Symbian OS Internals/05. Kernel Services

by Jane Sales

On two occasions I have been asked (by members of Parliament!):
Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?
I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
Charles Babbage

EKA2 provides a variety of services to user-mode threads. In this chapter I will explain the mechanism it uses to do so, which we call an *executive call*, and then I will describe a few example services to give you a feel for them.

Of course, the kernel does not just provide services for user-mode threads - each part of the kernel provides services to the other parts of the kernel too. I will consider interfaces between modules such as the nanokernel and the memory model, and interfaces between the different abstraction levels of the kernel, such as the independent layer and the CPU layer.

But first of all I will look inside the basic object and handle mechanism used by Symbian OS. This is at the heart of the communication between the user side and the kernel.

Objects and handles

Handles - the `RHandleBase` class

User-side code always references a kernel-side object through an object known as a handle. Handles are objects derived from the base class RhandleBase:

```
class RHandleBase
  {
public:
  enum
    {
    EReadAccess=0x1,
    EWriteAccess=0x2,
    EDirectReadAccess=0x4,
    EDirectWriteAccess=0x8,
    };

public:
  inline RHandleBase();
  inline TInt Handle() const;
  inline void SetHandle(TInt aHandle);
  inline TInt SetReturnedHandle(TInt aHandleOrError);
  static void DoExtendedClose();
  IMPORT_C void Close();
  IMPORT_C TName Name() const;
  IMPORT_C TFullName FullName() const;
  IMPORT_C void SetHandleNC(TInt aHandle);
  IMPORT_C TInt Duplicate(const RThread& aSrc, TOwnerType aType=EOwnerProcess);
  IMPORT_C void HandleInfo(THandleInfo* anInfo);
  IMPORT_C TUint Attributes() const;

protected:
  inline RHandleBase(TInt aHandle);
  IMPORT_C TInt Open(const TFindHandleBase& aHandle,TOwnerType aType);
  static TInt SetReturnedHandle(TInt aHandleOrError,RHandleBase& aHandle);
  TInt OpenByName(const TDesC &aName,TOwnerType aOwnerType,TInt aObjectType);

private:
  static void DoExtendedCloseL();

protected:
  TInt iHandle;
  };
```

Here you can see some of the fundamental methods that we can perform on handles: we can open and close them, retrieve their short name and their full name, and we can duplicate them. You can also see that `RHandleBase`'s only member data is a single 32-bit integer, `iHandle`. To show you how the kernel forms this integer, I will first need to explain a container class, `DObjectIx`, which is known as the object index. This class is a container for kernel-side objects derived from `DObject`, which I will discuss first.

Reference-counted kernel objects

A large part of the kernel interface presented to user-side code is concerned with creation and manipulation of kernel objects

represented by user-side `RHandleBase`-derived classes. These kernel objects have some basic properties in common.

Reference counted

Kernel objects are reference counted: multiple references can exist to each object and the kernel only destroys the object when all references have been removed.

Accessed using handles

User-side code accesses kernel objects indirectly using handles, rather than directly using pointers. The kernel translates a handle into a pointer by looking it up in a thread or process handle array. The use of handles allows the kernel to check the validity of kernel object references made by user code.

Named

Kernel objects may have names that you can use to find the object. Moreover, the name can be scoped relative to another kernel object (the owner). I will expand more on this later.

The `DObject` class

As I mentioned earlier, kernel objects are represented using classes derived from the `DObject` class. This base class provides the necessary reference counts, object names and name scoping relative to the owner object. `DObject` is in turn derived from `DBase`, this class provides kernel-side behavior equivalent to that provided by the user-side class `CBase`; that is, it zero-fills memory before object construction and provides a virtual destructor. It also offers the ability to trigger asynchronous deletion of the object, which is important in time-critical code. Here is a slightly cut-down version of the `DObject` class:

```
class DObject : public DBase
  {
public:
  enum TCloseReturn
    {
    EObjectDeleted=1,
    EObjectUnmapped=2,
    };
  enum TObjectProtection
    {
    ELocal=0,
    EProtected,
    EGlobal,
    };

public:
  inline TInt Inc() {return NKern::SafeInc(iAccessCount);}
  inline TInt Dec() {return NKern::SafeDec(iAccessCount);}
  IMPORT_C DObject();
  IMPORT_C ~DObject();
  inline TInt Open() { return(Inc()?KErrNone:KErrGeneral); }
  IMPORT_C void CheckedOpen();
  IMPORT_C virtual TInt Close(TAny* aPtr);
  IMPORT_C virtual TInt RequestUserHandle(DThread* aThread,TOwnerType aType);
  IMPORT_C virtual TInt AddToProcess(DProcess* aProcess);
  IMPORT_C TInt AsyncClose();
  IMPORT_C virtual void DoAppendName(TDes& aName);
  IMPORT_C void DoAppendFullName(TDes& aFullName);
  IMPORT_C void Name(TDes& aName);
  IMPORT_C void AppendName(TDes& aName);
  IMPORT_C void FullName(TDes& aFullName);
  IMPORT_C void AppendFullName(TDes& aFullName);
  IMPORT_C TInt SetName(const TDesC* aName);
  IMPORT_C TInt SetOwner(DObject* aOwner);
  IMPORT_C void TraceAppendName(TDes8& aName, TBool aLock);
  IMPORT_C void TraceAppendFullName(TDes8& aFullName, TBool aLock);
  inline DObject* Owner();
  inline TInt AccessCount();
  inline TInt UniqueID();
  inline HBuf* NameBuf();
  inline void SetProtection(TObjectProtection aProtection);
  inline TUint Protection();

public:
  TInt iAccessCount;
  DObject* iOwner;
  TUint8 iContainerID;
```

```
  TUint8 iProtection;
  TUint8 iSpare[2];
  HBuf* iName;

public:
  static NFastMutex Lock;
};
```

Key member data of Dobject

`iAccessCount`	This counts how many references exist to the object - it is always non-negative.
`iOwner`	This is a reference-counted pointer to the `DObject` (thread or process) that is the owner of this object.
`iContainerID`	This is the ID of the `DObjectCon` that contains this object. I will discuss this later in this chapter.
`iName`	This is a pointer to a kernel-heap-allocated descriptor that holds this object's name. It is NULL if the object is unnamed.
`Iprotection`	This is a `TObjectProtection` value, which notes if the object is private to the owning thread or process.

`DObjects` explained

The `DObject` class is new to EKA2. In EKA1 we derived our kernel classes from the user library's object class, `CObject`. In EKA2, we chose to create a new, kernel-only, `DObject` class to break the dependency between the kernel and the user library. In the same way, we created `DObjectIx` for the kernel to use instead of CObjectIx.

When a user thread requests the creation of an object represented by a handle, the kernel creates a `DObject` with an access count of 1, representing the pointer returned to the creating thread. If another thread then wishes to open this object, the kernel calls <tt style="font-family:monospace;">DObject::Open()</tt> on its behalf, incrementing the `DObject`'s access count. We wanted it to be possible to call this method from anywhere, even in an ISR or DFC, so we prevented it from being over-ridden in a derived class. The result is that <tt style="font-family:monospace;">DObject::Open()</tt> always atomically executes the following operation:

```
if (iAccessCount==0)
  return KErrGeneral;
else
  {
  ++iAccessCount;
  return KErrNone;
  }
```

The access count is incremented, unless it was zero - this is an error, because, as we've seen, every `DObject` is created with an access count of 1.

The <tt style="font-family:monospace;">DObject::Dec()</tt> method does the opposite - it atomically executes the following operation:

```
if (iAccessCount==0)
  return 0;
else
  return iAccessCount--;
```

The `Open()` and `Dec()` methods are not protected by fast mutexes; they simply use atomic instructions or disable interrupts for a short time.

When a user thread closes a handle, the kernel invokes the <tt style="font-family:monospace;">DObject::Close(TAny*)</tt> method to remove a reference from the object. It calls `Dec()`, then proceeds to delete the object if the returned value is 1, indicating that the last reference has been closed:

```
EXPORT_C TInt DObject::Close(TAny* aPtr)
  {
  if (Dec()==1)
    {
    NKern::LockSystem(); // in case it is still in use
    NKern::UnlockSystem();
    DBase::Delete(this);
    return EObjectDeleted;
    }
  return 0;
  }
```

Since `Close()` may cause the freeing of memory on the kernel heap, the rules about when kernel heap operations may be

performed apply; this means that we can't call it from an ISR or a DFC, for example. This contrasts with `Open()`, which as we've seen can be called from anywhere. We therefore allow the `Close()` method to be over-ridden by making it virtual.

The kernel deletes a `DObject` only when its access count becomes zero - in fact, this always happens via the `Close()` method. It is possible that a `DObject` with a zero access count is in the process of being destroyed. This is why `Open()` must fail if the object's access count is zero.

The parameter aPtr passed to `Close()` is either NULL or a pointer to the process that is closing a handle on the object. The kernel uses the pointer when the object being closed is a chunk, to remove the chunk from the process address space.

DObject also provides an `AsyncClose()` method. This is the same as `Close()` except that the parameter is always NULL and the kernel does the delete (if one is needed) asynchronously in the supervisor thread. Of course, `AsyncClose()` will only work if the derived class does not over-ride Close().

There are two names associated with a `DObject` - the name (also known as the short name) and the full name. The short name is either:

1. The string pointed to by `iName`
2. If `iName=NULL`, it is *Local-XXXXXXXX* where XXXXXXXX is the hexadecimal address of the `DObject`.

Object short names can be up to 80 characters in length. This makes them shorter than in EKA1, where the maximum was 128 characters. There's another difference too: EKA1 supported Unicode names, whereas in EKA2, names must be in ASCII. We made this decision for several reasons:

- If the kernel were to support Unicode internally, then we would have to duplicate many Unicode functions and large folding tables inside the kernel
- An ASCII compare is much simpler than a Unicode folded compare, so searching for objects by name is faster
- The naming of objects is a programmer convenience, and programmers generally write code in ASCII source files

The object's full name is longer; it can be anything up to 256 characters in length. We define it recursively as the full name of the `DObject`'s owner appended with *::<short name of this object>*. The limit of 80 characters on the length of the short name guarantees that the full name cannot exceed 256 characters, because there can be a maximum of three objects in the owner chain: the `DObject` might be owned by a thread that is owned by a process. For example, a semaphore named ALAZON, owned by the thread EPOS, in turn part of the LEXIS process, would be called LEXIS::EPOS::ALAZON. If you're worrying about thread-relative threads, don't - we no longer allow them in EKA2.

We use a global fast mutex, `DObject::Lock`, to protect the operations of getting an object's name, setting its name and setting its owner. We do this to avoid inconsistent results when one thread renames an object while another is reading its name or full name. (Obviously, we protect the setting of the owner because this changes the full name of the object.)

The method that reads an object's short name, <tt style="font-family:monospace;">DObject::DoAppendName()</tt>, can be over-ridden in a derived class. In fact, the `DLibrary` and `DProcess` classes do over-ride it, because they both include the UID in the object name, and `DProcess` adds a generation number too.

Object indexes and handles

Now that I've described the `DObject` class, I can return to the object index class that is used to record the handles held by user threads or processes on kernel objects

A handle is a 32-bit integer, split into bit fields like this:

Bits	Function
0 - 14	15-bit index into the `DObjectIx` holding the handle.
15	No close flag. If set to 1 the handle cannot be closed using <tt style="font-family:monospace;">RHandleBase::Close()</tt>.
16-29	14-bit instance count (taken from `DObjectIx::iNextInstance`).
30	This field is never zero for a valid handle.
31	0 for normal handles, 1 for special handles. Supported special handles are: FFFF8000 - always refers to the current process FFFF8001 - always refers to the current thread.

Let's have a look at the `DObjectIx` class, along with the SDObjectIxRec structure that it makes use of:

```
struct SDObjectIxRec
  {
  TInt16 instance;
  TInt16 uniqueID;
  DObject* obj;
  };

class DObjectIx : public DBase
  {
public:
  enum {ENoClose=KHandleNoClose, ELocalHandle=0x40000000};
```

```
public:
  IMPORT_C static DObjectIx* New(TAny* aPtr);
  IMPORT_C ~DObjectIx();
  IMPORT_C TInt Add(DObject* aObj, TInt& aHandle);
  IMPORT_C TInt Remove(TInt aHandle, DObject*& aObject, TAny*& aPtr);
  IMPORT_C DObject* At(TInt aHandle,TInt aUniqueID);
  IMPORT_C DObject* At(TInt aHandle);
  IMPORT_C TInt At(DObject* aObject);
  IMPORT_C TInt Count(DObject* aObject);
  IMPORT_C DObject* operator[](TInt aIndex);
  TInt LastHandle();
  static void Wait();
  static void Signal();
  inline TInt Count();
  inline TInt ActiveCount();

protected:
  IMPORT_C DObjectIx(TAny* aPtr);

private:
  void UpdateState();
  TInt iNextInstance;
  TInt iAllocated; // Max entries before realloc needed
  TInt iCount; // At least 1 above the highest active index
  TInt iActiveCount; // No of actual entries in the index
  SDObjectIxRec* iObjects;
  TAny* iPtr;
  TInt iFree; // The index of the first free slot or -1.
  TInt iUpdateDisabled;

public:
  static DMutex* HandleMutex;
  };
```

Key member data of `DObjectIx`

`iNextInstance`	This is a counter that starts at 1, and is incremented every time an object is added to the index. It is incremented again if it would become zero modulo 16384, so that the lower 14 bits range from 1 to 16383.
`iAllocated`	This is the number of slots currently allocated in the `iObjects` array.
`iCount`	This field is 1 + the highest index of any occupied slot in the `iObjects` array.
`iActiveCount`	This is the number of occupied slots in the `iObjects` array.
`iObjects`	This is a pointer to the array of object index records. Each record contains a pointer to a `DObject`, the instance counter modulo 16384 when the entry was added and the unique ID of the `DObjectCon` in which the `DObject` is held.
`iPtr`	This is a pointer to the process that is the ultimate owner of all handles in this index (that is, the thread's owning process for a thread-local handle array). This is passed as a parameter to `<tt style="font-family:monospace;">DObject::Close()</tt>` when a handle is closed.

Finding objects from handles

To translate a handle into a `DObject` pointer, the kernel follows the following steps, which are shown graphically in Figure 5.1:

1. Uses bit 30 of the handle to choose a `DObjectIx` (either the current thread's or the current process's handle array)
2. Takes the bottom 15 bits to use as an index, and checks this index against the `DObjectIx::iCount` value to ensure it is within the array
3. Uses the index to access an entry in the `iObjects` array, SDObjectIxRec structure, of the `DObjectIx`
4. Compares the handle's instance value (bits 16-29) against the instance value stored in the `iObjects` array entry (note that the instance value provides protection against a stale handle being re-used after it has been closed and after the kernel has reallocated its index slot to a new handle. Handle lookup always occurs with the system locked to protect against changes in the handle array while it is being examined). We set the latter from the `DObjectIx::iNextInstance` value when the `DObjectIx` entry was made (which is when the handle was created.)
5. If the two instance values are the same, then the handle is valid
6. Checks the unique ID value in the `iObjects` array entry to ensure that the object pointed to is of the expected type
7. Finally, extracts the `DObject` pointer from the `iObjects` array entry.

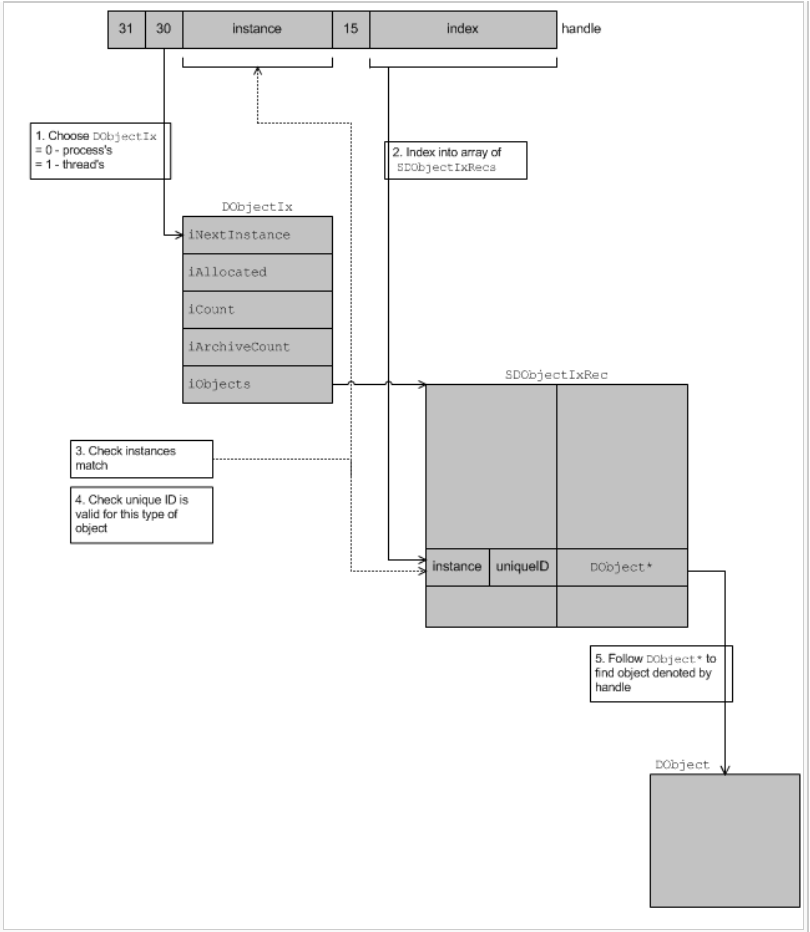

Figure 5.1 Finding an object from a handle

Protection of handle mechanisms

The adding and removing of handles requires some care. The kernel protects additions and removals from `DObjectIx` arrays with a global mutex, `DObjectIx::HandleMutex`. The mutex allows the `Exec::HandleInfo()` function to prevent any handle creation and deletion while it inspects the handle array of every thread and process. The kernel doesn't protect the lookup of handles, though - this would slow it down too much. Instead it looks up handles while holding only the system lock.

If the `iObjects` array needs to grow as a result of adding a handle, the kernel uses the `Kern::SafeReAlloc()` function. This allocates a new larger block first, copies the old contents into it, fills the extra space in the new block with zeros, and then acquires the system lock before replacing the pointer with the address of the new block and deleting the old block. This ensures that any code running with the system locked always sees a valid handle array.

Since a handle contains an index into the `iObjects` array, the removal of an entry from the `iObjects` array cannot result in all the later entries being moved as that would make all the existing handles to those objects invalid. Instead, the kernel sets the entry's object pointer to NULL and adds the entry to the front of the linked list of free slots - that is, it sets `iFree` to point to the entry. When it next adds an object to the index, it will use up the slot denoted by `iFree`.

Object containers

Object containers exist for two reasons:

1. So that the kernel can find objects by name
2. So that the kernel can enumerate all the objects of a certain type.

With some exceptions, such as internal kernel mutexes, whenever the kernel creates a DObject-derived object it adds that object to the `DObjectCon` that corresponds to the object's type.

The kernel removes the object from the container when it deletes it - which, as we saw earlier, happens when the object's access count drops to zero. In fact, the removal of the object from the container is the last action that the kernel does before it frees the memory - because it is the `DObject` destructor that removes the dying `DObject` from the container.

When the kernel adds an object to a container, it checks that the object's full name is unique among the objects in that container. This ensures that the kernel can find the object unambiguously using its full name.

Each `DObjectCon` has its own `DMutex` that protects all accesses to the container, including those that simply index into the container. The kernel indexes into a `DObjectCon` much more rarely than it indexes into a `DObjectIx`, so the overhead of waiting on the mutex is not significant in this case.

Array management for `DObjectCon` is simpler than for `DObjectIx`. The first `iCount` slots are occupied with no gaps - removing an entry will move all the following entries down one. When the kernel adds a new entry, it always goes at the end of the array.

Here is the `DObjectCon` class:

```
class DObjectCon : public DBase
  {
protected:
  enum {ENotOwnerID};

public:
  ~DObjectCon();
  static DObjectCon* New(TInt aUniqueID);
  IMPORT_C void Remove(DObject* aObj);
  IMPORT_C TInt Add(DObject* aObj);
  IMPORT_C DObject* operator[](TInt aIndex);
  IMPORT_C DObject* At(TInt aFindHandle);
  IMPORT_C TInt CheckUniqueFullName(DObject* aOwner,  const TDesC& aName);
  IMPORT_C TInt CheckUniqueFullName(DObject* aObject);
  IMPORT_C TInt FindByName(TInt& aFindHandle, const TDesC& aMatch, TKName& aName);
  IMPORT_C TInt FindByFullName(TInt& aFindHandle, const TDesC& aMatch, TFullName&
aFullName);
  IMPORT_C TInt OpenByFullName(DObject*& aObject, const TDesC& aMatch);
  inline TInt UniqueID() {return iUniqueID;}
  inline TInt Count() {return iCount;}
  inline void Wait() {Kern::MutexWait(*iMutex);}
  inline void Signal() {Kern::MutexSignal(*iMutex);}
  inline DMutex* Lock() {return iMutex;}

protected:
  DObjectCon(TInt aUniqueID);
  TBool NamesMatch(DObject* aObject, DObject* aCurrentObject);
  TBool NamesMatch(DObject* aObject, const TDesC& aObjectName, DObject* aCurrentObject);

public:
  TInt iUniqueID;

private:
  TInt iAllocated;
  TInt iCount;
  DObject** iObjects;
  DMutex* iMutex;
  };
```

Key member data of `DObjectCon`

Field	Description
`iUniqueID`	This is an identity number indicating the type of kernel object held in this container. The value used is 1 + the corresponding value in the `TObjectType` enumeration - for example the identity number for threads is 1.
`iAllocated`	This is the number of slots currently allocated in the `iObjects` array.
`iCount`	This is the number of slots currently occupied in the `iObjects` array.
`iObjects`	This is the pointer to the array of pointers to `DObjects` that are currently held in this container.
`iMutex`	This is the pointer to the `DMutex` mutex object that the kernel uses to protect accesses to this container.

Services provided to user threads

Executive call mechanism

The kernel provides services to user-mode code using a mechanism that we call executive calls, or exec calls for short. Exec calls begin as a standard user-side function, and then use a software exception as a gateway to allow them to enter kernel code. The software exception instruction switches the CPU into supervisor mode and starts the execution of kernel code at a defined entry point - see Chapter 6, *Interrupts and Exceptions*, for more on this.

The CPU's instruction set generally limits the number of possible entry points from software interrupts or traps - for example, on an ARM CPU there is only one SWI instruction to enter supervisor mode. Because of this, we use a dispatcher in the nanokernel to decode a parameter passed from user side, determine the function required and then call it. On ARM CPUs, the parameter is the opcode used with the SWI instruction, and this determines the function that the dispatcher calls.

This calling mechanism results in a very loose coupling between the kernel and user processes, and this means that we can make design changes within the kernel more easily.

Flow of execution in an executive call

Now I'll show the flow of execution from a user-mode application to supervisor-mode kernel code and back again. Let's choose an example to trace:

```
TUint8* Exec::ChunkBase(ChunkHandle)
```

This executive call returns a pointer to the start of a chunk belonging to the calling thread. The parameter passed is the handle of the chunk within the thread.

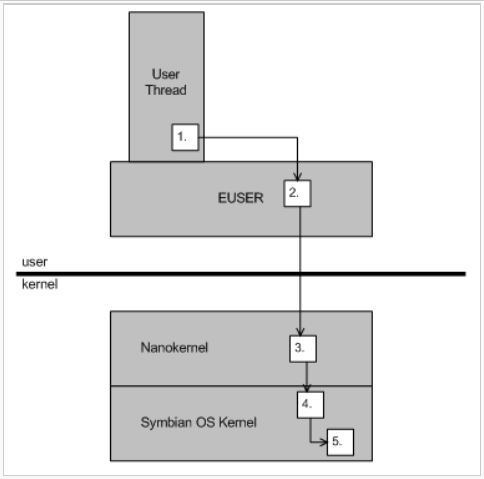

Figure 5.2 *Kernel executive call*

You can follow my explanation in Figure 5.2.

1. **User thread:** Let's assume that a user-side thread is executing the following section of code:

```
RChunk newChunk=0;
newChunk=OpenGlobal(_L(*SharedChunk),ETrue);
TUint* base=0;
base=newChunk.Base();
```

This code segment opens a shared chunk and stores the handle returned in newChunk. Next it wants to find out the base address of this chunk, which it does by calling <tt style="font-family:monospace;">RChunk::Base()</tt>. I will trace this operation from the user side into the kernel, via an executive call.

The code for the <tt style="font-family:monospace;">RChunk::Base()</tt> method is found in the file \e32\euser\us_exec.cpp and looks like this:

```
EXPORT_C TUint8 *RChunk::Base() const
  {
  return(Exec::ChunkBase(iHandle));
  }
```

So `RChunk::Base()` calls `Exec::ChunkBase()`, which is in the user library, EUSER.DLL.

2. **User library:** `Exec::ChunkBase()` is in the file \epoc32\include\exec_user.h, and is generated by entering *ABLD MAKEFILE GENEXEC* in the E32 directory. The ABLD tool takes the file execs.txt, and uses it to generate the source code for the user-side executive calls. The portion of execs.txt we are interested in is this:

```
slow
 {
  name = ChunkBase
  return = TUint8*
  handle = chunk
 }
```

You can see that it tells the tools to generate a function named ChunkBase which returns a pointer to `TUint8`, and which is passed a handle to a chunk.

The generated `Exec::ChunkBase()` function looks like this:

```
__EXECDECL__ TUint8* Exec::ChunkBase(TInt)
  {
  SLOW_EXEC1(EExecChunkBase);
  }
```

In \e32\include\u32exec.hwe have:

```
#elif defined(__CPU_ARM)
// Executive call macros for AR
```

```
#define EXECUTIVE_FAST 0x00800000
#define EXECUTIVE_SLOW 0x00000000
#define __DISPATCH(n) \
    asm("mov ip, lr "); \
    asm("swi %a0" : : "i" (n));
#define FAST_EXEC0(n) __DISPATCH((n)|EXECUTIVE_FAST)
#define FAST_EXEC1(n) __DISPATCH((n)|EXECUTIVE_FAST)
#define SLOW_EXEC0(n) __DISPATCH((n)|EXECUTIVE_SLOW)
#define SLOW_EXEC1(n) __DISPATCH((n)|EXECUTIVE_SLOW)
#define SLOW_EXEC2(n) __DISPATCH((n)|EXECUTIVE_SLOW)
#define SLOW_EXEC3(n) __DISPATCH((n)|EXECUTIVE_SLOW)
#define SLOW_EXEC4(n) __DISPATCH((n)|EXECUTIVE_SLOW)
```

When you disentangle the macros, you can see that `Exec::ChunkBase()` makes this SWI call to enter supervisor mode:

```
SWI EExecChunkBase
```

EExecChunkBase is an enumeration that gives the opcode for the SWI call.

3. **Nanokernel dispatcher:** We enter the nanokernel at the function __ArmVectorSwi, in vectors.cia. This function makes much use of the executive tables, which are defined like this:

```
GLREF_D const TUint32 EpocFastExecTable[];
GLREF_D const TUint32 EpocSlowExecTable[];
```

Essentially, the fast executive table consists of a number of 32-bit entries, the nth of which is the address of the handler for the nth fast exec call. The slow executive table consists of pairs of 32-bit entries, the first of which is a set of attribute flags, and the second of which is the address of the slow exec call handler. I will cover this subject in more detail in Section 5.2.1.6.

The function __ArmVectorSwi first checks bit 23 of the ARM opcode to find out whether this is a slow exec call or a fast one. If bit 23 is 1, then this is a fast exec call, and the dispatcher will switch interrupts off before indexing into the fast exec table, and calling the relevant kernel function.

In our case, bit 23 is 0, so ours is a slow exec call. Next the dispatcher checks bit 31 in the attribute word of the slow exec table. `EExecChunkBase` has this bit set, so the dispatcher locks the system by taking the system lock fast mutex.

The dispatcher goes on to check another bit in the attribute word to see if it should call the Symbian OS preprocessing handler, PreprocessHandler, the address of which it discovers from the second word of the slow exec table. The dispatcher always claims the system lock before calling PreprocessHandler.

Again, in our case this bit is set, so the dispatcher calls PreprocessHandler. I'll discuss this in the next section.

On returning from Preprocess Handler, the dispatcher finally calls the relevant OS function: in our case this is `ExecHandler::ChunkBase()`.

Finally the dispatcher checks a bit to see whether it should release the system lock fast mutex, and after doing so if required, it returns to the user library.

4. **Preprocessing handler (optional):** The preprocessing handler is part of the Symbian OS kernel (rather than the nanokernel) and is found in cexec.cia. It looks up handles to kernel objects. The preprocessing handler has access to the following information:

- The arguments passed to the executive function, which include the handle to look up. The preprocessing handler may modify these arguments as part of its execution
- The attribute flags of the executive call, the bottom five bits of which specify the type of kernel object that the handle refers to. On return, the preprocessing handler will have replaced the handle with a pointer to the kernel object to which it refers. There are various special handles that the preprocessing handler must pay attention to. Firstly, there are the two handles defined in e32const.h:

```
e32const.h:

//A flag used by the kernel to indicate the current process.
const TInt KCurrentProcessHandle=0xffff0000|KHandleNoClose;

//A flag used by the kernel to indicate the current thread.
const TInt KCurrentThreadHandle=0xffff0001|KHandleNoClose;
```

Then there are three special handle types:

```
// lookup IPC message handle, allow disconnect
EIpcMessageD=0x20,
```

```
// lookup IPC message handle, don't allow disconnect
EIpcMessage=0x21,

// lookup IPC message client, don't allow disconnect
EIpcClient=0x22,
```

Handles like of this type are *magic* values that refer to a client/server IPC message. In the case of `EIpcClient` type, this means *the thread that sent the message*. The magic value is in fact the address of the `RMessageK` object stored within the kernel! Don't worry - the kernel performs strict validation checks on this object to prevent security breaches. Returning to our simpler example, the preprocessing handler merely looks up the handle in the owning thread or process, and returns with a pointer to the corresponding `DChunk`.

5. **OS function:** The exec handling function that the dispatcher calls may be almost anywhere in kernel - in the nanokernel, the Symbian OS kernel, the memory model or even the variant. In our example, `ExecHandler::ChunkBase()` is in the file sexec.cpp, which is part of the Symbian OS kernel. This function simply retrieves the base of the chunk from the `DChunk`, like this:

```
TUint8 *ExecHandler::ChunkBase(DChunk* aChunk)
  // Return the address of the base of the Chunk.
  {
  return (TUint8 *)aChunk->Base();
  }
```

Context of executive call

An exec call executes in the context of the calling user-mode thread, not that of any kernel thread. The only changes that happen on entry to the kernel are:

- The processor switches into supervisor mode
- The active stack changes from the current thread's user stack to the current thread's supervisor stack.

Because of this, you can't make an exec call from an interrupt service routine or an IDFC, because in these situations there is no thread context.

Changes from EKA1

The exec call mechanism has changed considerably from EKA1 to EKA2. On EKA1, exec calls borrow the kernel server or the null thread stack, rather than running on the calling thread's own supervisor stack as they do on EKA2. For this, and other reasons, EKA1 exec calls have the following restrictions:

1. They are not preemptible
1. They can't block in the kernel
2. They can't allocate and free kernel memory.

On EKA1, if a user-mode thread needed to call a service that allocated or freed kernel memory (for example, a service that created or destroyed objects derived from `CObject`), then that user-mode thread had to make a special kind of kernel call, known as a kernel server call. This is no longer the case in EKA2.

As we've seen, on EKA2 exec calls run on the supervisor stack of the calling thread. This means that exec calls can be preempted and they can block in the kernel. Furthermore, because EKA2 does not link to EUSER, exec calls may allocate and free kernel memory.

Accessing user-mode memory

Earlier in this chapter, I said that an exec call runs in the context of the calling thread. This means that on systems with an MMU and multiple processes running in separate address spaces, the active address space is still that of the process to which the calling thread belongs. It is therefore theoretically possible for the kernel-side exec call to directly access the memory of the user process that called it, by dereferencing a pointer or using memcpy(). However, in practice we do not allow this. This is because the exec call is executing kernel code with supervisor privileges, and can therefore read and write anywhere in the processor's address space, which of course includes kernel memory. If the exec call dereferences a pointer given to it by a user thread without checking that pointer, then we are effectively giving the user thread the freedom to access all of the address space too. This defeats platform security and makes it more likely that an invalid pointer from the user application will overwrite a key part of the kernel, crashing the mobile phone.

The kumem functions

Does this mean that exec calls can't access the memory of the user process that called them? No, because we provide the special kernel functions kumemget(), kumemput() and kumemset() to dereference the pointers that are passed from user code. You should use these functions yourself if you are writing a device driver or an extension that is passed pointers to user data from user-side code.

The kumem functions access memory with special CPU instructions that perform the access at user privilege level - for example LDRT/STRT on ARM. Here is the relevant portion of the kumemget() function, this time on X86 for a change:

```
_asm mov ax, gs
_asm mov ds, ax
```

```
_asm call CopyInterSeg
```

On entry to the function, GS contains the data segment of the caller - this is obviously a user-mode data segment in the case of an exec call. We move GS to DS before we call `CopyInterSeg()`, which copies ECX bytes from DS:ESI to ES:EDI. This means that the user's data segment is used as the source of the copy, and the memory move therefore respects the privileges of the caller.

Slow and fast executive calls compared

I mentioned earlier that the dispatcher checks a bit in the SWI opcode to determine whether the exec call is a slow or a fast one. In this section, I'll discuss these two forms of exec call in more detail and point out the differences between them.

Slow executive calls Slow exec calls run with interrupts enabled and the kernel unlocked. This means that they can be preempted at any point in their execution.

As we saw in the walk-through, slow exec calls have a mechanism for automatically performing certain actions when in the dispatcher. This mechanism relies on particular bits being set in the attribute word of the slow executive table.

Using this mechanism, a slow exec call may:

- Acquire the system lock fast mutex before calling the kernel handler
- Release the system lock after calling the kernel handler
- Call a Symbian OS preprocessing handler to look up a Symbian OS handle. In this case, the call always acquires the system lock too.

A key difference between slow and fast execs is that the user side can pass many more parameters to a slow exec call. In their standard form, slow execs can have up to four direct 32-bit arguments and can return one 32-bit value. If this isn't enough, then the slow exec call can also copy as many as eight additional 32-bit values from user space to the current thread's supervisor stack. If this is done, then we have to use one of the four direct arguments to point to the additional arguments, so we can pass a maximum of eleven arguments in total.

These extra exec call arguments are a new feature of EKA2 that is not available on EKA1. On EKA1, you could pass extra arguments, but only by passing a pointer to an arbitrary amount of additional user-mode data as one of the four standard parameters. The EKA1 kernel would then access this data directly, which, as I discussed in Section 5.2.1.4, is not safe. EKA2 allows the extra arguments to be passed in a way that does not compromise robustness or security, because the new kernel uses the kumem functions to access the additional data.

The mechanism by which the extra parameters are passed is dependent on the CPU architecture. If the processor has sufficient registers, then we use those that are not already in use. For example, on ARM, we pass the extra arguments in R4-R11; this means that the user-side Exec:: functions must save these registers and load the additional arguments into them before executing the SWI instruction to enter the kernel. The Exec:: functions must then restore those registers on return from the kernel. The dispatcher pushes R4-R11 onto the supervisor stack and then sets R2 (the third normal argument) to the address of the saved R4.

On X86, we use the third argument to pass a pointer to the additional arguments in user memory. The dispatcher copies the specified number of arguments from user memory space to the current thread's supervisor stack and modifies the third argument to refer to the copied arguments. If the SWI opcode has its system lock bit set, then the dispatcher copies the arguments before it acquires the system lock. We do it this way in case an exception occurs during the copying of the additional arguments because the supplied address is invalid. Then, if this does happen, the kernel can terminate the current thread without a problem.

Regardless of the CPU architecture, the executive handler always accesses the additional arguments by using the third normal argument as a pointer to them. By the time the executive handler runs, the dispatcher will have copied the additional arguments to the current thread's supervisor stack, and changed the third argument to refer to that copy. This means that the executive handler does not need to check that the referenced address is a valid user mode address.

Fast exec calls As we saw earlier, slow exec calls run with interrupts enabled. Fast exec calls, on the contrary, run with all interrupts disabled. This is another difference from EKA1, where they ran with IRQ interrupts disabled and FIQ interrupts enabled. Because of this, EKA2 fast exec calls must be very short. There aren't many of them, and typically they get or set a single, easily accessible, item of kernel data. For example, there is a fast exec call to get the current thread's heap pointer.

We saw that slow exec calls can pass up to eleven parameters. Fast exec calls, on the other hand, can only pass one 32-bit parameter. They may also return a single 32-bit value.

Executive tables

I have already mentioned that we specify the range of valid fast and slow executive calls and their associated handlers using two tables - the fast executive table and the slow executive table. Every nanokernel thread in the system has two pointers, one to each of these tables. The kernel sets up these pointers when the thread is created, which means that the available executive calls can be changed on a thread-by-thread basis. All Symbian OS threads do in fact use the same tables, but this feature makes it possible for threads in an RTOS personality layer to use different tables, if desired. It is worth noting that you would only need to use this feature if you had user-mode personality layer threads. If your threads only ever run in supervisor mode, then you can call your required personality layer services directly.

The fast executive table

The fast executive table is composed of a number of 32-bit entries, like so:

Word index	Description
0	Number of fast executive calls supported.
n >= 1	Address of handler for fast executive call number n.

You can see that fast executive call 0 has no entry in the table. This is because it is always assigned to wait on the current thread's request semaphore.

If a thread makes a fast executive call with a number that is greater than or equal to the number of calls specified in the table, then the kernel calls the invalid executive handler, which is specified in the slow executive table.

The slow executive table

The slow executive table is composed of three single-word entries followed by an arbitrary number of two-word entries, like so:

Word index	Description
0	Number of slow executive calls supported.
1	Address of handler for invalid call number.
2	Address of handler for argument preprocessing.
3+2n	Attribute flags for slow executive call number n.
4+2n	Address of handler for slow executive call number n.

If a thread makes a slow executive call with a number that is greater than or equal to the number of calls specified in the table, then the kernel calls the invalid executive handler, which is specified in word 1 of the table. Invalid fast exec calls are routed here too, but even in this case the kernel treats the invalid handler as a slow executive call with its attribute flags all zero.

I've mentioned the attribute flags already in the walk-through and in my discussions about the differences between slow and fast exec calls. These flags determine any additional actions that the dispatcher performs before calling the handler and after returning from it. Here are the details of the functions associated with each bit:

Bit	Description
31	If this bit is set to 1, the system lock fast mutex will be acquired prior to calling the executive handler
30	If this bit is set to 1, the system lock fast mutex will be released after returning from the executive handler.
29	If this bit is set to 1, the preprocessing handler will be called prior to calling the executive handler. Note that if bit 31 is also set to 1, the system lock is acquired before calling the preprocessing handler.
26, 27, 28	These bits make a three-bit wide field indicating the number of additional arguments required by the executive call. A value of 0 indicates that there are no additional arguments; a value of n, where 1 <= n <= 7 indicates that there are n + 1 additional arguments. Thus up to eight additional arguments may be specified.

Kernel server calls

If you know EKA1, you may be wondering why I haven't mentioned kernel server calls. Let me explain a little bit about them, and then I hope the reason will become clear.

As I've said, EKA1 makes use of the EUSER library. The heap functions in EUSER allocate and free memory on the heap of the current thread. This made it difficult for any EKA1 exec calls that resulted in the creation (or destruction) of kernel objects - those objects must be created on the kernel heap, but during the executive call the thread context is that of the thread making the executive call.

So, to ensure that the memory was allocated on the kernel heap, we had to engineer a switch to a kernel thread context. To do this, an EKA1 thread executes a special exec call that makes a request from the kernel server thread and then blocks awaiting the reply. At the next reschedule, the kernel server thread will run (as it is the highest priority thread in the system) and obviously it can then create or destroy objects on its own heap on behalf of the user thread.

EKA2 has its own memory allocation routines, and does not link to EUSER. This means that EKA2 exec calls can allocate and free kernel memory and we do not need kernel server calls.

Executive calls in the emulator

The emulator can't use a software interrupt to implement executive calls, so instead it uses a function call but with a special calling convention.

The executive dispatcher lives in the nanokernel, but the calls themselves are in the user library (EUSER.DLL). To prevent EUSER.DLL depending on EKERN.EXE, this call is not done using the standard import machinery. Instead, there is a function pointer to the dispatcher in EUSER, which is initialized lazily to point to the first ordinal in EKERN.EXE - this is the only export from EKERN that must be maintained in EKA2's emulator. The executive functions in EUSER first set up two parameters (the executive number and the pointer to the parameters, which are all on the thread stack), then they jump to the nanokernel dispatcher function.

The dispatcher then handles the executive in a way which is similar to that on a phone: the executive function runs with the thread in *kernel mode*, fast executive calls run with interrupts disabled and slow executive calls can manipulate the system lock and have preprocessing done on their parameters.

Example user-accessible services

In this section, I'm just aiming to give you a feel for the kind of services that the kernel provides via EUSER, and how we decide to categorize each exec call.

Fast exec calls

As we saw, fast executive calls run with all interrupts off, so they must do their tasks very quickly and then return to the user. Generally these calls just get or set a single word of kernel memory. Here are some examples:

`RAllocator* Exec::Heap()` Returns the current thread's heap.

`TUint32 Exec::FastCounter()` Returns the value of the fast counter, which can be used in profiling.

`Exec::SetDebugMask(TUint32)` Sets the kernel's debug bit mask to determine the level of printf() debugging displayed on the serial port. Often used in debug code to restrict debug printing to key areas of interest.

Slow exec calls

Services that don't claim the system lock

These services are ones which do not need to lock the system to protect them from their own side effects - that is, two concurrent calls to the same exec call will not interfere with each other. These services often read, rather than modify, kernel data. Examples are:

`void Exec::IMB_Range(TAny* aBase, <tt style="font-family:monospace;">TUint` aLength)</tt> Performs all necessary cache management for the address range aBase to aBase+aLength in order that whatever has been written there can be executed. This is known as an instruction memory barrier (IMB).

TUint `Exec::TickCount()` Returns the number of system ticks since boot.

`void Exec::DebugPrint(TAny* aDebugText, <tt style="font-family:monospace;">TInt` aMode)</tt> Passes in a descriptor with text to print out as a debug string, and a mode to print in.

Services that claim the system lock

As we've seen, certain slow exec calls have a bit set in their attribute word to say that the dispatcher should lock the system before calling the executive handler in the kernel. The main reason for this is to protect certain kernel resources against multiple accesses.

Examples of this type of service are:

`TUint32 Exec::MathRandom()` Returns a random number. Since this code is not re-entrant, the system is locked.

`void Exec::CaptureEventHook()` The window server calls this function to capture the event hook. Only one thread may own this event hook, so the system is locked to prevent a second thread gaining access to the function before the first thread has flagged that it has taken the hook by setting the kernel variable K::EventThread to point to itself. On the secure kernel, this function panics if the thread taking the event hook is not the window server thread.

Services passing handles

Certain slow exec calls have a bit set in their attribute word to say that the dispatcher should call a preprocessing handler in the Symbian OS kernel before calling the executive handler in the kernel. The preprocessing handler takes the first argument of the slow exec call, which is always a handle, and translates it into a `DObject` derived object pointer.

Any slow exec call that calls the preprocessing handler also claims the system lock.

Examples of this type of service are:

`TUint8* Exec::ChunkBase(ChunkHandle aHandle)` Returns a pointer to the start of a chunk.

`TInt Exec::ThreadId(ThreadHandle aHandle)` Returns the ID of the given thread.

`TlibraryFunction LibraryLookup(LibraryHandle aHandle, aFunction)` Returns the address of the required function number in the given library.

Services where the dispatcher doesn't release the lock

These exec calls claim the system lock on entry, but don't unlock it on exit. This is because the exec handler functions release the system lock themselves.

Examples of this type of service are:

`void Exec::MutexWait(MutexHandle aHandle)` Waits on the given mutex.

`void Exec::ProcessSetPriority (ProcessHandle aProcess, TProcessPriority aPriority)` Sets the priority of the given process.

`void Exec::SemaphoreSignalN(SemHandle aHandle, aNum)` Signals the given semaphore a number of times.

HAL functions

As we've seen, the EKA2 kernel is not linked to, and never calls, the user library, EUSER.DLL. This is a major difference from EKA1, which often used the user library as a way to call its own services, going via an executive call and a supervisor mode SWI to the required service, even though it was already executing in supervisor mode.

Not only does EKA2 not call EUSER, it rarely makes a SWI call either - clearly a good thing for its performance. In fact, there is only one place where EKA2 does make a SWI call - `Kern::HalFunction()`. This function is used to request a service from a kernel extension, and user threads call it via the function `UserSvr::HalFunction()`.

The hardware abstraction layer, or HAL, consists of a set of hardware or system attributes that can be set or read by software. These are broken down into groups of like functionality, as enumerated by `THalFunctionGroup:`

```
enum THalFunctionGroup
  {
  EHalGroupKernel=0,
  EHalGroupVariant=1,
  EHalGroupMedia=2,
  EHalGroupPower=3,
  EHalGroupDisplay=4,
  EHalGroupDigitiser=5,
  EHalGroupSound=6,
```

```
  EHalGroupMouse=7,
  EHalGroupEmulator=8,
  EHalGroupKeyboard=9,
  };
```

Each of these groups then has a set of attributes. For example, the first group, `EHalGroupKernel`, has these attributes:

```
enum TKernelHalFunction
  {
  EKernelHalMemoryInfo,
  EKernelHalRomInfo,
  EKernelHalStartupReason,
  EKernelHalFaultReason,
  EKernelHalExceptionId,
  EKernelHalExceptionInfo,
  EKernelHalCpuInfo,
  EKernelHalPageSizeInBytes,
  EKernelHalTickPeriod,
  EKernelHalMemModelInfo,
  };
```

Each HAL group has a handler function that manages the group's attributes. This handler can be dynamically installed by using the function `Kern::AddHalEntry()`. For example, some HAL groups correspond to a particular hardware device, like the screen display or keyboards, and the kernel extension or device drivers for these devices will install a handler.

As I said earlier, the kernel accesses HAL functions via `Kern::HalFunction()`:

```
EXPORT_C __NAKED__ TInt Kern::HalFunction(TInt aGroup, TInt aFunction, TAny* a1, TAny*
a2, TInt aDeviceNumber)
  {
  asm("ldr ip, [sp, #0] ");
  asm("orr r0, r0, ip, lsl #16 ");
  asm("mov ip, lr ");
  asm("swi %a0" : : "i"(EExecHalFunction|EXECUTIVE_SLOW));
  }
```

You can see that the first and second parameters are the group and the number of the function. The remaining parameters, if present, are passed to the HAL function itself.

Services provided by the kernel to the kernel

In the introduction to this book, I mentioned that we could consider the architecture of EKA2 from a software layering perspective, as shown in Figure 5.3, and went on to discuss the kind of software that appeared at each layer.

In this chapter, I am more concerned with the services each layer provides to the other layers.

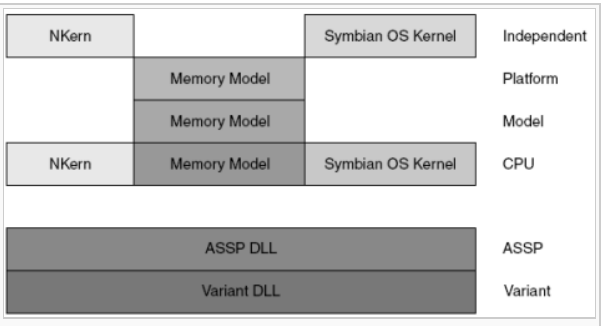

Figure 5.3 *Software layering*

Independent layer

Nanokernel

The static interface to the independent nanokernel is provided through the class `NKern`, which is defined in nkern.h. The APIs in this class cover a few key areas of interest, which I'll discuss now.

Threads

NKern provides a static interface to nanothread manipulation, using an `NThread*` parameter. This allows callers to create a nanothread, to kill it, to suspend it, to release it and more. Here are a couple of examples:

```
static void ThreadKill(NThread* aThread)
```

```
static void ThreadSetPriority(NThread* aThread, TInt aPriority);
```

Timers

As we saw in Chapter 2, *Hardware for Symbian OS*, the kernel needs hardware to provide a periodic tick interrupt; this timer must be started from the ASSP's or variant's `Init3()` function. The period of this tick determines the timer resolution and is usually set to 1 ms - hence it is frequently known as the millisecond timer. The tick interrupt's interrupt handler calls the `Tick()` method in the nanokernel's timer queue class, `NTimerQ`.

Nanokernel timers provide the most fundamental system timing functions in the operating system. Symbian OS tick-based timers and time-of-day functions are both derived from nanokernel timers. In addition, the nanokernel timer service supports timed wait services, if implemented. The tick interrupt is also used to drive the round-robin scheduling for equal-priority thread.

I will discuss timers in more detail in Section 5.5.

Fast semaphores and mutexes

The NKern semaphore and mutex APIs allow their callers to wait on and signal nanokernel fast mutexes and semaphores. Here are the two fast mutex APIs:

```
static void FMWait(NFastMutex* aMutex);
static void FMSignal(NFastMutex* aMutex);
```

Interrupts

The NKern interrupt APIs allow their callers to enable and disable interrupts: globally, or to a certain level. For example:

```
static TInt DisableAllInterrupts();
void EnableAllInterrupts();
```

Read-modify-write

The NKern read-modify-write APIs allow their callers to atomically increment or decrement a counter, preventing side-effects from two threads attempting to access the same counter. For example:

```
static TInt LockedInc(TInt& aCount);
static TInt LockedDec(TInt& aCount);
```

Key concrete classes

The independent nanokernel also provides key classes that are used by the rest of the kernel. I have covered or will cover these in other chapters, so here it will suffice to enumerate them:

- `NFastSemaphore`
- `NFastMutex`
- `TDfc`.

Symbian OS kernel

The static interface to the independent Symbian OS is provided through the class Kern, which is defined in kernel.h. The APIs in this class cover a wide miscellany of topics, of which I'll pick out a few.

Thread read and write

The Kern class provides APIs to allow other parts of the kernel to safely read and write from threads' address spaces.

```
static TInt ThreadDesRead(DThread* aThread, const TAny* aSrc,
  TDes8& aDest, TInt aOffset, TInt aMode);
static TInt ThreadRawRead(DThread* aThread, const TAny* aSrc,
  TAny* aDest, TInt aSize);

static TInt ThreadDesWrite(DThread* aThread, TAny* aDest,
  const TDesC8& aSrc, TInt aOffset, TInt aMode, Thread* aOrigThread);
static TInt ThreadRawWrite(DThread* aThread, TAny* aDest,
  const TAny* aSrc, TInt aSize, DThread* aOrigThread=NULL);
```

Access to kernel variables

In this case, a variety of examples is worth a thousand words:

```
static TTimeK SystemTime();
static DPowerModel* PowerModel();
static DObjectCon* const *Containers();
static TSuperPage& SuperPage();
```

```
static TMachineConfig& MachineConfig();
static DThread& CurrentThread();
static DProcess& CurrentProcess();
```

Key concrete classes

At this level, the Symbian OS kernel provides the abstractions of key kernel objects such as `DThread`, `DProcess`, and `DChunk`. I discuss these in detail in Chapter 3, *Threads, Processes and Libraries* and Chapter 7, *Memory Models*.

Platform (or image) layer

Memory model

The memory model is the only module in the platform layer, because this layer is essentially concerned with executable images on disk, and processes in memory. This means that there are only two possibilities at the platform layer: EPOC for a real mobile phone platform or WIN32 for the emulator.

The platform layer provides static APIs to the independent layer in the class P, which is defined in kern_priv.h. This is very short, so I'll show you all of it:

```
class P
  {
public:
  static TInt InitSystemTime();
  static void CreateVariant();
  static void StartExtensions();
  static void KernelInfo(TProcessCreateInfo& aInfo, TAny*& aStack, TAny*& aHeap);
  static void NormalizeExecutableFileName(TDes& aFileName);
  static void SetSuperPageSignature();
  static TBool CheckSuperPageSignature();
  static DProcess* NewProcess();
  };
```

You can see that the platform layer takes part, as expected, in certain key initializations. It starts the system clock (reading the system time on Win32, the RTC on a mobile phone), starts the extensions (including the variant) and then creates the actual variant object by calling `A::CreateVariant()`. I will talk about this more in Chapter 16, *Boot Processes*.

Key concrete classes

The most important class with a platform specific implementation is the Symbian OS process, `DProcess`. The implementation is provided by the derived `DEpocProcess` class on the EPOC platform and `DWin32Process` on the emulator.

Model layer

Memory model

The model layer is the place in which we have isolated all the kernel's assumptions about memory hardware and layout. The main functions that this layer provides are low-level memory management - how the MMU is used and how the address space is configured.

Symbian OS currently supports four memory models - one for the WIN32 platform (the emulator model) and three for the EPOC platform (moving, multiple and direct). If you want to find out more, turn to Chapter 7, *Memory Models*.

There are two static interfaces to the memory model. The first is defined in the class Epoc, in platform.h. This is a common interface to all EPOC memory models, which is provided for use by extensions and device drivers. It looks like this:

```
class Epoc
  {
public:
  IMPORT_C static void SetMonitorEntryPoint(TDfcFn aFunction);
  IMPORT_C static void SetMonitorExceptionHandler(TLinAddr aHandler);
  IMPORT_C static TAny* ExceptionInfo();
  IMPORT_C static const TRomHeader& RomHeader();
  IMPORT_C static TInt AllocShadowPage(TLinAddr aRomAddr);
  IMPORT_C static TInt FreeShadowPage(TLinAddr aRomAddr);
  IMPORT_C static TInt FreezeShadowPage(TLinAddr aRomAddr);
  IMPORT_C static TInt AllocPhysicalRam(TInt aSize,
    TPhysAddr& aPhysAddr, TInt aAlign=0);
  IMPORT_C static TInt FreePhysicalRam(TPhysAddr aPhysAddr, TInt aSize);
  IMPORT_C static TInt ClaimPhysicalRam(TPhysAddr aPhysAddr, TInt aSize);
  IMPORT_C static TPhysAddr LinearToPhysical(TLinAddr aLinAddr);
  IMPORT_C static void RomProcessInfo(TProcessCreateInfo& aInfo,
    const TRomImageHeader& aRomImageHeader);
```

```
  };
```

You can see that this interface provides functions for allocating physical RAM, for finding information in ROM, and for converting linear addresses to physical ones.

The second interface to the memory model is in class M, in kern_priv.h. This consists of functions provided by the memory model to the independent layer. Here it is:

```
class M
  {
public:
  static void Init1();
  static void Init2();
  static TInt InitSvHeapChunk(DChunk* aChunk, TInt aSize);
  static TInt InitSvStackChunk();
  static TBool IsRomAddress(const TAny* aPtr);
  static TInt PageSizeInBytes();
  static void SetupCacheFlushPtr(TInt aCache, SCacheInfo& c);
  static void FsRegisterThread();
  static DCodeSeg* NewCodeSeg(TCodeSegCreateInfo& aInfo);
  };
```

You can see that this class mainly provides initialization functions that the independent layer calls during startup.

Key concrete classes

At this level you can find model specific implementations of many key Symbian OS classes. For example, `DMemModelChunk` derives from `DChunk` and `DMemModelThread` derives from `DThread`. On the EPOC platform the `DMemModelProcess` class derives from `DEpocProcess`, which in turn derives from `DProcess`. On the emulator, the concrete class representing a process is `DWin32Process`, which derives directly from `DProcess`.

CPU layer

Nanokernel and Symbian OS kernel

The CPU layer is where we make assumptions about the particular processor we're running on - is it X86 or ARM? This is the layer in which you might expect to see some assembler making an appearance. In fact, a sizable proportion of the code in the ARM CPU layer of the Symbian OS kernel is actually independent layer functionality that has been assembler coded for improved performance.

There are two static interfaces to the CPU layer nanokernel and Symbian OS kernel. The first is provided in the class Arm, which is defined in arm.h, and is an interface to the ARM CPU layer for the use of the variant. (There is a similar class X86 for the X86 CPU layer.) The Arm class looks like this:

```
class Arm
  {
public:
  enum {EDebugPortJTAG=42};
  static void Init1Interrupts();
  static TInt AdjustRegistersAfterAbort(TAny* aContext);
  static void GetUserSpAndLr(TAny* aReg[2]);
  static void SetUserSpAndLr(TAny* aReg[2]);
  IMPORT_C static void SetIrqHandler(TLinAddr aHandler);
  IMPORT_C static void SetFiqHandler(TLinAddr aHandler);
  IMPORT_C static TInt DebugOutJTAG(TUint aChar);
  IMPORT_C static TInt DebugInJTAG(TUint32& aRxData);
  IMPORT_C static void SetCpInfo(TInt aCpNum, const SCpInfo* aInfo);
  IMPORT_C static void SetStaticCpContextSize(TInt aSize);
  IMPORT_C static void AllocExtraContext(TInt aRequiredSize);
  static void CpInit0();
  static void CpInit1();
  static Uint64 IrqStack[KIrqStackSize/8];
  static Uint64 FiqStack[KFiqStackSize/8];
  static Uint64 ExceptionStack[KExceptionStackSize/8];
  };
```

You can see that a key use case is to allow the variant to install primary interrupt dispatchers.

The second interface class, class A, provided in kern_priv.h, contains CPU layer APIs that are called by both the memory model and independent layer - but mainly the latter.

```
class A
  {
public:
  static void Init1();
  static void Init2();
  static void Init3();
  static void DebugPrint(const TDesC8& aDes);
  static void UserDebugPrint(const TText* aPtr, TInt aLen, TBool aNewLine);
  static TInt CreateVariant(const TAny* aFile);
  static TInt NullThread(TAny*);
  static DPlatChunkHw* NewHwChunk();
  static TPtr8 MachineConfiguration();
  static void StartCrashDebugger(const TDesC8& aDes, TInt aFault);
  static TInt MsTickPeriod();
  static TInt CallSupervisorFunction(TSupervisorFunction aFunction, TAny* aParameter);
  static TInt VariantHal(TInt aFunction, TAny* a1, TAny* a2);
  static TInt SystemTimeInSecondsFrom2000(TInt& aTime);
  static TInt SetSystemTimeInSecondsFrom2000(TInt aTime);
  };
```

Again you can see that a large part of this interface's purpose is to assist at initialization time.

Memory model

The memory model also appears in the CPU layer. In fact, the bottom layer of the memory model is both CPU- and MMU-specific, as well as specific to the type of memory model.

The key class that the memory model provides is ArmMmu (or X86Mmu on X86 processors). This class is derived from Mmu, which in its turn is derived from MmuBase. The methods provided by this class allow the standard MMU operations, such as the mapping and unmapping of pages, the changing of page permissions and so on. Here are a few examples:

```
virtual void Map(TLinAddr aLinAddr, TPhysAddr aPhysAddr, TInt aSize,
  TPde aPdePerm, TPte aPtePerm, TInt aMapShift);
virtual void Unmap(TLinAddr aLinAddr, TInt aSize);
virtual void ApplyTopLevelPermissions(TLinAddr anAddr, TUint aChunkSize,
  TPde aPermissions);
```

Key concrete classes

At this level, you can see MMU-specific portions of key Symbian OS classes, namely DArmPlatThread, `DArmPlatChunk` and `DArmPlatProcess`.

Variant layer

The variant provides the hardware-specific implementation of the control functions expected by the nanokernel and Symbian OS kernel.

The class Asic, provided in assp.h, contains pure virtual APIs, which are to be provided by the variant and called by the CPU layer. So, if you are creating a variant, you would derive it from the Asic class:

```
class Asic
  {
public:
  // initialisation
  virtual TMachineStartupType StartupReason()=0;
  virtual void Init1()=0;
  virtual void Init3()=0;

  // debug
  virtual void DebugOutput(TUint aChar)=0;

  // power management
  virtual void Idle()=0;

  // timing
  virtual TInt MsTickPeriod()=0;
  virtual TInt SystemTimeInSecondsFrom2000(TInt& aTime)=0;
  virtual TInt SetSystemTimeInSecondsFrom2000(Tint aTime)=0;
  virtual TUint32 NanoWaitCalibration()=0;

  // HAL
  virtual TInt VariantHal(TInt aFunction, TAny* a1, TAny* a2)=0;
```

```
  // Machine configuration
  virtual TPtr8 MachineConfiguration()=0;
  };
```

The variant provides other interfaces that are available for use by device drivers and extensions. A key example is the Interrupt class provided in assp.h:

```
class Interrupt
  {
public:
  IMPORT_C static TInt Bind(TInt aId, TIsr aIsr, TAny* aPtr);
  IMPORT_C static TInt Unbind(TInt aId);
  IMPORT_C static TInt Enable(TInt aId);
  IMPORT_C static TInt Disable(TInt aId);
  IMPORT_C static TInt Clear(TInt aId);
  IMPORT_C static TInt SetPriority(TInt aId, TInt aPriority);
  };
```

The variant performs interrupt dispatch for the system; the methods in the Interrupt class allow device drivers and extensions to install their own interrupt handlers.

The CPU layer can also provide hardware-specific implementations of HAL functions, although these may equally be implemented in the kernel itself or in an extension.

Timers

Timers are both a fundamental need for the functioning of EKA2, and a service that EKA2 provides to its users. In this section, I will discuss the detailed operation of nanokernel and Symbian OS timers.

Nanokernel timers

Earlier in this chapter, I said that nanokernel timers, `NTimer`, provide the most fundamental system timing functions in the operating system. Let's look now at how they are implemented.

The main requirements for `NTimer` are:

- Timers can be started and stopped from any kernel code - ISRs, IDFCs or threads, so the timer start and stop functions should have small deterministic execution times
- It should be possible to generate periodic timers with no drift due to delays in servicing the timer
- It should be possible to disable the timer tick if the CPU is expected to be idle for several ticks without affecting the accuracy of the timed intervals, to minimize system power consumption.

The timer queue uses 67 separate doubly linked lists. Of these, the 32 pairs of final queues hold timers that are due to expire within the next 32 ticks. Of the other three, one is used to support timers whose handlers are called back in a DFC (the completed queue) and the other two (the holding queue and the ordered queue) hold timers which are due to expire more than 32 ticks in the future.

The timer queue contains a tick count, which is incremented on every tick interrupt. The tick count modulo 32 determines which of the 32 pairs of linked lists is checked on that tick. One list of the pair holds timers that require the handler to be called at the end of the tick ISR itself, and the other holds timers that require the handler to be called from a DFC following the tick interrupt. This second list, if non-empty, is appended to the end of the completed queue and the timer DFC is queued to process the callbacks. A 32-bit mask is also maintained - this corresponds to the 32 pairs of final queues, with one bit representing each pair. A bit is set if either of the corresponding pair of final queues has an entry.

If a timer is queued for a time less than 33 ticks in the future, the kernel just places that timer on the respective final queue. Timers that are queued for more than 32 ticks in the future are placed on the holding queue in FIFO order. Every 16 ticks, the tick interrupt service routine checks the holding queue, and if it is not empty, queues the timer DFC. This transfers any timers on the holding queue that are now due to expire in less than 33 ticks to their respective final queue. It transfers timers that still expire in more than 32 ticks to the ordered queue. As its name implies, entries on this queue always appear in increasing order of expiry time.

The timer DFC also drains the ordered queue. Every 16 ticks the interrupt service routine checks the ordered queue; if this is non-empty and the first entry expires in less than 33 ticks, then the ISR queues a DFC. The DFC will then walk the ordered queue, transferring entries to the final queues, until it reaches the end of the ordered queue or reaches an entry that expires in more than 32 ticks.

The kernel uses the ordered queue, in combination with the bit mask for the final queues and the holding queue, to determine the number of ticks until the next timer queue operation. In fact, this would generally be done in the null (idle) thread, just before it puts the CPU into idle mode. The null thread can then disable the timer tick for that number of ticks, allowing the CPU to sleep undisturbed for longer, and possibly allowing a lower-power sleep mode to be used. The bit mask for final queues is used to determine the number of ticks before the next final queue expiry. If the holding queue is non-empty, the number of ticks before the sort operation is calculated from the tick number - the sort operation is triggered if the tick count is zero modulo 16. If the ordered queue is non-empty, the time at which transfer of the first entry (that is, the one that expires first) to the relevant final queue would occur is calculated. The minimum of these three time values gives the number of ticks that can be skipped. It can be seen that this calculation has a small, predictable execution time, which is just as well since it will be done with interrupts disabled.

To be able to cancel timers, we need to keep track of which queue a timer is on. Each timer has a state that gives this information, and the following states are defined:

State	Description
Idle	The timer is not linked into any queue and is not currently set to expire. However the expiry handler may actually be running. No action is required to cancel a timer in this state
Holding	The timer is linked into the holding queue. To cancel a timer in this state, simply remove it from the holding queue
'Transferring	The timer is in transit from the holding queue to the ordered queue. It is not actually linked into either. To cancel a timer in this state, no dequeuing is needed, but a flag must be set to notify the timer DFC that the timer currently being transferred has been canceled. The timer DFC will then abort the transfer
Ordered	The timer is linked into the ordered queue. To cancel a timer in this state, simply remove it from the ordered queue
Critical	The timer is linked into the ordered queue and is currently being inspected by the timer DFC while transferring another timer from the holding queue to its correct position on the ordered queue. To cancel a timer in this state it is removed from the ordered queue and a flag is also set to notify the timer DFC that the current critical timer has been canceled. The timer DFC will then restart the sort operation
Final	The timer is linked into the final queue corresponding to its expiry time. To cancel a timer in this state, first remove it from the queue, then check the two final queues corresponding to the expiry time of the timer being canceled; if both are now empty, clear the corresponding bit in the `iPresent` bit mask.

Timers for less than 32 ticks in the future will simply transition from Idle to Final, whereas timers for longer periods will generally transition through all these states.

When a timer expires, we set its state back to Idle just before calling the timer handler. This means that care needs to be taken when canceling a timer whose expiry handler runs in the timer DFC. If the thread calling `Cancel()` has a priority above the timer DFC thread or `Cancel()` is called from an ISR or IDFC then `Cancel()` may occur during the execution of the timer handler. Since the state has been set back to Idle, the cancel will not do anything. If the memory containing the timer control block is now freed and reassigned to something else, contention may occur with the expiry handler. This is not usually a problem since threads of such high priority will not usually delete objects. It would, however, be a problem on an SMP system since the canceling thread could overlap the handler even if it had a lower priority.

We provide two functions to start a nanokernel timer:

OneShot(aTime, aDfc)

This sets a timer for aTime ticks from now. If aDfc is TRUE, the callback occurs in the context of the timer DFC, otherwise it occurs in the timer ISR.

Again(aTime)

This sets a timer for aTime ticks from its last expiry time. This is used to implement periodic timers that are immune to delays in processing the timer callbacks. The callback occurs in the same context as the previous one.

Summary of nanokernel timer control block:

Field	Description
`iNext, iPrev`	Link pointers for linking the timer into timer queues.
`iPtr`	Argument passed to callback function when timer completes.
`iFunction`	Pointer to timer expiry handler function.
`iTriggerTime`	Number of the tick at which timer is due to expire.
`iCompleteInDfc`	Boolean flag - TRUE means run timer expiry handler in DFC, FALSE means run it in ISR.
`iState`	Indicates which queue the timer is currently linked into, if any, and whether the timer is currently being moved.

Summary of nanokernel timer queue control block:

Field	Description
`iTickQ[32]`	32 pairs of linked lists, one pair corresponding to each of the next 32 ticks. One of the pair holds timers to be completed in the tick ISR and the other holds timers to be completed in the timer DFC.
`iPresent`	Bit mask corresponding to `iTickQ[32]`. Bit n is set if and only if `iTickQ[n]` is non-empty - that is at least one of the two linked lists is non-empty.
`iMsCount`	The number of the next tick.
`iHoldingQ`	Queue of timers that expire more than 32 ticks in the future, ordered by time at which timers were queued.
`iOrderedQ`	Queue of timers that expire more than 32 ticks in the future, ordered by expiry time.
`iCompletedQ`	Queue of timers that have expired and are waiting to have their handlers called back in the timer DFC.
`iDfc`	DFC used to transfer timers between queues and to call back handlers for timers requiring DFC callback.
`iTransferringCancelled`	Boolean flag set if the timer that is currently being transferred from `iHoldingQ` to `iOrderedQ` is canceled. Cleared when a new timer is removed from `iHoldingQ` for transfer to `iOrderedQ`.
`iCriticalCancelled`	Boolean flag set if the timer on the ordered queue that is currently being inspected during a sort is canceled. Cleared when the sort steps on to a new timer.
`iDebugFn`	Only used for testing/debugging.
`iDebugPtr`	Only used for testing/debugging.
`iTickPeriod`	The period of the nanokernel timer tick in microseconds.

`iRounding`	Spare entry for use by the ASSP/variant code involved in generating the tick interrupt.

Figure 5.4 gives an approximate overview of the nanokernel timer and shows how it fits with the Symbian OS tick timer, which I will cover in the next section. To the left of the figure, you can see the control block of the nanokernel timer, which has pointers to the final timer queues (timers due to expire in less than 32 ticks) and pointers to the holding queue and ordered queue, for timers that are further in the future. You can also see how Symbian OS tick timers interact with nanokernel timers - the head of a doubly linked list of *SymbianTimers* (actually `TTickLink`) objects is used to schedule a nanokernel timer for the next tick timer event.

The figure also shows how DFCs drain the nanokernel timer queues, with some of the callbacks being used to schedule Symbian OS tick timers - which I'll discuss next.

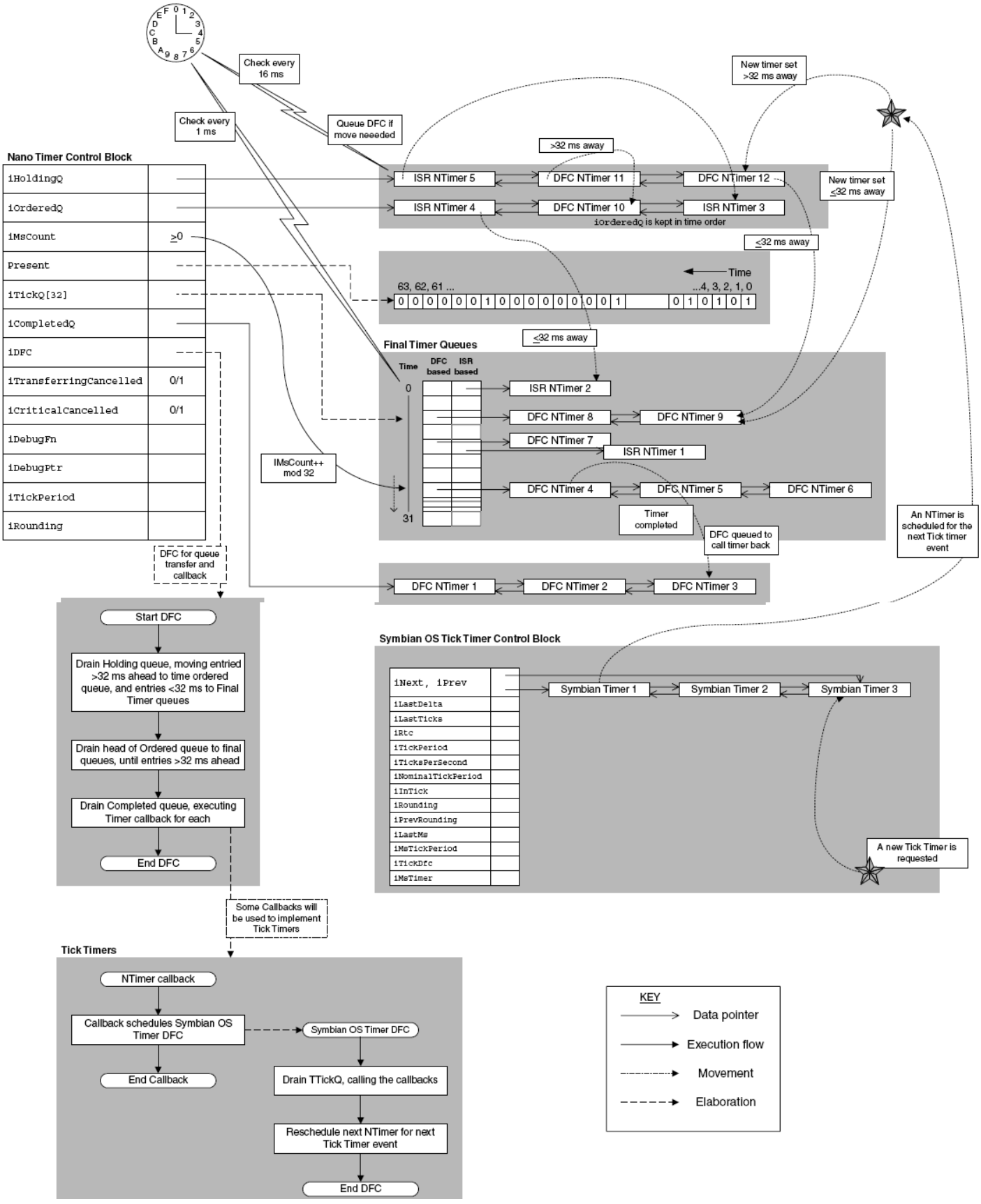

Figure 5.4 *Nanokernel timers*

Symbian OS tick timers

Tick timers are general-purpose interval timers that are used where there is no need for high resolution or great accuracy. They correspond to the timing functions available to user-side code on EKA1. We represent a tick timer using a `TTickLink` object, which I will describe in detail later in this section. We keep active `TTickLink` objects on a doubly linked queue in order of expiry time. This queue is contained in a single `TTickQ` object instance, along with other global state related to Symbian OS timers. Adding `TTickLinks` to this queue is an O(N) operation so we use a mutex, `TTickQ::Mutex`, to protect the queue.

The tick queue operates according to a notional *Symbian OS nominal tick* which has a period of 15.625 ms (64 Hz frequency) - that is, the behavior is the same as that which would be obtained by using a 64 Hz periodic tick interrupt. In fact there is no such periodic tick - instead a single `NTimer` is used to trigger expiry of `TTickLink` timers. It is always set to expire either when the first `TTickLink` should expire or in 65536 Symbian OS ticks (1024 seconds), whichever is the sooner. The limit is imposed to ensure that differences calculated in microseconds never overflow a 32-bit signed quantity. When the `NTimer` expires, it activates a DFC that runs in the Symbian OS timer thread (TimerThread). This DFC dequeues and calls handlers for any `TTickLink` timers that have expired, and then requeues the `NTimer` for the next `TTickLink` timer expiry. The timer mutex is acquired at the beginning of the DFC and released at the end, so the mutex is held while the expiry handlers are called.

Under this system, adding a new `TTickLink` to the queue will in general mean that we need to requeue the `NTimer` if the new timer expires earlier than the previous earliest timer. The exception to this rule is if the `TTickLink` is added from another `TTickLink` expiry handler; in this case the main DFC routine will requeue the `NTimer` after all `TTickLink` expiry handlers for this tick have been called. We use the `TTickQ::iInTick` flag to indicate that the DFC is in progress; it is set by the DFC after acquiring the timer mutex, so the code to add a `TTickLink` (which also runs with the timer mutex held) will see it set if and only if called from the tick DFC itself.

We've seen that the usual `NTimer` resolution is 1 ms, which means that a period of 15.625 ms cannot be generated exactly. And, of course, the `NTimer` resolution may not be 1 ms for manufacturer/device-specific reasons. Hence the `TTickLink` timer queue uses a *pulse swallowing* type algorithm - it sets up the `NTimer` to generate intervals that are a multiple of 1 ms, such that the average period of the Symbian OS tick is 15.625 ms. For example, if a periodic `TTickLink` were active with a period of 1 nominal Symbian OS tick, the `NTimer` would actually trigger at either 15 ms or 16 ms intervals with five out of every eight intervals being 16 ms and the other three out of eight being 15 ms. This works by calculating the required `NTimer` interval in microseconds and accumulating the error incurred in rounding to the period of `NTimer`. The error is taken into account on the next calculation. In addition, we use the zero-drift mode of `NTimer`, where the interval is timed relative to the last timer expiry. In fact the Symbian OS timers are all calculated relative to the last such expiry. A count of nominal ticks is maintained to support the `User::TickCount()` function and a similar count is maintained to serve as universal time. These counts are updated at the beginning of the DFC that services the `TTickQ`. The nanokernel tick count at which the `NTimer` triggered is saved and the tick count and RTC count are incremented by the number of nominal ticks elapsed between this and the previous `NTimer` expiry. To obtain the current universal time in microseconds since 00:00:0001-01-0AD (standard Symbian OS time storage format), we use the following formula:

```
iRtc * iNominalTickPeriod + (NTickCount()-iLastMs-1) * NTimer period
```

where NTickCount() is the current `NTimer` tick count and the other fields are defined in the tables below. The extra -1 in the second term is due to the fact that `NTickCount()` is incremented immediately after determining which timers to complete. This scheme allows the system time to be obtained to a 1 ms resolution (or whatever the resolution of `NTimer` is on a particular platform).

Summary of fields in `TTickLink`:

Field	Description
`iNext`, `iPrev`	Link pointers used to attach this object to the system tick timer queue (TTickQ).
`iDelta`	Number of OS ticks between the expiry of this timer and the expiry of the following one (pointed to by `iNext`). Never negative, but could be zero for timers expiring at the same time.
`iPeriod`	Period of this timer in OS ticks or zero for a one-shot timer.
`iPtr`	Argument passed to callback function when this timer expires.
`iCallBack`	Pointer to function to be called when this timer expires.
`iLastLock`	If this timer is being used to implement a Symbian OS *locked* timer, this holds the value of `TTickQ::iRtc` at the last expiry of this timer. If this timer is not being used for a locked timer or has not yet expired, this value is -1.

Summary of fields in `TTickQ`:

Field	Description
`iNext, iPrev`	Link pointers used to point to first and last entries on a time-ordered queue of `TTickLink` objects.
`iLastDelta`	Number of OS ticks which elapse between the last tick timer expiry and the time when `iMsTimer` next triggers - used to increment `iLastTicks` and `iRtc`.
`iLastTicks`	OS tick count at point when `iMsTimer` last triggered.
`iRtc`	The absolute time at the point when `iMsTimer` last triggered, measured in nominal OS ticks from 00:00:00 1st January 0AD.
`iTickPeriod`	The current actual length of an OS tick in microseconds. This may differ from the nominal tick period if a tracking system is being used to make the `iRtc` value follow a hardware RTC. This value may change as a result of the operation of any such tracking system.
`iTicksPerSecond`	Number of nominal OS ticks in one second of elapsed time.
`iNominalTickPeriod`	The nominal length of an OS tick in microseconds. This value is never changed, unlike `iTickPeriod`.

iInTick	Boolean flag set to indicate that processing of the tick queue initiated by iMsTimer expiry is underway.
iRounding	The number of microseconds added to the last delta value when iMsTimer was last set up in order to make the period an integral number of nanokernel timer ticks.
iPrevRounding	The value of iRounding at the point where iMsTimer last triggered. Each time the timer is queued, iPrevRounding is used in the calculation of when the timer should trigger and the rounding applied to that time to obtain an integral number of nanokernel ticks is stored in iRounding.
iLastMs	The nanokernel tick count at which iMsTimer last triggered.
iMsTickPeriod	The period of the nanokernel tick in microseconds.
iTickDfc	DFC queued by the expiry of iMsTimer. Runs in context of TimerThread and processes any Symbian OS timers which have just expired.
iMsTimer	Nanokernel timer used to initiate Symbian OS timer processing. It is always queued to trigger at the time when the next TtickLink timer should expire.

Second timers

Second timers are used when an event needs to occur at a specific date and time of day rather than after a specified interval, and are typically used for system alarms. They have a resolution of 1 second. They will also power up the system at the expiry time if they need to.

We represent a second timer by a TSecondLink object and attach active timers to a TSecondQ absolute timer queue object, of which a single instance exists. Each TSecondLink stores the absolute time at which it should trigger (measured in nominal OS ticks from 00:00:00 1st January 0AD UTC) and they are linked into the queue in chronological order of expiry time, earliest first. The second timer queue is driven from the tick timer queue. It contains a TTickLink timer which is set to expire at either the trigger time of the first TSecondLink on the queue or at the next midnight local time, whichever is the earlier. When this TTickLink timer triggers, it calls back the handlers for TSecondLink timers that have expired, and then requeues the TTickLink timer. The same mutex (timer mutex) is used to protect the TTickQ and TSecondQ objects, and the handlers are called with the timer mutex held. We use the expiry at midnight to signal change notifiers that midnight crossover has occurred. In a similar way to TTickQ, when a new TSecondLink is queued, the TTickLink timer may need to be canceled and requeued, unless it is queued from inside the TSecondQ expiry handler. Again we use an iInTick field to indicate the latter condition.

Summary of fields in TSecondLink:

Field	Description
iNext, iPrev	Link pointers used to attach this object to the system absolute timer queue (TSecondQ).
iTime	The absolute time when this timer should trigger, measured in nominal OS ticks from 00:00:00 1st January 0AD.
iPtr	Argument passed to callback function when this timer expires.
iCallBack	Pointer to function to be called when this timer expires.

Summary of fields in TSecondQ:

Field	Description
iNext, iPrev	Link pointers used to point to first and last entries in a time-ordered queue of TSecondLink objects.
iExpired	Boolean flag set when any TSecondLink timer expires and cleared by the power model just before initiating the machine power down sequence. Used by the power model to abort power down if an absolute timer expires during the power-down sequence.
iInTick	Boolean flag set to indicate that processing of the second timer queue initiated by iTimer expiry is underway.
iNextTrigger	The absolute time when iTimer will next trigger, measured in nominal OS ticks from 00:00:00 1st January 0AD.
iMidnight	The absolute time of the next midnight, measured in nominal OS ticks from 00:00:00 1st January 0AD.
iTicksPerDay	Number of nominal OS ticks in 1 day.
iTimer	TTickLink timer object used to initiate second queue timer processing. It is always queued to trigger either at the time when the next TSecondLink timer should expire or at the next midnight, whichever is earlier.
iWakeUpDfc	DFC used to restart the TTickQ and TSecondQ following machine power down and power up and changes to the system time.

Summary

In this chapter, I have described the wide variety of services that EKA2 provides, both to user-mode threads and within the kernel too. I have also described the basic objects used by the kernel, and the handle mechanism used to identify them.

A key part of the executive call was the SWI instruction, or software interrupt, used to switch the processor from user mode to supervisor mode.