A Quick Tour of GUID Partition Table (GPT)
15 minute read

Introduction

What is GUID Partition Table, GPT ? I will demonstrate it on Linux.

Warning: There are commands used in this article that may harm your computer pretty bad when used improperly (e.g. if you write, instead of read, to the LBA 0), and you may lose data. If you would like to experiment on your own, be careful to not write anything to the disk. The best is to install a fresh Linux on a VM and use that for experimenting, not your own computer.

How does a computer boot ? What happens until you see the login prompt on Linux ? I am sure many, including myself, does not know all the steps. A firmware known as or compliant to BIOS (Basic Input/Output System) before (2000s), and its successor UEFI (Unified Extensible Firmware Interface) now (after 2000s), loads the OS (Operating System). For this, it needs to know where the first executable code is on the storage media. Exactly at this point, the boot record and/or partition table enters into the picture and that is also why GPT (GUID Partition Table) is described in the UEFI specification.

> If you are older than 30, like me, you probably remember BIOS and MBR (Master Boot Record) terms. I have learned all about computers when BIOS and MBR was in use, and then they are replaced by UEFI and GPT, and I knew nothing about them for a long time. If you are in the same position or younger and did not learn about these before, this article will help you.

In order to load the OS, the location of the system files of OS has to be found, accessed, read and execution should be transferred to OS. It is becoming like a chicken and egg scenario, you need to load a file but file is in a file system which is in a storage media, both of which requires a driver to be loaded, and driver is also a file.

The solution is to define something fixed so firmware can depend on that without thinking about individual files. The fixed concept is start booting by loading data from the very start of the storage. This is same both for BIOS and UEFI.

Probably not very well known, it would be good to say, in order to do things with the storage hardware, you send commands to it and it replies. So you can send a read or write command, together with its location, and it just does that - in a very complicated way -. There are of course many other command types other than read and write. These are either ATA/ATAPI Commands for SATA drives (specified in ATA/ATAPI specs) or NVM Commands for NVM Express drives (specified in NVM Express spec).

We still need to know the size or length of data to load. Here some hardware terms related to storage media enters into the picture. In the past (before 2000s), storage layout was according to C/H/S (Cylinder / Head / Sector). Basically you had to specify 3 numbers (C,H,S) to read from or write to something to storage. These were first corresponding the actual physical structure of storage (e.g. floppy and hard drives) then after embedding controllers (after 80s) to the storage devices, they were logical numbers but being translated to physical numbers by the storage device before doing the actual work. Coming back to our question about the length of data to be read, it is the size of one sector. So BIOS had to load 0/0/1 (Cylinder=0, Head=0, Sector=1), cylinder and head numbers start from 0, sector number starts from 1, so it is the address of the first sector. The actual size (in bytes) of a sector depends on the storage media, it can be queried from the hardware (e.g. ATA Command Identify Device). Sector size of 512 bytes is a common value.

Instead of C/H/S, another system is developed after 90s, and basically this is the system in use today in all computers, and it is called Logical Block Addressing (LBA). LBA is simpler, it is basically a single number addressing the storage media, e.g. the start of storage media is LBA 0, then it goes like LBA 1, 2. Each individual number is said to address a Logical Block, whose size can also be queried from the hardware. 512 bytes is also a common Logical Block size.

You may wonder where LBA ends or what is the maximum it can support. Initially specified in IDE spec, it was a 22 bits number, so maximum would be 2GB which is very small in today's standard. In ATA-1 spec, it is increased to 28 bits (max. 128 GB) then with ATA-6 spec, it is increased to 48 bits (max. 128 PB).

128 PB is possible only with GPT because it allows 64-bit LBA values, whereas MBR allows only 32-bit. So 2TB is the maximum partition size for MBR.

All these maximum values are of course assuming a sector / logical block size of 512 bytes. Now there are hard drives supporting 4K also, which is called Advanced Format.

An NVM device (not all devices but if supported) may have variable sector sizes / different LBA formats. This is specified when NVM device is formatted and it can be queried what LBA formats NVM device supports.

You can use nvme utility to learn what is the LBA format, thus what is the logical block size. Not surprisingly, it is 512 bytes written in the Format column.

```
$ sudo nvme list
Node             SN                   Model                                    Namespace Usage                      Format           FW Rev
---------------- -------------------- ---------------------------------------- --------- -------------------------- ---------------- --------
/dev/nvme0n1     S27FNY0HB04880       SAMSUNG MZVPV512HDGL-000H1               1         399.86  GB / 512.11  GB    512   B +  0 B   BXW74H0Q
```

Another, more detailed, way to look to this is:

```
$ sudo nvme get-ns-id /dev/nvme0n1

nvme0n1: namespace-id:1
```

So `/dev/nvme0n1` uses namespace id=1

```
$  sudo nvme id-ns /dev/nvme0n1 -n 1

NVME Identify Namespace 1:
nsze    : 0x3b9e12b0
ncap    : 0x3b9e12b0
nuse    : 0x12d5b870
nsfeat  : 0
nlbaf   : 0
flbas   : 0
mc      : 0
dpc     : 0
dps     : 0
nmic    : 0
rescap  : 0
fpi     : 0
nawun   : 0
```

```
nawupf  : 0
nacwu   : 0
nabsn   : 0
nabo    : 0
nabspf  : 0
nvmcap  : 0
nguid   : 00000000000000000000000000000000
eui64   : 0025386b610046b6
lbaf  0 : ms:0   ds:9  rp:0 (in use)

nlbaf=0
number of supported LBA formats=0
meaning only LBA Format 0 is supported.

flbas=0
formatted LBA size
last 3 bits indicate LBA Format, which is 0 here.

at the bottom, lbaf=0
LBA Format 0

next to lbaf=0, ds=9 (LBADS in NVM Spec)
LBA Data Size (2^ds)
so here, 2^9=512, that is the value we are looking for.

So to sum up, this SSD is formatted with LBA Format 0 (and also supports only LBA Format 0), and LBA Format 0 has Logical Block size of 512 bytes.
```

Lets summarize now. In order to load the OS, we probably first need to query the storage media, learn about its parameters (maybe not needed now but will be needed anyway), and then request a read of LBA 0 from the storage media.

The data in LBA 0 is MBR and it can be:

- Legacy MBR or
- Protective MBR

Legacy MBR is actually the MBR we know before UEFI and GPT. However, an UEFI system can also boot from Legacy MBR. The main difference is the partition table they describe are very different.

- Legacy MBR contains max. 4 (primary/extended) partitions, if one of them is marked as an EFI System Partition (partition type 0xEF), you can think of this as bootable flag, it is loaded by the UEFI firmware. Legacy MBR may also not contain any EFI System Partition and can be a currently obsolete pure MBR system. In that case there is a boot code in the beginning of MBR, that is loaded and executed. That is what BIOS was doing in the past.
- Protective MBR contains only 1 partition, and it spans all the storage media (up to the maximum addressable size in MBR, because it is an 32-bit value) and its type is GPT Protective type 0xEE. I believe this is the current default in modern OSes. You will see an example of this in this article.

Now lets continue with real examples. All examples below is from a Linux desktop computer and Linux is installed on an NVMe drive.

First question, what are my storage devices ?

```
$ lsblk

NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
nvme0n1     259:0    0   477G  0 disk
├─nvme0n1p3 259:3    0  31.8G  0 part [SWAP]
├─nvme0n1p1 259:1    0   512M  0 part /boot/efi
└─nvme0n1p2 259:2    0 444.7G  0 part /
```

nvme0n1 is the physical device (TYPE=disk), others are the partitions on it (TYPE=part).

Lets also see what fdisk shows. We actually do not need any information from fdisk, as we will derive all these information ourselves later.

```
$ sudo fdisk /dev/nvme0n1

Welcome to fdisk (util-linux 2.27.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): p
Disk /dev/nvme0n1: 477 GiB, 512110190592 bytes, 1000215216 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: D199E4F8-1D1C-4A56-BE5D-B279BB54EFA8

Device             Start        End   Sectors   Size Type
/dev/nvme0n1p1      2048    1050623   1048576   512M EFI System
/dev/nvme0n1p2   1050624  933529599 932478976 444.7G Linux filesystem
/dev/nvme0n1p3 933529600 1000214527  66684928  31.8G Linux swap
```

This shows:

- Disk (nvme0n1) is 512110190592 bytes and has 1000215216 sectors.
- Each unit, is a sector, and is 512 bytes.
- Both logical and physical sector sizes are 512 bytes.
- There is GPT on it (Disklabel type: gpt). There is no particular flag or field encoding this. It basically means it contains a Protective MBR, you will see in detail later.
- There are 3 partitions, starting from sector 2048 to sector 1000214527.

Because of the history of storage media, there are so many terms here. In a modern computer, you can forget about all these terms, and think only in terms of logical blocks (LB), which has a fixed size like 512 bytes, and logical block addressing (LBA), starting from 0.

According to GPT spec.:

- LBA Block 0 is either Legacy MBR or Protective MBR
- LBA Block 1 is Primary GPT Header
- Last LBA Block is Backup GPT Header (a copy of Primary GPT Header)

Having a backup copy of GPT Header and also GPT Partition Entries is a feature of GPT. MBR only exists on LBA 0.

Python gpt package and its utilities

I could not find a proper tool to describe MBR and GPT records, so I wrote the gpt package that I use in this post. It can be installed by `pip install gpt`.

Master Boot Record

Lets look at the MBR first. Lets save LBA 0.

```
$ sudo dd if=/dev/nvme0n1 bs=512 count=1 skip=0 > lba.0
```

If you have NVMe device, it is also possible to read data directly with nvme-cli tools:

```
$ sudo nvme read /dev/nvme0n1 --start-block=0 --block-count=0 --data-size=512 --data=lba.0
```

This submits a Read command starting from LBA 0 for 1 block to NVMe device, and writes the data to lba.0 file. It is strange but block count is a 0 based value, so 0 means read one block.

Now lets look into the MBR:

```
$ cat lba.0 | print_mbr
```

```
<<< MBR >>>
BootCode: 0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
UniqueMBRDiskSignature:                                     0x5d3f5955
Unknown:                                                        0x0000
PartitionRecord: 0x00000100eefeffff01000000af129e3b000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Signature:                                                      0xAA55
<<< MBR Partition #0 >>>
#0.BootIndicator:                                                  0x0
#0.Is Bootable? (syn):                                              No
#0.StartingCHS:                                                0, 1, 0
#0.OSType:                                                        0xEE
#0.OSType (syn):                                        GPT Protective
#0.EndingCHS:                                            255, 254, 255
#0.StartingLBA:                                                      1
#0.SizeInLBA:                                               1000215215
<<< MBR Partition #1 >>>
#1.BootIndicator:                                                  0x0
#1.Is Bootable? (syn):                                              No
#1.StartingCHS:                                                0, 0, 0
#1.OSType:                                                         0x0
#1.OSType (syn):                                                 Empty
#1.EndingCHS:                                                  0, 0, 0
#1.StartingLBA:                                                      0
#1.SizeInLBA:                                                        0
<<< MBR Partition #2 >>>
#2.BootIndicator:                                                  0x0
#2.Is Bootable? (syn):                                              No
#2.StartingCHS:                                                0, 0, 0
#2.OSType:                                                         0x0
#2.OSType (syn):                                                 Empty
#2.EndingCHS:                                                  0, 0, 0
#2.StartingLBA:                                                      0
#2.SizeInLBA:                                                        0
<<< MBR Partition #3 >>>
#3.BootIndicator:                                                  0x0
#3.Is Bootable? (syn):                                              No
#3.StartingCHS:                                                0, 0, 0
#3.OSType:                                                         0x0
#3.OSType (syn):                                                 Empty
#3.EndingCHS:                                                  0, 0, 0
#3.StartingLBA:                                                      0
#3.SizeInLBA:                                                        0
```

Signature (0xAA55) shows it is a valid MBR. There is no BootCode (all zeroes) and there is only one partition of type GPT Protective, which covers the entire storage (C/H/S 255/254/255 is the last addressable value). This means it is a Protective MBR. The actual partition table follows.

GPT Header

Now lets look into the second block, LBA.1, because the MBR is a Protective MBR, LBA.1 contains the GPT Header.

```
$ sudo dd if=/dev/nvme0n1 bs=512 count=1 skip=1 > lba.1
```

Now lets look into LBA 1:

```
$ cat lba.1 | print_gpt_header
```

```
Warning: Using only the first 92 bytes of input
<<< GPT Header >>>
Signature:                                          0x4546492050415254
Revision:                                                   0x00000100
HeaderSize:                                                         92
HeaderCRC32:                                                0xb9611f71
HeaderCRC32 (calculated):                                   0xb9611f71
Reserved:                                                   0x00000000
MyLBA:                                                               1
AlternateLBA:                                               1000215215
FirstUsableLBA:                                                     34
LastUsableLBA:                                              1000215182
PartitionEntryLBA:                                                   2
NumberOfPartitionEntries:                                          128
```

```
SizeOfPartitionEntry:                                  128
PartitionEntryArrayCRC32:                          0xcf402365
```

The header says:

- This header is on LBA 1 (MyLBA)
- Its copy is on LBA 1000215215 (AlternateLBA)
- The first usable LBA for data storage is 34 (FirstUsableLBA)
- The last usable LBA for data storage is 1000215182 (LastUsableLBA)
- Partition Entries start on LBA 2 (PartitionEntryLBA)
- There are 128 Partition Entries (NumberOfPartitionEntries), each having size of 128 bytes (SizeOfPartitionEntry)

The reason of first usable LBA being 34 is simple. LBA 0 is Protective MBR, LBA 1 is GPT Header and the required space for partition entries are:

```
128 partition * 128 bytes/partition / 512 bytes/block = 32 blocks
```

So 1 + 1 + 32 = 34 blocks are needed to store all GPT information, so first usable LBA can be minimum 34. As you might realize, these numbers change if the logical sector size (LBA block size) is not 512 bytes.

```
For example: a logical sector size of 4K is possible now and it is called Advanced Format. For example you can see at the [WD Red Hard Drive Specs](ht

As an example, lets calculate above for a logical block size of 4K.

So LBA 0 is MBR. LBA 1 is GPT Header.
128 partition * 128 bytes/partition / 4096 bytes/block = 4 blocks

So required blocks are 1+1+4 = 6. So in this case FirstUsableLBA would be 6 instead of 34.
```

Now lets look at the alternate LBA:

```
$ sudo dd if=/dev/nvme0n1 bs=512 count=1 skip=1000215215 > lba.1000215215
```

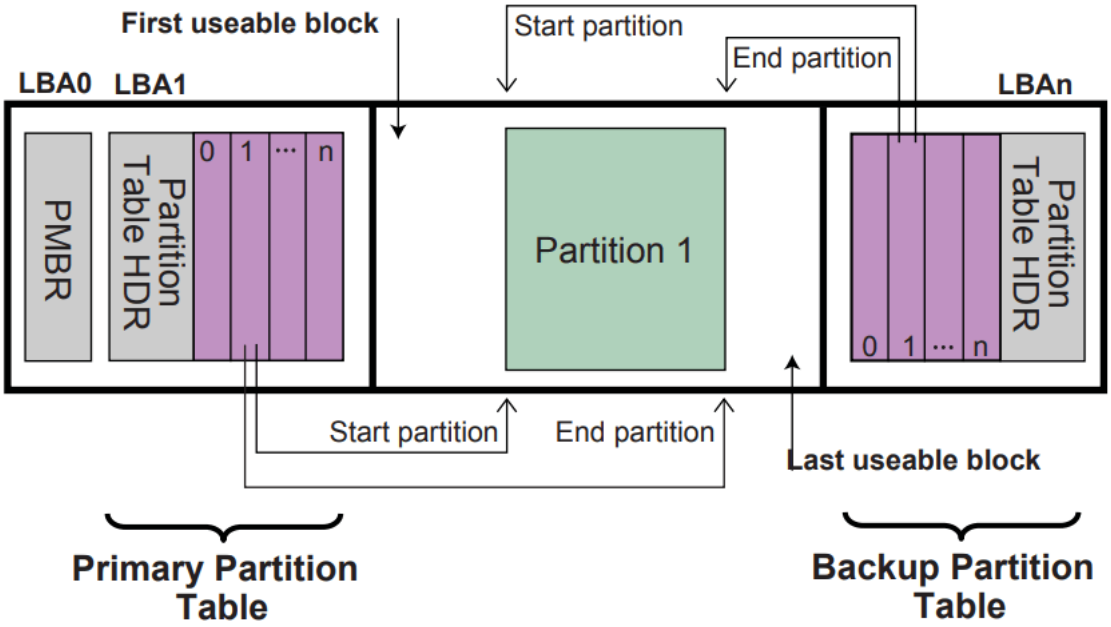
Looking into contents of lba.last:

```
$ cat lba.1000215215 | print_gpt_header

Warning: Using only the first 92 bytes of input
<<< GPT Header >>>
Signature:                                  0x4546492050415254
Revision:                                          0x00000100
HeaderSize:                                                92
HeaderCRC32:                                       0x5e60ec2b
HeaderCRC32 (calculated):                          0x5e60ec2b
Reserved:                                          0x00000000
MyLBA:                                             1000215215
AlternateLBA:                                               1
FirstUsableLBA:                                            34
LastUsableLBA:                                     1000215182
PartitionEntryLBA:                                 1000215183
NumberOfPartitionEntries:                                 128
SizeOfPartitionEntry:                                     128
PartitionEntryArrayCRC32:                          0xcf402365
```

It is basically same, but of course MyLBA and AlternateLBA is reversed and HeaderCRC32 is different. Not only the header but also the partition entries are backed up like this.

As you see, there is also CRC32 error detecting codes on GPT records. This is also a feature of GPT, MBR has no error detection mechanism. `print_gpt_header` also calculates the GPT Header CRC32 and not surprisingly matches the one on the records.



GUID Partition Table (GPT) example (from UEFI Spec 2.7)

The figure above from the original UEFI Spec. shows the structure quite nice. So basically the usable space for OS is between the first usable block to last usable block.

GPT Partition Entry Array

Now lets look at the GPT Partition Table entries. As calculated before, lets save blocks from 2 to 34.

```
$ sudo dd if=/dev/nvme0n1 bs=512 count=32 skip=2 > lba.2-34
```

and now lets look at what it contains:

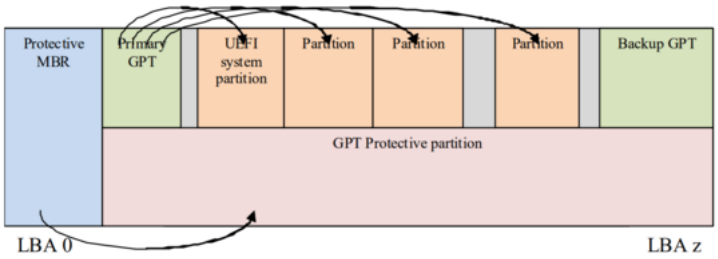
```
$ cat lba.2-34 | print_gpt_partition_entry_array

<<< GPT Partition Entry #0 >>>
#0.PartitionTypeGUID:                   0x28732ac11ff8d211ba4b00a0c93ec93b
#0.PartitionTypeGUID (syn):           c12a7328-f81f-11d2-ba4b-00a0c93ec93b
#0.PartitionType (syn):                                EFI System Partition
#0.UniquePartitionGUID:                 0x41418ac02f30694e8af1c0c83cc3ab9e
#0.UniquePartitionGUID (syn):         c08a4141-302f-4e69-8af1-c0c83cc3ab9e
#0.StartingLBA:                                                      2048
#0.EndingLBA:                                                     1050623
#0.Attributes:                                                        0x0
#0.Attributes (syn):                                                   []
#0.PartitionName: 0x450046004900200053007900730074006500 6d00200050006100720074006900740069006f006e0000000000000000000000000000000000000000000000000000000000000000
#0.PartitionName (syn):                                 EFI System Partition
<<< GPT Partition Entry #1 >>>
#1.PartitionTypeGUID:                   0xaf3dc60f838472478e793d69d8477de4
#1.PartitionTypeGUID (syn):           0fc63daf-8483-4772-8e79-3d69d8477de4
#1.PartitionType (syn):                               Linux filesystem data
#1.UniquePartitionGUID:                 0xca772bc0c39ad74a827edd51b91f34c2
#1.UniquePartitionGUID (syn):         c02b77ca-9ac3-4ad7-827e-dd51b91f34c2
#1.StartingLBA:                                                   1050624
#1.EndingLBA:                                                   933529599
#1.Attributes:                                                        0x0
#1.Attributes (syn):                                                   []
#1.PartitionName: 0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
#1.PartitionName (syn):
<<< GPT Partition Entry #2 >>>
#2.PartitionTypeGUID:                   0x6dfd5706aba4c44384e50933c84b4f4f
#2.PartitionTypeGUID (syn):           0657fd6d-a4ab-43c4-84e5-0933c84b4f4f
#2.PartitionType (syn):                                       Swap partition
#2.UniquePartitionGUID:                 0xc6c1f8bfa67b2843b1eb40f3aaecf1a3
#2.UniquePartitionGUID (syn):         bff8c1c6-7ba6-4328-b1eb-40f3aaecf1a3
#2.StartingLBA:                                                  933529600
#2.EndingLBA:                                                   1000214527
#2.Attributes:                                                        0x0
#2.Attributes (syn):                                                   []
#2.PartitionName: 0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
#2.PartitionName (syn):
<<< Calculated >>>
PartitionEntryArrayCRC32 (calculated):                        0xcf402365
```

There are 128 entries, but by default, `print_gpt_partition_entry_array` does not display the empty partitions, and there are only 3 non-empty partitions. Empty partition record is all zeroes.

You see why it is called GUID Partition Table, as partitions are identified by a GUID. Also, each partition has a name defined in the partition table. These are features of GPT and does not exist in MBR.

To combine this with Protective MBR, we have this structure, again shown nicely from the original UEFI Spec:



GPT disk layout with protective MBR example (from UEFI Spec 2.7)

You may ask what happens if the storage is larger than what MBR can address. In this case GPT Protective partition covers up to maximum size that can be addressed, but GPT Partition Entries, as expected, can use the entire storage.

You may also ask why the first partition starts from LBA 2048, not from LBA 34 since it is the first usable LBA as indicated in GPT. I think the reason is to align partition boundaries. Quoting directly from GPT spec:

“To avoid the need to determine the physical block size and the optimal transfer length granularity, software may align GPT partitions at significantly larger boundaries. For example, assuming logical block 0 is aligned, it may use LBAs that are multiples of 2,048 to align to 1,048,576 byte (1 MiB) boundaries, which supports most common physical block sizes and RAID stripe sizes.”

fdisk on Linux is updated at version 2.17.1 to use 1MB partition boundaries. So by default it starts from sector 2048 (512 bytes * 2048 = 1MB) and any partition if sized in multiples of 1MB is aligned to 1MB boundaries.

If you want to learn more about NVMe and ext4, you can look to my posts:

- A Minimum Complete Tutorial of Linux ext4 File System
- A Quick Tour of NVM Express (NVMe)
