Technical Paper

Data Center & Artificial Intelligence
Firmware Solution

coreboot on Eagle Stream

A guidebook for FSP-coreboot firmware solution on Eagle Stream platform

Introduction

Flexibility, as one of the concerns in server deployment of internet business, is getting more attentions in today's market with growing diversity. For example: high performance computing may prefer more CPU cores to scale up. Cloud service calls for huge amount of DRAM to hold VMs. AI business requires more PCI/CXL interfaces to plug compute cards. Each scenario has its own performance metrics and could receive a boost by customizing hardware or firmware, to better serve the business, and to squeeze out the potential of CPU.

Although Intel has a tradition to provide several reference platform designs and associated UEFI firmware stacks, they cannot cover all the use cases by CSPs, so OEMs and IBVs are involved to make variants of the reference platform, and cooperate with CSPs and Intel to develop drivers and features for the final product. This routine has been paced for a fair long time.

However, this tradition now appears to be inefficient and too slow to meet the fast-changing demand of the market. From a perspective of firmware, UEFI calls for a unique skill set and hence takes a long time to breed developers, which has subsequently slowed down the development of firmware products. On the other hand, a smaller community also means a lower chance to uncover those sneaky defects, which result in longer bug shooting cycles.

In comparison, Linux* uses popular tool chains, possesses enormous drivers as well as a giant community with participation of scientists, engineers, and hobbyists. It still flourishes after decades of evolution while UEFI firmware remains almost proprietary. In this situation, absorbing Linux* into firmware becomes an appealing topic since there is no reason to reject diversity in looking for flexibility.

To embrace this possibility, Intel used to have developed 'MinPlatform', a simplified UEFI firmware, which does only the memory and silicon initialization job, and hand over control to a Linux* kernel thereafter. The kernel will then take care of onboard devices initializations. This solution invokes Linux* at late-DXE phase, which is right before when the production OS is loaded, but it works, as a first try of LinuxBoot* in cloud firmware.

Then the Firmware Support Package (FSP) was introduced. FSP packs tedious memory and silicon initialization routine into binaries and provides interfaces to let them be called from outside. This mechanism makes it possible to also leverage open-source firmware stacks at pre-mem phase like coreboot, which will be the main topic to be covered in this article.

Reading Guide

This document covers two major topics: "coreboot Basics" and "Application Guide". In the basic section, we elaborate on tool chain, boot flow and call trace in each phase. While the latter one is specifically on how to set up coreboot firmware stack on Intel's Eagle Stream platform with 4th Gen Intel® Xeon® Scalable processors.

If you are looking for a quick setup and do not care about the design of coreboot

Table of Contents

architecture, feel free to skip the first section and go directly to the application guide. If you are interested in the mechanism of coreboot or intended to contribute to this open-source firmware project, going through the first topic will help clearing the way for you.

Glossary

BIOS: A legacy firm, now vaguely referred to as the complete set of software running between operating system and hardware. Equals to “firmware” in this document.
Bootloader: A modern concept in replace of “BIOS”, referred to as a framework, often provides bootstrap code bringing processors alive and interfaces for third parties to do platform-specific initialization.
CAR: Cache-As-RAM, a special operation mode of cache that allows it to be used as RAM.
CSP: Cloud Service Provider, third-party companies offering cloud-based solutions.
EGS: Eagle Stream, the reference server platform for 4th Gen Intel® Xeon® Processor Scalable Family, Codename Sapphire Rapids.
FSP: Firmware Support Package. A set of software packed in binaries, provided by Intel for silicon and platform initialization. Could be integrated by bootloaders. Contains three parts: FSP-T (Temp RAM), FSP-M (Memory), FSP-S (Silicon).
IBV: Independent BIOS vendor.
Payload: A software to be executed when bootloader exits and OS yet to be loaded. Could be a Linux* kernel or UEFI shell. It usually takes care of device drivers and other initialization routines according to firmware design.
SMM: System Management Mode.
SPR: Sapphire-Rapids, the codename of the 4th Gen Intel® Xeon® Processor Scalable Family.

Reference

[1] "coreboot documentation," [Online]. Available: https://doc.coreboot.org/.

[2] "kconfig documentation," [Online]. Available: https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

[3] "coreboot: Starting from scratch," [Online]. Available: https://doc.coreboot.org/tutorial/part1.html.

[4] "System Management Mode," in *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2021, pp. vol 3C, Chapter 31.

[5] "coreboot architecture," [Online]. Available: https://doc.coreboot.org/_images/comparison_coreboot_uefi.svg.

[6] "u-root," 2023. [Online]. Available: https://github.com/u-root/u-root.

[7] "coreboot Flashmap Descriptor," [Online]. Available: https://doc.coreboot.org/lib/flashmap.html?highlight=flashmap.

[8] "coreboot table: code," [Online]. Available: https://github.com/coreboot/coreboot/blob/master/src/commonlib/include/commonlib/coreboot_tables.h.

1 coreboot Basics

This chapter is a recapitulation of coreboot documentation [1], with focus on what we think to be the most helpful to coreboot starters. Including how to configure, build, and not to get lost in the confusing source tree.

Figure 1. 'coreboot' should be all-lowercase, with no space

1.1 Toolchain and Repository

1.1.1 The Build System

coreboot uses a GNU make build system, with custom language to support various configurations of projects. In many ways, this system is like the one of Linux*, for the author had obviously held such intention to resemble Linux* in development. Hence naturally, coreboot has become an out-of-the-box firmware with generic support for Linux*.

In addition to the 'Makefile' file, coreboot toolchain also requires 'Makefile.**inc**' file in its custom language rule. Distinguished by file extensions, these '.inc' files separate coreboot build into different *classes*. A typical snippet of '.inc' file could be:

```
bootblock-y += bootblock.c spi.c lpc.c pch.c
romstage-y += romstage.c reset.c utilc spi.c pmutil.c
ramstage-y += memmap.c pch.c lockdown.c finalize.c
ramstage-$(CONFIG_HAVE_ACPI_TABLES) +=
uncore_acpi.c acpi.c
```

Figure 2. '.inc' files control which class the source to be compiled into

'bootblock', 'romstage' and 'ramstage' are the most important *classes* to describe different parts of coreboot build. We will have detailed discussion on it later. Basically, they are different phases during boot process. We can add '-n' or '-y' suffixes following with source name to determine when and where our code should take place.

Notably, there is only one Makefile at top level, but many .inc files (one per subdirectory), which means coreboot has a generic build process to which every platform-wised configuration must comply. And that is how we do it.

1.1.2 Repository Hierarchy

As an open-source firmware project, coreboot must support various mainboards that come from varied brands, different manufacturers, with different board designs or even chip architectures.

To accommodate such diversity, coreboot use a straight-forward method, that is to place source into directories conforming a Processor-Platform-Board hierarchy.

For example, first comes the generic type of processor (x86, ARM, RISC and so on), then it is a specific processor model that must be applied on dedicated platform (4th Gen Intel® Xeon® Processor Scalable Family, Codename Sapphire Rapids on Eagle Stream in our case), and finally it is a more specific board model (the board of your project).

Moreover, coreboot provides **callbacks** according to such hierarchy. For example, coreboot has many generic initializations, there may be say a *coreboot_init*() function at some point. But for extensibility, this method must have a nested *soc_init()* callback for specific processor models, developers should overwrite this callback function to port to their SoCs, and finally *platform_init()* or *mainboard_init()* callback shall also be nested. These callbacks have been defined as weak methods and would directly return if not overwritten.

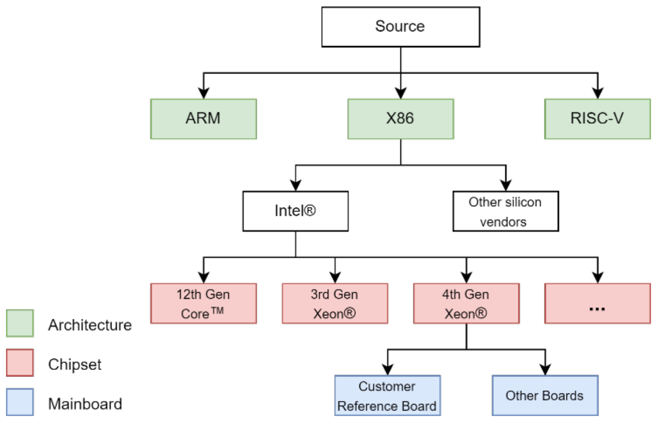


Figure 3. The structure of the coreboot source tree

1.2 Config and Build

1.2.1 Kconfig

coreboot uses Kconfig, which has also been also adopted by Linux* as the main configuration mechanism (the source is integrated under util/ by coreboot). If you are familiar with the Linux* config system, then this one of coreboot may not be a thing, but in case you are confused, referring to Kconfig docs [2] may be a good choice.

```
config MAINBOARD_DIR
        string
        default "intel/archercity_crb"
config MAX_SOCKET
        int
        default 2
config BOARD_SPECIFIC_OPTIONS
        def_bool y
        select BOARD_ROMSIZE_KB_65536
        select IPMI_KCS
        select VPD
        select MAINBOARD_HAS_TPM2
        select MAINBOARD_USES_FSP2_0
        select SOC_INTEL_SAPPHIRERAPIDS_SP
        select SUPERIO_ASPEED_AST2400
        select HAVE_ACPI_TABLES
```

Figure 2. Sample Kconfig snippet of Archer City

Essentially, Kconfig allows you to define variables and set default values for them to be used in source code. They can also be nested in each platform or mainboard directory.

As shown in Figure 4 Sample Kconfig snippet of Archer City, we defined the path to our board *intel/archercity_crb*, it is used by coreboot build system to locate the source. Then a variable 'MAX_SOCKET' with default value '2'. It can also be referred in source as 'CONFIG_MAX_SOCKET' and has int value '2'.

Besides, a 'menuconfig' make target could also be used for graphical Kconfig setting in coreboot.

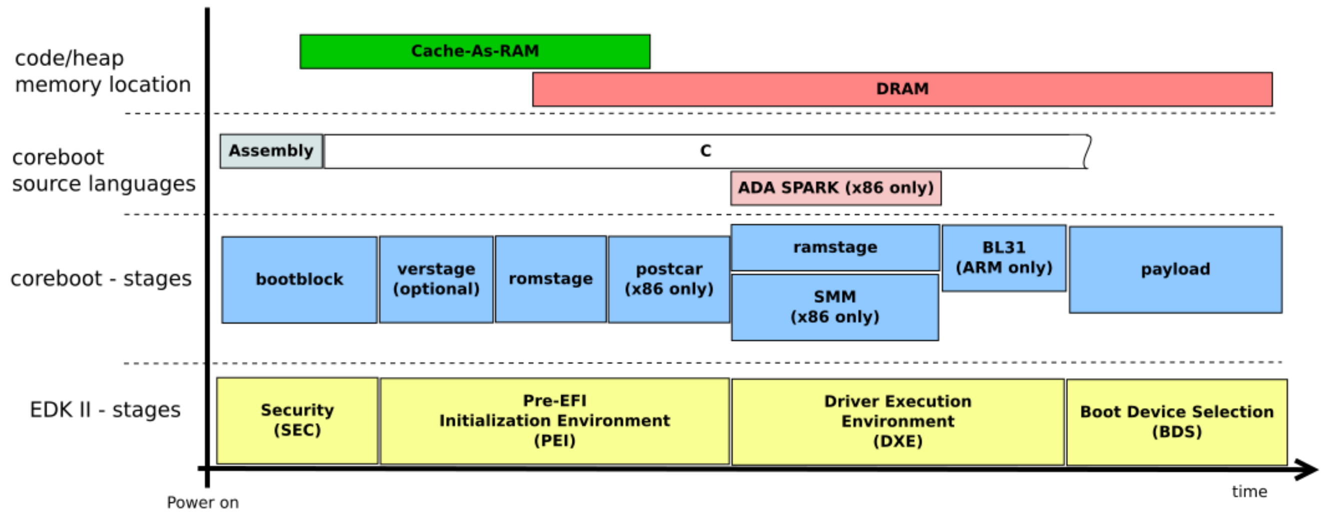


Figure 3. coreboot in a nutshell [5], and a comparison with UEFI firmware.

1.2.2 Build the Project

Refer to 'coreboot: Starting from scratch' [3] webpage for guidance.

1.3 Boot Flow

coreboot has three main stages in its boot flow: bootblock, romstage, and ramstage (as shown in Figure 5). Each stage has its own entry code for environment setup. In this way, they are mutually independent.

1.3.1 Bootblock

The first stage after machine power-up. Upon entering this stage, CPU will fetch its first instruction which is called 'reset vector' from a fixed address. The reset vector has been placed by coreboot at top sector of flash ROM, it contains a jump to the bootstrap code in flash image. The bootstrap code, written with assembly, differs per different processor type. But in our case (4th Gen Intel® Xeon® Processor Scalable Family, Codename Sapphire Rapids on Eagle Stream), it looks for the entry point of FSP-T in flash image and execute in-place.

Now that FSP-T enters, does its magic, load microcode and setup Cache-As-RAM (CAR). Once CAR is available, coreboot sets up protected mode for CPU and let C-code into the play. After all these preparations, the bootblock *main()* function would now come up.

Bootblock *main()* is like '*SecMain()*' on UEFI side. It does early initializations such as console and GPIO inits. You could also override the *bootblock_mainboard_early_init()* callback here to do early setup for your project. But there is not much to do, bootblock is a transient stage and will soon pass its control to Romstage, to do the main memory setup.

1.3.2 Romstage

In the past, Romstage was pre-compiled with ROMCC, which turns C code into stack-less assembly code. But now situation has changed since CAR is available. Romstage simply calls into the hook *fsp_memory_init()* on entry, and standby until main memory is ready.

During its lifetime, FSP-M will cover uncore initialization, main memory training, memory topology and clustering setup and so on. It takes up almost 80% of Romstage time, and returns with pointers to Hand-Off-Blocks (HOBs), which are data structures preserved in memory containing memory training result and crucial system information).

When FSP-M exits, main memory is standing by, and the control is again returned to coreboot. Romstage would now preserve a region called '**cbmem**' in main memory. This region prevails through all stages, from boot time to OS, thus can be used to hold global variables, HOBs, ACPI tables and so on.

Finally at the end of Romstage, everything is migrating to main memory, so CAR will no longer be used and will be torn down. This is done by calling into FSP again via the hook *late_car_teardown().* Sometimes this stage is called 'post-CAR'. After it returns, Romstage would also exit.

1.3.3 Ramstage

Like the other two stages, Ramstage also starts with calling FSP, but FSP-S (that is, Silicon) in its case. FSP-S takes care of proprietary CPU and platform features such as Intel® Software Guard Extensions (Intel® SGX) and Intel® Server Platform Services (Intel® SPS). It is also at this time that the firmware flash ROM is locked to prohibit any further modification.

Notably, Ramstage is when coreboot does the init work during boot time. It is accomplished by many boot state machines called 'Hardwaremain' state machine. Marked as '*BS_{FUNCTION}*'. By design they are generic, with hooks for chipset and mainboards customization. In the case of Eagle Stream, these state machines make up three major phases:

PCI init: When FSP-S exits after a relatively short time, coreboot starts to scan PCI devices according to IIO stack info. In this process:
BS_DEV_ENUMERATE scans PCI devices and update them to the device tree.
BS_DEV_RESOURCE allocates memory resource like MMIO windows for devices.
BS_DEV_ENABLE sends commands to let devices operate properly.
BS_POST_DEVICE provides hooks for additional feature initializations like PCI Advanced Error Reporting (AER) mechanism.

MP init: In this phase, coreboot sets each CPU thread into correct state and assign interrupt controller IDs, to bring inter-processor-interrupt service online.

Besides, this phase also covers System-Management Mode init for each thread, including SMRAM allocation and SMBASE relocation. The flow is devised according to *Software Developer's Manual* [4]

So far, MP init involves *BS_DEV_INIT_CHIPS* and *BS_DEV_INIT.*

ACPI fill-out: This phase comes after the other two above, with only one state machine *BS_WRITE_TABLES.* ACPI and SMBIOS tables can be filled here. These tables are allocated in cbmem, and their locations will be reported through system memory map to operating system.

A simplified version of Figure 5 coreboot in a nutshell [5], plus a comparison with UEFI firmware. indicating the boot flow is shown next.

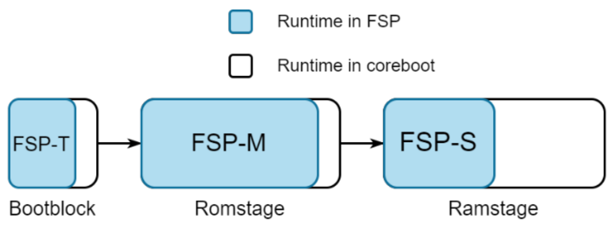


Figure 6. Runtime in each stage, the wider the longer

1.3.4 Payload

In a way, Payload is all that coreboot is meant for. Back in the time when DRAM was not as complicated as today's, 'coreboot' only consisted of very few steps, the last of which was copying a Linux* kernel from ROM into main memory, so that the abundant device drivers and utilities from Linux* community could be used for platform init. The Linux* kernel at that time was the first instance of Payload, since then, all the evolution of coreboot serves for only one goal: "Getting chipset and memory ready for Payload to do the rest".

The category of Payload has expanded now and not only includes Linux* kernel, but also Tianocore, GRUB and U-Boot and so on. Notably, a Payload image must reside in ROM and hence is still within the scope of firmware.

Payload is like a counterpart of post-DXE phase of UEFI firmware, it provides an environment for firmware applications such as PXE-boot* and busybox*. The difference is that coreboot does not define the feature of Payload or come with one, which means Payloads must be integrated as third-party components.

And it is the duty of Payload to boot the machine into the final operating system.

2. Application Guide

This section is particularly useful for engineers who are looking for quick practices with coreboot. In the scope of this document, we demonstrate a best-known method of porting coreboot to the Eagle Stream platform.

Before everything, make sure that all dependencies are met (1.2.2 Build the project) and xgcc toolchain has been built:

$make crossgcc

2.1 Create Project Folder

Mainboard instances take up a unique level of coreboot source tree. You can find them under *src/mainboard/.* The instance of Eagle Stream Customer Reference Board (CRB) is located under:

src/mainboard/intel/archercity_crb

This will be the basis of our practice. To work on it, we need

```
config VENDOR_MYORG
        bool “My org name”
```

Figure 7. Sample Kconfig.name of 'myorg'

to first create a directory that represents your organization or company (for example, *src/mainboard/myorg),* then copy the *Kconfig* and *Kconfig.name* file from *src/mainboard/Intel/* into your org folder and change their values accordingly.

2.1.1 Kconfig of Organization

Now we created a new vendor named "My org name" (see Figure 7). with *src/mainboard/myorg* as directory. Do not forget to also change the corresponding values in *Kconfig*, shown as next:

```
If VENDOR_MYORG
choice
   prompt “Mainboard model”
source “src/mainboard/myorg/*/Kconfig.name
endchoice
……
endif
```

Figure 8. Sample Kconfig file of 'myorg'

2.1.2 Kconfig of Mainboard

For the next step, copy the entire Archer City CRB folder to your org directory, and rename it to:

src/mainboard/myorg/my_mainboard

Here we have a duplicated instance with all configurations including GPIO settings identical to Archer City CRB. Note that there are also *Kconfig* and *Kconfig.name* files in the board instance folder, they also need to be modified according to our mainboard. Starting from *Kconfig.name* (see Figure 9), then all relative fields in *Kconfig* and board_info.txt.

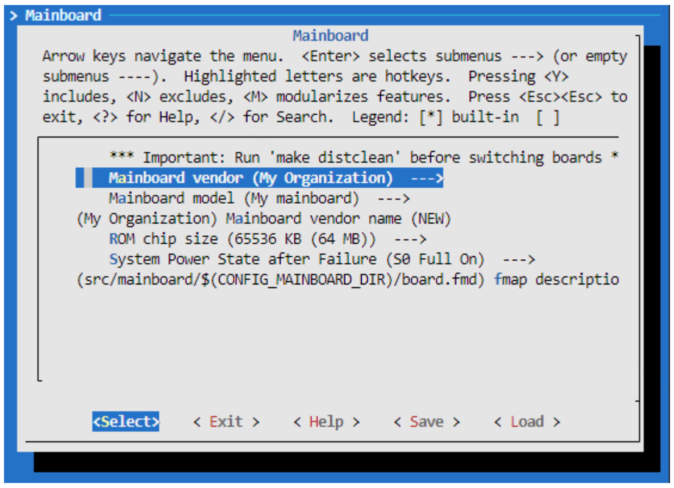
```
config BOARD_MYORG_MYBOARD
        bool “My mainboard”
```

Figure 4 Sample Kconfig.name of 'my_mainboard'

Make sure the key config **MAINBOARD_DIR** in Kconfig has been set to '*myorg/my_mainboard*'.

After all steps being done correctly, you will see a visible entry in the 'mainboard' list prompted by command line:

$make menuconfig



```
> Mainboard
                              Mainboard
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

        *** Important: Run 'make distclean' before switching boards *
        Mainboard vendor (My Organization)  --->
        Mainboard model (My mainboard)  --->
    (My Organization) Mainboard vendor name (NEW)
        ROM chip size (65536 KB (64 MB))  --->
        System Power State after Failure (S0 Full On)  --->
    (src/mainboard/$(CONFIG_MAINBOARD_DIR)/board.fmd) fmap descriptio

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 5 The mainboard we just created

2.1.3 Use defconfig

The 'menuconfig' (Figure 10) provides a convenient GUI that allows us to change basically every configuration declared in Kconfig files. However, there is a much more efficient one-click solution for this: using defconfig files.

defconfig files are generated by command line:

$make savedefconfig

They are extracted records of different mainboard configs, and can be re-used anytime by command line:

$make defconfig KBUILD_DEFCONFIG="defconfig_path"

This is equivalent to setting configurations via menuconfig, but with only one click. The defconfig for Archer City CRB is *src/configs/builder/config.intel.crb.ac*. Again, we can copy this Kconfig and rename it to:

src/configs/builder/config.myorg.myboard

Now we have a defconfig file which spots out all pivot configs of Intel Archer City board. By reading this file, it is easy to have knowledge of what is changed in the build. In our practice, we merely created a new vendor "my org" and a new board "my board". So, for the first hand, we will change VENDOR_INTEL to VENDOR_MYORG, and ARCHERCITY_CRB to MYORG_MYBOARD.

```
CONFIG_VENDOR_MYORG=y
CONFIG_BOARD_MYORG_MYBOARD=y
…
CONFIG_IFD_BIN_PATH = "…"
CONFIG_ME_BIN_PATH = "…"
CONFIG_GBE_BIN_PATH = "…"
…
CONFIG_FSP_T_FILE = "…"
CONFIG_FSP_M_FILE = "…"
CONFIG_FSP_S_FILE = "…"
…
~~CONFIG_PAYLOAD_LINUX=y~~
~~CONFIG_PAYLOAD_FILE = "…"~~
CONFIG_PAYLOAD_TIANOCORE=y
CONFIG_TIANOCORE_UPSTREAM=y
…
```

Figure 11 Image components and payload option in sample defconfig config.myorg.myboard

As shown in the example above, the default defconfig of Archer City CRB uses LinuxBoot* as payload, in which case an external LinuxBoot* payload file shall be provided. We recommend U-root* + latest kernel solution here, for build instructions please refer to the U-root* GitHub* repository [6]. But if you will not bother to investigate it, it is fine to change the default payload to Tianocore, and let coreboot build it from EDKII repository automatically.

Attentions are required here that till now we are not yet ready to build the project. According to Figure 12, there are other binary components in the firmware volume that need to be added, including "Intel Firmware Descriptor" (IFD), "Management Engine" (ME), "Gigabit Ethernet Conf File" (GbE) as well as FSPs.

2.2 Binaries and Image Layout

To boot on Eagle Stream platform, the firmware image must comply Intel firmware layout. We could use the Intel® Modular Flash Image Tool to decompose a UEFI IFWI from Intel BKC release, and get the IFD, Intel® ME, FBE binaries mentioned in previous section. Then on the coreboot side, we use an ".fmd" file (Figure 11) to control how to re-compose these binaries into coreboot firmware volume. Logic and syntax of this flash-map descripting file is well documented at [7].

The ".fmd" file of our project is under mainboard folder:

src/mainboard/myorg/my_mainboard/board.fmd

```
FLASH@0xfc000000 64M {
    SI_ALL@0x0 0x03000000 {
        SI_DESC@0x0 0x1000
        SI_GBE@0x1000 0x2000
        SI_ME@0x3000 0x2fed000
        SI_PT@0x2ff0000 0x10000
    }
    RW_MRC_CACHE@0x3000000 0x18000
    FMAP 0x800
    RW_VPD(PRESERVE) 0x4000
    RO_VPD(PRESERVE) 0x4000
    COREBOOT(CBFS)
}
```

Figure 12. The firmware volume memory-mapped at 0xFC000000, with 4K-Descriptor aligned at bottom and 16M-CBFS at the top

By default, this file is the same with Archer City and will not be touched. We only need to prepare corresponding binaries and feed their paths to defconfig file.

2.2.1 Use site-local Directory

The "site-local" directory is an optional folder ignored by coreboot git repo yet still be visited by its make and Kconfig system. "The intent is to provide a single location to store local modifications." – Says coreboot documentation. There is no magic of it. You could create another GitHub* repository named "site-local" in the top-level of coreboot repo anytime and manage local changes like binaries inside.

The only thing remarkable is that coreboot has actually integrated site-local (if available) into build process, which means you can have custom *"Makefile.inc"* and *"Kconfig"* in this folder to control local changes even they are ignored.

In our practice, for example, we could create a folder named *"site-local/myboard"* and put all required binaries inside, then modify the defconfig:

```
CONFIG_IFD_BIN_PATH = "site-local/myboard/ifd.bin" …
CONFIG_FSP_T_FILE = "site-local/myboard/Server_T.fd" …
```

2.3 FSP Integration

To correctly integrate FSP into coreboot, FSP headers are also important other than binaries. Hence for each set of FSP binaries, you must fetch corresponding UPD and HOB headers (*FsptUpd.h, FspmUpd.h, FspsUpd.h and *Hob.h).* and put them under:

src/vendorcode/intel/fsp/fsp2_0/sapphirerapids_sp/

2.4 Build the Project

After all configs, binaries and headers are ready, we can now build the project via command line:

$make clean && make distclean
$make defconfig KBUILD_DEFCONFIG= "configs/builder/config.myorg.myboard"
$make -j $job_count

The *$job_count* is the maximum supported CPU thread count of your source building machine, in the case of 32-thread it will take only ~8 seconds to finish the build. But if you have selected CONFIG_PAYLOAD_TIANOCORE, it would take some extra few minutes to build the payload from EDKII.

When the build is done, copy and flash build/coreboot.rom into the ROM on your board and try to boot.

3. Development Guide

This section provides supplementary reading for firmware developers, including a minimal guide on porting ACPI and SMBIOS tables to customized platforms with different GPIO configurations, plus a brief introduction to coreboot SMM architecture and RAS feature tuning methods.

3.1 coreboot Table

At the end of Ramstage, yet before payload is loaded, boot state jumps to *BS_WRITE_TABLES* when most system tables are being written to reserved locations in RAM. Among these tables, there is one uniquely named as "coreboot table" which worth being mentioned here.

"The coreboot table is for conveying information from the firmware to the loaded OS image. Primarily this is expected to be information that cannot be discovered by other means, such as querying the hardware directly." [8] Hence on x86 platform, coreboot table is suitable for passing memory mapping info to OS, in which case it is similar to the E820 table on EFI side.

The pointer to coreboot table is passed to payload as a parameter in boot state *BS_PAYLOAD_LOAD* so that Payload could read and parse the pointed mem region for system information. The "Root System Descriptor Pointer" (RSDP) is also exposed in coreboot table, for compatibility with legacy, it is copied to lower memory (0xF0000) to be auto detected by OS.

Architecturally, coreboot table is just one allocated entry in cbmem, which is the same with ACPI and SMBIOS tables and global non-volatile variable area.

3.2 ACPI and SMBIOS

SMBIOS is enabled by default on x86 platform, while ACPI is toggled by Kconfig *HAVE_ACPI_TABLES.* Both of these two types of tables are cbmem items allocated at high memory region, besides, they both have entry point addresses copied to lower memory around 0xF0000.

ACPI implementations locate in *src/acpi/acpi.c,* within

```
for (dev = all_devices; dev; dev = dev->next)
    if (dev->ops && dev->ops->write_acpi_tables) {
        current = dev->ops->write_acpi_tables(dev, current, rsdp);
        current = acpi_align_current(current);
    }
```

Figure 6 ACPI fill-out routine for all devices

function "*write_acpi_tables()".* The function creates generic ACPI tables like SSDT, DSDT, and FACS. For any table that needs platform or mainboard specific modifications, we can create corresponding "__weak" function hooks and override them in private code.

Additionally, for each device on the platform, we could add an ACPI creating method (Figure 13) and link it to the device's *"write_acpi_tables"* pointer (different from the previous function, this is a member function in device driver with same name)

SMBIOS is implemented in a similar way. The main routine locates in *src/arch/x86/smbios.c: smbios_write_table(),* platform specific changes can be hooked with *"get_smbios_data"* in device drivers.

3.3 GPIO pins

Onboard GPIO pins can be initialized at very early phase in Bootblock. To do this, a common way is to carve a GPIO table (pin definitions usually lies in *gpio_defs.h*) that reflects the board config, then pass it as a parameter to function *gpio_configure_pads()*. This function can be invoked at any stage via mainboard overrider, in the case of Bootblock it is *bootblock_mainboard_early_init()*.

3.4 SMM and RAS Features

On teh 4th Gen Intel® Xeon® platform, Intel provides only three FSP binaries: FSP-T, FSP-M, FSP-S, none of these covers System Management Mode (SMM) initialization or System Management Interrupt (SMI) handling, so coreboot has to take over this part.

SMM-related features are controlled by a main switch *HAVE_SMI_HANDLERS* defined in x86 Kconfig. On Intel® Xeon®SP platform, it is enabled by default. Once enabled, coreboot will cover SMRAM relocation and SMI handler registration during multi-processor init. This happens in boot state *BS_DEV_INIT*.

The symmetric multiprocessing environment for SMI handling has been re-designed by coreboot community, so it differs a little from the one of UEFI firmware. Generally, the re-designed environment has 2 major rules:

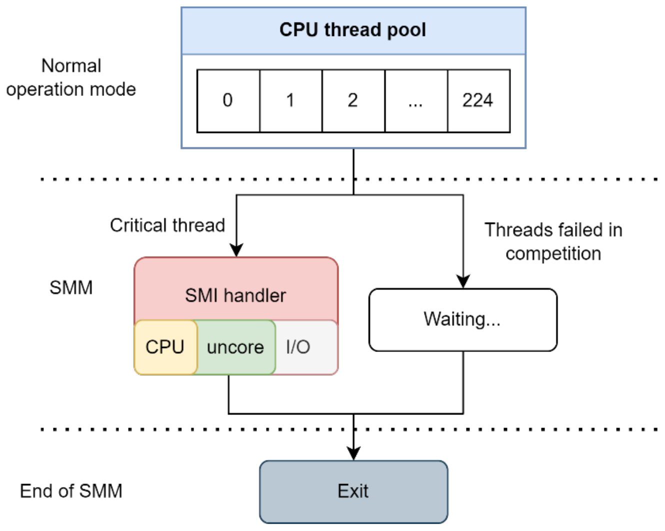


Figure 7. SMM module execution flow

1. Only 1 thread is allowed to handle SMI event at a time, others entering SMM will wait until being released.
2. Threads are allowed to enter SMM asynchronously; that is, SMI events can be handled even while some threads are not in SMM.

RAS features can be integrated into SMI handler as a common solution. Related code could be found in *smm_module_handler.c*. In the main handler function *smm_handler_start()* where CPU, uncore and I/O events are handled in order.

At present, there is one example RAS feature *SOC_RAS_ELOG* (that is, Enhanced Log) selected on Intel® Xeon® platform. The feature embeds additional error reporting mechanism, which saves fatal machine check errors as detailed log to cbmem, into SMI handler. The entire operation needs a medium containing info about where error logs are saved to be passed to OS, so it is often coupled with Kconfig *SOC_ACPI_HEST*.