Optimizing Performance in Windows Embedded Compact 7

Windows Embedded Compact 7 Technical Article

Writer: Stan Sinasohn

Published: May 2011

Applies To: Windows Embedded Compact 7

Abstract

When you design more complex embedded devices, optimizing performance becomes critical. Windows Embedded Compact 7 provides several ways to optimize and track system performance. This paper describes specific techniques to improve device performance by reducing system boot time and the size of the run-time image. In addition, this paper discusses tools you can use to measure and track system performance of your existing Windows Embedded Compact OS.

Introduction

You can optimize an embedded device OS in many ways. This paper focuses on two areas for optimizing your device system performance, because they provide the most substantial benefit for customer experience and system implementation.

First, it describes how to design your OS with optimization and performance in mind from the beginning. For example, you should understand what contributes to the image size, how image size affects the OS load time, and the responsiveness of OS components. This paper describes OS development best practices and how to maximize OS performance by tuning the paging pool and other configuration files.

Second, it describes how to reduce the OS boot time by analyzing boot stages and system component load time, configuring and analyzing the boot loader, and optimizing driver load time.

Optimizing the OS and Run-Time Image

When you first begin to develop your OS and run-time image, it is important to keep optimization and performance in mind. This section describes the following ways in which you can optimize your OS and increase performance:

- Understanding image size
- Managing your files in the OS
- Optimizing configuration files
- Code optimization best practices

Understanding Image Size

When you design and build your OS, consider the size of your built run-time image. Properly sizing your image will help you maximize responsiveness and the amount of memory available to the OS and reduce load time. You can use nk.bin file size or ROM image size to estimate your image size.

The nk.bin file is a single, compressed binary file that the **romimage** tool creates during the build process. This file is the actual run-time image that contains the kernel and downloads to your device from the development computer.

The ROM image contains uncompressed files that execute in place (XIP), including system executables and DLLs. The ROM image also contains miscellaneous files that these applications use, such as fonts, waveform audio files (.wav), and bitmaps.

Many things can affect ROM image size, including the CPU, the board support package (BSP), the type of build (retail, checked, or debug) and the SYSGEN variables included.

Because the ROM image contains additional files, it provides a more accurate estimate of image size than nk.bin does. Therefore, we recommend using the ROM image size to estimate your final image size.

Using the Viewbin Tool to Determine ROM Size

There are multiple ways to determine your ROM image size; however, the most accurate is to use the **Viewbin** tool to view the Table of Contents pointer (pTOC) information in the nk.bin header.

To use the Viewbin tool to find the size of the ROM image

1. In Platform Builder, right-click the name of your OS design, and then click **Release Directory in Build Window**.
2. At the command prompt, type **viewbin -t nk.bin**.

   In the output, the difference between `Physical First:` and `Physical Last:` is the actual size of your ROM image. In the example below, the size of the ROM image is 2AE3828, or almost 50 MB.

```
Found pTOC  = 0x8aaf1800
ROMOFFSET = 0x00000000

ROMHDR ---------------------------------------
    ...
    Physical First      : 0x88001000
    Physical Last       : 0x8AAF4828
    ...

    Num Modules         :        282
    Num Files           :        115
    CPU                 :     0x01c2 (Thumb)
```

Managing OS Files

You can improve OS response time and optimize memory usage by correctly managing OS files in the binary image builder (.bib) configuration file. Keep in mind that there is always a tradeoff between image size and memory usage. For example, you can reduce image size by placing all your files in the Files section of the .bib configuration file because they will be compressed by default. However, performance will be slower because the system has to uncompress the files before use.

When determining which section of the .bib configuration file to place a file in, consider the file type and how frequently the system accesses the file.

If the system does not access the file frequently, do not compress the file. Instead, use the file cache to keep the data in RAM so the system can access the data without accessing the actual file.

The following table shows in which section of the .bib configuration file different file types belong.

Section	File Types	Additional Notes
MODULES	• Executable • Code binaries • DLLs • Uncompressed	Executable files in the MODULES section that are not compressed will be executed in place.
FILES	• Non-executable • Data • Resource DLLs • Audio • Bitmaps that are compressed by default	Managed code must be located in the FILES section because the kernel does not load managed code. Instead, the .NET Compact Framework opens managed code as a memory-mapped file.

Non-Executable Files

In addition to ensuring that non-executable files are located in the FILES section of the .bib configuration file, you can take the following steps to optimize how the system loads and uses these files:

- Use image files that match display resolutions and color depths. This eliminates lag time for the system to resize or otherwise adjust image files for display on the device screen. If you use multiple display configurations, use multiple image files with IF/ENDIF statements in the configuration file to specify the file to use for each display. However, multiple copies of the image file increase overall image size.
- Experiment with different image formats to identify which one displays with the best quality and load time.
- Use the appropriate bitrate for audio files. Audio bitrate affects the amount of data the system buffers for playback. For example, the Buffering Stream Filter in DirectShow uses a 1 MB buffer by default. This size allows the buffer to store 64 seconds of data at 128 kbps, but only 25 seconds of data at 320 kbps.
- Reuse image and audio files that are already in ROM to minimize system latency when the system has to access the file repeatedly.

Executable Files

To optimize executable files in your OS image, do the following. For additional information about best practices to use when you develop your code, see Optimizing Code.

- Remove redundant and unused code in the file.
- Avoid using floating point arithmetic when possible. This minimizes the need for the system to link to external libraries such as the Floating Point Unit (FPU) or software

emulation to perform floating point calculations, which are slower and require additional code that the compiler generates.

- Avoid using static linking libraries when possible. Static libraries often require that code is duplicated throughout your image, which results in a larger image size.
- When the image contains dynamically linking libraries the system is less likely to page out shared library code, which results in fewer page faults. In addition, files that load by using dynamically linked libraries automatically inherit changes to the libraries without recompiling or rebinding.
- If you use dynamically linked libraries, verify in the sources file and makefile file that you link to the correct library files for your file type.
- Place constant or large data tables of code in a separate file from binary files.
- Allocate large read/write data at run time to minimize system lag when it accesses the data.
- Place your resources in a separate resource DLL. Use **dumpbin** with the `/headers` option to display file header and section headers. The information in the headers indicates the collective size of your code, data, and resources.

  In the following example, the data (`.data`) size is 5000 (20480 kb), the code (`.text`) size is FA000 (1024000 kb), and the resource (`.rsrc`) size is 1A000 (106496 kb).

```
dumpbin /headers <your module>
```

```
Dump of file c:\WINCE600\release\MyDll.dll

File Type: DLL

Summary

        5000 .data
        C000 .pdata
       11000 .reloc
       1A000 .rsrc
       FA000 .text
```

Optimizing Configuration Files

Your image contains different types of configuration files, not all of which affect performance. However, the binary image builder (.bib), DIRS, and DAT configuration files will affect your system's performance. The following sections describe customizations to these files that can improve performance.

Binary Image Builder (.bib) File

The binary image builder (.bib) file specifies the files and modules that are in the OS image and the attributes that dictate how the OS handles each file and component. The .bib file also defines the physical memory in your system.

The kernel requires a memory table that identifies each region of memory as either RAM or ROM and the amount of RAM available for applications. The following example shows the MEMORY section of a .bib file.

```
1.   MEMORY
2. NK        9f800000  00800000  RAMIMAGE
3.   ; Only specify one RAMIMAGE section
4.   ; Reserve 4K at reset vector for boot loader and diagnostics
5.   RESERVE    9fc00000    00001000
6.   RAM       80080000  00780000  RAM
7.   ; Common RAM regions
8.   AUD_MDD    80002000  00000800  RESERVED
9.   DISPLAY    80008000  00013000  RESERVED
10.  DRV_GLB    80020000  00001000  RESERVED
11.  DBGSER_DMA  80022000  00002000  RESERVED
12.  SER_DMA    80024000  00002000  RESERVED
13.  IR_DMA    80026000  00002000  RESERVED
```

In the preceding example, line 2 defines the memory region for the kernel (NK); it starts at the address 0x9f800000 and is 8 MB in size:

```
NK        9f800000  00800000  RAMIMAGE
```

Line 6 defines the region allocated for RAM; it starts at the address 0x80080000 and is 7.6 MB. The kernel allocates RAM for use by the file system or object store. The kernel also uses RAM to process virtual address spaces such as heaps and stacks, memory mapped files, and writable data sections.

```
RAM       80080000  00780000  RAM
```

Note When Romimage.exe runs, the kernel fixes up all uncompressed executable code to run at a virtual address in slot 0 (zero), an address space that the kernel creates; actual code is located in the region specified in the RAM entry.

The preceding example also specifies six regions as reserved. The system uses the six memory regions, but the regions are not included in the run-time image or RAM space.

Config.bib File

Use the Config.bib file to customize Romimage.exe output to optimize the image size and file performance. You can use the Config.bib file to create an s-record file (.sre) to use on a Motorola 32-bit device or an absolute binary data (.abx) file version of your ROM image, which contains the raw binary image.

The Config.bib file also offers a number of options available for you to optimize your system:

- AUTOSIZE – If you store your image in RAM, you can set all RAM that the image does not occupy as system RAM and as object store RAM without modifying the memory parameters set in the .bib files. Set `AUTOSIZE = ON` and Romimage.exe combines the system RAM and run-time image memory regions into one region. The run-time image fills this region from the lowest address to the highest address, and modifies RAM parameters in the system's Table of Contents so that the system can use all remaining RAM.

  For example, if you have a 5 MB image and you install it on a device that has 20 MB of RAM, the system will assign 10 MB for the image itself and 10 MB for system RAM. However, this leaves 5 MB of unused RAM. If you use AUTOSIZE, your image occupies 5 MB of the available RAM on the device and you can pool the remaining 15 MB to use as system RAM and object store RAM.

- COMPRESSION – If you set `COMPRESSION = ON`, Romimage.exe compresses the writable sections in your image. If the data and files in the writable sections are compressed, the kernel uncompresses them as it loads them into RAM. If you choose to use compression, file size will be smaller, but load time will be longer. You must also add the compression component to the image.

  Note The system always automatically copies writable sections to RAM, even if they are not compressed.

- FSRAMPERCENT – Use FSRAMPERCENT to specify the percentage of RAM you want to allocate to the file system. By default, FSRAMPERCENT defines 50 percent of RAM for the file system. You can adjust FSRAMPERCENT to reduce the percentage of RAM available to the file system and make more RAM available to the system to use.

For more information about the options available in the Config.bib file, see Binary Image Builder (.bib) File CONFIG Section (http://go.microsoft.com/fwlink/?LinkId=220015).

DIRS File

Use the DIRS file to specify to Build.exe the subdirectories that contain the source code you want to build into your image. By including only the code necessary to support your platform, you can speed up your build process and minimize your image size.

Build.exe builds the source code in the directory in which it runs and in any subdirectories named in the DIRS file. To update only a part of your image without rebuilding all of it, run Build.exe in the subdirectory you want to update.

You also need to ensure that any driver dependences specified in the Platform.bib file have matching relationships specified in the DIRS file that builds the driver. Use `@CESYSGEN` tags to ensure that the DIRS file that points to the driver code matches the dependencies defined in the Platform.bib file. The following code example shows a segment of a Platform.bib file that specifies a dependency between the `ddi_gx.dll` driver for a target device and the CE_MODULES_DEVICE SYSGEN variable.

```
; @CESYSGEN IF CE_MODULES_DEVICE
   ddi_gx.dll      $(_FLATRELEASEDIR)\ddi_gx.dll     NK  SH
; @CESYSGEN ENDIF CE_MODULES_DEVICE
```

In this example, the DIRS file that points to the `ddi_gx.dll` driver must contain the same `@CESYSGEN IF` logic.

For more information about the .DIRS file, see DIRS File (http://go.microsoft.com/fwlink/?LinkId=220016).

DAT File

Use the DAT files to define the directory and file locations of the initial settings in your image. When you cold boot your image on the device, Filesys.exe uses system DAT files to create the directories, links, and files in the root file system on the device. When it finds a DAT file, Filesys.exe copies the files defined in the DAT file to their defined locations.

This results in two copies of the file stored in your system, but the system can access the file in RAM much faster.

For more information about the DAT file, see File System (.dat) File (http://go.microsoft.com/fwlink/?LinkId=220017).

Optimizing Code

As you design and build your OS image code, consider the following optimization and performance conditions so you can reduce latency and performance issues in your system.

Priority Inversion

Sometimes a resource used by a lower-priority thread delays the running of a higher-priority thread that is waiting for the same resource. To free the higher-priority thread, the system triggers a process called priority inversion. The system inverts the priority level of the two threads, allowing the thread holding the resource to run at a higher priority until it releases its use of the resource.

For example, if you have two threads at different priorities as follows:

- The NDIS IST thread runs at priority 116
- The TCP send thread normally runs at priority 251

When the lower-priority TCP send thread has locked a resource that the higher-priority NDIS IST thread needs, the NDIS IST execution will switch the priority of the TCP send thread with the priority of the NDIS IST thread until the required resource is released. To estimate the impact of the priority inversion, manually switch the TCP send thread to priority 116 and lower the NDIS IST thread to priority 251 so no priority inversion occurs. In this example, priority inversion reduces performance by almost 50 percent.

Windows Embedded Compact 7 uses priority inheritance to avoid priority inversion. However, other ways to avoid priority inversion include the following:

- Adjust thread priorities
- Use **TryEnterCriticalSection** to synchronize resources
- Serialize your send/receive operations

Depending on your system design and needs, you can implement one or more of these solutions.

Thread Switching

Excessive thread switching slows system performance even when priority inversion is reduced or eliminated. Kernel Tracker indicates between 1 ms and 4 ms lag per interrupt.

You can use Kernel Tracker to view and analyze the activity that could be causing high CPU loads. A high number of hits in **KCNextThread**, **DoWaitForObjects**, **WaitForSingleObject**, or **WaitForMultipleObjects**, all located in kernel.dll, might indicate many short instances of thread switching are occurring. When you see a high number of hits in these functions, use **CeLog** and Kernel Tracker to view the thread behavior to determine why threads are running for such short periods of time. You can then alter your code to make a thread switch run for a longer period of time.

You can also reduce thread switching by batching jobs and processing as much as possible before reenabling interrupts.

Code Design

To optimize your code design during development, minimize the number of times your system polls the power management component to reduce CPU consumption. Also, to ensure that your system does not repeatedly read the same data, consider caching bus traffic, CPU consumption, and speed data.

Tuning the Paging Pool

A paging pool is a reserved block of RAM that the system uses to store pages of code. The kernel has two pools: one for loader pages and another for the file system.

Loader pages include executable code from the FILES section of the .bib configuration file and compressed executable code from the MODULES section of the .bib configuration file. File system pages include file-backed memory-mapped files and the file cache filter.

If you do not enable the paging pool, the device can use all the RAM for paging, which significantly increases RAM usage. In Windows Embedded Compact 7, the paging pool is enabled by default.

Choosing Paging Pool Size

Increasing the size of the paging pool makes more RAM available for paging, which results in fewer page faults. However, setting the paging pool volume too high limits the amount of RAM available for the remainder of the system to use. The default size of the paging pool is 3 MB for the loader and 1 MB for files.

It is important to determine the right balance between RAM usage for the paging pool and for paging performance. For example, if you have XIP code with no compression in the MODULES section of the .bib file, you can reduce or eliminate the paging pool for the loader and make that memory available to the system.

Consider the following factors when determining how large to set the paging pool on your system:

- **OEM component attributes, such as device drivers**

  Total size and pageability may require a larger paging pool.

- **Total available RAM**

  The more RAM available, the larger paging pool you can specify without sacrificing the amount of RAM available for the system to use.

- **Processor speed**

  A faster processor will allow faster paging so you can reduce the size of the paging pool.

- **What type of flash memory is being used**

  If the image is running from NOR memory, uncompressed executables from the MODULES section will run directly out of the image without using any pool memory. However, if the image is running from NAND memory, executables from the MODULES section will be paged using the pool so you will need a larger paging pool.

- **Common, critical scenarios**
  - System start

- Running multiple applications at the same time
- Running big applications
- Running applications that read a lot of file data

To analyze what effect the size of the paging pool has on system performance, use **CeLog** with boot zones set to view the number of faults on each page. If you see a high number of faults, you can increase the loader paging pool size to reduce the number of faults. If you see a low number of faults, you can decrease the loader paging pool to maximize the amount of RAM available to the system.

Setting Paging Pool Size

You initially set the paging pool size in the config.bib file when you build your image. However, you can also modify the paging pool size at run time using an OAL I/O control. Setting the paging pool size at run time will override the size set in the config.bib file.

To set the paging pool size when you build your image

1. Use the `FIXUPVAR` tag in the MEMORY section of your config.bib file as follows:

```
MEMORY
nk.exe:LoaderPoolTarget    00000000 00300000 FIXUPVAR
nk.exe:FilePoolTarget      00000000 00100000 FIXUPVAR
nk.exe:PageOutLow          00000000 00100000 FIXUPVAR
nk.exe:PageOutHigh         00000000 00300000 FIXUPVAR
```

- `LoaderPoolTarget` defines the size of the paging pool for loader pages.
- `FilePoolTarget` defines the size of the paging pool for file system pages.
- `PageOutLow` starts the trimmer thread when the page-free count is below this value.
- `PageOutHigh` stops the trimmer thread when the page-free count is above this value.

To modify the paging pool at run time

1. Use the **NKPagePoolParameters** structure passed to the **IOCTL_HAL_GET_POOL_PARAMETERS** OAL I/O control as follows:

```
typedef struct {
  WORD NKVersion;
  WORD OEMVersion;
  PagePoolParameters Loader;
  PagePoolParameters File;
  NKPagePoolParameters;
}
```

For more information about the `NKPagePoolParameters` structure, see NKPagePoolParameters (http://go.microsoft.com/fwlink/?LinkId=220018).

Minimizing Boot Time

This section focuses on optimizing the following three areas of the boot process:

- **Analyzing boot times**

Using processes and tools to analyze boot stages and how long it takes each component to load. This analysis includes identifying lags in the boot time and determining how to minimize them.

- **Optimizing the boot loader**

  Identifying operations and settings that affect performance and steps you can take to minimize load time and improve performance.

- **Driver loading**

  Optimizing driver load times and performance by using tools to identify delays during driver initialization.

Using CeLog to View and Analyze Boot Times

CeLog is an event-tracking tool that logs a set of predefined kernel and coredll events. **CeLog** is easy to activate and configure. Use it to track the boot time for different components of the boot sequence, and use this information to help improve system performance.

Note When you analyze boot times, use a Retail build that is as similar as possible to your own shipping configuration. Add the KITL and **CeLog** tools as described in the following sections.

Using CeLog to Track Boot Times

To use CeLog to view boot time events, you must first turn on **CeLog** tracking in the OS by setting the following build options and registry settings.

To enable CeLog in your image

1. Right-click your OS design and then click **Properties**.
2. In the **OS Design Property Pages** window, select the **Build Options** node.
3. Select the following check boxes:
   - **Enable event tracking during boot** (IMGCELOGENABLE=1)
   - **Flush tracked events to release directory** (IMGAUTOFLUSH=1)
   - **Enable Profiling = Yes** (IMGPROFILE=1)
   - **Enable KITL = Yes**

1. In the device platform.reg file, set the following key:

```
HKEY_LOCAL_MACHINE\System\CeLog

"FlushTimeout"=dword:AFC8  ;45 seconds
```

2. In the registry of your development computer, set the following key:

```
HKEY_CURRENT_USER\Pegasus\Zones:

CeLogZoneCE = 0x00010000 (CELZONE_BOOT_TIME)
```

After you have enabled and configured CeLog, build your OS image, attach your device to your development computer, and then download the image.

Processing CeLog Output

When your build is complete and the image has been downloaded to your device, CeLog uses KITL to create a log file, **bootlog.clg**, in your device release directory. This log file is not user-readable, so you must post-process the file by using the **Readlog** tool.

To process the log file to a user-readable text file

1. Right-click your OS design and then click a **Release Directory in Build Window**.
2. At the command prompt, type **Readlog bootlog.clg bootlog.txt**.
3. When **Readlog** is finished running, open the **bootlog.txt** file.

The following bootlog.txt file shows a sample boot process for a device:

```
0:00:15.015.193 : nk.exe: Launching filesys.exe
0:00:15.106.055 : filesys.exe: Object store initalized
...
0:00:16.699.839 : filesys.exe: File system initialized
0:00:16.727.518 : filesys.exe: Launching CeLogFlush.exe
0:00:16.779.136 : filesys.exe: Launching shell.exe
0:00:17.091.626 : filesys.exe: Launching device.exe
0:00:18.038.225 : device.exe: Activating Drivers\BuiltIn
0:00:18.126.507 : device.exe: Activating Drivers\BuiltIn\notify
0:00:19.887.368 : device.exe: Activating Drivers\BuiltIn\WaveDev
0:00:19.960.906 : device.exe: Activating Drivers\BuiltIn\NDISUIO
0:00:19.986.069 : device.exe: Activating Drivers\BuiltIn\autoras
0:00:19.998.966 : device.exe: Activating Drivers\BuiltIn\NdisPower
0:00:20.011.025 : device.exe: Activating Drivers\BuiltIn\Ethman
...
0:00:20.115.738 : device.exe: Finished initializing
...
0:00:21.731.785 : filesys.exe: PSLs notified of system started
```

The log displays the following information:

- The time that the module or process began loading
- The boot process that loaded the module or process
- The name of the module or process that was loaded

For example, at 18.038.225, the `device.exe` process started loading the `BuiltIn` driver:

```
0:00:18.038.225 : device.exe: Activating Drivers\BuiltIn
|                 |           |_ Module being loaded/initialized
|                 |_____________ Boot Process loading the module
|_____________________________ Timestamp that the module started loading
```

Evaluating Boot Sequence and Boot Time

Use the information in the boot log to analyze the load time for each module and to identify problems in the boot sequence. After you identify latencies in the boot sequence, use the log to verify that modules load in the correct sequence to support operation dependencies. If you identify a problem, you can use the Device Manager (device.exe) registry to set the dependent driver to load at the correct step of the boot sequence.

For example, the preceding sample log shows that the \BuiltIn\notify driver starts loading at 0:00:18.126.507 and that the following driver begins loading at 0:00:19.887.368. Compared to the other drivers, the notify driver takes significantly longer to load. If the notify driver has a dependency on the Ethman driver, which loads later, this might be the cause of the notify driver loading delay. You can set the **Order** sub-key in the **HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Ethman** and the **HKEY_LOCAL_MACHINE\Drivers\BuiltIn\notify** registry entries to ensure that the Ethman driver loads before the notify driver.

> **Note** Depending on your system architecture, you might have the option to use other methods to enforce the boot sequence. For more information about Device Manager, see Device Manager (http://go.microsoft.com/fwlink/?LinkId=220019).

After you generate the initial log that details boot events and sequence, expand CeLog zones captured at boot time to further analyze boot time performance. The following table describes the zones relevant to boot time that CeLog supports and the events that each zone tracks:

Zone	Types of events tracked
CELZONE_THREAD	Thread events, except for thread switch events, which are located in CELZONE_RESCHEDULE
CELZONE_PROCESS	Process events
CELZONE_RESCHEDULE	Scheduler and thread switch events
CELZONE_SYNCH	Synchronization events, including event, mutex, and semaphore operations, and calls to **WaitForMultipleObjects**
CELZONE_CRITSECT	Critical sections events
CELZONE_MIGRATE	Events for migration of threads between processes

For detailed descriptions of the CeLog zones, see CeLog Zones (http://go.microsoft.com/fwlink/?LinkId=220020).

Optimizing the Boot Loader

You typically use the boot loader to place the run-time image into memory and then begin the OS start routine. In addition, you can use a boot loader to quickly download a new run-time image to a device from your development computer. The boot loader is usually located in nonvolatile storage on a device and executes when the system starts or resets.

The following boot loader factors affect performance:

- Device CPU and hardware characteristics
- The process you use to copy your image to RAM by using **KernelRelocate**
- Kernel and OAL initialization
- File system initialization

CPU and Hardware Characteristics

During device start, the boot loader initializes the hardware associated with your device before it launches the OS. To minimize boot time, remove any nonessential hardware initialization from the boot process.

Before the boot loader transfers control to the kernel, it calls the StartUp routine to set the CPU to an initialized state. Depending on the hardware you use, you might need to configure the boot loader to perform additional tasks to make the CPU ready for the OS to access. In addition, you might need to set initializations in the boot loader that are specific to your BSP or CPU.

For some hardware platforms, you can remove the boot loader altogether from the final image to reduce the boot time. When the system is reset, the reset process bootstraps the run-time image, which is stored on the device. However, you need to keep the boot loader to perform pre-boot tasks such as image updates or if are using an x86 platform or other platform that does not support the reset process. For more information about whether your platform will support the reset process, refer to your platform's documentation.

Using KernelRelocate

Use the **KernelRelocate** function to move writeable data sections, such as global variables, to RAM. If you declare global RAM variables—not constants—with initial values instead of **KernelRelocate**, these initial values are stored in the ROM image along with a pointer to where the variable exists in RAM. When the boot loader loads an OS image, it must move these initial values out of ROM and into corresponding RAM locations. If you use **KernelRelocate**, the boot loader does not need to move the values to RAM during startup or set uninitialized global variables to 0 (zero).

KernelRelocate uses the following basic syntax:

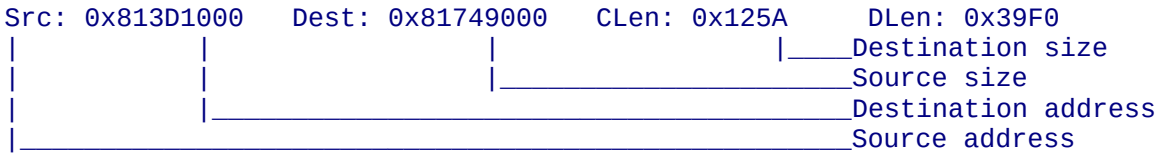
```
static BOOL KernelRelocate (ROMHDR *const pTOC)
```

For more information about **KernelRelocate** and the full syntax, see the blcommon.c file in the %winceroot%\platform\common\src\common\boot\blcommon\ directory.

Use the **Viewbin** tool to view the contents of the image and see how **KernelRelocate** affects boot time. In the following example, COPY Sections indicate where variables are

copied during boot and the difference between the source size (CLen) and destination size (DLen). The difference in size is zero-filled.

```
VIEWBIN –t nk.bin

Image Start = 0x00070000, length = 0x0001634C
...
Extensions          : 0x00000000


COPY Sections ---------------------------------

    Src: 0x813D1000   Dest: 0x81749000   CLen: 0x125A     DLen: 0x39F0
    Src: 0x80A35998   Dest: 0x81753000   CLen: 0x665      DLen: 0x49568
    Src: 0x80B72BC4   Dest: 0x8179F000   CLen: 0x42D      DLen: 0x4568
    Src: 0x814D639C   Dest: 0x817A7000   CLen: 0xC7538    DLen: 0xC7538


MODULES ---------------------------------------

10/13/2008  16:46:11       80896  nk.exe


FILES -----------------------------------------

Done.
```



To minimize the boot time, copy only the required variables during the boot process and then use **KernelRelocate** to move the remaining variables. Analyze the information in the **Viewbin** output to see which variables are copied and if you can remove them from the boot process.

Kernel and OAL Initialization

In addition to optimizing the boot loader, you can make changes to the kernel and OAL to minimize OS initialization and start time.

After **StartUp** completes its tasks, the boot loader calls the kernel initialization function. Depending on your platform, the boot loader will call either **KernelInitialize** (for x86 platforms) or **KernelStart** (for all other platforms). The kernel initialization function calls additional OAL functions, schedules threads, and completes tasks that are specific to each class of CPU.

- To reduce overall boot time, you can optimize kernel and OAL initialization in the following ways: Avoid calling operations that block on I/O or perform large memory operations in **OEMInit** for OAL initialization steps.
- Avoid duplicating operations in the kernel and OAL that were performed in the boot loader operation.

- Verify that the paging pool size is optimally set. For additional information about the paging pool, see Tuning the Paging Pool.
- Avoid adding an IOCTL_HAL_POSTINIT callback to the OAL during initialization.

File System Initialization

The kernel initializes a file system during the boot process that the OS will use after it starts. Several file system initialization operations affect boot time, including:

- Initializing the RAM file system and RAM registry.
- Mounting a HIVE-based registry from persistent storage. This requires the registry to wait for Storage Manager and Device Manager to initialize, which adds time to the boot process.
- Launching OS components and applications by using the **HKey_Local_Machine\Init** registry key. Load any components and applications started in the **HKey_local_Machine\Init** key as late in the boot process as possible. Ensure that they start in the appropriate order so that all the dependencies run correctly.

Initializing the RAM File System

When you initialize the file system, set the size of your RAM file system by using FSRAMPERCENT in the CONFIG.BIB file. FSRAMPERCENT allocates memory to the object store, which is persistent, nonvolatile RAM where the file system is stored. The minimum value of FSRAMPERCENT 0x00000004, or 32 KB. The default value is 0x80808080, which specifies 50 percent of available RAM is allocated for the file system.

To calculate the value of FSRAMPERCENT from the percentage of RAM you want the file system to use, do the following calculation. For example, if you want to reserve 12.5 percent of RAM for the file system, set each byte of FSRAMPERCENT to ((12.5/100)*(1 MB/4 KB)), or 12.5 * 2.56. This example works out to 32, or 0x20. Then, set the 4-byte value of FSRAMPERCENT to 0x20202020.

Optionally, call the **OEMCalcFSPages** function in the OAL to change how much memory to allocate to the object store for the file system. This function specifies the number of pages; the maximum number of pages you can have is 32,000 (128 MB/4 KB). To implement **OEMCalcFSPages**, the *pfnCalcFSPages* member of OEMGLOBAL must point to this function.

Improving File Access Performance

When using a disk-based file system, disk fragmentation can cause critical performance issues. Large partition or small cluster sizes with significant fragmentation can cause the system to scan the disk for free sectors multiple times to write even small files. Do the following to decrease the fragmentation risk and improve performance:

- Increase the cluster size
- Decrease partition size
- Avoid small file writes and deletes

Improving Boot Loader Performance

Do the following to help improve the performance of your boot loader:

- Avoid unnecessary RAM clearing during the boot process.
- Use a binary ROM image file system (BINFS) instead of a single BIN image. The BinFS is a file system that reads the binary image file format that Romimage.exe generates. BinFS parses the .bin file records of each region and reduces the time it takes **KernelRelocate** to copy the variables.
- Remove unnecessary items from your image. This will also reduce the time it takes **KernelRelocate** to copy the variables and reduce the time it takes the image to load. For more information, see Optimizing the OS and Run-Time Image.
- Use a HIVE-based registry instead of ObjectStore. For more details about implementing the registry, see Kernel and OAL Initialization.
- To improve user experience, add a splash screen or small text string to display during boot for devices with a display. This indicates to the user that the device is starting.

Optimizing the Driver Loader

Depending on when the OS needs a particular device driver, you can set a device driver to load automatically during system startup (synchronously) or on demand after system startup (asynchronously).

Although the Bus Enumerator (BusEnum) driver typically loads drivers synchronously, it also supports asynchronous driver loading. Therefore, you can configure threads to load drivers asynchronously, so that the main BusEnum thread can continue loading required drivers while the new thread is delayed. To prevent the drivers from timing out while waiting for dependencies, define BusEnum to synchronize the driver loading threads to support any dependencies between drivers. To further reduce boot time, group drivers under a bus key to be loaded by another instance of BusEnum.

The following driver issues increase load time:

- Initialization functions waiting for hardware initialization or other driver dependencies
- Drivers that time out waiting for hardware that is not present
- Large memory operations that display drivers, or other drivers loaded by GWES, execute during startup (for example, drawing complex interfaces or loading large DLLs)

For more information on driver loading, see Device Manager Functions (http://go.microsoft.com/fwlink/?LinkId=220021)

Minimizing Driver Load Time

Do the following to reduce driver load time:

- Remove drivers that support hardware not currently connected to the device. This will prevent the start process from waiting for the driver initialization to time out.
- Remove drivers that are not required at boot time. Load them only when needed by using ActivateDeviceEx or by using a separate instance of BusEnum.

- Eliminate redundant operations have already finished running in the boot loader or OAL.
- Modify initialization functions to delay slow tasks by using the **IOCTL** value in registry keys to get a Post-Init callback, and then use a thread to finish initialization.

Conclusion

In Windows Embedded Compact 7, optimizing your OS design and increasing performance is an end-to-end process and there are a number of ways to track and optimize performance. Begin by designing and building your OS with the proper configurations and settings. Make sure that your memory is properly allocated for your device and your files are in the appropriate sections of the configuration files. By optimizing the boot process and making sure your system components initialize properly, and configuring how your drivers are loaded, you can create a more efficient and effective OS and embedded device.

Additional Resources

Windows Embedded website (http://go.microsoft.com/fwlink/?LinkID=203338)

Config.bib File (http://go.microsoft.com/fwlink/?LinkId=220015)

DIRS File (http://go.microsoft.com/fwlink/?LinkId=220016)

File System (.dat) Files (http://go.microsoft.com/fwlink/?LinkId=220017)

NKPagePoolParameters (http://go.microsoft.com/fwlink/?LinkId=220018)

Device Manager (http://go.microsoft.com/fwlink/?LinkId=220019)

CeLog Zones (http://go.microsoft.com/fwlink/?LinkId=220020)

Device Manager Functions (http://go.microsoft.com/fwlink/?LinkId=220021)

Copyright