MICROSOFT CORPORATION

VHDX Format Specification

Version 1.00

25-August-2012

Table of Contents

List of Structures

List of Figures

List of Tables

1 Introduction

This paper describes the VHDX virtual hard disk format that provides a disk-in-a-file abstraction. This document is written to help guide development of VHDX virtual hard disk format implementations that are compatible with those provided by Microsoft.

This specification assumes that you are familiar with hard disk technologies, including how hard disks interface with the operating system or a virtual machine and understand how data is accessed and laid out on the physical medium.

1.1 Concepts

This specification uses certain disk technology concepts that are detailed below.

Table 1: Concepts

Concept	Explanation
Host volume or disk	The volume or disk on which the virtual hard disk file resides.
Logical sector size	The minimum required alignment of IO to a disk. The disk cannot handle IO that is aligned to an offset that is not a multiple of the logical sector size. The logical sector size for the virtual hard disk and host disk could differ.
Physical sector size	The disk will tend to be efficient in handling IO aligned to an offset and size that is a multiple of the physical sector size. The physical sector size for the virtual hard disk and host disk could differ.
Fixed Virtual Hard Disk Type	A virtual hard disk file that is allocated to the size of the virtual hard disk and does not change when data is added or removed from the virtual hard disk. For example, for a virtual hard disk that is 1 GB in size, the virtual hard disk file is approximately 1 GB and will not grow or shrink in size as data is added or deleted.
Dynamic Virtual Hard Disk Type	A virtual hard disk file that at any given time is as large as the actual data written to it plus the size of its internal metadata. As more data is written, the file dynamically increases in size by allocating more space. For example, for a 2 GB virtual hard disk, the size of the virtual hard disk file initially is around 2 MB. As data is written to this virtual hard disk, it grows in predetermined blocks to a maximum size of 2 GB.
Differencing Virtual Hard Disk Type	A differencing hard disk file represents the current state of the virtual hard disk as a set of modified blocks in comparison to a parent virtual hard disk file. Any new write to the virtual disk is captured in the latest child virtual hard disk. A read to a virtual disk offset is satisfied by looking for that virtual offset on the latest child virtual hard disk and traversing all the way to the parent if needed.

	This mechanism is used to create point in time snapshots of disks for backups and other scenarios. The differencing virtual hard disk to be fully functional file depends on another virtual hard disk file. The parent hard disk file can be any of the mentioned virtual hard disk types, including another differencing virtual hard disk file.
Blocks	Allocation of new space for a virtual hard disk that supports dynamic growth of the virtual hard disk file is done in fixed size units defined as blocks. The size of the blocks can be specified by the user during creation.
BAT	A redirection table called a block allocation table **(BAT)** is used in translation from the virtual hard disk offset to the virtual hard disk file offset.
Sector Bitmap	A structure that stores a Boolean value for each virtual disk sector in the blocks specifying whether that sector is present in a virtual block. Even in the case where a virtual block is present in a differencing virtual hard disk file, the sector bitmap indicates whether a virtual disk sector is present in that virtual block. If it is not present, then the parent virtual hard disk file is inspected for the presence of that sector.
UNMAP	UNMAP is the SCSI command by which an application or the system can communicate to the storage stack and the disk that a certain sector or range of sectors are currently not in use, including sectors that were previously in use by files that were later deleted.

1.2 Common Acronyms

This specification uses certain acronyms commonly used in the computer industry that is detailed below.

Table 2: Acronyms

Acronym	Full Text
ASCII	American Standard Code for Information Interchange
GUID	Globally Unique Identifier (see Section 4.1)
KB	1024 bytes
MB	1024 KB
GB	1024 MB
CRC	Cyclic Redundancy Check
SCSI	Small Computer System Interface

1.3 Specific Terminology

In the context of this specification, certain terms carry specific meaning for implementation of the VHDX

format.

Table 3: Terminology

Term	Definition
Must	This specification uses the term “must” to describe a specific behavior which is mandatory.
Should	This specification uses the term “should” to describe a specific behavior which it strongly recommends, but does not make mandatory.
May	This specification uses the term “may” to describe a specific behavior which is optional.
Mandatory	This term describes a field or structure which an implementation must interpret as this specification describes.
Optional	This term describes a field or structure which an implementation may or may not support. If an implementation supports a given optional field or structure, it must interpret the field or structure as this specification describes.
Reserved	This term describes field or structure contents which implementations: ● Must initialize to zero and should not use for any purpose ● Must not be interpreted or validated to be zero, except when computing checksums ● Must preserve across operations which modify surrounding fields or structure

1.4 Conventions

All multi-byte values must be stored in little endian format with the least significant byte first unless specified otherwise. Bit 0 always means the least significant bit of the least significant byte.

Unless specified otherwise, the CRC used to validate data is CRC-32C (see Section 4.2). This uses the Castagnoli polynomial, code 0x11EDC6F41.

The notation Ceil(X) shall mean the minimum integer that is greater than or equal to X.

The notation Fl(X) shall mean the maximum integer that is lesser than or equal to X.

2 VHDX Format

2.1 Features and Benefits

VHDX format features provide features at the virtual hard disk as well as virtual hard disk file layers and is optimized to work well with modern storage hardware configurations and capabilities.

At the virtual hard disk layer, benefits include the ability to represent a large virtual disk size up to 64 TB, support larger logical sector sizes for a virtual disk up to 4 KB that facilitates the conversion of 4 KB sector physical disks to virtual disks, and support large block sizes for a virtual disk up to 256 MB that enables tuning block size to match the IO patterns of the application or system for optimal performance.

At the virtual hard disk file layer, the benefits include the use of a log to ensure resiliency of the VHDX file to corruptions from system power failure events and a mechanism that allows for small pieces of user generated data to be transported along with the VHDX file.

On modern storage platforms, the benefits include optimal performance on host disks that have physical sector sizes larger than 512 bytes through improved data alignment and capability to use the information from the UNMAP command, sent by the application or system using the virtual hard disk, to optimize the size of the VHDX file.

The format is designed so that additional features could be introduced in the future by Microsoft or extended by other parser implementations. The format provides parsers the ability to detect features in a VHDX file that a parser does not understand.

2.2 Layout

The VHDX file begins with a fixed-sized **header section**. After this, non-overlapping objects and free space are intermixed freely in no particular order; the only restriction being that all objects have 1 MB alignment within the file.

The objects currently defined include the **BAT region** (also referred to as **BAT**), the **metadata region, header, log, payload blocks** and **sector bitmap blocks.** Each of these objects is described in detail in the following sections. The objects can also be moved around in the file as long as they are non-overlapping and the MB alignment is maintained.

VHDX is designed to support 3 types of virtual hard disks; fixed, dynamic and differencing. The logical and physical layout is similar for all 3 types and specific differences will be made clear in the following sections.

Figure 1: Logical Layout

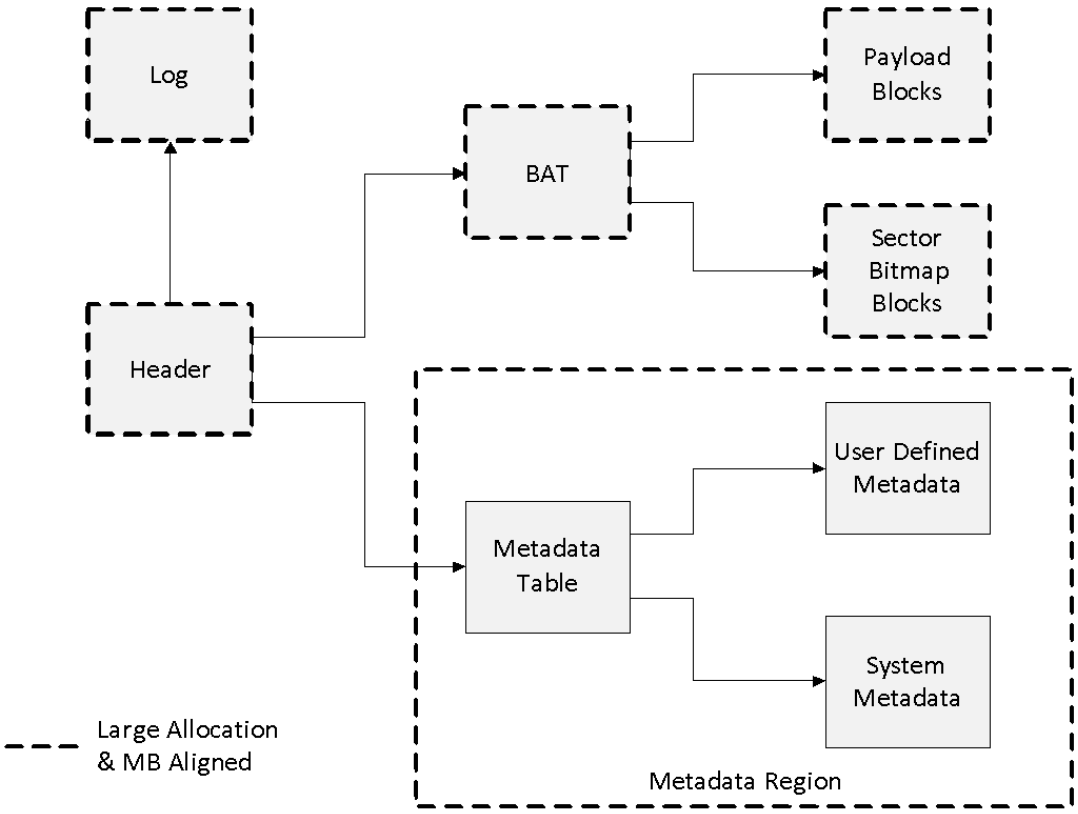
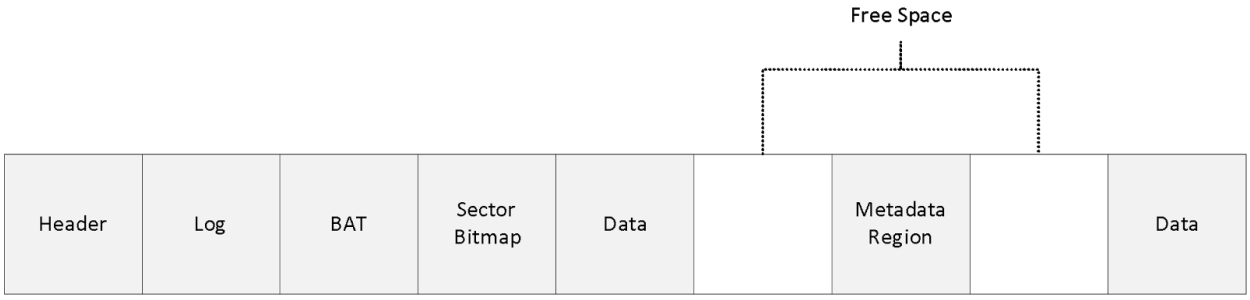


Figure 2: File Layout Example



2.3 Header Section

The header section is the first object on disk and is the structure that is examined first when opening a

VHDX file. The header section is 1 MB in size and contains five items that are 64 KB in size: the **file type identifier**, two **headers**, and two copies of the **region table**.

The file type identifier contains a short, fixed signature to identify the file as a VHDX. It is never overwritten. This ensures that even if a failed write corrupts a sector of the file, the file can still be identified as a VHDX.

Each header acts as a root of the VHDX data structure tree, providing version information, the location and size of the **log**, and some basic file metadata. Beyond that, the header contains as little data as possible; other properties that may be needed to open the file are stored elsewhere in other metadata.

Only one header is active at a time so that the other may be overwritten safely without corrupting the VHDX file. A sequence number and checksum are used to ensure this mechanism is safe.

The region table lists **regions**, which are virtually contiguous, variable-size, MB aligned pieces of data within the file. These objects currently include the **BAT** and the **metadata region**, but can be extended by parsers or future revisions of the specification without breaking compatibility with different implementations and versions of the parsers. Parsers must maintain objects that they don't understand without corrupting them. Parsers must fail to open a VHDX file that contains a region that is marked as required but is not understood by the parser.

2.4 Log

The log is arranged as a variable-sized, contiguous ring buffer and is pointed to by the header. Like a region, it is MB aligned and may reside anywhere after the header section. It consists of variable-sized log entries which contain data that must be rewritten when the file is reopened after an unexpected system failure event (for example, power failure). Each log entry has at least a 4 KB alignment, although the alignment may be greater if necessary to ensure writes are isolated on the host disk storage media.

2.5 Blocks

In addition to the log (pointed to by the header) and the regions (listed in the region table), the VHDX file contains fixed-sized **blocks** that are located by the **BAT**.

There are two types of blocks, **payload blocks** and **sector bitmap blocks**. Payload blocks contain virtually contiguous pieces of the virtual disk. The size of the payload blocks is defined when the VHDX file is created and must be a power of 2 and at least 1 MB and at most 256 MB. Sector bitmap blocks are always 1 MB in size and contain pieces of the sector bitmap. The MB alignment requirement applies to both block types.

Any updates to sector bitmap objects must be made using the log to ensure that the updates are safe to corruptions from system power failure events. Updates to payload data should not be logged, since the VHDX format is not intended to provide reliability guarantees for user data beyond that of a physical disk.

2.6 BAT

The BAT is a region listed in the region table and consists of a single contiguous array of entries specifying the state and the physical file offset for each block. The entries for payload blocks and sector bitmap blocks in the BAT are interleaved at regular intervals.

Any updates to the BAT must be made using the log to ensure that the updates are safe to corruptions from system power failure events.

2.7 Metadata Region

The metadata region is a variable-sized region listed in the region table and contains all the small, infrequently accessed metadata, including system and user metadata. The user metadata provides an extensibility point for parsers to embed parser specific metadata. **Metadata entries** are at most 1 MB in size and byte aligned. They are located by a **metadata table** residing at the beginning of the region.

Any updates to metadata region must be made using the log to ensure that the updates are safe to corruptions from system power failure events.

3 Detailed Specification

3.1 Header Section

The header section contains four items: the **file type identifier**, two **headers**, and the **region table**.

Figure 3: Header Section Layout

3.1.1 File Type Identifier

The file type identifier is a structure stored at offset zero of the file

Structure 1: File Type Identifier

```
struct VHDX_FILE_IDENTIFIER {
      UINT64      Signature;
      UINT16      Creator[256];
};
```

The **Signature** field must be 0x656C696678646876 (“vhdxfile” as ASCII).

The **Creator** field contains a UTF-16 string describing the parser that created the VHDX file. This field may not be null terminated. This field is optional for the parser to fill in during the creation of the VHDX file to identify, uniquely, the creator of the VHDX file. Parsers must not use this field as a mechanism to influence parser behavior; it only exists for diagnostic purposes.

A parser must write File Type Identifier structure when the file is created and must validate the **Signature** field when loading a VHDX file. The parser must not overwrite any data in the first 64 KB of the file after the file has been created.

The space between file identifier data and 64 KB alignment boundary for the file identifier structure is reserved.

3.1.2 Headers

Since the header is used to locate the log, updates to the headers cannot be made through the log. To provide power failure consistency guarantees, there are two headers in every VHDX file.

Each of the two headers is a 4 KB structure that is aligned to 64 KB boundary[1]. One header is stored at offset 64 KB and the other at 128 KB. Only one header is considered current and in use at any point in time.

Structure 2: Header

```
struct VHDX_HEADER {
      UINT32      Signature;
      UINT32      Checksum;
      UINT64      SequenceNumber;
      GUID        FileWriteGuid;
      GUID        DataWriteGuid;
      GUID        LogGuid;
      UINT16      LogVersion;
      UINT16      Version;
      UINT32      LogLength;
      UINT64      LogOffset;
      UINT8       Reserved[4016];
};
```

The **Signature** field must be 0x64616568 (“head” as ASCII).

The **Checksum** field is a CRC-32C hash over the entire 4 KB structure, with the **Checksum** field taking the value of zero during the computation of the checksum value.

The **SequenceNumber** field is a 64-bit unsigned integer.

A header is **valid** if **Signature** and **Checksum** both validate correctly. A header is **current** if it is the only valid header or if it is valid and its **SequenceNumber** field is greater than the other header’s **SequenceNumber** field. The parser must only use data from the current header. If there is no current header, then the VHDX file is corrupt.

The **FileWriteGuid** field specifies a 128-bit unique identifier that identifies the file’s contents. On every open of a VHDX file, a parser must change this GUID to a new and unique identifier before the first modification is made to the file, including system and user metadata as well as log playback[2]. The parser can skip updating this field if the storage media on which the file is stored is read-only, or if the file is opened in read-only mode.

The **DataWriteGuid** field specifies a 128-bit unique identifier that identifies the contents of the user visible data. On every open of the VHDX file, a parser must change this field to a new and unique identifier before the first modification is made to user visible data. If the user of the virtual disk can observe the change through a virtual disk read, then the parser must update this field.[2] This includes changing the system and user metadata, raw block data, disk size or any block state transitions that will

[1] This ensures that isolated updates to the headers can be performed on storage systems with up to a 64 KB sector size. The resiliency guarantees break down if the hosting volume physical sector size beyond 64 KB.
[2]The **DataWriteGuid** field is used in the integrity validation of a differential VHDX chain. Parsers must pay special attention to ensure that they are updated as described.

result in a virtual disk sector read being different from a previous read. Notably, this does not include movement of blocks within a file, because this only changes the physical layout of the file, not the virtual identity.

The **LogGuid** field specifies a 128-bit unique identifier used to determine the validity of log entries. If this field is zero, then the log is empty or has no valid entries and must not be replayed. Otherwise, only log entries that contain this identifier in their header are valid log entries. Upon open, the parser must update this field to a new non-zero value before overwriting existing space within the log region.

The **LogVersion** field specifies the version of the log format used within the VHDX file. This field must be set to zero. If it is not, the parser must not continue to parse the file unless the LogGuid field is zero, indicating that there is no log to replay.

The **Version** field specifies the version of the VHDX format used within the VHDX file. This field must be set to 1. If it is not, a parser must not attempt to parse the file using the details from this format specification.

The **LogLength** and **LogOffset** fields specify the byte offset in the file and the length of the log. These values must be multiples of 1 MB and **LogOffset** must be at least 1 MB. The log must not overlap any other structures.

The space between 4 KB structure containing header data and 64 KB alignment boundary for the header is reserved.

3.1.2.1 Updating the Headers

On every open of a VHDX file that allows the VHDX file to be written to, the headers must be updated before any other part of the file is modified. While the VHDX file is in use, the headers may also need to be updated when an operation on the virtual disk or the VHDX file impacts one of the fields contained in the header.

The first time the headers are updated after the open of a VHDX file, the parser must generate a new random value for the FileWriteGuid field. The parser should not update the DataWriteGuid field until the user has requested an operation that may affect the virtual disk contents or user-visible virtual disk metadata such as the disk size or sector size.

A parser may follow the following procedure to change the non-current header to the current header:

1. Identify the current header and non-current header.
2. Generate a new header in memory. Set the SequenceNumber field to the current header's SequenceNumber field plus one.
3. Set the other fields to their current values or updated values as desired. If this is the first header update within the session, use a new value for the FileWriteGuid.
4. Checksum the header in memory.
5. Overwrite the non-current header in the file with the in-memory header. Issue a flush command to ensure the header update is stable on the host disk storage media.

After this procedure, the non-current header becomes the current header. The parser should perform the update procedure a second time so that both the current and non-current header contains

up-to-date information; this ensures that if one header is corrupted, the file can still be opened.

3.1.3 Region Table

The region table consists of a header followed by a variable number of entries, which specify the identity and location of regions within the file. There are two copies of the region table stored at file offset 192 KB and 256 KB. Updates to the region table structures must be made through the log.

Structure 3: Region Table Header

```
struct VHDX_REGION_TABLE_HEADER {
      UINT32      Signature;
      UINT32      Checksum;
      UINT32      EntryCount;
      UINT32      Reserved;
};
```

The **Signature** field must be 0x69676572 (“regi” as ASCII).

The **Checksum** field is a CRC-32C hash over the entire 64 KB table, with the **Checksum** field taking the value of zero during the computation of the checksum value.

The **EntryCount** field specifies the number of valid entries to follow. This must be less than or equal to 2047.

Structure 4: Region Table Entry

```
struct VHDX_REGION_TABLE_ENTRY {
      GUID        Guid;
      UINT64      FileOffset;
      UINT32      Length;
      UINT32      Required:1;
      UINT32      Reserved:31;
};
```

The **Guid** field specifies a 128-bit identifier for the object and must be unique within the table.

The **FileOffset** and **Length** fields specify the 64-bit byte offset and 32-bit byte length of the object within the file. The values must be a multiple of 1 MB, and **FileOffset** must be at least 1 MB.

All objects within the table must be non-overlapping, not only with respect to each other but with respect to the log (defined in the headers) and payload and sector bitmap blocks (defined in the BAT).

The **Required** field specifies whether this region must be recognized by the parser in order to load the VHDX file. If this field’s value is 1 and the parser does not recognize this region, the parser must refuse to load the VHDX file.

The space between the last region table entry and 64 KB alignment boundary for the region table is reserved.

The table below summarizes the properties of the regions defined in this version of the specification.

Table 4: Known Region Properties

Known Regions	GUID	IsRequired
BAT	2DC27766-F623-4200-9D64-115E9BFD4A08	True
Metadata region	8B7CA206-4790-4B9A-B8FE-575F050F886E	True

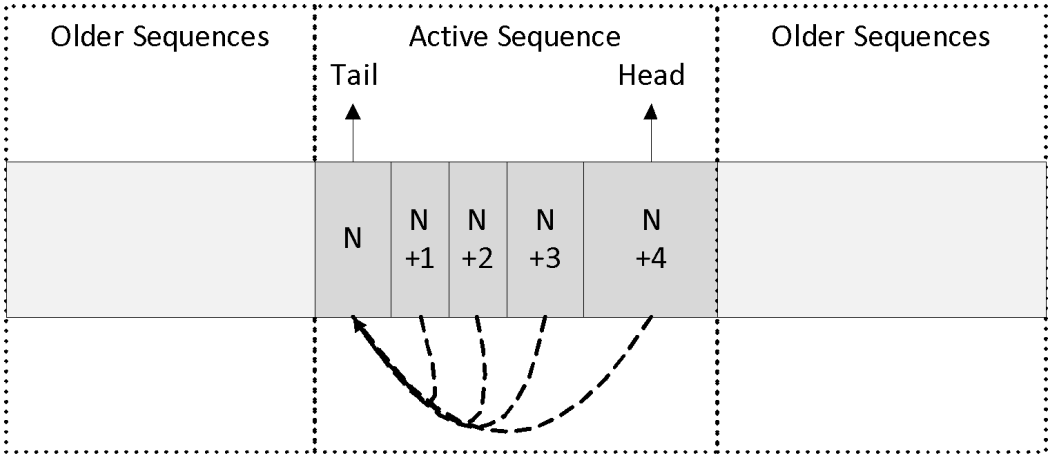
3.2 Log

The log is a single circular buffer stored contiguously at a location that is specified in the VHDX header. It consists of an ordered sequence of variable-sized entries, each of which represents a set of 4 KB sector updates that need to be performed to the VHDX structures.

The log must be used to perform updates to all metadata except the header and must not be used for updates to the payload blocks. Updates to a metadata structure through the log involves a sequence of steps; writing all metadata updates to the log, flush the updates to the log to ensure that they are stable on the host disk storage media, apply the metadata updates to their final location in the VHDX file, and flush these metadata updates to ensure that they are stable on the host disk storage media.

Updates that were not written and flushed to their final location on the VHDX file due to a power failure will be detected, validated and then replayed from the log on a subsequent open of the VHDX file.

Figure 4: Log Layout Example



3.2.1 Log Entry

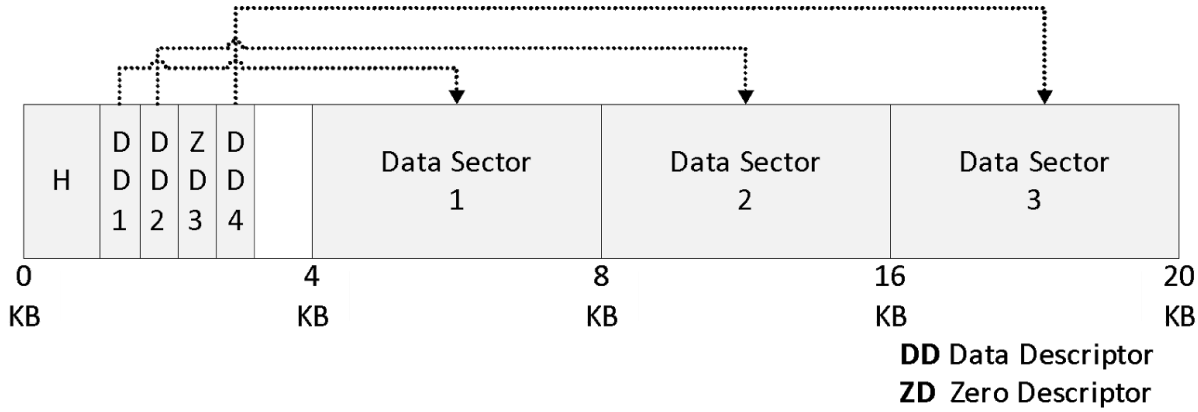
A **log entry** is a sequence of 4 KB sectors, aligned to a 4 KB sector[3]. This sequence is divided into two

[3] Each log entry has at least a 4 KB alignment but may be greater if necessary to ensure writes are isolated on the host disk storage disk. All parsers must be prepared to replay from a log with as small as 4 KB alignment, and no parser is obligated to maintain an alignment greater than 4 KB. A parser running on a large-sector-size disk may need to make a copy of the log, expanding the writes to the larger sector size, before replaying.

parts: a sequence of one or more descriptor sectors, and a sequence of zero or more data sectors.

The descriptor sectors contain header information and describe the writes that are being performed through the log, while the data sectors contain the actual updates. The first descriptor sector contains a 64-byte header and could contain up to 126 32-byte descriptors; any subsequent descriptor sectors that are part of the same log entry does not contain a header and could contain up to 128 32-byte descriptors. The total number of descriptors contained in the log entry is described in the log entry header. Each descriptor could be either a data descriptor or a zero descriptor. Each of the data descriptor describes a 4 KB sector write to perform on the file, and must be associated with a data sector that contains the actual 4 KB update to be written. The number of data sectors in the log entry must be equal to the number of data descriptors contained in that log entry. The zero descriptor describes a section of the file to zero but does not need an associated sector to describe the update as it is implied to be zeroes.

Figure 5: Log Entry Structure Layout Example



The above example describes a log entry that contains 4 descriptors. The layout contains, in the first 4 KB descriptor sector, a log header with a descriptor count of 4 and associated descriptor sectors, 3 data descriptors and 1 zero descriptor. It will be followed by 3 data sectors each of length 4 KB.

A variety of techniques is applied to validate the log entry. The entry header sequence number is duplicated in each descriptor and data sector. In the case of data sector, the sequence number is split between the beginning and end of each data sector. Most cases of torn, missing, or misdirected writes will result in this sequence number validation being incorrect either between the data descriptor and the associated data sector or between the descriptors and the entry header. Furthermore, a 32-bit CRC is computed over the entire log entry, including the sequence number; this improves the probability that a random entry corruption is detected.

The space for the log entries can only be reused for subsequent log writes when the log entries are not part of the active log sequence (See section 3.2.3).

3.2.1.1 Entry Header

Structure 5: Log Entry Header

```
struct VHDX_LOG_ENTRY_HEADER {
      UINT32      Signature;
      UINT32      Checksum;
      UINT32      EntryLength;
      UINT32      Tail;
      UINT64      SequenceNumber;
      UINT32      DescriptorCount;
      UINT32      Reserved;
      GUID        LogGuid;
      UINT64      FlushedFileOffset;
      UINT64      LastFileOffset;
};
```

The **Signature** field must be 0x65676F6C (“loge” as ASCII).

The **Checksum** field is a CRC-32C hash computed over the entire entry specified by the EntryLength field, with the **Checksum** field taking the value of zero during the computation of the checksum value.

The **EntryLength** field specifies the total length of the entry in bytes. The value must be a multiple of 4 KB.

The **Tail** field is the byte offset of the beginning log entry of a sequence ending with this entry. The value must be a multiple of 4 KB. A tail entry could point to itself as would be the case when a log is initialized.

The **SequenceNumber** field is a 64-bit number incremented between each log entry. It must be larger than zero.

The **DescriptorCount** field specifies the number of descriptors that are contained in this log entry. The value may be zero.

The **LogGuid** field contains the **LogGuid** value in the file header that was present when this log entry was written. When replaying, if this **LogGuid** do not match the **LogGuid** field in the file header, this entry must not be considered valid.

The **FlushedFileOffset** field stores the VHDX file size in bytes that must be at least as large as the size of the VHDX file at the time the log entry was written. The file size specified in the log entry must have been stable on the host disk such that, even in the case of a system power failure, a non-corrupted VHDX file will be at least as large as the size specified by the log entry. Before shrinking a file while the log is in use, a parser must write the target size to a log entry and flush the entry so that the update is stable on the log on the host disk storage media; this will ensure that the VHDX file is not treated as truncated during log replay. A parser should write the largest possible value that satisfies these requirements. The value must be a multiple of 1 MB.

The **LastFileOffset** field stores a file size in bytes that all allocated file structures fit into, at the time the log entry was written. A parser should write the smallest possible value that satisfies these

requirements. The value must be a multiple of 1 MB.

3.2.1.2 Zero Descriptor

Structure 6: Log Zero Descriptor

```
struct VHDX_LOG_ZERO_DESCRIPTOR {
      UINT32      ZeroSignature;
      UINT32      Reserved;
      UINT64      ZeroLength;
      UINT64      FileOffset;
      UINT64      SequenceNumber;
};
```

The **ZeroSignature** field must be 0x6F72657A (“zero” as ASCII).

The **ZeroLength** field specifies the length of the section to zero. The value must be a multiple of 4 KB.

The **FileOffset** field specifies the file offset to which zeroes must be written. The value must be a multiple of 4 KB.

The **SequenceNumber** field must match the **SequenceNumber** field of the log entry’s header.

3.2.1.3 Data Descriptor

Structure 7: Log Data Descriptor

```
struct VHDX_LOG_DATA_DESCRIPTOR {
      UINT32      DataSignature;
      UINT32      TrailingBytes;
      UINT64      LeadingBytes;
      UINT64      FileOffset;
      UINT64      SequenceNumber;
};
```

The **DataSignature** field must be 0x63736564 (“desc” as ASCII).

The **TrailingBytes** field contains the 4 trailing bytes that were removed from the update when it was converted to a data sector. These trailing bytes must be restored before the data sector is written to its final location on disk.

The **LeadingBytes** field contains the first 8 bytes that were removed from the update when it was converted to a data sector. These leading bytes must be restored before the data sector is written to its final location on disk.

The **FileOffset** field specifies the file offset to which the data described by this descriptor must be written. The value must be a multiple of 4 KB.

The **SequenceNumber** field must match the **SequenceNumber** field of the entry’s header.

3.2.1.4 Data Sector

Structure 8: Log Data Sector

```
struct VHDX_LOG_DATA_SECTOR {
      UINT32      DataSignature;
      UINT32      SequenceHigh;
      UINT8       Data[4084];
      UINT32      SequenceLow;
};
```

The **DataSignature** field must be 0x61746164 (“data” as ASCII).

The **SequenceHigh** field must contain the 4 most significant bytes of the **SequenceNumber** field of the associated entry.

The **Data** field contains the raw data associated with the update, bytes 8 through 4091, inclusive. Bytes 0 through 7 and 4092 through 4096 are stored in the data descriptor, in the **LeadingBytes** and **TrailingBytes** fields, respectively.

The **SequenceLow** field must contain the 4 least significant bytes of the **SequenceNumber** field of the associated entry.

3.2.2 Log Sequence

At any given moment in the operation of the log, the newest entry in the log (or **head entry**) always points to the oldest entry in the log that has the updates yet to be written and flushed to their final location (**tail entry**). Taken as a snapshot in time, this sequence of entries between the tail entry and head entry are called a **log sequence**. Log sequences are ordered in time by the sequence number of the head entry.

To determine whether a log sequence is both valid and complete, a parser must validate each entry within the sequence to be valid and also that the entries were actually written as a sequence.

To check that the entries contained within a sequence were actually written as a sequence, each non-tail entry in a valid sequence must have a sequence number one greater than the previous entry and also each entry log guid matches the log guid field in the file header[4].

3.2.3 Log Replay

If the log is non-empty when the VHDX file is opened, the parser must replay the log[5] before performing any I/O to the file other than reading the VHDX header and the log. If a parser cannot replay the log because it lacks the capability, then it must not attempt to open the file.

When a parser needs to perform a log replay, it must find the newest valid and complete log sequence

[4] Having a matching log guid in the log entry is necessary to ensure that a non-current but fully written entry is not incorrectly used in log replay, which would be possible with the sequence number check alone.
[5] If the storage media on which the file is stored is read-only or the file is opened in read-only mode, the parser must not allow reads from the virtual disk without replaying the log in memory.

called the **active sequence**, and then replay all entries from it and no others.

A parser can follow the following steps to find the active sequence:

1. Set the candidate active sequence to the empty sequence, with a sequence number of zero. Set the current and old tail to zero.
2. Set the current sequence to the empty sequence with a head value equal to the current tail, and with a sequence number of zero.
3. Evaluate the validity of the log entry starting at the current sequence's head, verifying each field in the header and descriptors, including the checksum. If it is a valid entry, and either the current sequence is empty or the new entry's sequence number is one greater than the current sequence's sequence number, then extend the current sequence to include the log entry and repeat this step.
4. If the current sequence's head entry's tail is contained within the current sequence, then the current sequence is valid.
5. If the current sequence is valid and has a greater sequence number than the candidate active sequence, then set the candidate active sequence to the current sequence.
6. If the current sequence is empty or invalid, increase the current tail by 4 KB, wrapping to zero if equal to the log size. Otherwise, set the current tail to the current sequence's head, wrapping for values greater than or equal to the log size.
7. If the current tail is less than the old tail, then the log has been completely scanned. Stop. Otherwise, set the old tail to current tail and go to step 2.

If the candidate active sequence is empty, then there are no valid log sequences and the file is corrupt and the parser must fail to open it. If the file's size is less than the FlushedFileOffset field of the head entry of the candidate active sequence, then the file has been truncated, and the parser must fail to open it. Otherwise, the active sequence is the candidate active sequence.

Once the active sequence has been found, the parser must replay each descriptor within each entry within the sequence, in order beginning with the tail entry, by writing the data or zeroing as directed by the descriptors. The file offsets referred to by the descriptors may be larger than the current file offset, in which case the parser must extend the file.

After all the entries have been replayed, the parser must expand the file size to be at least as large as the LastFileOffset field of the head entry of the active sequence. This ensures that, even in the case where a VHDX file expansion operation could not be written and flushed to the host disk storage media due to a system power failure, all VHDX structures are fully contained within the VHDX file.

3.3 Blocks

Blocks are of two types: payload and sector bitmap. **Payload blocks** contain virtual disk payload data, while **sector bitmap blocks** contain parts of the sector bitmap.

Payload blocks are the size of the **BlockSize** field defined by the VHDX file parameter field (Section 3.5.2.1) and can be virtually indexed: payload block 0 contains the first BlockSize bytes of the virtual

disk; payload block 1 contains the second BlockSize bytes of the virtual disk; etc.

Sector bitmap blocks are always 1 MB in size, and can be virtually indexed similarly: sector bitmap block 0 contains the first 1 MB of the sector bitmap; sector bitmap block 1 contains the second 1 MB of the sector bitmap; etc.

Each sector bitmap block contains a bit for each logical sector in the file, representing whether the corresponding virtual disk sector is present in this file. Bit 0 (that is, bit 0 of byte 0) is the entry for the first virtual sector, bit 1 is the entry for the second, etc. For each bit, a value of 1 indicates that the payload data for the corresponding virtual sector should be retrieved from this file, while a value of zero indicates that the data should be retrieved from the parent VHDX file.

The number of sectors that can be described in each sector bitmap block is 2^{23}, so the number of bytes described by a single sector bitmap block is 2^{23} times the logical sector size (LogicalSectorSize – Section 3.5.2.4). This value is known as the **chunk size**. A virtually contiguous, chunk-size aligned and chunk-sized portion of the virtual disk is known as a **chunk**. The **chunk ratio** is the number of payload blocks in a chunk, or equivalently, the number of payload blocks per sector bitmap block.

$$Chunk\ Ratio = \frac{2^{23} * LogicalSectorSize}{BlockSize}$$

3.4 BAT

BAT is a region consisting of a single array of 64-bit values, with an entry for each block that determines the state and file offset of that block. The entries for the payload block and sector bitmap block are interleaved in a way that the sector bitmap block entry associated with a chunk follows the entries for the payload blocks in that chunk. For example, if the chunk ratio is 4, the table's interleaving would look like by the figure below.

Figure 6: BAT Layout Example

PB 0	PB 1	PB 2	PB 3	SB 0	PB 4	PB 5	PB 6	PB 7	SB 1	

PB Entry for Payload Block
SB Entry for Sector Bitmap Block

The BAT for a dynamic VHDX is laid out identically to a differencing VHDX; BAT entries for sector bitmap blocks exist for dynamic VHDX even though the sector bitmap block will never be allocated in a dynamic VHDX file. The presence of the entries avoids the need to insert or remove the interleaving sector bitmap entries during conversion between dynamic and differencing virtual hard disk types.

The BAT region must be at least large enough to contain as many entries as required to describe the possible blocks for a given virtual disk size (VirtualDiskSize - Section 3.5.2.2).

The number of data blocks can be calculated as

$$Data\ Blocks\ Count\ = Ceil\ (\ \frac{VirtualDiskSize}{BlockSize}\ )$$

The number of sector bitmap blocks can be calculated as

$$Sector\ Bitmap\ Blocks\ Count = Ceil\ (\ \frac{Data\ Blocks\ Count}{Chunk\ Ratio}\ )$$

For a dynamic VHDX, the last BAT entry must locate the last payload block of the virtual disk. The total number of BAT entries can be calculated as

$$Total\ BAT\ Entries = \ Data\ Blocks\ Count + \ Fl(\frac{Data\ Blocks\ Count-1}{Chunk\ Ratio})$$

For a differencing VHDX, the last BAT entry must be able to locate the last sector bitmap block that contains the last payload sector. The total number of BAT entries can be calculated as

$$Total\ BAT\ Entries\ =\ Sector\ Bitmap\ Blocks\ Count * (Chunk\ Ratio + 1)$$

3.4.1 BAT Entry

A BAT entry is 64 bits in length that is divided into bit fields.

Structure 9: BAT Entry

```
struct VHDX_BAT_ENTRY {
      UINT64      State:3;
      UINT64      Reserved:17;
      UINT64      FileOffsetMB:44;
};
```

The **State** field specifies how the associated data block or sector bitmap block should be treated.

The **FileOffsetMB** field specifies the offset within the file in units of 1 MB. The payload or sector bitmap block must reside after the header section and must not overlap any other structure. The FileOffsetMB field value must be unique across all the BAT entries when it is other than zero.

3.4.1.1 Payload BAT Entry States

The payload BAT entry determines the state of a virtual block and the associated offset in the VHDX file of that block. The table below summarizes the validity of the various states for the 3 VHDX types.

Table 5: Payload BAT Entry States

Payload BAT Entry State	Fixed	Dynamic	Differencing
PAYLOAD_BLOCK_NOT_PRESENT	Valid[6]	Valid	Valid
PAYLOAD_BLOCK_UNDEFINED	Valid[7]	Valid	Valid
PAYLOAD_BLOCK_ZERO	Valid[7]	Valid	Valid
PAYLOAD_BLOCK_UNMAPPED	Valid[7]	Valid	Valid

[6] Parsers should accept the VHDX files with these block states, but a parser should not transition blocks to these states in fixed VHDX files.

PAYLOAD_BLOCK_FULLY_PRESENT	Valid	Valid	Valid
PAYLOAD_BLOCK_PARTIALLY_PRESENT	Not Valid	Not Valid	Valid

Values 4 and 5 are reserved.

The behavior for the various block states for fixed VHDX is the same as that of the associated dynamic VHDX block state.

- `#define PAYLOAD_BLOCK_NOT_PRESENT 0`

This is the default state for all new blocks in dynamic and differencing VHDX types.

For fixed or dynamic VHDX files, this block state specifies the block contents are undefined and may contain arbitrary data, with the same restrictions as in PAYLOAD_BLOCK_UNDEFINED.

For a differencing VHDX file, this block state specifies that the block contents are not present in the file, and the parent virtual disk should be inspected to determine the associated contents.

The FileOffsetMB field for entries in this state is reserved.

- `#define PAYLOAD_BLOCK_UNDEFINED 1`

For all VHDX file types, this block state indicates that the block contents are not defined in the file and may contain arbitrary data, including data that was previously present elsewhere on the disk.

When a block entry is transitioned to this block state, parsers should implement one of the following options for the FileOffsetMB field;

1. Leave the field unmodified.
2. Set the field to zero value.
3. Set the field to a non-zero value that must point to a location in the VHDX file that must contain the contents of the block immediately before it was moved to this state with no modifications or some parts of it replaced with zeroes.

For reads from blocks in this state, parsers should implement one of the following behaviors[7];

1. Return arbitrary data including data that was previously present elsewhere on the disk[8].
2. Return zero data.
3. Return the contents of that the block immediately before it was moved to this state with no modifications or some parts of it replaced with zeroes.

- `#define PAYLOAD_BLOCK_ZERO 2`

This block state implies that the block contents are defined to be zero.

The FileOffsetMB field for entries in this state is reserved.

For reads from the blocks in this state, parsers must return zeroes.

- `#define PAYLOAD_BLOCK_UNMAPPED 3`

For all VHDX file types, this block state indicates that all the virtual disk sectors in the payload block were issued an UNMAP command and that the contents of this block are no longer being relied upon by the application or the system using the virtual disk. The block contents are defined to be zero data or the contents of that the block immediately before it was moved to this state with no modifications or some parts of it replaced with zeroes.

When a block entry in transitioned to this block state, parsers should implement one of the following options for the FileOffsetMB field;

1. Leave the field unmodified.
2. Set the field to zero.
3. Set the field to a non-zero value must point to a location in the VHDX file that must contain the contents of the block immediately before it was moved to this state with no modifications or

[7] If a parser chooses to preserve sector stability guarantees, it must continue to return the same data on subsequent block reads while the block remains in this state. To preserve sector stability across different parsers, it should transition the block to a state that has a tightly defined read behavior.
[8] If the parser chooses to return data that was previously present elsewhere on the disk, it could result in private data being leaked.

some parts of it replaced with zeroes.

For reads from blocks in this state, parsers should implement one of the following behaviors[8];

1. Return zero data.
2. Return the contents of that the block immediately before it was moved to this state with no modifications or some parts of it replaced with zeroes.

- `#define PAYLOAD_BLOCK_FULLY_PRESENT 6`

This is the default state for all blocks in fixed VHDX type.

For all VHDX file types, the block's contents are defined in the file at the location specified by the `FileOffsetMB` field.

The FileOffsetMB field may be modified to point to a new non-zero value that contains the contents of that block.

For reads from the blocks in this state, parsers should return the block contents defined in the file at the location specified by the `FileOffsetMB` field. For differencing VHDX files, the sector bitmap must not be inspected as the block is fully present in the VHDX file.

- `#define PAYLOAD_BLOCK_PARTIALLY_PRESENT 7`

This block state must not be present in a block entry for fixed or dynamic VHDX file.

For differencing VHDX files, the block's contents are defined in the file at the location specified by the FileOffsetMB field. When the block entry is in this state, the associated sector bitmap block must be already allocated and valid.

The FileOffsetMB field may be modified to point to a new non-zero value that contains the contents of that block.

For reads from the blocks in this state, parsers should return the block contents defined in the file at the location specified by the `FileOffsetMB` field after the associated sector bitmap for that block is inspected to check if the sector being read is present in the VHDX file. If not, the parent VHDX file needs to be inspected for that sector.

3.4.1.2 Sector Bitmap BAT Entry States

Sector Bitmap BAT entry indicates the presence of sector bitmap blocks for the associated chunk of payload blocks.

The various states and values are defined below. Values 1-5 and 7 are reserved. The table below summarizes the validity of the various states for the 3 VHDX types.

Table 6: Sector Bitmap BAT Entry States

Sector Bitmap BAT Entry State	Fixed	Dynamic	Differencing
SB_BLOCK_NOT_PRESENT	Valid	Valid	Valid
SB_BLOCK_ PRESENT	Not Valid	Not Valid	Valid

- #define SB_BLOCK_NOT_PRESENT 0

This state indicates that this sector bitmap block's contents are undefined and that the block is not allocated in the file.

For a fixed or dynamic VHDX file, all sector bitmap block entries must be in this state.

For a differencing VHDX file, a sector bitmap block entry must not be in this state if any of the associated payload block entries are in the PAYLOAD_BLOCK_PARTIALLY_PRESENT state.

The FileOffsetMB field for entries in this state is reserved.

- #define SB_BLOCK_PRESENT 6

This state indicates that the sector bitmap block contents are defined in the file at a location pointed to by the FileOffsetMB field.

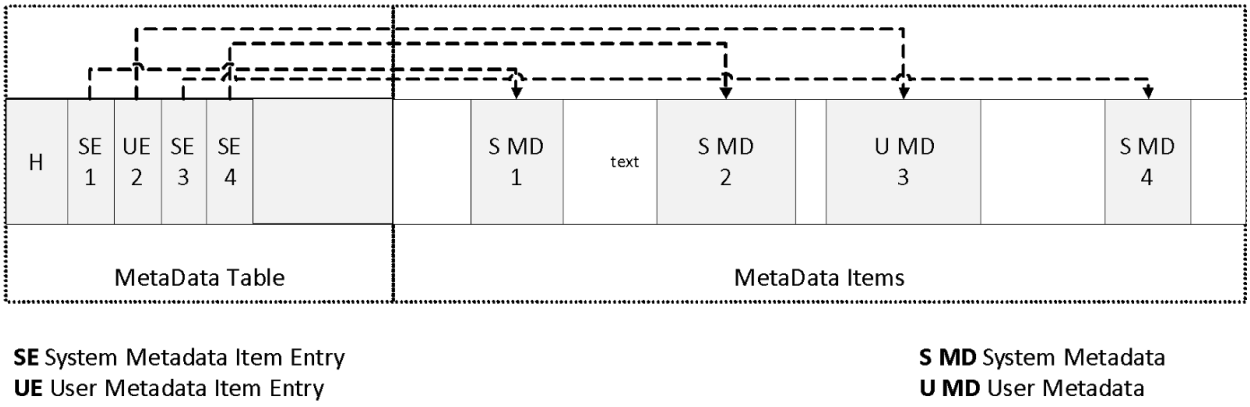
For a fixed or dynamic VHDX file, a sector bitmap block entry must not be in this state.

For differencing VHDX file, a sector bitmap block entry must be set to the SB_BLOCK_ PRESENT state if any associated payload blocks are the PAYLOAD_BLOCK_ PARTIALLY_PRESENT state. The sector bitmap block contents are defined in the file at the location specified by the FileOffsetMB field.

3.5 Metadata Region

The metadata region consists of a fixed-size, 64 KB, unsorted metadata table, followed by unordered, variable-sized, unaligned metadata items and free space. The metadata items represent both user and system metadata, which are distinguished by a bit in the table.

Figure 7: Metadata Region Layout Example



3.5.1 Metadata Table

A metadata table contains a 32-byte header followed immediately by a variable number of valid 32-byte

entries.

Structure 10: Metadata Table Header

```
struct VHDX_METADATA_TABLE_HEADER {
      UINT64      Signature;
      UINT16      Reserved;
      UINT16      EntryCount;
      UINT32      Reserved2[5];
};
```

The **Signature** field must be 0x617461646174656D (“metadata” as ASCII).

The **EntryCount** field specifies the number of entries in the table. This value must be less than or equal to 2047.

Structure 11: Metadata Table Entry

```
struct VHDX_METADATA_TABLE_ENTRY {
      GUID        ItemId;
      UINT32      Offset;
      UINT32      Length;
      UINT32      IsUser:1;
      UINT32      IsVirtualDisk:1;
      UINT32      IsRequired:1;
      UINT32      Reserved:29;
      UINT32      Reserved2;
};
```

The **ItemId** field specifies a 128-bit identifier for the metadata item. The **ItemId** and **IsUser** value pair for an entry must be unique within the table.

The **Offset** and **Length** fields specify the byte offset and length of the metadata item in bytes. **Offset** must be at least 64 KB and is relative to the beginning of the metadata region. **Length** must be less than or equal to 1 MB. The item described by this pair must fall entirely within the metadata region without overlapping any other item. Finally, if **Length** is zero, then **Offset** must also be zero, in which case the metadata item should be considered present but empty.

The **IsVirtualDisk** field specifies whether the metadata is file metadata or virtual disk metadata. This determines the behavior when forking a new differencing VHDX file from an existing VHDX file, or when merging a differencing VHDX file into its parent. When forking, a parser must copy all metadata items with this field set in the existing VHDX file to the new file, while leaving items with this field clear. When merging, a parser must destroy any metadata with this field set in the parent and copy all metadata with this field set in the child to the parent.

The **IsUser** field specifies whether this metadata item is considered system or user metadata. Only up to 1024 entries may have this bit set; if more entries have this set, the metadata table is invalid. A parser

should generally not allow users to query metadata items that have this bit set to False.

The **IsRequired** field specifies whether the parser must understand this metadata item to be able load the file. If this field is set to True and the parser does not recognize this metadata item, the parser must fail to load the file.

3.5.2 Known Metadata Items

There are certain Metadata items that are defined in this specification, some of which are optional and some of which are required. The table below summarizes the known metadata items and their optional or required state.

Table 7: Known Metadata Item Properties

Known Items	GUID	IsUser	IsVirtualDisk	IsRequired
File Parameters	CAA16737-FA36-4D43-B3B6-33F0AA44E76B	False	False	True
Virtual Disk Size	2FA54224-CD1B-4876-B211-5DBED83BF4B8	False	True	True
Page 83 Data	BECA12AB-B2E6-4523-93EF-C309E000C746	False	True	True
Logical Sector Size	8141BF1D-A96F-4709-BA47-F233A8FAAB5F	False	True	True
Physical Sector Size	CDA348C7-445D-4471-9CC9-E9885251C556	False	True	True
Parent Locator	A8D35F2D-B30B-454D-ABF7-D3D84834AB0C	False	False	True

3.5.2.1 File Parameters

Structure 12: File Parameters Metadata Item

```
struct VHDX_FILE_PARAMETERS {
      UINT32      BlockSize;
      UINT32      LeaveBlocksAllocated:1;
      UINT32      HasParent:1;
      UINT32      Reserved:30;
};
```

The **BlockSize** field specifies the size of each payload block in bytes. The value must be a power of 2 and at least 1 MB and at most 256 MB.

The **LeaveBlocksAllocated** field specifies whether blocks may be unallocated from the file. If this field is set to 1, the parser should not change the state of any blocks to a BLOCK_NOT_PRESENT state, and the parser must not reduce the size of the file to a value lower than would be required to allocate every

block. This field is intended to be used to create a fixed VHDX file that is fully provisioned.

The **HasParent** field specifies whether this file has a parent VHDX file. If set, the file is a differencing file, and one or more parent locators specify the location and identity of the parent.

3.5.2.2 Virtual Disk Size

Structure 13: Virtual Disk Size Metadata Item

```
struct VHDX_VIRTUAL_DISK_SIZE {
      UINT64      VirtualDiskSize;
};
```

The **VirtualDiskSize** specifies the virtual disk size, in bytes. This field must be a multiple of the **LogicalSectorSize** (See section 3.5.2.4) metadata item, and must be at most 64 TB.

3.5.2.3 Page 83 Data

Structure 14: Page 83 Data Metadata Item

```
struct VHDX_PAGE83_DATA {
      GUID        Page83Data;
};
```

This **Page83Data** Item is a GUID field and should be set to a value that unique across all SCSI devices that properly support page 0x83.

3.5.2.4 Logical Sector Size

Structure 15: Logical Sector Size Metadata Item

```
struct VHDX_VIRTUAL_DISK_LOGICAL_SECTOR_SIZE {
      UINT32      LogicalSectorSize;
};
```

The **LogicalSectorSize** field specifies the virtual disk's sector size, in bytes. This value must be set to 512 or 4096. A parser must expose the virtual disk as having the specified sector size, but it may fail to load files with sector sizes that it does not support. If the file has a parent, the logical sector size for the parent and child must be the same. Note that the **LogicalSectorSize** value also determines the chunk size.

3.5.2.5 Physical Sector Size

Structure 16: Logical Sector Size Metadata Item

```
struct VHDX_VIRTUAL_DISK_PHYSICAL_SECTOR_SIZE {
      UINT32      PhysicalSectorSize;
};
```

The **PhysicalSectorSize** field specifies the virtual disk's physical sector size, in bytes. This value must be set to 512 or 4096. A parser must expose the virtual disk as having the specified physical sector size, but it may fail to load files with sector sizes that it does not support.

3.5.2.6 Parent Locator

The parent locator specifies the type of the parent virtual block device as a GUID and a set of key-value pairs describing anything necessary to locate and connect to the parent block device.

When the **HasParent** field of the file parameters metadata item is set, there must be a parent locator. This metadata item specifies the identity and location of a parent virtual block device, whether this device is another VHDX file, a different type of virtual disk file, a logical unit in a SAN, or otherwise.

This specification describes only one such parent locator type, but other parsers may extend this as required.

The item is made up of a 20-byte header immediately followed by a table of 12-byte entries specifying the offset and length of each key and value.

Structure 17: Parent Locator Header

```
struct VHDX_PARENT_LOCATOR_HEADER {
      GUID        LocatorType;
      UINT16      Reserved;
      UINT16      KeyValueCount;
};
```

The **LocatorType** field specifies the type of the parent virtual disk. This value will be different for each type (for example, VHDX, VHD or ISCSI), so a parser must validate that it understands that type.

The **KeyValueCount** field specifies the number of key-value pairs defined for this parent locator.

Structure 18: Parent Locator Entry

```
struct VHDX_PARENT_LOCATOR_ENTRY {
      UINT32      KeyOffset;
      UINT32      ValueOffset;
      UINT16      KeyLength;
      UINT16      ValueLength;
};
```

The **KeyOffset** and **KeyLength** fields specify the offset within the metadata item and length in bytes of the entry's key. Both values must be must be greater than zero.

The **ValueOffset** and **ValueLength** fields specify the offset within the metadata item and length in bytes of entry's value. Both values may be zero.

The key and value strings are to be UNICODE strings with UTF-16 little-endian encoding. There must be no internal NUL characters, and the length field must not include a trailing NUL character. The key string is case sensitive, and lower-case keys are recommended. All keys must be unique, and there is no

ordering to the entries.

3.5.2.6.1 VHDX Parent Locator

The only parent locator type defined by this specification is the VHDX locator type with a GUID value of "B04AEFB7-D19E-4A81-B789-25B8E9445913".

VHDX Parent locator has several possible key-value pair entries, some of which are required, as described below.

Table 8: VHDX Parent Locator Entries

Entry	Type	Example
parent_linkage	GUID	{83ed0ec3-24c8-49a6-a959-5e4bf1288bfb}
parent_linkage2	GUID	{83ed0ec3-24c8-49a6-a959-5e4bf1288bfb}
relative_path	Path	..\..\path2\sub3\parent.vhdx
volume_path	Path	\\?\Volume{26A21BDA-A627-11D7-9931-806E6F6E6963}\path2\sub3\parent.vhdx
absolute_win32_path	Path	\\?\d:\path2\sub3\parent.vhdx

The two entries with key values of **parent_linkage** and **parent_linkage2** specify possible values of the parent's identity. The parent_linkage entry must be present, and parent_linkage2 may not be present. The value field is encoded as a lower-case string with enclosing braces; for example: {83ED0EC3-24C8-49A6-A959-5E4BF1288BFB}. When a differencing VHDX file is created, the parser must populate the parent's **DataWriteGuid** field in this field. When opening the parent VHDX file of a differencing VHDX, the parser must verify that the **DataWriteGuid** field of the parent's header matches one of these two fields[9].

At least one entry with key value of **relative_path**, **volume_path**, or **absolute_win32_path** must be present to locate the parent VHDX file. A parser should evaluate the paths in a specific order to locate the parent; relative_path, volume_path and then absolute_path. Upon successful open of a chain, a parser should update any existing stale path entries to point to its current parent file.

The entry with key value of **relative_path** specifies the path of the parent VHDX file relative to the path of the current VHDX file, using "\" as the path separator and ".." to mean parent directory. For example, if the VHDX file is in "d:\path1\sub2\file.vhdx" and the parent file is in "d:\path2\sub3\parent.vhdx", the value field for this entry could contain "..\..\path2\sub3\parent.vhdx".

The entry with key value of **volume_path** specifies the path of the parent VHDX using an absolute Win32 path containing the volume GUID of the volume on which the parent resides (for example, "\\?\Volume{26A21BDA-A627-11D7-9931-806E6F6E6963}\path2\sub3\parent.vhdx"). This helps locate the parent when drive letters are not available or stable. It must not contain other forms of Win32

[9] There are 2 fields so that parent-child links can be safely maintained while merging child data into the parent. In that case, the parent's write GUID must be changed. When this occurs, the parser should first store the new GUID in the parent identifier, and then update the parent's write GUID.

paths.

The entry with key value of **absolute_win32_path** specifies the path of the parent VHDX using an absolute extended-length Win32 path on a local drive. This path must begin with "\\?\" , and may be followed by a drive letter (for example, "d:") or a UNC share (for example,"\\ServerName\ShareName"), then the path to the file on that drive using "\" as the path separator (for example, "\path2\sub3\parent.vhdx"). Then value field for this entry could contain "\\?\d:\path1\sub2\file.vhdx" or "\\?\MyServer\MyShare\ path3\sub4\file.vhdx".

4 Appendix

4.1 Globally Unique Identifiers (GUIDs)

A GUID is the Microsoft implementation of a universally unique identifier. A GUID is a 128-bit value that consists of one group of 8 hexadecimal digits, followed by three groups of 4 hexadecimal digits each, and followed by one group of 12 hexadecimal digits. All fields are mandatory to be populated.

```
struct GUID {
      UINT32      Data1;
      UINT16      Data2;
      UINT16      Data3;
      UINT8[8]    Data4;
};
```

For example, A GUID value of {2A29FC40-FA49-8167-D31D-10EE010662DA} would result in the individual fields in the GUID structure populated as below.

Data1 = 2A29FC40

Data2 = FA49

Data3 = 8167

Data4 = D31D10EE010662DA

4.2 CRC-32C

CRC-32C algorithm is used in the VHDX specification for CRCs as it is very robust and supported on certain newer CPUs which decrease the overhead for computation of the checksum.

This specification uses the Castagnoli polynomial, code 0x11EDC6F41

For a detailed analysis of the CRC-32C, see

[Castagnoli93]: G. Castagnoli, S. Braeuer and M. Herrman
"Optimization of Cyclic Redundancy-Check Codes with 24 and 32 Parity Bits", IEEE Transact. on Communications, Vol. 41, No. 6, June 1993.

5 Documentation Change History

Table below describes the history of releases of, corrections to, additions to, removals from, and clarifications of this document.

Date	Version	Description of Change
25-April-2012	0.95	Draft version of VHDX specification.
25-August-2012	1.00	Update to the VHDX specification, which includes general readability improvements as well as the following changes and clarifications: ● Section 2.3: Updated the header section description to clarify that there are two copies of the region table ● Figure 3: Corrected to show two copies of the region table ● Structure 2:Corrected the length of the reserved section in the VHDX header structure ~~UINT8[502] Reserved~~ UINT8 Reserved[4016] ● Section 3.1.3: Update the region table descriptions to clarify that there are two copies of the region table and updates to these structures must be made through the log ● Section 3.2.1: 1. Updated the text to clarify the layout of the log entry 2. Corrected the alignment requirements for the EntryLength field of the log entry structure ~~1 MB~~ 4 KB 3. Clarified the checksum description CRC-32C hash computed over the entire entry *specified by the EntryLength field* ● Section 3.2.3: Clarified the old tail field values in the log replay algorithm ● Section 3.4.1: Removed footnote 6 in the previous version ~~1 MB is used rather than BlockSize so that this value can be changed without rewriting the BAT.~~ ● Section 3.4.1.1: Corrected the payload block UNMAP state enumeration value ~~PAYLOAD_BLOCK_UNMAPPED 5~~ PAYLOAD_BLOCK_UNMAPPED 3