[UEFI Basics] SMBIOS Basics and Usage

jiangwei0512 Modified on 2024-09-04 23:39:33 Read 3.6k Collection 39 Likes 18

Category Column: UEFI Development Basics Article Tags: uefi smbios

Definition of SMBIOS

The full name of SMBIOS is System Management BIOS . Its understanding includes:

- It is not a BIOS. The word BIOS appears because it is related to BIOS, that's all.
- It is a specification that defines the system management information that BIOS passes to the operating system. For specific information, please refer to the SMBIOS specification.
- It can also represent a series of data structures containing various types of information, which are created during the BIOS boot process and placed in a specific memory, which can then be used by the operating system.

If you still don't understand it, the best way is to check the SMBIOS specification. You can download various versions of the specification documents at System Management BIOS (SMBIOS) | DMTF. The document defines various types of data. Each type is called a Type . Type x (x represents a number) is used to define different types of data. For example, Type 0 represents BIOS information (note that this screenshot is incomplete, and the following content is omitted):

Table 6 – BIOS Information (Type 0) structure

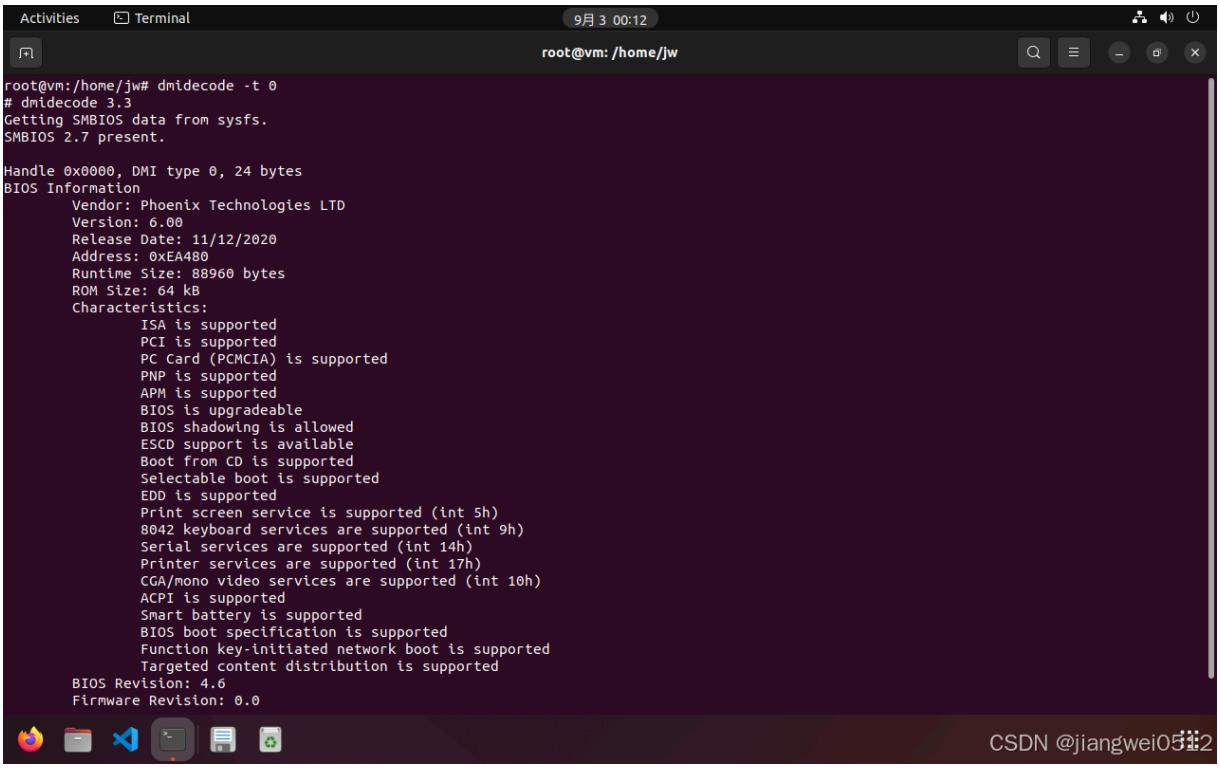
Offset	Spec. Version	Name	Length	Value	Description
00h	2.0+	Type	BYTE	0	BIOS Information indicator
01h	2.0+	Length	BYTE	Varies	12h + number of BIOS Characteristics Extension Bytes. If no Extension Bytes are used the Length is 12h. For version 2.1 and 2.2 implementations, the length is 13h because one extension byte is defined. For version 2.3 and later implementations, the length is at least 14h because two extension bytes are defined. For version 2.4 and later implementations, the length is at least 18h because bytes 14-17h are defined.
02h	2.0+	Handle	WORD	Varies	
04h	2.0+	Vendor	BYTE	STRING	String number of the BIOS Vendor's Name.
05h	2.0+	BIOS Version	BYTE	STRING	String number of the BIOS Version. This value is a free-form string that may contain Core and OEM version information.
06h	2.0+	BIOS Starting Address Segment	WORD	Varies	Segment location of BIOS starting address (for example, 0E800h). NOTE: The size of the runtime BIOS image can be computed by subtracting the Starting Address Segment from 10000h and multiplying the result by 16.
08h	2.0+	BIOS Release Date	BYTE	STRING	String number of the BIOS release date. The date string, if supplied, is in either mm/dd/yy or mm/dd/yyyy format. If the year portion of the string is two digits, the year is assumed to be 19yy. NOTE: The mm/dd/yyyy format is required for SMBIOS version 2.3 and later.
09h	2.0+	BIOS ROM Size	BYTE	Varies (n)	Size (n) where 64K * (n+1) is the size of the physical device containing the BIOS, in bytes

It contains some basic information of BIOS, such as BIOS version, BIOS vendor, etc. Some of them are strings, while others are fixed data types.

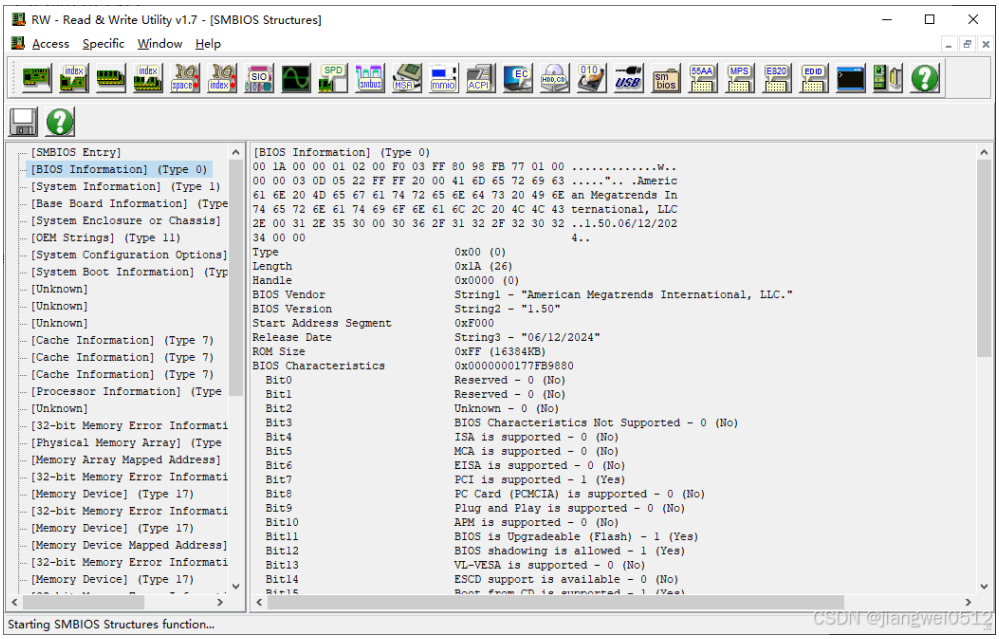
However, it should be noted that there are several different versions of SMBIOS, such as SMBIOS2.x for 32-bit and SMBIOS3.x for 64-bit. Also, the location of SMBIOS data is different for UEFI and Legacy versions of BIOS. This article only focuses on the most commonly used one, **SMBIOS3.x** , which is delivered by **UEFI BIOS** .

View SMBIOS

We can view SMBIOS information through some basic tools, such as `dmidecode` the command in Linux:



For example, in Windows, you can use the RW tool (RWEverything – Read & Write Everything) to view:



As for how the operating system finds this space, it can be found according to the specification:

> On UEFI-based systems, the SMBIOS Entry Point structure can be located by looking in the EFI 738 Configuration Table for the SMBIOS 3.x GUID (SMBIOS3_TABLE_GUID, {F2FD1544-9794-4A2C-992E-739 E5BBCF20E394}) and using the associated pointer.

That is, you can find it through the UEFI interface and specify the GUID. This GUID can be found in the Linux source code (include/linux/efi.h):

c AI generated projects 登录复制 run

```c
#define SMBIOS3_TABLE_GUID          EFI_GUID(0xf2fd1544, 0x9794, 0x4a2c,  0x99, 0x2e, 0xe5, 0xbb, 0xcf, 0x20, 0xe3, 0x94)
```

Since only Linux has source code (note that it is Linux source code, not `dmidecode` the source code of , which only parses the DMI table generated by Linux), the following is to determine how the operating system finds SMBIOS information based on it. Through the above GUID, it can be found that the kernel will obtain many tables under UEFI, SMBIOS is one of them, corresponding to the global variable `common_tables` :

c AI generated projects 登录复制 run

```c
static const efi_config_table_type_t common_tables[] __initconst = {
    {ACPI_20_TABLE_GUID,            &efi.acpi20,        "ACPI 2.0"  },
    {ACPI_TABLE_GUID,           &efi.acpi,      "ACPI"      },
    {SMBIOS_TABLE_GUID,         &efi.smbios,        "SMBIOS"    },
    {SMBIOS3_TABLE_GUID,            &efi.smbios3,       "SMBIOS 3.0"    },  // 这里存放SMBIOS3.x的数据，可以看到前面还有一个SMBIOS2.x的
```

`efi` The corresponding structure in the above code is:

c AI generated projects 登录复制 run

```c
/*
 * All runtime access to EFI goes through this structure:
```

```c
 */
extern struct efi {
    const efi_runtime_services_t    *runtime;       /* EFI runtime services table */
    unsigned int            runtime_version;    /* Runtime services version */
    unsigned int            runtime_supported_mask;

    unsigned long           acpi;           /* ACPI table  (IA64 ext 0.71) */
    unsigned long           acpi20;         /* ACPI table  (ACPI 2.0) */
    unsigned long           smbios;         /* SMBIOS table (32 bit entry point) */
    unsigned long           smbios3;        /* SMBIOS table (64 bit entry point) */
```

The values for these parameters come from the function `efi_config_parse_tables()`:

c AI generated projects 登录复制 run

```c
int __init efi_config_parse_tables(const efi_config_table_t *config_tables, // 注意这个参数
                   int count,
                   const efi_config_table_type_t *arch_tables)
{
    const efi_config_table_64_t *tbl64 = (void *)config_tables;
    const efi_config_table_32_t *tbl32 = (void *)config_tables;
    const efi_guid_t *guid;
    unsigned long table;
    int i;

    pr_info("");
    for (i = 0; i < count; i++) {
        if (!IS_ENABLED(CONFIG_X86)) {
            guid = &config_tables[i].guid;
            table = (unsigned long)config_tables[i].table;
        } else if (efi_enabled(EFI_64BIT)) {
            guid = &tbl64[i].guid;
            table = tbl64[i].table;

            if (IS_ENABLED(CONFIG_X86_32) &&
                tbl64[i].table > U32_MAX) {
                pr_cont("\n");
                pr_err("Table located above 4GB, disabling EFI.\n");
                return -EINVAL;
            }
        } else {
            guid = &tbl32[i].guid;
            table = tbl32[i].table;
        }

        if (!match_config_table(guid, table, common_tables) && arch_tables)
            match_config_table(guid, table, arch_tables);
    }
```

The key point is to assign values to members in through `config_tables` the value in this variable `common_tables`, and the former is passed by BIOS, which is part of the UEFI System Table (the following is the BIOS code):

c AI generated projects 登录复制 run

```c
///
/// EFI System Table
///
typedef struct {
  /// 前面的略
  ///
  /// The number of system configuration tables in the buffer ConfigurationTable.
  ///
  UINTN                           NumberOfTableEntries;
  ///
  /// A pointer to the system configuration tables.
  /// The number of entries in the table is NumberOfTableEntries.
  ///
  EFI_CONFIGURATION_TABLE         *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

The `EFI_CONFIGURATION_TABLE` structure is as follows:

c AI generated projects 登录复制 run

```c
///
/// Contains a set of GUID/pointer pairs comprised of the ConfigurationTable field in the
/// EFI System Table.
///
typedef struct {
  ///
  /// The 128-bit GUID value that uniquely identifies the system configuration table.
  ///
  EFI_GUID    VendorGuid;
  ///
  /// A pointer to the table associated with VendorGuid.
  ///
  VOID        *VendorTable;
} EFI_CONFIGURATION_TABLE;
```

`efi_config_table_type_t` The first parameter is the GUID, and the second parameter is the address, which corresponds to the structure in the Linux code .

In summary, BIOS passes the Configuration Table in the UEFI System Table to the kernel, and the kernel traverses the table to find the address of the corresponding GUID, so that the data in the table corresponding to the GUID can be accessed. As for how BIOS passes this Configuration Table to the kernel, this part is not the focus of this article, so it will not be introduced; and how BIOS fills the Configuration Table of this System Table will be further explained in the implementation of SMBIOS later.

SMBIOS Implementation

The basic module for SMBIOS implementation under BIOS is edk2\MdeModulePkg\Universal\SmbiosDxe\SmbiosDxe.inf, and its entry is `SmbiosDriverEntryPoint()`. It mainly includes the following steps:

1. initialization `mPrivateData`:

```c
  mPrivateData.Signature           = SMBIOS_INSTANCE_SIGNATURE;
  mPrivateData.Smbios.Add          = SmbiosAdd;
  mPrivateData.Smbios.UpdateString = SmbiosUpdateString;
  mPrivateData.Smbios.Remove       = SmbiosRemove;
  mPrivateData.Smbios.GetNext      = SmbiosGetNext;
  mPrivateData.Smbios.MajorVersion = (UINT8)(PcdGet16 (PcdSmbiosVersion) >> 8);
  mPrivateData.Smbios.MinorVersion = (UINT8)(PcdGet16 (PcdSmbiosVersion) & 0x00ff);

  InitializeListHead (&mPrivateData.DataListHead);
  InitializeListHead (&mPrivateData.AllocatedHandleListHead);
  EfiInitializeLock (&mPrivateData.DataLock, TPL_NOTIFY);
```

It contains Protocol, List and Lock. The latter two are related to code implementation, while the first one is used by other modules after installation, which will be further introduced later.

2. Install the SMBIOS interface, which is the Protocol initialized in the previous step:

```c
  //
  // Make a new handle and install the protocol
  //
  mPrivateData.Handle = NULL;
  Status              = gBS->InstallProtocolInterface (
                              &mPrivateData.Handle,
                              &gEfiSmbiosProtocolGuid,
                              EFI_NATIVE_INTERFACE,
                              &mPrivateData.Smbios
                              );
```

3. Determine whether there is SMBIOS data, if so, add it to the SMBIOS data. The data is passed to this module via HOB. The reason for such HOB is that UEFI BIOS can be used as Payload in coreboot, Slimbootloader, etc. The latter is responsible for hardware initialization, so it contains hardware information. They package it into HOB and pass it to UEFI Payload, so that SMBIOS data can be used. The above operations come from the function `RetrieveSmbiosFromHob()`:

```c
EFI_STATUS
RetrieveSmbiosFromHob (
  IN EFI_HANDLE  ImageHandle
  )
{
  for (Index = 0; Index < ARRAY_SIZE (mIsSmbiosTableValid); Index++) {
    GuidHob = GetFirstGuidHob (mIsSmbiosTableValid[Index].Guid);
    if (GuidHob == NULL) {
      continue;
    }

    GenericHeader = (UNIVERSAL_PAYLOAD_GENERIC_HEADER *)GET_GUID_HOB_DATA (GuidHob);
    if ((sizeof (UNIVERSAL_PAYLOAD_GENERIC_HEADER) <= GET_GUID_HOB_DATA_SIZE (GuidHob)) && (GenericHeader->Length <= GET_GUID_HOB_DATA_SIZE (GuidHob))) {
      if (GenericHeader->Revision == UNIVERSAL_PAYLOAD_SMBIOS_TABLE_REVISION) {
        //
        // UNIVERSAL_PAYLOAD_SMBIOS_TABLE structure is used when Revision equals to UNIVERSAL_PAYLOAD_SMBIOS_TABLE_REVISION
        //
        SmBiosTableAdress = (UNIVERSAL_PAYLOAD_SMBIOS_TABLE *)GET_GUID_HOB_DATA (GuidHob);
        if (GenericHeader->Length >= UNIVERSAL_PAYLOAD_SIZEOF_THROUGH_FIELD (UNIVERSAL_PAYLOAD_SMBIOS_TABLE, SmBiosEntryPoint)) {
          if (mIsSmbiosTableValid[Index].IsValid ((VOID *)(UINTN)SmBiosTableAdress->SmBiosEntryPoint, &TableAddress, &TableMaximumSize, &MajorVersion, &MinorVersion)) {
            Smbios.Raw = TableAddress;
            Status     = ParseAndAddExistingSmbiosTable (ImageHandle, Smbios, TableMaximumSize, MajorVersion, MinorVersion);
            if (EFI_ERROR (Status)) {
              DEBUG ((DEBUG_ERROR, "RetrieveSmbiosFromHob: Failed to parse preinstalled tables from Guid Hob\n"));
              Status = EFI_UNSUPPORTED;
            } else {
              return EFI_SUCCESS;
            }
          }
        }
      }
    }
  }
}
```

The focus is on `ParseAndAddExistingSmbiosTable()` this function, which will call further functions `SmbiosAdd()`, which will build the SMBIOS table:

```c
  //
  // Some UEFI drivers (such as network) need some information in SMBIOS table.
  // Here we create SMBIOS table and publish it in
  // configuration table, so other UEFI drivers can get SMBIOS table from
  // configuration table without depending on PI SMBIOS protocol.
  //
  SmbiosTableConstruction (Smbios32BitTable, Smbios64BitTable);
```

Its implementation:

```c
VOID
EFIAPI
SmbiosTableConstruction (
  BOOLEAN  Smbios32BitTable,
  BOOLEAN  Smbios64BitTable
  )
{
  if (Smbios32BitTable) {

```

```
    Status = SmbiosCreateTable ((VOID **)&Eps);
    if (!EFI_ERROR (Status)) {
      gBS->InstallConfigurationTable (&gEfiSmbiosTableGuid, Eps);
    }
  }

  if (Smbios64BitTable) {
    Status = SmbiosCreate64BitTable ((VOID **)&Eps64Bit);
    if (!EFI_ERROR (Status)) {
      gBS->InstallConfigurationTable (&gEfiSmbios3TableGuid, Eps64Bit);
    }
  }
}
```

Taking SMBIOS3.x as an example, it creates the SMBIOS table:

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
SmbiosCreate64BitTable (
  OUT VOID  **TableEntryPointStructure
  )
{
  UINT8                           *BufferPointer;
  UINTN                           RecordSize;
  UINTN                           NumOfStr;
  EFI_STATUS                      Status;
  EFI_SMBIOS_HANDLE               SmbiosHandle;
  EFI_SMBIOS_PROTOCOL             *SmbiosProtocol;
  EFI_PHYSICAL_ADDRESS            PhysicalAddress;
  EFI_SMBIOS_TABLE_HEADER         *SmbiosRecord;
  EFI_SMBIOS_TABLE_END_STRUCTURE  EndStructure;
  EFI_SMBIOS_ENTRY                *CurrentSmbiosEntry;

  Status        = EFI_SUCCESS;
  BufferPointer = NULL;

  if (Smbios30EntryPointStructure == NULL) {
    //
    // Initialize the Smbios30EntryPointStructure with initial values.
    // It should be done only once.
    // Allocate memory at any address.
    //
    DEBUG ((DEBUG_INFO, "SmbiosCreateTable: Initialize 64-bit entry point structure\n"));
    Smbios30EntryPointStructureData.MajorVersion = mPrivateData.Smbios.MajorVersion;
    Smbios30EntryPointStructureData.MinorVersion = mPrivateData.Smbios.MinorVersion;
    Smbios30EntryPointStructureData.DocRev       = PcdGet8 (PcdSmbiosDocRev);
    Status                                       = gBS->AllocatePages (
                                                          AllocateAnyPages,
                                                          EfiRuntimeServicesData,
                                                          EFI_SIZE_TO_PAGES (sizeof (SMBIOS_TABLE_3_0_ENTRY_POINT)),
                                                          &PhysicalAddress
                                                          );
    if (EFI_ERROR (Status)) {
      DEBUG ((DEBUG_ERROR, "SmbiosCreate64BitTable() could not allocate Smbios30EntryPointStructure\n"));
      return EFI_OUT_OF_RESOURCES;
    }

    Smbios30EntryPointStructure = (SMBIOS_TABLE_3_0_ENTRY_POINT *)(UINTN)PhysicalAddress;

    CopyMem (
      Smbios30EntryPointStructure,
      &Smbios30EntryPointStructureData,
      sizeof (SMBIOS_TABLE_3_0_ENTRY_POINT)
      );
  }

  // 中间略

    Status = gBS->AllocatePages (
                    AllocateAnyPages,
                    EfiRuntimeServicesData,
                    EFI_SIZE_TO_PAGES (Smbios30EntryPointStructure->TableMaximumSize),
                    &PhysicalAddress
                    );
    if (EFI_ERROR (Status)) {
      DEBUG ((DEBUG_ERROR, "SmbiosCreateTable() could not allocate SMBIOS 64-bit table\n"));
      Smbios30EntryPointStructure->TableAddress = 0;
      return EFI_OUT_OF_RESOURCES;
    } else {
      Smbios30EntryPointStructure->TableAddress = PhysicalAddress;
      mPre64BitAllocatedPages                   = EFI_SIZE_TO_PAGES (Smbios30EntryPointStructure->TableMaximumSize);
    }
```

It is important to allocate memory from `EfiRuntimeServicesData` this type of memory segment because other memory types may not be accessible to the operating system.

Then install:

AI generated projects 登录复制

```
gBS->InstallConfigurationTable (&gEfiSmbios3TableGuid, Eps64Bit);
```

The so-called installation is actually just putting it in the Configuration Table in the UEFI System Table.

This is related to the previous step of getting SMBIOS data. However, there is still some content to be added.

- First is the code for allocating memory:

```c
    Status                                  = gBS->AllocatePages (
                                                    AllocateAnyPages,
                                                    EfiRuntimeServicesData,
                                                    EFI_SIZE_TO_PAGES (sizeof (SMBIOS_TABLE_3_0_ENTRY_POINT)),
                                                    &PhysicalAddress
                                                    );

    Status = gBS->AllocatePages (
                    AllocateAnyPages,
                    EfiRuntimeServicesData,
                    EFI_SIZE_TO_PAGES (Smbios30EntryPointStructure->TableMaximumSize),
                    &PhysicalAddress
                    );
```

There are several different allocations here. The first one contains a structure `SMBIOS_TABLE_3_0_ENTRY_POINT`, which is generally called an SMBIOS Entry. The following is the SMBIOS 3.x version:

```c
typedef struct {
  UINT8     AnchorString[SMBIOS_3_0_ANCHOR_STRING_LENGTH];
  UINT8     EntryPointStructureChecksum;
  UINT8     EntryPointLength;
  UINT8     MajorVersion;
  UINT8     MinorVersion;
  UINT8     DocRev;
  UINT8     EntryPointRevision;
  UINT8     Reserved;
  UINT32    TableMaximumSize;
  UINT64    TableAddress;
} SMBIOS_TABLE_3_0_ENTRY_POINT;
```

These members are described below:

Table 2- SMBIOS 3.0 (64-bit) Entry Point Structure

Offset	Name	Length	Description
00h	Anchor String	5 BYTEs	_SM3_, specified as five ASCII characters (5F 53 4D 33 5F).
05h	Entry Point Structure Checksum	BYTE	Checksum of the Entry Point Structure (EPS) This value, when added to all other bytes in the EPS, results in the value 00h (using 8-bit addition calculations). Values in the EPS are summed starting at offset 00h, for Entry Point Length bytes.
06h	Entry Point Length	BYTE	Length of the Entry Point Structure, starting with the Anchor String field, in bytes, currently 18h
07h	SMBIOS Major Version	BYTE	Major version of this specification implemented in the table structures (for example, the value is 0Ah for revision 10.22 and 02h for revision 2.1)
08h	SMBIOS Minor Version	BYTE	Minor version of this specification implemented in the table structures (for example, the value is 16h for revision 10.22 and 01h for revision 2.1)
09h	SMBIOS Docrev	BYTE	Identifies the docrev of this specification implemented in the table structures (for example, the value is 00h for revision 10.22.0 and 01h for revision 2.7.1).
0Ah	Entry Point Revision	BYTE	EPS revision implemented in this structure and identifies the formatting of offsets 0Bh and beyond as follows: 00h Reserved for assignment by this specification 01h Entry Point is based on SMBIOS 3.0 definition; 02h-FFh Reserved for assignment by this specification; offsets 0Ch-17h are defined per revision 01h
0Bh	Reserved	BYTE	Reserved for assignment by this specification, set to 0

Offset	Name	Length	Description
0Ch	Structure table maximum size	DWORD	Maximum size of SMBIOS Structure Table, pointed to by the Structure Table Address, in bytes. The actual size is guaranteed to be less or equal to the maximum size.
10h	Structure table address	QWORD	The 64-bit physical starting address of the read-only SMBIOS Structure Table, which can start at any 64-bit address. This area contains all of the SMBIOS structures fully packed together

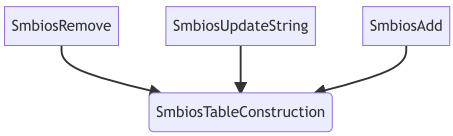
Since each member is relatively simple, I will not go into details here.

This structure is not large, but the actual allocated memory is passed `EFI_SIZE_TO_PAGES`, so the minimum size is 4K. `SMBIOS_TABLE_3_0_ENTRY_POINT` Then follows other SMBIOS data. However, 4K is not necessarily enough to store all SMBIOS data. In fact, the specification defines the maximum size of SMBIOS data:

```c
//
// The length of the entire structure table (including all strings) must be reported
// in the Structure Table Length field of the SMBIOS Structure Table Entry Point,
// which is a WORD field limited to 65,535 bytes.
//
#define SMBIOS_TABLE_MAX_LENGTH  0xFFFF
```

So when there is not enough data, the memory will be reallocated and the size used is `Smbios30EntryPointStructure->TableMaximumSize` the 4K aligned version.

- Secondly, `SmbiosTableConstruction()` it will be called by many SMBIOS interfaces:



Because the HOB that stores SMBIOS data does not necessarily exist, it is necessary to call the above basic interface to complete the final operation `SmbiosTableConstruction()`.

SMBIOS interface under BIOS

The SMBIOS interface is completed by a Protocol:

```c
struct _EFI_SMBIOS_PROTOCOL {
  EFI_SMBIOS_ADD              Add;
  EFI_SMBIOS_UPDATE_STRING    UpdateString;
```

```c
  EFI_SMBIOS_REMOVE           Remove;
  EFI_SMBIOS_GET_NEXT         GetNext;
  UINT8                       MajorVersion; ///< The major revision of the SMBIOS specification supported.
  UINT8                       MinorVersion; ///< The minor revision of the SMBIOS specification supported.
};
```

It `Add` is used to add SMBIOS types, `UpdateString` update existing SMBIOS types, `Remove` delete existing SMBIOS types, and `GetNext()` traverse SMBIOS types. These interfaces are relatively simple and will be explained through code examples.

Add

`Add` The interface is used to add a new SMBIOS type. Its interface is as follows:

c AI generated projects 登录复制 run

```c
/**
  Add an SMBIOS record.

  This function allows any agent to add SMBIOS records. The caller is responsible for ensuring
  Record is formatted in a way that matches the version of the SMBIOS specification as defined in
  the MajorRevision and MinorRevision fields of the EFI_SMBIOS_PROTOCOL.
  Record must follow the SMBIOS structure evolution and usage guidelines in the SMBIOS
  specification. Record starts with the formatted area of the SMBIOS structure and the length is
  defined by EFI_SMBIOS_TABLE_HEADER.Length. Each SMBIOS structure is terminated by a
  double-null (0x0000), either directly following the formatted area (if no strings are present) or
  directly following the last string. The number of optional strings is not defined by the formatted area,
  but is fixed by the call to Add(). A string can be a place holder, but it must not be a NULL string as
  two NULL strings look like the double-null that terminates the structure.

  @param[in]        This               The EFI_SMBIOS_PROTOCOL instance.
  @param[in]        ProducerHandle     The handle of the controller or driver associated with the SMBIOS information. NULL means no handle.
  @param[in, out]   SmbiosHandle       On entry, the handle of the SMBIOS record to add. If FFFEh, then a unique handle
                                       will be assigned to the SMBIOS record. If the SMBIOS handle is already in use,
                                       EFI_ALREADY_STARTED is returned and the SMBIOS record is not updated.
  @param[in]        Record             The data for the fixed portion of the SMBIOS record. The format of the record is
                                       determined by EFI_SMBIOS_TABLE_HEADER.Type. The size of the formatted
                                       area is defined by EFI_SMBIOS_TABLE_HEADER.Length and either followed
                                       by a double-null (0x0000) or a set of null terminated strings and a null.

  @retval EFI_SUCCESS                  Record was added.
  @retval EFI_OUT_OF_RESOURCES         Record was not added.
  @retval EFI_ALREADY_STARTED          The SmbiosHandle passed in was already in use.
**/
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_ADD)(
  IN CONST      EFI_SMBIOS_PROTOCOL     *This,
  IN            EFI_HANDLE              ProducerHandle OPTIONAL,
  IN OUT        EFI_SMBIOS_HANDLE       *SmbiosHandle,
  IN            EFI_SMBIOS_TABLE_HEADER *Record
  );
```

The important parameters are the last two. The value of the first parameter is fixed for the interface. `SmbiosHandle` The description of this macro is as follows: `Add` `SMBIOS_HANDLE_PI_RESERVED`

c AI generated projects 登录复制 run

```c
///
/// Reference SMBIOS 2.7, chapter 6.1.2.
/// The UEFI Platform Initialization Specification reserves handle number FFFEh for its
/// EFI_SMBIOS_PROTOCOL.Add() function to mean "assign an unused handle number automatically."
/// This number is not used for any other purpose by the SMBIOS specification.
///
#define SMBIOS_HANDLE_PI_RESERVED  0xFFFE
```

The second parameter `Record` is the SMBIOS type data that needs to be added.

The following is a code example for adding SMIBOS Type11:

c AI generated projects 登录复制 run

```c
EFI_STATUS
EFIAPI
SmbiosTestDxeEntry (
  IN  EFI_HANDLE                    ImageHandle,
  IN  EFI_SYSTEM_TABLE              *SystemTable
  )
{
  EFI_STATUS              Status = EFI_ABORTED;
  EFI_SMBIOS_PROTOCOL     *Smbios = NULL;
  UINTN                   StringSize = 0;
  UINTN                   Size = 0;
  EFI_SMBIOS_TABLE_HEADER *Record = NULL;
  CHAR8                   *Str = NULL;
  EFI_SMBIOS_HANDLE       SmbiosHandle = 0;

  Status = gBS->LocateProtocol (
                  &gEfiSmbiosProtocolGuid,
                  NULL,
                  (VOID **)(&Smbios)
                  );
  if (EFI_ERROR (Status)) {
    DEBUG ((EFI_D_ERROR, "[%a][%d] Failed. - %r\n", __FUNCTION__, __LINE__, Status));
    return Status;
  }

  Size = gSmbiosType11Template.Hdr.Length;
  StringSize = AsciiStrSize (gSmbiosType11Strings);
  Size += StringSize;
  Size += 1;

```

```c
  Record = (EFI_SMBIOS_TABLE_HEADER *)(AllocateZeroPool (Size));
  if (NULL == Record) {
    DEBUG ((EFI_D_ERROR, "[%a][%d] Out of memory\n", __FUNCTION__, __LINE__));
    return EFI_OUT_OF_RESOURCES;
  }

  CopyMem (Record, &gSmbiosType11Template, gSmbiosType11Template.Hdr.Length);
  Str = ((CHAR8 *)Record) + Record->Length;
  CopyMem (Str, gSmbiosType11Strings, StringSize);

  SmbiosHandle = SMBIOS_HANDLE_PI_RESERVED;
  Status       = Smbios->Add (
                           Smbios,
                           NULL,
                           &SmbiosHandle,
                           Record
                           );

  return Status;
}
```

Type11 is a relatively simple one in the SMBIOS specification, and its initial structure is:

c AI generated projects 登录复制 run

```c
///
/// OEM Strings (Type 11).
/// This structure contains free form strings defined by the OEM. Examples of this are:
/// Part Numbers for Reference Documents for the system, contact information for the manufacturer, etc.
///
typedef struct {
  SMBIOS_STRUCTURE    Hdr;
  UINT8               StringCount;
} SMBIOS_TABLE_TYPE11;
```

The corresponding code is `gSmbiosType11Template` :

c AI generated projects 登录复制 run

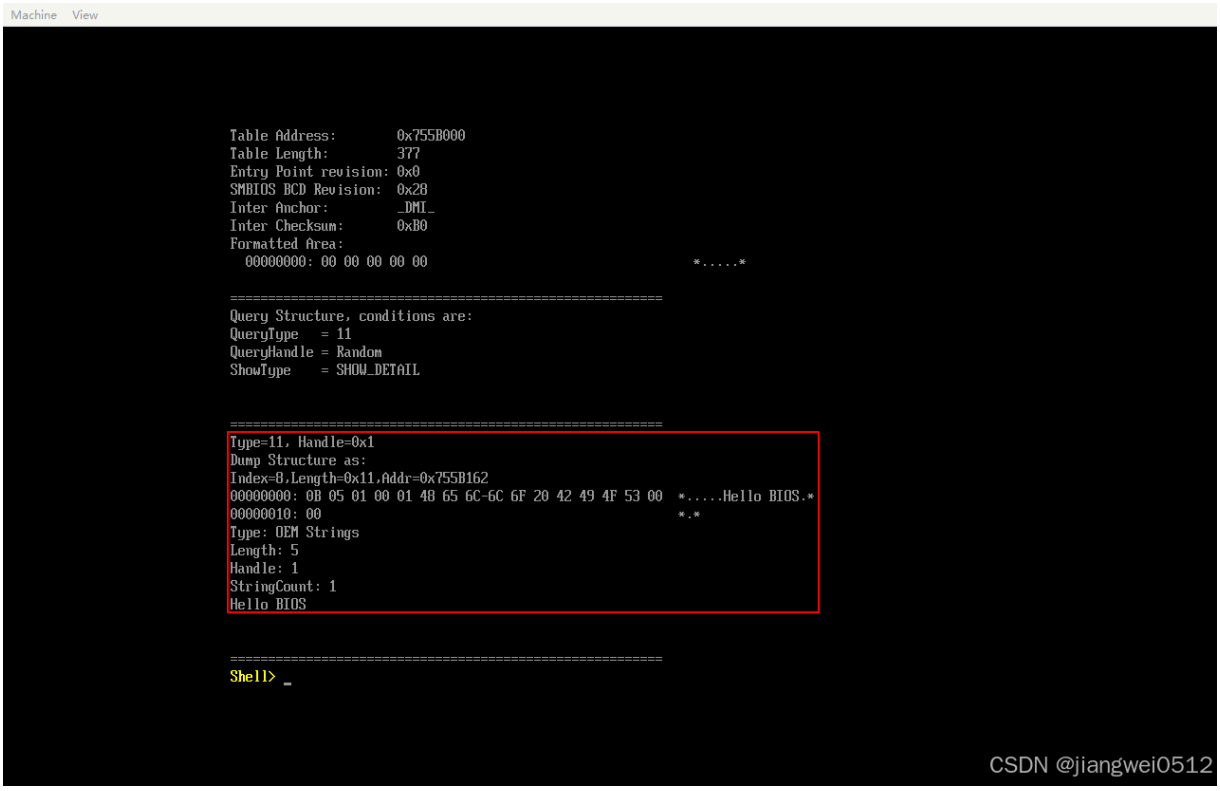
```c
SMBIOS_TABLE_TYPE11  gSmbiosType11Template = {
  { EFI_SMBIOS_TYPE_OEM_STRINGS, sizeof (SMBIOS_TABLE_TYPE11), 0 },
  1 // StringCount
};
```

The first line is the SMBIOS header. All SMBIOS Types are fixed. The next line is one `StringCount` . Here, only one string is added, so the value is 1. The content after that is not clearly defined, but it is obviously a string. According to the size of this string, the size of SMBIOS Type11 can be calculated. It should be noted that two are needed at the end `'\0'` , so there is the following code:

c AI generated projects 登录复制 run

```c
Size += 1;  // 字符串本身已经有一个'\0'了，还需要添加一个'\0'
```

Finally `Add` , SMBIOS Type11 is added to the SMBIOS database through the interface. You can view it through the command under BIOS Shell `smbiosview` :



GetNext

`GetNext()` Used to obtain the SMBIOS type, its prototype is as follows:

c AI generated projects 登录复制 run

```c
/**
  Allow the caller to discover all or some of the SMBIOS records.

```

```
  This function allows all of the SMBIOS records to be discovered. It's possible to find
  only the SMBIOS records that match the optional Type argument.

  @param[in]        This            The EFI_SMBIOS_PROTOCOL instance.
  @param[in, out]   SmbiosHandle    On entry, points to the previous handle of the SMBIOS record. On exit, points to the
                                    next SMBIOS record handle. If it is FFFEh on entry, then the first SMBIOS record
                                    handle will be returned. If it returns FFFEh on exit, then there are no more SMBIOS records.
  @param[in]        Type            On entry, it points to the type of the next SMBIOS record to return. If NULL, it
                                    indicates that the next record of any type will be returned. Type is not
                                    modified by the this function.
  @param[out]       Record          On exit, points to a pointer to the the SMBIOS Record consisting of the formatted area
                                    followed by the unformatted area. The unformatted area optionally contains text strings.
  @param[out]       ProducerHandle  On exit, points to the ProducerHandle registered by Add(). If no
                                    ProducerHandle was passed into Add() NULL is returned. If a NULL pointer is
                                    passed in no data will be returned.
  @retval EFI_SUCCESS               SMBIOS record information was successfully returned in Record.
                                    SmbiosHandle is the handle of the current SMBIOS record
  @retval EFI_NOT_FOUND             The SMBIOS record with SmbiosHandle was the last available record.
**/
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_GET_NEXT)(
  IN     CONST EFI_SMBIOS_PROTOCOL     *This,
  IN OUT       EFI_SMBIOS_HANDLE       *SmbiosHandle,
  IN           EFI_SMBIOS_TYPE         *Type              OPTIONAL,
  OUT          EFI_SMBIOS_TABLE_HEADER **Record,
  OUT          EFI_HANDLE              *ProducerHandle    OPTIONAL
  );
```

The more important ones are the three parameters in the middle: the first parameter `SmbiosHandle` is 1 as input `SMBIOS_HANDLE_PI_RESERVED`, which means finding the first matching SMBIOS type, and the output is a specific value representing the SMBIOS. If the returned value is `SMBIOS_HANDLE_PI_RESERVED`, it means that the traversal of SMBIOS is over; the second parameter `Type` indicates the SMBIOS type to be found. If not specified, it means directly returning the next SMBIOS of any type; the third parameter `Record` is the returned SMBIOS type data.

In this example, we get SMBIOS Type 0:

c AI generated projects 登录复制 run

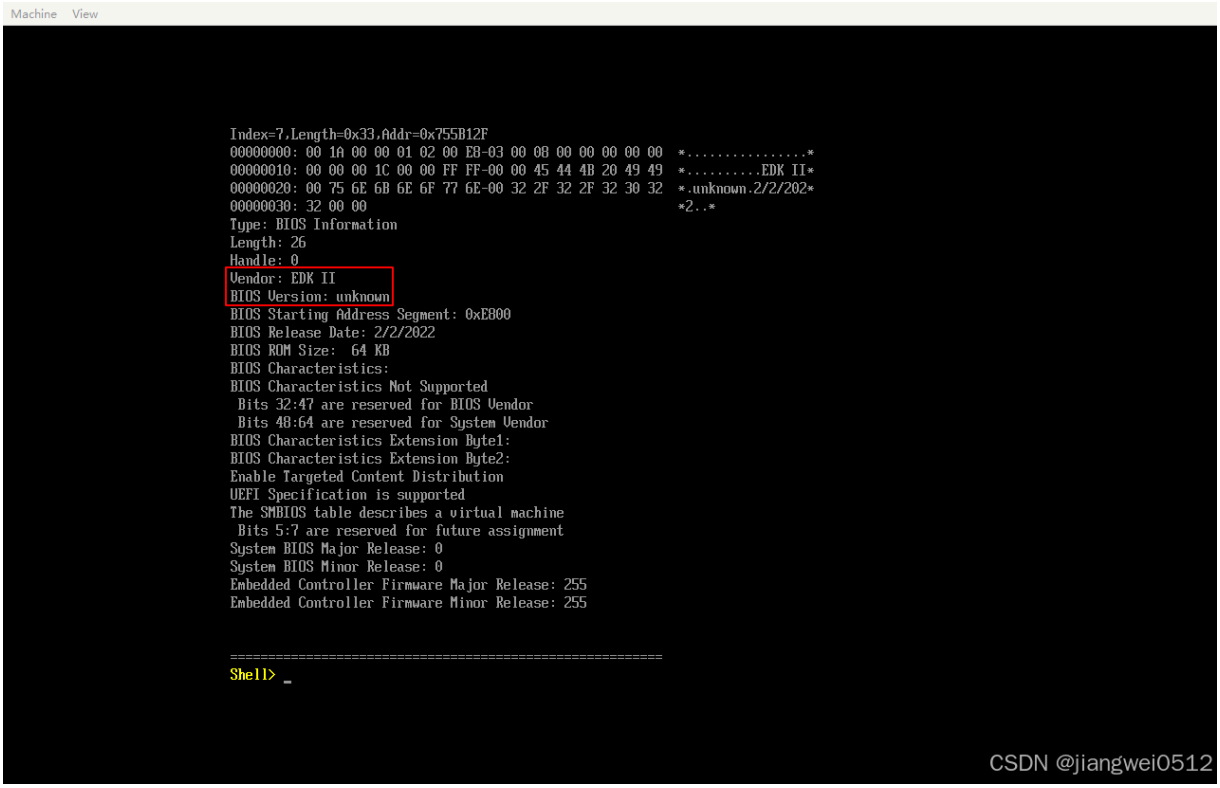
```c
  SmbiosHandle = SMBIOS_HANDLE_PI_RESERVED;
  Type = SMBIOS_TYPE_BIOS_INFORMATION;
  Status = Smbios->GetNext (
                   Smbios,
                   &SmbiosHandle,
                   &Type,
                   &Header,
                   NULL
                   );
  if (!EFI_ERROR (Status)) {
    Type0 = (SMBIOS_TABLE_TYPE0 *)Header;
    Str = ((CHAR8 *)Type0) + Type0->Hdr.Length;
    DEBUG ((EFI_D_ERROR, "Vendor: %a\n", Str));
    Str += AsciiStrSize (Str);
    DEBUG ((EFI_D_ERROR, "BiosVersion: %a\n", Str));
  }
```

The result is:

c AI generated projects 登录复制 run

```c
Vendor: EDK II
BiosVersion: unknown
```

Comparison with BIOS Shell:



The two are consistent.

One thing to note here is how to get the string field:

```c
    Type0 = (SMBIOS_TABLE_TYPE0 *)Header;
    Str = ((CHAR8 *)Type0) + Type0->Hdr.Length;
    DEBUG ((EFI_D_ERROR, "Vendor: %a\n", Str));
```

instead of:

```c
    DEBUG ((EFI_D_ERROR, "Vendor: %a\n", Type0->Vendor));
```

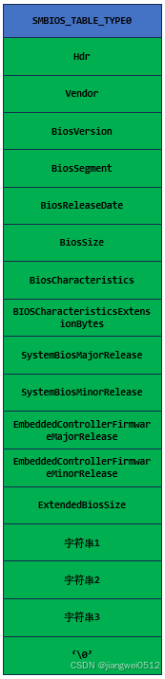
Because this `Vendor` is not a string, it is just a number:

```c
///
/// Text strings associated with a given SMBIOS structure are returned in the dmiStrucBuffer, appended directly after
/// the formatted portion of the structure. This method of returning string information eliminates the need for
/// application software to deal with pointers embedded in the SMBIOS structure. Each string is terminated with a null
/// (00h) BYTE and the set of strings is terminated with an additional null (00h) BYTE. When the formatted portion of
/// a SMBIOS structure references a string, it does so by specifying a non-zero string number within the structure's
/// string-set. For example, if a string field contains 02h, it references the second string following the formatted portion
/// of the SMBIOS structure. If a string field references no string, a null (0) is placed in that string field. If the
/// formatted portion of the structure contains string-reference fields and all the string fields are set to 0 (no string
/// references), the formatted section of the structure is followed by two null (00h) BYTES.
///
typedef UINT8 SMBIOS_TABLE_STRING;
```

Its value is 1, which means the first string. `BiosVersion` It is the second string, and so on. All the following `SMBIOS_TABLE_STRING` types represent an Index of the string, and Index also means the stacking order of the strings after this structure.

```c
///
/// BIOS Information (Type 0).
///
typedef struct {
  SMBIOS_STRUCTURE            Hdr;
  SMBIOS_TABLE_STRING         Vendor;          // 字符串1
  SMBIOS_TABLE_STRING         BiosVersion;     // 字符串2
  UINT16                      BiosSegment;
  SMBIOS_TABLE_STRING         BiosReleaseDate; // 字符串3
  UINT8                       BiosSize;
  MISC_BIOS_CHARACTERISTICS   BiosCharacteristics;
  UINT8                       BIOSCharacteristicsExtensionBytes[2];
  UINT8                       SystemBiosMajorRelease;
  UINT8                       SystemBiosMinorRelease;
  UINT8                       EmbeddedControllerFirmwareMajorRelease;
  UINT8                       EmbeddedControllerFirmwareMinorRelease;
  //
  // Add for smbios 3.1.0
  //
  EXTENDED_BIOS_ROM_SIZE      ExtendedBiosSize;
} SMBIOS_TABLE_TYPE0;
```

Therefore, the above structure corresponds to the real SMBIOS data as follows:



Note that each string `'\0'` ends with , and there is one at the end of the SMBIOS data `'\0'` . Therefore, in order to print these strings, you need to use the pointer + string length method.

UpdateString

`UpdateString` The interface is used to update the original SMBIOS type, and its prototype is as follows:

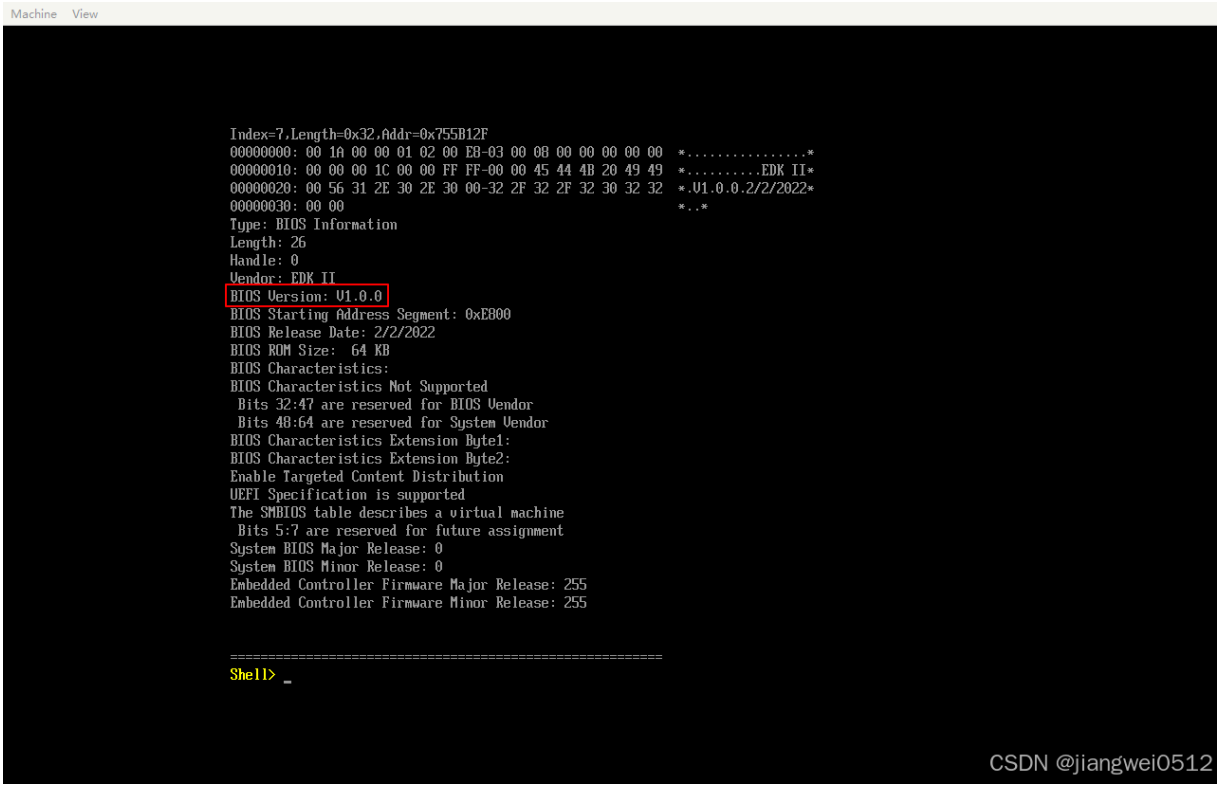
```c
/**
  Update the string associated with an existing SMBIOS record.

  This function allows the update of specific SMBIOS strings. The number of valid strings for any
  SMBIOS record is defined by how many strings were present when Add() was called.

  @param[in]    This            The EFI_SMBIOS_PROTOCOL instance.
  @param[in]    SmbiosHandle    SMBIOS Handle of structure that will have its string updated.
  @param[in]    StringNumber    The non-zero string number of the string to update.
  @param[in]    String          Update the StringNumber string with String.

  @retval EFI_SUCCESS           SmbiosHandle had its StringNumber String updated.
  @retval EFI_INVALID_PARAMETER SmbiosHandle does not exist.
  @retval EFI_UNSUPPORTED       String was not added because it is longer than the SMBIOS Table supports.
  @retval EFI_NOT_FOUND         The StringNumber.is not valid for this SMBIOS record.
**/
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_UPDATE_STRING)(
  IN CONST EFI_SMBIOS_PROTOCOL *This,
  IN       EFI_SMBIOS_HANDLE   *SmbiosHandle,
  IN       UINTN               *StringNumber,
  IN       CHAR8               *String
  );
```

The more important ones are the following three parameters: the first parameter `SmbiosHandle` is `GetNext()` the characteristic value corresponding to the obtained SMBIOS type; the second parameter is the index of the corresponding string in the SMBIOS type (starting from 1, but in fact there is no need to specify a special value, which will be explained later); the third parameter is the string to be updated.

In the GetNext section, you can see the BIOS Version in the result of getting SMBIOS Type0 `unknown` . Here, we will write code to update the value.

```c
    StringIndex = Type0->BiosVersion;
    Status = Smbios->UpdateString (
                      Smbios,
                      &SmbiosHandle,    // 通过GetNext()返回的值
                      &StringIndex,
                      "V1.0.0"  // 更新之后的字符串
                      );
```

This code directly uses it `Type0->BiosVersion` as the Index, so you don't need to specify the value yourself. The result after the update:



It should be noted that this interface is only used to update the string in SMBIOS, so how to update the ordinary interface without string? The answer is very simple, `GetNext()` just modify it directly after obtaining the structure.