O S R
open
systems
resources
inc.

DMK ARCHITECTURE

A Reference

ARCHITECTURAL REFERENCE

The function of the DMK is to provide the infrastructure necessary to allow arbitrary data modification algorithms to function on top of the existing Windows file systems layers. Traditionally, complex (size changing) modifications are implemented directly within the file system so the storage allocation policy can be handled within the underlying file system. For example, in Windows, the NTFS solution has been to directly manage the allocation "runs" of the file so it allocates sufficient disk space to store the data contents using 64KB as the "magic number" for compression. This makes sense in the NTFS model because it controls that allocation policy. The choice of 64KB fits well with the 16-bit LZW compression algorithm since this is implemented within the Windows OS. [1]However, when implemented as a layer outside the file system, the challenge is to manage the allocation in a file system independent fashion. The file system has its own allocation mechanism and provides no visibility into that mechanism.

The DMK provides a layer of functionality on top of existing file systems to greatly simplify the implementation of a variety of complex size-changing data modification components. We refer to this technique as a **layered file system** or **stackable file system**. In Windows it is common to implement this using a **file system filter driver**. However, due to the nature of integration between file systems and the Windows virtual memory system, a normal file system filter driver is not sufficient to easily implement functionality that requires direct control of the file system data cache.

For this reason, the DMK is implemented as a set of cooperative drivers, which are described below.

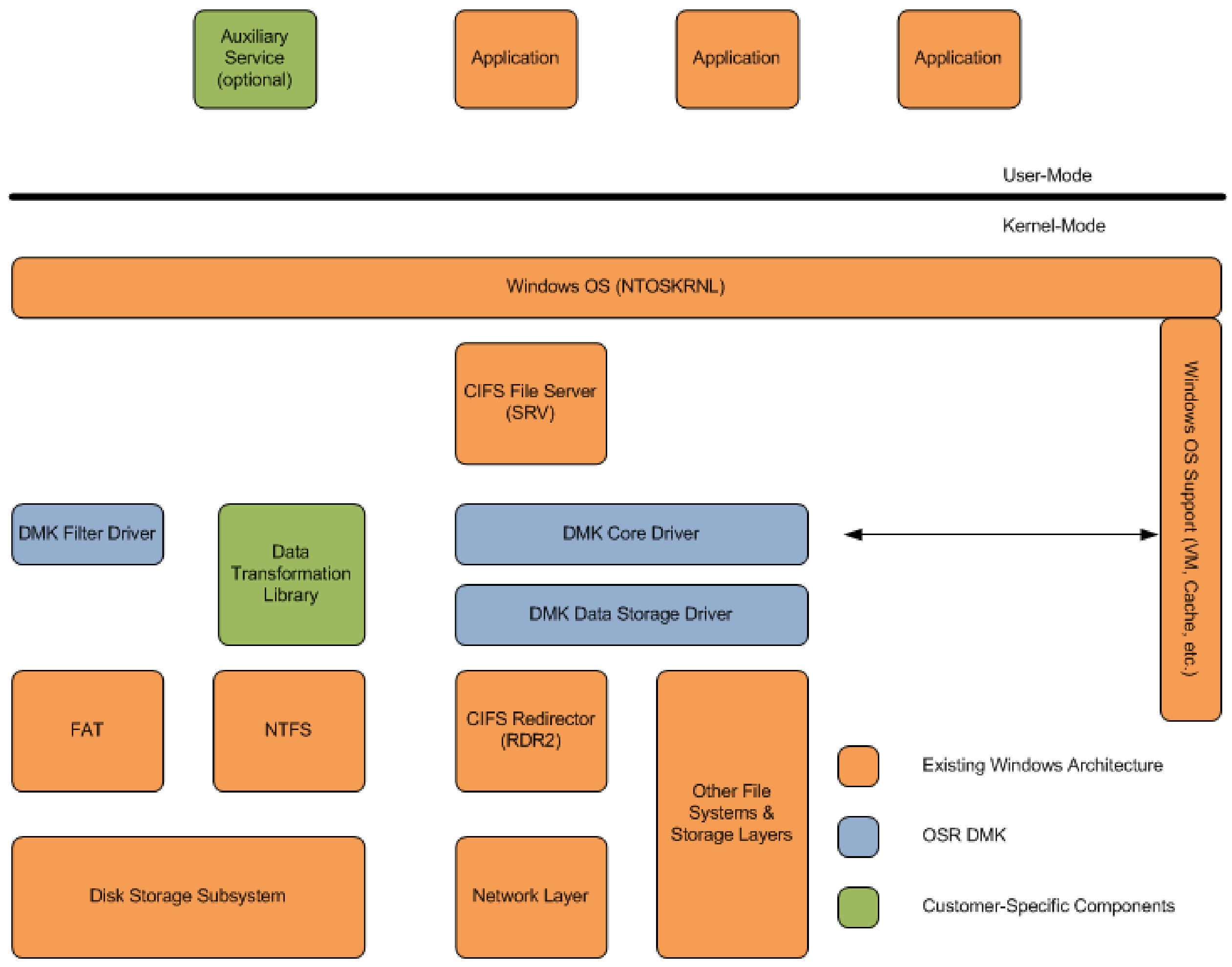
- A **Filter Driver** that is responsible for analyzing operations against the file system to determine if the operation is of interest to the DMK. If it is not of interest, then no further processing is required for the lifetime of that open file instance. If it is of interest, the file open operation is redirected to the DMK Core Driver. We note that there are, in fact, some operations that are performed in-line within the filter driver itself (e.g., certain self-contained stateless operations.)
- **File system driver:** Referred to as the DMK Core Driver. This driver is responsible for providing cache management. It is also primarily responsible for interacting with the DT library driver. Since DMK 1.1 we have sent virtually all operations from the mini-filter to the file system driver. The exception here is for specific cases that must be done against the native file system for correct operation (sadly, these are often found on a case-by-case basis, as there is no documentation from Microsoft to guide our discovery of these special cases.)
- **Data Transformation (DT) library driver:** Used to implement unique data modification algorithms (e.g., encryption and compression) to files that are of interest as well as the policy used to determine which files should be converted into the DMK's native storage format (DS/LFS) as well as which files should be transformed. In our experience, it is the

[1] Realistically, though, we would expect customers seriously interested in high efficiency compression to utilize a more modern algorithm (e.g., BWT.) The LZW implementation is simple to use – and low overhead – but does not provide the best level of general purpose compression.

policy that the DT library implements that creates substantial complication – both in terms of simply making those decisions as well as impacting the performance of the overall system.

- **Data Storage (DS) driver:** Responsible for managing the details of the structure of the information as required by the DMK and DT Library.[2] We provide two alternative implementations of this layer, but the interface is sufficiently general that it is possible to implement other storage layers. This allows the core DMK to remain the same even while the on-disk format used by the transformation system changes.

The following diagram provides a simple conceptual model.



[2] For example, we currently have two implementations of this driver. One works only on NTFS and utilizes the NTFS stream facility (DS/NTFS). The other works with any file system, but uses a structured storage technique referred to as *log structuring* in order to allow management of the complex data needed by the DMK to implement its functionality (DS/LFS).

The three components shown in blue are provided as part of the DMK package. The components shown in green are normally written by a customer to implement their unique data modification functionality. However, the DMK package includes sample DT libraries to facilitate this development. Other components of the system are also shown (in orange) including two key sources of I/O traffic: 1) application programs; and 2) the CIFS File Server, i.e., the Lan Manager Server. The Lan Manager Server is normally referred to as the CIFS Server or “SRV” because the driver that implements it has traditionally been called srv.sys.

The structure of the DMK has been developed to focus the kit on the specific needs of products that wish to provide arbitrary data modifications. One of the key aspects is the ability to provide transparent operation. To the extent possible, users of the system with a DMK-based driver should not observe any differences in operation when the DMK is present.

We have designed the DMK to work with essentially any known file system provided with Windows. Our design has focused on NTFS, FAT, and CIFS Client (redirector) because these are of predominate interest. Specific features that we have worked to support include:

- Support for arbitrary data modification algorithms. In this arena, we consider data compression algorithms to be the “litmus test.”[3]
- Support for transmission of modified files between the client and server system, so that data is transmitted over the network in its modified form.[4]
- Support for access to both raw (transformed) and clear (untransformed) data.
- Support for both read/write and memory mapped access mechanisms.
- Support for additional security and access checks.
- Independence on the features of the underlying file system. The DMK handles the specific details of the format for the underlying data. The representation in the underlying storage appears as a flat file.
- Ability for the external transformation library to work with an integral data size (within some reasonable limit). For example, this is important for allowing support for CBC mode encryption. This is also useful for optimizing certain compression algorithms such as LZW.

[3] The reason for this is that when updating a portion of the file, the new data may not compress as well as the old data. This is the primary motivation for having the Data Storage layer since it provides the abstraction necessary to support relocating the new data to a different region of the file.

[4] Translation: “Send the file over the network in encrypted and compressed form.”

- Support for the storage of additional meta-data. Correspondingly, it will also be possible to support storing alternate data streams, extended attributes, reparse point tags, symbolic links, etc.[5]

To accomplish this, the goal is primarily for supporting both compression and encryption algorithms. The DMK does not provide these mechanisms, but rather relies on an external interface to a customer-provided library for accomplishing these transformations. An important competitive advantage of the DMK is the ability to support compression in addition to encryption, which is a unique feature that is not present in existing encryption products. Even NTFS supports compression *or* encryption, but not both. By providing support for both, our goal is to provide a compelling reason for companies to license the DMK technology.

The items listed below are explicitly **not** goals for the DMK:

1. **Encryption of directory contents** - We do not provide native support for encrypting individual file names or the contents of directories. If this is necessary, we suggest either a modified (custom) DS layer implementation or a disk level filter.
2. **Encryption of the paging file** - This restriction is a consideration primarily due to the restrictions on paging file modifications. We expect that supporting this (in a more restricted fashion) will be a goal for a future release (although with special DT callbacks, reflective of the restrictions inherent in encrypting the paging file.)
3. **Encryption of crash dumps** - Crash dumps bypass the storage stack and write directly to disk. Modification of the data written to a crash dump is not supported by the DMK. If you require this functionality, we can provide some additional insight into the potential issues (there is no OS level support for this functionality in all supported Windows versions.)
4. **Encryption of hibernation files** - The hibernation process bypasses the storage stack and writes directly to disk. Modification of the data written to a hibernation file is not supported by the DMK. If you require this functionality, we can provide additional insight into potential issues (there is no OS level support for this functionality in all supported Windows versions.)
5. **Encryption of boot drivers** - The DMK is not available in the bootstrap loader, and thus cannot be used to encrypt boot drivers. In addition, we caution you that files essential to proper loading and log-in to the system require that the keys be available at the point of decryption. Otherwise, you will observe incorrect behavior.

[5] While not a goal of the DMK, it takes little imagination to realize that the log structured storage technique we use in the implementation could easily be extended to support NTFS features on non-NTFS file systems. Such an extension model is likely to lead to a product other than those that are the primary objective of the DMK.

6. **Block headers with DS/NTFS –** The implementation of the DS/NTFS component restricts the size of block headers to no larger than the corresponding data blocks.[6] Note that block headers are only available for blocks of non-zero length. Thus, if a file is empty, you cannot have a data block header.
7. **Sum of all individual stream sizes in one file** - Limited to approx 1EB. Note that the underlying system may enforce more stringent size requirements.[7]

These are the specific restrictions of which we are aware as of DMK Version 1.2 release. As we gain further operational experience, we will update this list to include any additional restrictions and remove restrictions that have been resolved.

[6] In other words, if the data block size is 64KB, the block header cannot be larger than 64KB. The actual data contained within the data block can be smaller than the data block size (e.g., in the case of compression.) This restriction is NTFS specific.

[7] For example, FAT has a limitation that files must be less than 4GB in size. Some versions of Windows limit the size of a volume to 16TB.