WHITE PAPER - MYTH AND CONSIDERATION TO MAKE THE SWITCH TO UEFI

TABLE OF CONTENTS

INTRODUCTION

The Unified Extensible Firmware Interface (UEFI) is a standardized firmware specification that simplifies and secures platform initialization and firmware bootstrap operations.

UEFI is currently developed and supported by representatives from more than 250 industry-leading technology companies.

The newest revision of the Unified Extensible Firmware Interface (UEFI) Specification now includes the support for ARM architecture.

ARM and the Linaro Enterprise Group is also promoting the use of UEFI on ARM architecture, because in contrast to some firmware without standard specification, UEFI specification help standardize boot process for ARM processor-based platforms.

This paper outlines and clarifies the points about UEFI technology implemented on the Altera Arria 10 SoCFPGA to help you decide whether to make the switch from other firmware technology to UEFI technology based bootloader firmware.

BENEFITS OF UEFI TECHNOLOGY

UEFI technology is future proof through standardization of firmware design rather than proprietary firmware design. UEFI specifications promote business and technological efficiency, improve performance and security, facilitate interoperability between devices, platforms and systems and comply with next-generation technologies.

UEFI specification is peer reviewed and published, this allow firmware code to write once per platform and can be re-use easily without much modification. This translates into cost saving and time saving in boot loader development.

There are many tested UEFI drivers available from 3rd party peripherals providers which can be from one of the 250 industry-leading technology companies who already adopted UEFI technology. There are also many tested ARM architectural support drivers in the UEFI source code folder for both ARMv7 and ARMv8 architectures. By reusing codes we do not have to worry about the correctness of code by reinventing the wheel.

The SoCFPGA implementation of UEFI is based on UEFI Development Kit (UDK2015) publish by tianocore.org UDK2015 (a.k.a. EDK II) is a modern, feature-rich, cross-platform firmware development environment for the UEFI specifications. This framework uses the BSD license which has the benefit of permitting anyone to retain the option of commercializing final results with minimal legal issues.

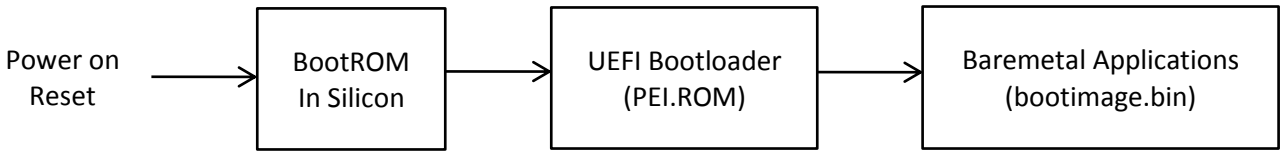
MYTH AND CONSIDERATION TO MAKE THE SWITCH TO UEFI

When one is considering whether to make the switch to UEFI technology from other none UEFI based firmware solution, questions often arises in one's mind could be UEFI based firmware may be too large to fit or it could be slow to boot and many other misunderstandings about UEFI.

SIZE

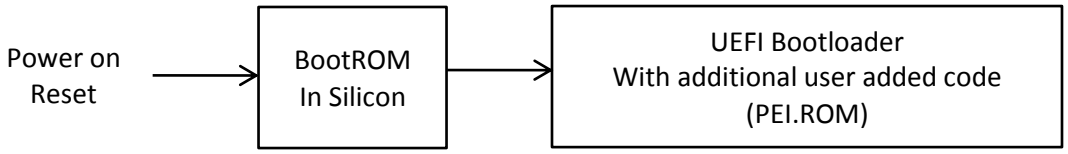
For user of Arria 10 SoCFPGA, the hard processor system (HPS) is built-in with 256 KB of on-chip memory. For design that just wants to utilize the on-chip memory without a DDR SDRAM, the primary concern will be size and memory foot prints.

Let's say you just want to have some simple code not more than 256 KB running on the on-chip memory for this design. When you heard about the UEFI Bootloader, you might be worry that the UEFI binary size could be big and by switching to use the UEFI Bootloader it will take up all the available space in the on-chip memory. Actually this not something to be worried about, because your boot flow will be:



As you can see the UEFI Bootloader will load on the on-chip RAM first, your application will be compiled as a hand-off boot image. After UEFI initialized the HPS, it will jump to your application. So you can use all the available 256 KB of on-chip memory for your application.

A second scenario could be that you just want to have some very simple code that take not more than 32 KB and you are considering adding it as part of the UEFI Bootloader, you want your code to be executed after the UEFI Bootloader finished initializing the HPS.



In this situation, the UEFI Bootloader already has callback functions in place which you can add your custom code there in the file named *Board.c* with function name *BoardSpecificInitialization*. Size wise there are plenty of space, in Debug build with all the HPS debug message there is around 60 KB available free space and if you turn off the Debug message, you will get around 100 KB of free space for your custom code all running within the on-chip memory as part of UEFI Bootloader. Note that the size mentioned here is rough estimation and not absolute as it may change as UEFI code get updated with patch, but the idea of adding your own code into the UEFI bootloader is supported and very easy to do so.

BOOT TIME

For user of Arria 10 SoCFPGA, whether you boot with UEFI based bootloader or other type of bootloader, the choice of firmware technology actually has insignificant impact on the boot time. Most of the time how fast the bootloader can boot is actually bottle necked by the hardware choice and configuration, because the HPS processor is typically running at more than 1 GHz, it spend more time polling for hardware readiness before it can execute the next line of code than executing the code itself.

The table below is a measurement of UEFI boot time taken on Arria 10 SoC Dev Kit Rev A board using the first release of UEFI for SoC EDS version 15.0.1:

Table 1: UEFI Boot Time

	SDMMC (us)	QSPI (us)
Clock Configuration	57224	57264
IO Configuration	8168	8171
Flash Initialization	703388	454
FPGA Full Configuration	3686805	1186426
Hard Memory Configuration	111082	111265
Firewall Configuration	7495	7494
Others	20421	20369
Total Pei Stage Boot Time	**4594583**	**1391443**

As you can see from the above table, most of the boot time is taken by the FPGA Full Configuration task and Hard Memory Configuration task.

The reason that FPGA Full Configuration took the longest time is because it need to read around 14 MB of FPGA configuration bit-steam from Flash storage and reroute the same amount of data to the FPGA control block to shift the data into FPGA to setup the user hardware design. The reading part of this task is affected by the choice of the Flash technology, for example the SD card could be running at 25MHz or 50MHz, the QSPI can be in quad mode or single mode. The programming part also has been further reduce with the support of Early IO release features of the latest version of UEFI bootloader released together with SoC EDS version 16.0. This is done by optimizing the configuration algorithm in to two separate phases.

The Hard Memory Configuration actually is the bootloader waiting for the Hard Memory Controller to report back the DDR SDRAM calibration status and the type of DDR SDRAM it detected. The HPS processor is essentially just waiting and doing nothing during this time. Keep in mind that the calibration wait time depend on what type of DDR memory the board use and not related to UEF bootloader itself.

DEVICE TREE SUPPORT

A common misconception about UEFI is that one must use ACPI instead of Device Tree in UEFI. In reality UEFI Development Kit comes with support for both. The FdtLib for Device Tree support is located within the EmbeddedPkg folder of the UEFI source tree.

This comes in handy because when using SoCFPGA, the design phase start with hardware tools featuring the Altera's Quartus development environment and the Qsys system integration tool. Once the hardware design is ready, the handoff utilities will take the Quartus and Qsys output files and automatically generate a handoff files for the UEFI bootloader firmware to consume. The handoff files are known as the Device Tree (.DTS/.DTB files) which contains information about all user-customizable settings of the SoCFPGA represented in a structural format. Two different type of handoff can be produce by the handoff utilities, type one is for OS such as for Linux boot consumption and type two is a what we call Flattened Device Tree (FDT) form for boot loader consumption. UEFI firmware uses the FdtLib library to support extraction of information from Device Tree Binary (.DTB) file.

DIAGNOSTIC FEATURES

SoCFPGA being a highly configurable device is actually harder to debug when compare to (for example) a microcontroller, due to it has more parameters to look at when issues arise debugging support of any kind will be meet with warm welcome by the engineer using it.

When a new board is first power-on, the most feared situation by most firmware engineers is "nothing happened", the board appear to be powered but the processor does not appear it is executing the boot loader firmware because there is no output on the serial console terminal nor any form of LEDs pattern indicating the board is booting. Inexperience engineer may jump quickly into conclusion that it is due to broken firmware. In reality, to successfully bring up a new board involves 4 different aspects of activities from 4 different parties. First the factory need to assembly the board with all the components soldered correctly. The hardware engineers need to measure important signals to make sure it is within specification allowable range; if not a board rework may be required. Then only it came to potential firmware issue where firmware engineer need to re-Flash with working image. Finally the operating system booted and the application level software engineer started debugging and recoding the higher level software.

UEFI APPLICATION

An UEFI application is an executable binary with file extension .EFI than run under the UEFI Shell environment. The different of UEFI application with the bare-metal application is that bare-metal application is compiled as to an elf file with .AXF file extension which is then converted to binary format using FROMELF tool to become the final bootimage.bin which you can boot to from UEFI bootloader. When you boot to the bare-metal application, you do not intent to return back to UEFI bootloader until the next reset, in contrast, when you exit a UEFI application (exit here means the main C function returned) it return back to the UEFI shell environment. It takes very little effort to convert any bare-metal application to become a UEFI shell application. There is a separate application note on this topic.

The benefit of UEFI application is that your application resembles typical operating system style application but when you write the application code you can directly write to the hardware registers in contrast to when writing for Linux or Windows operating system the code that touches hardware need to be written as an OS driver separated from the application layer. So there is much less work involved. You can have can multiple different application on a single SD card for different purpose, execute each of them and return back to the shell environment and then execute a different one. If you use the boot-image approach, you will have to build multiple SD card image for different boot image applications.

UEFI BOOTFLOW

BOOTROM PRE-BOOTLOADER STAGE

The requirement before the UEFI Boot loader can boot is that the CPU must be started by running the BootROM which will find a copy of UEFI Bootloader image from one of the Flash storage device then copy it to the on-chip RAM and transfer control to it after verifying it has a valid header. The UEFI Bootloader build process automated the creation of correct BootROM header.

EARLY OCRAM BOOT STAGE

The first major task UEFI boot loader performed is to decode the Device Tree Binary (.DTB) file concatenated to the end of UEFI Boot loader PEI.ROM image. Assuming all information in the device tree is correct, PLL clock frequency, IO pins settings all peripherals held in the reset will have been initialized successfully. However if some clock frequency or IO mux are set incorrectly it might have causes hang at early boot stage before UART is available to send any characters to the serial terminal.

UEFI Bootloader is designed to support easier debug at this stage of boot. After finished compiling the source code, user can find a DS-5 Script template for doing source level debug generated under the build folder. The steps needed are to launch DS-5, import the script and then step though the code causing the problem. To further the depth of analysis, an informative Semi-hosting log will be produced during runtime, this is very helpful because it will dump out the decoded device tree information in human readable form, and because it is a common mistake that outdated handoff .DTB file is being used with newer hardware during development cycle. UEFI Bootloader for SoCFPGA also support memory serial log feature which come as a backup tool when both UART and semi-hosting fail in some odd situation.

LATE OCRAM BOOT STAGE

This is a stage when clocks, I/O pins and Flash devices are configured and it is time to program the FPGA with user custom design stored in a bit-stream file (.RBF file). The Arria 10 SoC architecture required that the FPGA to be program first before DDR memory can be up and running due to the fact that hard memory controller (HMC) and EMIF (external memory interface) resides in the FPGA with configuration part of the bit-stream.

What could possibility goes wrong in this stage is may be the user forget to insider the micro SD card, no .RBF file inside the Flash device, FPGA programming failure or memory initialization failure. To allow easy debug of these possible scenario, diagnostic features that can be switch on or switch off are built into a control panel like .DSC file which is implemented from EDK II DSC File Specification. When the diagnostic switch is turned on, additional information is printed for engineer to analyze the situation, this consume ROM space, thus allowing it to be turn off speed up boot time and have more space for customer board specific code.

DDR SDRAM READY STAGE

If memory diagnostic switch are switch own, users not only can see the timing parameters being set to the DDR Scheduler, the UEFI Bootloader take an extra mile to draw up the entire memory map structure based on detected DDR size in as a short summary which is very useful for engineer who is unfamiliar with the new system to help understand the memory layout structure.

At this point, all SoCFPGA specific initiation has been done, however user might which to find a place to add their custom board specific initialization code before loading their next stage boot image. UEFI Bootloader provides a place to file call Board.c with function entry point that will be call at this stage.

READY TO BOOT TO NEXT STAGE

The ready to boot to next image stage is reach as the last job of the PEI.ROM image. User will choose to boot the optional DXE.ROM image only when required to boot an UEFI aware operating system such as Windows, Mac OS, Windows and Android. In most case, the user of UEFI Bootloader for Arria 10 SoCFPGA will only boot simple bare-metal application or RTOS like Green Hills Software's INTEGRITY or Wind River’s VxWorks.

To ensure a clean handoff to next stage boot image, the boot loader will ensure that the MMU is turned off, CPU instruction cache is disabled and invalidated, CPU data cache is also disabled and cleaned, the Global Interrupt Controller (GIC) is disabled and with all pending interrupt if any are acknowledged, before making the CPU handoff.

CONCLUSION

The Altera implementations of UEFI Boot loader firmware for SoCFPGA devices make use of the scalable architecture of open source UEFI development Kit. When it comes to choices of boot loader firmware for a SoCFPGA based device, Altera is the first SoCFPGA Company that adopted this future-proof technology firmware solution ready to support modern features like 64-bits processors, hypervisor and secure boot. Our UEFI Bootloader is design to ease customer board bring up in mind with jam packed list of powerful diagnostic features. Start using UEFI Bootloader today.