**Проектирование ОС и компонентов**

Загрузчик ОС UEFI (анализ возможностей)



Что такое UEFI?

UEFI или (унифицированный) расширяемый интерфейс прошивки - это спецификация платформ x86, x86-64, ARM и Itanium, которая определяет программный интерфейс между операционной системой и прошивкой / BIOS платформы. Оригинальная EFI была разработана в середине 1990-х годов компанией Intel для использования в разработке прошивки / BIOS для платформ Itanium.

**OS**

**Loader**

* **Unified (EFI) / Extensible Firmware Interface (EFI)**

**Compatibility**

* **UEFI - это спецификация интерфейса**
* **Модульная и расширяемая**

**Provide efficient Option ROM Replacement**

U**EFI**

**Общий источник для многопроцессорных архитектур**

* **Дополняет существующие интерфейсы**

**BIOS**

**Hardware**

Unified EFI (UEFI) Forum [www.uefi.org](http://www.uefi.org)

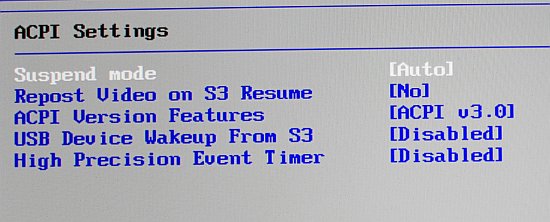
В 2005 году Intel перенесла эту спецификацию в новую рабочую группу под названием Unified EFI Forum, состоящую из таких компаний, как AMD, Microsoft, Apple и Intel. Все современные ПК поставляются с прошивкой UEFI, а UEFI широко поддерживается как коммерческими, так и открытыми исходными кодами. Обратная совместимость предоставляется для устаревших операционных систем.

* Промоутеры
  + OEMs: Dell, HP, IBM, Lenovo
  + IBVs: AMI, Insyde, Phoenix
  + AMD, Apple, Intel, Microsoft
* Спецификация UEFI
  + Спецификация EFI 1.10 была предоставлена Форумом Intel и Microsoft для использования в качестве исходного проекта
  + Выпущена спецификация UEFI 2.0 - 2.3.
  + Intel внесла свой вклад в EFI 1.10 SCT, который используется в качестве начальной базы для UEFI

Поддержка загрузочного устройства:

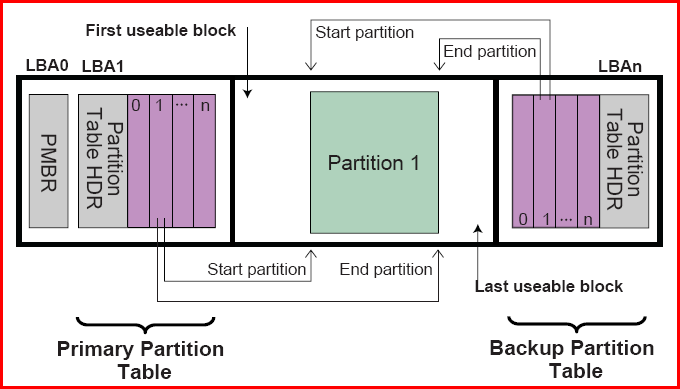
* Жесткий диск
* Съемные медиа
  + CD-ROM, DVD-ROM
  + El Torito 1.0 "Без эмуляции“
  + Флоппи-дисковод, USB-накопитель и т. д.
* Сеть
  + Спецификация поддержки PXE BIOS (провод для управления)
  + ISCSI
* Будущие среды средствами расширяемости

Эти технологии не рассматривается UEFI

* Инициализация памяти
* Восстановление
* Обновление FLASH
* ACPI S3
* Инициализация платформы
* Режим управления системой (SMM)
* Настроить

Понимание GPT

**GPT** — стандарт формата размещения [таблиц разделов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D1%80%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%BE%D0%B2) на физическом [жестком диске](https://ru.wikipedia.org/wiki/%D0%96%D0%B5%D1%81%D1%82%D0%BA%D0%B8%D0%B9_%D0%B4%D0%B8%D1%81%D0%BA). Он является частью [Расширяемого микропрограммного интерфейса](https://ru.wikipedia.org/wiki/Extensible_Firmware_Interface) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Extensible Firmware Interface, EFI) — стандарта, предложенного [Intel](https://ru.wikipedia.org/wiki/Intel) на смену [BIOS](https://ru.wikipedia.org/wiki/BIOS). EFI использует GPT там, где BIOS использует [Главную загрузочную запись](https://ru.wikipedia.org/wiki/%D0%93%D0%BB%D0%B0%D0%B2%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BE%D1%87%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C)

* Разбиение таблицы разделов GUID (GPT) требуется для загрузки UEFI
* Информация таблицы разделов, хранящаяся в заголовке GPT.
* Сохраняет запись MBR в первом секторе диска.
* Использует логическую адресацию блоков.
* Обеспечивает резервирование, запись заголовка и таблицы разделов как в начале, так и в конце.

Преимущества GPT по сравнению с таблицей разделов MBR:

**Master Boot Record** (**MBR**) — это первый сектор (первые 512 байт) устройства хранения данных. **MBR** - это не партиция, а участок памяти, зарезервированный для загрузчика операционной системы и таблицы разделов диска.

* 64-разрядная адресация логических блоков.
* Поддерживает неограниченное количество разделов
* Использует основную и резервную таблицы для резервирования.
* Использует поля номера версии и размера для будущего расширения.
* Использует поля CRC32 для улучшения целостности данных.
* Определяет GUID для уникальной идентификации каждого раздела.
* Использует GUID и атрибуты для определения типа содержимого раздела.
* Каждый раздел содержит 36 символов Unicode, читаемых человеком.
* Никакой магический код не должен выполняться как часть загрузки
* Устраняет проблему 2,2 терабайта

Структуры данных UEFI - Системная таблица UEFI

Input Console

Active Consoles

Output Console

Standard Error Console

EFI Runtime Services Table

Variable Services

Real Time Clock Services

Reset Services

Status Code Services

Virtual Memory Services

EFI System Table

**Boot Service Data Structures**

Protocol Interface

Protocol Interface

Protocol Interface

Protocol Interface

Protocol Interface

Protocol Interface

Handle Database

EFI Boot Services Table

Task Priority Level Services

Memory Services

Event and Timer Services

Protocol Handler Services

Image Services

Driver Support Services

Version Information

EFI Specification Version

Firmware Vendor

Firmware Revision

**Runtime Data Structures**

Handles: (Дескриптор)

* Все протоколы имеют дескриптор, который связан с протоколом
* Каждое устройство и исполняемый образ в UEFI имеют протокол обработки в базе данных дескриптора
* Каждое устройство загрузки должно иметь протокол пути устройства для его описания

Протоколы (API)

GUID, Структура интерфейса, Сервисы DEVICE\_PATH, DEVICE\_IO, BLOCK\_IO, DISK\_IO, FILE\_SYSTEM, SIMPLE\_INPUT, SIMPLE\_TEXT\_OUTPUT, SERIAL\_IO, PXE\_BC, SIMPLE\_NETWORK, LOAD\_FILE, UNICODE\_COLLATION

**Handle**

**Function Ptr 1**

**Private Data**

**GUID**

**Function Ptr 2**

**. . .**

**Protocol Interface**

**GUID 1**

**GUID 2**

**Function 1**

**Function 2**

**. . .**

**. . .**

**BlkIo->ReadBlocks(BlkIo, …)**

Построение приложения UEFI:

У приложения EFI PE нет никаких фантазий в Windows PE, таких как таблицы символов, экспорт, данные для обработки статических исключений и т. Д. У него даже нет импорта - все, что вам понадобится в приложении EFI, передается как аргумент для точки входа Функции. Единственное, что необходимо помимо данных и кода - это перемещения. Таким образом, это простейшее приложение EFI:

* #include <efi.h>

EFI\_STATUS main(EFI\_HANDLE ImageHandle, EFI\_SYSTEM\_TABLE \*SystemTable)

{

return EFI\_SUCCESS;

}

EFI файл:

* Например, на системе UEFI с установленной Windows 10 файл EFI будет располагаться в следующем месте на этом скрытом разделе:

\EFI\boot\bootx64.efi или \EFI\boot\bootia32.efi

* На некоторых компьютерах Windows файл winload.efi действует как загрузчик и обычно хранится в следующем месте:

C:\Windows\System32\Boot\winload.efi

* Например, на установочном DVD-диске Windows 10 или ISO существуют следующие два файла, которые менеджер загрузки UEFI вашего компьютера быстро обнаружит:

D:\efi\boot\bootx64.efi или D:\efi\boot\bootia32.efi

GUID:

* Глобальная уникальная идентификация

128-битное количество, определенное спецификацией Wired for Management WfM 2.0

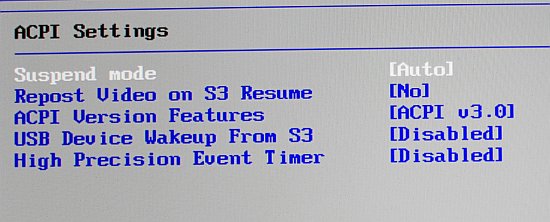
* Используется для идентификации протоколов

1: 1 с интерфейсами

* Регулировать механизм расширения

Документировано в спецификации

Эти технологии не рассматриваемая UEFI

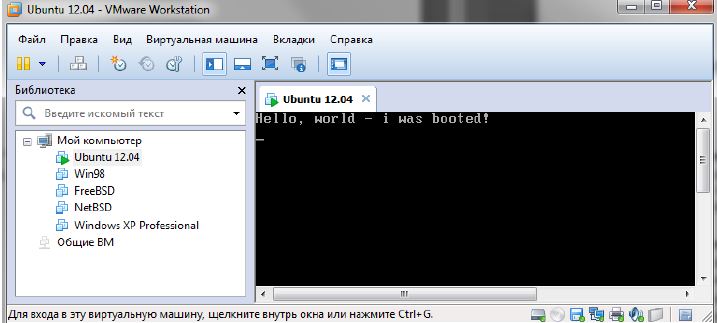
* Инициализация памяти
* Восстановление
* Обновление FLASH
* ACPI S3
* Инициализация платформы
* Режим управления системой (SMM)
* Настроить

Рассматривается пример кода:

|  |
| --- |
| ;Пример реализации вывода сообщения из MBR  use16  org 0x7C00  ; Clean registr  xor ax, ax  mov es, ax  mov ds, ax  mov ss, ax  mov sp, 0x1000  ; Create video rejim for clean monitor  mov ax, 3  int 10h  ; Hello on the monitor  mov si, mHello  call print  ; Write just  die: jmp short die  mHello db 'Hello, world - i was booted!',10,13,0  ; subprogram present  print:  cld  pusha  .PrintChar:  lodsb  test al, al  jz short .Exit  mov bl, 7  int 10h  jmp short .PrintChar  .Exit:  popa  ret |

Компилируем программу:

>nasm –f bin boothi.asm –o boothi



Рассмотрим 64-битной версии UEFI, потому что 32-разрядная версия в этой области мало используется (скорее всего, из-за решения Microsoft не поддерживать UEFI в 32-разрядной версии Vista). Итак, понадобится 64-разрядный процессор (но не 64-разрядная ОС, можно использовать любую 32-разрядную ОС).

Для продолжения разработки UEFI понадобятся два пакета разработки: EFI Development Kit (EDK) и EFI Toolkit.

Первый пакет, EFI Development Kit содержит исходный код TianoCore (публичная часть ссылочного исходного кода UEFI от Intel), а также множество примеров и двоичных файлов EFI Shell (об этом мы поговорим позже), все готовые к созданию с хорошим make система. Он даже может быть встроен в эмулятор Win32 UEFI для более удобной разработки и тестирования.

Второй пакет, EFI Toolkit - набор дополнительных приложений UEFI, таких как FTP-клиент, порт Python, текстовый редактор и т. Д.

Наряду с заголовками также потребуется документация интерфейса UEFI. Это можно загрузить после заполнения простой формы по адресу http://www.uefi.org/specs. Это официальная спецификация UEFI.

Кроме спецификации UEFI, есть еще один документ под названием «Инициализация платформы». Это описывает реализацию этапов инициализации UEFI (до загрузки драйверов и приложений), и что более важно для нас, также описывает интерфейс подпрограмм, используемых для реализации UEFI. Можно понять эту спецификацию инициализации платформы как описание реализации Tiano UEFI. Он обеспечивает более низкий уровень управления, чем UEFI, в случаях, когда такое управление необходимо. Строго говоря, кто-то может реализовать спецификацию UEFI, не внедряя ничего из спецификации инициализации платформы. Но не менее важное понадобится 64-битный компилятор C. Можно использовать Microsoft Visual C ++.

Минимально базовый код первичной заагрузчика показано на файле boot.asm

|  |
| --- |
| %include "io.inc"  section .text  global CMAIN  CMAIN:  BITS 16  org0x700  ; Обнулим регистры, установим стек  cli  mov ax, 0  mov ds, ax  mov es, ax  mov ss, ax  mov sp, 0x700  sti  ;Сообщение о приветствии  mov si, msg\_start  call kputs  ; Сообщение о переходе в защищенный режим  mov si, msg\_entering\_pmode  call ps  ; Отключение курсора (просто так)  mov ah, 1  mov ch, 0x20  int 0x10  ; Установим базовый вектор контроллера прерываний в 0x20  mov al,00010001b  out 0x20,al  mov al,0x20  out 0x21,al  mov al,00000100b  out 0x21,al  mov al,00000001b  out 0x21,al  ; Отключим прерывания  cli  ; Загрузка регистра GDTR:  lgdt [gd\_reg]  ; Включение A20:  in al, 0x92  or al, 2  out 0x92, al  ; Установка бита PE регистра CR0  mov eax, cr0  or al, 1  mov cr0, eax  ; С помощью длинного прыжка мы загружаем селектор нужного сегмента в регистр CS  jmp 0x8: \_protect  ps:  pusha  .loop:  lodsb  test al, al  jz .quit  mov ah, 0x0e  int 0x10  jmp short .loop  .quit:  popa  ret  ; Следующий код — 32-битный  [BITS 32]  ;При переходе в защищенный режим, сюда будет отдано управление  \_protect:  ; Загрузим регистры DS и SS селектором сегмента данных  mov ax, 0x10  mov ds, ax  mov es, ax  mov ss, ax  ; Наше ядро слинковано по адресу 2мб, переносим его туда. ker\_bin — метка, после которой вставлено ядро  mov esi, ker\_bin  ; Адрес, по которому копируем  mov edi, 0x200000  ; Размер ядра в двойных словах (65536 байт)  mov ecx, 0x4000  rep movsd  ; Ядро скопировано, передаем управление ему  jmp 0x200000  gdt:  dw 0, 0, 0, 0  ; Нулевой дескриптор  db 0xFF  ; Сегмент кода с DPL=0 Базой=0 и Лимитом=4 Гб  db 0xFF  db 0x00  db 0x00  db 0x00  db 10011010b  db 0xCF  db 0x00  db 0xFF  ; Сегмент данных с DPL=0 Базой=0 и Лимитом=4Гб  db 0xFF  db 0x00  db 0x00  db 0x00  db 10010010b  db 0xCF  db 0x00  ; Значение, которое мы загрузим в GDTR:  gd\_reg:  dw 8192  dd gdt  msg\_start: db «Get fun! New loader is on», 0x0A, 0x0D, 0  msg\_epm: db «Protected mode is greeting you», 0x0A, 0x0D, 0  xor eax, eax  ret |