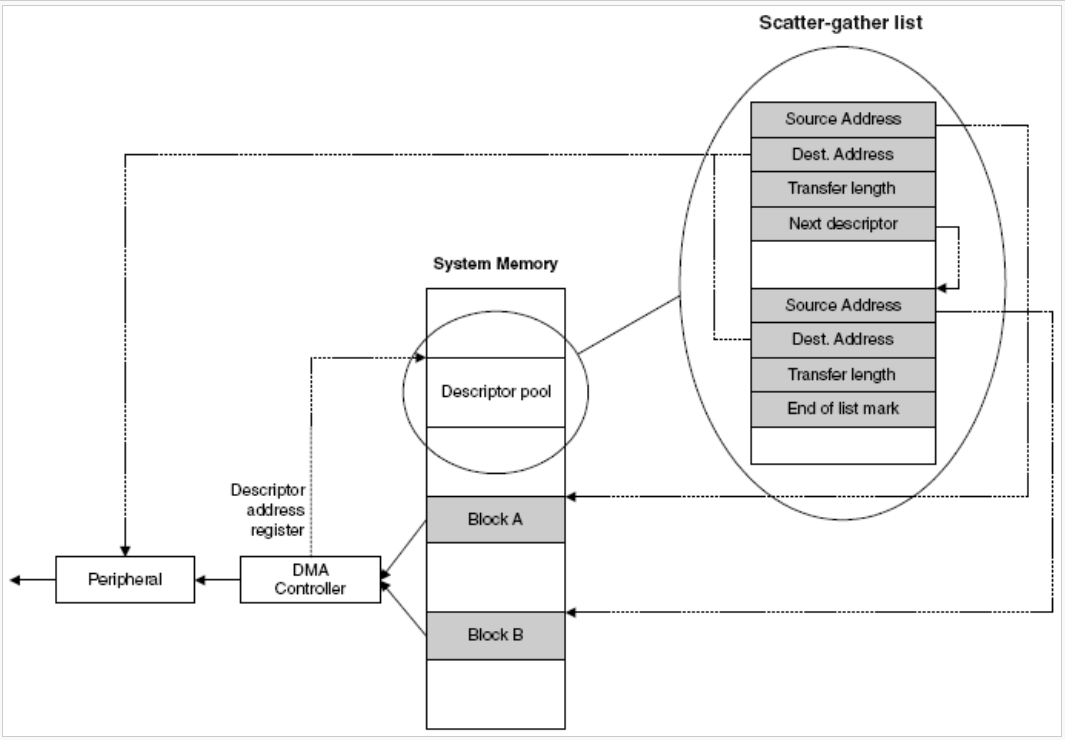

Figure 13.1 *Scatter-gather DMA transfer*

The scatter-gather list contains two linked descriptors - each providing information on one of the blocks. This list is supplied by the control software, which also loads a descriptor address register in the DMA controller with the address of the first descriptor. The actual descriptor format is generally hardware dependent and often more complex than that shown in the diagram. For example, it may contain information on the addressing mode for the source and destination addresses.

For each channel that involves peripheral-related DMA transfer, the controller normally has to be programmed with information on the burst size and port width of the peripheral. Burst size is the amount of data that has to be transferred to service each individual DMA request from the peripheral device. Many peripherals employ a FIFO buffer and for these, the burst size depends on the size of the FIFO and the threshold level within the FIFO that triggers a DMA request from it (for example, FIFO half-empty or quarter-empty). The port width specifies the granularity of data transfer (for example, byte, half-word and so on).

DMA software framework

Symbian OS provides kernel-side DMA services through its DMA framework. We leave the choice of whether or not a particular device will use DMA for data transfer to the developers who are creating a new phone platform - and because of this, the consumers of this service are generally components such as physical device drivers (PDDs) and the platform-specific layers of peripheral bus controllers (see Section 13.4). The framework itself is divided into a platform-independent layer (PIL) and a platform-specific layer (PSL), with both parts being combined into the kernel-side DLL, DMA.DLL. As with most platform-specific components, the PSL interfaces with the controller hardware via functions exported from the variant or ASSP DLL. Figure 13.2 shows this arrangement.

(C) Copyright Nokia 2014. All rights reserved.

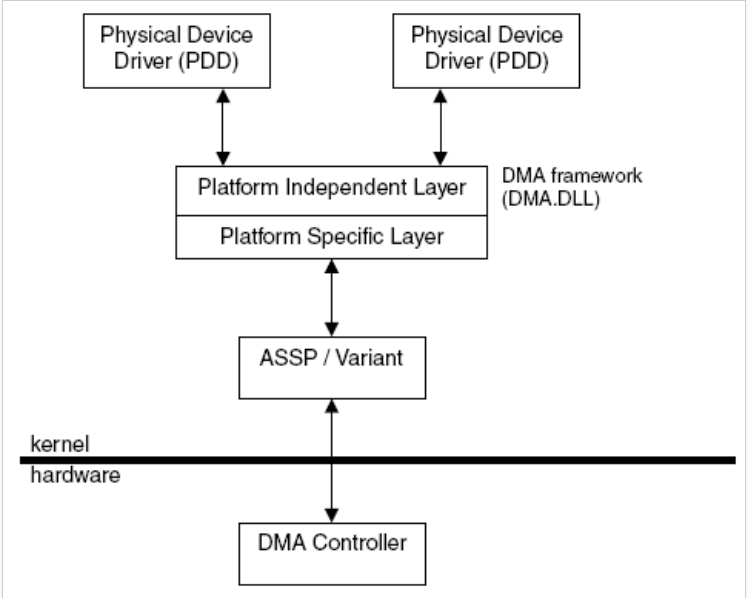

Figure 13.2 *DMA.DLL*

DMA support may be required by certain peripheral services that are started at system boot time. Because of this, the DMA framework is implemented as a kernel extension and is normally one of the first of these to be initialized by the kernel.

Drivers may request a transfer involving a memory buffer that is specified in terms of its linear address. As I explained in Chapter 7, *Memory Models*, this contiguous linear memory block is often made up of non-contiguous physical areas. Even when a transfer request involves a physically contiguous region of memory, the total length of the transfer may exceed the maximum transfer size that the underlying DMA controller supports. You don't need to worry about this though - the Symbian OS DMA framework performs any fragmentation necessary, due either to requests exceeding the maximum transfer size, or to buffers not being physically contiguous. The framework specifies each fragment with a separate descriptor. If the controller doesn't support scatter-gather then each fragment has to be handled by the framework as a separate DMA transfer - but the framework insulates the device driver from this detail, by only signaling completion back to the driver at the end of the entire transfer.

Figure 13.3 shows a diagram of the classes that make up the DMA framework.

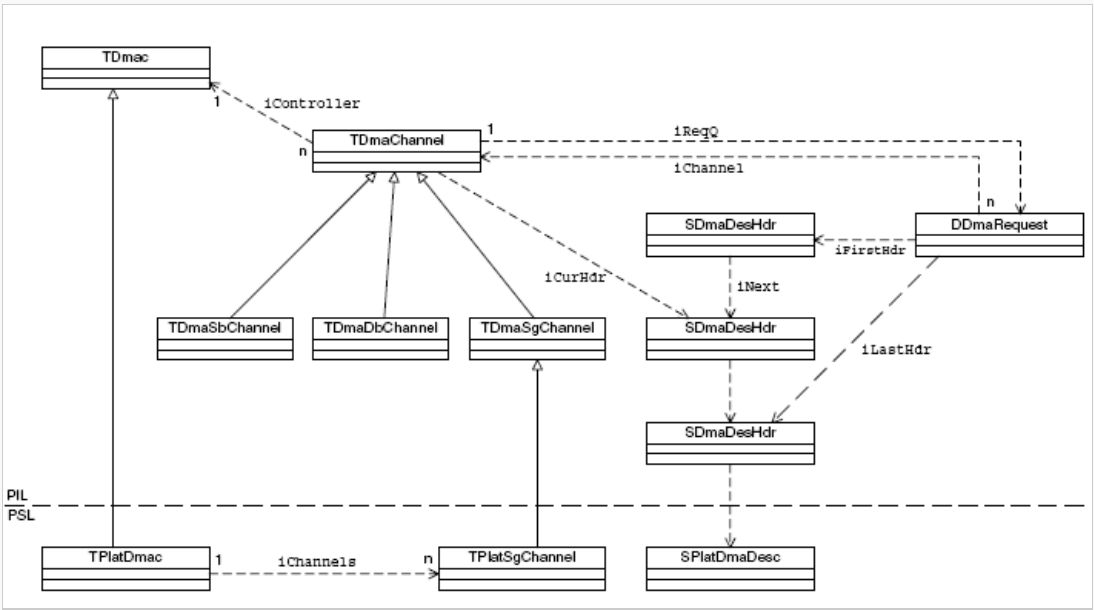

Figure 13.3 *Class diagram for the DMA framework*

The framework includes the singleton `TDmac`, which is the abstraction of the DMA controller. This object manages all the DMA channels and defines the main interface between the PIL and PSL. It is an abstract class with each phone platform deriving a controller object from it. This is shown as `TPlatDmac` on the diagram.

TDmaChannel is the base class for a DMA channel. We derive three different channel classes from this as follows:

- `TDmaSbChannel` - a single-buffer DMA channel
- `TDmaDbChannel` - a double-buffer DMA channel
- `TDmaSgChannel` - a scatter-gather DMA channel.

(C) Copyright Nokia 2014. All rights reserved.

These in turn are base classes, and the programmers creating a new phone platform will derive a platform-specific channel object from whichever of these is appropriate for the buffering scheme that the controller hardware provides. In the diagram, I've assumed scatter-gather mode and shown it as `TPlatSgChannel`. When the DMA framework extension is initialized at system boot time, an instance of this derived channel class is created for each general-purpose hardware channel that the controller manages.

The `TDmaChannel` class is one of the two major elements of the DMA API provided to device drivers. (The other is the `DDmaRequest` class that I will introduce later.) The main public parts of this class and data members are as follows:

```
class TDmaChannel
    {
public:
    // Information passed by client when opening channel
    struct SCreateInfo
        { /** ID used by PSL to select channel to open */
        TUint32 iCookie;
        /** Number of descriptors this channel can use. */
        TInt iDesCount;
        /** DFC queue used to service DMA interrupts. */
        TDfcQue* iDfcQ;
        /** DFC priority */
        TUint8 iDfcPriority;
        };
public:
    static TInt Open(const SCreateInfo& aInfo, TDmaChannel*& aChannel);
    void Close();
    void CancelAll();
    inline TBool IsOpened() const;
    inline TBool IsQueueEmpty() const;
protected:
    TDmac* iController; //DMAC this channel belongs to.
    TDfc iDfc; //Transfer complete/failure DFC.
    SDmaDesHdr* iCurHdr; //Fragment being transferred.
    SDblQue iReqQ; //Being/about to be transferred request queue.
    TInt iReqCount; //Number of requests attached to this channel
    };
```

The `TDmaChannel` class has a pointer to the controller object that manages the channel: `iController`. It also owns a DFC object `iDfc`, which is called whenever a request for this channel completes.

Next I will describe the public `TDmaChannel` methods:

```
static TInt Open(const SCreateInfo& aInfo, TDmaChannel*& aChannel);
```

This static method opens the DMA channel identified by the information supplied within the parameter aInfo. If the controller is able to open the channel successfully, then it returns a pointer to the appropriate channel object in aChannel. A device driver must open each channel that it needs to use - it normally attempts this in its initialization. You can see the format of the structure passed by the driver in the class definition I've just given - it includes a 32-bit channel identifier, which the platform-specific layer uses to identify the corresponding channel object. The rest of the information in the structure specifies the channel configuration. This includes the queue that is to be used for the channel's DFC, and the priority of this DFC relative to others in the same kernel thread.

It is normal for a driver to close any DMA channels used when it has finished with them. It generally does this when the driver itself is being closed using the channel method:

```
void Close();
```


(C) Copyright Nokia 2014. All rights reserved.

A channel should be idle when it is closed. The following channel method is used to cancel both current and pending requests:

```
void CancelAll();
```

The framework uses descriptor objects to hold transfer request information, whether or not the controller supports scatter-gather. Controllers that do support scatter-gather need the descriptor information to be supplied in a format that is specific to that particular hardware controller. This is shown as the `SPlatDmaDesc` structure on the diagram. If the controller doesn't support scatter-gather, then the framework uses a default descriptor structure, `SDmaPseudoDes` (not shown on the diagram). The pseudo descriptor contains the following information listed. Hardware-specific descriptors generally contain similar information:

1. Transfer source location information. This can be either the address of a memory buffer or a 32-bit value identifying a particular peripheral. For memory addresses, this may hold either the linear or physical address of the buffer
2. Transfer destination location information (same format as 1)
3. The number of bytes to be transferred
4. General information, such as whether the source/destination is a memory address or a peripheral identifier, whether memory addresses contain linear or physical addresses and whether these need to be post-incremented during transfer
5. 32-bits of PSL-specific information provided by the client.

For controllers that do support scatter-gather, because the hardware imposes the structure of the descriptor, it is difficult to include any additional descriptor information required by the framework alone. Therefore, the framework associates a separate descriptor header with each descriptor. This is the `SDmaDesHdr` structure. The descriptor header is generic and the PIL manipulates descriptors via their associated header. The framework still creates associated descriptor headers even when pseudo descriptors are used.

Each time a driver opens a channel, it has to supply information on the number of descriptors that the framework should reserve for it. This depends on the buffering scheme being used and the maximum number of DMA requests that are likely to be outstanding at any time.

The class `DDmaRequest` encapsulates a DMA request over a channel. This is the second main element of the DMA device driver API. The main public parts of this class and data members are as follows:

```
class DDmaRequest : public DBase
    {
public:
    // Signature of completion/failure callback function
    typedef void (*TCallback)(TResult, TAny*);
public:
    DDmaRequest(TDmaChannel& aChannel, TCallback aCb=NULL, TAny* aCbArg=NULL, TInt
aMaxTransferSize=0);
    ~DDmaRequest();
    TInt Fragment(TUint32 aSrc, TUint32 aDest, TInt aCount, TUint aFlags, TUint32
aPslInfo);
    void Queue();
public:
    TDmaChannel& iChannel; //Channel this request is bound to
    TCallback iCb; //Called on completion/failure
    TAny* iCbArg; //Callback argument
    TInt iDesCount; //Number of fragments in list
    SDmaDesHdr* iFirstHdr; //The first fragment in the list.
    SDmaDesHdr* iLastHdr; //The last fragment in the list.
    };
```

Typically a driver will allocate itself one or more request objects for each channel that it has open. The constructor for the `DDmaRequest` class is as follows:

```
DDmaRequest(TDmaChannel& aChannel, TCallback aCb=NULL, TAny* aCbArg=NULL, TInt
aMaxTransferSize=0);
```


(C) Copyright Nokia 2014. All rights reserved.

In constructing a request object, the driver must specify the channel with which it is to be used: aChannel. It must also supply a callback function, aCb, which the channel DFC will call when the request completes (either following successful transfer or due to an error). The next parameter, aCbArg, is a driver-specified argument that will be saved by the framework and passed as an argument into the callback function. Often a device driver will pass in a pointer to itself, allowing the callback function to access driver member functions when it is invoked. The final parameter, aMaxTransferSize, is used if the driver needs to specify the maximum fragment size to be applied during the transfer. If the driver does not specify this, then the transfer size defaults to the maximum transfer size that the DMA controller supports, so this parameter is only used when a particular peripheral needs to impose a smaller limit.

Section 13.2.6.1 contains example code showing how a driver might open a DMA channel and construct a request object.

To initiate a DMA transfer, the driver supplies the transfer details and then queues the request object on the channel. Each channel maintains a queue of transfer requests, `TDmaChannel::iReqQ`. Once a transfer request is complete, the framework de-queues it and invokes the request callback function. The driver is then able to reuse the request object for a subsequent transfer.

A driver is able to queue a series of requests on a channel and, if the channel supports double buffering or scatter-gather, then the framework will manage these so that they are transferred as an uninterrupted stream. However, the framework does continue to invoke each request callback as each request is completed. It's worth noting that the DMA device driver API does not allow a driver to specify a transfer that involves two or more disjoint memory buffers as a single DMA request. But, as I have just explained, if the driver queues separate requests for each memory buffer, the framework can take advantage of a scatter-gather facility to transfer these as an uninterrupted stream.

If a channel is being used to transfer an uninterrupted stream of data, then the channel request queue may contain several requests - the first being the one in progress, and those remaining being pending requests. Information on the total number of requests queued on a channel at any time is held in its data member, `TDmaChannel::iReqCount`.

Before a request is queued, the driver has to specify the details of the transfer and then allow the framework to analyze these and possibly split the overall request into a list of smaller transfer fragments. To do this, the driver calls the following method:

```
TInt Fragment(TUint32 aSrc, TUint32 aDest, TInt aCount, TUint aFlags, TUint32 aPslInfo);
```

Arguments {Icode|aSrc}} and `aDest` specify the transfer source and destination respectively. Each of these can be a linear address, a physical address or a 32-bit value identifying a particular peripheral. This format is specified by bit masks in the argument aFlags, which also indicates whether memory addresses need to be post-incremented during transfer. Argument aCount holds the number of bytes to transfer and clients may use aPslInfo to specify 32-bits of PSL-specific information.

Where these request arguments specify a memory buffer in terms of its linear address, this contiguous linear memory block may consist of non-contiguous physical areas. In such cases, the fragmentation means that the framework must split this area into smaller, physically contiguous, fragments. Later, in Section 13.1.3, I will discuss methods that you can use to avoid the need for this fragmentation, with the driver allocating memory buffers that are physically contiguous.

The framework may also have to split large transfers into a series of smaller fragments - each being smaller than or equal to the maximum transfer size.

In this way, the framework breaks up the request into a series of descriptors, each specifying how to transfer one fragment. Each descriptor has an associated descriptor header, and these headers are formed into a linked list. The transfer request object contains pointers to the first and last headers in this list: `iFirstHdr` and `iLastHdr` respectively. When there is more than one request queued on the channel, then the headers of all the requests are linked together into a single linked list. During transfer, the channel maintains a pointer to the header corresponding to the descriptor currently being transferred, `TDmaChannel::iCurHdr`. To illustrate this arrangement, Figure 13.4 shows a channel with a three-fragment request being transferred, and a two-fragment request pending. The fragment that is currently being transferred is the last one of the first request.

Once a request has been fragmented, the driver needs to queue it on its associated channel:

```
void Queue();
```

If this channel is idle, the framework transfers the request immediately; otherwise it stores it on a queue for transfer later. Once the transfer completes, either successfully or with an error, the framework executes the callback function associated with the request object. Here the driver should check and handle any transfer errors and queue another DMA transfer if it needs to.

(C) Copyright Nokia 2014. All rights reserved.

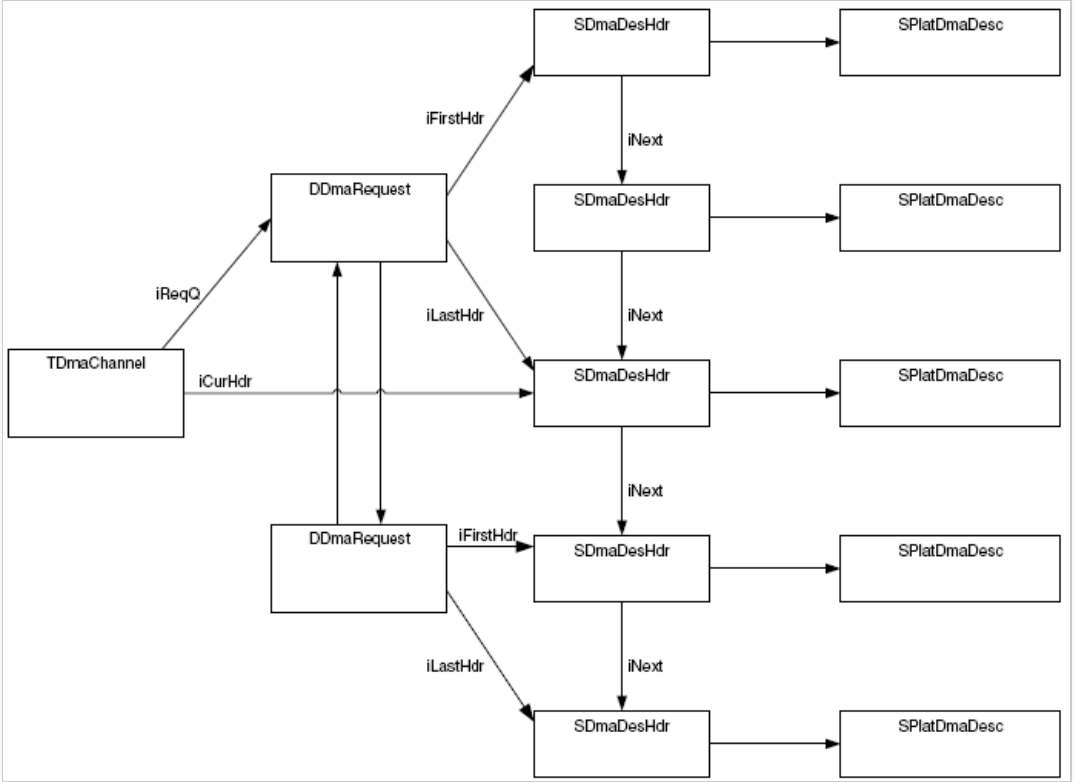

Figure 13.4 *Transferring fragmented requests*

Section 13.2.6.3 contains example code showing how a driver might perform a DMA data transfer. If you are working with DMA, you need to pay special attention to the way in which you handle data transfers that turn out to be shorter than the requested length, leaving the DMA request outstanding. This is most likely to happen on DMA reads (that is, transfers from peripheral to memory buffer). The normal way in which you would handle this is to timeout the transfer after an appropriate period, at which point the DMA transfer is cancelled. You may then need to initiate the retransmission of the entire data sequence. Alternatively it may be necessary to recover any data that has been transferred to the buffer. If the amount received is not an exact multiple of the burst size, then there may also be data residing in the peripheral FIFO (in other words, trailing bytes).

Memory allocation for DMA transfer

The DMA framework is not responsible for managing the memory buffers used for the DMA transfer. This is left to the users of the framework.

You can't safely perform DMA transfer directly to or from memory that has been allocated to a user process in the normal way - that is, to user chunks. There are a number of reasons for this:

1. While a DMA transfer is in progress to the user chunk, the owning user process might free the memory - or the kernel might do so, if the process dies. This is a problem because the freed memory could be reused for other purposes. Unaware of the reallocation, the DMA controller would continue with the transfer, using the physical addresses supplied, and trash the contents of the newly allocated memory. You can overcome this problem by ensuring that the driver opens the chunk representing the user memory for the duration of the transfer - but this can be inefficient
2. A process context switch may change the location of the memory. To be suitable for DMA, the memory needs to be available to the kernel at a fixed location
3. The peripheral may mandate DMA to a specific physical memory region and the allocation of user-mode memory doesn't allow this attribute to be specified
4. Since the DMA controller interfaces directly with the physical address space, it bypasses the MMU, cache and write buffer. Hence, it is important to ensure that DMA memory buffer and cache are coherent. One way to achieve this is to disable caching in the buffers used for DMA transfer. Again, the allocation of user-mode memory doesn't allow these caching and buffering options to be specified.

You can avoid all these problems by allocating the DMA buffers kernel-side, and so it is usual for device drivers that support DMA to do the allocation of memory that is to be used for DMA transfers themselves.

The example code that follows shows how a driver would use a hardware chunk to allocate a buffer that is non-cacheable and non-bufferable to avoid cache incoherency issues. This creates a global memory buffer - accessible kernel-side only. By using RAM pages that are physically contiguous, this also avoids the issue of memory fragmentation.

(C) Copyright Nokia 2014. All rights reserved.

```
TUint32 physAddr=0; TUint32 size=Kern::RoundToPageSize(aBuffersize); // Get contiguous
pages of RAM from the system's free pool
if (Epoc::AllocPhysicalRam(size,physAddr) != KErrNone) return(NULL);
// EMapAttrSupRw – supervisor read/write, user no access
// EMapAttrFullyBlocking – uncached, unbuffered DPlatChunkHw* chunk; 
if(DPlatChunkHw::New(chunk,physAddr, size,EMapAttrSupRw|EMapAttrFullyBlocking) !=
KErrNone)
    {
    Epoc::FreePhysicalRam(physAddr,size);
    return(NULL);
    }
TUint8* buf;
buf=reinterpret_cast<TUint8*>(chunk->LinearAddress());
```

On the other hand, you may have reasons that make it preferable to allocate DMA buffers that are cached - for instance if you want to perform significant data processing on data in the buffer. You can do this using the same example code - but with the cache attribute `EMapAttrFullyBlocking` replaced with `EMapAttrCachedMax`. However, to maintain cache coherency, you must then flush the cache for each DMA transfer.

For a DMA transfer from cacheable memory to peripheral (that is, a DMA write), the memory cache must to be flushed before transfer. The kernel provides the following method for this:

```
void Cache::SyncMemoryBeforeDmaWrite(TLinAddr aBase, TUint aSize, TUint32 aMapAttr);
```

For DMA transfer from peripheral to cacheable memory (that is, a DMA read), the cache may have to be flushed both before and after transfer. Again, methods are provided for this:

```
void Cache::SyncMemoryBeforeDmaRead(TLinAddr aBase, TUint aSize, TUint32 aMapAttr); void
Cache::SyncMemoryAfterDmaRead(TLinAddr aBase, TUint aSize);
```

It's worth pointing out that only kernel-side code can access the types of hardware chunk I've described so far. So, if the ultimate source or destination of a data transfer request is in normal user memory, you must perform a two-stage transfer between peripheral and user-side client:

1. DMA transfer between peripheral and device driver DMA buffer
2. Memory copy between driver DMA buffer and user memory.

Obviously a two-stage transfer process wastes time. How can it be avoided? In the previous example buffer-allocation code, if you set the access permission attribute to `EMapAttrUserRw` rather than `EMapAttrSupRw`, the driver creates a user-accessible global memory buffer. The driver must then provide a function to report the address of this buffer as part of its user-side API. Note that you can't make these chunks accessible to just a limited set of user processes and so they are not suitable for use when the chunk's contents must remain private or secure.

A much more robust scheme for avoiding the second transfer stage is for client and driver to use a shared chunk as the source or destination of data transfer requests between peripheral and user accessible memory. I will discuss this in the next section.

Shared chunks

As described in Section 7.3.1, chunks are the means by which memory is allocated and made available to code outside of the memory model. In Symbian OS, we represent chunks by `DChunk` objects, and we support various types of these. I mentioned user chunks and hardware chunks in the previous section, and discussed the problems with DMA transfer involving these chunk types.

In this section I describe a third type of chunk - the shared chunk. These provide a mechanism for device drivers to safely share memory buffers with user-mode code. Shared chunks are only available on EKA2. They should not be confused with global chunks, (created for example using `RChunk::CreateGlobal()` which are also accessible by multiple user processes. However global chunks, being a type of user chunk, have all the problems that I listed in the previous section when accessed by device drivers.

Another type of chunk is the shared I/O buffer. These are supported in EKA1, but deprecated in EKA2, and also allow memory to

(C) Copyright Nokia 2014. All rights reserved.

be safely shared between kernel and user code. However, unlike shared chunks, these buffers have the drawback that only one user-side process at a time can open them. Another difference is that, for a user-side application to access a shared chunk, it must create a handle for that chunk and assign it to an `RChunk` object. For shared I/O buffers, there is no user-mode representation. Instead the user process is generally just supplied with an address and size for the buffer by the driver that performs the user process mapping.

We represent a shared chunk with a `DChunk` object. This class is internal to the kernel - it has no published methods - but instead the Kern class publishes a set of methods for performing operations on shared chunks. The following sections describe these methods and provide an example of their use.

Creating a shared chunk

Only kernel-side code can create shared chunks, by using the following function:

```
TInt Kern::ChunkCreate(const TChunkCreateInfo& aInfo, DChunk*& aChunk, TLinAddr&
aKernAddr, TUint32& aMapAttr);
```

The argument aInfo is used to supply details of the attributes of the chunk required. If chunk creation is successful, the function returns with aChunk containing a pointer to the new chunk object. This object owns a region of linear address space, but it is empty - the kernel has not committed any physical RAM to it. You have to map either RAM or I/O devices to the chunk before it can be used. The argument aKernAddr returns the base address of this linear address space in the kernel process - only kernel code can used this address. The argument aMapAttr returns the mapping attributes that apply for the chunk created. The caller will often save this value and pass it as an argument into the DMA cache flushing functions described in Section 13.1.3. I will talk more about these mapping attributes later in this section.

The structure `TChunkCreateInfo` is used to specify the attributes of the required chunk. This is defined as follows:

```
class TChunkCreateInfo
    {
public:
    enum TType
        {
        ESharedKernelSingle = 9,
        ESharedKernelMultiple = 10,
        };
public:
    inline TChunkCreateInfo();
public:
    TType iType;
    TInt iMaxSize;
    TUint32 iMapAttr;
    TUint8 iOwnsMemory;
    TInt8 iSpare8[3];
    TDfc* iDestroyedDfc;
    TInt32 iSpare32[2];
    };
```

The member `iType` specifies the type of shared chunk, which can be one of the following values:

Type	Description
EsharedKernelSingle	A chunk that may only be opened by one user-side process at a time.
EsharedKernelMultiple	A chunk that may be opened by any number of user-side processes.

The member `iMaxsize` specifies the size of the linear address space to reserve for the chunk.

The member `iMapAttr` specifies the caching attributes for the chunk's memory. This should be constructed from the cache/buffer values for the `TMappingAttributes` enumeration defined in the file \e32 \include\memmodel\epoc\platform.h. Frequently used values are:

(C) Copyright Nokia 2014. All rights reserved.

1. `EMapAttrFullyBlocking` for no caching
2. `EMapAttrCachedMax` for full caching.

However, it is possible that the MMU may not support the requested caching attribute. In this case, a lesser attribute will be used, with this being reported back to the caller of `Kern::ChunkCreate()` via the parameter aMapAttr.

You should set the member `iOwnsMemory` to true if the chunk is to own the memory committed to it. This applies where that memory is RAM pages from the system's free pool. If I/O devices will be committed to the chunk or RAM set aside at system boot time, then `iOwnsMemory` should be set to false.

You can supply a pointer to a DFC, `iDestroyedDfc`. This DFC is then called when the chunk is destroyed.

The members `iSpare8[3]` and `iSpare32[2]` are reserved for future expansion.

Destroying a shared chunk

Chunks are reference-counted kernel objects. When the kernel creates them, it sets the reference count to one, and each subsequent open operation increases this count by one. Closing a chunk decrements the access count by one. When the count reaches zero, the chunk is destroyed. Shared chunks are closed using the following function:

```
TBool Kern::ChunkClose(DChunk* aChunk);
```

The parameter aChunk is a pointer to the chunk that is to be closed. If this results in the chunk object being destroyed then the function returns true, otherwise it returns false.

The kernel may destroy chunks asynchronously, and so they may still exist after the close function returns. If you need to know when a chunk is actually destroyed, then you should supply a DFC when you create the chunk, using the member `iDestroyedDfc` of the `TChunkCreateInfo` argument. The kernel then queues the DFC when it finally destroys the chunk, which is after the point when the kernel guarantees that the memory mapped by the chunk will no longer be accessed by any program.

Committing memory to a shared chunk

Once a shared chunk has been created, you need to commit either RAM or I/O devices to it before it can be used. We provide four functions for this.

The first function that I show is used to commit a set of RAM pages with physically contiguous addresses from the system's free pool of memory:

```
TInt Kern::ChunkCommitContiguous(DChunk* aChunk, TInt aOffset, TInt aSize, TUint32&
aPhysicalAddress);
```

The argument `aChunk` is a pointer to the chunk into which the memory should be committed.

The argument `aSize` holds the size of the region to commit, and `aOffset` holds an offset within the chunk's linear address space that should become the start of the committed memory region. The units for both these arguments are bytes and both must be a multiple of the MMU page size. (Use the function `Kern::RoundToPageSize(TUint32 aSize)` to round up to the size of an MMU page).

On return, the final argument, `aPhysicalAddress`, is set to the physical address of the first page of memory that has been committed. This is useful for DMA transfer. By using `aPhysicalAddress` as a base, you can specify memory locations within the committed area in terms of physical address, saving the DMA framework from the overhead of converting from a linear address.

This function can create a buffer within the shared chunk, which is equivalent to the physically contiguous buffer created in the example code in Section 13.1.3.

We provide a similar function, which commits an arbitrary set of RAM pages from the system's free pool. In this case these aren't necessarily physically contiguous:

```
TInt Kern::ChunkCommit(DChunk* aChunk, TInt aOffset, TInt aSize);
```

You can use a third function to commit a specified physical region to a shared chunk. For example, a region that represents memory mapped I/O or RAM that was set aside for special use at boot time:

(C) Copyright Nokia 2014. All rights reserved.

```
TInt Kern::ChunkCommitPhysical(DChunk* aChunk, TInt aOffset, TInt aSize, TUint32
aPhysicalAddress);
```

The first three arguments are identical to those described for the first version of the function. The fourth argument, aPhysicalAddress, is the physical address of the memory to be committed to the chunk (which again must be a multiple of the MMU page size).

The fourth function is similar, except that this time the physical region is specified as a list of physical addresses. The list must contain one address for each page of memory to be committed, with the length of the list corresponding to size of the region to be committed:

```
TInt Kern::ChunkCommitPhysical(DChunk* aChunk, TInt aOffset, TInt aSize, const TUint32*
aPhysicalAddressList);
```

Providing access to a shared chunk from user-side code

As I have already mentioned, before a user-side application can have access to the memory in a shared chunk, the kernel must create a handle to the chunk for it. The following function is used to create such a handle. If successful, the function also maps the chunk's memory into the address space of the process to which the specified thread belongs. It also increases the access count on the object:

```
TInt Kern::MakeHandleAndOpen(DThread* aThread, DObject* aObject)
```

The argument aThread specifies the thread which is to own the handle and aObject specifies the shared chunk to which the handle will refer. The function returns the handle (that is, a value greater than zero) if it is successfully created. Otherwise, a standard error value (less than zero) is returned.

The handle is normally passed back to the user thread, where it is assigned to an `RChunk` object, using one of the methods derived from `RHandleBase` - either `SetHandle()` or `SetReturnedHandle()`.

Once this has happened, it normally becomes the responsibility of that application to close the handle once it no longer needs access to the shared chunk.

Providing access to a shared chunk from kernel-side code

A user application that has obtained access to a shared chunk from one device driver may wish to allow a second driver access to that shared chunk. In this case, the second device driver needs a method to obtain a reference to the chunk object and the addresses used by the memory it represents. Before code in the second device driver can safely access the memory in the shared chunk, it must first open that chunk. Once this is done, the reference counted nature of chunk objects means that the shared chunk and its memory will remain accessible until it closes the chunk again.

A user application can pass a shared chunk handle to a device driver, which can then use the following function to open it:

```
DChunk* Kern::OpenSharedChunk(DThread* aThread, TInt aChunkHandle, TBool aWrite);
```

The argument aChunkHandle supplies the handle value, and aThread is the thread in which this is valid. You use the Boolean aWrite to indicate whether you intend to write to the chunk memory or not. To give an example of how you might use this argument, imagine that the user application intends to write to a chunk that contains read only memory - in this case, the error that is returned can be handled gracefully when the chunk is opened rather waiting until a fault occurs much later on.

The function returns a pointer to the chunk if the chunk handle is valid for the thread concerned, is of the correct shared chunk type and opens successfully. If the function is successful in opening the chunk, the access count on the chunk will of course be incremented.

In some cases, a user application and a driver would have been able to transfer data using a shared chunk, except that the driver only supports a descriptor-based API, rather than an API designed specifically for shared chunks. A similar scenario is where driver and user application can transfer data via a shared chunk, but the user application obtained the data source or destination

(C) Copyright Nokia 2014. All rights reserved.

information from another application, and so this was presented to it as a normal descriptor rather than a chunk handle. In both cases, the driver will receive information on the data, via a descriptor, in the form of an address and a size.

If the driver wishes to optimize the case in which the data address resides in a shared chunk, it won't be able to use the open function I described previously, since it doesn't have a chunk handle. Instead it can make use of the following method below to speculatively attempt to open a shared chunk:

```
DChunk* Kern::OpenSharedChunk(DThread* aThread, const TAny* aAddress, TBool aWrite,
TInt& aOffset);
```

If the address aAddress supplied is within a shared chunk that is mapped to the process associated with thread aThread, then the function returns a pointer to the chunk. If not, then it returns zero. When a chunk pointer is returned, the chunk access count is incremented and the argument aOffset returns the offset within the chunk corresponding to the address passed.

Let's make this clearer with an example. Suppose we have a media driver that is designed to optimize data transfer to data addresses that are within a shared chunk. For example, a request might come via the file server from a multimedia application to save data to a file from a buffer in a shared chunk. The file server and media driver only support descriptor-based APIs, but if the driver uses the `Kern::OpenSharedChunk()` function, then we can still optimize the transfer using the shared chunk.

Once the driver has opened the chunk, it next needs to obtain the address of the data within it. Remember that shared chunks may contain uncommitted regions (gaps) and the driver needs to detect these to avoid making an access attempt to a bad address, which would cause an exception. There are two functions provided for this - the first obtains just the linear address, the second also obtains the physical address. Taking the first of these:

```
TInt Kern::ChunkAddress(DChunk* aChunk, TInt aOffset, TInt aSize, TLinAddr&
aKernelAddress)
```

If chunk aChunk is a shared chunk, and if the region starting at offset aOffset from the start of the chunk and of size aSize (both in bytes) contains committed memory, then the function succeeds. In this case, the argument aKernelAddress returns the linear address in the kernel process corresponding to the start of the specified region. However, if the region isn't within the chunk, or the whole region doesn't contain committed memory, then an error is returned. Now the second function:

```
TInt Kern::ChunkPhysicalAddress(DChunk* Chunk, TInt aOffset, TInt aSize, TLinAddr&
aKernelAddress, TUint32& aMapAttr, TUint32& aPhysicalAddress, TUint32* aPageList)
```

The first four arguments are identical to those described for the previous function. If the function is successful, the argument aMapAttr will contain the mapping attributes that apply for the chunk, and the argument aPhysicalAddress will contain the physical address of the first byte in the specified region. The argument aPageList returns the addresses of each of the physical pages that contain the specified region.

An example driver using a shared chunk for DMA transfer

To illustrate how shared chunks can be used, let us consider as an example a minimal device driver for an unspecified peripheral. Our driver supports only one type of request - the asynchronous transmission of data out of the phone from a memory buffer residing within a shared chunk. The user-side interface to the driver is as follows:

```
const TInt KMyDeviceBufSize=0x2000; // 8KB
class RMyDevice : public RBusLogicalChannel
    {
public:
    enum TRequest
        {
        EWriteBuf=0x0,
        EWriteBufCancel=0x1,
        };
    enum TControl
        {
```


(C) Copyright Nokia 2014. All rights reserved.

```
        EGetChunkHandle,
        EGetBufInfo,
        };
#ifndef __KERNEL_MODE__
public:
    inline TInt Open();
    inline TInt GetChunkHandle(RChunk& aChunk);
    inline TInt GetBufInfo(TInt aBufNum,TInt& aBufOffset);
    inline void WriteBuffer(TRequestStatus& aStatus, TInt aBufNum,TUint aBufOffset,TInt
aLen);
#endif
    };
```

The driver creates a shared chunk when it is opened. At the same time, it commits memory to two separate buffers within the chunk, each of size `KMyDeviceBufSize`, and each containing physically contiguous RAM pages.

To gain access to these buffers, the user application must first create a handle on the chunk, using the method `GetChunkHandle()`. This maps the chunk's memory into the address space of the process. The application obtains a pointer to the base of this region using the method `RChunk::Base()`. The application must then determine the offsets of the two buffers relative to this base address using the method `GetBufInfo()` - the argument aBufNum specifying the buffer and aBufOffset returning its offset.

Now the application can fill a buffer with the transmission data - taking care not to access beyond the perimeter of the buffer, as this would normally cause it to panic. Finally, the application issues a request to transmit the contents of the buffer using the method `WriteBuffer()`. The argument aBufNum identifies which buffer should be used for the transfer and the argument aBufOffset provides the offset within this to start the transfer from. Argument aLen provides the number of bytes to transfer.

When exchanging information on data locations within a shared chunk between driver and user code, you must always take care to specify this information as an offset rather than an address, since the chunk appears at different addresses in the address spaces of the different processes. Of course, the same applies when exchanging this information between user processes.

Again, since this user-side interface header is also included in the kernel-side implementation of the driver, I use #ifndef_*KERNEL_MODE*_ around the user-side specific methods to prevent compiler errors when building the kernel-side driver - see Section 12.4.6.1 for more details.

Here is the driver class definition:

```
// Driver object making use of shared chunks
class DMyDevice : public DBase
    {
    ...
private:
    TInt CreateChunk(TInt aChunkSize);
    void CloseChunk();
    TInt MakeChunkHandle(DThread* aThread);
    TInt InitDma();
    static void WrDmaService(DDmaRequest::TResult aResult, TAny* aArg);
    TInt InitiateWrite(TInt aBufNo,TUint aBufOffset, TInt aLen,TRequestStatus* aStatus);
private:
    DThread* iClient;
    TRequestStatus* iWrStatus;
    DChunk* iChunk;
    TLinAddr iChunkKernAddr;
    TUint32 iBuf1PhysAddr;
    TUint32 iBuf2PhysAddr;
    TDmaChannel* iDmaChannel;
    DDmaRequest* iDmaRequest;
    };
```


(C) Copyright Nokia 2014. All rights reserved.

The member `iChunk` is a pointer to the shared chunk created and `iChunkKernAddr` is the base address of this in the kernel process. The member `iClient` is the user thread that opened the channel. This will be used when creating a handle on the shared chunk for that thread. The members `iBuf1PhysAddr` and `iBuf2PhysAddr` save the physical addresses of the two buffers. This information will allow us to specify physical rather than linear addresses for DMA data transfers from these buffers, which is more efficient.

Operations on opening and closing the driver

The following code shows how the driver creates the shared chunk and commits memory to the pair of buffers. It commits physically contiguous RAM pages and disables caching. Each buffer is the shared chunk equivalent of that created in the example code, shown in Section 13.1.3. In this case, we leave an uncommitted page either side of each buffer; these act as guard pages. If the user application writes beyond the buffer region when it is filling one of the buffers with the transmission data, this will panic the application rather than corrupting adjacent memory regions in the chunk:

```
TInt DMyDevice::CreateChunk(TInt aChunkSize)
    {
    // Round the chunk size supplied upto a multiple of the
    // MMU page size. Check size specified is large enough.
    aChunkSize=Kern::RoundToPageSize(aChunkSize);
     __ASSERT_DEBUG(aChunkSize>=((3*KGuardPageSize)+
(KMyDeviceBufSize<<1)),Panic(KMyDevPanicChunkCreate)); // Thread must be in critical
section to create a chunk
    NKern::ThreadEnterCS(); // Create the shared chunk.
    TChunkCreateInfo info;
    info.iType = TChunkCreateInfo::ESharedKernelMultiple;
    info.iMaxSize = aChunkSize;
    info.iMapAttr = EMapAttrFullyBlocking; // No caching
    info.iOwnsMemory = ETrue; // Using RAM pages
    info.iDestroyedDfc = NULL; // No chunk destroy DFC
    DChunk* chunk;
    TUint32 mapAttr;
    TInt r = Kern::ChunkCreate(info,chunk,iChunkKernAddr, mapAttr);
    if (r!=KErrNone)
        {
        NKern::ThreadLeaveCS();
        return(r);
        }
    // Map two buffers into the chunk – each containing
    // physically contiguous RAM pages. Both buffers
    // surrounded by 4K guard pages.
    TInt bufOffset=KGuardPageSize;
    r=Kern::ChunkCommitContiguous(chunk,bufOffset, KMyDeviceBufSize,iBuf1PhysAddr);
    if (r==KErrNone)
        {
        bufOffset+=(KMyDeviceBufSize+KGuardPageSize);
        r=Kern::ChunkCommitContiguous(chunk,bufOffset, KMyDeviceBufSize,iBuf2PhysAddr);
        }
    if (r!=KErrNone) Kern::ChunkClose(chunk); // Commit failed – tidy-up.
    else iChunk=chunk;
    NKern::ThreadLeaveCS();
    return(r);
    }
```

The following code shows how the driver closes the chunk again:

```
void DMyDevice::CloseChunk()
    {
```


(C) Copyright Nokia 2014. All rights reserved.

```
    // Thread must be in critical section to close a chunk
    NKern::ThreadEnterCS();
    // Close chunk
    if (iChunk) Kern::ChunkClose(iChunk); 
    // Can leave critical section now
    NKern::ThreadLeaveCS();
    }
```

Next we see how the driver initializes the DMA objects required for data transfer. First it opens a DMA channel for data transfer. In this simple example, it only asks the framework to reserve one descriptor, since we assume a controller supporting a single buffer scheme and we allow only one DMA request to be outstanding at any time.

KPlatDevice1TxChan is the platform-specific channel identifier, which in this example selects data transmission over the peripheral device concerned. The driver elects to use DFC thread 0 to queue the DMA channel's DFC. Next it constructs a single request object, specifying the callback function as `DMyDevice::WrDmaService()` and passing a pointer to itself as the callback argument:

```
TInt DMyDevice::InitDma()
    {
    // Open and configure the channel for data
    // transmission
    TDmaChannel::SCreateInfo info;
    info.iCookie = KPlatDevice1TxChan;
    info.iDesCount = 1;
    info.iDfcPriority = 4;
    info.iDfcQ = Kern::DfcQue0();
    TInt r = TDmaChannel::Open(info,iDmaChannel);
    if (r!=KErrNone) return(r);
    // We're only ever going to have one
    // outstanding transfer
    iDmaRequest = new DDmaRequest(*iDmaChannel, DMyDevice::WrDmaService, this);
    if (iDmaRequest == NULL) return(KErrNoMemory);
    return(KErrNone);
    }
```

Getting a handle on the shared chunk

Next we see the how the driver creates the chunk handle for the user-thread concerned:

The inline code that follows shows how the user application assigns the handle created for it by the driver to the `RChunk` object passed into the *get handle* method:

```
inline TInt RMyDevice::GetChunkHandle(RChunk& aChunk)
    {
    return aChunk.SetReturnedHandle (DoControl(EGetChunkHandle));
    }
TInt DMyDevice::MakeChunkHandle(DThread* aThread)
    {
    TInt r;
    // Thread must be in critical section to make a handle
    NKern::ThreadEnterCS();
    if (iChunk) r=Kern::MakeHandleAndOpen(aThread,iChunk);
    else r=KErrNotFound;
    NKern::ThreadLeaveCS();
    return(r);
    }
```


(C) Copyright Nokia 2014. All rights reserved.

DMA data transfer using the shared chunk

The user application initiates data transfer using the following method:

```
TInt RMyDevice::WriteBuffer(TRequestStatus& aStatus, TInt aBufNum, TUint aBufOffset,TInt
aLen);
```

Next we see how the driver initiates transfer over the DMA channel in response to this. To calculate the source memory address, it combines the buffer offset passed by the client with the physical address of the start of the buffer that it saved earlier. `KPlatDevice1TxIdis` the transfer destination information - an identifier for the peripheral concerned:

```
TInt DMyDevice::InitiateWrite(TInt aBufNo, TUint aBufOffset, TInt aLen,TRequestStatus*
aStatus)
    {
    // Validate buffer no, buffer offset
    // and length supplied
    iWrStatus=aStatus;
    // Platform specific code to enable TX on device
    TUint32 src=(aBufNo==1)?iBuf2PhysAddr:iBuf1PhysAddr;
    TUint32 dest=KPlatDevice1TxId;
    TInt r=iDmaRequest->Fragment((src+aBufOffset),
dest,aLen,(KDmaMemSrc|KDmaIncSrc|KDmaPhysAddrSrc),0);
    if (r != KErrNone) return(r);
    iDmaRequest->Queue();
    return(KErrNone);
    }
```

Finally we see the driver's DFC callback function handing the end of the DMA transfer. In this example, it simply checks whether the transfer was successful or not and completes the request back to the user application. In a more complex implementation it might check if there is more data to be transferred:

```
void DMyDevice::WrDmaService(DDmaRequest::TResult aResult, TAny* aArg)
    {
    DMyDevice &driver = *((DMyDevice*)aArg);
    // Platform specific code to disable TX on device
    TInt r = (aResult==DDmaRequest::EOk) ? KErrNone : KErrGeneral;
    Kern::RequestComplete(driver.iClient, driver.iWrStatus,r);
    }
```

Media drivers and the local media sub-system

Local media sub-system overview

Media drivers are a form of PDD (physical device driver) that are used almost exclusively by the file server to access local media devices. Phones contain both fixed media devices that are internal to the phone such as NAND/NOR flash disks, and removable media devices such as MultiMediaCards and SD cards. The set of media drivers installed on a device, together with a local media LDD (logical device driver) and a user-side interface class, are referred to as the local media sub-system. Figure 13.5 shows an overview of the architecture. In this example, I show a Symbian OS phone containing three local drives:

1. A NAND user data drive (C:)
2. A MultiMediaCard drive (D:)
3. Code stored in NAND (Z:).

As we saw in Section 9.3.3.1, the file server supports at most 26 drives, each identified by a different drive letter (A: to Z:). For the file server, the `TDrive` class is the abstraction of a logical drive, and when a drive is mounted, this class provides the interface with

 (C) Copyright Nokia 2014. All rights reserved.

the associated file system. Of the 26 drives supported, 16 are allocated as local drives - that is, they are available for mounting drives on media devices that are located within the phone. This is more than on EKA1, which only supports nine local drives.

The interface to the local media sub-system is provided by the `TBusLocalDrive` class. Each instance of this user-side class represents a channel of communication with a local drive and to establish a channel, a client must connect a `TBusLocalDrive` object to a specified drive. A single instance of the `TBusLocalDrive` class can be switched between different drives.

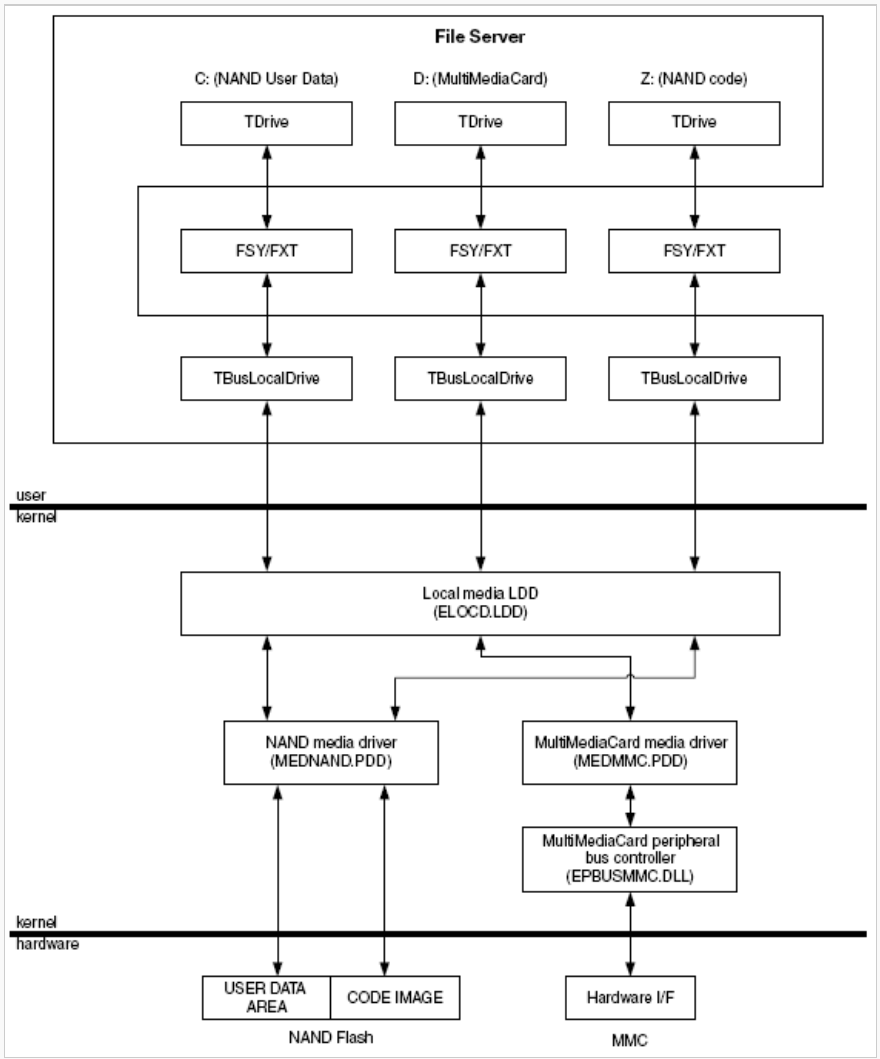

Figure 13.5 *Local media sub-system overview*

The file server always contains 16 separate `TBusLocalDrive` instances - one for each local drive. Those drive objects that correspond that two or more channels can be open on the same drive simultaneously.

Local drives are distinguished by their drive number (0-15). ESTART is an executable started during system boot, which completes the initialization of the file server and is responsible for handling the mapping between drive letter and the local drive number. This can be configured for each platform. However, apart from the emulator, most platforms adopt the default local drive-mapping scheme, which is:

Local drive number	Drive letter
0	C:
1	D:
2	E:
. . .	. . .
14	Q:
15	R:

Figure 13.5 shows drive Z: mapped to a local drive, which seems to deviate from the mapping scheme I've just described. In fact, this mapping to Z: happens because the composite file system combines the ROFS and ROM drives into a single drive designated as Z: - see Section 9.4.5. Without the composite file system, the ROFS local drive would be mapped to a drive letter in the range shown in the previous table.

The rest of the local media sub-system consists of kernel-side components. This includes a logical device driver layer called the

(C) Copyright Nokia 2014. All rights reserved.

local media LDD (ELOCD.LDD) together with a set of installed media drivers, which are essentially physical device drivers. However, the local media sub-system differs in a number of ways from a standard device driver configuration, as I will now describe.

The local media LDD abstracts various aspects of an interface with a local media device: for example, the handling of disks that have been divided into more than one partition. This LDD is involved in any connection to a local drive - which means that any functionality specific to a particular family of media device (NAND, MultiMediaCard and so on) is implemented in the media driver rather than the LDD. The result is that rather than each media driver abstracting just the platform specifics, it generally also includes a portion that is generic across those platforms that support the same family of media. Indeed, certain media drivers don't directly control the hardware interface at all - instead they use the services provided by a peripheral bus controller (see Section 13.4) that handles hardware interfacing. Such media drivers then become completely platform-independent and are built as part of the set of generic E32 components. An example of this is the MultiMediaCard driver, which uses the generic services of the MultiMediaCard peripheral bus controller.

Other media drivers do control the hardware interface themselves, and so contain both platform-specific and generic elements. These drivers are built as part of the platform variant, but they do include generic source files from E32 as well as variant-related source. The NAND flash media driver is an example of this type of driver. As with most platform-specific components, this type of media driver interfaces with the media hardware via functions exported from the variant or ASSP DLLs.

The EKA2 local media sub-system architecture differs from that provided on EKA1, where there is no local media LDD. The EKA1 architecture is less modular as in this case the kernel contains the equivalent functionality.

Figure 13.5 shows the file server mounting the two NAND device partitions as two separate drives. When both are connected, two open channels exist on the NAND device. However, rather than this resulting in two separate PDD objects, requests for both channels are fed into a single media driver PDD object. This is another aspect that differs from a standard driver configuration.

Before it is possible to connect to a local drive, a media driver or kernel extension must have registered for that drive. Registering involves specifying a local media ID that identifies the media device family. After this is done, only media drivers that support that particular family will open on that drive. On a particular platform, there may be multiple media drivers present for a certain media ID. A media driver may support only a small sub-set of the media devices within that family: for example, the sub-set might be constrained to devices from a particular manufacturer, or to devices with a particular part number. So a ROM image might include two versions of a NAND media driver to support two different NAND parts that the phone could be built to contain. However, other media drivers will include support for a wider sub-set and some drivers, for example the MultiMediaCard media driver, aim to support the whole family of devices.

The set of media `IDs` that are supported and the list of local drives that are allocated to each ID are highly dependent on the target hardware platform. Each variant includes the header file, variantmediadef.h, where this information is specified.

Removable media drives normally involve a Symbian OS peripheral bus controller as well as a media driver to manage the removable media bus. Here, the platform-specific code lies in the controller extension rather than the media driver, and so it is normally the controller that registers for such drives. In this situation, there could be multiple media drivers associated with that controller, each supporting a different type of removable memory card. For example, a platform including the SD card controller may contain drivers for both the user and protected area SD card sections.

The following table lists the possible media `IDs`. The association between ID and media device family can vary between platforms. What is essential is that each media family supported on the platform has a unique ID. However, the most common media types supported on Symbian OS have acquired default `IDs` which are shown in the table:

Local media ID	Default media device family
`EFixedMedia0`	Internal RAM
`EFixedMedia1`	NOR flash
`EFixedMedia2`	NAND flash
`EFixedMedia3`	-
`EFixedMedia4`	-
`EFixedMedia5`	-
`EFixedMedia6`	-
`EFixedMedia7`	-
`ERemovableMedia0`	MultiMediaCard/SD
`ERemovableMedia1`	PC card
`ERemovableMedia2`	Code Storage Area (SDIO)

(C) Copyright Nokia 2014. All rights reserved.

Note that the definition of media `IDs` for removable media devices has altered between EKA1 and EKA2. On EKA1, the ID indicates the slot (or socket) number rather than the media device family.

Media drivers and the local media LDD are generally built as combined device driver and kernel extension components. Being extensions means that the kernel will call their DLL entry points early in its boot process (before the file server is started), and it is at this stage that each media driver registers itself for one or more of the local drives.

Later on, during the initialization of the file server, a separate F32 startup thread runs, and this continues local media sub-system initialization. It loads the local media LDD and then attempts to load all media drivers it finds, by searching for *MED*.PDD* in the system directory (\Sys\Bin) on the ROM file system (Z:). Like any other drivers, media drivers and the local media LDD export a function at ordinal 1 to create a driver factory object - and the kernel calls this export for each driver as they are loaded. Once the relevant factory objects have been created, it becomes possible to connect to the corresponding local drives. ESTART completes the initialization of the file server. As well as being responsible for configuring the mapping between drive letter and the local drive number, it is also responsible for assigning an appropriate file system, and possibly a file server extension to each active drive. However, this has to be co-ordinated with the media ID assigned for each of these drives - that is, with the contents of variantmediadef.h for the platform concerned.

ESTART may use one of two methods for determining this local drive file system configuration. The first is to use a platform-specific local drive mapping file - an ASCII text file which specifies precisely which file system/extension to associate with which local drive. (This can also be used to customize the mapping between drive letter and the local drive number.) The second method is to allow ESTART to automatically detect which file system to mount on which local drive, by allowing it to interrogate the capabilities of each local drive and then use this information to decide an appropriate FSY. This second scheme is not as efficient as the first and therefore tends only to be used for development reference platforms, where the flexibility of drive mapping is more important than the time taken to boot the system. The local drive file system configuration performed by ESTART is discussed further in Chapter 16, *Boot Processes*.

During ESTART, the file server connects to all the active drives on the platform and reads their drive capabilities. So before file server initialization is complete, media drivers will normally be open on all these drives.

User-side interface class

Figure 13.6 shows the derivation of the `TBusLocalDrive` class. Normally, the user-side interface to a device driver consists solely of an `RBusLogicalChannel`-derived class containing only inline methods. In this case, `RLocalDrive` provides this thin class. However, here we further derive `TBusLocalDrive` from `RLocalDrive` to provide the local media user interface and this contains functions exported from the user library (EUSER.DLL). `TBusLocalDrive` adds code to handle the user-side processing of drive format and password protection operations.

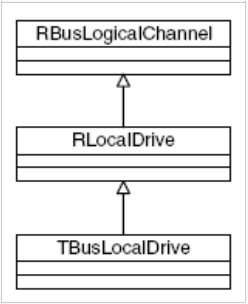

Figure 13.6 *Derivation of* `TBusLocalDrive`

However, the main reason for the derivation of `TBusLocalDrive` from `RLocalDrive` is to preserve compatibility with EKA1. It is needed there so that it can perform the far greater amount of user-side processing, which is necessary to cope with the issues associated with handling asynchronous I/O in device drivers. These are the major elements of the public interface to the `TBusLocalDrive` class:

```
class TBusLocalDrive : public RLocalDrive
    {
public:
    TBusLocalDrive();
    TInt Connect(TInt aDriveNumber, TBool& aChangedFlag);
    void Disconnect(); TInt Caps(TDes8& anInfo);
```


(C) Copyright Nokia 2014. All rights reserved.

```
    TInt Read(TInt64 aPos,TInt aLength,const TAny* aTrg, TInt aMessageHandle,TInt
aOffset);
    TInt Write(TInt64 aPos,TInt aLength,const TAny* aSrc, TInt aMessageHandle,TInt
aOffset);
    Format(TFormatInfo& anInfo); Format(TInt64 aPos,TInt aLength);
    TInt Enlarge(TInt aLength);
    TInt ReduceSize(TInt aPos, TInt aLength);
    TInt ForceRemount(TUint aFlags=0);
    SetMountInfo(const TDesC8* aMountInfo, TInt aMessageHandle)
    };
```

The method `Connect()` is used to open a channel to the specified local drive, aDriveNumber. The second parameter, aChangedFlag, is used to provide notification that a drive remount is required. Once the drive is connected, this flag is set true on each media change. When connecting to each local drive, the file server passes in a reference to the data member `iChanged` belonging to the corresponding `TDrive` object, and this is how it receives notification of a possible change of volume - see Section 9.3.3.1. The method `Disconnect()` dissociates the object from any drive.

Next I will list the standard local media operations. The `Caps()` method returns information on the capabilities of a connected drive. Three forms of both the read and write methods are provided (although I've only listed one of each for brevity). The read version shown is the one used for inter-thread communication. It fully specifies the target memory location:

```
TInt Read(TInt64 aPos,TInt aLength,const TAny* aTrg, TInt aMessageHandle, TInt aOffset);
```

This method reads aLength bytes from offset aPos on the drive. Parameter aTrg is a pointer to a target descriptor in memory and aOffset specifies the offset within this to start storing the data. Parameter aMessageHandle is a handle to the message object associated with the F32 client request and this allows the local media LDD to identify the target thread. The corresponding inter-thread write method is also shown.

Two versions of the `Format()` method are provided. The first is used when formatting the entire connected drive - that is, setting each memory element of the drive to a default state and detecting any hardware faults across the drive. The second method is used to format (or erase) just a specified region within the drive.

The methods `Enlarge()` and `ReduceSize()` are used to control the size of a variable sized disk - typically only used for internal RAM drives.

The method `ForceRemount()` is used to close the media driver currently associated with the drive and force the local media sub-system to reopen the most appropriate driver. This is useful in situations where a new media driver has recently been installed on the system. `ForceRemount()` is then used to replace the existing driver with the new version. Also, some media drivers may need to be supplied with security information to open. This is achieved using the `SetMountInfo()` function. `ForceRemount()` is then used to retry the opening the driver once the appropriate mount information has been supplied.

A second media change notification scheme, in addition to that provided via the `TBusLocalDrive::Connect()` method, is available from the base class `RLocalDrive`. This is the method:

```
RLocalDrive::NotifyChange(TRequestStatus* aStatus);
```

The file server also uses this second scheme. The active object `CNotifyMediaChange` makes use of it when requesting notification of media change events to pass on to F32 clients - see Section 9.4.3.4.

Local media LDD

Figure 13.7 shows the main classes that comprise the local media LDD. The diagram also includes the `TBusLocalDrive` class and the main NAND media driver class to show the relationships between the local media LDD and these other components.

I will now describe these classes.

The `DLocalDrive` class

The class `DLocalDrive` is the local drive logical channel abstraction. An instance of this class is created each time a

(C) Copyright Nokia 2014. All rights reserved.

TBusLocalDrive object is connected to a local drive, and destroyed each time it is disconnected. If two channels are connected to the same drive, then two instances of this class will exist. DLocalDrive is derived from the abstract base class for a logical channel, DLogicalChannelBase. In this case, however, the fact that it derives from this rather than DLogicalChannel does not imply that requests on the channel are always executed in the context of the client thread. Media drivers can also be configured to perform requests in a kernel thread - as we will see shortly.

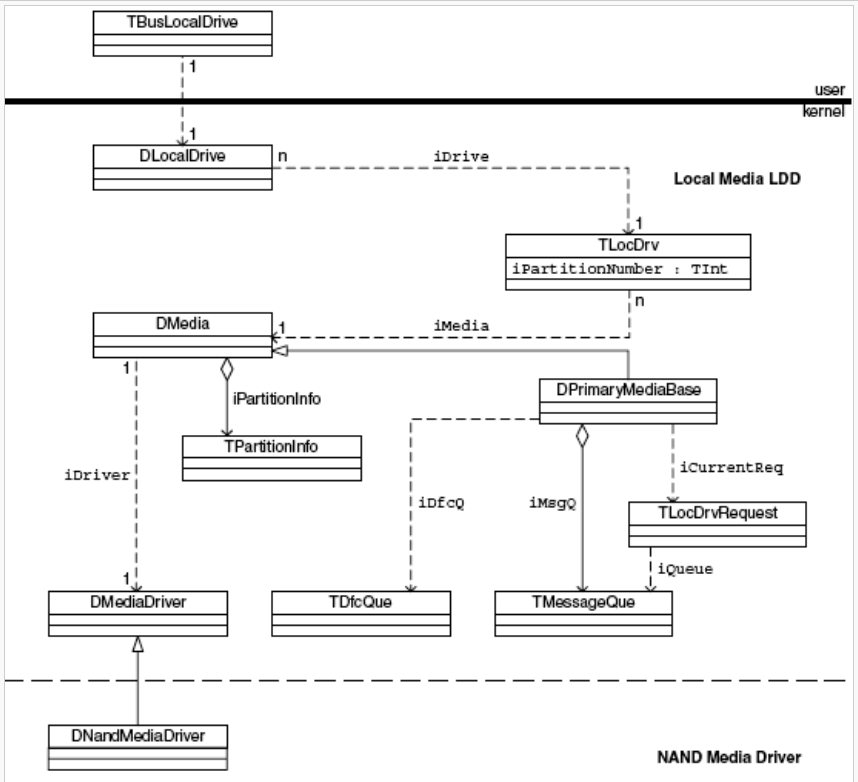

Figure 13.7 *Class diagram for the local media LDD*

The DLocalDrive class contains the member iDrive, a pointer to an associated TLocDrv object.

The TLocDrv class

TLocDrv encapsulates the local drive itself and, unlike the DLocalDrive class, there is always just a single instance of this class per local drive. When a driver or extension registers for a set of local drives (normally during kernel boot), a TLocDrv instance is created for each. Since it is not possible to de-register drives, these objects are never destroyed. Each TLocDrv instance represents an individual partition on a media device, so if a media device contains multiple partitions, then a separate TLocDrv object is required for each one.

The TLocDrv class contains the member iMedia, a pointer to its associated DMedia object.

The DMedia class

The DMedia class is the abstraction for a media device. It owns a single media driver, iDriver, which it uses to perform any access to the media hardware. It also owns a TPartitionInfo object, iPartitionInfo, which holds information on each of the partitions contained on the media. Here is the class definition:

```
class TPartitionEntry
    {
public:
    // Start address of partition, described as the relative
    //offset in bytes, from the start of the media.
    Int64 iPartitionBaseAddr;
    // The length of the partition, in bytes.
    Int64 iPartitionLen;
    // Boot Indicator record, currently unused.
    TUint16 iBootIndicator;
    // Describes the type of partition.
```


(C) Copyright Nokia 2014. All rights reserved.

```
    TUint16 iPartitionType;
    };

const TInt KMaxPartitionEntries=0x4;

class TPartitionInfo
    {
public:
    TPartitionInfo();
public:
    Int64 iMediaSizeInBytes; //Total size of media in bytes.
    TInt iPartitionCount; //No of partitions on media.
    TPartitionEntry iEntry[KMaxPartitionEntries];
    };
```

The `TLocDrv` class contains a partition number, `iPartitionNumber`. This indicates the element of the partition array in the associated `DMedia` object,TPartitionInfo::iEntry[], which holds the data for that partition. Local drive requests are specified in terms of an offset relative to the start of the partition. By retrieving partition information from the appropriate `DMedia` object, the request is converted into an absolute address on the media device and then passed on to the media driver.

When a driver or extension registers for a set of local drives, it must also specify the number of associated `DMedia` objects required. Hence, drive registration is also the point at which the `DMedia` objects are allocated and again, they are never destroyed. A peripheral bus controller must specify at least one separate media object for each card slot that it controls. Some peripheral bus controllers may need to register for more than one media object per card slot if they are required to support dual media cards. For example, a single SD card, containing both a user and a protected area, requires separate media drivers for both areas, and so needs at least two `DMedia` objects available. As far as Symbian OS is concerned, this is effectively a dual function, or dual media card.

Where removable media are concerned, the relationship between `TLocDrv` and `DMedia` objects can change as the user of the phone removes one memory card and inserts another. The SD card configuration that I've just described requires two `DMedia` objects, with each having a single associated `TLocDrvobject`. If this card were removed and replaced by a MultiMediaCard containing two partitions, then this would require only one `DMedia` object, with two associated `TLocDrv` objects.

Figure 13.8 shows various `TLocDrv` and `DMedia` combinations that could result from different cards being inserted into a single card slot.

The `DPrimaryMediaBase` class

In the previous section I described how each media driver or extension (for the remainder of this section I shall refer to these merely as drivers) that registers for a set of local drives also has to register for a set of `DMedia` objects at the same time. This media set must contain just one primary media object. This object is responsible for controlling the overall state of the media (for example, whether power is applied, whether the partition information has been determined and so on). The `DPrimaryMediaBase` class, which is derived from `DMedia`, provides this functionality. The driver that performs drive registration is responsible for creating the primary media object itself, which it then passes over to the local media sub-system for ownership. If further media objects are specified in the set, then the local media sub-system itself creates `DMedia` instances for these on behalf of the driver.

(C) Copyright Nokia 2014. All rights reserved.

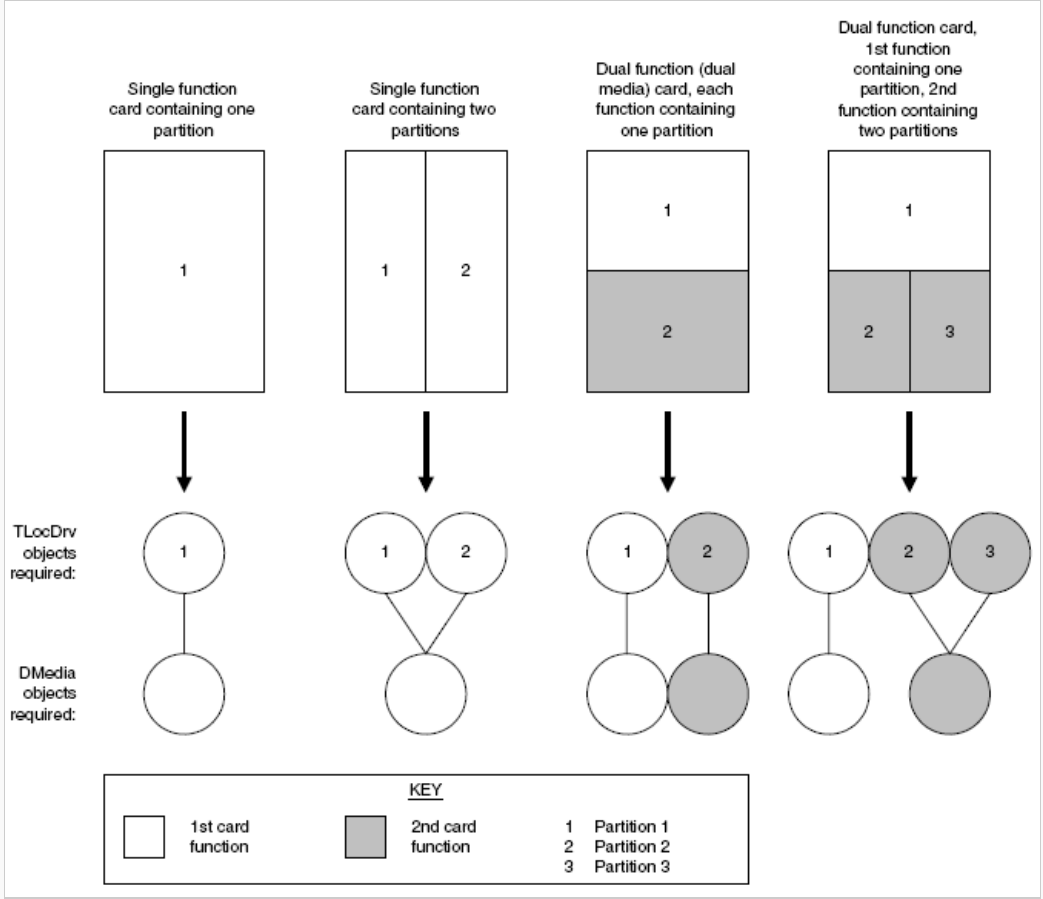

Figure 13.8 *`TLocDrv` and `DMedia` combinations*

The `DPrimaryMediaBase` class contains the member `iDfcQ`, a pointer to a DFC queue. As we have seen earlier in the book, a DFC queue is associated with a kernel thread. If the driver that creates the `DPrimaryMediaBase` object assigns a DFC queue to this member, then this configures the media set so that its requests are implemented in the context of the kernel thread associated with that DFC queue. The driver may use a standard kernel queue or create its own. If `iDfcQ` is left null, then this configures the media set so that its requests are executed in the context of the client thread.

Each local drive request is encapsulated as a `TLocDrvRequest` - a class derived from `TThreadMessage`, the kernel message class. A request ID is defined for every request type. `TLocDrvRequest` contains information pertaining to the request, including the ID and any associated parameters such as the drive position, length and source/destination location.

Requests for an entire set of `DMedia` objects are all delivered to the primary media object. This takes place in the context of the calling client thread (normally a file server drive thread). The `DPrimaryMediaBase` class owns a kernel message queue, `iMsgQ`. If the media is configured to use a kernel thread, then each request is sent to the queue and the client thread then blocks waiting for the message to complete. Meanwhile, in the context of the kernel thread, the request is retrieved from the queue and dispatched to the appropriate media driver for processing (which normally takes place within driver interrupt service routines and subsequent DFCs). If the media is configured to use the client thread, then requests are not queued, but instead dispatched straight to the media driver to be processed in the context of the client thread.

I discussed the differences between implementing driver requests in the context of the client thread or a kernel thread in Chapter 12, *Drivers and Extensions*.

Local drive power management

However, before a connected local drive is ready to process its first request, it must first be mounted. For certain media this can be a relatively long and complex task that is often handled asynchronously, while the client thread is blocked. It consists of the following phases:

1. Apply power and reset the media device, then wait for it to stabilize
2. Attempt to open each media driver loaded (installed). Each one that opens successfully is assigned to one of the media objects in the media set
3. Acquire the partition information for each media object for which a driver has opened successfully, and from this determine the relationship between `DMedia` and associated `TLocDrv` objects. This typically involves reading data from the media device itself.

(C) Copyright Nokia 2014. All rights reserved.

For media configured to use the client thread for execution (typically these are fixed media devices), drive mounting commences as soon as any local drive is connected to the media device. For media configured to use a kernel thread, drive mounting is deferred until the first request on the drive takes place - this generally being a request from the file server to read the drive capabilities.

The point at which drive dismounting occurs - that is, when all media drivers are closed for the media set and when power is removed - again depends on the type of media. For removable media devices, this is performed on each of the following occasions:

1. When a media removal event occurs, that is, the media door has been opened or the device has been removed
2. When the phone is being turned off or switched into standby mode
3. When a power-off request from a peripheral bus controller is received - it might do this after a period of bus inactivity to save power.

Cases 2 and 3 are collectively known as normal power down events.

In case 1, subsequent drive re-mounting does not occur until the first access to the drive after the door has been closed again. In case 2, it only occurs after the phone has been brought out of standby - on the first subsequent access to the drive. In case 3, it occurs on the next access to the drive. For removable media devices, closing and re-opening the media drivers is necessary in each of these power-down situations because the user could exchange the media device while power is removed. This is particularly likely, of course, in the case of a media removal event. An exchange could involve the introduction of a completely different type of media device into the phone. If so, then on a subsequent re-mounting of the drive, a different media driver will be opened (assuming that the new device is supported).

Irrespective of whether they are configured to use the client thread or a kernel thread for execution, it is likely that the drives for fixed media devices will remain mounted as long as there are `TBusLocalDrive` objects connected. In this situation, it is left to the media driver to implement its own power management policy, as it deems appropriate for the media - for example, power saving during periods of inactivity.

Before power is removed from a removable media device in response to a normal power down event, the local media LDD first notifies each of the affected media drivers of the impending power down. This is not the case on a media removal event.

Media change handling

The local media LDD is also responsible for providing user-side notification of media change events. When the peripheral bus controller notifies the local media LDD of either a media removal event or the presence of a card, following a door close event, then the local media LDD passes on this notification. Each event can potentially trigger both of the user notification schemes described in Section 13.3.2.

In the EKA1 version of the local media sub-system, the local media sub-system must also signal normal power down events to the user-side, as far as the `TBusLocalDrive` class, so that any subsequent drive-mounting may be performed asynchronously. This is no longer necessary with EKA2 since drive mounting can be handled asynchronously kernel-side.

Media drivers

A media driver is a special form of a physical device driver. The class `DMediaDriver` is the abstract base class from which all media drivers must be derived. Here are the major elements of the public interface to this class:

```
class DMediaDriver : public DBase
    {
public:
    DMediaDriver(TInt aMediaId);
    virtual ~DMediaDriver();
    virtual void Close();
    virtual TInt Request(TLocDrvRequest& aRequest)=0;
    virtual TInt PartitionInfo(TPartitionInfo &anInfo)=0;
     virtual void NotifyPowerDown()=0;
    void Complete(TLocDrvRequest& aRequest, TInt aResult);
    void PartitionInfoComplete(TInt anError);
    };
```

The method `Request()` is the main request handling method, which is called by the associated primary media object to deal with a

(C) Copyright Nokia 2014. All rights reserved.

request received for that drive. A reference to the corresponding request object is passed as a parameter.

Not all requests require access to the media hardware. Even when such access is required, requests can be processed very quickly for fast media memory such as internal RAM. However, any request that involves accessing the media hardware has the potential to be a long-running operation. Even just to read a few bytes, we may need to bring the device out of power saving mode, spin up a rotating disk and so on. To cope with this, the driver may complete requests either synchronously or asynchronously. The return value to the `Request()` method indicates the mode adopted, as follows:

Return value	Meaning
`KErrCompletion`	Request has been completed synchronously and the outcome was successful.
`KErrNone`	Request has been initiated successfully but is still in progress and will be completed asynchronously.
`KMediaDriverDeferRequest`	Request is not yet initiated since another is in progress - defer the request until later.
Other system-wide error code	Request has failed (during the synchronous phase of processing).

When a request is to be performed asynchronously, then its completion is signaled back to the LDD using the method `Complete()`.

The local media LDD calls the method `PartitionInfo()` during drive mounting to get partition information for the media device. Again, this operation may be performed either synchronously or asynchronously as indicated by the method's return value. If performed asynchronously then the method `PartitionInfoComplete()` is used to signal back completion to the LDD.

In response to a normal power down event, the local media LDD calls the method `NotifyPowerDown()` to allow the driver to terminate any outstanding requests and power down the device. However, for removable media devices, the peripheral bus controller takes care of powering down the bus.

Peripheral bus controllers

Symbian OS supports a number of peripheral bus standards for removable memory and I/O cards:

- MultiMediaCard
- SD card
- PC card
- Memory stick.

Symbian OS implements a software controller for each, and these controllers provide a set of generic kernel-side services that is available to device drivers and media drivers alike. There are many similarities between each of these peripheral bus systems: to share functionality common to each and to unify the interface to these components as far as possible, we have encapsulated these common characteristics into a set of abstract base classes for peripheral bus controllers which I will briefly describe in this section.

The features common to removable peripheral cards and their associated bus interface hardware include:

- Detection and handling of card insertion and removal events
- Support for the hot removal of cards - that is, removing a card when the bus is in operation
- Control of the bus power supply in relation to insertion and removal events and bus activity
- Adjustment of hardware interface characteristics according to the capabilities reported by the cards
- Rejection of cards that aren't compatible with the hardware interface - for example, low voltage cards, cards which consume too much current when active and so on
- Support for dual and multi-function cards.

Figure 13.9 shows part of the class diagram for a peripheral bus controller - the MultiMediaCard controller. It shows each of the peripheral bus controller base classes, and the classes derived from these for the particular MultiMediaCard controller case. It also shows some of the local media sub-system classes that I've already described, to show their relationships with the peripheral bus controller.

(C) Copyright Nokia 2014. All rights reserved.

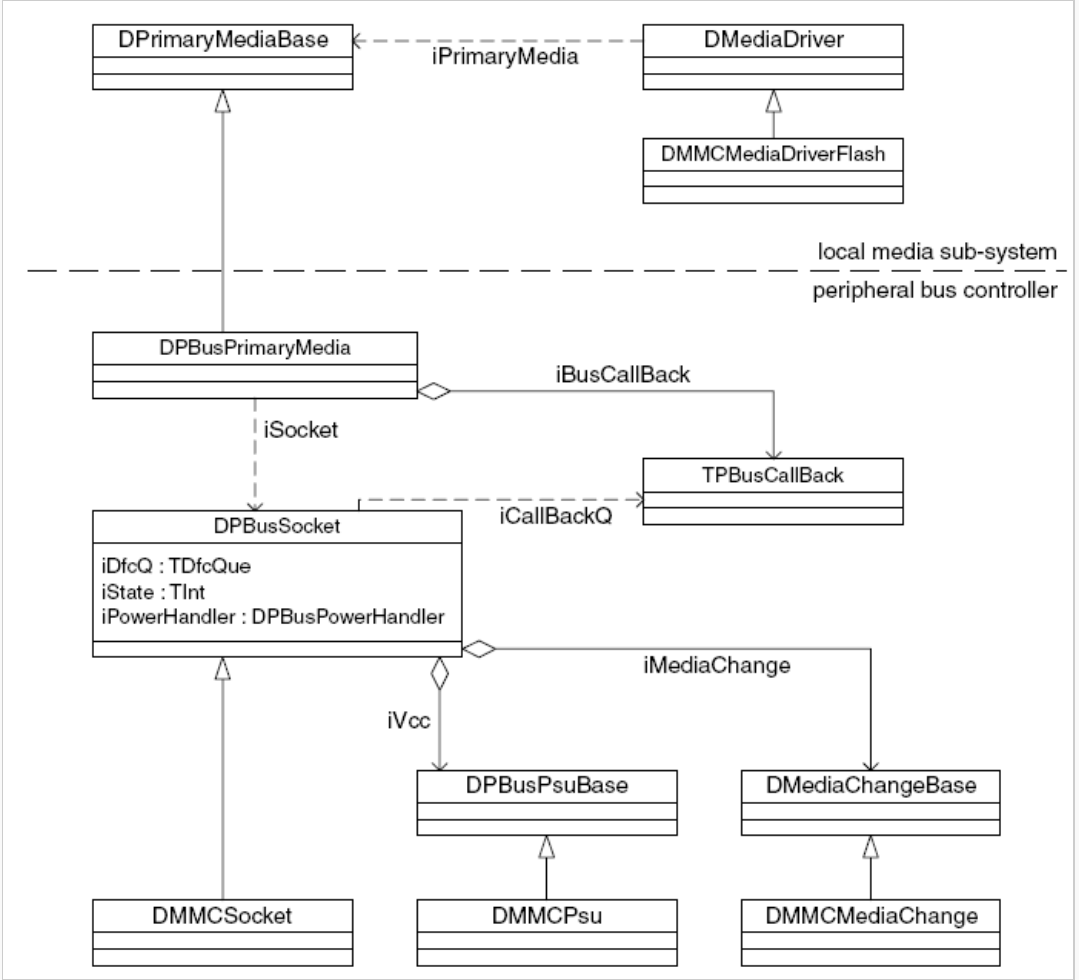

Figure 13.9 *(Class diagram for a peripheral bus controller (using the MultiMediaCard controller as an example)*

I discussed the `DPrimaryMediaBase` class in Section 13.3.3. Part of the local media sub-system, this is the object responsible for controlling the overall state of a media device or set of related media devices. For fixed media, this class is instantiated directly. However, for media involving a peripheral bus controller, a derived version is provided, DPBusPrimaryMedia. This class abstracts the interface between the local media sub-system and a peripheral bus controller - passing on requests from the sub-system to apply power to removable media devices and providing notification back of media change and power down events. The media driver base class, `DMediaDriver`, contains a pointer to its associated primary media object, `iPrimaryMedia.` For peripheral bus related media drivers (such as the MultiMediaCard media driver, DMMCMediaDriverFlash shown), this pointer is used to gain access to other peripheral bus objects via the DPBusPrimaryMedia object.

Associated with each DPBusPrimaryMedia object, there is a peripheral bus socket object, `iSocket.` This is a major element of every peripheral bus controller that essentially encapsulates a bus controller thread. Often, this also corresponds to a physical card slot for an individual removable media card - but not always. For example, if a platform contains two physical slots, each with separate hardware control, such that cards in both slots can be enabled and can actively be transferring data simultaneously, then each slot needs to be allocated a separate socket object. However, if the two slots are connected with a common set of control signals such that only one slot can be actively transferring data at any given time (as is the case with a MultiMediaCard stack), then the pair should be allocated a single socket object.

DPBusSocket is the abstract base class for a socket, with each type of controller providing a derived version - in this case a DMMCSocket class. The DPBusSocket class owns a DFC queue, `iDfcQ` and at system boot time each socket allocates itself a separate kernel thread to process DFCs added to this queue. I mentioned in Section 13.3.1 that peripheral bus controllers register for their associated removable media local drives, rather than leaving this to the relevant media drivers. Registering also involves setting a DFC queue for the primary media object, which is then used for handling all requests for these local drives. A peripheral bus controller always assigns this to the DFC queue of the relevant DPBusSocket object and so the socket's kernel thread is used for handling these local drive requests.

The DPBusSocket derived object oversees the power supply and media change functionality associated with the socket - owning an associated PSU object, `iVcc` and media change object, `iMediaChange.`

DPBusPsuBase is the abstract base class for the main bus power supply for the socket. Again, each type of controller provides a derived version - DMMCPsu in this case. The power supply can be set to one of three desired states:

(C) Copyright Nokia 2014. All rights reserved.

PSU state	Definition
`EPsuOff`	PSU is turned off.
`EPsuOnCurLimit`	PSU is turned on in a current limited mode: some supplies can be turned on in a mode that supplies a limited amount of current to the card. If a card draws excessive current then this causes PSU output voltage droop, which can be detected. Normally the PSU is only placed in this mode for a brief period, before being turned fully on. For PSUs that don't support current limit mode, this state is treated in the same way as `EPsuOnFull`.
`EPsuOnFull`	PSU is turned fully on.

While the supply is in either of its ON states, it can be configured to monitor the PSU output voltage level every second. The method used to perform voltage checking varies between platforms. If the voltage level goes out of range, then the PSU is immediately turned off. This PSU object also implements a bus inactivity timer (using the same 1 second tick). The controller resets the timer on each transfer over the bus. The PSU object can be configured so that if the timer is allowed to expire, this causes the associated socket to be powered down. The programmers creating a new phone platform set the duration of the inactivity period.

Similarly, each type of controller provides a derived version of the class `DMediaChangeBase`, which handles the insertion and removal of media on the socket. The derived class interfaces with the media change hardware - providing notification of media change events.

DPBusSocket also owns a power handler object, `iPowerHandler`. It registers this with the kernel-side power manager to receive notification of phone transitions into the standby or off state, and transitions out of standby, back into the active state.

The socket object combines status information from its power supply, media change and power handler objects into an overall power state, `iState`. The following six different socket states are defined:

Power state	Definition
EPBusCardAbsent	Either no card is present or the media door is open.
EPBusOff	The media door is closed and a card is present, but is not powered up.
EPBusPoweringUp	A request has been received from the local media sub-system or an I/O driver to power up the card and this is now in progress. This normally involves applying power, waiting for the PSU to stabilize, applying a hardware reset to the card and, finally, interrogating the capabilites of the card.
EPBusPowerUpPending	A request has been received to power up the card just as the phone is being placed in standby mode. Power up is deferred until the phone comes out of standby.
EPBusOn	The card has successfully been powered up and initialized.
EPBusPsuFault	In the process of powering up the card, it has been discovered that the power supply range for the card is not compatible with that of the host phone, or a hardware problem with the card has resulted in it drawing excessive current. The card is now powered off and no more power up requests will be accepted on this socket until a new card is inserted (that is, a media change event occurs).

I've assumed one physical card slot per socket object to simplify these descriptions.

Figure 13.10 shows the power state transition diagram. Referring still to Figure 13.9, clients of a controller, such as media drivers, use the TPBusCallBack class to implement peripheral bus event service routines.

These objects must be configured with details of the bus event concerned, and then queued on the appropriate socket object. The event of interest can be either a peripheral bus interrupt or a change in the status of the socket power state. Each TPBusCallBack object has an associated callback function supplied by the client, and, once queued, this is called on each occurrence of the event until the object is de-queued again. In the case of power state changes, information is passed to the callback indicating the new power state.

Each DPBusPrimaryMedia object owns a callback object, `iBusCallBack`, which it queues with the corresponding socket object for notification of power state changes. Of primary interest are card insertion/removal events, which it passes on to the local media LDD to trigger user-side media change notification. Power-down events are also signaled to the local media sub-system and lead to the relevant media drivers being closed, as do card removal events - see Section 13.3.3.5.

(C) Copyright Nokia 2014. All rights reserved.

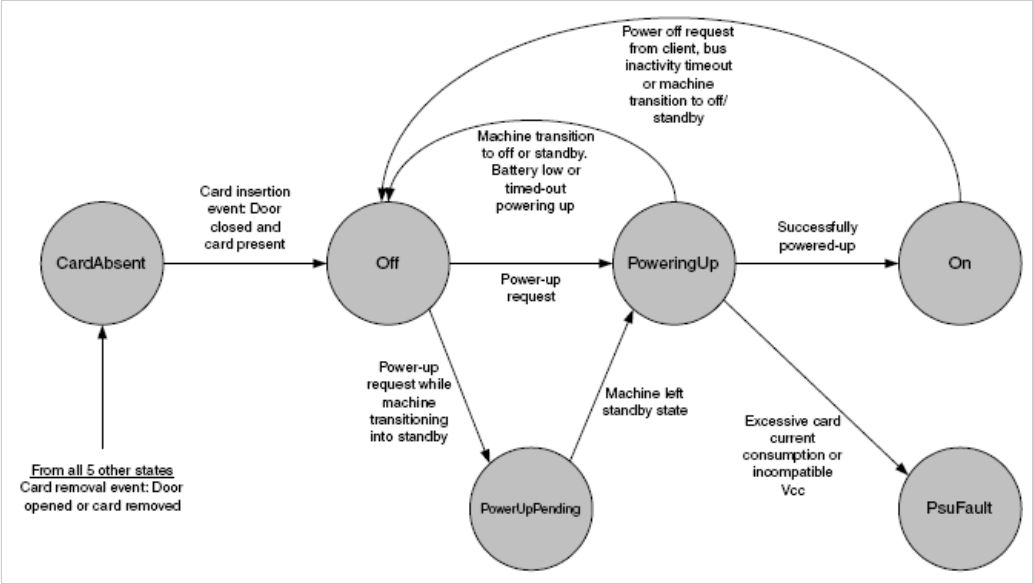

Figure 13.10 *Socket power state transition diagram*

MultiMediaCard support

MultiMediaCard overview

MultiMediaCards are miniature solid-state removable media cards about the size of a postage stamp. There are two main types:

1. Read-only memory (ROM) cards
2. Read/write cards - which generally use Flash memory.

Both types support a common command interface. In the case of Flash cards, this allows the host to write to any data block on the device without requiring a prior erase to be issued. This means that there is no need for the phone to implement a flash translation layer.

The standard MultiMediaCard provides a 7-pin serial bus for communication with the host computer. Cards may support two different communication protocols. The first is MultiMediaCard mode, and support for this is mandatory. The second, based on the Serial Peripheral Interface (SPI) standard, is optional and not supported on Symbian OS.

MultiMediaCard mode involves six of the card signals: three as communication signals and three which supply power to the card. The communication signals are:

Signal	Description
CLK	One bit of data is transferred on the CMD and DAT lines with each cycle of this clock.
CMD	Bidirectional command channel - used to send commands to the card and to receive back responses from it.
DAT	Bidirectional data channel - for data transfer between host and card.

This arrangement allows commands and responses to be exchanged over the CMD line at the same time that data is transferred over the DAT line. The maximum data transfer rate for standard cards is 20 Mbits/sec. However, high-speed 13-pin MultiMediaCards are available that can employ eight data signals, and here the maximum transfer rate is 416 Mbits/sec.

The MultiMediaCard architecture allows more than one card to be attached to a MultiMediaCard bus, with each card being connected to the same signals, and no card having an individual connection. A MultiMediaCard controller on the host machine - the bus master - controls this group of cards, known as a card stack. Communication over the bus begins with the controller issuing a command over the CMD line. There are two types of these: broadcast commands are intended for all cards, while, fairly obviously, addressed commands are intended for the addressed card only. Of course, many commands produce a response from the card. In the case of data transfer commands, such as the reading or writing of data blocks, data transfer commences over the DAT line after the command is issued. For normal multiple block transfers, this data flow is only terminated when the controller issues a stop command. Single block transfers end without the need for a stop command.

A minimal card stack that consists of only one card has a point-to-point connection linking that card and the controller, but you should be aware that this doesn't alter the communication protocol required.

Software MultiMediaCard controller

In the two previous sections, I introduced the software MultiMediaCard controller in Symbian OS, which provides a generic kernel-side API to media drivers. Figure 13.5 showed its position in relation to the local media sub-system. Here I will describe it in more

(C) Copyright Nokia 2014. All rights reserved.

detail.

Symbian OS also supports SD/SDIO cards, and we derive the software controller for these from the same MultiMediaCard classes. The MultiMediaCard specification also includes an I/O card class. For both of these reasons, we designed the MultiMediaCard controller to support I/O cards too. The clients for these I/O services are device drivers.

The MultiMediaCard controller is implemented as a peripheral bus controller, which means that we derive it from the peripheral bus controller base classes described in Section 13.4. Like the DMA framework, the MultiMediaCard controller is divided into a platform-independent layer (PIL) and a platform-specific layer (PSL). In this case, the two are built as separate kernel extensions, as shown in Figure 13.11. In this case too, the PSL normally interfaces with the controller hardware via functions exported from the variant or ASSP DLL.

The basic architecture of the MultiMediaCard controller and its relationship with a media driver is shown in Figure 13.12. Figure 13.12 omits the peripheral bus controller base classes, which I showed in Figure 13.9.

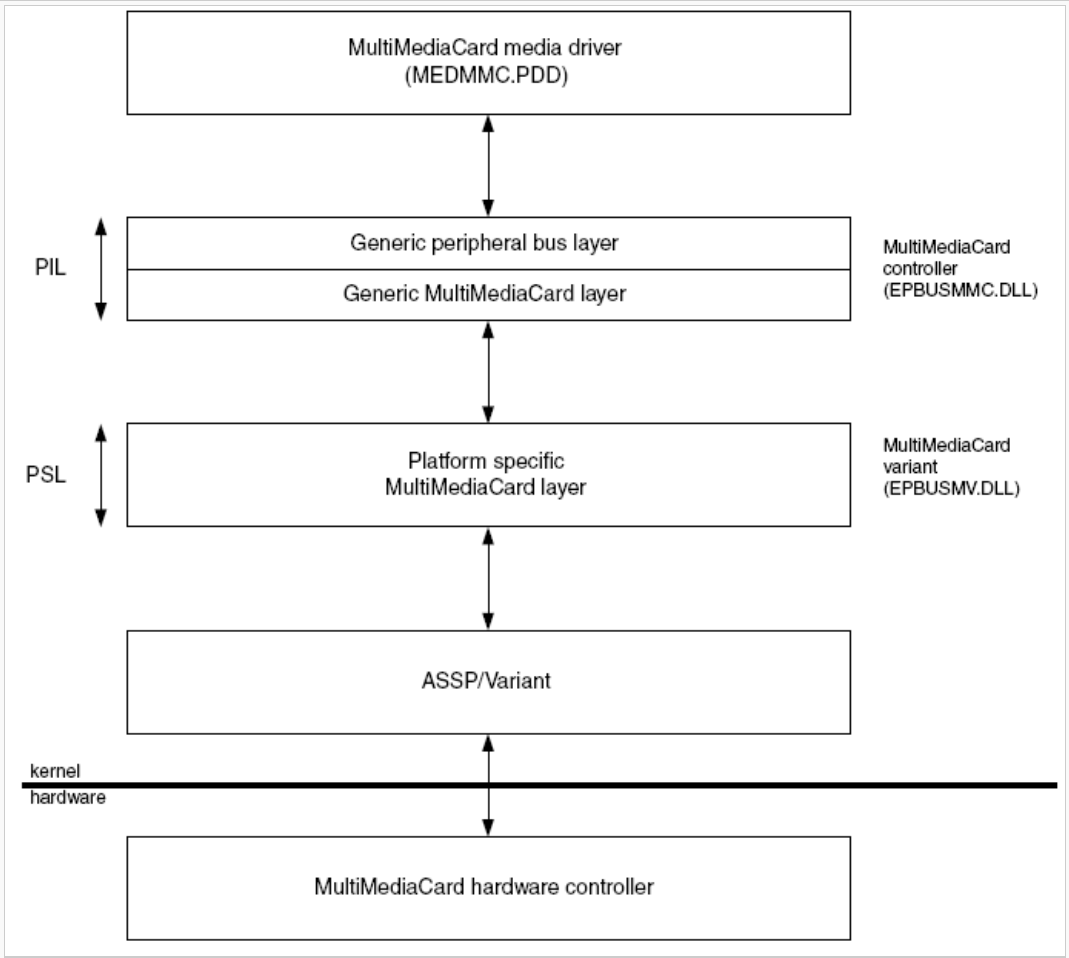

Figure 13.11 *The components of the MultiMediaCard controller*

On most phones, the MultiMediaCard controller manages only a single card stack - although it can be configured to control as many as four stacks. Each stack is implemented as a peripheral bus socket, which means that it has an associated kernel thread. We create an instance of the class `DMmcSocket` (derived from the peripheral bus socket base class DPBusSocket) for each stack. We make a distinction between this socket object - which oversees the bus power supply and media change functionality - and the object that controls access to the card stack, DMMCStack.

(C) Copyright Nokia 2014. All rights reserved.

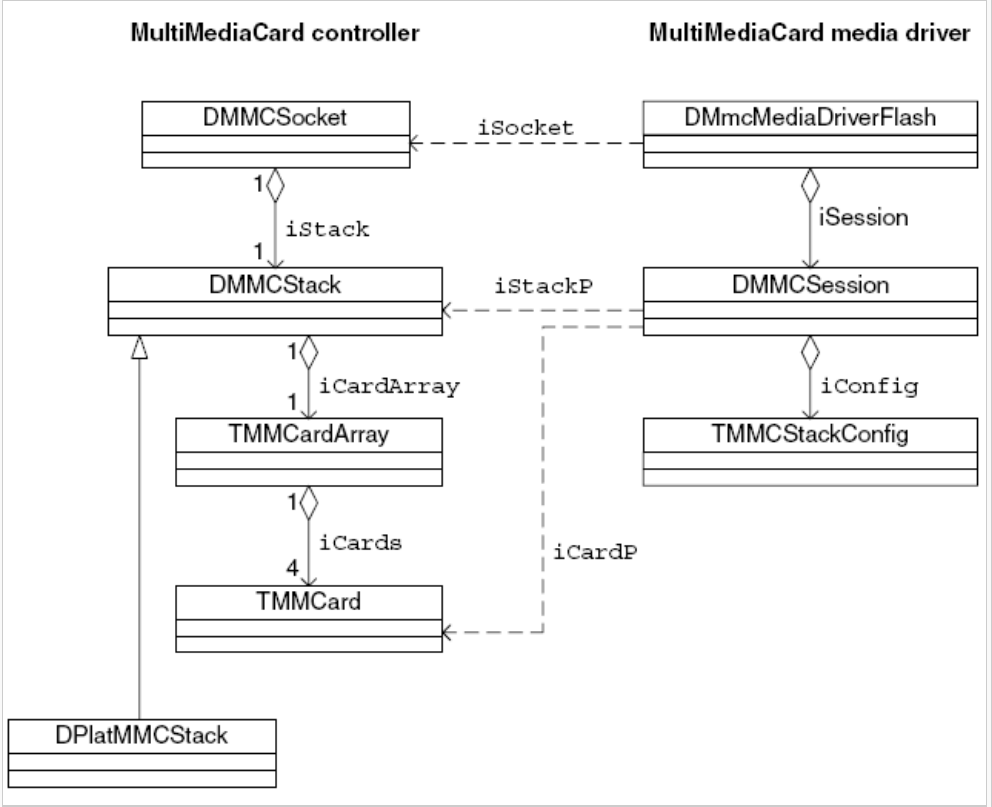

Figure 13.12 *Class diagrams for the MultiMediaCard controller and the MultiMediaCard media driver*

The DMMCStack class is responsible for issuing commands over the bus, receiving responses back from the cards, transferring card data, and the control of the bus clock speed. All of this involves the management of the MultiMediaCard hardware interface. The MultiMediaCard specification defines a set of predefined command sequences (called macro commands) for complex bus operations, such as identifying the cards present in the stack, reading more than one block from a card and so on. The DMMCStack class has been designed to implement these macro commands. It is an abstract class, which defines the main interface between the PIL and PSL. Each phone platform provides a derived stack object - shown as `DPlatMMCStack` on the diagram - which normally deals with such issues as hardware interface control, DMA transfer and the servicing of card interrupts.

The class TMMCard is the abstraction of a MultiMediaCard within the stack. The Symbian OS software controller supports up to four cards per stack, and allocates a card object for each. Clients can gain access to these cards via the corresponding stack object, which also provides information on number of cards supported on this particular phone platform (that is, the number of card slots allocated to the stack). The stack owns a card array object, TMMCardArray, which manages the cards. The TMMCard class is one of the main elements of the MultiMediaCard API provided to drivers. Here are some of the public parts of this class:

```
class TMMCard
    {
public:
    TMMCard();
    inline TBool IsPresent() const;
    TBool IsReady() const;
    inline TMMCMediaTypeEnum MediaType() const;
    inline TUint DeviceSize() const;
    virtual TUint MaxTranSpeedInKilohertz() const;
    };
```

MultiMediaCards can be inserted or removed at any time and the method `IsPresent()` indicates whether there is currently a card present in the slot concerned. `IsReady()` indicates whether the card is powered, initialized and ready to accept a data transfer command. The method `MediaType()` returns one of the following values to indicate the type of card present:

`EMultiMediaROM`	Read-only MultiMediaCard
`EMultiMediaFlash`	Writeable MultiMediaCard
`EMultiMediaIO`	I/O MultiMediaCard

(C) Copyright Nokia 2014. All rights reserved.

`DeviceSize()` returns the total capacity of the card in bytes. However, this doesn't take account of how this memory has been partitioned. (Partition information for the card is normally stored in a partition table in the card's first data block - which has to be read using a block read command by the media driver.)

`MaxTranSpeedInKilohertz()` returns the maximum supported clock rate for the card.

The `DMMCSession` class provides the other main part of the client interface to the MultiMediaCard controller. A `DMMCSession` represents a unit of work for the stack, and is used to issue commands - either to the entire stack using a broadcast command, or to an individual card in the stack. Each client creates its own instance of this class, and associates it with the stack object, `iStackP`, concerned. The client must also associate it with a card object, `iCardP`, if the session is to be used to send addressed commands. To issue a request, the client configures the session object with the relevant information for the request and submits it to the stack. The DMMCSession class contains methods for initiating macro commands, as well as lower level methods allowing a client to control the stack in a more explicit manner. Here are some of the public parts of this class:

```
class DMMCSession : public DBase { public: virtual ~DMMCSession(); DMMCSession(const
TMMCCallBack& aCallBack); void SetupCIMReadBlock(TMMCArgument aDevAddr, TUint32 aLength,
TUint8* aMemoryP); void SetupCIMWriteBlock(TMMCArgument aDevAddr, TUint32 aLength,
TUint8* aMemoryP); void SetupCIMReadMBlock(TMMCArgument aDevAddr, TUint32 aLength,
TUint8* aMemoryP, TUint32 aBlkLen); void SetupCIMWriteMBlock(TMMCArgument aDevAddr,
TUint32 aLength, TUint8* aMemoryP, TUint32 aBlkLen); TInt Engage(); inline TUint8*
ResponseP(); };
```

When creating a DMMCSession object, the client supplies a callback function as part of the class constructor. Once a client has engaged a session on the stack, the controller will inform it of the completion of the request by calling this callback function.

Next, you can see four methods used to configure the session for data transfer macro commands. The first pair of methods involves single block transfer. Looking at the first of these in detail:

```
void SetupCIMReadBlock(TMMCArgument aDevAddr, TUint32 aLength, TUint8* aMemoryP);
```

This configures the session for a single block read from the card. When submitted, the stack starts by issuing a command to define the block length as aLength bytes for the subsequent block read command. Then it issues a read single block command - reading from offset `aDevAddr` on the card into system memory beginning at address `aMemoryP`. No stop command is required in this case.

The second pair of methods involves multi-block transfer. This time, I will look at the write version in more detail:

```
void SetupCIMWriteMBlock(TMMCArgument aDevAddr, TUint32 aLength, TUint8* aMemoryP,
TUint32 aBlkLen);
```

When submitted, the stack issues a command to define the block length as {{lcode|aBlkLen} bytes for the subsequent block write command. It then issues a write multiple block command to continually transfer blocks from the host to the card, starting at address `aMemoryPin` system memory, and offset `aDevAddr` on the card. Once `aLength` bytes have been transferred, the stack issues a stop command to terminate the transfer. `Engage()` is used to enque the session for execution on the `DMMCStack` object once it has been configured.

`ResponseP()` returns a pointer to a buffer containing the last command response received by the session.

The controller is designed to accept more than one client request on a stack at any given time. This could happen on multi-card stacks, or on single card stacks containing multi-function cards where multiple drivers have session engaged simultaneously. The controller attempts to manage the sessions as efficiently as it can, by internally scheduling them onto the bus. When the current session becomes blocked waiting on an event, the controller will attempt to reschedule another session in its place.

Bus configuration and error recovery

Referring still to Figure 13.12, the class `TMmcStackConfig` is used to hold bus configuration settings for a stack. These settings are such things as the bus clock rate, whether to try re-issuing commands on error, how long to wait for a response from the card and so on. The stack owns an instance of this class (not shown on the diagram) containing the default settings that are normally applied. Each session also owns an instance of this class, the member `iConfig`, which normally contains a copy of the defaults.

(C) Copyright Nokia 2014. All rights reserved.

However, if it chooses, the client may over-ride the configuration settings for any bus operation it submits by altering the contents of `iConfig`. These changes only remain in effect for the period that the session remains current.

The controller is normally configured to automatically retry failed operations when any of the following errors are detected:

- Timeout waiting for a command response from a card
- A CRC error is detected in a response
- A timeout waiting for data transfer to commence during a data read or write command
- A CRC error detected in a data block during data transfer.

For certain other errors, such as if the card state is found to be inconsistent with the command being issued, the controller will attempt to recover by re-initializing the entire stack before retrying the failed operation.

Card power handling

When the controller detects a door-open event, it tries to remove power from the card as soon as possible. It does not remove power immediately if a bus operation is in progress, because it wouldn't be a good idea to remove power from a card in the middle of writing a block, as this could corrupt the block. In this case, power-down is deferred until the end of the MultiMediaCard session. Attempts to engage a new session while the door is open will fail immediately though.

So, to avoid the situation in which a card is physically unplugged while a command is still completing, driver requests have to be kept short enough to ensure that they can always be completed in the time between the door open event and the time the card is physically removed. This means that long multi-block write commands have to be avoided, despite the improved rate of data transfer they provide over shorter block transfers. It is very important that the phone provides a door mechanism and circuitry that gives early warning of potential card removal.

The controller is normally configured to implement a bus inactivity power-down scheme to save power. If the inactivity period elapses, then the controller automatically removes power from the cards. The length of this inactivity timeout period is set by the particular mobile phone.

As I said in Section 13.3.3.5, the local media sub-system does not initialize removable media devices as soon as they are inserted, but instead waits until the first request on the drive. Nevertheless, this request generally arrives almost immediately after card insertion, because applications receive notification of the disk insertion event from the file server and then interrogate the new card.

For MultiMediaCards, initialization involves applying bus power and then performing the card identification process. This entails issuing a series of broadcast and addressed commands over the bus, and is handled asynchronously by the controller. (All requests on the stack that involve bus activity are inherently long running operations that have to be handled asynchronously.) Initialization proceeds as follows.

First, the cards in the stack are reset, and then their operating voltage range is ascertained to ensure this is compatible with that of the host phone. The host reads the 128-bit unique ID that identifies each card. It then allocates each card a shorter Relative Card Address (RCA), which is used thereafter to address that card. Finally, the host reads back data from the card concerning its operating characteristics, to check that these are compatible with the host. Now the card is available for data transfer. This entire process is carried out in the first phase of drive mounting - before any media drivers are opened.

I/O drivers don't use the local media sub-system, and so they need to ensure that the bus is powered and the stack is initialized when they open. However, once an I/O driver has opened successfully, it doesn't need to bother about the card subsequently becoming powered down again. If the controller receives a data transfer request for a card that has been powered down due to a normal power down event it automatically applies power and initializes the stack first.

USB device support

USB overview

Universal Serial Bus (USB) is a bus standard for connecting peripheral and memory devices to a host computer. It supports hot insertion and removal of devices from the bus - devices may be attached or removed at any time. The bus consists of four signals: two carrying differential data and two carrying power to the USB device. The USB specification revision 2.0 defines three data rates:

Data rate	Data transfer rate
USB High Speed	Up to 480 Mbits/sec
USB Full Speed	12 Mbits/sec
Limited capability low speed	1.5 Mbits/sec

(C) Copyright Nokia 2014. All rights reserved.

The USB system consists of a single host controller connected to a number of USB devices. The host includes an embedded root hub that provides one or more attachment points. The host is the bus master and initiates all data transfers over the bus. Each USB device passively responds to requests addressed to it by the host.

The host is often a desktop computer, but a supplement to the USB specification introduces a dual-role USB device. As well as being a normal USB device, this kind of device is also able to take on the role of a limited USB host, without the burden of having to support full USB host functionality. This is the On-The-Go (OTG) supplement aimed at portable devices.

Many USB devices implement just a single function - USB keyboards and data storage devices are examples of these - but multi-function devices are also possible. These are called composite devices, an example being a USB headset that combines a USB headphone and microphone. Likewise, although the functionality of most devices remains static, some devices can alter the USB function or functions they implement. A mobile phone is an example of this - it may use various different USB functions to exchange data with a host computer. Related USB devices that provide similar functionality are grouped into USB device classes, and standard protocols are defined to communicate with them. This means that a generic device class driver on the host machine can control any compliant device. Many classes are further subdivided into subclasses. The USB Implementers' Forum assigns unique codes to each class and subclass, and USB devices report these codes for each function they support. Examples of USB classes include the USB Mass Storage class for devices such as MultiMediaCard readers, and the Communications Device class for modem devices.

A USB device is made up of a collection of independent endpoints. An endpoint is the terminus of a communication flow between host and device that supports data flow in one direction. Each endpoint has its own particular transfer characteristics that dictate how it can be accessed. Four transfer types are defined:

Transfer type	Description
Bulk	Used for transferring large volumes of data that has no periodic or transfer rate requirements (for example, a printer device).
Control	Used to transfer specific requests to a USB device to configure it or to control aspects of its operation.
Isochronous	Used where a constant delivery rate is required (for example, an audio device). Given guaranteed access to USB bandwidth.
Interrupt	Used to poll devices that send or receive data infrequently, to determine if they are ready for the next data transfer.

Every USB device contains at least one input and one output control endpoint - both with endpoint number zero (ep0). The host uses this pair to initialize and control the device. Full speed devices can have a maximum of 15 input and 15 output endpoints, in addition to ep0. Each USB function on a device has an associated set of endpoints, and this set is known as an interface.

Before a device can be used, it must first be configured. This is the responsibility of the host, and is normally done when the device is first connected. In a process known as bus enumeration, the host requests information on the capabilities and requirements of the device. The data returned specifies the power requirements of the device. It also describes each interface, in terms of its class type, the endpoints it contains and the characteristics of each endpoint. This is the device configuration. Certain devices offer alternative configurations. This information is contained in a set of device descriptors - once more, these are not to be confused with Symbian OS descriptors! The host checks whether it can support the power and bandwidth requirements, and that it has a compatible class driver. It may also have to select the configuration of choice. The host is said to have configured the device by selecting and accepting a configuration.

An interface within a configuration may also have alternative settings that redefine the number of associated endpoints or the characteristics of these endpoints. In this case the host is also responsible for selecting the appropriate alternate setting.

A Symbian OS phone is unlikely to be configured as a standard USB host because such devices have to be able to supply a high current to power devices attached to the bus. Until now, most Symbian OS phones have been configured as USB devices and connected to a USB host computer. Classes supported by Symbian OS include the Abstract Control Model (ACM) modem interface - this is a subclass of the Communications Device Class (CDC) and provides serial communications over USB. It is used for backup, restore and data synchronization with a desktop computer. Another class Symbian OS supports is the Mass Storage class, which allows direct access to certain drives on the phone from the host computer.

We expect that Symbian OS OTG devices will be produced in the near future. These will be able to be used as host computers, interfacing with USB devices such as printers, video cameras and mass storage devices.

USB software architecture

The standard Symbian OS USB software architecture provides support for USB device (or client) functionality. Our implementation is designed to work with a hardware USB Device Controller (UDC). UDCs provide a set of endpoints of varying transfer type (bulk, control and so on), of varying direction (IN, OUT or bidirectional) and varying packet size. UDCs normally allow

(C) Copyright Nokia 2014. All rights reserved.

these endpoints to be grouped together into various different USB interfaces and configurations. This means that a single Symbian OS phone can be configured for different USB device functions so long as it contains the appropriate (device) class driver. This also means that as long as the UDC provides enough endpoints, the phone can be configured as a multi-function device.

Figure 13.13 shows an overview of the Symbian OS USB software architecture.

As an example, Figure 13.13 shows the setup for a phone configured as a combined Abstract Control Model (ACM) and mass-storage USB device. (However, when the host configures a device containing multiple functions, it enables each of these and requires a class driver for each. In practice, it can be difficult to obtain the corresponding composite host-side driver setup for this type of combined functionality.)

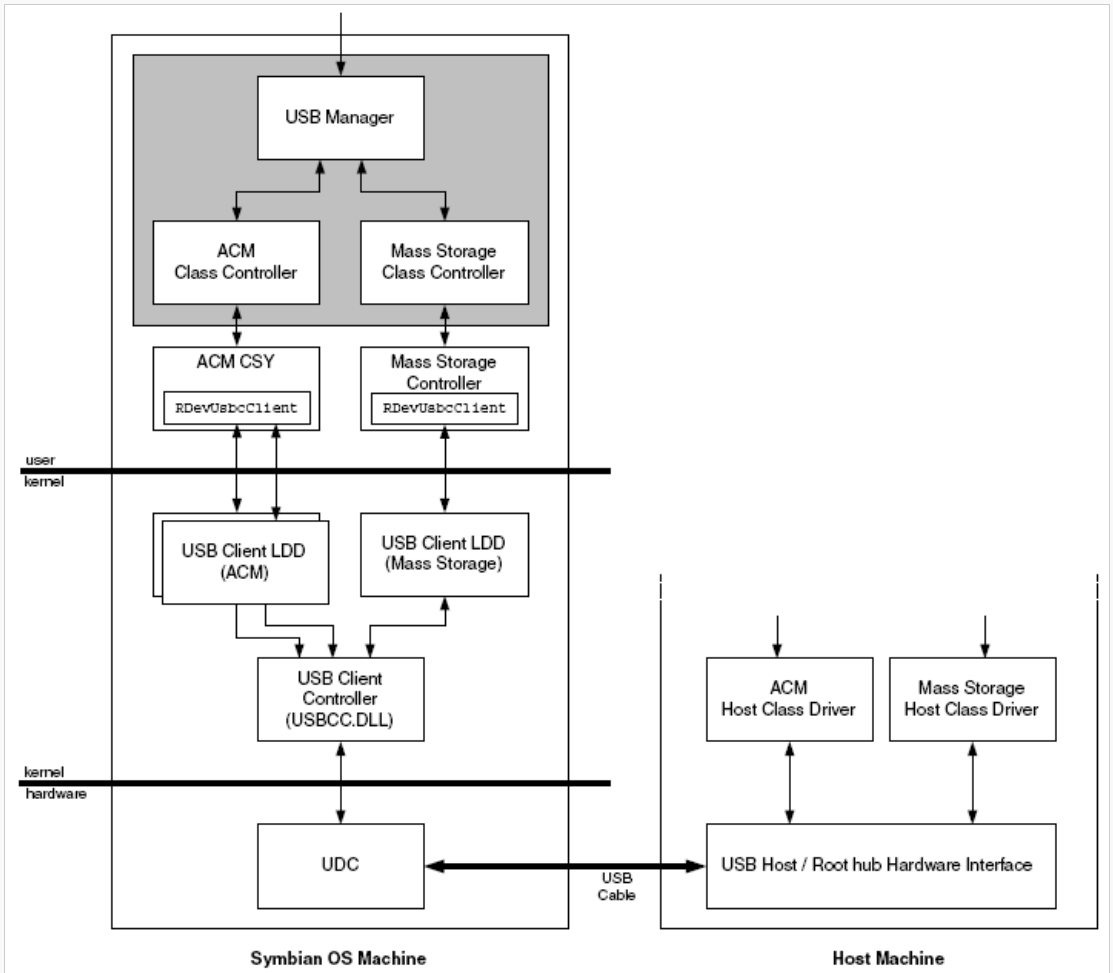

Figure 13.13 *An overview of the Symbian OS USB architecture*

The USB manager ensures the orderly startup and shutdown of all the USB classes on the Symbian phone, as well as allowing its clients to determine the status of these classes and to be informed of changes in the overall USB state. To this end, the USB Manager implements a class controller for each supported class on the device. A class controller provides the interface between the USB manager and the class implementation - but does not implement any class functionality itself.

USB class implementations normally exist in a separate thread from the USB manager. To gain access to the USB hardware (UDC), the USB class implementation must open a channel on the USB client device driver. The class `RDevUsbcClient` provides the user-side interface to this driver. Each channel supports only a single main USB interface (although it may support multiple alternate interfaces). This means that class implementations that use two or more main interfaces must open multiple channels.

Once a channel has been opened, the class implementation is able to read the USB capabilities of the phone to determine the total number of endpoints, their type, direction, maximum packet size, availability and so on. If the phone provides the required USB resources, and they are not already in use, the class implementation then sets up each USB interface by setting a class type and claiming its endpoints. All the channels automatically have access to ep0, and of course each of them can make a request on it. The other endpoints may only be used by a single channel, and can't be shared. Each channel may claim up to five endpoints as well as ep0.

The ACM class is implemented as a comms server (C32) plug-in, or CSY. Clients that wish to use this CSY do so via the C32 API. The ACM comprises two interfaces. The first is a communications interface consisting of an interrupt endpoint and a control endpoint (ep0) for transferring management information between host and device. The second is a data interface consisting of a

(C) Copyright Nokia 2014. All rights reserved.

pair of bulk endpoints (one IN, one OUT) - this acts like a legacy serial interface. This means that this class opens two channels on the USB client driver - one for each interface.

The mass storage controller provides the mass storage class implementation, which is built as a file system component (MSFS.FSY). It is implemented using the Bulk-Only Transport protocol (a protocol specific to USB) which provides a transport for the communication of standard SCSI Primary Commands (SPC) between host and device. This requires a single USB interface consisting of a pair of bulk endpoints (one IN, and one OUT) over which the majority of the communication takes places, and a control endpoint (ep0) to issue class-specific requests and clear stall conditions.

Each USB client LDD manages client requests over each endpoint and passes these on to the USB client controller. It also creates and manages the data buffers involved in transferring data to and from the UDC.

The USB client controller is a kernel extension that manages requests from each of the channels and controls the hardware UDC. It is divided into a platform-independent layer (PIL) and a platform-specific layer (PSL).

This architecture allows the current USB function (or functions) of the phone to be changed without the need to physically remove the USB cable or restart the phone. The USB manager allows classes to be started or stopped, and doing so will result in new USB interfaces being setup or existing ones released. The USB driver API also supports the simulated removal and insertion of the cable (so long as the hardware interface does too). However, the host assumes that once a device has been enumerated, the functions described will be available until disconnection. The host is also unable to discover new classes that are started after enumeration. This means that the host sees such changes in USB function as the removal of one device and the attachment of a different one, which causes it to re-enumerate. This terminates any active USB communication.

The software architecture I have described supports only Full Speed USB 2.0 device functionality - not USB Host or OTG. Neither does it support USB High Speed.

The kernel-side components and the UDC handle the USB device protocol layer, whereas the various class implementers handle the USB device side of the class layers. The next section concentrates on the device protocol layer implementation.

USB client controller and LDD

Figure 13.14 shows part of the class diagram for the USB client controller and LDD.

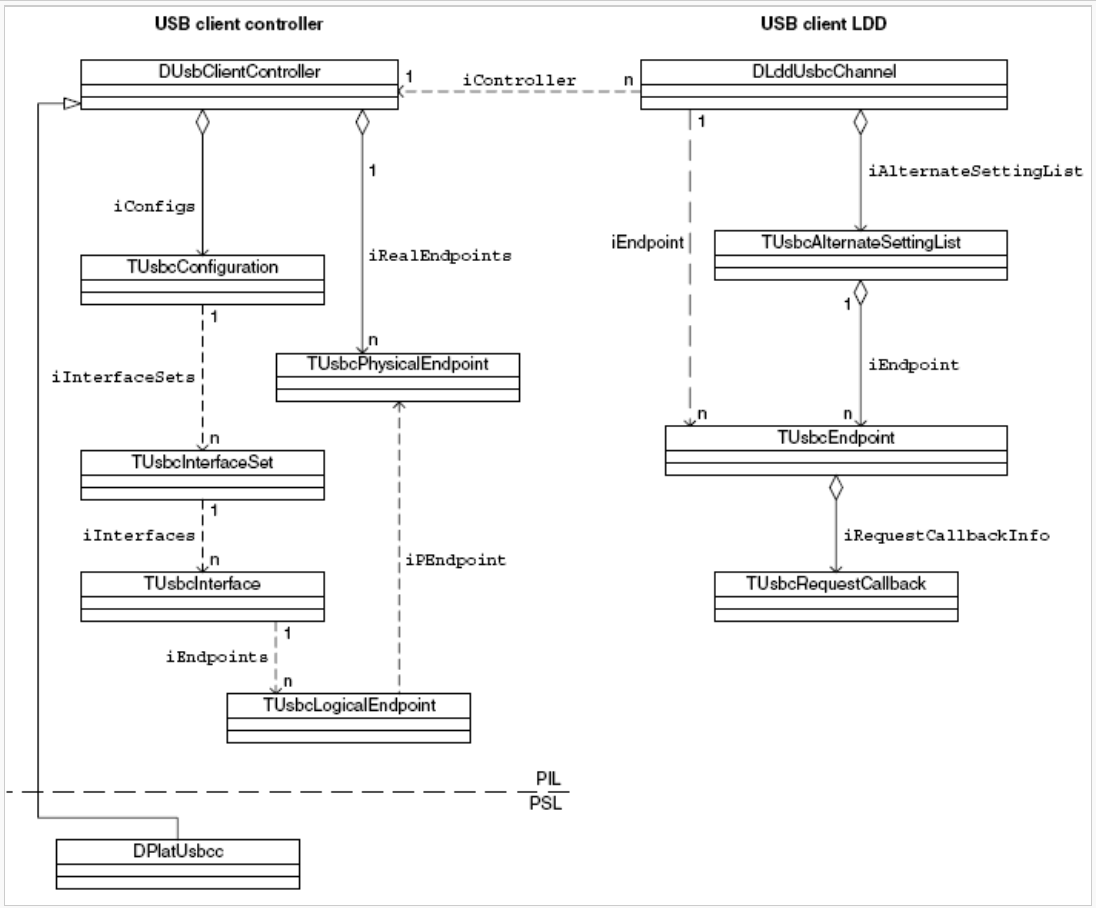

Figure 13.14 *Class diagram for the USB client controller and LDD*

The USB controller

DUsbClientController is a singleton that embodies the USB device controller; it is an abstract class that defines the interface

(C) Copyright Nokia 2014. All rights reserved.

between the PIL and the PSL. Each platform provides a derived controller object, shown as `DPlatUsbcc` on the diagram, which handles USB functionality specific to the platform UDC - such as the management of data transfer over an endpoint. Use of DMA is recommended for USB transfers, and the PSL handles all aspects of DMA data transfer.

The main role of the controller is the handling of packets on ep0, and the relaying of LDD read and write transfer requests from the other endpoints. The PIL part of the controller processes and responds to all standard USB device requests, (as described in chapter 9 of the Universal Serial Bus Specification Revision 2.0 (www.usb.org)) - if they are not already handled by the UDC hardware itself.

The controller also creates and manages the USB descriptor pool (or database). Descriptors contain information about the properties of specific parts of the USB device in a well-defined format, and they are propagated to the host - normally during enumeration. The USB client API allows class implementers to specify and alter many of the elements of each different type of descriptor as well as to add class-specific descriptors. However to preserve the USB device's integrity, the controller creates other parts of the descriptors itself and clients of the controller cannot alter these.

The class `TUsbcPhysicalEndpoint` is the abstraction for a physical endpoint present on the device. At system boot time, the controller creates an instance of this class for each endpoint that the UDC supports - and these are never destroyed. The controller class owns these objects and holds them in the array `iRealEndpoints`. An endpoint capabilities class is associated with the `TUsbcPhysicalEndpoint` class (this is not shown on the diagram). This class stores information on the set of endpoint types, directions and maximum packet sizes supported by the endpoint. Physical endpoints are used at the interface between PIL and PSL.

The `TUsbcConfiguration` object encapsulates a USB configuration. The Symbian OS USB client API only supports a single configuration per device and so the controller owns just one instance of this class, `iConfigs`, which it creates at system boot time.

As I mentioned earlier, Symbian OS supports multiple interfaces (or USB functions) within this single configuration. It also supports alternate USB interfaces - so an interface within the configuration may have alternative settings, each potentially having differing numbers of endpoints or differing endpoint characteristics.

To accommodate multiple interfaces, the configuration object manages an array of `TUsbcInterfaceSet` objects, `iInterfaceSets`. Each set object corresponds to an individual main interface within the configuration. It is called an interface set because, for interfaces with alternative settings, this object represents the set of alternative interfaces supported. A configuration consisting of a single main interface has just a single interface set object. Each set object is created when the associated LDD client requests a first interface on the channel (which might be the first of a number of alternative settings) and destroyed when the last setting is released. Remember that there can only be one main interface (and therefore one interface set) per channel.

The interface set manages an array of `TUsbcInterface` objects:iInterfaces. Each interface object encapsulates one of the alternate interface settings. For interfaces without alternative settings, only a single instance of this class is created. For interfaces that do have alternative settings, the associated set object keeps track of the current alternative setting. A `TUsbcInterface` object is created each time an LDD client is successful in requesting an interface and destroyed when that setting is released again.

Associated with each `TUsbcInterface` object is a group of endpoint objects that make up (or belong to) that interface setting. However, these are logical endpoint objects - `TUsbcLogicalEndpoint`. An interface may claim up to a maximum of five endpoints in addition to ep0. Each is locally numbered between one and five and the LDD client uses this number to identify an endpoint when it issues requests. This number need not correspond to the actual endpoint number of the UDC. (LDD clients can discover the physical endpoint address of a logical endpoint by requesting the endpoint descriptor for the endpoint). When an interface is being created, the controller is supplied with the details of each of the endpoints required by the client. It scans through the list of physical endpoints, searching for ones that are available and that have matching capabilities. Obviously, interface setting can only succeed if the search is successful for all endpoints specified within the interface. If successful, a `TUsbcLogicalEndpoint` instance is created for each - and this has the same lifetime as the associated interface object. `TUsbcLogicalEndpoint` holds information on endpoint transfer type, direction and maximum packet size together with a pointer to its corresponding physical endpoint object, `iPEndpoint`.

The USB client LDD

The class `DLddUsbcChannel` is the USB client LDD channel object - an instance being created for each main interface that is set on the UDC. It is derived from logical channel base class `DLogicalChannel`- which means that channel requests are executed in the context of a kernel thread. A DFC queue is associated with the controller object, and this determines which kernel thread is used to process these requests. It is set on a per-platform basis, with the default being DFC thread 0. The channel owns an instance of the `TUsbcAlternateSettingList` class for each alternative setting that exists for the interface, `iAlternateSettingList`. In turn, each alternative setting object owns an instance of the `TUsbcEndpoint` class for each endpoint that it contains, apart from ep0.

(C) Copyright Nokia 2014. All rights reserved.

Instead, the channel owns the `TUsbcEndpoint` instance for ep0 and also maintains a pointer to each of the endpoint objects for the current alternate interface via `DLddUsbcChannel::iEndpoint`. An important function of the `TUsbcEndpoint` class is to manage the buffers used for data transfer. However, the channel object owns these buffers since they are shared with other endpoints in the interface.

Up to three hardware memory chunks, each containing physically contiguous RAM pages, are allocated to every channel object, and these chunks are each divided into separate buffers for use during data transfers. All IN endpoints (that is, ones which transfer data back to the host) share one chunk, OUT endpoints share the second, and the third is used for ep0. These chunks are created when an interface is first set on the channel. The size of chunk for ep0 is fixed, containing four 1024-byte buffers. However, fairly obviously, the size of the IN and OUT chunks depends on the number of IN and OUT endpoints that are included in the interface. The number of buffers created for each of these endpoints is fixed, but the size of the buffers is configurable by the LDD client, using bandwidth priority arguments specified when setting an interface. A single buffer is created for each IN endpoint and four buffers are created for each OUT endpoint. The default buffer size for Bulk IN is 4 KB, and for Bulk OUT it is 4 KB too. We have selectable OUT endpoint buffer sizes for performance reasons - large buffer sizes are recommended for high bandwidth data transfers. Since different alternate interfaces may specify different groups of endpoints and different buffer sizes, the chunks often have to be reallocated each time the LDD client sets a different alternative interface. (The chunk size finally used is the maximum of each alternate setting's requirements.)

The `TUsbcRequestCallback` class encapsulates an LDD transfer request. It holds data specifying the request, together with a DFC that the controller uses to call back the LDD when transfer completes. The `TUsbcEndpoint` class owns a request object, `iRequestCallbackInfo`, which it uses to issue requests to the controller. A channel can have asynchronous requests outstanding on all of its endpoints at once, and this includes ep0. Since ep0 is shared with other channels, the client controller has to manage multiple requests on the same endpoint.

The mass storage file system

This is quite different from any other file system. It contains null implementations of the file system API described in Section 9.4.1, and, when it is mounted on a drive, that drive is inaccessible from the Symbian OS device. Instead, the desktop host computer is allowed exclusive block level access to the drive. The mass storage file system implements the mass storage controller function that I introduced in Section 13.6.2, which involves the handling of SCSI commands received from the host via a USB client device driver channel. The file server is not involved in the processing of the commands. Instead they are processed entirely by the mass storage controller. Being a file system component, it has access to the media device concerned via the local drive interface class, `TBusLocalDrive`. You should note that if the drive has a file server extension mounted on it (for example a NAND drive with the flash translation layer implemented in a file server extension), then all media accesses are routed through the extension. This allows Symbian OS to support a mass storage connection to a NAND flash drive, as well as to a normal FAT drive. Only FAT-formatted drives may be connected as mass storage drives. Drives C: or Z: cannot be connected, because these must always be accessible to the rest of the OS.

The mass storage file system is not normally loaded automatically during file server startup. Instead, a USB mass storage application (a component provided by the phone manufacturer) loads it later, and also mounts the file system on a particular drive. However, before it does this, the application has to dismount the FAT file system from that drive. This can only happen if there are no file or directory resources open on the drive. This may mean that the application has to request that the user shuts down certain applications that have these resources open.

Once the mass storage connection is terminated, the same application is responsible for dismounting the mass storage file system and remounting the FAT file system again.

Figure 13.15 shows the two configurations of aNAND drive configured for mass storage connection. The first configuration shows it mounted and accessible from a Symbian OS device. The second shows it disconnected from the Symbian OS device, with a host computer accessing the drive.

Granting direct access to a drive on the Symbian OS phone from a host machine poses a security threat. To counter this, all drives available for mass storage connection are subject to the same restrictions as removable drives. For instance, installed binaries on the mass storage drive could be altered while the desktop computer is remotely accessing the drive.

So we need tamper evidence to detect if the contents of these binary files have been altered since they were known to be safe, at install time. Section 8.5.2.3 covers this in a little more detail.

(C) Copyright Nokia 2014. All rights reserved.

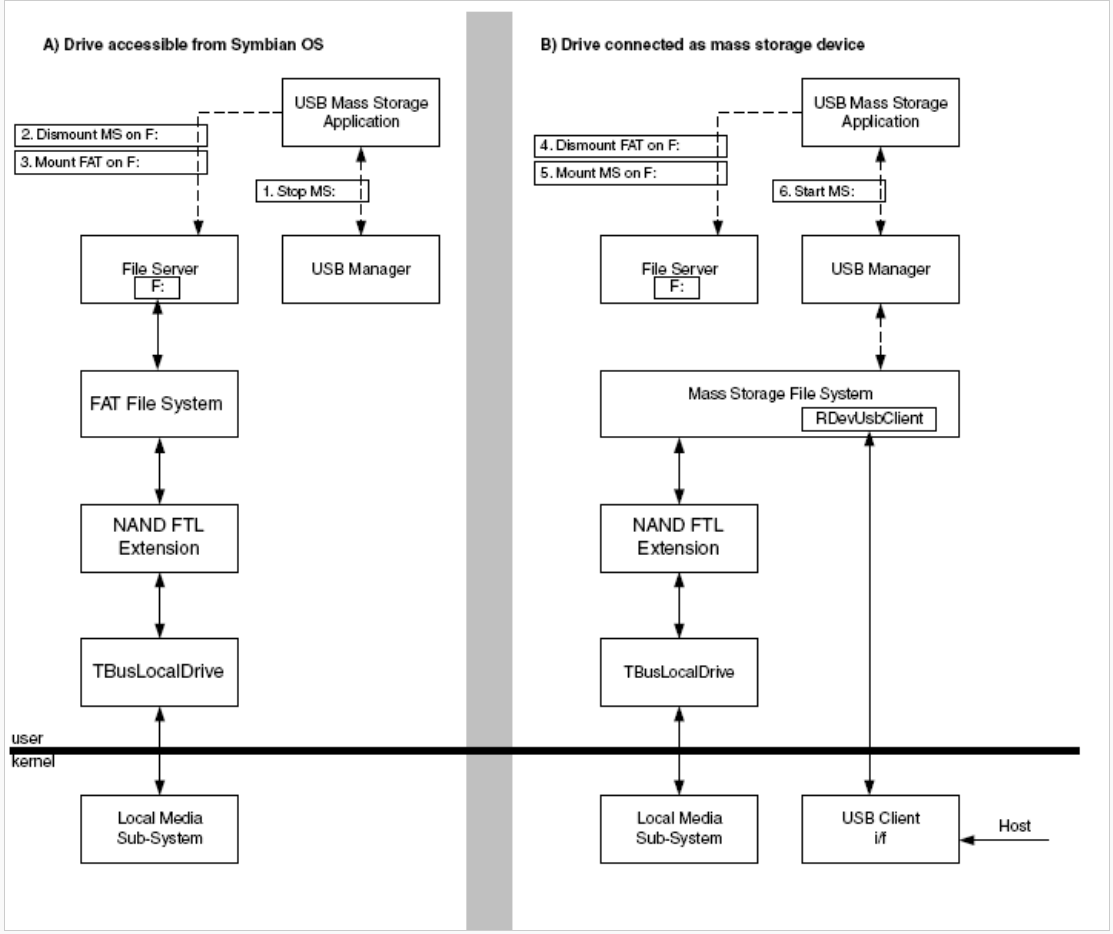

Figure 13.15 *Two configurations of a NAND drive used for mass storage connection*

Summary

In this chapter, I began by describing two more of the services available to device drivers and peripheral bus controllers: DMA and shared chunks. I went on to describe media drivers and the local media sub-system. Then I examined peripheral bus controllers - looking specifically at the MultiMediaCard controller as an example. Finally, I introduced the kernel-side components of the USB software architecture and the USB mass storage file system. In the next chapter, I will describe debugging in the Symbian OS environment.

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(C) Copyright Nokia 2014. All rights reserved.