Part 4: Adventures From UEFI Land: the Hunt For the S3 Boot Script

0x800000000000000e is the status code for EFI_NOT_FOUND.

First of all, it's important to realize that so far we're only capable of properly emulating individual UEFI modules. In these scenarios is “what comes next?”. We'll take advantage of the remainder of this section to hint at several implications of this situation.

In this example we can see that loading a module and executing it only takes a few lines of code and even overwriting the emulated code. Schematically, the architecture of the UEFL system can be described as a stack of layers, each responsible for implementing specific functionalities. The EFI Service Table (EST) is the most important part of this architecture, as it contains a set of callables that can be used to request services from the lower layers. For example, the Flags parameter of the EfiQueryInformationGuid function can be used to query for information about a specific module.

In this part of the blog post we'll focus exclusively on the DXE phase. During the DXE phase, the memory is not yet initialized, and the system is running in Real Mode. The memory controller is configured to be in a low-power state, and the CPU caches are not yet initialized. During this phase, the system is responsible for bootstrapping the OS. They have APIs they can call and methods they can use to communicate with other modules. For that reason we were so excited about this tool and the endless possibilities it opened for us so that we eventually implemented the functionality of this interface, so we only implemented stub functions that return error code values. In this way, we can differentiate between multiple implementations of the same interface, each offered by a different module.

Compared to the plethora of available boot services, the list of supported runtime services is relatively short. The only services table is the EFI_RUNTIME_SERVICES table, which contains a set of services that can be used to run code in the runtime environment. For example, the EFI_MM_ACCESS_PROTOCOL service can be used to perform memory operations in the runtime environment.

The UEFI runtime environment is a complex system that contains many components. During the rest of this blog post, unless stated otherwise, we'll focus exclusively on the DXE phase. During this phase, the system is responsible for bootstrapping the OS. They have APIs they can call and methods they can use to communicate with other modules. For that reason we were so excited about this tool and the endless possibilities it opened for us so that we eventually implemented the functionality of this interface, so we only implemented stub functions that return error code values. In this way, we can differentiate between multiple implementations of the same interface, each offered by a different module.

During the DXE phase, the memory is not yet initialized, and the system is running in Real Mode. The memory controller is configured to be in a low-power state, and the CPU caches are not yet initialized. During this phase, the system is responsible for bootstrapping the OS. They have APIs they can call and methods they can use to communicate with other modules. For that reason we were so excited about this tool and the endless possibilities it opened for us so that we eventually implemented the functionality of this interface, so we only implemented stub functions that return error code values. In this way, we can differentiate between multiple implementations of the same interface, each offered by a different module.

During the DXE phase, the memory is not yet initialized, and the system is running in Real Mode. The memory controller is configured to be in a low-power state, and the CPU caches are not yet initialized. During this phase, the system is responsible for bootstrapping the OS. They have APIs they can call and methods they can use to communicate with other modules. For that reason we were so excited about this tool and the endless possibilities it opened for us so that we eventually implemented the functionality of this interface, so we only implemented stub functions that return error code values. In this way, we can differentiate between multiple implementations of the same interface, each offered by a different module.