

# Symbian OS support for writeable static data in DLLs

Hamish Willee

Published by the Symbian Developer Network

Version: 2.3 – January 2008

<b>1 PURPOSE AND SCOPE</b> .....	<b>2</b>
<b>2 INTRODUCTION</b> .....	<b>2</b>
2.1 What is global writeable static data? .....	2
2.2 Support for global writeable static data on Symbian OS.....	3
2.3 Alternatives to using global writeable static data .....	3
<b>3 ENABLING GLOBAL WRITEABLE STATIC DATA</b> .....	<b>4</b>
<b>4 WSD: COSTS AND CAVEATS</b> .....	<b>4</b>
4.1 EKA2 emulator only allows a DLL with WSD to load into a single process .....	4
4.2 RAM usage for WSD data chunk .....	5
4.3 Chunks are a finite resource on ARM v5 .....	5
4.4 ARM v4 & v5 architecture specific costs and limitations .....	5
4.5 A few specific DLLs cannot have WSD.....	6
4.6 Limit on the number of DLLs in a process with WSD.....	6
<b>5 FREQUENTLY ASKED QUESTIONS</b> .....	<b>6</b>
5.1 How does Symbian’s WSD implementation work .....	6
5.2 Why does the EKA2 emulator only allow a DLL with WSD to load into a single process? .....	7
5.3 What happens if 'epocallowdlldata' isn't declared for a DLL with WSD? .....	8
5.4 Can Kernel DLLs have WSD.....	8
5.6 Support for emulator WSD .....	9
<b>6 SUMMARY</b> .....	<b>10</b>
<b>7 FURTHER INFORMATION</b> .....	<b>11</b>
7.1 References.....	11
7.2 Glossary .....	11

# 1 Purpose and Scope

Symbian's Real-Time Kernel (EKA2) introduced support for DLL global writeable static data (WSD).

Global writeable static data in DLLs has potentially significant costs in terms of RAM and chunk resource usage, and creates tight constraints on what can be done with the Symbian OS emulator. Symbian therefore recommend that static data should only be used when its costs and limitations are well understood.

ISV developers may find the costs of WSD acceptable, particularly when porting code from other less resource-constrained operating systems. This is particularly true when the DLL to be ported includes significant amounts of WSD, or needs to be loaded by only a single process. Note that this is a change from previous Symbian recommendations, in which ISV developers were strongly discouraged from using WSD.

For developers working in device creation (Symbian, UI platform developers, and handset manufacturers) the tradeoffs are different; in this segment the costs and constraints mean that using global data in DLLs is unlikely to be justifiable.

This document includes a definition of WSD, the alternatives to using WSD (section 2), and the costs and caveats (section 4). Section 5 includes some explanations of our implementation, in the form of frequently asked questions.

Note that the currently supported version of GCC-E<sup>1</sup> has a defect such that DLLs with static data may cause a panic during loading. This issue, and workarounds, is discussed in Symbian FAQ1574 [R4].

## 2 Introduction

### 2.1 What is global writeable static data?

Global writeable static data (WSD) is any per-process variable which exists for the lifetime of the process. In practice, this means any globally scoped data - data that is declared outside of a function, struct, or class, and function scoped static variables.

```
TBufC<20> fileName; //WSD
void SetFileName()
{
    static TInt iCount; //WSD
    ...
}
```

A common miscomprehension is that "const" global variables are read-only data rather than global writeable static data [R1]. This is true for const objects with trivial constructors. However if a const class has a non-trivial constructor, the const object will require a real variable and must be stored as WSD. For example:

```
const TInt myVariable=...; //OK - truly const
const TPtrC KSomeConstPtr=...; //NOT OK - non trivial constructor
const TRgb KSomeConstCol=...; //NOT OK - non trivial constructor
```

---

<sup>1</sup> gcc version 3.4.3 (release) (CodeSourcery ARM Q1C 2005)

## 2.2 Support for global writable static data on Symbian OS

Symbian OS supports global writable static data in EXEs on all versions and handsets.

The EKA2 kernel introduced support for WSD in user-side DLLs on target hardware. EKA1 does not, and will never, support global WSD in DLLs.

The mapping between kernel, Symbian OS version, and support for global writable static data is summarised in the table below.

Kernel	Symbian OS version	DLL	EXE
<b>EKA1</b>	Symbian OS v6.1, v7.0, v7.0s, v8.0a v8.1a	No	Yes
<b>EKA2</b>	Symbian OS v8.0b, v8.1b, v9.x	Yes	Yes

Note that kernel side DLLs may use writable static data in all versions of Symbian OS, as discussed in section 5.4.

## 2.3 Alternatives to using global writable static data

Symbian OS C++ “platform” code rarely uses WSD.

However code ported from other operating systems may contain large amounts of static data. For example, code written in the C programming language often makes use of WSD as the “glue” between C function calls.

In EKA1, WSD is not supported, so there is no choice but to use the alternative mechanisms provided by Symbian OS to port such code. Even in EKA2 where global data is supported, there are costs that may make using WSD unviable – for example, WSD may prevent your DLL being testable on the emulator.

The following sections describe the alternatives that can be used to port code that makes use of global writable static data.

### 2.3.1 Use thread-local storage (TLS)

Thread Local Storage (TLS) is a single per-thread word that can be used to simulate global writable static data.

All the static data in the DLL is grouped into a single struct or class. On creation of the thread, an instance of the thread is allocated on the heap and a pointer to this data is saved to TLS (using `Dll::SetTls()`). On destruction of the thread the data is destroyed. Throughout the DLL the code references the TLS data (using `Dll::Tls()`) rather than the original global writable static data. How this is done is described in detail in [FAQ-0852: “How can I use Thread Local Storage to maintain static data in a DLL?”](#)..

It is also possible to implement process-wide writable global data using TLS. For more information see [FAQ-1089: “Process local storage: How can I use TLS to implement writable global data which is shared between threads in the same process?”](#).

### 2.3.2 Wrap in a server

Symbian OS supports writable global static data in EXEs.

A common porting strategy is therefore to wrap the code in a Symbian server (which is an EXE), and expose its API as a client interface.

Note that it is also possible to use a server to make global data available to all threads at the cost of context switch to the server. This approach is discussed in [FAQ-0938 "Can I use writeable static data in a multi-threaded context in Symbian OS?"](#).

### 2.3.3 Move global variables into your classes

With relatively small amounts of code, it may be possible to move most global data inside classes.

### 2.3.4 Build the library as a static library

WSD in EXEs is supported on all versions of Symbian OS. Therefore, where the code is only intended to be used directly by EXEs, you may be able to build it as a static LIB rather than as a DLL.

This approach can't be used in a framework plug-in, or any other case where a DLL is a requirement of the architecture. Nor can it be used where the library is itself to be loaded by a DLL.

If the DLL code is likely to be shared by multiple processes it's important to consider the potential code size and RAM usage costs of this approach.

## 3 Enabling global writeable static data

In order to enable global writeable static data on EKA2, simply add the EPOCALLOWDLLDATA (case insensitive) to your MMP file:

```
TARGET                my.dll
TARGETTYPE            dll
EPOCALLOWDLLDATA
...
```

## 4 WSD: Costs and caveats

### 4.1 EKA2 emulator only allows a DLL with WSD to load into a single process

The Symbian OS EKA2 emulator only allows a DLL with WSD to be loaded into a single process. The reasons for this limitation are discussed in section 5.2.

This is a very serious restriction. If you have a shared DLL with WSD, then the second process that attempts to load it in the emulator will fail with `KErrNotSupported`.

For reasons described in section 5.3.1, the emulator will allow WSD in DLLs even if EPOCALLOWDLLDATA is not declared in the MMP file. However the data will be truly global – there will be one copy for the entire emulator rather than one copy for each emulated process. The only restriction is that if the data's initialisers call any Symbian OS kernel functions (i.e. executive calls) the emulator will fault.

Symbian OS v9.4 introduces emulator-specific support for process-wide WSD through the "EWSD" library. This support allows developers to load a DLL with WSD into multiple processes on the emulator, at the cost of potentially significant additional development effort. A brief overview of the EWSD support, and the costs, is provided in section 5.5.

## 4.2 RAM usage for WSD data chunk

When a process loads its first DLL containing WSD it creates a single chunk to store the data. The data for subsequent WSD-enabled DLLs is loaded into the same chunk.

The data chunk will occupy at least one 4K RAM page (the smallest possible RAM allocation), irrespective of how much static data is required. Any memory not used for WSD will be wasted. Since the memory is per-process, the memory wastage on the machine is:

$$(4\text{Kb} - \text{WSD Bytes}) \times \text{number-client-processes}$$

It is very easy for a developer to add a few words of WSD to their DLL thinking that's all the memory that they are using. However the cost is actually 4K for every process if a DLL with WSD has not already been loaded into the process. If for example the DLL is used by 4 processes, that's potentially an "invisible" cost of 16K.

For an ISV application developer porting/writing DLLs with a single client process the (approximately) 4Kb wasted memory may be justifiable.

For Device Creators, where much of the code written is shared DLLs, using WSD has proportionally greater costs. The largest possible amount of wastage is 400Kb - calculated by assuming that there are 100 processes running on the device, each loading a single DLL with 1 byte of WSD.

## 4.3 Chunks are a finite resource on ARM v5

Every process loading WSD enabled DLLs uses a chunk to hold the data. On ARM v6, there is no chunk limit, and this is not an issue.

However on ARM v5, EKA2 has a hard coded limit of 16 chunks-per-process; a limit that is required to ensure real-time behaviour. Programs using more than 16 or more chunks need to be aware that one will already have been used for WSD.

## 4.4 ARM v4 & v5 architecture specific costs and limitations

There are other significant costs that apply only to DLLs that link against "fixed processes". Fixed processes are a feature of ARM v4 or v5 architecture only; the following behaviour does not apply to devices based on the ARM v6 architecture.

### 4.4.1 Non Execute-in-place DLLs

For non-execute-in-place (non XIP) DLLs an additional code chunk is required for every fixed process which links against the DLL.

Imagine a 20Kb DLL (with a few bytes of WSD) that is loaded into four normal "non-fixed" processes, and two fixed processes. The (static) memory consumed is:

$$\begin{aligned} \text{Code chunk shared by all moving processes} &= 20 \text{ Kb} \\ \text{Code chunk for each fixed process loading the DLL} &= 40 \text{ Kb} \\ \text{Data chunk for each process loading the DLL} &= 6 \times 4\text{Kb} = 24 \text{ Kb} \end{aligned}$$

So to allow a few bytes of WSD there is a 64Kb increase in consumed memory! Note that the 20Kb code chunk shared by all processes is consumed whether or not WSD is enabled.

### 4.4.2 Execute-in-place DLLs

For XIP DLLs there is no additional RAM cost other than the size of the WSD itself, rounded up to the next multiple of 4KB.

However:

- An XIP DLL can be loaded by any non-fixed process, OR
- it may be loaded by a single fixed process (and therefore cannot be loaded by any other processes whatsoever)

The ROM build fails if a DLL with static data links to a fixed process and any other process.

#### **4.5 A few specific DLLs cannot have WSD**

DLLs that are required to initialise the file server cannot have WSD, e.g. HAL.DLL, EUSER.DLL, EFSRV.DLL.

#### **4.6 Limit on the number of DLLs in a process with WSD**

The number of DLLs with WSD in a process is limited only by the address space available for the writeable static data.

Fixed processes on the moving model (ARM v5 architecture) have an address space 1MB for WSD. Moving processes on the moving model, and all processes on the multiple model, have an address space of  $\text{MIN}(\text{RAM\_SIZE}/2, 128\text{MB})$ , where RAM\_SIZE is the total size of RAM on the device.

## **5 Frequently asked questions**

### **5.1 How does Symbian's WSD implementation work**

#### **5.1.1 For moving processes**

Symbian OS supports a moving memory model, in which the data for a process is moved into a fixed virtual address space (the "run section") when the process is active, and then back into the process's "home" section when another process is active:

- The data of processes in the "home section" is uniquely addressed both in terms of physical RAM pages, and in the MMU virtual address space. The data is protected from other processes, except for the Kernel.
- The data of processes in the run section occupies the same virtual address space as all other processes when they run (the MMU moves the physical RAM pages into the appropriate virtual address space).

When a DLL with WSD is loaded (or at ROM build time), its code is fixed to point to static data at specific addresses. In order to ensure that only a single copy of the DLL code is required, Symbian's moving-process WSD implementation ensures that the virtual addresses of static data is the same across every running process.

The way this works is:

1. A specific address space is reserved for a per-process static data chunk. This chunk is to be used to hold all the static data for all the DLLs loaded into the process.
2. At ROM build time, the kernel reserves specific addresses within the static data chunk for all the WSD in all the ROM loaded DLLs. The addresses for ROM-based DLLs are reserved from the top of the static data chunk address space to the bottom. Note that static data addresses for RAM based DLLs are reserved when the DLL is first loaded

into any process. In this case, addresses are reserved from the bottom of the static data chunk address space.

3. When the first DLL with WSD is loaded into a process, a static data chunk is created to hold the static data for all DLLs that are loaded into the process.
4. Any global static data in the DLL is written to its specific reserved addresses. Note that addresses are reserved for that particular static data across all processes; if the DLL is loaded into another process, any static data will get the same virtual address.

### 5.1.2 For fixed processes

A fixed process is one in which the process data does not move; code is run on process data stored in the process “home section”.

Since the static data for every fixed process is uniquely addressed, and a DLL can only point to a single address for its data, the implication is that a separate copy of the DLL code is required for every fixed process that loads the DLL.

The restrictions listed in section 4.4 directly result.

- For XIP based devices the DLL code chunk address is fixed at ROM build time, and there can only be one copy of the DLL code. Therefore the DLL code can address the data in either a single fixed process or the virtual address used by all moveable addresses.
- For non-XIP based devices the DLL is run from RAM, and the loader is able to fix-up the address that a DLL expects its data at load time (rather than ROM build time). Therefore in this case the loader creates a separate copy of DLL code for each fixed process that loads the DLL and a single copy shared by all moving processes.

#### Notes:

Fixed processes are not supported or required on ARM v6 architectures. This discussion only applies to devices based on ARM v4 or v5. On ARM v6 architectures, each DLL with WSD has a reserved address, similar to the ARM v5 moving process case. However there is no ‘home section’ and memory is not relocated between low and high addresses on a context switch. Instead, each process uses its own set of page tables for the bottom half of virtual address space.

## 5.2 Why does the EKA2 emulator only allow a DLL with WSD to load into a single process?

Symbian OS permits a DLL with global data to be loaded into only one Symbian OS process. This limitation is a result of the way the emulator is implemented on top of the Windows process model.

On EKA2, separate Symbian OS processes are emulated within a single Windows process. To preserve Symbian OS semantics there should be one copy of the global data for each emulated process. However this is not possible since a DLL on the emulator is just a Windows DLL; Windows gives it a single copy of any global data it needs when it is loaded.

## 5.3 What happens if 'epocallowdlldata' isn't declared for a DLL with WSD?

### 5.3.1 On the EKA2 emulator

Most constant data should be treated by the compiler as read-only rather than writeable static data (the exception is when the const data has a non-trivial constructor, so a real variable is required during initialisation).

Unfortunately, different compilers sometimes treat const data as WSD. For example, CodeWarrior puts it in writeable data and initialises it at run time. MSVC generally puts it into read-only data, but occasionally puts it into writeable data.

As most DLLs have const data, this means that the compilers have “accidentally” created WSD in almost every DLL. Symbian cannot therefore rigorously enforce the “single process can load a DLL with WSD” rule, as the emulator would not work.

On the EKA2 emulator, the workaround Symbian has implemented is to recognise two types of DLL global data:

- 'Deliberate' global data is where the programmer specifies that they want DLL global data (using the `epocallowdlldata` keyword in the MMP file. In this case any global data in the DLL is assumed to be deliberate, and the “DLL loaded into one-emulated-process” rule applies.
- 'Accidental' global data is the data introduced by the compiler with no encouragement from the programmer. If `epocallowdlldata` is absent, global data is assumed to be accidental and the rule does not apply. Note that the global data includes both const and non-const static data; there is no way to tell the compilers to only apply it to non-const data – if we could do that then we could force correct handling of const static data.

In order to prevent abuse of this workaround there are restrictions on what can be done with accidental global data; specifically, the emulator will fault if any of the global data's initialisers attempt to call Symbian OS kernel functions (i.e. executive calls).

Note that there is only one copy of the global data. Therefore it is possible for two processes to write to the same ‘accidental’ global data (causing undefined behaviour). The “DLL loaded into one-emulated-process” rule prevents this being a problem for deliberate global data.

### 5.3.2 On the EKA1 emulator

The EKA1 emulator has no concept of separate Symbian OS processes. Global data is allowed on the emulator as there is only one copy of any global data.

Developers often have problems when porting to real hardware, which does not support this type of data. This is discussed in the following section.

### 5.3.3 On real hardware

On either EKA1 or EKA2, the compiler will fail the build with an error indicating that the code has initialised or un-initialised data. See [FAQ-0329: "How can I find the "uninitialised data" in my DLL?"](#) for information on how to locate the cause of these errors.

## 5.4 Can Kernel DLLs have WSD

Yes, WSD is supported for kernel DLLs in both EKA1 and EKA2 (through alternative mechanisms to those described in this paper)

Of course, kernel DLLs are guaranteed to be loaded into only one process, so the per-process multiplication of RAM usage does not apply. EKA2 will work correctly with global data in any kernel DLL. However EKA1 does not call constructors for global C++ objects in kernel extensions or device drivers, and does not call destructors for global C++ objects in device drivers at driver unload time.

See [R2] for more information.

## 5.5 Support for emulator WSD

Symbian OS v9.4 provides emulator WSD (EWSD), a mechanism to allow DLLs with WSD to be able to load into multiple processes.

The support is not transparent to the DLL writer; they have to wrap the WSD in a data structure and pass it to the EWSD library. This library maintains the relationship between the library, and its WSD values in each process.

The DLL writer has to make emulator-conditional changes to the source code to:

- wrap all the WSD in the DLL into a single “wrapper” data structure
- change code to refer to the wrapper data member rather than directly
- prevent EPOCALLOWDLLDATA being declared in the emulator build

For example, consider a DLL with a function `foo()` that uses WSD `iState` as shown:

```
// source for foo.cpp
int iState;

void foo()
{
    if (iState == ESomeState)
    {
        //do something
    }
    else
    {
        //do something else
    }
}
```

You would change as shown:

```
// source for foo.cpp
struct MyWSD
{
int iState;
};
void foo()
{
#ifdef _USE_EWSD_
    MyWSD *MyWSDPtr = PIs<MyWSD>(ThisLibUID);
    // PIs is an API provided by the ewsd - it fetches the ptr for this
    // (process, libuid) tuple.
    // You can also pass an initialisation function to PIs() to initialise the WSD
    // - initialisation is only done the 1st time the templated function is called
#endif
```

```
if (MyWSDPtr->iState == ESomeState)
{
    //do something
}
else
{
    //do something else
}
}
```

The MMP file of that DLL must not have EPOCALLOWDLLDATA for the emulator build, so you would make its specification conditional on use of EWSD in the MMP file:

```
#ifndef _USE_EWSD_
    EPOCALLOWDLLDATA
#endif
```

If the DLL has a lot of WSD then recoding as above may take significant effort. Note that this recoding is not necessary if your DLL is only to be loaded into a single process, or if you only need to test on target.

## 6 Summary

WSD in DLLs is not, and never will be, supported on EKA1 based versions of Symbian OS. However WSD in DLLs is supported on EKA2.

Support for WSD in DLLs is provided primarily for ISVs porting code with WSD from other operating systems, where the cost of using the other alternatives is prohibitive.

## 7 Further Information

### 7.1 References

No.	Document Reference	Description
[R1]	FAQ-0128 <a href="http://www3.symbian.com/faq.nsf/0/48EC9738F76B1EA480256A570051B9C3?OpenDocument">http://www3.symbian.com/faq.nsf/0/48EC9738F76B1EA480256A570051B9C3?OpenDocument</a>	Don't be misled by const static data that isn't really const
[R2]	FAQ-1105 Extranet-only	Can I have writeable static data in my LDD, PDD, or kernel extension?
[R3]	FAQ-907 <a href="http://www3.symbian.com/faq.nsf/0/71C212DB06DCE71380256D6E005AD2A8?OpenDocument">http://www3.symbian.com/faq.nsf/0/71C212DB06DCE71380256D6E005AD2A8?OpenDocument</a>	Why doesn't Symbian OS support writeable static data (EKA1)
[R4]	FAQ-1574 <a href="http://www3.symbian.com/faq.nsf/AllByDate/B8542F039C193CCC802573DA0011DFA7?OpenDocument">http://www3.symbian.com/faq.nsf/AllByDate/B8542F039C193CCC802573DA0011DFA7?OpenDocument</a>	Global writeable static data in DLLs does not appear to work when projects are compiled with GCC-E. What can I do about this?

### 7.2 Glossary

Term	Definition
Execute in place (XIP)	Execute in place. On an XIP "ROM" the ROM memory is mapped into the process address space. Code is executed directly from the ROM. NOR flash is XIP memory.  Non XIP ROM is not mapped into the process address space. Code is loaded into RAM before being executed.
Chunk	A collection of physical RAM pages mapped to contiguous virtual addresses. In EKA2 a process may have only 8 chunks.

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.