WHITE PAPER

Deploy Fast & Secure IoT Solutions with Insyde® Software BlinkBoot® and Intel® Firmware Support Package (Intel® FSP)

Executive Summary

Insyde® Software BlinkBoot® is a UEFI-based boot loader that leverages Intel® Firmware Support Package (Intel® FSP) to deliver simple, fast and scalable firmware solutions for Internet of Things (IoT) platforms. When you just want to boot quickly, or make your changes without a fuss, take advantage of industry-best standards and avoid getting tangled in the business process. Insyde* BlinkBoot* featuring Intel FSP support, gets you there in a blink.

Tim Lewis
Chief Technology Officer,
Insyde* Software

Stephen Gentile
Senior Vice President, Marketing,
Insyde* Software

Rangarajan, Ravi
Firmware Architect,
Intel Corporation

Wang, Stephanie
IOTG FSP Product Line Manager,
Intel Corporation

Venkataramani, Sathish
Strategic Business Development Manager,
Intel Corporation

Birru, Prasanna
Graduate Business Intern,
Intel Corporation

Insyde* Software BlinkBoot*

You have an idea for a new gadget, or maybe it is the newest generation in a long line of gadgets. You have selected your operating system, the application engineers completed their interface design, and you're working up the backend cloud infrastructure needed to handle the user and sensor interactions. You even have a reference board from Intel that's close, but you want to swap out a part, maybe change the sensors and pin routing, and need to add a recovery path for service technicians.

Then someone brings up the dreaded topic: Basic Input/Output System (BIOS) (or, maybe they are more polite and say "firmware"). Maybe you have dabbled with a licensed codebase. Maybe you went the open-source route, but confronted with two million lines of source code and limited access to silicon support, you threw up your hands and said, "I just want to quickly boot my Operating System (OS) on all the boards I have to support!" This really highlights the first three requirements for any good IoT firmware solution is:

- It must be *simple*: Simple means getting from reset to the OS or your application with minimum fuss, but still giving you the control you need over hardware and feature settings.

- It must be optimized for *speed*: Speed means a solution that includes and executes only what your platform needs.
- It must save time and *money*: Saving time means re-using as much of the engineering work as possible between product lines and product generations by leveraging industry standards. Saving money means having the right business model.

This is where BlinkBoot* with Intel FSP comes in. The FSP provides the intelligence to initialize Intel's silicon and BlinkBoot provides the simple, fast environment for your product's firmware.

What is Intel® FSP?

In rapidly evolving Internet of Things ecosystem with multiple innovative solutions, IoT developers look for simple, economical and scalable firmware solutions to support devices based on Intel® architecture.

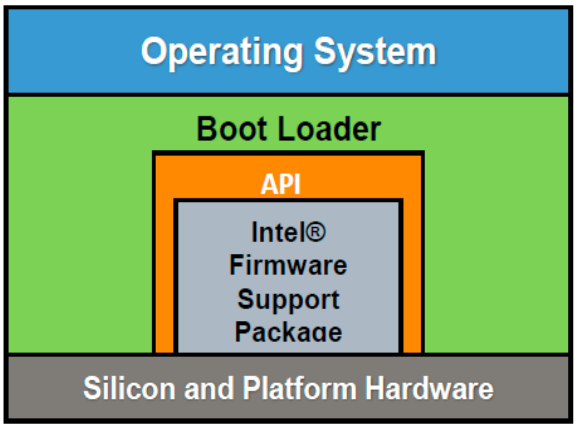
Focusing efforts on design aspects and value added differentiators rather than complexities of silicon initialization is the gateway for faster time to market for IoT designs.

Intel FSP provides silicon initialization in a binary format that is easy to adopt and reduces Time to Market (TTM). Intel FSP is provided free of cost and helps to build economical firmware solutions for IoT designs.

Intel FSP provides standard Application Programming Interface (APIs) making it easy to be integrated into different bootloaders. The *Intel® Firmware Support Package External Architecture Specification* defines a consistent set of APIs that helps design scalable firmware solutions.

It is important to note that Intel FSP is not a stand-alone bootloader; it needs to be integrated into a boot loader to make a complete bootable solution.

Figure 1. A System with Intel® FSP and Bootloader



Design Goal of Intel® FSP

Intel holds the key to the knowledge in initializing the silicon it produces. Silicon initialization is complex because of the wide feature set and the requirement to support a wide variety of designs with software configuration. For example, an Intel® Atom™ System-on-a-Chip (SoC) may support different memory technologies like DDR3L, LPDDR3 and LPDDR4 types with different data rates and data densities Moreover the SoC may support runtime detection of various memory topologies (channels and slots) and other advanced features, like memory training that dynamically computes the optimal configuration parameters depending on the board layout. The number of lines of initialization code that can support all these features may be 100,000 plus lines code - the silicon initialization contributes a few more thousand lines.

Integrating the silicon initialization code provided by Intel into bootloaders takes time and adds cost. Some of the initialization of advanced features includes intellectual property which also requires encapsulation.

Intel FSP is a binary distribution of silicon initialization code required to bring the silicon usable by industry standard software.

The design goal for Intel FSP is to provide a mechanism that abstracts the complexity of silicon initialization through well-defined interfaces, while the binary format of Intel FSP helps in abstracting the complexity of silicon initialization.

Benefits: Intel FSP – Boot Performance, Configurability and Scalability.

Performance: Intel FSP with its focus on silicon initialization, optimizes the initialization flow by employing several different techniques. The FSP performs only the necessary initialization required to bring the silicon usable by standard drivers, reducing the overhead by consolidating the silicon initialization in fewer modules to avoid dispatch time, and packaging the modules to satisfy dependency requirements naturally, etc.

Moreover, Intel FSP leverages the underlying silicon code features like fast MRC where pre-trained memory configuration is used and thereby reducing the memory initialization time and doing a full memory boot only when explicitly requested. Another feature enabled by Intel FSP is early graphics that allows a splash screen to be displayed to show signs of life very early in the boot process.

Configurability: Intel FSP provides a configuration interface for initializing silicon in many different ways. Including options like enabling and disabling of features, options to configure the controllers in different modes, and so on.

Moreover, Intel FSP supports both static configurations using the Binary Configuration Tool (BCT) as well as a bootloader runtime configuration through the FSP API interface. BlinkBoot*, for example, uses this method extensively.

Scalability: Intel FSP is scalable and easy to adopt. By abstracting the silicon initialization complexity through a well-defined set of interfaces, it lowers the threshold to get firmware enabled for an IoT design. Since the Intel FSP interfaces are consistent, once the interface to Intel FSP builds inside a bootloader, developers can use future releases of the Intel FSP binary and expect it to work with minimum porting effort. Since the Intel FSP is released after validation, plugging in a validated module helps to reduce validation efforts and time as well.

Since Intel FSP doesn't serve as a standalone boot loader, this brings into picture boot loaders to make Intel FSP a complete bootable solution, Insyde* Software's BlinkBoot which is one such solution.

What is BlinkBoot*?

BlinkBoot* is a fast, secure and customizable boot loader that is:

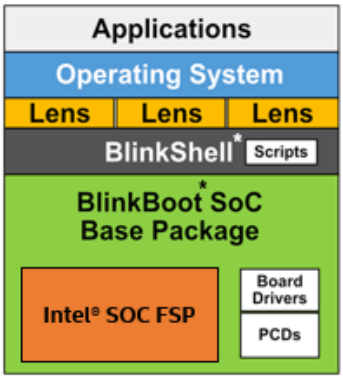
- **Fast**: BlinkBoot is designed to execute the minimal number of instructions necessary to boot a device in the fastest time possible. It initializes silicon, prepares hardware, and loads an OS kernel (e.g. Linux*) in milliseconds.
- **Small and Modular**: BlinkBoot allows you to add in only the pieces you need, leaving plenty of space in your flash device for pre-OS applications or even an entire OS image.
- **Based on UEFI**: Insyde* Software's Unified Extensible Firmware Interface (UEFI) implementation has shipped on hundreds of millions of platforms since the UEFI standard premiered over a decade ago. UEFI is a mature, well-understood, and flexible standard supported by nearly every modern operating system and silicon vendor.
- **Secure**: Built on UEFI's proven security infrastructure, BlinkBoot* secures the flash device and provides flexible authentication of all OS loaders and pre-OS applications.
- Packaged with unique tools:
  - **BlinkFlash*:** a script driven flash image editor and updater.
  - **BlinkShell***: a tiny implementation of the UEFI shell that gives full-control of the platform pre-boot.
  - **BlinkBuild***: our powerful build customization engine that lets you make configure settings before a build, during boot or even for the next boot.
- **Available** on a variety of IoT platforms.
- **Supported** by Insyde* Software, one of the most trusted suppliers of firmware for Intel platforms.
- **Licensed:** in a way that makes sense for IoT products.

Let's look at how BlinkBoot provides you a simple, fast and time-saving firmware solutions:

Simple

BlinkBoot comes in "base packages" that support a single Intel SoC and come configured for a single reference board. The BlinkBoot base packages strip out all of the non-essentials from the firmware, focusing on getting from reset to the BlinkShell prompt as quickly as possible.

Figure 2. BlinkBoot* Base Package



However, you are probably not shipping the reference board and you likely have your target on something more ambitious than a BlinkShell application. How can you make the necessary changes without earning a 2nd Ph.D.?

Let's take a look at how BlinkBoot does it, and how FSP's design helps.

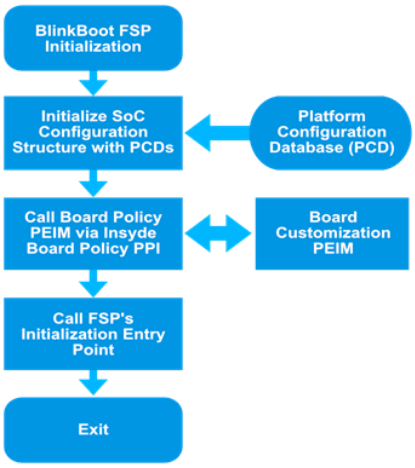
BlinkBoot customizations come in four flavors, all controlled from a single directory.

- **PCDs:** Platform Configuration Databases (PCDs) are build-time configuration settings controlled from a single project file and compiled into a database that is used during the boot process. PCDs control the inclusion or exclusion of features and default values for many structures.
- **Board Drivers**: For each of the UEFI pre-OS environments (Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), and System Management Mode (SMM)), the board driver provides runtime customization for initialization structures (including those used with the FSP) and callback hooks at all significant architectural milestones. This allows the developer to centralize board-related customizations in a single place, such as memory Serial Presence dDtect (SPD) data, General Purpose Input Output (GPIO) setup, or Super Input/Output (I/O) or Application Specific Integrated Circuit (ASIC) initialization.

NOTE: Intel's FSP simplifies this process because it uses a single configuration structure. BlinkBoot defaults this structure using PCDs, passes it to the board driver for updates, and then passes the updated version to Intel FSP for SoC initialization.

- **Scripts**: The BlinkBoot base package finishes by launching the startup script for BlinkShell—a UEFI-compliant shell. BlinkShell scripts can launch UEFI applications, load lenses, or launch operating systems. For example, BlinkShell scripts can load flash-resident applications, create RAM disks, or choose boot devices based on a GPIO or UEFI variables.
- **Lenses**: Lenses are add-ons to the base package that support a specific technology, such as USB or Debug or Setup or Networking. Lenses are collections of drivers, support applications and scripts that are packaged together. Most can be loaded from secondary storage. For example, if your OS is on a USB device, a BlinkShell startup script would load the USB lens and then launch the OS from that device.

Figure 3. BlinkShell* Initialization



Speed

There is a period of time, after the power button is pressed, where the platform is not doing what you want to do with your product and you have no control. It feels like a black box during this period and then, seconds later, poof, it boots.

With BlinkBoot*, this happens in less than 500 ms by getting you to the BlinkShell* prompt. From there, you have control over what happens next. Whether you are loading USB support from a Lens and then launching your OS, or launching it directly from flash, the choice is yours.

How does BlinkBoot achieve this?

By not doing things twice. In many products, the firmware and OS have nearly identical code for initializing devices such USB, video, SATA and eMMC*.

BlinkBoot focuses on doing only enough initialization so that the OS can take control of these devices cleanly. The devices must be discoverable and identifiable by the OS, which means they must be present on an industry-standard bus (like Peripheral Connect Interface (PCI) or Universal Serial Bus (USB) or described by an industry standard (such as Advanced Configuration and Power Interface (ACPI)).

Intel FSP simplifies this process because a single binary package contains much of the SoC initialization needed. BlinkBoot then adds the missing industry standard initialization (such as SMM, microcode and PCI support) and other device discover (via ACPI and SMBIOS).

Of course, some of these devices may be necessary in your product's boot path. Which leads to a corollary, our second key:

Load only what you need, when you need it. Many IoT devices have few boot device choices, no keyboard and no video. So BlinkBoot lets you choose which ones you load, on which ports, and under which conditions using BlinkShell scripts. This is true for storage devices, keyboard, mouse and video. Of course, the OS already knows how to use these but does your IoT firmware need to do this?

Reduce accesses on slow buses. Speeding up the boot means reducing usage of hardware registers with their attendant latency. The boot process comes to a grinding halt waiting for SMBus or I²C* or SPI transactions to complete. [1]For example, reading the SPD data from a DIMM requires about 7 ms in fast mode (400 Mhz) or 15 ms in slow mode (100 Mhz). That may not seem like a lot, but at 500 ms total boot time, two Dual In-line [2]Memory Modules (DIMMs) can be 30 ms or 6%. [3]The same is true with embedded controller (such as I/O port 0x60/0x64 or 0x62/0x66), super I/O registers (I/O ports 0x2e/0x2f) and Management Engine (ME) interactions. It also means that Serial Peripheral Interface (SPI) flash device access speed plays an important role. For example, decompression into Random Access Memory (RAM), rather than simple copy into RAM, gives better performance because of the reduced SPI flash device cycles.

[1] I²C performance data was derived from experiments recorded at from http://grist.livejournal.com/31371.html, retrieved 1 August 2016. They recorded 53.333 µs per byte at 400 kHz and 120 µs per byte at 100 kHz. There are 126 bytes per DIMM, resulting in 6.7 ms and 15 ms per DIMM respectively.

[2] These calculations assume the I²C* bus is running in slow (100 kHz) mode and two DIMMs are used. Based on data above, this means 30 ms. The 500 ms boot time reported represents typical BlinkBoot* performance from reset to the BlinkShell* prompt on a 1.6 Mhz Cherry Hill board, with two add-on lenses installed

[3] The performance data of communication with devices on the LPC bus, such as embedded controllers and super I/O controllers is discussed in https://en.wikipedia.org/wiki/Low_Pin_Count#Timing_and_performance (retrieved 1 August 2016). This time is above the actual time required for the device's operation.

Scalability

When you spend time writing firmware, you want it to keep working, even if you change the OS, the version of the firmware used, or even on another platform. That's what widely adopted standards like the UEFI found in BlinkBoot* are for: they provide a solid, well-documented foundation.

BlinkBoot saves you time by allowing to write code once and then having the control to leverage that across multiple platforms or multiple generations of platforms.

Consider these scenarios:

Scenario #1: 3rd Party Integration

You are integrating a 3rd party piece of silicon and it needs some pre-OS initialization or, worse, you need to boot from it. How do you share code with the vendor?

- With BlinkBoot (and UEFI) each driver is compiled separately, into its own executable. Interfaces between drivers are well-documented and discovered at runtime. This has several practical implications:
- If the 3rd party silicon follows the UEFI rules, it will work with BlinkBoot.
- The 3rd party silicon can compile a UEFI driver and it will load and work with BlinkBoot, even if one was built using Linux* and GNC Compiler Collection (GCC) and the other was built using Windows* and Visual Studio*.

The 3rd party silicon can distribute just the binary of the driver, without distributing the source code, so no Intellectual Property (IP) is exposed.

The question for the 3rd party hardware vendor is no longer: what revision of this open-source firmware project did you use? It is: Which is the minimum version of the specification required?

Scenario #2: Firmware Feature Re-use

- If you develop a feature on one project, it can easily be shared with another project, because they share the same technology foundation.

Working with an industry standard like UEFI, drivers written for one project can easily be shared with another project. It can be shared on another platform the same family. It can be shared with another engineer who is familiar with the standard. In fact, you can just copy the driver executable onto a USB key, walk over to the other system and just load it using the UEFI-standard UEFI shell commands in BlinkShell*.

Taking this one step further, using BlinkFlash*, you could actually insert the UEFI driver into the BlinkBoot image for the second project without actually having the original build environment.

Scenario #3: Long Product Lifespan

If your product has an extended lifespan of five or seven- years or longer, there are special considerations: will my chips still be available? Will my operating system version get hacked? Will marketing come up with some whiz-bang feature they want to support part way through this time?

BlinkBoot is built on top of UEFI. One of the good things about widely adopted standards like UEFI is that they consider backward compatibility as a part of the design process, often explicitly documenting how certain scenarios are handled for older code even as they move forward. So when version X+1 is released, it is done with awareness that an OS or a firmware stack will still have to deal with version X.

That gives the code you write today longevity.

Case Study: Embedded Platform with Custom ASIC, Memory Down and GPIO-based Selection of Boot Device

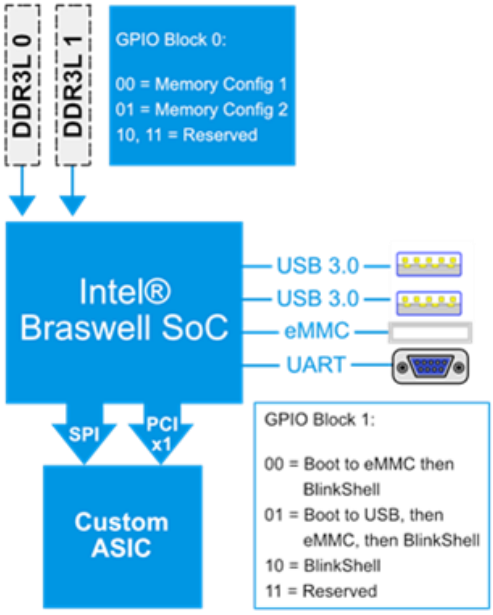
This case study looks at a fairly common embedded design based on the Intel® Pentium® and Celeron® Processor N3000 Product Families and Intel® Atom™ x5-E8000 Processor (formerly Braswell). The board sits at the middle of a custom signal processing solution, with an ASIC doing most of the heavy lifting for a custom application running on top of a tailored Linux* distribution.

The OS and application normally sit on an eMMC* device soldered on the motherboard. But the design also called for a backup or recovery application running on a USB 3.0 mass storage device.

The memory devices are also soldered to the motherboard, in one of two shipping configurations (8 GB and 16 GB). Since there are no SPD Electrically Erasable Programmable Read-Only Memorys' (EEPROMs) tied to the lower-power DDR3 chips on the board, two GPIOs are used to select which configuration is present.

The ASIC has two connections: a normal PCI Express x1 link and also a Serial Peripheral Interface (SPI). The device doesn't respond as a PCI device until after it has passed some initialization data over the SPI link, so this has to happen fairly early in the boot process so that the normal PCI enumeration can process it.

Figure 4. GPIO Block Diagram



Also notable is the lack of Serial Advanced Technology Attachment (SATA) and video, so there is no need to include either of them. Debug and console output are handled through the high-speed UART.

Memory Configuration

Typical desktop platforms use DIMMs with the memory configuration information supplied by an EEPROM packaged with the memory devices. But many IoT platforms use memory without these EEPROMS (to save space and cost) and instead rely on the firmware to provide the same memory configuration data.

BlinkBoot* works hand-in-hand with Intel FSP to supply the platform-specific memory configuration data needed for this platform. Each instance of FSP has one or more data structures that control the silicon initialization, including the memory controller.

BlinkBoot provides a simplified interface for the developer to use to provide these structures. Using a board-specific driver provided with BlinkBoot, each initialization data structure is set to build-time defaults and then passed to the `UpdatePpi()` function, along with an identifying Global Unique Identifier (GUID). Here's an example of the basic skeleton with usage with the Braswell FSP. (Refer to Figure 5)

Figure 5. Configure I2C3_SCL and I2C3_SDA as GPIO

```
// Configure I2C3_SCL and I2C3_SDA as GPIO[0:1]
UINT32 I2c3SdaCfg0 = CONF_MMIO_ADDRESS(GPIO_MMIO_OFFSET_SW, BSW_PAD_NUM_I2C3_SI
UINT32 I2c3SclCfg0 = CONF_MMIO_ADDRESS(GPIO_MMIO_OFFSET_SW, BSW_PAD_NUM_I2C3_SC
MmioWrite32(I2c3SdaCfg0, PAD_GPIO_EN | (2 << 8) | PULL_20K;
MmioWrite32(I2c3SclCfg0, PAD_GPIO_EN | (2 << 8) | PULL_20K;
UINT32 Gpio = mioRead32(I2c3SclCfg0) & 1 + ((MmioRead32(I2c3SdaCfg0) & 1) << 1)
if (Gpio == 0) {
// Enable Channel 0 with LPDDR3 data
MemInit->PcdMemChannel0Config = SolderDownMemory;
MemInit->PcdMemorySpdPtr = (UINT32)&LPDDR3SpdData;
// Mark Channel 1 as empty
MemInit->PcdMemChannel1Config = DimmDisabled;
} else if (Gpio == 1) {
// Enable Channel 0 & 1 with identical LPDDR3 data
MemInit->PcdMemChannel0Config = SolderDownMemory;
MemInit->PcdMemChannel1Config = SolderDownMemory;
MemInit->PcdMemorySpdPtr = (UINT32)&LPDDR3SpdData;
} else {
DEBUG((“Unknown memory configuration %d. Trying configuration 0.\n”, Gpio);
MemInit->PcdMemChannel0Config = SolderDownMemory;
MemInit->PcdMemorySpdPtr = (UINT32)&LPDDR3SpdData;
MemInit->PcdMemChannel1Config = DimmDisabled;
}
```

Using this driver is especially important for this design because the example platform actually has two different memory configurations. Selection of the correct configuration is determined by a GPIO. On Braswell, there's one additional twist, as the GPIO configuration data and the memory configuration data are in the same data structure. But we can't initialize the memory configuration data without reading the GPIO! So we have to add a little bit of GPIO initialization code first (refer to Figure 6).

Figure 6. GPIO Initialization Code

```
EFI_STATUS
PeiBoardPolicyUpdatePpi (
IN CONST EFI_GUID *PpiGuid,
IN OUT VOID **PpiStructure
)
{
DEBUG ((EFI_D_INFO, "PEI Board Policy Update called for Guid:
%g\n", PpiGuid));
if (CompareGuid(PpiGuid, &gEfiFspCustomizePpiGuid)){
// FSP silicon initialization structure.
UPD_DATA_REGION *UpdData = (UPD_DATA_REGION *)*PpiStructure;
// FSP's memory configuration data structure.
MEMORY_INIT_UPD *MemInit = &UpdData->MemoryInitUpd;
…insert memory configuration updates here…
}
return EFI_SUCCESS;
}
```

ASIC Initialization

This embedded platform uses a custom ASIC that must be programmed with a downloaded configuration data set via the SPI connection. In addition, the motherboard has a simple Super I/O chip that provides two UARTs and a PS/2 style mouse and keyboard interface. The initialization for both of these devices must occur early. In particular, the ASIC data download must complete and be reset before PCI enumeration begins.

BlinkBoot provides the milestone mechanism to provide a callback at critical, architecturally-define points in the boot flow. In this case, the milestone `PEI_MS_INIT` will work perfectly. The `BoardCustomPei` driver provides a skeleton version of the milestone function, and can be easily extended using the PCI and I/O libraries provided with BlinkBoot (refer to Figure 7).

Figure 7. BoardCustomPei Driver

```
UINT8 AsciInitData[0x4000] =
{
  ...
};

typedef struct {
  UINT8 Register;
  UINT8 Value;
} SIO_DATA;

SIO_DATA SioInitData[] =
{
  // COM1 Initialization
  {SIO_LDN, SIO_COM1},
  {SIO_BASE_IO_ADDR1_MSB, 0x03},
  {SIO_BASE_IO_ADDR1_LSB, 0xf8},
  {SIO_IRQ_SET, 0x4},
  {SIO_DEV_ACTIVE, 1},

  // COM2 Initialization
  {SIO_LDN, SIO_COM2},
  {SIO_BASE_IO_ADDR1_MSB, 0x02},
  {SIO_BASE_IO_ADDR1_LSB, 0xf8},
  {SIO_IRQ_SET, 0x3},
  {SIO_DEV_ACTIVE, 1},

  // Keyboard/Mouse Initialization
  ...

  // End of table marker
  {0, 0}
};

const UINT8 IndexPort = 0x2e;
const UINT DataPort = 0x2f;

BOOLEAN
PeiBoardPolicyMilestone (
  IN OUT INSYDE_MS_DATA *PeiMilestoneData OPTIONAL
  )
{
  UINT8 *AsicData;
  UINT32 AsicDataSize;
  SIO_DATA *SioTable;

  // Send out the ASIC initialization data and then reset the device.
  AsicData = AsicInitData;
  AsicDataSize = sizeof(AsicInitData);
  SpiAsicChipSelect(TRUE);
  while (AsicDataSize--) {
    SpiWrite(*AsicData);
  }
  SpiAsicChipSelect(FALSE);
  GpioAsicReset(FALSE);            // assert RESET#
  StallUs(200);                    // stall 200us
  GpioAsicReset(TRUE);             // deassert RESET#

  // Write the SIO configuration data.
  SioTable = SioInitData;
  EnterSioCfgMode();
  while ((SioTable->Register != 0) || (SioTable->Value != 0)) {
    IoWrite8 (IndexPort, SioTable->Register);
    IoWrite8 (DataPort, SioTable->Value);
    SioTable++;
  }
  ExitSioCfgMode();

  return EFI_SUCCESS;
}

// Wrapper function for all PEI milestones.
EFI_STATUS
PeiBoardPolicyMilestone (
  IN     UINT32         PeiMilestone,
  IN OUT INSYDE_MS_DATA *PeiMilestoneData OPTIONAL
  )
{
  switch (PeiMilestone) {
    case PEI_MS_INIT:
      return PeiBoardPolicyInitMilestone(PeiMilestoneData);
  }
  return EFI_SUCCESS;
}
```

Boot Option Selection

While UEFI solutions (including BlinkBoot) can use the traditional Boot Manager to select the boot device based on the contents of non-volatile UEFI Variables, BlinkBoot offers another alternative. Since BlinkBoot always passes through BlinkShell*, it is possible to divert boot to a device and executable of your choice using a simple shell script. In this case, we want to read a GPIO and then use standard script commands to mount the right storage devices and launch the OS.

This simple script file takes four parameters: %1 = the type of value used, %2 = the address of the value used, %3 = a delimiter and %4 = the value read. This style is used, because it matches the output from the 'mm' command used in Figure 8.

Figure 8. BootOs.nsh

```
#if the input parameter is 0, then load/connect USB. Otherwise, load/connect SATA
if %1% == 0 then
  lens -i usb
else
lens -I sata
endif
# Launch the standard UEFI boot loader from the recently mounted device.
Launch0s
```

The script file in Figure 9 `startup.nsh` is executed every time BlinkShell launches. In this case, it redirects the output from a built-in command (mm) which reads memory-mapped I/O, into an environment variable, and then passes that environment variable as the parameter to the `BootOs.nsh` shell script defined in Figure 8.

Figure 9. startup.nsh:

```
mm 0xfed8000 –w 1 >v tmp
BootOs %tmp%
```

IoT Friendly Business Model

BlinkBoot* was designed to address many of the business challenges that IoT device makers face when preparing to bring a new product to market. These business imperatives include, but are not limited to:

- Implementing a lightweight boot loader optimized for IoT devices
- Deploying a cost-effective licensing model
- Using a standards based UEFI solution that aligns with firmware used on other product lines
- Finding a reliable and tested solution sourced from an experienced firmware vendor

BlinkBoot meets these challenges and more by simplifying the licensing model and providing a focused UEFI-based solution targeted at single-application, dedicated-function Intel IoTG platforms.

BlinkBoot is royalty-free and pricing is based on product usage (single product, product line or product family) and the required add-on lenses. The base product license includes six months of free maintenance and available for extended periods.

Conclusion

When you just want to boot quickly, make your changes without fuss, take advantage of industry-best standards and not get tangled in the business process. BlinkBoot*, featuring Intel FSP support, gets you there in a blink.

For more information:

BlinkBoot*:

https://iotsolutionsalliance.intel.com/solutions-directory/blinkboot%C2%AE-uefi-based-boot-loader

Intel® FSP:

http://www.intel.com/content/www/us/en/intelligent-systems/intel-firmware-support-package/intel-fsp-overview.html

Printed in USA Deploy Fast & Secure IoT Solutions with Insyde® Software BlinkBoot® and Intel® FSP Please Recycle 334779-002US