Demystifying the DMG File Format

Jonathan Levin, http://newosxbook.com/ - 6/12/13

1. About

As part of writing HFSleuth, a "bonus" tool for my book, I decided to implement DMG (disk image support). I realized, however, that the DMG file format (being Apple proprietary) was woefully undocumented. I briefly mention DMGs (pages 589-590), but due to the page constraints of an already large book, I had failed to delve into their format sufficiently. This article, therefore, is an attempt to rectify that shortcoming. The DMG file format has been painstakingly reverse-engineered by several[1,2], and this article/addendum aims to consolidate their hard work into a single document. HFSleuth can operate fully on all known DMG types (to date), and can serve as a complementary tool to Apple's hdiutil(1), or - as it is POSIX portable - even as a replacement for it, on non OS X systems. When set to verbose mode, HFSleuth also provides step by step information as it processes DMGs, and is used in the examples below.

2. The Disk Image file format

The first noteable fact about the DMG file format is, that there is no DMG file format. DMGs come in a variety of sub-formats, corresponding to the different tools which create them, and their compression schemes. The common denominator of most of these is the existence of a 512-byte trailer at the end of the file. This trailer is identifiable by a magic 32-bit value, 0x6B6F6C79, which is "koly" in ASCII. As other references to this trailer call it the "koly" block, we can do the same. Note, that "most" is not "all": images created with hdiutil(1), for example, can simply be raw dd(1)-like images of the disk layout, with no metadata. In those cases, however, there is nothing special or noteworthy about the file, which can be read as any disk would, by its partition table (commonly APM, or GPT). Images created with the DiscRecording.Framework contain the koly block. The koly block, when present, is formatted according to the following:

```
typedef struct {
        char     Signature[4];           // Magic ('koly')
        uint32_t Version;                // Current version is 4
        uint32_t HeaderSize;             // sizeof(this), always 512
        uint32_t Flags;                  // Flags
        uint64_t RunningDataForkOffset;  //
        uint64_t DataForkOffset;         // Data fork offset (usually 0, beginning of file)
        uint64_t DataForkLength;         // Size of data fork (usually up to the XMLOffset, be
        uint64_t RsrcForkOffset;         // Resource fork offset, if any
        uint64_t RsrcForkLength;         // Resource fork length, if any
        uint32_t SegmentNumber;          // Usually 1, may be 0
        uint32_t SegmentCount;           // Usually 1, may be 0
        uuid_t   SegmentID;              // 128-bit GUID identifier of segment (if SegmentNuml

        uint32_t DataChecksumType;       // Data fork
        uint32_t DataChecksumSize;       //  Checksum Information
        uint32_t DataChecksum[32];       // Up to 128-bytes (32 x 4) of checksum

        uint64_t XMLOffset;              // Offset of property list in DMG, from beginning
        uint64_t XMLLength;              // Length of property list
        uint8_t  Reserved1[120];         // 120 reserved bytes - zeroed

        uint32_t ChecksumType;           // Master
        uint32_t ChecksumSize;           //  Checksum information
        uint32_t Checksum[32];           // Up to 128-bytes (32 x 4) of checksum

        uint32_t ImageVariant;           // Commonly 1
        uint64_t SectorCount;            // Size of DMG when expanded, in sectors

        uint32_t reserved2;              // 0
        uint32_t reserved3;              // 0
        uint32_t reserved4;              // 0

} __attribute__((__packed__)) UDIFResourceFile;
```

Listing 1: The koly block format

Note: All fields in the koly block (and, in fact, elsewhere in the DMG format) are in big endian ordering. This is to preserve compatibility with older generations of OS X, which were PPC-based. This requires DMG implementations to use macros such as be##_to_cpu (16, 32, and 64).

The most important elements in the koly block are the fields pointing to the XML plist: This property list, embedded elsewhere in the DMG, contains the DMG block map table. Commonly, the plist is placed in the blocks leading up to the koly block, which fits the simple algorithm to create a DMG: First compress the image blocks, then place the XML plist, and finalize with the koly block. This is shown in figure 1:

The Data Fork: Disk blocks, compressed in various ways ...
XML property list (variable)
koly trailer (512 bytes)

Using HFSleuth in verbose mode on a DMG will dump the KOLY header, as shown in the following output:

```
HFSleuth> ver
Verbose output is on
HFSleuth> fs iTunes11.dmg
KOLY header found at 200363895:
        UDIF version 4, Header Size: 512
        Flags:1
        Rsrc fork: None
        Data fork: from 0, spanning 200307220 bytes
        XML plist: from 200307220, spanning 56675 bytes (to 200363895)
        Segment #: 1, Count: 1
        Segment UUID: 626f726e-7743259b-6086eb93-4b42fb65
        Running Data fork offset 0
        Sectors: 1022244
```

Output 1: Using HFSleuth on the iTunes 11.0 DMG

This method of creating DMGs also explains why commands such as "file" have a hard time identifying the DMG file type: In the absence of a fixed header, a DMG can start with any type of data (disk or partition headers), which can be further compressed by myriad means. DMG files compressed with BZlib, for example, start with a BZ2 header. They cannot be opened with bunzip2, however, since compression methods are intermingled, and bunzip2 will discard blocks which do not start with a bz2 header.

```
root@Erudite (/tmp)# file DMG/install_flash_player_osx.dmg
DMG/install_flash_player_osx.dmg: bzip2 compressed data, block size = 100k
root@Erudite (/tmp)# hdiutil imageinfo DMG/install_flash_player_osx.dmg | grep Format
Format Description: UDIF read-only compressed (bzip2)
Format: UDBZ
```

Output 2: a BZ2-compressed DMG

DMGs compressed with zlib often incorrectly appear as "VAX COFF", due to the zlib header.

```
root@Erudite (/tmp)# file DMG/xcode46.dmg

DMG/xcode46.dmg: VAX COFF executable not stripped - version 376
root@Erudite (/tmp)# hdiutil imageinfo DMG/xcode46.dmg | grep Format
Format Description: UDIF read-only compressed (zlib)
Format: UDZO
```

Output 3: a zLib-compressed DMG

The XML Property list (which is uncompressed and easily viewable by seeking to the DOCTYPE declaration using more(1) or using tail(1)) is technically the resource fork of the DMG. The property list file contains, at a minimum, a "blkx" key, though it may contain other key/values, most commonly "plst", and sometimes a service level agreement (SLA) which will be displayed by the OS (specifically, /System/Library/PrivateFrameworks/DiskImages.framework/Versions/A/Resources/DiskImages UI Agent.app/Contents/MacOS/DiskImages UI Agent) as a pre-requisite to attaching the DMG[*]. Due to XML parser restrictions, data in the property list is 7-bit. This forces all binary (8-bit) data to be encoded using Base-64 encoding (a wiser choice would have been using CDATA blocks). The output of such a property list is shown below:

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
<plist version="1.0">
<dict>
        <key>resource-fork</key>
        <dict>
                <key>blkx</key>
                <array>
                        <dict>
                                <key>Attributes</key>
                                <string>0x0050</string>
                                <key>CFName</key>
                                <string>Driver Descriptor Map (DDM : 0)</string>
                                <key>Data</key>
                                <data>
                                bWlzaAAAAAEAAAAAAAAAAAAAAAAAAAABAAAAAAAAAAAA
                                AAIB/////wAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                AAIAAAAg6P2c0P9/AAAwGb9f/38AAAAAAAAAAAAAAwBi/
                                X/9/AAAoAAAAAAAAAAAAAAAAAAAAAKAFAAEAAAAYAAAA
                                AAAAACAkv1//fwAA8Bi/X/9/AACKWZeI/38AACAZv1//
                                fwAAKAAAAAAAAAAAAAAAAAAAAAFhAAgEBAAAAABm/X/9/
                                AAAAAAACgAAABgAAAAAAAAAAAAAAAAAAAAAAAAAABAAAA
                                AAAAAAAAAAAAAAAAOP////8AAAAAAAAAAAAAAAEAAAAA
                                AAAAAAAAAAAAAAA4AAAAAAAAAAA=
                                </data>
                                <key>ID</key>
                                <string>-1</string>
                                <key>Name</key>
                                <string>Driver Descriptor Map (DDM : 0)</string>
                        </dict>
                        <dict>
                                <key>Attributes</key>
                                <string>0x0050</string>
                                <key>CFName</key>
                                <string>Apple (Apple_partition_map : 1)</string>
                                <key>Data</key>
                         <data>
                                bWlzaAAAAAEAAAAAAAAAAQAAAAAAAAAA/AAAAAAAAAAAA
                                AAIBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                AAIAAAAgk1UtLv9/AAAwGb9f/38AAAAAAAAAAAAAAwBi/
                                X/9/AAAoAAAAAAAAAAAAAAAAAAAAAKAFAAEAAAAYAAAA
                                AAAAACAkv1//fwAA8Bi/X/9/AACKWZeI/38AACAZv1//
                                fwAAKAAAAAAAAAAAAAAAAAAAAAFhAAgEBAAAAABm/X/9/
                                AAAAAAACgAAABgAAAAAAAAAAAAAAAAAAAAAAAAAA/AAAA
                                AAAAADgAAAAAAAAAAu/////8AAAAAAAAAAAAAAD8AAAAA
                                AAAAAAAAAAAAAADzAAAAAAAAAAA=
                                </data>
                                <key>ID</key>
                                <string>0</string>
                                <key>Name</key>
                                <string>Apple (Apple_partition_map : 1)</string>
                        </dict>
                        <dict>
                                <key>Attributes</key>
                                <string>0x0050</string>
                                <key>CFName</key>
                                <string>DiscRecording 5.0.9d2 (Apple_HFS : 2)</string>
                                <key>Data</key>
                                <data>
                                bWlzaAAAAAEAAAAAAAAAAQAAAAAAAAIRcAAAAAAAAAAAAA
                                AAQEAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                AAIAAAAgbcaqTv9/AAAwGb9f/38AAAAAAAAAAAAAAwBi/
                                X/9/AAAoAAAAAAAAAAAAAAAAAAAAAKAFAAEAAAAYAAAA

                                ...
```

Listing 2: Sample XML plist in an APM formatted DMG (Flash installer)

A detailed discussion of both APM and GPT can be found in chapter 15 of the book[3], as well as Apple's notes on APM[4] and GPT[5]. What makes the blxx data useful, however, is that it allows an implementation to *skip past* the partition table data, and isolate the partition of interest directly from the DMG. The "data" in the blxx header is a structure, which (like its sibling, koly) is also identifiable by a fixed signature - in this case "mish". In Base-64 this encodes as "bWlza", which is readily evident in the previous listing. The mish block is formatted like this:

```
typedef struct {
        uint32_t Signature;          // Magic ('mish')
        uint32_t Version;            // Current version is 1
```

```
        uint64_t SectorNumber;        // Starting disk sector in this blkx descriptor
        uint64_t SectorCount;         // Number of disk sectors in this blkx descriptor

        uint64_t DataOffset;
        uint32_t BuffersNeeded;
        uint32_t BlockDescriptors;    // Number of descriptors

        uint32_t reserved1;
        uint32_t reserved2;
        uint32_t reserved3;
        uint32_t reserved4;
        uint32_t reserved5;
        uint32_t reserved6;

        UDIFChecksum checksum;

        uint32_t NumberOfBlockChunks;
        BLKXChunkEntry [0];
} __attribute__((__packed__)) BLKXTable;

// Where each  BLXKRunEntry is defined as follows:

typedef struct {
        uint32_t EntryType;          // Compression type used or entry type (see next table)
        uint32_t Comment;            // "+beg" or "+end", if EntryType is comment (0x7FFFFFFE
        uint64_t SectorNumber;       // Start sector of this chunk
        uint64_t SectorCount;        // Number of sectors in this chunk
        uint64_t CompressedOffset;   // Start of chunk in data fork
        uint64_t CompressedLength;   // Count of bytes of chunk, in data fork
} __attribute__((__packed__)) BLKXChunkEntry;
```

Listing 3: The mish block format

In other words, for each entry, the chunk of *SectorCount* sectors, starting at *SectorNumber* are stored at *CompressedLength* bytes, at offset *CompressedOffset* in the data fork. When expanded, each such chunk will take *SectorCount* * *SECTOR_SIZE* bytes. Each chunk of blocks in a given entry is stored using the same compression, but different entries can contain different compression methods.

> Question: What are two advantages of breaking the image into block chunks, as described above? (Answer at end of document)

The various block chunk entry types are shown below:

Table: DMG blxx types

Type	Scheme	Meaning
0x00000000	---	Zero-Fill
0x00000001	UDRW/UDRO	RAW or NULL compression (uncompressed)
0x00000002	---	Ignored/unknown
0x80000004	UDCO	Apple Data Compression (ADC)
0x80000005	UDZO	zLib data compression
0x80000006	UDBZ	bz2lib data compression
0x7ffffffe	---	No blocks - Comment: +beg and +end
0xffffffff	---	No blocks - Identifies last blxx entry

Running HFSleuth on a DMG in verbose and debug mode will produce detailed output of the decompression, demonstrating the above:

```
..
Decompressing 0x345 blocks, Desc 4
1022160 sectors - 523345920 bytes
Blk 0 - 0 (512 sectors) - Compressed to 397, 1407 bytes Zlib
Blk 1 - 512 (512 sectors) - Compressed to 1804, 1167 bytes Zlib
Blk 2 - 1024 (512 sectors) - Compressed to 2971, 1167 bytes Zlib
Blk 3 - 1536 (512 sectors) - Compressed to 4138, 1167 bytes Zlib
Blk 4 - 2048 (512 sectors) - Compressed to 5305, 1167 bytes Zlib
Blk 5 - 2560 (512 sectors) - Compressed to 6472, 1167 bytes Zlib
Blk 6 - 3072 (512 sectors) - Compressed to 7639, 1167 bytes Zlib
Blk 7 - 3584 (512 sectors) - Compressed to 8806, 1167 bytes Zlib
Blk 8 - 4096 (512 sectors) - Compressed to 9973, 1167 bytes Zlib
...
Blk 39 - 19968 (512 sectors) - Compressed to 46202, 1167 bytes Zlib
```

```
Blk 40 - 20480 (216 sectors) - Compressed to 47369, 506 bytes Zlib
Blk 41 - 20696 (103200 sectors) - Compressed to 47875, 0 bytes IGNORE # zeroed/unused
sectors
Blk 42 - 123896 (512 sectors) - Compressed to 47875, 7904 bytes Zlib
Blk 43 - 124408 (512 sectors) - Compressed to 55779, 1167 bytes Zlib
...
Blk 828 - 1019624 (512 sectors) - Compressed to 199079734, 250792 bytes Zlib
Blk 829 - 1020136 (512 sectors) - Compressed to 199330526, 262144 bytes RAW # Note 262,144
= 512 * 512
Blk 830 - 1020648 (512 sectors) - Compressed to 199592670, 230554 bytes Zlib
Blk 831 - 1021160 (512 sectors) - Compressed to 199823224, 238101 bytes Zlib
Blk 832 - 1021672 (480 sectors) - Compressed to 200061325, 245760 bytes RAW # Note 245,760
= 480 * 512
Blk 833 - 1022152 (6 sectors) - Compressed to 200307085, 0 bytes IGNORE
Blk 834 - 1022158 (1 sectors) - Compressed to 200307085, 135 bytes Zlib
Blk 835 - 1022159 (1 sectors) - Compressed to 200307220, 0 bytes IGNORE
Blk 836 - 1022160 (0 sectors) - Compressed to 200307220, 0 bytes Terminator
decompression done
```

Output 4: Decompressing a DMG image in HFSleuth, debug mode

Note in the example above the mix of Zlib and RAW compression methods: Zlib uses highly efficient compression algorithms, but sometimes it just makes sense to leave data in raw form (e.g. chunks 829 and 832). In these cases, the "compressed" size is actually the same as the uncompressed size. It's also worth noting that (though it is commonly the case) there is no guarantee that the blocks are compressed in order.

3. Mounting DMGs

DMGs can be mounted, just like any other file system, though technically this is what is known as a "loopback" mount (i.e. a mount backed by a local file, rather than a device file). To mount a DMG, the system uses the DiskImages kernel extension (KExt), also known as the IOHDIXController.kext. This is clearly visible in both OS X and iOS, using kextstat (or jkextstat, in the latter):

```
Index Refs Address            Size       Wired      Name (Version) >Linked Against<
    1   77 0xffffff7f80756000 0x686c     0x686c     com.apple.kpi.bsd (12.2.0)
    2    6 0xffffff7f80741000 0x46c      0x46c      com.apple.kpi.dsep (12.2.0)
    3  101 0xffffff7f80760000 0x1b7ec    0x1b7ec    com.apple.kpi.iokit (12.2.0)
    4  106 0xffffff7f8074c000 0x99f8     0x99f8     com.apple.kpi.libkern (12.2.0)
    5   92 0xffffff7f80742000 0x88c      0x88c      com.apple.kpi.mach (12.2.0)
    6   39 0xffffff7f80743000 0x500c     0x500c     com.apple.kpi.private (12.2.0)
    7   60 0xffffff7f80749000 0x23cc     0x23cc     com.apple.kpi.unsupported (12.2.0)
    8    0 0xffffff7f8146e000 0x41000    0x41000    com.apple.kec.corecrypto (1.0) >7 6 5 4 3 1<
    9   22 0xffffff7f80d44000 0x9000     0x9000     com.apple.iokit.IOACPIFamily (1.4) >7 6 4 3<
   10   30 0xffffff7f8088d000 0x25000    0x25000    com.apple.iokit.IOPCIFamily (2.7.2) >7 6 5 4 3<
   11    2 0xffffff7f81dbf000 0x57000    0x57000    com.apple.driver.AppleACPIPlatform (1.6) >10 9 7 6 5 4 3
   12    1 0xffffff7f80a9e000 0xe000     0xe000     com.apple.driver.AppleKeyStore (28.21) >7 6 5 4 3 1<
   13    6 0xffffff7f8077c000 0x25000    0x25000    com.apple.iokit.IOStorageFamily (1.8) >7 6 5 4 3 1<
   14    0 0xffffff7f80e4d000 0x19000    0x19000    com.apple.driver.DiskImages (344) >13 7 6 5 4 3 1<
...
```

The kext is provided with a number of "PlugIn" kexts, namely:

- AppleDiskImagesCryptoEncoding.kext
- AppleDiskImagesKernelBacked.kext
- AppleDiskImagesReadWriteDiskImage.kext - for UDRO/UDRW
- AppleDiskImagesFileBackingStore.kext
- AppleDiskImagesPartitionBackingStore.kext - Uses the Apple GUID 444D4700-0000-11AA-AA11-00306543ECAC
- AppleDiskImagesSparseDiskImage.kext - for UDSP
- AppleDiskImagesHTTPBackingStore.kext - Allows DMGs to reside on a remote HTTP server. Uses a "KDISocket" with HTTP/1.1 partial GETs (206) to get the chunks it needs from a DMG
- AppleDiskImagesRAMBackingStore.kext
- AppleDiskImagesUDIFDiskImage.kext

The attachment of a DMG starts in user mode, by an I/O Kit call to IOHDIXController, preparing a dictionary with the following keys:

- hdik-unique-identifier - A UUID created by the caller (e.g. CFUUIDCreate())
- image-path - the path to the DMG in question

Some types of disk images (sparse, uncompressed, and z-Lib compressed) are natively supported by the kernel and can be mounted directly by it. A good example is the DeveloperDiskImage.dmg found in the iOS SDK. More often than not, however, mounting resorts to user-mode helper processes. This, in fact, is default on OS X (q.v. hdiutil -nokernel vs. hdiutil -kernel). When attaching a DMG, the DiskImages private framework spawns diskimages-helper and hdiejectd. The former is started with a -uuid argument per invocation, allowing the mounting of the same DMG multiple times. If the process is stopped, filesystem operations on its contents will likewise hang (with the exception of those already cached by VFS). You can demonstrate this with a simple experiment by mounting a DMG, sending the corresponding diskimages helper a STOP signal, and performing a filesystem intensive operation, such as an ls -lR, witnessing the hang, then sending a CONT. Further inspection in GDB with a breakpoint on mach_msg will enable you to peek at the Mach messages which are passed between the process and the kernel over the I/O Kit interface. This will show a backtrace similar to:

```
#0 0x00007fff8cfd4c0d in mach_msg ()                 # Actual message passer
#1 0x00007fff887e3fbc in io_connect_method ()        # I/O Kit internal connect
#2 0x00007fff887978ea in IOConnectCallMethod ()      # I/O kit connector, generic argument
#3 0x00007fff88797ae8 in                             # I/O Kit connector, with structure
IOConnectCallStructMethod ()                         argument
#4 0x00007fff86e5b79f in DI_kextDriveGetRequest      # DiskImages framework function
()
```

Looking at the arguments to DI_kextDriveGetRequest (specifically, $rdx+ 0x20), will reveal a pointer to the data returned from the DMG file by the diskimages-helper.

```
(gdb) x/20x $rdx # Note this is the $rdx value at a breakpoint on DI_kextDriveGetRequest
0x103e68c30: 0x4279726f 0x00000001 0x00040208 0x00000000
0x103e68c40: 0x00000100 0x00000000 0x2dd13e80 0xffffff80
0x103e68c50: 0x03e6b000 0x00000001 0x00200000 0x00000000


(gdb) x/s 0x0000000103e6b000 Data from a cat /Volumes/KernelDebugKit/kgmacros
0x103e6b000: "\n# Kernel gdb macros\n#\n# These gdb macros should be useful during kernel
development in\n#
determining what's going on in the kernel.\n#\n# All the convenience variables used by
these macros begin wit"...
```

Commands and support

Apple provides extensive support for DMGs, which is only natural given their role in everything, from aspects of OS installation to software distribution. The DMG support is provided by the DiskImages project, which contains both the user mode (hdid, hdiutil) and kernel mode (kexts) required for operation. Lamentably, Apple keeps this as one of the non-open source projects in Darwin.

- hdid
- hdiutil
- DiskImages.framework - The private framework lending support to both the above tools, communicating with the KExts (below), as well as the user mode helper processes for mounting images (diskimages-helper and hdiejectd)
- IOHDIXController.kext

Answer: Advantages of using per-block compression, rather than a single compression algorithm for entire file:

1. Optimize compression for type of data: For example, discard blocks of zeros rather than compressing them, or even leaving data uncompressed
2. Allow an implementation to selectively decompress chunks, rather than the whole image, which may take a lot of filesystem space and/or memory (especially in kernel-mode).

References:

1. DMG2IMG: http://vu1tur.eu.org
2. DMG2ISO: at sourceforge.net
3. The book http://www.newosxbook.com
4. APM: [Discussion at informit.com](#)
5. TN2166: Secrets of the GPT [developer.apple.com](#)