Symbian OS Internals/10. The Loader

- Symbian OS Internals Table of Contents

by Dennis May and Peter Scobie

It said, Insert disk #3 *but only two will fit!*
Unknown

The file server process contains a second server, the loader server, whose purpose is to load executables (that is, DLLs and EXEs). It runs in a separate thread from the main file server and is implemented using the Symbian OS client-server framework.

In this chapter I describe the process of loading executables and the management of executable code in Symbian OS. In Section 10.3, I show how the loader server operates: how it handles client requests, how it interacts with the file server and I introduce some of the kernel-side code management services. In Section 10.4, I describe kernel-side code management in more detail.

However, before any of this, we should start by examining the format of the executable files that the loader has to handle.

E32 image file format

This is the Symbian OS executable file format. Symbian provides tools to convert from standard object file formats into the E32 image format: you would use the PETRAN pre-processing tool to convert PE format files (as output by GCC98r2 for example). Similarly, you would use ELFTRAN to convert from the ELF format files produced by ARM's RealView compiler.

You can configure these conversion tools to produce compressed executable files and it is the responsibility of the loader to detect a compressed executable and decompress it on loading.

E32 image files consist of up of nine sections, in the order specified as follows:

1) The image header The E32ImageHeader. I describe this in Appendix 2, The E32Image-Header.

2) Code section - .text This section contains the executable code.

3) The constant data section - .rdata This section contains constant (read-only) data. It doesn't exist as a separate entity in an E32 image file. It may exist in the PE or ELF format file from which the E32 image file is derived, but the tools then amalgamate it into the .text section.

4) The import address table (IAT) This table contains an entry for each function imported by the executable, as follows:

Offset	Description
00	Ordinal of import 1
04	Ordinal of import 2
. . .	
4(n - 1)	Ordinal of import n
4n	NULL

For each function that the executable imports, the file will contain an import stub function within the code section. When executed, each stub will load the program counter with the value from the corresponding entry of the IAT.

Therefore, when the executable is loaded, the loader has to fix up each entry in the IAT, replacing the ordinal number with the run address of the imported function. It has to calculate the address using the contents of the .idata section of this executable together with the .edata section of the DLL that exports the function requested.

Note that executables originating from ELF format files don't contain an IAT. For these, the tools fix up the import stubs directly - they obtain the location of the stub from the import data section.

The order of the entries in the IAT corresponds precisely with the order that imports are listed in the .idata section.

5) The export directory - .edata The export directory is a table supplying the address of each function exported from this executable. Each entry holds the start address of the function as an offset relative to the start of the code section:

00	Address of 1st function exported from this executable.
04	Address of 2nd function exported from this executable.
. . .	
4n - 4	Address of last function exported from this executable.

The order of exports in the table corresponds to the order in the DEF file for this executable. The table is not null terminated.

Instead, the number of entries in the table is available from the file's image header.

6) Initialized data section - .data This section contains the initialized data values that are copied to RAM when this executable runs.

7) Import data section - .idata This section contains data on each function that this executable imports. The loader uses this information to identify each referenced DLL that it also needs to load. Additionally, the loader uses this information to fix up each entry in the import address table (IAT).

The format of this section is as follows:

Field	Description
Size	A word holding the size of this section in bytes (rounded to 4-byte boundary).
Import block for DLL1	
Import block for DLL2	
. . .	. . .
Import block for DLLn	
Name of DDL1	NULL terminated ASCII string.
Name of DLL2	NULL terminated ASCII string.
. . .	. . .
Name of DLLn	NULL terminated ASCII string.

As well as the file name itself, the DLL name string also includes the required third UID. If the file was built with the EKA2 tool set, the name string will also contain the required version number of the DLL (see Section 10.3.1). The loader will have to match all of these when it searches for the imported DLL. The format of the name is as follows, with UID and version in hexadecimal:

```
<filename>{versionNum}[uid3]<extension>
for example, efsrv{00010000}[100039e4].dll
```

The format of each import block depends on whether the executable originated from a PE or an ELF format file. For PE derived files, it has the following format:

Offset	Description
00H	Offset of DLL name from start of .idata section.
04H	Number of imports for this DLL.

For PE-derived executables built with the EKA1 tool set, this import block also contains a list of ordinals for the DLL concerned. However, this information is a duplicate of that contained in the import address table, so import blocks no longer contain this information when built with the EKA2 tool set.

Import blocks for ELF derived files have the following format:

Offset	Description
00H	Offset of DLL name from start of .idata section.
04H	Number of imports for this DLL.
08H	The location of import stub for the 1st function imported from this DLL. (This is an offset within the code segment of the importing executable to the location that will be fixed up with the address of the imported function.)
0CH	The location of import stub for the 2nd function imported from this DLL.
. . .	. . .
	The location of import stub for the last function imported from this DLL.

8) Code relocation section This section contains the relocations to be applied to the code section. The format of the table is shown next:

Offset	Description
00H	The size of the relocation section in bytes (rounded to 4-byte boundary).
04H	Number of relocations.
08H	Relocation information.
0CH	Relocation information.
. . .	. . .
Nn	00000H

The format used for the relocation information block differs slightly to the standard Microsoft PE format. It consists of a number of sub-blocks, each referring to a 4 KB page within the section concerned. Each sub-block is always a multiple of 4 bytes in size and has the following format:

Offset	Description
00H	The offset of the start of the 4 KB page relative to the section being relocated.
04H	The size of this sub-block in bytes.
08H	2 byte sub-block entry. The top 4 bits specify the type of relocation: 0 - Not a valid relocation. 1 - Relocate relative to code section. 2 - Relocate relative to data section. 3 - Try to work it out at load time (legacy algorithm). The bottom 12 bits specify the offset within the 4 K page of the item to be relocated.
0AH	2 byte sub-block entry.
. . .	. . .

9) Data relocation section This section contains the relocations to be applied to the data section.

The format of the table is the same as that for code relocations.

The nine sections that I have just listed apply to the structure of the executable files as they are stored on disk. However, once an executable has been loaded by the Symbian OS loader, it consists of two separately relocatable regions (or sections). The first is the code section, which includes the code, import stubs, constant data, IAT (if present) and the export directory (.edata). The second is the data section, which includes the un-initialized data (.bss) and initialized data (.data).

ROM image file format

The format that I described previously applies to executables that are located on drives which are not execute-in-place (XIP) - for example, executables contained within a NAND Flash ROFS image, on the user-data drive (C:) or a removable media disk (D:). On such media, executable code first has to be loaded into RAM (at an address which is not fixed beforehand) before it can be run. However, by definition, for executables stored in XIP memory (such as ROM) there is no need to load the code into RAM for execution.

The ROMBUILD tool assembles the executables and data files destined for the ROM into a ROM image. The base address of the ROM and the address of the data sections are supplied to this tool as part of the ROM specification (that is, the obey file) when it is run. In fact, for certain executables, ROMBUILD itself calculates the address of the data sections. These include fixed processes, variant DLLs, kernel extensions, device drivers and user-side DLLs with writeable static data. With this information, ROMBUILD is able to pre-process the executables, perform the relocations and fix up the import stubs. The result is that on XIP memory, executables have a format that is based on E32 image format but differs in certain ways, which I will now list:

- They have a different file header, `TRomImageHeader`, which is described in [[Symbian OS Internals/Appendix 3: The `TRomImageHeader` 3]], The `TRomImageHeader`
- They have no IAT; it is removed and each reference to an IAT entry is converted into a reference to the associated export directory entry in the corresponding DLL
- They have no import data (.idata) section; it is discarded
- They have no relocation information; it is discarded
- They include a DLL reference table after the .data section. This is a list of any libraries referenced, directly or indirectly, by the executable that have static data. In other words, libraries with initialized data (.data) or un-initialized data (.bss) sections. The file header contains a pointer to the start of this table. For each such DLL referenced in the table, the table holds a fixed-up pointer to the image header of the DLL concerned. See the following table:

Offset	Description
00H	Flags.
02H	Number of entries in table.
04H	Image header of 1st DLL referenced.
08H	Image header of 2nd DLL referenced.
. . .	. . .
nn	Image header of last DLL referenced.

These differences mean that the size of these files on ROM is smaller than the corresponding E32 image file size.

Another consequence of the pre-processing of the IAT and the removal of the import section for ROM files is that it is not possible to over-ride a statically linked DLL located in ROM by placing a different version of this referenced DLL on a drive checked earlier in the drive search order, such as C:.

The loader server

The `RLoader` class provides the client interface to the loader and is contained in the user library, EUSER.DLL. However, user programs have no need to use this class directly - and indeed they must not use it since it is classified as an internal interface. Instead the `RLoader` class is used privately by various file server and user library methods. These include:

- `RProcess::Create()` - starting a new process
- `RLibrary::Load()` - loading a DLL
- `User::LoadLogicalDevice()` - loading an LDD
- `RFs::AddFileSystem()` - adding a file system.

RLoader is derived from `RSessionBase` and is a handle on a session with the loader server. Each request is converted into a message that is sent to the loader server.

Unlike the file server, the loader server supports only a small number of services. These include:

- Starting a process - loading the executable concerned Loading a DLL
- Getting information about a particular DLL. (This includes information on the DLL's UID set, security capabilities and module version)
- Loading a device driver
- Loading a locale
- Loading a file system or file server extension.

Version numbering

In EKA1, it is only possible for one version of a particular executable to exist in a Symbian OS phone at any time. This is not the case for EKA2, whose loader and build tools support version numbering - a scheme that associates a version number with each executable. In this scheme, import references state the version number of each dependency. This makes it possible for us to make changes to an executable that are not binary compatible with the previous version, since we can now place both the new and old versions on the device simultaneously. The import section of any preexisting binary that has a dependency on this executable will indicate that it was built against the old version, allowing the loader to link it to that version. At the same time, the loader links new or re-built binaries to the new version. In some cases, rather than have two entire versions of the same DLL, the old version can be re-implemented as a shim DLL, using the functionality in the new DLL, but presenting the original DLL interface.

The EKA2 tools tag each executable with a 32-bit version number, which is stored in the image file header (see Appendix 2, The E32ImageHeader and [[Symbian OS Internals/Appendix 3: The `TRomImageHeader` 3]], The `TRomImageHeader`, for details). This number is made up of a 16-bit major and a 16-bit minor number. Regarding linkage, each entry in an executable's import table now specifies the required version number of the DLL concerned (see Section 10.1 for details). Where two or more versions of an executable exist on a device, both will generally reside in the same directory to save the loader searching additional directories. (Indeed, if platform security is enabled, they must reside in the same restricted system directory.) To prevent file name clashes, older versions have the version number appended to the file name (for example, `efsrv{00010000}.dll`), whereas the latest version has an unadorned name (for example, efsrv.dll). When searching for a candidate executable to load, the loader ignores any version number in the file name - but subsequently checks the version number in the header.

The EKA2 tools tag an executable with a default version number of either 1.0 (for GCC98r2 binaries) or 10.0 (for binaries built with an EABI compliant compiler). The same applies for the default version specified in each element of the import table. This allows inter-working with binaries built with pre-EKA2 versions of the tools, which are assumed to have the version 0.0. (Executables which pre-date versioning can only ever be introduced on non-XIP media. This is because all ROM resident binaries in EKA2-based systems will be built with the new tools.)

We assign a new version number to a DLL each time its published API is changed. If the change is backward compatible (for example, just adding new APIs) and all executables that worked with the original will continue to work with the new version, then we only increment the minor number.

When the new version removes or breaks an existing API, then we increment the major number and reset the minor number to zero. We assign modified APIs a new ordinal number, and remove the original ordinal (leaving a hole). This means that, whether an API is removed or modified, it appears that it has been removed. Of course, it will generally be the case that we break

compatibility for just a small number of APIs and the majority of APIs will remain compatible. Executables that don't use removed APIs can then continue to run successfully against the new version. So, whenever APIs are removed, we include information in the image header to indicate which exports are affected (see Appendix 2, The *E32ImageHeader*, for more details).

When it is loading an executable and resolving import dependencies, if the loader finds more than one DLL in the search path that matches the requested name, UID and security capabilities, but has differing version numbers, then it employs the following selection algorithm:

1. If there is a DLL in this set with the requested major version number and a minor version number greater than or equal to the requested minor version number, then it uses that one. If there is more than one of these, then it uses the one with the highest minor version number
2. If no DLL exists satisfying (1), the loader looks for a DLL with a higher major version number than the one requested. If there is more than one of these, then it selects the one with the lowest major version number and the highest minor version number. If the executable does not request any exports that no longer exist in this DLL, then the loader uses it
3. If no DLL exists satisfying (1) or (2), the loader looks for a DLL with the requested major version number. If there is more than one of these, it finds the one with the highest minor version number. If the executable currently being loaded does not request any exports that are not present in this DLL, then the loader uses it
4. If no DLL exists satisfying either (1), (2) or (3) then the load fails.

An implication of the previous algorithm is that as the loader searches for an executable across multiple drives (and multiple paths in non-secure mode), it can't afford to stop at the first match. Instead it has to continue to the end of the search and then evaluate the best match. Fortunately, platform security cuts down the number of paths which have to be searched. The loader cache, which I describe in Section 10.3.3, also reduces the impact of this searching.

Searching for an executable

The EKA2 version of F32 is normally built with platform security enabled - see Chapter 8, *Platform Security*, for more on this. In this secure version, the loader will only load executables from the restricted system area, which is located in the *\sys \bin* directory of a given drive.

In non-secure mode, the loader will load executables from any directory. However, there are a default set of directories where executables are generally located and the loader scans these directories when searching for an executable to load.

Search rules on loading a process

When a client calls `RProcess::Create()` to start a new process, it specifies the filename of the executable (and optionally the UID type). If the filename includes the drive and path, then the task of locating the executable is straightforward. When either of these is not supplied, the loader has a fixed set of locations that it searches. This set is much more limited when platform security is enabled. The search rules that it uses are as follows:

1. If the filename includes a path but no drive letter, then the loader searches only that path, but it does this for all 26 drives
2. If the filename doesn't contain a path, then instead the loader searches each of the paths that I will now list, in the order given. (Again, for each of these, if a drive letter was supplied then the loader searches these paths only on this specified drive. However, if no drive was supplied either, then it checks all listed paths on all 26 drives)* sys\bin

- system\bin (non-secure mode only)
- system\programs (non-secure mode only)
- system\libs (non-secure mode only).

When searching all 26 drives, the search order starts with drive Y: and then works backwards through to drive A:, followed finally by the Z: drive. Searching the Z: drive last makes it possible for us to over-ride a particular EXE in the ROM drive by replacing it with an updated version on an alternative drive that is checked earlier in the search order, for example, the C: drive.

When loading a process, the loader follows these rules to select which executable to load:

- Check that the filename and extension match
- Check that the UID type matches (if specified)
- Out of all possible matches, select the one with the highest version number. (Remember, the selection algorithm I described in Section 10.3.1 applies only when resolving import dependencies.)

Once the process executable is loaded, the loader goes on to resolve all its import dependencies. For a non-XIP executable, the name of each DLL that it statically links to is contained in the import data section of the image. The module version and third UID are included in the DLL name (as I discussed in Section 10.1) - but the path and drive are not. Therefore the loader again has to search a fixed set of locations for each dependency:

1. The drive and path that the process executable was loaded from

2. All of the paths listed, in the order given, on all 26 drives in turn:* sys\bin

- system\bin (non-secure mode only)
- system\libs (non-secure mode only).

When loading DLL dependencies, the properties that the loader checks when searching for a match are more substantial:

- Check that the filename and extension match
- Check that the third UIDs match
- Check that the candidate DLL has sufficient platform security capabilities compared with the importing executable. Refer to Section 8.4.2.1 for precise details of this capability check
- Check that the module versions are compatible. This could potentially include checking the export table bitmap in the image header of a candidate DLL to ensure that an export hasn't been removed.

As I mentioned in Section 10.3.1, when resolving import dependencies, the version numbering selection scheme means that the loader must continue to the end of the search and evaluate the best match.

Search rules when loading a library

When a client calls `RLibrary::Load()` to dynamically load a library, it may provide a path list as well as the filename of the DLL. Again, if the filename includes the drive and path, the loader loads the DLL from that location. However, if it does not, then the loader searches each of the paths specified in the path list before searching the standard paths for DLL loading, which I listed in the previous paragraph. Again, once the DLL is loaded, the loader goes on to resolve its import dependencies in the same way as I described previously.

The loader cache

From the previous section, you can see that the loading of an executable that has dependencies can involve the loader searching many directories on multiple drives. Furthermore, to determine if a candidate executable fully matches the criteria required by the client or importing executable, then the loader must read header information from those files that match the required filename.

To read directory entries and file headers, the loader server needs to makes requests on the file server and so it permanently has a file server session open. But to optimize the speed at which executables are loaded, and to reduce the number of requests made to the file server, we implemented a loader cache.

For a small number of directories, the loader caches the names of every file that they contain. It stores only the files' *rootnames* in ASCII. The *rootname* is the basic name and extension with any version or UID information removed. We use ASCII since names of executables do not include Unicode characters. (As I showed in Section 10.1, the import section of the image file specifies the names of implicitly linked libraries as 8-bit strings, so an executable with a Unicode name cannot be accommodated within the E32 image format.)

When the loader searches for an executable, if it gets a possible *hit* on a cached root name entry, it further populates the entry (if it hasn't already done so) opening the file and reading the image header to extract the following file data:

- UID triplet
- Module version
- Security information
- Export bitmap - for V-formatted headers (see Appendix 2, *The E32ImageHeader*)

The cache can contain file entries for up to six different directories. It maintains all the file entries for these directories for every mounted drive on the system. It also stores the path to each directory in a linked list.

The class diagram for the loader cache is shown in Figure 10.1. We encapsulate each cached directory by a `TDirectoryCacheHeader` object. This has an array of pointers, `iCache,` to all its cached file entries, each of which is represented by an instance of a `TFileCacheRecord` class, holding the rootname of the file and other cached file data. The pointers are sorted by name, allowing us to perform a binary search for a given rootname.

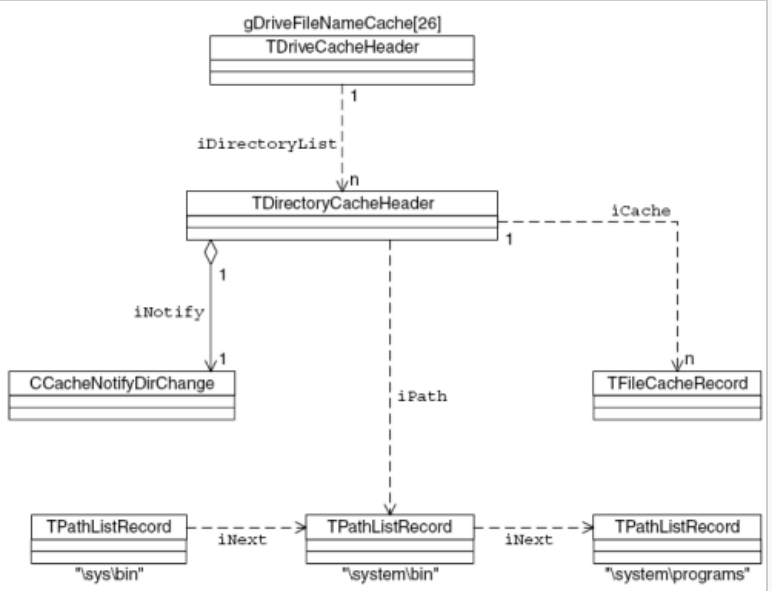

Figure 10.1 *Class diagram for the loader cache*

We represent each of the 26 drives by a `TDriveCacheHeader` object; this has a list: `iDirectoryList` of all the current cached directories on it.

We encapsulate the path name of every cached directory by a `TPathListRecord` object; there is a linked list of up to six of these. In fact, the loader always caches the default directories that it uses. So, in non-secure mode, it always caches these directories: sys\bin, system\bin, system\programs and system\libs. In secure mode, it always caches the directory sys\bin. Because of this, the loader only recycles the remainder of the six entries between other directories. Because the loader tends to search the same paths on each drive, for each directory path there will generally be a corresponding directory cache on each of the active drives.

When performing a search, the loader quickly iterates through the path list, checking whether the directory in question is already cached. Assuming it is, the loader navigates to the corresponding `TDirectoryCacheHeader` object via the appropriate drive object, where it can then scan through each of the file entries.

The cache must accurately reflect the directory filenames for those directories that it holds. Any changes to these must trigger a refresh before the next query results are returned. The loader uses file server notifications for this purpose - one for each drive\path. The `CCacheNotifyDirChange` class handles change notification. Each time a notification is triggered, the loader destroys that directory cache for the corresponding drive. This has the effect of forcing a cache reload for that directory when the drive\path is next queried.

Code and data section management

On Symbian OS, the low level abstraction describing a segment of code is the `DCodeSeg` class, a kernel-side object that I will describe in detail in Section 10.4.1. This class represents the contents of an executable that has been relocated for particular code and data addresses. Each `DCodeSeg` object has an array of pointers to the other code segments to which it links. In this section I will concentrate on how the loader uses `DCodeSeg` objects.

The code segment object for a non-XIP image file owns a region of RAM, and it is into this that the loader copies the code and data prior to execution. The kernel allocates space in this RAM region for the code section, the constant data section, the IAT (if present), the export directory and the initialized data section, in that order. The loader applies code and data relocations to this segment.

XIP image files, by definition, can have their code executed directly from the image file. Likewise, the initialized data section and export directory can be read directly. Because of this, there is no need for a RAM region and instead the `DCodeSeg` object just contains a pointer to the ROM image header for that file.

The loader needs to create new code segments and manage those that it has already created. For this purpose, it has a private set of executive functions, grouped with other loader services into the E32Loader class, which is contained in the user library. Only threads in the file server process may use these functions - the kernel panics any other thread trying to use them.

Once it has created a code segment for non-XIP media, the kernel returns the base addresses of the code area and the initialized data area within the segment to the loader. These are known as the code load and data load addresses respectively, and the loader uses them when copying in the various sections from the image file. The kernel also returns the code and data run addresses, and the loader uses this information when applying code and data relocations.

To conserve RAM, the kernel shares code segments whenever possible, and generally only one is required for each executable, even when this is shared between processes - I list the only exceptions later, in Section 10.4.1. So, for example, where a process has been loaded twice, generally the kernel will create only one main code segment for that process. Both processes have their own separate global data chunks, however. When each of these processes is first run, the contents of the initialized data section is copied out of the code segment into this global data chunk.

Loader classes

In Figure 10.2 I show the main classes that the loader server uses. The E32Image class encapsulates a single executable image. When starting a process or dynamically loading a DLL, the loader creates an instance of this class for the main executable and each statically linked DLL. (For non-XIP media, that includes every implicitly linked DLL, for XIP media, it only includes those which have static data.) The E32Image class has a pointer to a header object appropriate to the type of executable - either a `TRomImageHeader` object for XIP or an E32ImageHeader object for non-XIP media. The object denoted by `iHeader` holds a RAM copy of the entire E32ImageHeader, whereas `iRomImageHeader` points directly to the original header in ROM.

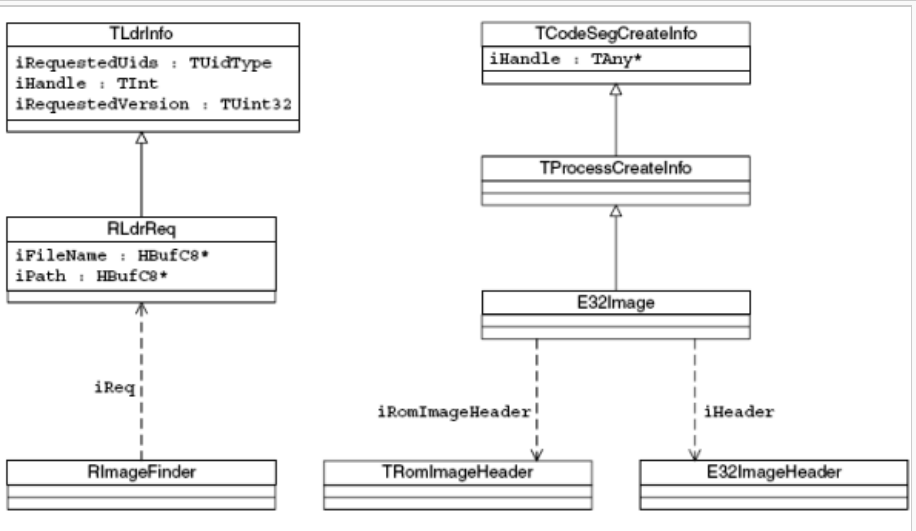

Figure 10.2 *Class diagram for the loader server*

The E32Image class is derived from the class `TProcessCreateInfo`, which adds information that is only required when creating a new process - such as stack size, heap size and so on. In turn, `TProcessCreateInfo` is derived from `TCodeSegCreateInfo`, which is used to assemble the data required to create a `DCodeSeg` object. The data member `iHandle` is a pointer to corresponding kernel-side `DCodeSeg` object and is used as a handle on it.

The `RLdrReq` class encapsulates a request on the loader. This is used to store request information copied from the client thread, such as the filename of the executable to load or a path list to search. It is derived from the `TLdrInfo` class, which is used to pass request arguments from client to server and, if a load request completes successfully, to return a handle on a newly loaded executable back to the client.

The loader uses the `RImageFinder` class when searching for a particular image file. To perform a search, this uses a reference to an `RLdrReq` object, `iReq`, which is the specification for the file being searched for.

Loading a process from ROM

Let us look at the steps involved in loading a process - first looking at the more straightforward case where this is being loaded from ROM.

Issuing the request to the server

The client calls the following user library method (or an overload of it):

```
TInt RProcess::Create(const TDesC &aFileName, const TDesC &aCommand,
const TUidType &aUidType, TOwnerType aType)
```

The argument `aFileName` specifies the filename of the executable to load. The descriptor `aCommand` may be used to pass a data argument to the new process's main thread function. The argument `aUidType` specifies a triplet of UIDs which the executable must match and `aType` defines the ownership of the process handle created (current process or current thread).

This method creates an `RLoadersession` object and calls its `Connect()` method, which results in a connect message being sent to the server. Once the connection is established, control returns to the client thread. This then calls `RLoader::LoadProcess()`, which assembles the arguments passed from the client into a message object and sends this to the server. The kernel then suspends the client thread until the loader completes the request.

On receipt of the message, the server calls the request servicing function belonging to the relevant session, `CSessionLoader::ServiceL()`.

This creates an `RLdrReq`, a loader request object, and populates it with data such as the filename, command descriptor and so on, reading these from the client thread. Next it allocates an E32Imageobject for the main process executable and calls the `LoadProcess()` method on it. Figure 10.3 illustrates this first phase.

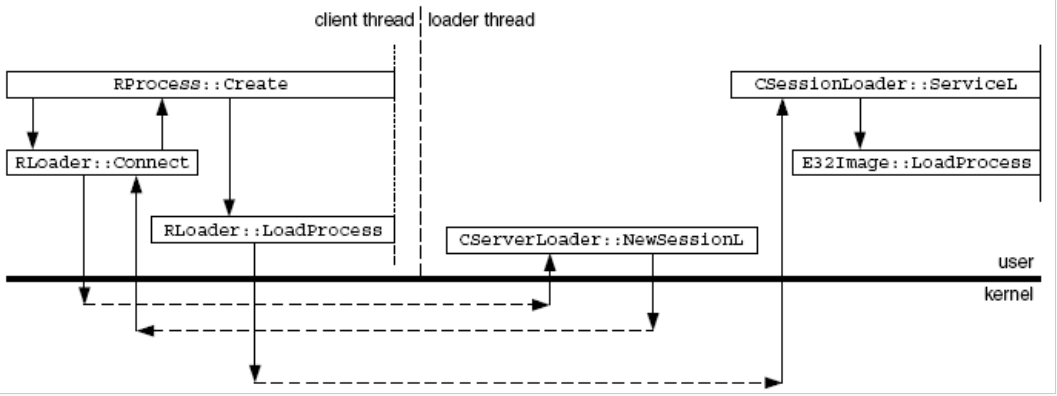

Figure 10.3 *Issuing the request to the loader server*

Locating the process executable

The next task to be performed is to search for the main executable file. The loader creates an `RImageFinder` object and sets its member `iReq` to the newly created request object, which contains the data specifying the executable. It calls the finder's `Search()` method - commencing the search scheme as outlined in the first part of Section 10.3.2.1. Assuming the search is successful, the loader now populates the main E32Image object with information on the executable file it has found, by calling the `E32Image::Construct()` method.

Figure 10.4 shows this second phase. Note that we assume here that the paths searched are not held in the loader cache, and so searching involves making requests on the file server to read the directory entries. The loader also needs to open candidate files to read information from their ROM image header, but this is not shown in the figure. Because the entire ROM image is always mapped into user-readable memory, the loader can access the ROM image headers via a pointer rather than having to use the file server to read them.

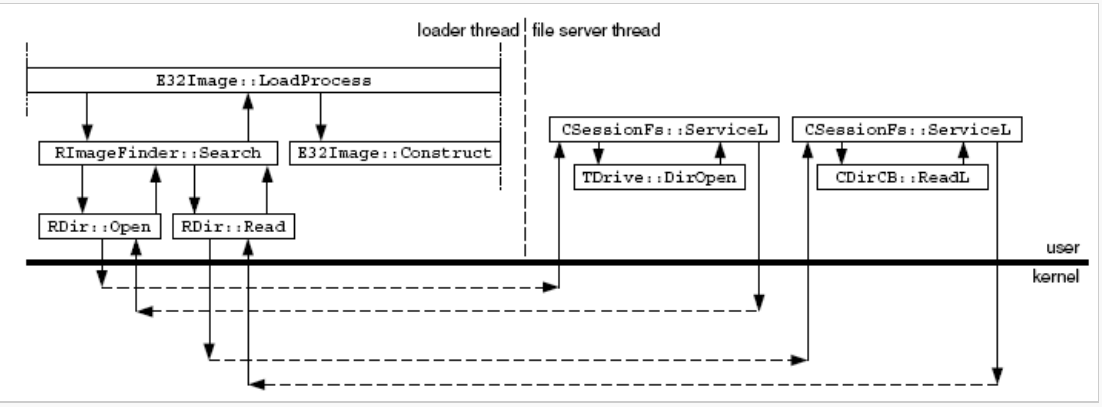

Figure 10.4 *Locating the process executable*

Creating the code segment and process

The next stage is to check whether a `DCodeSeg` object corresponding to this executable already exists (that is, the executable is already currently loaded). To do this, the loader uses the executive function `E32Loader::CodeSegNext()` to request that the kernel search its entire list of code segments looking for one with a ROM image header pointer that matches with the one about to be loaded. If a match is found, the loader sets E32Image::iHandle for this segment, and then opens it for a second process.

The loader now calls the executive function `E32Loader::ProcessCreate()` to request that the kernel create the process structure, main thread and global data chunk. This results in a `DProcess` object being created for the process. If no `DCodeSeg` object was

found to exist earlier, this in turn leads to the creation of a `DCodeSeg` object for this executable.

As I have said, since this is a code segment for a ROM executable, there is no associated memory area and no need to load any code into the segment from the executable file or perform any code relocations.

I describe this phase of process loading in more detail at the start of Section 10.4.3.

Processing imports

Next the loader must load any DLLs that the executable depends on and which contain static data. To do this, it gets the address of the DLL reference table from the ROM image header and starts to process the entries. Each entry holds a pointer to the image header of the DLL concerned and, for each, the loader generates a further E32Imageobject, which it then populates with information read from the corresponding image header. The loader must also obtain a `DCodeSeg` object for each entry.

This is a procedure that starts in a similar manner to the one performed earlier - searching through the list of existing code segments. However, if it needs a new `DCodeSeg` object, the loader calls the executive function `E32Loader::CodeSegCreate()` to create the object, followed by `E32Loader::CodeSegAddDependency()` to register the dependency in code segments between importer and exporter. If any dependencies themselves link to other DLLs, then they too must be loaded and the loader may call the function that handles loading dependant DLLs, `E32Image::LoadDlls()`, recursively to process these.

Now the loader calls the executive function `E32Loader::CodeSegLoaded()` for each new `DCodeSeg` object created (except that of the `DCodeSeg` belonging to the process executable itself) to mark these as being loaded.

I describe the kernel's involvement in this DLL loading in more detail in Section 10.4.4.

Figure 10.5 shows the program flow as the loader creates the process and handles its import dependencies.

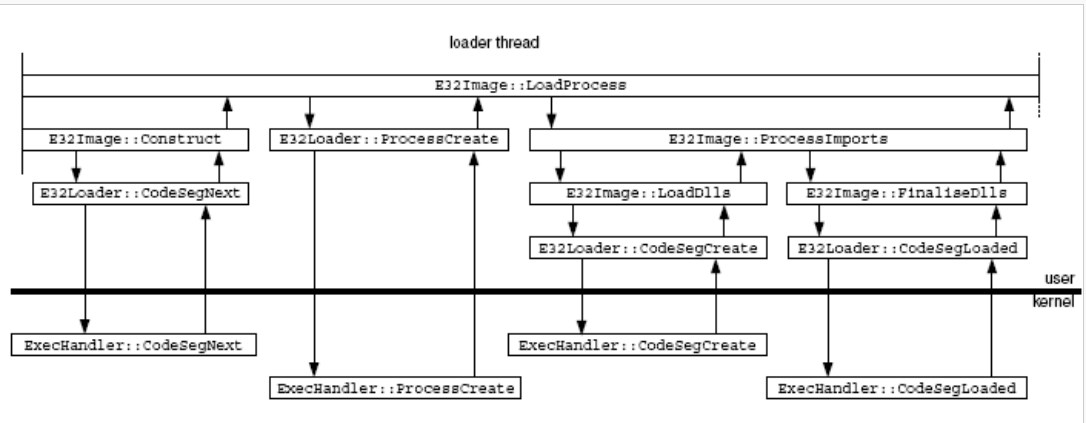

Figure 10.5 *Creating the process and handling its import dependencies*

Completing the request

In the final stage of loading a process, the loader calls the executive function `E32Loader::ProcessLoaded()` to update the state of the new process's thread. (If the `DCodeSeg` object of the process executable is new, and not shared from a preexisting process, this internally marks the `DCodeSeg` object as being loaded.) This function also generates a handle on the new process, relative to the original client thread. The loader then writes the handle back to the client. Next, the loader deletes all the image objects it has created for the executables involved and closes its own handle on the new process.

Finally it completes the original load process message, and control returns to the client thread. This closes the `RLoader` session object and the method `RProcess::Create()` finally returns. I describe the kernel involvement in this final phase in more detail at the end of Section 10.4.3.

Figure 10.6 shows this last phase.

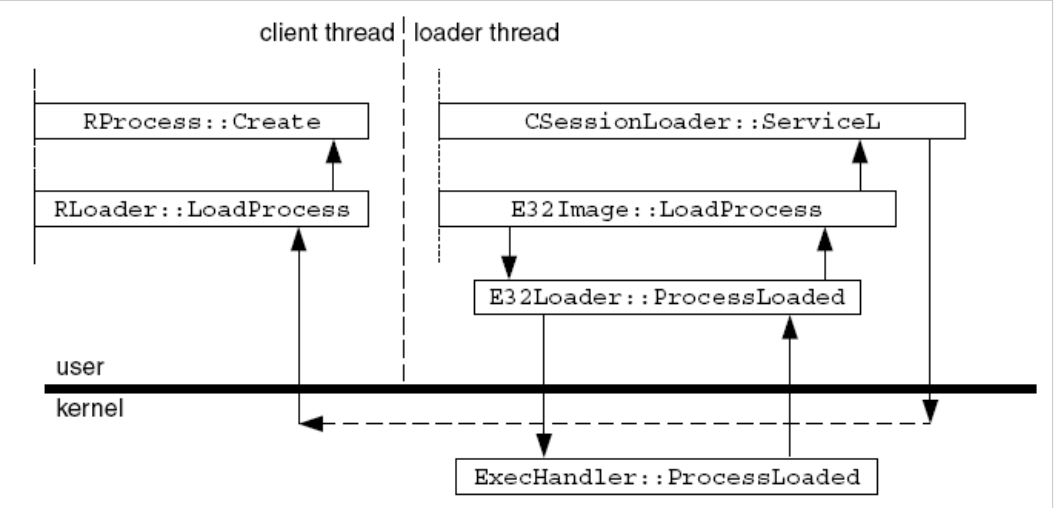

Figure 10.6 *Completing the request*

Assuming the process load was successful, the client then needs to resume the new process for it to run. However, the loader has done its job by this stage and is not involved in this.

The kernel then calls the process entry point, which creates the main thread heap. Next it copies the initialized data section to its run address and clears the un-initialized data area. Then it calls constructors for the EXE itself and for implicitly linked DLLs. Finally it calls the process's public entry point, the E32Main() function.

There are a number of differences in the way an executable is loaded from ROM on EKA2, compared to EKA1. Firstly, in EKA1 a third server, the kernel server, is involved; this runs in supervisor mode. The loader makes a request on the kernel server, asking it to create the new process.

On EKA2, process creation is done in supervisor mode too - but in the context of the loader server. Secondly, in EKA1, the loader copies the initialized data section from ROM to the global data run address as part of the process load operation. However, in EKA2, this data copying is performed kernel-side, and is deferred until the initialization phase of process execution.

Loading a process from non-XIP media

This is similar to the loading of a process from XIP media, but more complex. Since the media is non-XIP, the code must be loaded into RAM for execution. Also, since the import sections and relocations have not been fixed up in advance, these sections need to be processed too.

Locating the process executable

The first difference occurs at the point where the loader searches for the process executable. If it needs to load header information from a candidate file (on a loader cache *miss*), then this requires a full file server read request, rather than relying on the ROM image being memory mapped.

Creating the code segment and process

The next difference occurs when the loader calls `E32Loader::ProcessCreate()` and the kernel creates a `DCodeSeg` object for the process executable. Since the process is not XIP, the kernel allocates a RAM code segment, associates this with the `DCodeSeg` object and returns the code load address within this segment.

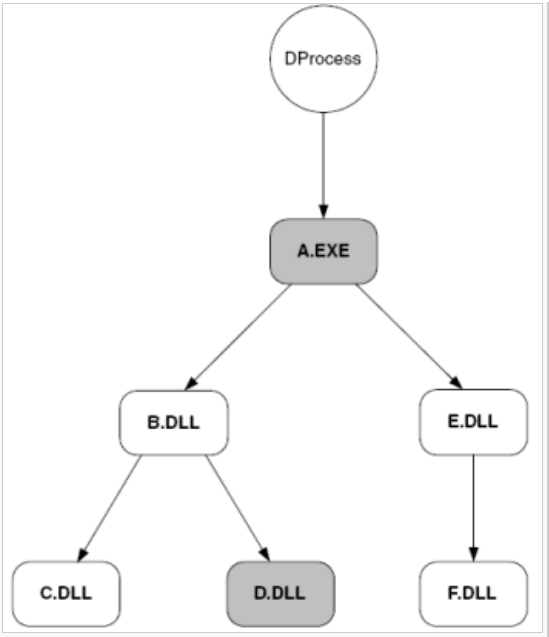

Figure 10.7 *Sample non-XIP code graph*

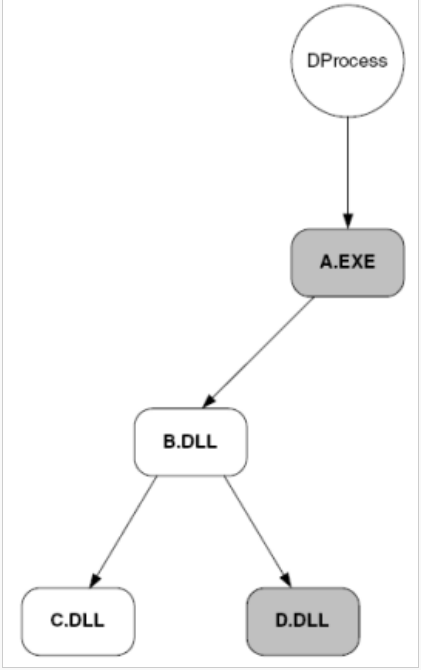

Figure 10.8 *Code graph with some XIP modules*

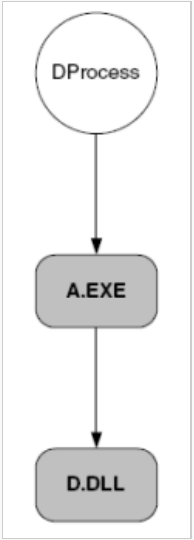

Figure 10.9 *Code graph with all XIP modules*

The loader now needs to load the entire code section from the image file into this code segment at the code load address. If it is a compressed executable, the loader also needs to decompress the code section as it loads it. However, unlike on EKA1, the code segment does not have user-write permissions and so the loader can't directly copy data into it. Instead it uses an executive function to perform this in supervisor mode. If decompression is required, the loader reads portions of the code from the file into a buffer, decompresses it, and then makes use of the user library function `UserSvr::ExecuteInSupervisorMode()` to move it from buffer to code segment in supervisor mode. If decompression isn't needed, then no special scheme is required to transfer the code in supervisor mode. The loader simply issues a file server read request, specifying the code load address as the target address. A media driver will perform the transfer and this can use the kernel inter-thread copy function to write to the code segment.

The loader then reads all the rest of the image (after the code section) from the file into a temporary image buffer in one operation, again decompressing if necessary. Next the loader checks the header information to determine whether the file contains a code relocation section. If so, the loader reads the relocation information from the image buffer and calls a function in supervisor mode to perform each of the relocations on the code section just loaded. (This essentially involves the loader calculating the difference between the code run address provided by the kernel and the base address that the code was linked for. Then it applies this adjustment to a series of 32 bit addresses obtained from the relocation section - see Section 10.1 for more detail.)

If the executable contains an export directory, then the loader fixes up each entry for the run address and writes it into the kernel-side code segment, after the IAT. Again, it uses an executive function to perform this in supervisor mode. (This is only required for PE-derived images. ELF marks export table entries as relocations, so the loader deals with them as part of its code relocation handling.)

If the executable contains an initialized data section, then the loader now copies this from the image buffer into the code segment at the data load address supplied (which is in fact immediately after the code, IAT and export directory).

If the image contains a data relocation section, then the loader applies these relocations in much the same way as it handled code relocations.

Processing imports

Next the loader must load all the DLLs referenced by this executable that are not already loaded (rather than just those that contain static data as was the case for XIP images). The procedure is similar to that described for an XIP executable. However, with non-XIP files, the dependency information is contained in the import section rather than a DLL reference table and it specifies the names of the DLLs rather than pointers to their image headers. Hence, for each import block, the loader has to follow the DLL search sequence described in Section 10.3.2.1. It reads each import block from the image buffer and processes it in turn. If a dependency needs to be loaded from non-XIP media, the loader again has to load and relocate its code and data sections, fix up the export tables and so on, in much the same way that it is loading the main executable.

Once this is done, the loader now has to fix up the IAT for the main executable and any dependencies that it has just loaded from non-XIP media. It examines the import sections of each of these executables, which are now loaded in a series of image buffers. It processes the import blocks in turn, identifying the corresponding exporter and loading its export table into a buffer. Then it calls a supervisor function to fix up the entries in the IAT that correspond with this export table.

Once it has processed all the dependencies, the process load request now continues in much the same way as it did for XIP media.

For executables loaded from non-XIP media, the calling of the executive function `E32Loader::CodeSegLoaded()` for the new `DCodeSeg` objects marks the point when the loader has finished loading any data into these segments.

Loading a library file

There are various overloads of the method to load a library but all eventually call:

```
 TInt RLibrary::Load(const TDesC& aFileName, const TDesC& aPath,
const TUidType& aType, TUint32 aModuleVersion)
```

The argument `aFileName` specifies the name of the DLL to be loaded. The descriptor `aPath` contains a list of path names to be searched, each separated by a semicolon. The argument `aType` specifies a triplet of UIDs which the DLL must match and `aModuleVersion` specifies the version that the DLL must match. As versioning was introduced for EKA2, this final argument is not present in the EKA1 version of the method.

This method first creates an `RLoader` session object and establishes a connection with the loader server. Next it calls `RLoader::LoadLibrary()`, which assembles the request arguments into a message object and sends this to the server.

On receipt of the message, the server handles the request in a similar manner to process load requests. It reads the arguments from the client thread and searches for the specified DLL, following the sequence described in Section 10.3.2.2. Then it creates a `DCodeSeg` object for the DLL and, if loading from non-XIP media, copies the code and initialized data sections into this segment, performing any relocations necessary.

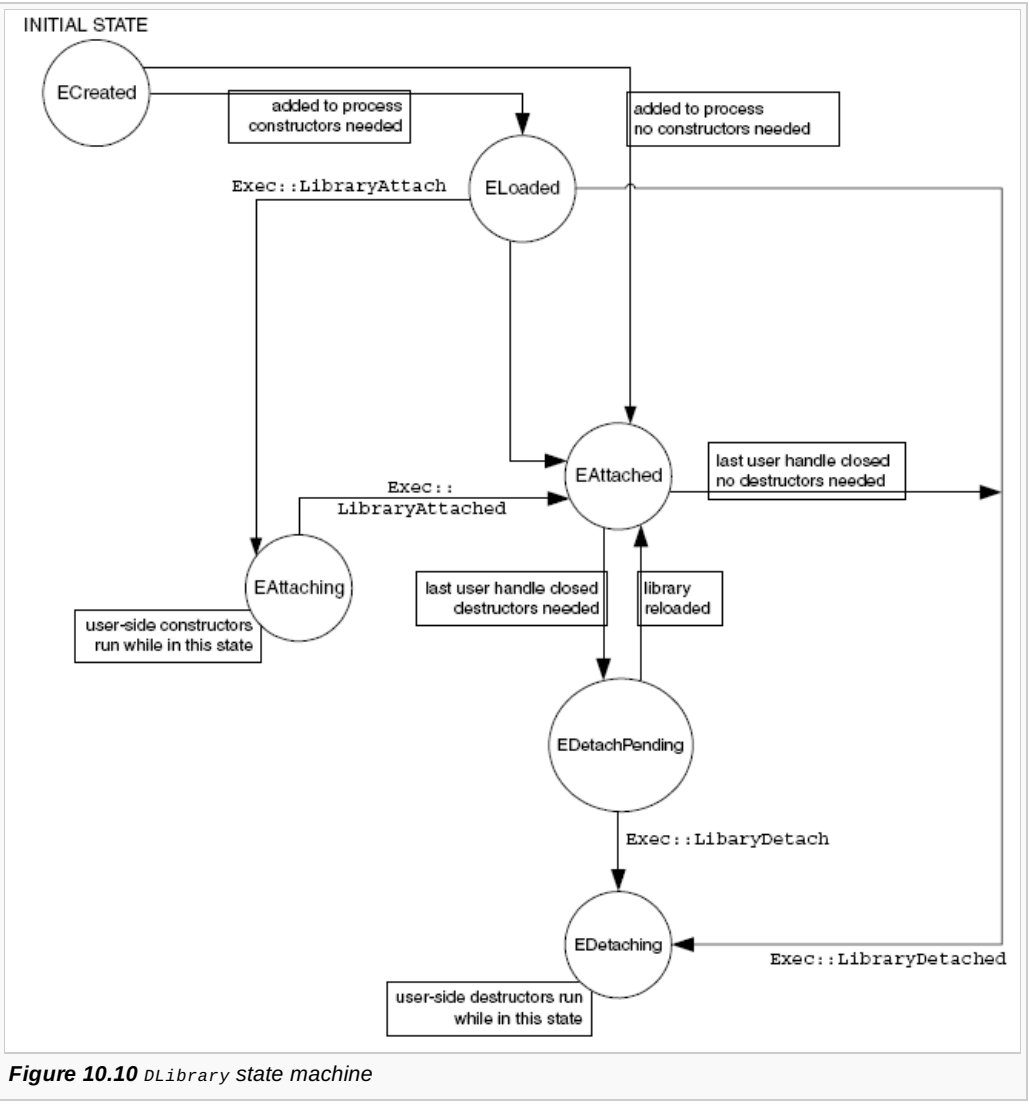

Figure 10.10 *`DLibrary` state machine*

Next it processes any dependencies for the DLL and marks every new code segment created as now being fully loaded. Next it calls the executive function, `E32Loader::LibraryCreate()`, which creates a `DLibrary` object and returns a handle to it. See Section 10.4.4 for more details on this.

Finally, the loader server writes the handle back to the client, deletes the image objects it has created and completes the load library message. Control returns to the client thread which closes the `RLoader` session object.

If the library was loaded successfully, then the method `RLibrary::Init()` is called, still in the context of the client thread. This in turn calls the executive function `E32Loader::LibraryAttach()` to extract the list of entry points for the DLL and all its dependencies. Each entry point is called in turn, passing in the value `KModuleEntryReasonProcessAttach` as the entry reason. This runs any C++ constructor functions for static objects associated with these DLLs.

Finally, a call of the executive function `E32Loader::LibraryAttached()` signals that the entry points have been completed - marking the library as being attached. The method `RLibrary::Load()` now returns.

Note that, in EKA2, the public DLL entry-point, E32Dll(TDllReason) is no longer invoked. This function must be present in every EKA1 DLL, to be called when the DLL is attached to or detached from a process or thread. Unfortunately, this entry-point system cannot provide any guarantees that E32Dll() will be called with the appropriate parameter at the specified time. Because it is not possible to support this functionality reliably, EKA2 removes support for it. This removal simplifies the kernel-side architecture for managing dynamically loaded code, which improves reliability and robustness.

Kernel-side code management

The kernel maintains a representation of all the code resident in the system. The representation includes information on which

modules link to which other modules (the code graph), which processes each module is used by and whether a given module consists of XIP ROM-resident code or non-XIP RAM-loaded code. The generic concepts involved are implemented in the Symbian OS kernel, but the memory model is responsible for the details of how memory is allocated for RAM-loaded code, and how it is mapped into the address space of each process.

The kernel and memory model only allocate memory for code storage and store information about loaded code; they do not actually load the code. The user-mode loader thread, described in the preceding sections, is responsible for loading images from the file system into RAM, relocating them to the addresses allocated by the kernel and resolving linkages to other modules prior to execution.

Code segments

As we have seen, the main object involved in the management of code is the code segment (DCodeSeg). This represents the contents of a single image file (EXE or DLL) relocated for particular code and data addresses. Note that in some memory models, the kernel will create more than one code segment from the same image file if different data section addresses are required. Of course this is expensive in terms of RAM usage, so it only happens if absolutely necessary. The kernel only loads multiple copies where a code segment has writeable static data and is loaded into multiple processes in such a way that the data cannot be at the same address in all those processes. For example, on the moving memory model, a fixed process cannot have its data at the same address as a non-fixed process.

A code segment can be created from an execute-in-place (XIP) image file, in which case it just contains a pointer to the `TRomImageHeader` for that file. Alternatively it can be created from a non-XIP image file; in this case the code segment owns an amount of RAM into which the loader copies the code. The kernel keeps all code segments on three separate lists. There is a doubly linked list (DCodeSeg::GlobalList) to which it adds code segments in order of creation. There is an array of pointers to `DCodeSeg`, sorted in lexicographic order of the root name (DCodeSeg::CodeSegsByName). Finally there is an array of pointers to `DCodeSeg` sorted in order of code run address (DCodeSeg::CodeSegsByAddress). The kernel uses the two sorted lists to allow it to perform a fast binary search for code segments by name, and to allow fast location of the code segment that contains a given run address (to enable C++ exception unwinding to work). The kernel protects these lists, and all other aspects of the global code graph, by the `DCodeSeg::CodeSegLock` mutex.

Each code segment has an array of pointers to the other code segments to which it implicitly links, thus defining a global code graph (iDepCount specifies how many other code segments this one links to, `iDeps` points to an array of pointers to the `DCodeSeg` s on which this one depends). This graph is exact for RAM-loaded code segments (that is, all code segments and dependencies are present) but it is reduced for XIP code segments.

This reduction takes the form of transitively closing the dependence graph, and then omitting any XIP code segments that have no .data or .bss, or are kernel extensions or variants, and are not the explicit target of a load request. (We avoid creating code segments for these since no actual work is required to load them - the code is always visible, being XIP, and no data initialization is required since there is either no data or it has been initialized by the kernel before the loader even started up. Effectively these DLLs are always loaded.)

A code segment can have various attributes (iAttr), as follows:

- `ECodeSegAttKernel` - this indicates that the code is intended to execute in supervisor mode. Such code segments will be accessible only in privileged processor modes
- `ECodeSegAttGlobal` - this indicates that the code should be visible to all user processes regardless of whether they have explicitly loaded it. Used for locales
- `ECodeSegAttFixed` - this is only used in the moving memory model; it indicates that an EXE code segment is associated with a fixed process and so will be fixed up for a non-standard data address
- ABI attribute - this is a 2-bit field indicating which ABI a code segment conforms to. (ABI stands for Application Binary Interface and covers such things as the calling convention used to pass parameters to functions and receive the return value and the manner in which objects are laid out in memory.) Currently we define two ABIs - GCC98r2 and EABI. We use this attribute to facilitate systems in which multiple ABIs coexist, and multiple versions of the same DLL are present. If we are finding an already-loaded code segment, we must find the one with an ABI matching the importing code segment.

Code segments without either kernel or global attributes are standard user code segments. The kernel needs to attach such code segments to a process before they can execute. It will either perform the attach operation at process load time (for the EXE code segment and its dependencies) or when a running process loads a DLL. Each process maintains a list of all code segments directly attached to it (DProcess::iExeCodeSeg points to the main code segment for the process, `DProcess::iDynamicCode` is an array of all explicitly dynamically loaded code segments, each with its corresponding `DLibrary` object). This list only contains directly loaded code segments, not those present solely because they are implicitly linked to by another code segment.

Depending on the memory model, non-XIP code segments may either be visible to all user processes or visible only to those user processes to which they have been attached. The multiple memory model uses the latter scheme; the other memory models use the former. Kernel code segments are attached to the kernel process, but they are not mapped and unmapped; the code segment is visible from privileged modes immediately on creation. Global code segments do not need to be attached to any process. They are visible to all processes immediately after creation.

A code segment also has flags (iMark), as follows:

- `EMarkListDeps` - temporarily used to mark code segments during traversal of the code graph to add code segments to a dependency list
- `EMarkUnListDeps` - temporarily used to mark code segments during traversal of the code graph to remove code segments from a dependency list
- `EMarkLdr` - indicates that the code segment is in use by a load operation
- `EMarkLoaded` - indicates that the code segment and all its dependencies are fully loaded and fixed up
- `EMarkDataFlagsValid` - indicates that the DataInit and DataPresent flags are valid (taking into account dependencies)
- `EMarkDataFlagsCheck` - used to mark that a code segment has been visited when calculating the DataInit and DataPresent flags
- `EMarkData` - indicates that this code segment has .data/.bss and is not an extension or variant (so may require initialization at load time - extensions and variants are loaded and initialized before the loader comes into existence so don't need to be initialized as a result of any loader operation)
- `EMarkDataInit` - indicates that either this code segment or one in the sub-graph below it is a DLL with writeable static data, excluding extensions and variants
- `EMarkDataPresent` - indicates that either this code segment or one in the sub-graph below it has writeable static data, excluding extensions and variants. Note that the difference between `EMarkDataPresent` and `EMarkDataInit` is that the former includes EXEs with writeable static data whereas the latter does not
- `EMarkDebug` - reserved for debuggers.

We need some form of reference count for code segments to cope with the case in which several processes are using a code segment (for example, two instances of the same EXE or two processes loading the same DLL); the kernel can only destroy a code segment when all processes have relinquished their reference on it. A code segment should only be destroyed if no currently running process depends on it, and such dependence may be indirect via implicit linkages from other code segments. We could have done this by reference counting the dependency pointers from each code segment. However, because there may be cycles in the code graph, this scheme makes it difficult to determine when a code segment may actually be removed. The way to do this would be to traverse the graph in the reverse direction - from the exporting code segment to the importing code segment - to see if any process is currently using any of the code segments. This would mean that we would need to maintain two sets of dependence pointers for each code segment, one pointing in each direction in the dependence graph.

In the reference counting scheme we actually use, we do not reference count the dependency pointers. Instead, the reference count of each code segment is equal to the number of processes which currently have the code segment loaded, plus 1 if the (user-side) loader is currently working with the code segment. We indicate the latter case by setting the `EMarkLdr` flag in the `iMark` field. When the kernel creates a `DCodeSeg` object, its access count is 1 and the `EMarkLdr` flag is set. Following a successful load, each code segment in the new dependency tree will be in this state (unless it was already loaded, in which case the access count will be greater than 1). The kernel then adds the entire dependency tree to the address space of the loader's client process (or the newly created process if an EXE is being loaded), which causes the access count of each code segment to be incremented. Finally, during loader cleanup, the kernel decrements the access counts of all code segments with the `EMarkLdr` flag set and resets the flag, which leaves all code segments with the correct access count according to the previous rule. The kernel performs the second and third steps (traversing dependence trees or the global code segment list and modifying the access counts of several code segments) with the CodeSegLock mutex held. The access counts of all code segments must be consistent with the previous rule whenever this mutex is free.

To conserve RAM, the kernel shares code segments whenever possible. There will generally only be one code segment for each loaded EXE or DLL with the same code being shared between all processes. There are some exceptions, which all arise from memory model specific restrictions on data addresses:

- On the moving memory model, non-fixed processes share the same data addresses and so they can share all code segments. Fixed processes must have unique data addresses, since we designed them so that a context switch to or from a fixed process should require no change in the virtual to physical memory map. Symbian OS executables and DLLs do not use position independent code or data, so if a DLL needs to have its .data and .bss at a different address in different processes, the kernel must load multiple copies of the DLL and relocate each copy for different code and data addresses. This means that any code

segments which either have .data or .bss sections or which implicitly link to other such code segments must be unique to that process
- On the direct memory model and the emulator, all processes must have unique data addresses and so sharing is possible only for code segments which don't have .data/.bss sections and which don't link implicitly (directly or indirectly) to any such code segments.

If such an exceptional case is encountered with a non-XIP image file, the kernel will create a second code segment (that is, a second RAM-loaded copy of the code) from the same image file and relocate it for a different data address. If an exceptional case is encountered with an XIP image file, the kernel can't do this, since XIP files no longer have relocation information present. In this case, the load request is rejected.

To implement sharing of code segments, the loader will search for an already loaded code segment with a root file name (ignoring the drive and path, just including the file name and extension) matching the requested file before attempting to create a new code segment. The kernel provides a *find matching code segment* function for this purpose. It finds the next fully loaded code segment (that is, one that is relocated, and has all imports loaded and fixed up) that matches the provided UIDs and attributes and that can be loaded into the client process. The loader then checks the filename and, if it matches, opens a reference on the matching code segment and on each member of the sub-graph below it, marking each one with `EMarkLdr`.

To support the exceptional cases where sharing is not allowed, a code segment may have one of two restrictions on its use:

1. If the `iExeCodeSeg` field is non-null, it indicates that this code segment may only be loaded into a process with that EXE code segment. A code segment which links directly or indirectly to an EXE will have this field pointing to the code segment representing that EXE. This restriction arises in the case where the code segment has an EXE code segment in the sub-graph below it. When this restriction applies, the code segment could potentially be loaded into multiple processes, but these processes must all be instantiations of the same EXE file
2. If the `iAttachProcess` field is non-null, it indicates that this code segment may only be loaded into that specific process. This restriction arises in the case where the code segment, or one in the sub-graph below it, has .data or .bss and this data must reside at a unique address, for example if the code segment is loaded into a fixed process in the moving memory model. When the `iAttachProcess` field is non-null, the `iExeCodeSeg` field points to the EXE code segment of the attach process. The `iAttachProcess` pointer is reference counted.

Figures 10.7, 10.8 and 10.9 show example code graphs. They depict a process instantiated from an EXE file which links implicitly to five DLLs. Figure 10.7 shows the graph which results where all these modules are non-XIP. Shaded modules signify the presence of writeable static data. Figure 10.8 shows the graph which results from loading the same process if two of the DLLs (E.DLL and F.DLL) are in XIP ROM. Since these DLLs are XIP and they have no writeable static data, they do not appear in the code graph. Figure 10.9 shows the graph resulting from loading the same process where all modules are in XIP ROM. Only the EXE and any DLLs with writeable static data appear in the code graph.

Libraries

The kernel creates a kernel-side library object (DLibrary) for every DLL that is explicitly loaded into a user process (in other words, one that is the target of an `RLibrary::Load` rather than a DLL that is implicitly linked to by another DLL or EXE). Library objects are specific to, and owned by, the process for which they were created; if two processes both load the same DLL, the kernel creates two separate `DLibrary` objects. A library has two main uses:

1. It represents a link from a process to the global code graph. Each process has at least one such connection - the `DProcess::iCodeSeg` pointer links each process to its own EXE code segment. The loader creates this link when loading the process. `DLibrary` objects represent additional such links that the kernel creates at run time
2. It provides a state machine to ensure constructors and destructors for objects resident in .data and .bss are called correctly.

Libraries have two reference counts. One is the standardDObject reference count (since `DLibrary` derives from `DObject`); a non-zero value for this reference count simply stops the `DLibrary` itself being deleted - it does not stop the underlying code segment being deleted or removed from any process. The second reference count (iMapCount) is the number of user references on the library, which is equal to the number of handles on the library opened by the process or by any of its threads. This is always updated with the CodeSegLock mutex held. When the last user handle is closed, `iMapCount` will reach zero and this triggers the calling of static destructors and the removal of the library code segment from the process address space. (We need a separate count here because static destructors for a DLL must be called in user mode whenever a library is removed from a process, and those destructors must be called in the context of the process involved. However, the normal reference count may be incremented and decremented kernel-side by threads running in other processes, so it would not be acceptable to call destructors when the normal reference count reached zero.) The loader creates `DLibrary` objects on behalf of a client loading a DLL. A process may only have one `DLibrary` referring to the same code segment. If a process loads the same library twice, a second handle will be

opened on the already existing `DLibrary` and its map count will be incremented.

A `DLibrary` object transitions through the following states during its life, as shown in Figure 10.10.

- `ECreated` - transient state in which object is created. Switches to `ELoaded` or Attached when library and corresponding code segment are added to the target process. The state transitions to `ELoaded` if constructors must be run in user mode and directly to `EAttached` if no such constructors are required; this is the case if neither the DLL itself nor any DLLs that it depends on have writeable static data
- `ELoaded` - code segment is loaded and attached to the target process but static constructors have not been called
- `EAttaching` - the target process is currently running the code segment static constructors. Transition to `EAttached` when constructors have completed
- `EAttached` - static constructors have completed and the code segment is fully available for use by the target process
- `EDetachPending` - the last user handle has been closed on the `DLibrary` (that is, the map count has reached zero) but static destructors have not yet been called. Transitions to `EDetaching` just before running static destructors
- `EDetaching` - the target process is currently running the code segment static destructors. Transitions to `ELoaded` when destructors have completed, so that if the library is reloaded by another thread in the same process before being destroyed, the constructors run again.

Problems could be caused by multiple threads in a process loading and unloading DLLs simultaneously - for example, static constructors running in one thread while destructors for the same DLL run in another thread. To prevent this, each process has a mutex (the DLL lock), which protects static constructors and destructors running in that process. The mutex is held for the entire duration of a dynamic library load from connecting to the loader to completion of static constructors for the DLL. It is also held when unloading a library, from the point at which the `iMapCount` reaches zero to the point where the code segment is unmapped from the process following the running of static destructors. The kernel creates the DLL lock mutex, named DLL$LOCK, during process creation, but it is held while running user-side code; it is the only mutex with this property.

Loading a process

The kernel's involvement in process loading begins after the loader has completed its search for the requested executable image and decided which of the available files should be used to instantiate the new process. The loader will have created an E32Imageobject on its heap, to represent the new image file it is loading. The loader then queries the kernel to discover if the selected image file is already loaded. The `E32Loader::CodeSegNext()` API, which, like all the E32Loader functions, is a kernel executive call, is used for this purpose. If the selected image file is an XIP image, the loader will already have found the address of the `TRomImageHeader` describing it. In this case, the kernel will search for a code segment derived from the same `TRomImageHeader`. If the selected image file is not an XIP image, the loader will know the full path name of the image file and will have read its E32Image header.

The kernel will search for a code segment with the same root name, same UIDs and same version number. In either case, if the kernel finds that the code segment is already loaded, it returns a *code segment handle* to the loader. This is not a standard Symbian OS handle but is actually just the pointer to the kernel-side `DCodeSeg` object. The loader then calls `E32Loader::CodeSegInfo()` on the returned handle; this populates the E32Image object with information about the code segment being loaded, including full path name, UIDs, attributes, version number, security information, code and data sizes, code and data addresses and export information. The loader then calls `E32Loader::CodeSegOpen()` on the handle. This call checks the `EMarkLdr` flag on the code segment; if this flag is clear, it sets the flag and increments the reference count of the code segment. The `E32Loader::CodeSegOpen()` function then performs the same operation recursively on all code segments in the sub-graph below the original one. At the end of this, the loader has a reference on the code segment and on all the code segments upon which it depends, so these will not be deleted.

If the kernel does not find the selected code segment, the loader populates the E32Image object with information read from the E32Image header of the selected file.

The loader then calls `E32Loader::ProcessCreate()`, passing in the E32Imageobject; the latter derives from `TProcessCreateInfo`, which contains all the information the kernel needs to create a new process. Figure 10.2 illustrates the relationship of these classes.

The kernel-side handler, `ExecHandler::ProcessCreate()`, verifies that the calling thread belongs to the F32 process and then does some argument marshaling to pass the parameters over to the kernel side. It then calls `Kern::ProcessCreate()` to do the actual work; this function is also used by the startup extension (EXSTART.DLL) to create the F32 process after the kernel has booted.

Actual process creation proceeds as follows:

1. The kernel creates an object of the concrete class derived from `DProcess`. There is only one such class in any given system (DArmPlatProcess or DX86PlatProcess) and its definition depends on both the processor type and the memory model in use. The generic kernel code calls the function `P::NewProcess()` to instantiate such an object. Once created, this object will contain all the information that the kernel needs to know about the process
2. The kernel stores the command line, UIDs and security information (capabilities, SID and VID) in the new process. It sets the `DObject` name to be the root name of the image file used and calculates the generation number as one greater than the highest generation number of any existing process with the same root name and third UID. If you retrieve the full name of a process, it will be in the form name.exe[uuuuuuuu]gggg, where name.exe is the root filename, uuuuuuuu is the third UID in hexadecimal, and gggg is the generation number in base 10
3. The kernel allocates an ID to the process. Process and thread `IDs` occupy the same number space, beginning at 0 and incrementing for each new process or thread. If the 32-bit ID reaches 2^{32}-1, the kernel reboots to ensure uniqueness of `IDs` over all time. We consider it highly unlikely that this would occur in practice since it would require the system to be active for a very long time without rebooting due to power down
4. The kernel creates the process-relative handles array (DObjectIx) for the new process
5. The kernel creates the process lock and DLL lock mutexes
6. Some platform-dependent initialization of the process object now occurs. Under the moving memory model running on an ARM processor, the kernel just allocates an ARM domain to the process if it is a fixed address process and there is a free domain. Under the multiple memory model, the kernel allocates the process a new OS ASID, a new page directory and a new address allocator (TLinearSection) for the process local address area. Chapter 7 describes OS ASIDs and the address areas used in the multiple memory model. The kernel maps the new page directory at the virtual address corresponding to the allocated OS ASID and initializes it - it copies all global mappings into it if necessary and clears the local mapping area
7. If the process is being created from an already existing code segment, the kernel attaches the existing code segment to the new process. It increments its reference count and creates the process data/bss/stack chunk using the data and bss size information in the code segment. At this point it checks that the run address of the .data section created matches the run address expected by the code segment. If this is not the case (for example, an attempt to create a second instance of a fixed XIP process with .data and/or .bss), the process creation fails
8. If the process is not being created from an existing code segment, the kernel creates a new code segment using the information passed in by the loader. It increments the reference count of the code segment, so that it becomes 2 - one for the loader, one for the new process. I will describe code segment creation in more detail later in the chapter, but essentially, for non-XIP code, the kernel allocates an address and maps RAM to store the code and the initial values of the initialized data. For all code segments, it allocates a data address if necessary. For EXE code segments, the kernel creates the new process's data/bss/stack chunk using the data/bss size information passed from the loader. Finally, the kernel passes run and load addresses for both code and data back to the loader
9. The kernel creates the process's first thread. It sets its `DObject` name to Main, and sets its stack and heap sizes to the values passed by the loader; these values were originally read from the E32Image header. The kernel marks the thread as *process permanent*, which means that if it exits for any reason, the kernel will kill the whole process including any threads that are created after this main one. It also marks the thread as *original*, which signifies that it is the first thread. This information is used by the process entry point code - the original thread causes the process static data to be initialized and E32Main() to be called, whereas subsequent threads cause the specified thread entry point to be called. The first thread is created with an M-state of `DThread::ECreated`, so it cannot yet be resumed
10. The newly created process is added to the process object container (DObjectCon)
11. The kernel creates a handle from the calling thread (the loader) to the new process and passes this handle back to the loader.

After `E32Loader::ProcessCreate()` completes, if a new code segment has been created, the loader needs to load and relocate the code (if not XIP) and load all DLLs to which the new EXE code segment implicitly links - see Section 10.4.4 for more details of this. Finally, after it has resolved all dependencies, the loader calls `E32Loader::ProcessLoaded()`. This performs the following actions:

1. If a new EXE code segment was created for the process, the kernel calls `DCodeSeg::Loaded()` on it. This performs steps 1 to 3 of the `CodeSegLoaded()` function, described at the end of Section 10.4.4
2. The kernel maps the EXE code segment and all the code segments on which it depends into the process and increments their reference counts
3. The kernel sets the `EMarkLoaded` flag of the EXE code segment to enable it to be reused to launch another instance of the process

4. The kernel changes the first thread's M-state to `DThread::EReady`; it is now ready to be resumed
5. The kernel creates a handle from the loader's client to the new process; the client will eventually use this handle to resume the new process.

After `E32Loader::ProcessLoaded()` has completed, the loader copies the new handle to the process back to the client, and cleans up the E32Image object corresponding to the new process. Finally, it completes the client request and control returns to the client.

10.4.4 Loading a library

Similarly to process loading, the kernel's involvement in library loading begins after the loader has completed its search for the requested DLL image and decided which of the available files should be loaded. The loader will have created an E32Imageobject on its heap to represent the new image file being loaded. This procedure is carried out by the loader function `E32Image::LoadCodeSeg()`, which then calls `E32Image::DoLoadCodeSeg()` to perform the actual load; this latter function is also called while resolving implicit linkages to load the implicitly linked DLLs. In a similar way to process loading, the loader then queries the kernel to discover if the selected image file is already loaded. It makes an additional check while searching for a matching code segment - it must be compatible with the process it is destined to be loaded into. An incompatibility can result from any of the following considerations:

- If a code segment links directly or indirectly to an EXE, it may only be loaded into a process instantiated from that EXE. This is because only one EXE file can be loaded into any process; an EXE cannot be loaded as a DLL
- If a code segment has writeable static data or links directly or indirectly to such a code segment, it may only be loaded into a process with which the address of the static data is compatible. This rule affects the moving and direct memory models. A code segment with writeable static data loaded into a fixed process will not be compatible with any other process, and such a code segment loaded into a moving process will be compatible with all moving processes but not with fixed processes.

If the kernel finds that the selected code segment or the file being loaded is an XIP DLL with no .data/.bss and is not the explicit target of the load request, the function `E32Image::DoLoadCodeSeg()` returns at this point. If the DLL is already loaded the function will have populated the E32Imageobject with information about the already-loaded DLL, including its code segment handle. Execution then proceeds in the same way as it would after the creation of a new code segment.

If the kernel does not find the selected code segment, the E32Image object is populated with information read from the E32Imageheader of the selected file.

The loader then calls `E32Loader::CodeSegCreate()`, passing in the E32Imageobject; the latter derives from `TCodeSegCreateInfo`, which contains all the information the kernel needs to create a new code segment. Figure 10.2 illustrates the relationship of these classes.

The kernel-side handler `ExecHandler::CodeSegCreate()` verifies that the calling thread belongs to the F32 process, and then does some argument marshaling to get all the parameters over to the kernel-side, this time including a handle to the loader's client process, since that is the process into which the new code segment will be loaded. Actual code segment creation then proceeds as follows:

1. The kernel creates a `DMemModelCodeSeg` object; this is the concrete class derived from `DCodeSeg`. There is only one such class in any given system and its definition depends on the memory model in use. The generic kernel code calls the memory model function `M::NewCodeSeg()` to instantiate such an object. Once created, this object will contain all the information that the kernel needs about the code segment
2. The kernel copies the UIDs, attributes, full path name, root name, version number and dependency count (count of number of code segments to which this one implicitly links) into the object. It allocates an array to store the dependency linkages that make up the code graph immediately below this code segment
3. If the code segment is XIP, the kernel stores a pointer to the corresponding `TRomImageHeader` in it
4. If the code segment is a user-side EXE, then the process object with which the code segment was created will have been passed as a parameter. The kernel now creates the data/bss/stack chunk for that process and commits sufficient memory to it to hold the process's .data and .bss sections. On the multiple memory model, this chunk is always at the same virtual address - 0x00400000 on both ARM and IA32. On the moving memory model, the virtual address depends on whether the process is fixed or moving and, if fixed, whether it is XIP. If XIP, the kernel uses `iDataBssLinearBase` from the `TRomImageHeader`. Moving processes have their .data/.bss at 0x00400000
5. If the code segment is XIP, the kernel copies the `EMarkData`, `EMarkDataPresent` and `EMarkDataInit` flags from the `TRomImageHeader`, and sets the `EMarkDataFlagsValid` flag (since the ROM builder has already looked at all the dependencies). It reads the addresses of code and data, entry point, exception descriptor and export directory from the `TRomImageHeader` and passes them back to the loader. On the moving memory model, an XIP code segment with writeable static data loaded into a fixed process is marked as only available to that process by setting the `iAttachProcess` field of the

code segment to point to the process

6. If the code segment is not XIP, the kernel allocates memory to hold size and address information, and copies the size information from the information passed by the loader. If the code segment has writeable static data, the kernel sets the `EMarkData`, `EMarkDataPresent`, and `EMarkDataFlagsValid` flags, and also sets the `EMarkDataInit` flag if the code segment is not an EXE code segment. If the code segment does not have writeable static data, the kernel cannot determine the status of these flags until it has resolved all dependencies
7. For non-XIP code segments on the moving memory model, the kernel allocates an address and commits memory in either the kernel or user code chunks to hold the actual code. For kernel-side code segments with writeable static data, the kernel allocates space on its heap to store the data. For user-side code segments with writeable static data, the allocation of the data address depends whether the code segment is destined for a fixed or moving process. For moving processes, the kernel allocates an address in the DLL data address range (0x30000000 to 0x3FFFFFFF), but does not yet commit the memory. For fixed processes, the kernel creates the process DLL data chunk if necessary and then commits memory to it to hold the static data
8. Non-XIP code segments in the multiple memory model can be user, kernel or global (on the moving model, global is the same as user, since all code is visible to all processes). For kernel code segments, the kernel allocates an address for and commits memory to a special kernel code chunk to hold the code itself, and allocates space for writeable static data on the kernel heap. For global code segments, the kernel allocates an address for and commits memory to a special global code chunk to hold the code itself - writeable static data is not permitted. For user code segments, the kernel allocates an address in the standard user code range and, if necessary, in the user data range too. It allocates sufficient physical RAM pages to hold the code and attaches them to the code segment. At this point, memory is not yet allocated for any static data. The kernel then maps code pages into the current process (F32), at the allocated address
9. The kernel adds the code segment to the three global code segment lists: unordered, ordered by name and ordered by run address
10. The kernel sets the `iAsyncDeleteNext` field (usually zero for a `DBase`-derived object) to point to the code segment itself. This property is used subsequently in kernel executive calls that take a code segment handle to verify that the handle refers to a valid code segment object
11. The kernel passes the updated code segment information, including the allocated load-time and run-time addresses for the code and data sections, back to the loader.

After the new code segment is created, the loader reads in the code from the file system and relocates it to the address allocated by the kernel. The loader also examines the import section of the loaded DLL and loads all implicitly linked DLLs. This is a recursive process, since each new DLL will have its own implicit linkages, which will require further DLLs to be loaded. When all required DLLs have been loaded, the loader resolves all import references between them. Then the loader calls `E32Loader::CodeSegLoaded()` on each one, finishing with the DLL that was the explicit target of the original load request. In this function, the kernel performs the following actions:

1. It performs an IMB (Instruction Memory Barrier) operation on the address range of the loaded code. This ensures coherence between the D and I caches on a Harvard cache architecture. (ARM processors with separate instruction and data caches (everything after ARM7) do not maintain coherence between these two caches. So if a memory location is resident in the data cache and is dirty, that is, the value has been modified but not written back to main memory, the new value will not be fetched by an instruction fetch to that location. To ensure that the correct instruction is executed, it is necessary to clean the data cache - to write any modified data in the cache back to main memory - and to invalidate the instruction cache)
2. On the multiple memory model, if the code segment is a standard user-side code segment, the kernel unmaps it from the loader address space. If the code segment is kernel code, the kernel changes the permissions for the mapping from supervisor read/write to supervisor read-only. If the code segment is global code, it changes the permissions from supervisor read/write to user read-only
3. The kernel traverses the code graph recursively to ensure that the `EMarkDataPresent` and `EMarkDataInit` flags are set correctly for each code segment
4. Finally the kernel sets the `EMarkLoaded` flag to indicate that the code segment is fully loaded and ready for use.

At this point, the DLL and all the DLLs on which it depends have been loaded. This is the point at which execution continues after querying the kernel (in the case where the DLL was already loaded). The loader then calls the kernel function `E32Loader::LibraryCreate()`, passing in the code segment handle of the main subject of the load and a handle to the loader's client thread. In this call the kernel first looks at the client process to discover if a `DLibrary` representing the loaded code segment already exists in that process.

If it does, the DLL is already loaded into that process, so the only thing to do is to change the `DLibrary` state to `EAttached` if it was

originally `EDetachPending` (since the C++ destructors have not been called there is no need to call the constructors again), and to create a new handle from the client thread or process to the `DLibrary`, which is then passed back to the loader. If there is no `DLibrary` in the client process corresponding to the loaded code segment, the kernel creates a new `DLibrary` object and attaches it to the process. Then it maps the new code segment and all the code segments in the sub-graph below it into the process address space (and increments their reference counts correspondingly). It creates a new handle from the client thread or process to the new `DLibrary` and passes it back to the loader. If the main code segment has the `EMarkDataInit` flag set, the kernel sets the state of the `DLibrary` to `ELoaded`, since C++ constructors must be run before it is ready for use; otherwise it sets the `DLibrary` state to `EAttached`.

Control then returns to the loader, which writes the new handle back to the client and then completes the load request, at which point control returns to the client thread.

Summary

In this chapter I have described the process of loading executables and the management of executable code in Symbian OS, from both the file server's perspective, and the kernel's. In the next chapter, I shall go on to look at another key user-mode server, the window server.