OULUN AMMATTIKORKEAKOULU

Arttu Waldén

Customizing BIOS/UEFI From OS Using EFI Variables

Customizing BIOS/UEFI From OS Using EFI Variables

Arttu Walden
Bachelor's thesis
Autumn 2023
Degree Programme in Information Technology, Option of Software Development
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Arttu Waldén
Title of the thesis: Customizing BIOS/UEFI From OS Using EFI Variables
Thesis examiner(s): Jukka Jauhiainen
Term and year of thesis completion: Autumn, 2023 Pages: 35

BIOS/UEFI firmware serves as the backbone for system initialization and configuration, but it often lacks the flexibility to be dynamically customized once the operating system is running. Traditional means of accessing and modifying these settings often require a reboot and direct interaction with firmware interfaces, which is not very convenient. In the scope of a new device development project, an opportunity arose to implement a more user-friendly approach to BIOS customization.

The primary objective of this thesis was to enable BIOS/UEFI customization from within the OS. This involved altering the BIOS code to read EFI variables and creating a Windows application to set these variables. Data structure known as the Type-Length-Value (TLV) format was used to make data handling more efficient.

The thesis focused on creating BIOS customization through software changes to the existing BIOS codebase. The BIOS was extended with additional code, primarily written in C, to read EFI variables set by the operating system. A Windows application was also developed, utilizing the Type-Length-Value (TLV) data format, which allows for the modification of these EFI variables. The BIOS code was designed to read these variables and implement changes accordingly.

Keywords: BIOS, UEFI, Customization, TLV, EFI Variable

CONTENTS

VOCABULARY

- **UEFI (Unified Extensible Firmware Interface)**:
    - The modern replacement for BIOS, it initializes hardware components and starts the operating system loader.
- **BIOS (Basic Input/Output System)**:
    - Older firmware that initializes the hardware and loads the operating system.
- **OS (Operating System)**:
    - Software that manages hardware and allows users to run applications. Examples include Windows, Linux, and macOS.
- **PEI (Pre-EFI Initialization)**:
    - Early stage in UEFI booting responsible for initializing minimal hardware like CPU and memory.
- **DXE (Driver Execution Environment)**:
    - A stage in UEFI booting where drivers for the motherboard's hardware components are loaded.
- **ACPI (Advanced Configuration and Power Interface)**:
    - A standard for power management and device enumeration, commonly used in operating systems.
- **GUI (Graphical User Interface)**:
    - A type of user interface that allows users to interact with computers through graphical elements like windows, icons, and buttons.
- **EFI (Extensible Firmware Interface)**:
    - Older name for UEFI. It is a specification for the interface between the operating system and the firmware.
- **TLV (Type-Length-Value)**:
    - A data representation format where data is stored as a type identifier, followed by the length of the data, and then the actual data.

- **X86**:
    - A family of instruction set architectures initially developed by Intel, widely used in desktop and server CPUs.
- **I/O (Input/Output)**:
    - Refers to any operation, device, or channel that either inputs data into or outputs data from a computer.
- **CPU (Central Processing Unit)**:
    - The primary component of a computer that performs most of the processing inside the computer.
- **GUID (Globally Unique Identifier)**:
    - A unique reference number used in computing, often for identifying objects within a system.

1 INTRODUCTION

Unified Extensible Firmware Interface (UEFI) serves as a bridge between a computer's hardware and its operating system (OS). Replacing the older Basic Input/Output System (BIOS), UEFI offers advantages such as quicker startup times, the ability to handle larger storage devices, and better security features. (1.) Additionally, UEFI has open-source implementations that provide opportunities for community involvement and customization. However, despite its advantages, UEFI settings are generally static once the OS is running. The traditional method for changing these settings involves rebooting the computer and accessing the UEFI interface. This process is often challenging for those not well-versed in technology and tends to be time-consuming.

The aim of this thesis is to provide a better and more efficient way to change UEFI settings directly from the operating system. While a system restart is still required to implement these changes, the utilization of Extensible Firmware Interface (EFI) variables and the Type-Length-Value (TLV) data format allows for settings to be changed without entering the UEFI interface.

The motivation for developing this customization approach stems from the need for devices that can be easily adapted to meet various operational requirements. By making UEFI settings customizable from within the operating system, the process becomes more user-friendly and efficient. The primary way for customers to customize UEFI is through batch file execution for predefined settings, but for our newest device, we are also aiming for a more dynamic, user-defined customization, where the UEFI is customizable via a graphical user interface (GUI).

The scope of this thesis is to delve into the technical modifications made to the BIOS codebase to enable customizations of UEFI settings. This involves detailing the added code, primarily written in C, that allows for the reading of EFI variables set by the application. The thesis will also cover the development of a windows application designed to interact with these EFI variables and how these two parts come together.

2 THEORY

2.1 UEFI and BIOS Overview

What does "UEFI" stand for? The Unified Extensible Firmware Interface is a global standard outlining a uniform method for computer firmware to communicate with operating systems loaded into its memory. It offers multiple security functionalities such as Secure Boot and Secure Update among others. It is Managed by the nonprofit UEFI Forum, which has an open membership policy. UEFI defines the interface, but not the implementation. There are many different system firmware implementations out there that comply with this UEFI standard interface. (2.)

Originating in the late 1970s, BIOS served to load operating systems and offer basic I/O services on the initial x86 PCs, eventually becoming an unofficial standard for such machines. By the late 1990s, however, emerging complex processor types made it increasingly impractical to stick with BIOS, which was closely linked to x86 architecture. To address this, the Extensible Firmware Interface was developed to revamp the communication between BIOS and operating systems on Intel CPU-based systems. The UEFI Forum was established in 2005 to make the standard more accessible, expand its scope, and allow community participation in its maintenance and future development. (2.)

Various open-source implementations of UEFI include Tianocore, which builds on the EDK II codebase, and UBoot, often used in embedded systems. LinuxBoot uses Linux itself for system firmware, while coreboot offers features comparable to UEFI. Proprietary options like Insydes InsydeH2O and AMI Aptio V are also available. (2.)

Figure 1 illustrates the differences in boot sequences between the legacy BIOS boot process and the modern UEFI boot process. BIOS boot begins with the MBR that leads to a boot loader, which then loads the OS kernel. UEFI boot skips the MBR and uses an EFI System Partition where EFI boot loaders directly load the OS kernel, simplifying the process.

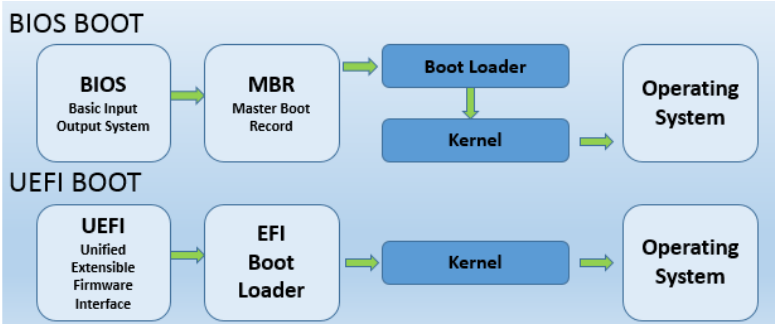


FIGURE 1. BIOS Boot vs UEFI Boot

2.2 EFI Variables

EFI variables are an important component of the Unified Extensible Firmware Interface (UEFI), serving as a standardized method for storing data that persists between boots. Managed by the firmware, these variables enable the operating system and applications to communicate with the UEFI environment. They are often used for a variety of tasks, ranging from system configuration to boot order settings.

The EFI variable namespace is organized using a combination of GUIDs (Globally Unique Identifiers) and variable names to prevent overlaps between variables from different vendors. Attributes can be assigned to EFI variables to regulate their visibility and accessibility in both the UEFI firmware environment and the operating system. Importantly, certain variables can be designated for "runtime access," allowing them to be read or modified by the operating system even after booting is complete. (3.)

EFI variables also play a crucial role in the UEFI Secure Boot process, where they store key databases used to validate the signatures of bootloaders and drivers. Given the sensitivity of some of these variables, certain measures like authentication and time-based updates can be enforced.

Multiple implementations exist for managing EFI variables, ranging from proprietary solutions to open-source libraries. The handling of EFI variables can vary significantly between different UEFI implementations, although they all adhere to the same foundational specifications outlined by the UEFI Forum. (4.)

2.3 Type-Length-Value (TLV) Data format

The Type-Length-Value (TLV) data format is a widely adopted encoding scheme designed for efficient data serialization and transmission. It is comprised of three core components: Type, which identifies the kind of data being transmitted; Length, indicating the size of the accompanying value in bytes; and Value, which contains the actual data payload. The format is particularly well-suited for nested or hierarchical data structures, as well as for streaming or packet-based communication systems. (5.)

In the context of UEFI firmware and EFI variables, the TLV data format can offer a structured approach to data storage and retrieval. This becomes particularly important for configuration settings or other complex data types that may need to be accessed both by the firmware and the operating system. TLV formatting ensures that each piece of data can be accurately extracted and manipulated, thereby providing a robust and extensible framework for complex data management tasks.

The TLV format's straightforward yet flexible structure makes it a popular choice for various applications, from network protocols to file formats. Its design enables easy parsing and modification, contributing to its widespread use in both software and hardware implementations. (5.)

In this project, using the Type-Length-Value (TLV) data format made managing EFI variables a lot simpler. The goal was to fit most custom settings into one EFI variable, and TLV helped to sort and use the data more easily.

First, the "Type" section indicates the category of the data, enabling a quick search for the desired settings without going through the entire dataset. Next, the "Length" section specifies the size of the data, improving the speed and efficiency of data reading, which is important when the system is starting up and needs to work fast.

Last, the "Value" part holds the actual settings or data. Because TLV is organized, it is easy to add or change data without messing up what is already there. Also, other tools or parts of the system can easily read the EFI variable if they need to.

Using TLV made handling lots of different settings in one EFI variable much easier and quicker, both for the firmware and the operating system. One of the biggest advantages was being able to clear almost all of the customizations with a single command using the application.

2.4 Boot Phases: PEI and DXE

When a device with UEFI firmware boots up, it goes through different steps or phases (Figure 2). Two important phases are PEI (Pre-EFI Initialization) and DXE (Driver Execution Environment). When working on firmware, it is good to know about these phases because they show what can and cannot be done at different times during the boot-up. (6.)

2.4.1 PEI (Pre-EFI Initialization)

PEI is the first phase in the UEFI boot process. In this phase, the system sets up the basic hardware like the CPU and memory. It is all about getting the core components ready for the more detailed work that comes later. PEI is simple and does not offer many extra features. (6.)

2.4.2 DXE (Driver Execution Environment)

After PEI comes the DXE phase. Here, the system goes into more detailed hardware setup. This includes working with things like hard drives, network cards, and other add-ons. DXE gives more options for doing complex tasks than PEI does. (6.)

2.4.3 Why PEI and DXE Matter

Understanding the PEI and DXE phases is beneficial for firmware development. These phases define the appropriate timing for utilizing various tools and accessing hardware components. For instance, certain hardware elements may only be accessible during the DXE phase and not during PEI.

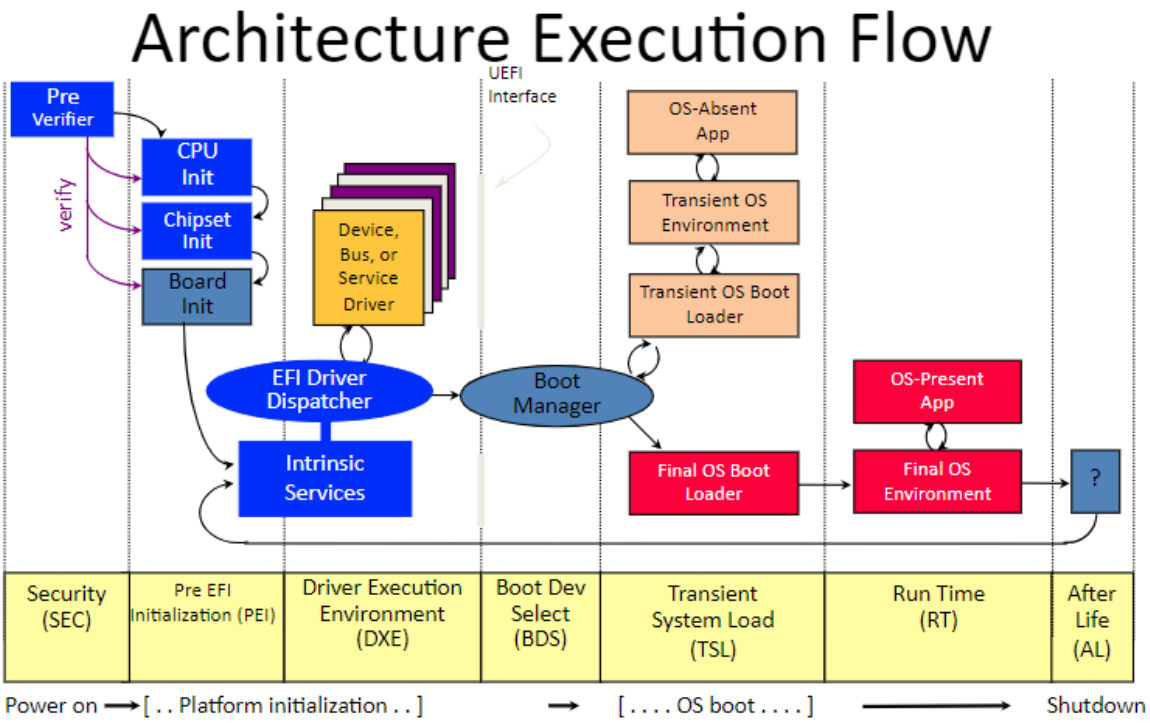


FIGURE 2. Platform Initialization Boot Phases

3 DESIGN

3.1 BIOS Codebase Modifications

The first significant step in modifying the BIOS codebase was the creation of a specialized library. The primary function of this library is to read EFI variables, parse their TLV-formatted data, and then employ this data in specific functions. These specialized functions are subsequently invoked at the appropriate execution points to enact various customizations.

3.1.1 Header File

The header file is crucial for data structure formatting, and it incorporates the use of `#pragma pack(1)` for byte-level packing, which ensures the correct memory layout for structures. It defines two core structures (Figure 3):

- **CUST_DATA_HEADER**: Contains metadata about the customization data, including a magic number that validates the data integrity, and version numbers.
- **TLV_HEADER**: Defines the Type-Length-Value (TLV) data format, essential for customization data parsing.

```
#pragma pack(1)
struct CUST_DATA_HEADER
{
  UINT32 magic;
  UINT16 major_version;
  UINT16 minor_version;
};
#pragma pack()

#pragma pack(1)
typedef struct TLV_HEADER
{
  UINT16 type;
  UINT16 length;
} TLV_HEADER;
#pragma pack()
```

FIGURE 3. Custom data structure and TLV header structure

3.1.2 Core Functions

The library consists of three primary functions: `ReadCustomData`, `FindCustTlv`, and `GetCustTlvData`. The following descriptions provide insight into the functionality of each component.

- **ReadCustomData** (Figure 4) is a function designed to target a specific EFI variable that stores customization data. It first tries to get the variable's data and size with the `CommonGetVariableDataAndSize()` function. If it cannot find it, the function returns NULL right away. After successfully fetching the variable, the function proceeds to check the integrity of the retrieved data. It utilizes the `CUST_DATA_HEADER` structure to find a 'magic' identifier value (0x41424344). If this value is not found, it suggests that the data is not in the expected format, leading to the disposal of the data and an exit with a NULL return value. if the variable data passes all integrity checks, the function returns the data through the `var_data` parameter.

```
static UINT8 *ReadCustomData(UINTN *data_size)
{
  EFI_STATUS Status;
  UINT8 *var_data;
  UINTN var_size;

  Status = CommonGetVariableDataAndSize(CUST_DATA_VAR_NAME,
                                        &gEfiAavaEfiVarGuid,
                                        &var_size,
                                        (VOID **)&var_data);

  if (Status != EFI_SUCCESS)
  {
    return NULL;
  }

  struct CUST_DATA_HEADER *header_pointer = (struct CUST_DATA_HEADER *)var_data;

  if (header_pointer->magic != 0x41424344)
  {
    FreePool(var_data);
    return NULL;
  }

  *data_size = var_size;
  return var_data;
}
```

FIGURE 4. ReadCustomData function

- **FindCustTlv** (Figure 5) function scans through the data obtained from the EFI variable to find a specific Type-Length-Value (TLV) entry based on its type. It starts by skipping the initial `CUST_DATA_HEADER` and then iterates through each TLV entry in the data. The function uses the `TLV_HEADER` structure to check if the current TLV entry's type matches the desired `tlv_type`. If a match is found, the function returns 0 and sets the `tlv_pointer` to point to the matched TLV header. Otherwise, it returns 1, indicating that the desired TLV type was not found in the data.

```
int FindCustTlv(UINT8 *cust_data, UINTN cust_data_length, UINT16 tlv_type, TLV_HEADER **tlv)
{
  UINT8 *data_pointer;
  TLV_HEADER *tlv_pointer;

  data_pointer = (UINT8 *)(cust_data + sizeof(struct CUST_DATA_HEADER));

  while (data_pointer < (cust_data + cust_data_length))
  {
    tlv_pointer = (TLV_HEADER *)data_pointer;

    if (tlv_pointer->type == tlv_type)
    {
      *tlv = tlv_pointer;
      return 0;
    }

    data_pointer += sizeof(TLV_HEADER) + tlv_pointer->length;
  }

  return 1;
}
```

FIGURE 5. FindCustTlv function

- **GetCustTlvData** (Figure 6) function is designed to get the actual data of a specific Type-Length-Value (TLV) entry. First, the function either retrieves the EFI variable data using `ReadCustomData` or uses pre-loaded data available in the `cust_info` parameter. The `cust_info` parameter was added later as an optimization to minimize calls to `ReadCustomData` when multiple customizations are required within the same function. Then it calls `FindCustTlv` to search for the TLV entry that matches the specified `tlv_type`. If the desired TLV type is found, the function allocates memory to store the actual data of the TLV entry and copies it there. It then updates the `tlv_data_length` to reflect the length of the copied data.

```
int GetCustTlvData(UINT8 **tlv_data, UINT16 *tlv_data_length, UINT16 tlv_type, CustDataInfo *cust_info)
{
  UINTN cust_data_length;
  UINT8 *cust_data;
  TLV_HEADER *tlv;

  if (cust_info == NULL)
  {
    cust_data = ReadCustomData(&cust_data_length);

    if (cust_data == NULL)
    {
      return 1;
    }
  }
  else
  {
    cust_data = cust_info->data;
    cust_data_length = cust_info->length;
  }

  int status = FindCustTlv(cust_data, cust_data_length, tlv_type, &tlv);

  if (status != 0)
  {
    if (cust_info == NULL)
    {
      FreePool(cust_data);
    }
    return 1;
  }

  *tlv_data = (UINT8 *)AllocatePool(tlv->length);

  if (*tlv_data == NULL)
  {
    if (cust_info == NULL)
    {
      FreePool(cust_data);
    }
    return 1;
  }

  CopyMem(*tlv_data, ((UINT8 *)tlv) + sizeof(TLV_HEADER), tlv->length);

  *tlv_data_length = tlv->length;

  if (cust_info == NULL)
  {
    FreePool(cust_data);
  }

  return 0;
}
```

FIGURE 6. GetCustTlvData function

3.1.3 Example Customization Function

`AavaCustSetupData` (Figure 7) acts as a showcase for using the utility functions like `ReadCustomData`, `FindCustTlv`, and `GetCustTlvData` to implement specific BIOS customizations, in this case, for disabling the camera.

The function starts by checking if the `SetupConfig` pointer is NULL. If it is, the function exits to avoid null pointer issues. Next, it reads customization data using `ReadCustomData` and stores it in a `CustDataInfo` structure.

The use of the `CustDataInfo` structure allows the function to read customization data more efficiently. This is particularly useful when the function needs to handle multiple customizations, as it minimizes the number of calls to `ReadCustomData`.

The function then calls `GetCustTlvData` to look for a specific TLV type (`CUST_TLV_TYPE_CAMERA_DISABLE`). If this type is found and its data is 1, the function sets the `MipiCam_Link0` and `MipiCam_Link1` variables in the `SetupConfig` structure to zero.

The `MipiCam_Link0` and `MipiCam_Link1` represent specific BIOS setup items, which are user-configurable options available within the BIOS menu. These settings control the initialization and availability of the camera interfaces on the device.

With this customization, the function programmatically sets both `MipiCam_Link0` and `MipiCam_Link1` to zero within the `SetupConfig` structure. This action effectively disables the camera by overriding the default or user-configured settings in the BIOS, enforcing the custom setting to always take precedence.

This function is designed to be called before retrieving the `SetupData` variable. The hook is placed at a specific point before the `MipiCam_Link0` and `MipiCam_Link1` settings are used. Calls are placed in both the DXE and PEI phases of the boot phase to ensure that the customization takes effect at the right time.

```
VOID AavaCustSetupData(SETUP_DATA *SetupConfig)
{
  if (SetupConfig == NULL)
  {
    return;
  }

  CustDataInfo cust_info;
  cust_info.data = ReadCustomData(&cust_info.length);

  if (cust_info.data == NULL)
  {
    return;
  }

  //Camera disable
  UINT8 *tlv_data = NULL;
  UINT16 tlv_data_length;
  int status = GetCustTlvData(&tlv_data, &tlv_data_length, CUST_TLV_TYPE_CAMERA_DISABLE, &cust_info);

  if (status == 0 && *tlv_data == 1)
  {
    SetupConfig->MipiCam_Link0 = 0;
    SetupConfig->MipiCam_Link1 = 0;
  }

  if (tlv_data)
  {
    FreePool(tlv_data);
  }

  FreePool(cust_info.data);
}
```

FIGURE 7. AavaCustSetupData function

3.1.4 Decision Process for Function Hook Placement

Initially, consideration was given to the approach of inserting calls to the customization functions directly before the usage of `MipiCam_Link0` and `MipiCam_Link1` settings in each instance. However, this approach soon proved to be tedious and challenging to manage.

The idea of embedding these calls at the very root of the variable initialization process was then explored. This would have involved invoking the customization function within the functions responsible for fetching the variables in the first place. Upon closer examination, it was discovered that this could lead to unintended consequences.

Implementing the changes at the variable fetching root could result in custom settings persisting unintentionally. If a customization is made so early in the process, any subsequent removal of the customization from the TLV data might not revert the settings, since the initial custom value would have already propagated through the system.

The final solution sought a balance between these two extremes. The function calls were placed at the first points where the `GetVariable` functions are invoked (Figure 8), targeting the PEI and DXE phases of the boot process. This ensures the customization is both effective and easier to manage. In some specific scenarios, the function was hooked before a setup item underwent any other changes. This placement ensures that the customizations are set and unset correctly, preventing them from sticking when they should no longer apply.

```
  VariableSize = sizeof (SETUP_DATA);
  Status = VariableServices->GetVariable (
                              VariableServices,
                              L"Setup",
                              &gSetupVariableGuid,
                              NULL,
                              &VariableSize,
                              &SetupVariables
                              );
  ASSERT_EFI_ERROR (Status);

  AavaCustSetupData(&SetupVariables);

  VariableSize = sizeof (SA_SETUP);
  Status = VariableServices->GetVariable (
                              VariableServices,
                              L"SaSetup",
                              &gSaSetupVariableGuid,
                              NULL,
                              &VariableSize,
                              &SaSetup
                              );
  ASSERT_EFI_ERROR (Status);
```

FIGURE 8. Example call in PeiPchPolicyUpdate.c

3.1.5 Password Management Customizations

One of the limitations of traditional BIOS/UEFI firmware is the restricted password management capabilities. Specifically, the native BIOS menu only allows a single password for both boot and settings, and these must be set directly within the BIOS interface. BIOS/UEFI password management system was expanded to remove these limitations. Here are the key changes:

Enhanced Flexibility

- **Separate Passwords for Boot and Settings**: Users can set different passwords for booting and accessing the BIOS settings. This feature is particularly useful for organizations or individual users who want to add an extra layer of security by having distinct passwords for different functions.
- **OS-Level Control**: Users no longer have to navigate to the BIOS menu to set or change these passwords. The passwords are now manageable through our developed Windows application, adding a layer of convenience to the process.

Bios Settings Password

- **Extended Coverage**: The settings password now applies to all submenus within the BIOS.

Password Storage and Verification

- **EFI-Based Password Storage**: We transitioned from storing the passwords in the CMOS (Complementary Metal-Oxide-Semiconductor) to storing them as EFI variables. This change safeguards the passwords from being reset if the RTC (Real-Time Clock) backup battery is drained.
- **EFI Variable for Password Status**: An additional feature was added to store the status of these passwords in an EFI variable. This variable is especially useful during production testing to verify whether passwords have been set on the device.

3.2 Windows Application Development

The application is a console-based utility developed in Visual Studio. Its primary role is to interact with EFI variables, specifically for reading and writing data in the TLV (Type-Length-Value) format. It leverages native Windows API functions like `GetFirmwareEnvironmentVariableA` and `SetFirmwareEnvironmentVariableA` to perform these tasks.

3.2.1 Architecture and Key Components

The core of this console application is a library containing the main functionalities. The architecture is as follows:

- **Data Functions**:
  - **Static Functions**: `GetEfiVar` and `SetEfiVar` are static functions that are intended to be used only within the library.
  - **Public Functions**: `GetCustData` and `SetCustData` are designed to interact with the static EFI variable functions and are accessible from the console application.
- **TLV Functions**:
  - The library incorporates functions for manipulating TLV data: `FindCustTlv`, `GetCustTlvData`, `RemoveCustTlv`, and `SetCustTlv`. These functions utilize `GetCustData` and `SetCustData` for their operations.

3.2.2 Headers and Information Sharing

Two header files are integral to the application:

- **AavaCustDataFormat.h**: This header is confined to the library to keep certain details internal and unexposed to the application.
- **AavaBiosCustLib.h**: This header file is used for sharing function prototypes and other miscellaneous information with the console application.

The modular architecture allows for efficient and secure operations on EFI variables, making the console application both easy to maintain and ready for future extensions.

3.2.3 Privilege Management

A critical aspect of working with EFI variables is privilege management. The library incorporates two functions, `GetPrivileges` and `RestorePrivileges`, to manage this.

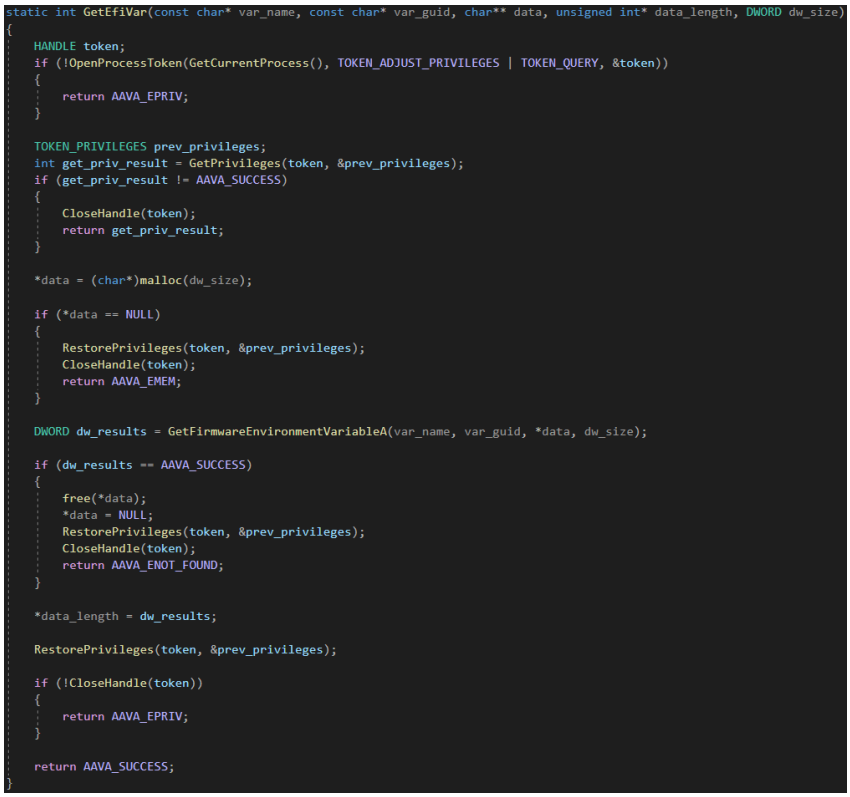
- **GetPrivileges**: This function is designed to temporarily elevate the program's access level to allow for modifications of EFI variables. It uses native Windows API functions to look up the required system privilege and enable it for the running process. If the function encounters an error during this process, it returns a predefined error code to signal that privilege escalation has failed.
- **RestorePrivileges**: After operations that require elevated privileges are completed, this function is invoked to bring the program's access level back down to its original state.

By encapsulating the privilege elevation and restoration logic within these two functions, the library ensures a controlled environment for operations that require elevated access.

3.2.4 Fetching and Setting Data

The core operations of interacting with EFI variables are done through two static functions in the library: `GetEfiVar` and `SetEfiVar`. These functions utilize Windows native API calls for their operations. Below there are some insights into these functions.

GetEfiVar Function



```
static int GetEfiVar(const char* var_name, const char* var_guid, char** data, unsigned int* data_length, DWORD dw_size)
{
    HANDLE token;
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &token))
    {
        return AAVA_EPRIV;
    }

    TOKEN_PRIVILEGES prev_privileges;
    int get_priv_result = GetPrivileges(token, &prev_privileges);
    if (get_priv_result != AAVA_SUCCESS)
    {
        CloseHandle(token);
        return get_priv_result;
    }

    *data = (char*)malloc(dw_size);

    if (*data == NULL)
    {
        RestorePrivileges(token, &prev_privileges);
        CloseHandle(token);
        return AAVA_EMEM;
    }

    DWORD dw_results = GetFirmwareEnvironmentVariableA(var_name, var_guid, *data, dw_size);

    if (dw_results == AAVA_SUCCESS)
    {
        free(*data);
        *data = NULL;
        RestorePrivileges(token, &prev_privileges);
        CloseHandle(token);
        return AAVA_ENOT_FOUND;
    }

    *data_length = dw_results;

    RestorePrivileges(token, &prev_privileges);

    if (!CloseHandle(token))
    {
        return AAVA_EPRIV;
    }

    return AAVA_SUCCESS;
}
```

FIGURE 9. GetEfiVar function

The `GetEfiVar` function (Figure 9) is designed to retrieve an EFI variable using the `GetFirmwareEnvironmentVariableA` Windows API function.

- **Privilege Escalation**: The function first gains the necessary privileges by calling `OpenProcessToken` and `GetPrivileges`.
- **Memory Allocation**: It allocates memory for storing the variable's data.
- **Data Retrieval**: It calls the native Windows function `GetFirmwareEnvironmentVariableA` to fetch the data.
- **Return and Cleanup**: The function restores any changed privileges and returns appropriate error codes or success indicators.

SetEfiVar Function

```
static int SetEfiVar(const char* var_name, const char* var_guid, char* data, unsigned int data_length)
{
    HANDLE token;
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &token))
    {
        return AAVA_EPRIV;
    }

    TOKEN_PRIVILEGES prev_privileges;
    int get_priv_result = GetPrivileges(token, &prev_privileges);
    if (get_priv_result != AAVA_SUCCESS)
    {
        CloseHandle(token);
        return get_priv_result;
    }

    if (var_name == NULL || var_guid == NULL || data == NULL || data_length == 0)
    {
        RestorePrivileges(token, &prev_privileges);

        if (!CloseHandle(token))
        {
            return AAVA_EPRIV;
        }

        return AAVA_EPARAM;
    }

    if (data_length > MAX_CUST_DATA_SIZE)
    {
        RestorePrivileges(token, &prev_privileges);

        if (!CloseHandle(token))
        {
            return AAVA_EPRIV;
        }

        return AAVA_ESIZE;
    }

    SetFirmwareEnvironmentVariableA(var_name, var_guid, data, data_length);

    RestorePrivileges(token, &prev_privileges);

    if (!CloseHandle(token))
    {
        return AAVA_EPRIV;
    }

    return AAVA_SUCCESS;
}
```

FIGURE 10. SetEfiVar function

The `SetEfiVar` function (Figure 10) sets the value of an EFI variable by utilizing `SetFirmwareEnvironmentVariableA`.

- **Privilege Escalation**: Like `GetEfiVar`, it also elevates privileges at the beginning.
- **Parameter Checks**: It validates input parameters for nullity and appropriate sizes.
- **Data Setting**: Utilizes the Windows native function `SetFirmwareEnvironmentVariableA` to set the data.
- **Cleanup**: Restores the previous state of privileges and closes handles.

The library makes use of custom return values, such as `AAVA_EPRIV` for privilege errors, `AAVA_EMEM` for memory errors, and `AAVA_SUCCESS` for successful operation. These custom return values provide a more comprehensive understanding of what exactly went wrong or right during the operation. This is especially useful for debugging and logging purposes.

3.2.5 Custom Data Management Functions

Dealing with custom data in EFI variables means a higher-level abstraction over the basic `GetEfiVar` and `SetEfiVar` methods is needed. The library handles this through two functions: `GetCustData` and `SetCustData`.

- **GetCustData**: This function (Figure 11) calls the underlying `GetEfiVar` method to fetch EFI variables but adds an extra layer of validation and structure. The function checks for the presence of a specific magic value and version information in the header of the fetched data. If these checks pass, the function returns the data as valid, otherwise it prepares a new data header with default values. The purpose is to ensure data integrity and version compatibility.

```
int GetCustData(char** cust_data, unsigned int* cust_data_length)
{
    int n_result = GetEfiVar(AAVA_CUST_EFI_VAR_NAME, AAVA_CUST_EFI_VAR_GUID, cust_data, cust_data_length, MAX_CUST_DATA_SIZE);

    if (n_result == AAVA_SUCCESS)
    {
        CUST_DATA_HEADER* p_header = (CUST_DATA_HEADER*)*cust_data;

        if (p_header->magic == AAVA_CUST_DATA_MAGIC)
        {
            if (p_header->major_version <= AAVA_CUST_DATA_MAJOR_VERSION)
            {
                return AAVA_SUCCESS;
            }
            else
            {
                return AAVA_EVERSION;
            }
        }

        free(*cust_data);
    }

    *cust_data_length = sizeof(CUST_DATA_HEADER);
    *cust_data = (char*)malloc(*cust_data_length);

    if (*cust_data == NULL)
    {
        return AAVA_EMEM;
    }

    CUST_DATA_HEADER* p_header = (CUST_DATA_HEADER*)*cust_data;
    p_header->magic = AAVA_CUST_DATA_MAGIC;
    p_header->major_version = AAVA_CUST_DATA_MAJOR_VERSION;
    p_header->minor_version = AAVA_CUST_DATA_MINOR_VERSION;
    return AAVA_SUCCESS;
}
```

FIGURE 11. GetCustData function

- **SetCustData**: This function (Figure 12) is much simpler and serves as a wrapper around `SetEfiVar`. It directly calls `SetEfiVar` with the custom data and length provided. Its primary role is to streamline the interface for setting custom EFI variables, making it easier for the main application to manage such data.

```
int SetCustData(char* cust_data, unsigned int cust_data_length)
{
    return SetEfiVar(AAVA_CUST_EFI_VAR_NAME, AAVA_CUST_EFI_VAR_GUID, cust_data, cust_data_length);
}
```

FIGURE 12. SetCustData function

These two functions work in tandem to simplify the application's interactions with EFI variables. They manage the data headers, perform validity checks, and handle versioning.

3.2.6 Type-Length-Value (TLV) Management Functions

While the application library has functions like `GetCustTlvData` and `FindCustTlv` that operate similarly to their BIOS counterparts, it introduces two application-specific functions, `SetCustTlv` and `RemoveCustTlv`, for additional TLV management. These new functions enable adding or removing TLV entries directly from the application level, whereas in the BIOS code, such manipulations aren't typically performed.

- **SetCustTlv**: This function (Figure 13) allows for the insertion of a new TLV record into the custom EFI data. It starts by calling `RemoveCustTlv` to ensure that any existing TLV entry with the same type is deleted. Then it fetches the existing custom data using `GetCustData`. A new buffer is allocated to accommodate the new TLV entry. The function finally uses `SetCustData` to write this new custom data back to the EFI variable.

```
int SetCustTlv(unsigned short tlv_type, unsigned short tlv_value_length, char* tlv_value)
{
    int result = RemoveCustTlv(tlv_type);

    if (result != 0)
    {
        return AAVA_EREM_FAIL;
    }

    char* cust_data;
    unsigned int cust_data_length;
    result = GetCustData(&cust_data, &cust_data_length);

    if (result != AAVA_SUCCESS)
    {
        if (cust_data != NULL)
        {
            free(cust_data);
        }
        return result;
    }

    unsigned int new_cust_data_length = cust_data_length + sizeof(TLV_HEADER) + tlv_value_length;
    char* new_cust_data = (char*)malloc(new_cust_data_length);

    if (new_cust_data == NULL)
    {
        free(cust_data);
        return AAVA_EMEM;
    }

    memcpy(new_cust_data, cust_data, cust_data_length);

    TLV_HEADER new_tlv_header;

    new_tlv_header.type = tlv_type;
    new_tlv_header.length = tlv_value_length;

    char* target_pointer;

    target_pointer = (new_cust_data + cust_data_length);
    memcpy(target_pointer, &new_tlv_header, sizeof(TLV_HEADER));

    target_pointer += sizeof(TLV_HEADER);
    memcpy(target_pointer, tlv_value, tlv_value_length);

    free(cust_data);
    SetCustData(new_cust_data, new_cust_data_length);
    free(new_cust_data);

    return AAVA_SUCCESS;
}
```

FIGURE 13. SetCustTlv function

- **RemoveCustTlv**: This function (Figure 14) aims to remove a specific TLV record from the custom EFI data. It starts by fetching the current custom data using `GetCustData`. If the TLV entry targeted for removal is found using `FindCustTlv`, a new custom data buffer is generated without that TLV entry. This updated data is then written back to EFI storage using `SetCustData`.

```
int RemoveCustTlv(unsigned short tlv_type)
{
    char* cust_data;
    unsigned int cust_data_length;
    int result = GetCustData(&cust_data, &cust_data_length);

    if (result != AAVA_SUCCESS)
    {
        if (cust_data != NULL)
        {
            free(cust_data);
        }
        return result;
    }

    TLV_HEADER* tlv_pointer;
    result = FindCustTlv(cust_data, cust_data_length, tlv_type, &tlv_pointer);

    if (result == AAVA_ENOT_FOUND)
    {
        free(cust_data);
        return AAVA_SUCCESS;
    }
    else if (result != AAVA_SUCCESS)
    {
        free(cust_data);
        return result;
    }

    unsigned int new_cust_data_length = cust_data_length - sizeof(TLV_HEADER) - tlv_pointer->length;
    char* new_cust_data = (char*)malloc(new_cust_data_length);

    if (new_cust_data == NULL)
    {
        free(cust_data);
        return AAVA_EMEM;
    }

    memcpy(new_cust_data, cust_data, sizeof(CUST_DATA_HEADER));

    char* target_pointer;
    char* source_pointer;

    source_pointer = cust_data + sizeof(CUST_DATA_HEADER);
    target_pointer = new_cust_data + sizeof(CUST_DATA_HEADER);

    while (source_pointer < (cust_data + cust_data_length))
    {
        tlv_pointer = (TLV_HEADER*)source_pointer;

        if (tlv_pointer->type != tlv_type)
        {
            memcpy(target_pointer, source_pointer, (sizeof(TLV_HEADER) + tlv_pointer->length));
            target_pointer += (sizeof(TLV_HEADER) + tlv_pointer->length);
        }
        source_pointer += (sizeof(TLV_HEADER) + tlv_pointer->length);
    }

    free(cust_data);
    SetCustData(new_cust_data, new_cust_data_length);
    free(new_cust_data);

    return AAVA_SUCCESS;
}
```

FIGURE 14. RemoveCustTlv function

3.2.7 Application Architecture

The application is architecturally designed to offer multiple interaction modes for users: a command-line mode and an interactive mode.

Command-line Mode

In this mode, the user can execute the application with specific commands and their corresponding arguments directly from the command line. This mode is particularly useful for batch processing or automation tasks where interactive input is not necessary.

For example, if a user wants to set a TLV value, they could run the application with a command like appname settlv [arguments], and the application would perform the task and exit. All commands are processed through a command processing function, which employs a mapping structure to route the command to its specific handler function. Each command like "settlv", "getdata", etc., has a corresponding function in the code, and command process function ensures that the right function is called for the right command.

Interactive Mode

In this mode, the user enters a shell-like interactive session where they can execute multiple commands in sequence. It welcomes the user and enters a loop where it continuously prompts the user for commands to execute. These commands are again processed through a command processing function, the same as in command-line mode. This setup ensures consistency in command processing, whether in interactive or command-line mode.

Interactive mode makes the application more user-friendly by allowing users to execute multiple commands without needing to restart the application. Users can work within this mode, making various system adjustments and running multiple commands in a single session. Users can also use command getdata to easily view the current data stored in the EFI variable in hexadecimal format.

Command Parsing and Processing

When a command is entered, either as a command-line argument or during an interactive session, it is broken down into its main command and any additional arguments. The command process function then takes over, mapping the command to its corresponding function using a pre-defined mapping structure.

Data Persistence Functions

In the application architecture, two functions exist for data persistence that permit users to save and reload system customizations.

One of these functions takes the current customization data and saves it to a specified binary file. It first fetches the current customization settings and writes these to the file, enabling users to create a snapshot of their system's configuration for backup or migration purposes.

The other function works in the reverse manner. It reads a specified binary file containing saved customization settings and applies them to the system. This is particularly useful for propagating a standard set of settings across multiple systems or restoring a system to a previous customization state.

Both of these functions are fully integrated into the command processing architecture of the application. They can be invoked either via the command-line mode for batch processing or through the interactive mode for individual customization sessions.

Example Command Function

The `CommandSettlv` function shown in Figure 15 below, serves to set or update TLV data in the customization data. It is integrated as a handler function within the application's architecture and can be invoked in both command-line and interactive modes. This integration is made possible by a command processing function that routes the "settlv" command to `CommandSettlv`.

- **Argument Validation**:
    - The function checks if a minimum of three arguments are provided (type and at least one data value). If not, it displays a usage message and exits with an error code.
- **Memory Allocation and Conversion**:
    - Memory is allocated for TLV values based on the argument count (`arg_count` - 2 to exclude the function name and type argument). The arguments beyond the first (`arg_values`[i + 2]) are then converted into TLV values using utility functions with error checking for each conversion.
- **TLV Data Management**:
    - The function fetches existing TLV data, if any, using `GetCustTlvData`. If existing data is found, it is removed by calling `RemoveCustTlv`. It then sets the new TLV data using `SetCustTlv`.
- **User Feedback and Cleanup**:
    - The function provides a success or error message and frees all dynamically allocated memory before exiting.

```c
int CommandSettlv(int arg_count, char* arg_values[])
{
    // Check if the required number of arguments is provided
    if (arg_count < 3)
    {
        printf("Usage: settlv [type] [data_1] [data_2] ...\n");
        return 1;
    }

    unsigned short tlv_value_length = arg_count - 2;
    char* tlv_values = (char*)malloc(tlv_value_length * sizeof(char));

    if (tlv_values == NULL)
    {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Convert the argument values to the TLV values
    for (int i = 0; i < tlv_value_length; i++)
    {
        // Use the error-checking wrapper function to convert the argument to a char
        if (UtilStrtolChar(arg_values[i + 2], 0, &tlv_values[i]) != 0)
        {
            printf("Invalid parameter value %s\n", arg_values[i + 2]);
            // Free memory and exit if conversion failed
            free(tlv_values);
            return 1;
        }
    }

    unsigned short tlv_type;
    // Use the error-checking wrapper function to convert the argument to an unsigned short
    if (UtilStrtolUShort(arg_values[1], 0, &tlv_type) != 0)
    {
        printf("Invalid parameter value %s\n", arg_values[1]);
        // Free memory and exit if conversion failed
        free(tlv_values);
        return 1;
    }

    char* tlv_data = NULL;
    unsigned short tlv_data_length;

    int result = GetCustTlvData(&tlv_data, &tlv_data_length, tlv_type);

    if (result == 0)
    {
        RemoveCustTlv(tlv_type);
    }

    result = SetCustTlv(tlv_type, tlv_value_length, tlv_values);

    if (result == 0)
    {
        if (tlv_data != NULL)
        {
            printf("TLV %u updated.\n", tlv_type);
        }
        else
        {
            printf("TLV %u created.\n", tlv_type);
        }
    }
    else
    {
        printf("Error updating/creating TLV %u.\n", tlv_type);
        free(tlv_values);
        return 1;
    }

    if (tlv_data != NULL)
    {
        free(tlv_data);
    }

    free(tlv_values);

    return 0;
}
```

FIGURE 15. Example command function

3.3 Using Customizations

Customization Packages

The standard approach for enabling customizations has been through customization packages. These packages are generally comprised of batch scripts along with essential tools that permit customers to implement their specific customizations. Typically, a customer's customizations may include modifying the boot logo to represent their brand, implementing specific SMBIOS changes, and disabling certain device functionalities. These batch scripts make use of the application's command-line features, thus allowing customers to roll out these modifications efficiently across multiple systems in a uniform manner.

Future Implementation: Dashboard Interface

For upcoming projects, the introduction of a dashboard interface is under consideration. This graphical user interface (GUI) aims to offer an accessible platform for conducting select customizations. Rather than generating a new customization package for each modification, the dashboard would allow customers to make these adjustments directly. This advancement would simplify the customization workflow and offer real-time management of system settings.

The objective is to provide end users with a versatile set of tools for system customization, ranging from batch-scripted packages for widespread changes to an interactive dashboard for more specific adjustments.

4 CONCLUSION

This thesis set out to make BIOS/UEFI customization more user-friendly by enabling changes directly from the operating system. The focus was on modifying the BIOS code to read EFI variables and creating a Windows application to handle these settings. To manage the data effectively, Type-Length-Value (TLV) data format was used.

New code was integrated into the existing BIOS, primarily written in C, which reads EFI variables set by the operating system. The Windows application developed allows users to adjust these EFI variables. While a system reboot is still required to implement these changes, the new method enables users to configure their settings in the OS before doing so, adding a layer of convenience to the process.

Diving into this project offered me a wealth of learning opportunities. I deepened my understanding of BIOS/UEFI mechanics, the nuances of computer boot-up processes, and honed my skills in C programming.

One of the biggest challenges was initially navigating the complex codebase of the BIOS. Determining where to insert new code and making architectural decisions about how certain features should be implemented was a steep learning curve. While the Windows application was generally easier to develop compared to the BIOS side, it presented its own set of challenges. Understanding how to effectively use the TLV data format was particularly difficult at first, as was comprehending how data should be managed and parsed. Over time, these challenges were overcome, significantly enhancing my expertise in the subject matter.

Looking forward, the next stage for this project involves developing tailored customization packages to meet the diverse needs of different customers. Additionally, we are receptive to incorporating new customization features based on specific customer requests.

5 REFERENCES

1. Krau, Michael & Wei, Dong. (2014). Clarifying the Ten Most Common Misconceptions about UEFI. Search date 4.10.2023. https://uefi.org/sites/default/files/resources/UEFI_Clarifying_Common_Misconceptions_White_Paper_April%202014_Final.pdf

2. Wilkins et al., (2023). Decoding UEFI Firmware. Search date 4.10.2023. https://uefi.org/sites/default/files/resources/What%20is%20UEFI-Aug31-2023-Final.pdf.

3. Davy Souza. (2021). Sending data from UEFI to OS through UEFI variables. Search date 15.10.2023. https://davysouza.medium.com/sending-data-from-uefi-to-os-through-uefi-variables-b4f9964e1883

4. Unified Extensible Firmware Interface (UEFI) Specification. (2020). Search date 15.10.2023. https://uefi.org/sites/default/files/resources/UEFI%20Spec%202.8B%20May%202020.pdf

5. parrotGPT & arvindpdmn. (2023). TLV Format. Search date 15.10.2023. https://devopedia.org/tlv-format

6. Brian Richardson. (2017). PI Boot Flow. Search date 16.10.2023. https://github.com/tianocore/tianocore.github.io/wiki/PI-Boot-Flow