View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Springer - Publisher Connector

CHAPTER 4

Building coreboot with Intel FSP

Empowerment of individuals is a key part of what makes open source work, since in the end, innovations tend to come from small groups, not from large, structured efforts.

—Tim O'Reilly

The Introduction of coreboot

Since 1999, developers from around the world, some as individual contributors and others working on behalf of businesses and corporations, have formed a community around coreboot, an open source firmware project. *coreboot* is boot firmware primarily focused on x86 processors and chipsets, but other processors, like Alpha, PPC, and ARM-based systems are supported. The coreboot logo is a European Brown Hare, Figure 4-1.

Figure 4-1. *coreboot logo*

coreboot firmware deals directly with system hardware configuration. As silicon has become more complicated, with more features and integrated peripherals, firmware developers have had to rely more and more on the silicon vendors for reference code and binaries for the latest silicon releases. Many silicon vendors have tried different solutions to help the developers in the community; for example, AMD's AGESA (AMD Generic Encapsulated Software Architecture), and now, Intel FSP (Firmware Support Package). With the support of silicon vendors, coreboot developers are able to develop and release current silicon devices and to concentrate on peripheral and platform customization.

We are excited to introduce you to the coreboot project. In this chapter, we will cover many of the different aspects of coreboot. The first few sections of this chapter lay the groundwork for working with the coreboot community. We cover the history of coreboot, coreboot's open source software development practices—including details on using Git, and how to build a sample coreboot image. Later in the chapter, we examine the technical details of coreboot, including the binary image structure, the execution flow, and the source code organization. The final sections include information about payloads, debugging, and optimizations for coreboot. Feel free to skip ahead and come back to these sections if you want.

The Philosophy of coreboot

coreboot is built on the belief that users and vendors deserve an open, fast, customizable, and purpose-built firmware for silicon and mainboard initialization. coreboot is designed to do critical hardware initialization before passing control to a payload.

The coreboot philosophy aligns with the Intel FSP philosophy. The coreboot hardware initialization framework handles the FSP silicon initialization API, configures system peripherals, and loads the payload.

Since coreboot is focused on hardware initialization, it does not contain any BIOS or other runtime services. Services, runtime code, and the operating system boot are provided by a payload. coreboot supports a number of different payloads, for disk boot, network boot, and legacy BIOS services. coreboot is often used to boot Linux, but depending on the payload, it can also boot most versions of BSD, Windows, or any other OS. While not part of coreboot, payloads are integral to a complete coreboot firmware image.

coreboot source code is licensed under the GNU General Public License, version 2 (GPLv2). This is the same license that the Linux kernel is released under. The GPL is a share-alike license, which means that each developer benefits from the efforts and the knowledge of the entire community, adding to the success and growth of the project. There are several restrictions about what you can and cannot do with GPL source code, which are clearly documented on the GNU web site at http://www.gnu.org/licenses/licenses.html#GPL. You need to be aware of this and should consult legal experts before integrating GPL code into your own proprietary code.

■ **Note** Payloads are separate projects and have their own license requirements.

A Brief History

coreboot has a long history, stretching back more than 15 years to when it was known as LinuxBIOS. While the project has gone through lots of changes over the years, many of the earliest developers still contribute today.

v1: 1999–2000

The coreboot project originally started as LinuxBIOS in 1999 at Los Alamos National Labs (LANL) by Ron Minnich. Ron needed to boot a cluster made up of many x86 mainboards without the hassles that are part of the PC BIOS. The goal was to do minimal hardware initialization in order to boot Linux as fast as possible. Linux already had the drivers and support to initialize the majority of devices. Ron and a number of other key contributors from LANL, Linux NetworX, and other open source firmware projects successfully booted Linux from flash. From there, they were able to discover other nodes in the cluster, load a full kernel and user space, and start the clustering software.

v2: 2000–2005

After the initial success of v1, the design was expanded to support more CPU architectures (x86, Alpha, PPC) and to support developers with increasingly diverse needs. One of the early design goals was to have as little assembly code as possible. With new and more complex CPUs and DDR initialization requirements, the developers realized that there would be too much assembly code in the firmware. The problem with assembly code is that it is difficult to write and maintain. It also lacks the flexibility and maintainability of a higher language like C. The reason standard C cannot be used in the initial firmware code is because the C compiler requires memory to store variables on a stack.

The first supported CPU memory initialization could be done in just a few instructions of assembly code, but the newer DDR memory controllers required significantly more configuration and a lot more assembly code. To address this problem, Eric Biederman wrote a special “precompiler” called ROMCC that turns C code into

stackless assembly code. ROMCC works around the stack issue by turning the C code into assembly code and using the internal CPU registers to hold all variables. ROMCC is extremely limited in the number of variable and function calls it can support, due to the small number of registers that a CPU has available. The ROMCC-generated assembly is included as an `.inc` file, and then compiled as part of LinuxBIOS. ROMCC could be used until the memory was initialized, and then LinuxBIOS used standard C for the majority of the firmware device configuration code.

As part of the v2 implementation, the LinuxBIOS device tree was introduced. The device tree is based on the PCI bus hierarchy and outlines the system devices. The concept is similar to the Linux kernel's PCI device driver hierarchy and uses some of the same concepts as the Linux tree and driver initialization.

Many target systems had flash devices that were too small to hold both the hardware initialization code and the Linux kernel. Image size was not the only issue. The needs of the users were changing, and additional boot device support was required. Payloads were created for flexible boot device support. A network boot solution was the obvious choice for clusters, so the "etherboot" project was modified to run directly from LinuxBIOS as a payload. Later, a disk-based boot option called FILO was added.

During this period, there were substantial silicon development contributions from Intel, VIA, SIS, Linux NetworX, SUSE, and AMD.

v2+: 2005–2008

The next advancement was the introduction of Cache as RAM (CAR) in 2005. With CAR, the CPU cache was used as temporary memory prior to memory controller initialization. It was a delicate process, but allowed the use of C code after a few hundred lines of assembly.

■ **Note** For more information, see the white paper *CAR: Using Cache as RAM in LinuxBIOS* at `http://rere.qmqm.pl/~mirq/cache_as_ram_lb_09142006.pdf`.

In 2005, Stefan Reinauer, a developer on the project, formed a company named coresystems GmbH to support LinuxBIOS. Stefan was one of the primary developers and co-leaders of LinuxBIOS with Ron Minnich. Stefan's significant contributions included the first AMD64 port, the original ACPI implementation, the original SMM implementation, the flashrom utility, and the FILO payload development and maintainer.

In 2005 the Free Software Foundation (FSF) started the Free BIOS campaign to support LinuxBIOS development. Ward Vandewege, of the FSF, ported LinuxBIOS to the FSF servers and other mainboards.

During this time, the AMD processors become the silicon of choice due the availability of good documents and vendor support. This support included the AMD K-8, Geode, and AMD Family 10 CPUs.

v3: 2006–2008

By 2006, LinuxBIOS had already supported hundreds of mainboards. With so many boards, there were problems with porting additional silicon and systems. Based on lessons learned from v2, LinuxBIOS v3 was a fresh start and a place to experiment and fix major problems. Developers fixed and clarified many driver and bus support issues in the device tree. New features included the new build configuration with Kconfig and a firmware image archive called LAR (LinuxBIOS ARchiver). LAR was improved upon and led to the more refined and flexible concept of CBFS.

v3 had a lot of great technical advancements, but it didn't support many mainboards and it was too unstable for commercial developers. For these reasons, it wasn't the main development branch; it was essentially an R&D branch, where the best ideas were backported to v2.

2008 LinuxBIOS Renamed "coreboot"

LinuxBIOS gained popularity and recognition within the open source community. The name became a bit of a misnomer, since Linux was no longer booted directly from flash, and other payloads and bootloaders had been substituted in its place. Since the original idea was about hardware initialization (core init) and booting quickly, it made sense to rename the project as "coreboot". At this time, co-leader Stefan Reinauer took over as the primary leader of the project, as Ron Minnich focused on other projects.

v4: 2009–2012

coreboot turned 10 years old in 2009. Open source projects should be measured in dog years, and 10 years was a major milestone. In early 2010, coreboot moved from SVN to Git for source control, and during that transition, the community took the opportunity to recognize the advancements of the past 10 years and updated to version 4.0.

coreboot continued to add developers and expanded its user base. Many mainboards were added; one of the largest contributions came from AMD, with the open source release of AMD Generic Encapsulated Software Architecture (AGESA), which started in 2004. AGESA reference code needed to be integrated with coreboot, but at the same time stand alone, as it was code directly from the silicon vendor and the same code used by the BIOS vendors. The initial support was for the AMD Family 14 silicon, but soon grew to include Family 15, Family 16, and the accompanying chipsets.

v4+: 2012–2014

In recent years, several other big vendors have become directly involved as contributors to and supporters of coreboot. The involvement of these vendors has pushed coreboot to be a viable firmware competitor on x86 processor systems at product launch.

In 2012, Google introduced the first x86-based Chromebook with coreboot as the firmware and Chrome OS as the operating system. Since then, Google, in cooperation with multiple computer manufacturers, has released several generations of Chromebooks—all using coreboot. Google is also porting and upstreaming an ARM port of coreboot to promote a consistent and common codebase.

In early 2013, Intel released coreboot FSP support with cooperation and support from Sage Electronic Engineering. Sage has been a coreboot contributor and commercial vendor since 2011, and has developed several coreboot ports with its partners AMD, Google, and Intel.

The coreboot community is also experiencing many new contributors joining it and providing new patches and support. There is a new distribution based on coreboot called *libreboot*. It is a nonproprietary software distribution for the Thinkpad T60. It is a major contribution to the coreboot source code and has the support and endorsement of Free Software developers around the world.

The following are statistics on coreboot (source: http://www.ohloh.net/p/coreboot, May 23, 2014):

- It has had 10,207 commits made by 285 contributors
- It represents 1,597,818 lines of code
- It is mostly written in C
- It has a very well-commented source code
- It has a well-established, mature codebase
- It is maintained by a very large development team
- It is with stable Y-O-Y commits
- It took an estimated 461 years of combined effort (COCOMO model) to create
- It has a codebase of 1,597,818 lines
- It has an estimated cost of $25,353,695

Further Reading

For more information on the history of coreboot, visit the following:

- `$ git log`: All coreboot history is easily accessible
- http://review.coreboot.org
- http://www.linuxjournal.com/article/4888
- http://www.linuxjournal.com/article/7170
- http://www.linuxjournal.com/magazine/coreboot-your-service
- http://www.socallinuxexpo.org/scale8x/blog/interview-ron-minnich-coreboot.html
- https://archive.fosdem.org/2007/interview/ronald+g+minnich

- http://2012.latinoware.org/2012/10/ron-minnich-and-details-of-coreboot/
- http://www.h-online.com/open/features/The-beginnings-746825.html

Prerequisites for Working with coreboot

coreboot uses a typical open source development process. The source code is developed by a community made up of individual contributors. It is submitted to the community for public review prior to being committed to the tree. The code is reviewed for bugs, style, and other improvements. Anyone (even you) can comment and make suggestions during the code review. Developers iterate the code and resubmit it for further review until it is accepted. Once accepted by a senior member, the source is submitted to the coreboot repository.

The coreboot web site (http://coreboot.org) contains a lot of valuable information about the project and it is the first place a new developer should go for information.

The coreboot community does the majority of its communication on the mailing list (http://www.coreboot.org/mailman/listinfo/coreboot) and in IRC (#coreboot on freenode.net).

All code reviews are done in Gerrit (more about Gerrit in a little bit) at http://review.coreboot.org.

If you are using Windows, you might also consider running a Linux virtual machine for coreboot development.

Community Organization

The coreboot community is a flat organization. There is a small leadership group that is informally organized, with Stefan Reinauer as the current chairman, but anyone can review or contribute code to the project. The community is led by developers with commit rights; commit rights are awarded to developers who act in the best interests of the community. These developers participate in the community regularly by developing high-quality code, reviewing other developers' code, and acting as mentors and liaisons for coreboot.

Git and Gerrit

The coreboot source code is maintained at coreboot.org in a Git repository. Git is a distributed SCM (Source Control Management) system that is commonly used in the open source community. We will cover some basic Git commands as part of the development process, but you will want to explore the power and flexibility of Git for your own development (see http://git-scm.com and http://git-scm.com/book).

The coreboot source review process uses the Gerrit tool. Gerrit provides a web-based review of source code with side-by-side differences and user-comment functionality (it also integrates very well with Git). Each Git commit is identified by a SHA-1 hash unique to that change and commit message. The hash is a 40-character hexadecimal sequence,

recalculated with every update to the code or commit message so that Gerrit can't use the hash to track a revision of code already under review. Instead of the commit hash, Gerrit uses a Change-ID hash in the commit message to track a patch through the source code review iteration process. The Change-ID in the commit message doesn't change; and when source is updated and pushed, Gerrit replaces the old version with the new version to be reviewed. The coreboot Git setup automatically adds a Change-ID to the commit message if one doesn't already exist (see https://code.google.com/p/gerrit/).

Git Commit Messages

Each `git commit` has an accompanying commit message. This is extremely helpful to the community; it allows you to see what changed without parsing all the code. Here are a few guidelines for `git commit` messages:

- The first line of the commit message has a short summary of the change. It should have helpful information about the subsection and what changed. It should be no more than 75 characters long.
- Skip the second line.
- The third line is the start of a detailed description. There should be enough information provided that other developers can understand what was going wrong, what changed, and any other relevant details. The description should be informative and clear enough that developers don't need to guess what happened when they read it five years later. Again, lines should never be longer than 75 characters.
- The next line is empty (no whitespace at all).
- The Change-Id line to let Gerrit track this logical change (this is generated by the commit hook).
- The Signed-off-by line according to the development guidelines. (Use `git commit -s` to have Git add your Signed-off-by line automatically. Also see the following "coreboot Sign-off Procedure" section and coreboot's development procedures at http://www.coreboot.org/Development_Guidelines#Sign-off_Procedure).

The following is an example of a well-formatted commit message from coreboot (note the additional lines inserted by Gerrit):

```
commit 48a749a89844ba76ff1564d5009e81d4d8e06db8
Author: Marc Jones <marc.jones@se-eng.com>
Date:   Tue Oct 29 22:13:38 2013 -0600

   intel/cougar_canyon2: Intel CRB FSP based mainboard

   Cougar Canyon 2 is a Ivybridge/PantherPoint reference board.
```

```
This implementation uses the Intel FSP (Visit the Intel FSP website for
details on FSP architecture and support).
   The FSP does not support s3 at this time. S3 may be added
   when it is available in the FSP. All other features and IO
   ports are functional. Booted on Ubuntu 12.04 and 13.04,
   Fedora 18 with SeaBIOS payload. Memtest86, FWTS, and
   other tests pass.

   Board support page will be updated on acceptance.

   Change-Id: I26c0b82d7ac295498376ad4c3517a9d6660d1c01
   Signed-off-by: Marc Jones <marc.jones@se-eng.com>
   Reviewed-on: http://review.coreboot.org/4018
   Tested-by: build bot (Jenkins)
   Reviewed-by: Stefan Reinauer <stefan.reinauer@coreboot.org>
```

coreboot Sign-off Procedure

Before the code can be pushed to coreboot Gerrit for review, the author must follow a sign-off procedure. This procedure is very similar to the Linux sign-off procedure, and the sign-off is enforced by Git and Gerrit tools. You must use your real (legal) name in the Signed-off-by line and in any copyright notices that you add.

By adding your sign-off, you agree to the Developer's Certificate of Origin 1.1.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

a. The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

b. The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

c. The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it; and

d. In the case of each of (a), (b), or (c), I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license indicated in the file.

■ **Note** The Developer's Certificate of Origin 1.1[1] is licensed under the terms of the Creative Commons Attribution-ShareAlike 2.5 License[2].

For more information, see the following web sites:

- http://web.archive.org/web/20070306195036/http://osdlab.org/newsroom/press_releases/2004/2004_05_24_dco.html
- HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/2.5/

Adding Your Sign-off

`git commit -s` will add your sign-off (as set in your git config) to the commit message; for example:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

■ **Note** See http://www.coreboot.org/Development_Guidelines#Sign-off_Procedure for additional sign-off procedure information.

Working with the coreboot Community

An active and productive community is a major component of a successful open source project. As part of any community, it is most constructive if people are civil and considerate of others. This is particularly important in online communities, where people are coming together from different cultures, backgrounds, and levels of technical expertise. Be mindful of one's own place as one among many within the community—in order to be a productive and worthy-of-respect contributor.

coreboot Do's

The following should be done in the coreboot community:

- DO engage the coreboot community e-mail list and IRC channel.
- DO review patches and engage in development discussion.
- DO publish source code for review by the community.
- DO publish small, logical, and understandable patches.

coreboot Don'ts

The following should *not* be done in the coreboot community:

- DON'T violate the GPL or other open source licenses.
- DON'T demand support from the coreboot community.
- DON'T expect every (your) device to have complete support.
- DON'T submit code and ignore the reviews (dump and run).

Nonsource Binaries in coreboot

Even though nonsource binaries have been part of the x86 ecosystem for many years, it remains a touchy subject to incorporate binaries into coreboot. coreboot attempts to use as few proprietary binaries as possible while still providing the base level of support for coreboot users. Binaries are located on the flash with coreboot, without being linked to coreboot. Binaries may include PCI Option ROMs, Video BIOS, payloads, or silicon-specific binaries (like the Intel FSP). Binaries are optional at build time and are not part of the coreboot repository, although some are stored in a SubModule repository called *3rdparty/*. Users may forgo binaries if the feature or capability isn't required. For users looking for a completely free source, the libreboot.org distribution has removed all proprietary binaries.

Intel FSP pairs with coreboot easily. The FSP binary is located at a fixed address within the coreboot image and is accessed with a coreboot driver interface based on the FSP requirements described in Chapter 3. The specific details of where the FSP is located and how the FSP are accessed are covered later in this chapter.

A Hands-on Example: Building coreboot for the MinnowBoard MAX Mainboard

This chapter is meant to provide hands-on training, so we will dive right in, get the code, and use it as reference as we guide you through building and modifying coreboot. There are a few things you will need prior to diving in.

Environment

It is expected that you are building coreboot in a Linux environment and that you are familiar with the standard application and kernel tools. coreboot can be built under most common shells (bash, csh, zsh). coreboot can also be built on BSD and on Windows with Cygwin or MinGW, but that is outside the scope of this book. If you are using Windows, you might also consider running a Linux virtual machine for coreboot development.

- Fedora: `$ sudo yum groupinstall "Development Tools" "Development Libraries"`
- Debian/Ubuntu: `$ sudo apt-get install build-essentials`

The following tools are required to get started:

- GCC/G++
- make
- Git
- ncurses-dev
- flex and bison

Please read the information at http://www.coreboot.org/Build_HOWTO.

Note: Ubuntu dash, the default Ubuntu shell, may have strange failures with the coreboot sh scripts. While coreboot has addressed these issues in the scripts, you might want to update to full bash.

```
$ sudo dpkg-reconfigure dash
```

Hardware: MinnowBoard MAX

The MinnowBoard MAX (MinnowMax) is a low-cost, open hardware development board. It uses the Intel E38xx 'Bay Trail-I' SoC. The compact, low-power, and affordable mainboard is idea for coreboot with FSP development (see http://www.minnowboard.org/meet-minnowboard-max/ for more information).

MinnowBoard MAX Platform Details

Please note the following information on the MinnowBoard MAX:

- SoC: 64-bit Intel E38xx 'Bay Trail-I'
- Video: HDMI Intel Integrated Graphics
- Memory: 1GB or 2GB DDR3
- IO: MicroSD, SATA2, USB3.0, USB2.0, 10/100/1000 Ethernet
- Low-speed expansion ports: SPI, I2C, I2S Audio, 2xUART, 8xGPIO
- High-speed expansion ports: 1xPCIe, 1xSATA, 1xUSB2.0, I2C, GPIO, JTAG

■ **Note** A Bus Pirate or similar device is required to get serial debug information via the low-speed expansion port.

Development Directory

For our example, we do development in `~/fsp_coreboot/`:

```
~/$ mkdir fsp_coreboot
~/$ cd fsp_coreboot
```

You may use any directory that you prefer.

Downloading Intel FSP

The E3800 (Bay Trail) FSP is distributed directly from Intel. You need to download it, uncompress it, and agree to the license before you can use it with coreboot. There is more extensive FSP download information in Chapter 3. The FSP download is at http://intel.com/fsp.

```
Download an Intel Firmware Support Package
 Intel® Atom™ processor E3800 product family (formerly Bay Trail)
   Linux* release version 003 >
```

Installing Intel FSP

Uncompress the `.tgz` file to the development folder. Then, install the FSP.

```
~/fsp_coreboot$ tar -xzvf ~/Downloads/BAY_TRAIL_FSP_KIT_GOLD3.tgz
~/fsp_coreboot$ ./BAY_TRAIL_FSP_KIT.se
                              INTEL CORPORATION
                        RESTRICTED USE LICENSE AGREEMENT
                  INTEL(R) PRODUCTION FIRMWARE SUPPORT PACKAGE
                              (Intel Confidential)

IMPORTANT - READ BEFORE COPYING, INSTALLING OR USING.

...<SNIP>...
```

```
Do you accept the license terms (y/n)? y
Extracting into ~/fsp_coreboot/BAY_TRAIL_FSP_KIT
Finished
```

■ **Note** Be aware that you will need to modify your paths later in the process if you install the FSP somewhere else.

The FSP package contains a number of important components besides the FSP binary. It also contains additional supporting software and binaries, including the Video BIOS and CPU microcode. Again, the FSP package is described in detail in Chapter 3.

Downloading the coreboot Source

The coreboot source download may take a few minutes.

```
~/fsp_coreboot$ git clone http://review.coreboot.org/coreboot
Cloning into 'coreboot'...
remote: Counting objects: 35863, done
remote: Finding sources: 100% (24537/24537)
remote: Total 167717 (delta 11917), reused 163083 (delta 11917)
Receiving objects: 100% (167717/167717), 47.14 MiB | 2.60 MiB/s, done.
Resolving deltas: 100% (121812/121812), done.
Checking connectivity... done
```

This will create a directory called `coreboot/` in the directory that the command was run.

```
~/fsp_coreboot$ cd coreboot/
~/fsp_coreboot/coreboot$ ls
3rdparty  documentation  Makefile.inc  README  toolchain.inc
COPYING   Makefile       payloads      src     util
```

coreboot Toolchain

To help alleviate build problems with many different distribution toolchains, coreboot builds its own small toolchain. The toolchain contains all the tools required to build coreboot and most payloads. We can use a make target to run the `coreboot/utils/buildgcc/buildgcc` script. It builds gcc, libraries, binutils, iasl, and checks for the required tool dependencies.

```
~/fsp_coreboot/coreboot$ make crossgcc-i386
Welcome to the coreboot cross toolchain builder v1.25 (November 19th, 2014)
```

```
Target arch is now i386-elf
Will skip GDB ... ok
Downloading tar balls ...
...<SNIP>...
Unpacked and patched ... ok
Building GMP 5.1.2 ... ok
Building MPFR 3.1.2 ... ok
Building MPC 1.0.1 ... ok
Building libelf 0.8.13 ... ok
Building binutils 2.23.2 ... ok
Building GCC 4.8.3 ... ok
Skipping Expat (Python scripting not enabled)
Skipping Python (Python scripting not enabled)
Skipping GDB (GDB support not enabled)
Building IASL 20140114 ... ok
Cleaning up... ok
```

You can now run your i386-elf cross toolchain from the following directory: `~/fsp_coreboot/coreboot/util/crossgcc/xgcc`.

You can make crossgcc-arm to build the ARM toolchain, but it isn't required for FSP-based mainboards. There is a make crosstools target, which builds additional tools that are not required to compile coreboot.

coreboot Commit Hooks

Back in the "Git and Gerrit" section of this chapter, we discussed the need for a Change-ID to be added to each `git commit`. This is added by the `commit-msg` hook. coreboot also has a pre-commit hook that runs lint on the patch. The commit hooks are set up by the following coreboot make target:

```
~/fsp_coreboot/coreboot$ make gitconfig
```

Creating a coreboot Development Branch

Create a branch in git to do the development on. For the purposes of this book, we will use a specific coreboot commit so that the code is consistent with the instructions and information within. Should you choose, you may use the HEAD code, but HEAD is being actively developed and it may have some differences. The following command creates the branch and sets it to the specific commit that works for the instructions in this book:

commit cf52f9761fef3a8e46ff28d6593e0d573ff1d4ac

```
~/fsp_coreboot/coreboot$ git checkout -b fsp_dev cf52f9
```

Building the Mainboard

The next step is to build the correct mainboard and to direct the build to the FSP and other binaries for inclusion. These settings are shown in Figures 4-2 through 4-5.

```
~/fsp_coreboot/coreboot$ make menuconfig
```

On the Menuconfig Menu

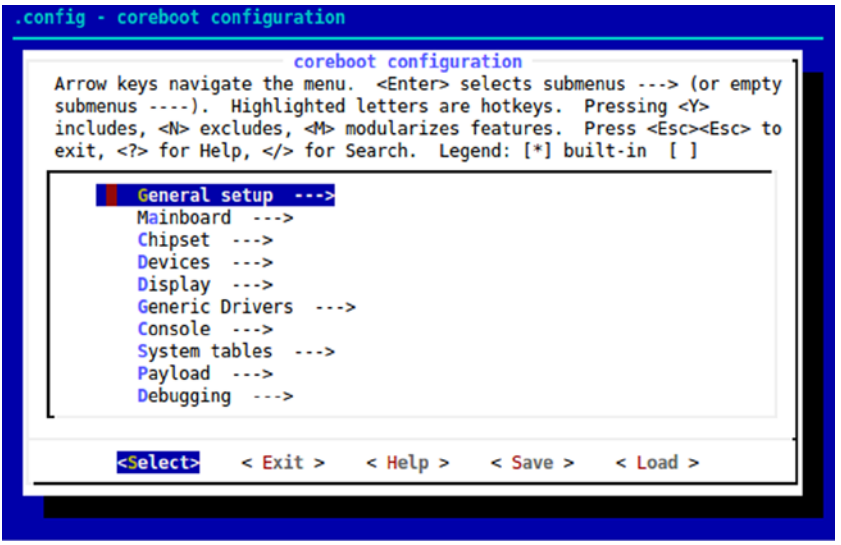

Figure 4-2. *Screenshot of coreboot menuconfig utility*

On the Mainboard Menu

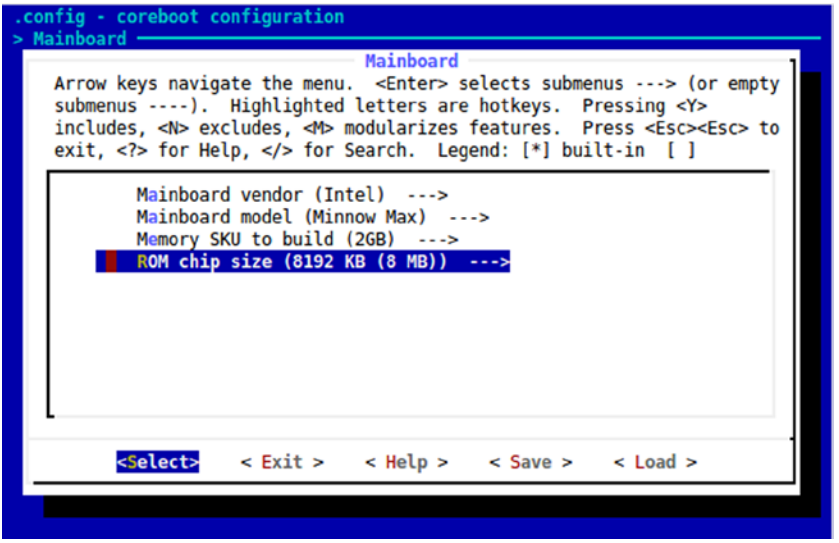

Figure 4-3. *Screenshot of coreboot menuconfig to select Mainboard*

```
Set Mainboard vendor (Intel)
Set Mainboard model (MinnowMax)
Set the Memory Size
Exit the submenu to return to the top level menu
```

On the Chipset Menu

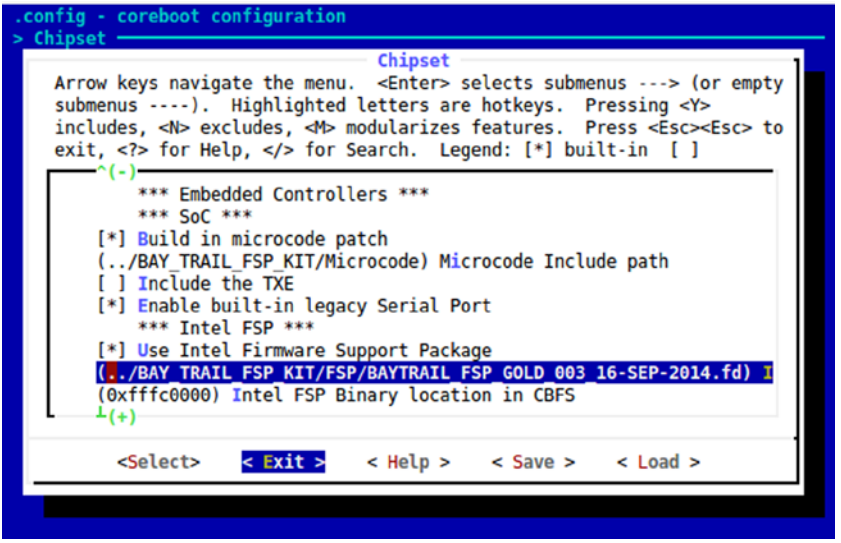

Figure 4-4. *Screenshot of coreboot menuconfig in selecting microcode and FSP path*

```
Set Microcode Path:  ../BAY_TRAIL_FSP_KIT/Microcode
Enable: Enable built-in legacy Serial Port
5Set the FSP file: ../BAY_TRAIL_FSP_KIT/FSP/BAYTRAIL_FSP_GOLD_003_16-
SEP-2014.fd
```

On the Devices Menu

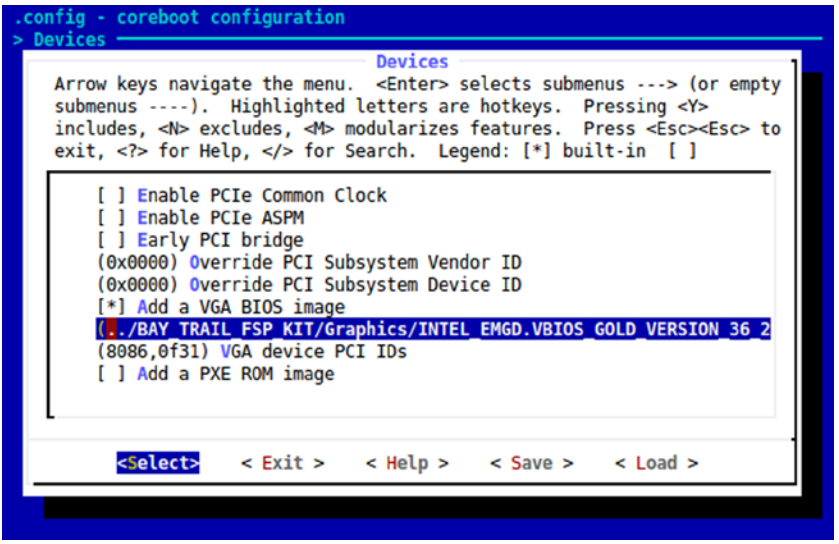

Figure 4-5. *Screenshot of coreboot menuconfig to select VGA BIOS file*

Set the VGA BIOS file, as follows:

```
../BAY_TRAIL_FSP_KIT/Graphics/INTEL_EMGD.VBIOS_GOLD_VERSION_36_2_3_3698/
Vga.dat
```

Once the preceding steps to configure the components of the project are done, select Exit and Save to preserve the configuration for this project.

Build

The menuconfig target creates a `.config` file, which coreboot uses to build the correct options for a given mainboard.

Let's build the project now:

```
~/fsp_coreboot/coreboot$ make
#
# configuration written to .config
#
   HOSTCC     nvramtool/cli/nvramtool.o
   HOSTCC     nvramtool/cli/opts.o
   HOSTCC     nvramtool/cmos_lowlevel.

...<SNIP>...
```

```
   CBFS       coreboot.rom
   PAYLOAD    build/seabios/out/bios.bin.elf (compression: LZMA)
   CONFIG     .config
   CBFSPRINT  coreboot.rom

coreboot.rom: 2048 kB, bootblocksize 1024, romsize 2097152, offset 0x0
alignment: 64 bytes

Name                           Offset     Type         Size
cmos_layout.bin                0x500000   cmos_layout  1352
pci8086,0f31.rom               0x500580   optionrom    65536
fallback/romstage              0x5105c0   stage        30444
fallback/ramstage              0x517d00   stage        65969
fallback/payload               0x527f00   payload      55583
config                         0x535880   raw          4321
revision                       0x5369c0   raw          693
(empty)                        0x536cc0   null         1938200
cpu_microcode_blob.bin         0x710000   microcode    156736
(empty)                        0x736480   null         105240
mrc.cache                      0x74ffc0   (unknown)    65536
(empty)                        0x760000   null         393112
fsp.bin                        0x7bffc0   (unknown)    229376
(empty)                        0x7f8000   null         31640
```

The build has completed successfully and the ROM image is here:

```
~/fsp_coreboot/coreboot/build/coreboot.rom
```

Summary of Commands

Here are the commands we have used so far to get a platform project configured and built:

```
$ mkdir fsp_coreboot
$ cd fsp_coreboot/
$ tar -xzvf ~/Downloads/BAY_TRAIL_FSP_KIT_GOLD3.tgz
$ git clone http://review.coreboot.org/coreboot
$ cd coreboot
$ ls
$ make crossgcc-i386
$ make gitconfig
$ git checkout -b fsp_dev cf52f9
$ make menuconfig
$ make
```

Flashing the ROM

flashrom is a utility for programming flash chips. It is one of the projects that has spun-off from the coreboot community. It is designed to program any type of firmware binary image (not only coreboot) onto a mainboard or other controller cards. It supports programming many flash devices in the system, including parallel, LPC, FWH, and SPI devices. It also supports many external programmers, including the commonly used Dediprog SF100 and BusPirate. It has common interface support for FT2232 and serprog-based devices. It is built for support on most operating systems.

Please check http://flashrom.org for more information.

■ **Note** It is strongly recommended that you have an external programmer for firmware development. At some point, you will "brick" your system and need to reflash the device.

For this example, we'll use the Dediprog SF100 to program the mainboard. Please see the mainboard user guide for additional programming requirements. The system may need to be powered on, powered off, or have a jumper set before you can program.

Preparing the Flash Programmer

The following are the steps to program System BIOS by using the Dediprog SPI flash programmer:

1. Power-off the board.
2. Port for BIOS flash update is J1 (MinnowBoard MAX).
3. No jumper settings.
4. Config the Dediprog voltage to 1.8V.
5. Program the device (W25Q64DW).

Save the entire existing flash image, just in case.

```
~/fsp_coreboot/coreboot$ flashrom -p dediprog -r backup.rom
flashrom v0.9.7-r1764 on Linux 3.11.0-20-generic (x86_64)
flashrom is free software, get the source code at http://www.flashrom.org

Calibrating delay loop... OK.
```

Flashing the ROM Image

The total coreboot ROM image is the same size as the SPI flash device—8MB. coreboot is not the only code in the SPI flash device and it may only use the BIOS section. For MinnowMax, the BIOS section is 3MB; the flash descriptor and the TXE binary are the other 5MB. We will discuss the descriptor and other binaries later in this chapter.

To update the flash with flashrom, we need to do the following:

1. Create an XML file with flashrom instructions.
2. Flash the device with the correct parameters (the MinnowMax flash device requires 1.8 volts from the Dediprog).
3. Create the XML instructions for flashrom:

```
~/fsp_coreboot/coreboot$ echo 00500000:007fffff cb > 8mb.xml
```

4. Write the image you have built to the BIOS region:

```
~/fsp_coreboot/coreboot$ sudo flashrom -p dediprog -l 8mb.xml -i
cb -w build/coreboot.rom
flashrom v0.9.7-r1764 on Linux 3.11.0-20-generic (x86_64)
flashrom is free software, get the source code at http://www.
flashrom.org

Calibrating delay loop... OK.
```

■ **Warning** You cannot program the entire flash with the coreboot image. There are other binaries located on the flash that are required to boot the system. Overwriting these files is bad. (You backed up the entire flash image as described earlier, right?)

Remove the Dediprog, replace the programming jumpers, and power up the system. The system should boot. If not, check out the "Troubleshooting and Debugging" section.

coreboot Internals

Now that you have a booting FSP coreboot MinnowMax, we can dig into the internals of coreboot. This section discusses what happens in the coreboot image during boot. We also cover how it is organized, the source tree, and the boot process.

Boot Stages

coreboot is made up of four boot stages. Each stage is a binary within the ROM image. From power-on, coreboot transitions from one binary stage to the next in the order shown in Table 4-1.

Table 4-1. *coreboot Boot Stages*

Stage	Description
bootblock	The reset vector and pre cache-as-RAM setup
romstage	Cache-as-RAM setup, early silicon initialization, memory setup
ramstage	Normal device setup and mainboard configuration
payload	The OS or application bootloader

Additional Files

The stage binaries require supporting files. These additional files are part of the coreboot image and critical for system functionality (see Table 4-2).

Table 4-2. *coreboot Supporting Files*

File Name	Description
fsp.bin	The FSP binary file.
pci8086,0166.rom	The video BIOS file; the name associates the binary to the PCI ID of the graphics device.
cmos_layout.bin	A map of the CMOS values used by coreboot. This file may be used by payloads or other utilities to safely manipulate CMOS.
config	The build options in the `.config` file are saved in the ROM image. This makes it possible to reproduce the image with the same options in the future.
mrc.cache	For saved memory configuration data. (More on this later.)

■ **Note** These are the file names in CBFS. They may be different than the menuconfig input path and file name.

CBFS

The coreboot stages and binaries require some organization in order to be found and loaded. This is accomplished in coreboot within CBFS, which is a scheme for managing independent binaries within a single firmware ROM image. Though not a true file system, the style and concepts are similar. CBFS binary headers contain information to help

identify the binary by type, such as stage, optionROM, and payload, and indicate if the binary is compressed. It is important to understand that each file in the CBFS is compiled separately. These binaries are not linked and each file is located, loaded, uncompressed, and executed as required.

■ **Note** Please visit http://www.coreboot.org/CBFS.

An Example of CBFS

At the end of the preceding coreboot build, the contents of the `coreboot.rom` file are printed out. We can check it again using the cbfstool:

```
~/fsp_coreboot/coreboot$ ./build/cbfstool ./build/coreboot.rom print
coreboot.rom: 2048 kB, bootblocksize 1024, romsize 2097152, offset 0x0
alignment: 64 bytes
Name                           Offset     Type         Size
cmos_layout.bin                0x0        cmos_layout  1132
pci8086,0f31.rom               0x4c0      optionrom    65536
fallback/romstage              0x10500    stage        27029
fallback/ramstage              0x16f00    stage        58969
fallback/payload               0x255c0    payload      59940
config                         0x34040    raw          4221
(empty)                        0x35100    null         896728
cpu_microcode_blob.bin         0x110000   microcode    52224
(empty)                        0x11cc40   null         209752
mrc.cache                      0x14ffc0   (unknown)    65536
(empty)                        0x160000   null         393112
fsp.bin                        0x1bffc0   (unknown)    229376
(empty)                        0x1f8000   null         31640
```

There are a couple things to note about the CBFS output.

You can find that all the stages are listed except for the bootblock. The bootblock stage is a mandatory piece and handled as a special case. It is located in the last 20K of the ROM space with the reset vector. It contains the location of the master header and the entry point for the loader firmware. It doesn't have a CBFS header due to its location at the end and how it is accessed, via a direct jump from the reset vector.

■ **Note** This may change in the future as ARM and other support are added, and which have different reset requirements for the reset vector and bootblock.

CBFS can have a directory-like structure; for example, `fallback/romstage` and `fallback/ramstage`. This is useful for grouping files that should be used together or for a specific boot purpose. In the preceding example, `fallback/` is the default boot path in coreboot. An additional set of binaries could be added for an alternate boot path that would be selected by the bootblock. The SeaBIOS payload also uses the directory structure for coreboot options.

■ **Note** For more information about SeaBIOS, please visit `http://www.coreboot.org/SeaBIOS#SeaBIOS_and_CBFS`.

CBFS Size

The size of the coreboot.rom file is not required to be the size of the flash device. It only needs to be large enough to fit the required files within CBFS. This leaves room on the flash device for files that are not part of coreboot. On a FSP-based system, the `coreboot.rom` file should be the same size as the BIOS descriptor region indicated by the flash descriptor. The `coreboot.rom` must be located at the end of the flash device to execute the reset vector.

Special Binaries

In addition to Intel FSP and microcode, there are some important binaries located on the flash device that are not part of coreboot. This was briefly described in the flash and boot section of this chapter. These files are required for proper system operation, so it is important that they are not overwritten with coreboot (see Table 4-3).

Table 4-3. *Special Binaries for coreboot*

Binary	Description
descriptor.bin	The Intel Firmware Descriptor describes the content of the flash device. This includes the locations of the binaries, which areas are write protected, and bootstrap options.
TXE/ME	Trusted Execution Engine(TXE) or Management Engine (ME) binaries. These binaries are run by the security and management processor prior to starting the CPU.
GigEthernet	Intel integrated Ethernet binary. This is not a PXE option ROM, but device firmware.

■ **Note** The descriptor and other binaries can be queried by the coreboot `utils/ifdtool`.

Boot Flow Using Intel FSP

As mentioned earlier, each stage is called consecutively after the other. In this section, we will follow the flow from the reset vector to loading a payload.

Reset Vector and Bootblock

On x86 systems, there is a lot of legacy cruft, which makes for some tedious details that must be dealt with by early boot firmware. To start with, the very first instruction executed by an x86 CPU is in 16-bit reset mode (sort of like real mode, but with 4GB selectors loaded as default); the first instruction is fetched and executed by the CPU at memory location FFFFFFF0, in hexadecimal value, 16 bytes below 4GB of the 32-bit architecture's addressing limit. There's a lot of history behind this design; therefore, we won't go into more detail in this book.

coreboot's reset vector contains a single jump instruction to the 16-bit entry code of the bootblock. coreboot then transitions immediately to 32-bit flat protected mode. This switch makes it much easier to use the 32-bit registers and to access the entire 4GB memory space.

The reset vector and bootblock code is run directly from ROM, doing what is called "execute in place" (XIP). The first few instructions are written in assembly code. As discussed in the preceding history section, assembly code is difficult to read and debug, so coreboot starts using C code within a few hundred instructions. This is accomplished by using a special compiler/assembler called ROMCC. ROMCC translates C code to a stackless assembly `.inc` file that is then compiled and linked by the assembler/linker. It must be stackless because there is no memory for stack at this point in the boot process, and normal C compilers assume memory and use the stack to pass variables.

The early C code in bootblock has a few basic functions. If required by the system, it can do very early silicon setup. For example, routing the Port 80h debug output, enabling the chipset flash features, or checking a signal to indicate which stage should be loaded next. The bootblock parses CBFS, locates the romstage, and jumps to its starting point.

romstage

The early part of romstage is very similar to the bootblock. It is execute in place (XIP) code written in assembly. The only difference is that coreboot is already in 32 bit protected flat mode. There is no system memory available, so the first step in romstage is to set up "Cache as RAM" (CAR). This allows coreboot to use the CPU cache as system RAM for a stack location. The FSP handles the CAR setup and has some very specific requirements to run. This is fully explained in the Intel FSP chapter, but we will do a quick review.

To call the first Intel FSP entry, coreboot contains a stack area that contains a pointer to the Intel FSP parameter structure and the return address to get back to coreboot when Intel FSP is finished. The parameter structure contains the microcode address and length, and start address and length of the ROM area that should be cached. With the stack pointer prepared, coreboot locates the FSP, verifies that the FSP headers are as expected for the platform and jumps to the FSP TempRamInit API entry point. The FSP executes, works its magic, sets up CAR, and returns to coreboot. coreboot sets the stack pointer and makes the first C-style call to do romstage system setup and memory setup.

Most x86 systems require a significant amount of setup to configure the hardware. This is even more the case in integrated silicon and System on a Chip (SoC) systems. Most integrated subsystem devices require additional configuration prior to being accessed in the normal methods (PCI Configuration Space, Memory Mapped I/O, System I/O, etc.). Romstage is where the few devices required for memory initialization are configured. It is also the first change to get additional debug information from the system. With most Intel FSP based systems, including MinnowMax, the serial port is configured and debug information can be streamed to the developer.

With a little bit of mainboard specific hardware initialized, coreboot is almost ready to make the second call into Intel FSP for memory initialization and the initial setup of the various peripherals. In order to do this, coreboot locates the UPD/VPD structures as discussed in Chapter 3. After getting the UPD/VPD data, coreboot modifies these based on mainboard specific configuration data from devicetree.cb. This allows coreboot to inform Intel FSP which devices should be enabled or disabled and what mode the devices should be configured in. The FspInit entry sets up the memory and disables CAR before it returns to the coreboot's return function. Intel FSP also passes back a Hand-Off Block (HOB), which contains data Intel FSP and coreboot may use later. coreboot saves the HOB data location and prepares for ramstage. The romstage code locates the ramstage in CBFS, copies it to memory and jumps to the entry point.

ramstage

Ramstage is a bare-metal application. The CPU and memory are functional and ramstage is running from memory with a normal stack and can use heap, global variables, and so forth. The purpose of ramstage is to configure the I/O devices, additional application processors, SMM, and to set up tables that may be passed to payloads or operating systems.

The heart of ramstage is a state machine running in the hardwaremain function and the device tree. The state machine states are defined by the standard stages of PCI device configuration and enumeration. There are additional states for chip and mainboard configuration to allow customization of device prior to the normal initialization process. The state machine also has pre and post hooks at each state, so chipset and mainboards can be customized as needed. The states and state machine are explained in detail later in this chapter.

The device tree is the hierarchical structure of the PCI and legacy devices in the system. The device tree is prepopulated at build time through the entries in the mainboard's devicetree.cb file and amended runtime as devices are discovered in the PCI enumeration process. The device tree structure has function pointers for every device for each state in the state machine. This allows chipset and onboard devices to have customer driver functions run during the enumeration process. We will discuss the specific of the state machine and device tree later in this chapter.

There are two calls to the FspNotifyPhase entry point in ramstage, AfterPCIEnumeration and ReadyToBoot. After all the devices are enumerated, the coreboot calls FspNotifyPhase(AfterPCIEnumeration). coreboot then sets up SMM, does legacy table setup, and finally ACPI table setup. The final call to Intel FSP is made, FspNotifyPhase(ReadyToBoot), where the lock registers are set to protect SMM and other sensitive registers. Then, ramstage locates the primary payload in CBFS, decompresses it to memory, and executes it.

Payload

The last part of coreboot is to execute a payload. The payload functions and features are not defined by coreboot. A payload could be a bare-metal application or it could boot an operating system. There may be more than one payload in a coreboot image. Some common payload options are discussed later in the chapter.

coreboot Source

coreboot contains initialization code for several different architectures, many different silicon devices, and hundreds of mainboards. This can be overwhelming for new coreboot developers, so we will highlight the areas of focus for coreboot FSP-based mainboards. Again, we focus on the MinnowMax mainboard.

coreboot Device Tree

Each device supported by coreboot has a corresponding driver. In order to associate the hardware to the driver, coreboot describes the onboard devices in the coreboot device tree. The mapping of devices to their custom functions is done in the mainboard devicetree.cb file. The devicetree.cb is evaluated during the build process by the sconfig tool (`coreboot/util/sconfig`), which creates a linked list of devices in the `build/mainboard/VENDOR/BOARD/static.c` file. During the boot process, the coreboot scans the devices, adds any found devices to the device tree, and links the drivers to the devices found. The device tree is an integral part of the coreboot build and boot process. The device tree code is located in the coreboot device tree source directory at `coreboot/src/device`.

The device tree has two root busses, the CPU bus and the PCI bus. The start of the device tree is called the *root complex*, which links the top level CPU bus and PCI bus 0. The CPU bus contains systems local APICs (Advanced Programmable Interrupt Controllers). PCI bus 0 contains all other system devices, including legacy and IO devices.

■ **Note** The coreboot device tree is not a Flattened Device Tree used by Linux ARM kernels.

Chips and Devices

The coreboot device tree has chip and device functions. A chip may be made up of one or more devices. Some chips require configuration prior to the device configuration. This is very common on southbridge devices. To accommodate the predevice setup, the chip functions are called prior to device functions. We will cover this in more detail in the section covering coreboot hardware state machines.

Device Tree Variables

Each device tree section starts with the variable name (see Table 4-4) and is closed with the `'end'` keyword.

Table 4-4. *coreboot Device Tree Variables*

Variable Name	Description
chip	Path to the chip source. The chip variable comes prior to all devices in the device tree. The path also corresponds with a chip_operations structure.
device	Defines a device type at the indicated address.
register	Is used to pass mainboard customization to generic chip code as defined in its `chip.h`. This is different than a Kconfig build option.

Each device type has its own set of function pointers, as listed in Table 4-5.

Table 4-5. *coreboot Device Types*

Device Type	Description
domain	Sets the PCI bus number. All PCI devices must be within a domain keyword. Only bus 0 must be set up in a system, leaving all other busses to be configured using the default configuration.
cpu_cluster	Specifies the top-level APIC and the CPU root cluster.
pci	Devices with PCI configuration space.
i2c	Sets the 7-bit I2C address of a device on an I2C bus. This keyword must be within a PCI I2C/SMBUS controller device.
pnp	Devices in the legacy (ISA) memory and I/O range (e.g., SuperIOs).
ioapic	The ID of a chipset's IO APIC. A default configuration is used if this is not set in the device tree.
lapic	The ID of a CPU's Local APIC. One lapic is required in the device tree.

There are additional keywords used in the device tree, which are listed in Table 4-6.

Table 4-6. *coreboot Additional Keywords Used in the Device Tree*

Keyword	Description
subsystemid	Sets the PCI config register subsystem device and vendor IDs. This may be set at the top level and inherited, or within a specific device. See inherit.
inherit	Sets a value for all the devices after it. Used for subsystem ID.
io	Sets an IO register value for a pnp device.
irq	Sets an IRQ line for a pnp device.
drq	Sets a DRQ line for a pnp device.
ioapic_irq	Is used to generate mptable from the devicetree.cb.
on	Sets a device state to enabled.
off	Sets a device state to disabled (may hide device on some chipsets).
end	Closes a block.

A Device Tree Example

The following example is at `coreboot/src/mainboard/intel/minoxmax/devicetree.cb`.

```
chip soc/intel/fsp_baytrail
#### ACPI Register Settings ####
register "fadt_pm_profile"         = "PM_UNSPECIFIED"
register "fadt_boot_arch"          = "ACPI_FADT_LEGACY_FREE"

#### FSP register settings ####
register "PcdSataMode"             = "SATA_MODE_AHCI"
register "PcdMrcInitSPDAddr1"      = "SPD_ADDR_DEFAULT"
register "PcdMrcInitSPDAddr2"      = "SPD_ADDR_DEFAULT"
register "PcdMrcInitMmioSize"      = "MMIO_SIZE_DEFAULT"
register "PcdeMMCBootMode"         = "EMMC_FOLLOWS_DEVICETREE"
register "PcdIgdDvmt50PreAlloc"    = "IGD_MEMSIZE_DEFAULT"
register "PcdApertureSize"         = "APERTURE_SIZE_DEFAULT"
register "PcdGttSize"              = "GTT_SIZE_DEFAULT"
register "PcdLpssSioEnablePciMode" = "LPSS_PCI_MODE_DEFAULT"
register "AzaliaAutoEnable"        = "AZALIA_FOLLOWS_DEVICETREE"
register "LpeAcpiModeEnable"       = "LPE_ACPI_MODE_DISABLED"
register "IgdRenderStandby"        = "IGD_RENDER_STANDBY_ENABLE"
register "EnableMemoryDown"        = "MEMORY_DOWN_ENABLE"
register "DRAMSpeed"               = "DRAM_SPEED_1066MHZ"
```

```
register "DRAMType"                = "DRAM_TYPE_DDR3L"
register "DIMM0Enable"             = "DIMM0_ENABLE"
register "DIMM1Enable"             = "DIMM1_DISABLE"
register "DIMMDWidth"              = "DIMM_DWIDTH_X16"
register "DIMMDensity"             = "DIMM_DENSITY_2G_BIT" # Setting for 1GB
board - modified runtime for 2GB board in romstage.c to DIMM_DENSITY_4G_BIT
register "DIMMBusWidth"            = "DIMM_BUS_WIDTH_64BIT"
register "DIMMSides"               = "DIMM_SIDES_1RANK"
register "DIMMtCL"                 = "11"
register "DIMMtRPtRCD"             = "11"
register "DIMMtWR"                 = "12"
register "DIMMtWTR"                = "6"
register "DIMMtRRD"                = "6"
register "DIMMtRTP"                = "6"
register "DIMMtFAW"                = "20"

device cpu_cluster 0 on
device lapic 0 on end
end

device domain 0 on
device pci 00.0 on end # 8086 0F00 - SoC router -
device pci 02.0 on end # 8086 0F31 - GFX micro HDMI
device pci 03.0 off end # 8086 0F38 - MIPI -

device pci 10.0 off end # 8086 0F14 - EMMC Port -
device pci 11.0 off end # 8086 0F15 - SDIO Port -
device pci 12.0 on end # 8086 0F16 - SD Port MicroSD on SD3
device pci 13.0 on end # 8086 0F23 - SATA AHCI Onboard & HSEC
device pci 14.0 on end # 8086 0F35 - USB XHCI - Onboard & HSEC  - Enabling
both EHCI and XHCI will default to EHCI if not changed at runtime
device pci 15.0 on end # 8086 0F28 - LP Engine Audio LSEC
device pci 17.0 off end # 8086 0F50 - MMC Port -
device pci 18.0 on end # 8086 0F40 - SIO - DMA -
device pci 18.1 off end # 8086 0F41 -   I2C Port 1 (0) -
device pci 18.2 on end # 8086 0F42 -   I2C Port 2 (1) - (testpoints)
device pci 18.3 off end # 8086 0F43 -   I2C Port 3 (2) -
device pci 18.4 off end # 8086 0F44 -   I2C Port 4 (3) -
device pci 18.5 off end # 8086 0F45 -   I2C Port 5 (4) -
device pci 18.6 on end # 8086 0F46 -   I2C Port 6 (5) LSEC
device pci 18.7 on end # 8086 0F47 -   I2C Port 7 (6) HSEC
device pci 1a.0 on end # 8086 0F18 - TXE -
device pci 1b.0 off end # 8086 0F04 - HD Audio -
device pci 1c.0 on end # 8086 0F48 - PCIe Port 1 (0) -
device pci 1c.1 off end # 8086 0F4A - PCIe Port 2 (1) -
device pci 1c.2 on end # 8086 0F4C - PCIe Port 3 (2) Onboard GBE
device pci 1c.3 on end # 8086 0F4E - PCIe Port 4 (3) HSEC
```

```
device pci 1d.0 on end # 8086 0F34 - USB EHCI - Enabling both EHCI and XHCI
will default to EHCI if not changed at runtime
device pci 1e.0 on end # 8086 0F06 - SIO - DMA -
device pci 1e.1 on end # 8086 0F08 -   PWM 1 LSEC
device pci 1e.2 on end # 8086 0F09 -   PWM 2 LSEC
device pci 1e.3 on end # 8086 0F0A -   HSUART 1 LSEC
device pci 1e.4 on end # 8086 0F0C -   HSUART 2 LSEC
device pci 1e.5 on end # 8086 0F0E -   SPI LSEC
device pci 1f.0 on end # 8086 0F1C - LPC bridge No connector
device pci 1f.3 on end # 8086 0F12 - SMBus 0 SPC
end
end
```

■ **Note** These are not the only PCI devices in the system, but they are the only ones that require drivers. Devices may be added to slots and use the standard device initialization functions.

Chip Operations

The chip operations structure contains pointers to a function to initialize the chip and to enable a device, as well as a finalize function and a chip name string. The device enable function is called prior to the device operations (see Table 4-7). This is particularly important for devices that need to enable PCI devices before the initial scan and initialization. For example, some chipsets require additional setup for each device to be visible on the PCI bus.

Table 4-7. *coreboot Chip Functions*

Chip Function	Description
init	Chip initialization function.
enable_dev	The function called for each chip in the device tree.
final	The final function for each chip in the device tree. The last function before payload loading.

Device Operations

During the coreboot initialization process, each device operations function is run on the device in the order that it is scanned. Any device operation function pointer can be set to point to a custom device function. The device operations structure contains the function pointers listed in Table 4-8.

Table 4- 8. *coreboot Device Operations*

Device Operation	Description
read_resources	Read and save the device resources to be arranged and assigned.
set_resources	Assigned memory and IO space.
enable_resources	Enable memory and IO in the PCI command register.
init	Load the PCI device option ROMs.
finalize	Perform any final cleanup.
scan_bus	Bus or bridge devices scan and enable function.
enable	Activate the device (very late function call; not normally used).
disable	Deactivate the device, turning it off (very late function call; not normally used).
ops_pci	Sets the devices default operation functions.

Set the function pointer to NULL to skip the function for the device; otherwise, the default device function is used.

coreboot Hardwaremain State Machine

At the heart of the coreboot ramstage is a state machine for enumerating mainboard devices. coreboot starts device enumeration with the top-level device in the device tree and begins a bus scan. PCI devices that do not require special setup are added to the device tree as they are found during the scan, and are set up by the default PCI configuration functions. PCI devices that require special setup are linked with their custom drivers in the initial scan. Then, the state machine enumerates each PCI device's functions in five stages: read_resource, set_resource, enable_resource, init, and enable (see the "Device Operations" section). At each state, custom device functions can be called. The coreboot hardwaremain state machine source is `coreboot/src/lib/hardwaremain.c`.

State Machine States

Table 4-9 lists the state machine states used in coreboot.

Table 4-9. *coreboot State Machine States*

State	Descrtiption
BS_PRE_DEVICE	Before any device tree actions take place
BS_DEV_INIT_CHIPS	Init all chips in device tree
BS_DEV_ENUMERATE	Device tree probing
BS_DEV_RESOURCES	Device tree resource allocation and assignment
BS_DEV_ENABLE	Device tree enabling/disabling of devices
BS_DEV_INIT	Device tree device initialization
BS_POST_DEVICE	All device tree actions performed
BS_OS_RESUME_CHECK	Check for OS resume vector
BS_OS_RESUME	Resume to OS
BS_WRITE_TABLES	Write coreboot tables
BS_PAYLOAD_LOAD	Load payload into memory
BS_PAYLOAD_BOOT	Boot to payload

State Machine Callbacks

Each state has an Entry Callback and an Exit Callback, which may be used by any coreboot code to hook any state; for example, the Bay Trail FSP mrc.cache is saved during the table write state, after all devices have been setup.

```
Enter State  ->  Entry Callback  ->  Execute State  ->  Exit Callback  ->
Next State
```

■ **Note** Do not use multiple hooks to the same state callback. The order in which multiple hooks to the same state's callback are executed is undetermined.

Mainboard

The coreboot mainboard directory is the primary location that new mainboard developers will begin working in. It is located in the mainboard vendor directory and contains the files that make one mainboard unique from another (see Table 4-10). coreboot is architected to share as much common code as possible. The mainboard files access the CPU's, the chipset's, and the device driver's common code to do the majority of the work. Let's review the contents of the MinnowMax directory and break down the purposes of these key files.

Table 4-10. *coreboot Mainboard Files*

File Name	Description
acpi_tables.c	Functions that patch the DSDT and other ACPI table runtime.
cmos.layout	CMOS entries used by the mainboard.
devicetree.cb	Prepopulate mainboard chips and devices used to configure and enable and disable certain device options.
dsdt.asl	The mainboard ACPI ASL file.
fadt.c	Generates and checksums the ACPI FADT file.
gpio.c	Sets the default configuration for the mainboards GPIOs. GPIO configuration is fairly complex on Bay Trail and there are a lot of options to set up.
irqroute.c	Required to compile the IRQ macros defined in IRQ.h.
irqroute.h	Macros for each device IRQ routing in APIC and PIC modes.
Kconfig	Selects the default build options for CPU-, chipset-, and mainboard-specific options.
mainboard.c	The mainboard-specific file called in ramstage.
mainboard_smi.c	The mainboard-specific SMI calls.
Makefile.inc	Required to build the mainboard directory.
onboard.h	Mainboard-specific SMBIOS table settings.
romstage.c	The mainboard-specific function for romstage.
thermal.h	Critical temperature definitions for ACPI.
acpi/ ec.asl mainboard.asl superio.asl video.asl	Contains mainboard-specific ACPI ASL files that are included by the chipset ASL files.

The directory is `coreboot/src/mainboard/intel/<mainboard>/`.

It is easiest to begin working on a new mainboard using the reference design. It will already have the basic calls to the chipset and other devices.

The Chipset Driver

When the coreboot device enumeration finds a new device, it checks for a custom driver to set up the device. For Bay Trail, the basic setup is handled by the romstage and ramstage files located in the SoC directory. When Intel FSP access is required, the chipset code and the Intel FSP driver cooperate to send the correct information for the chipset-specific Intel FSP.

The Bay Trail FSP source files are at `coreboot/src/soc/intel/fsp_baytrail`. Key files are listed in Table 4-11.

Table 4-11. *Key Chipset Files Under coreboot*

File Name	Description
northcluster.c	Memory and PCIe resource allocation
southcluster.c	I/O device resource allocation
ramstage.c	
romstage/romstage.c	FSP `early_init()` call and return point
chip.h	Bay Trail FSP variables, includes UPD options
fsp/chipset_fsp_util.c	

■ **Note** Bay Trail is an SoC, so it has northcluster and southcluster files within the `src/soc/` directory. A typical chipset pair would have their files in `src/northbridge/` and `src/southbridge/` directories.

Chipset FSP UPD Options

The chipset UPD options in Intel FSP are defined in chip.h and set in the mainboard-specific devicetree.cb. See the section discussing UPD in Chapter 3 for more details on the options that are passed.

The FSP Driver

The coreboot FSP driver handles standard access functions to Intel FSP. While the access functions are standardized per the API, each chipset and mainboard may have custom FSP requirements, capabilities, and options. Chipset-specific options such as configuring the UPD data are handled by calls from the driver back to the chipset's FSP files. The mainboard-specific configuration is set in the devicetree.cb file, and then can

be customized further during the romstage callback, as previously mentioned. The FSP driver is based on the reference code provided in Intel FSP documentation, but resides in coreboot. The driver runs in both romstage and ramstage. The first FSP API call to TempRAMInit is part of the normal driver code, but is included in early romstage, cache_as_ram.inc.

The FSP driver source directory is located at `coreboot/src/drivers/fsp`.

Table 4-12 lists the coreboot fsp_util functions.

Table 4-12. *coreboot Functions that Interface with Intel FSP*

Function Name	Description
find_fsp	Function to find the FSP in memory.
fsp_early_init	FSP memory and early device setup function. Called in romstage by the chipset driver.
romstage_fsp_rt_buffer_callback	Callback from fsp_early_init for mainboard-specific RT buffer customizations (soldered down memory timings, etc.).
FspNotify	There are two notify calls in ramstage. AfterPCIEnumeration during device finalize and ReadyToBoot during chip finalize.
save_mrc_data	Called in romstage after fsp_early_init to save the memory configuration to CBMEMh.
update_mrc_cache	Moves the mrc data from CBMEM to NVRAM in late ramstage.

Kconfig

coreboot uses the Linux build configuration tool, Kconfig, to select build options. Kconfig files are in nearly all coreboot source directories. The Kconfig options are used by the makefiles to include the correct source files. In the preceding coreboot mainboard build section, you used the Kconfig Text User Interface—menuconfig—to select options for your example coreboot build. Typically, there are options for the mainboard, chipsets, debugging, and which payload to include in the coreboot.rom image file. The Kconfig options are saved as `.config` file and converted to a `config.h` for definitions to be used by the coreboot source code. The file is also saved in the `coreboot.rom` image, where it can be extracted and used to build with the same coreboot options.

The Kconfig tool is built by the coreboot make process and is located here: `coreboot/util/kconfig`

xcompile

The coreboot make process needs to locate a compatible toolchain. This is done by the xcompile script. On each build, the coreboot makefile checks for the `.xcompile` file, which is generated by the `utils/xcompile/xcompile` script, and if it is not found, the makefile calls the script to generate it. The xcompile script locates the coreboot toolchain and copies the path into the `.xcompile` file. The generated `.xcompile` file is included in the make to set variables `CC`, `CFLAGS`, `CPP`, `AS`, `LD`, `NM`, `OBJCOPY`, `OBJDUMP`, `READELF`, `STRIP`, `AR`.

■ **Warning** The `.xcompile` file isn't built on every make. If the file already exists, the script will not be re-run. This is a problem if you didn't have the toolchain built previously and the `.xcompile` is empty. Without a "make clean," the old path to the distribution toolchain is used.

Payloads

A payload may be any ELF binary. It must be able to execute on bare metal and without any support services. Payloads are typically separate projects from coreboot and have their own development community (although there is some obvious overlap with coreboot developers). As a separate project and binary, payloads may have a different license than coreboot. The cbfstool supports converting the ELF format to the SELF format, which can be loaded by coreboot. SeaBIOS is the default payload, but any ELF may be added in the Payload section of the menuconfig.

See `http://www.coreboot.org/Payloads` and `http://www.coreboot.org/SELF` for more information.

There are several Payloads available for you to choose from.

SeaBIOS

SeaBIOS provides the legacy BIOS services for booting most operating systems. The coreboot build process makes it easy to use SeaBIOS by downloading and building it if it is selected. SeaBIOS supports booting from SATA and USB. It also supports loading Option ROMs and additional payloads. SeaBIOS runtime options, like boot order, are added to configuration files in CBFS.

SeaBIOS has been tested with Linux, NetBSD, OpenBSD, FreeDOS, and Windows XP/Vista/7. Classic GRUB, GRUB2, lilo, and isolinux work well with SeaBIOS. Other x86 bootloaders and operating systems will likely also work.

The SeaBIOS development license uses GPLv2+.

See `http://www.coreboot.org/SeaBIOS` and `http://www.seabios.org/SeaBIOS` for more information.

GRUB 2

You can use GRUB2 as a coreboot payload to boot an operating system from a hard drive, for instance. You can also boot via an existing GRUB2 MBR on your hard drive by using SeaBIOS as your coreboot payload.

The GRUB2 development license uses GPLv3. See http://www.coreboot.org/GRUB2 and https://www.gnu.org/software/grub/grub.html for more information.

FILO

FILO is a simple bootloader with filesystem support. It can boot from hard drives and USB mass storage. It does not require any legacy BIOS callbacks.

The FILO development license uses GPLv2. See http://www.coreboot.org/FILO for more information.

iPXE

iPXE is a network bootloader and is a fork of GPXE/Etherboot. It provides a direct replacement for proprietary PXE ROMs. It can be run as a payload or as an OptionROM by SeaBIOS.

The iPXE development license uses GPL v2+.

See http://www.coreboot.org/IPXE and http://ipxe.org/ for more information.

TianoCore

There is limited porting and support work in the community for TianoCore, a bootloader providing the UEFI interface.

The TianoCore development license uses BSD.

See http://www.coreboot.org/TianoCore for more information.

Depthcharge

Depthcharge is a payload for the Google Chromebooks.

The Depthcharge development license uses GPLv2 (or later).

See https://chromium.googlesource.com/chromiumos/platform/depthcharge for more information.

U-Boot

The U-Boot bootloader can be configured as a coreboot payload for Google Chromebooks.

The U-Boot development license uses GPLv2.

See http://www.denx.de/wiki/U-Boot for more information.

Memtest86+

Aimed at memory failures detection, this memory test is available as a coreboot payload as well.

See http://www.coreboot.org/Memtest86 and http://memtest.org/ for more information.

libpayload

Libpayload is a library to assist with developing and building custom payloads. It contains entry point, build options, and basic libc functions. libpayload is built separately from the developers' payload code and it is statically linked. libpayload may be built with a number of different options configured with a libpayload-specific Kconfig. See the FILO or TINT payloads for an example.

The libpayload development license uses BSD.

See http://www.coreboot.org/Libpayload for more information.

coreboot Troubleshooting and Debugging

There are lots of complicated parts to modern systems, and silicon initialization and development and testing don't always go smoothly. There are a number of troubleshooting and debugging options when debugging with coreboot.

Postcodes

The earliest debug information available from coreboot is postcodes on port 80h. Many CRBs have integrated postcode hardware to display this early debug information. coreboot's first instruction after the reset vector is an out 01h (POST_RESET_VECTOR_CORRECT) to port 80h. There are two common failures early in coreboot with FSP:

- postcode 00h: The system is on, but no there are no postcodes. This is usually a problem with the flash device. Check that the flash jumpers are correctly populated.

The other problem is that the flash image descriptor.bin and TXE/ME have been overwritten. Reflash the backup image and only update the last 2MB of the MinnowMax flash device with coreboot.

- postcode BBh: The system is alternating between BBh and one of the following postcodes:

```
00h - FSP_SUCCESS: Temp RAM was initialized successfully.
02h - FSP_INVALID_PARAMETER: Input parameters are invalid.
03h - FSP_UNSUPPORTED: The FSP calling conditions were not met.
07h - FSP_DEVICE_ERROR: Temp RAM initialization failed
0Eh - FSP_NOT_FOUND: No valid microcode was found in the
      microcode region.
14h - FSP_ALREADY_STARTED: Temp RAM initialization has been invoked
```

The most common failure is 0Eh, no valid microcode was found. Check that you are using the latest Intel microcode for your silicon version. The Intel FSP package may not have the latest version and you need to update it.

Serial Debug

Serial debug is the most common method of debug in coreboot. The serial port and console configuration is one of the earliest functions after CAR (TempRAMInit) is set up. coreboot can be configured to output different levels of information on the serial port. A typical development coreboot build defaults to DEBUG level, which outputs a lot of information for the developer. The level can be turned up to SPEW, which is way too much information, and it can also be turned down to ERROR or other lower settings to speed up the boot process by printing less information to the serial port. Intel FSP also supports serial output for sending debug information.

EHCI USB Debug

If a serial port is not available for normal debug, coreboot may set up the EHCI controller USB port 0 for debug mode. The EHCI debug port provides a special mode of operation that requires neither RAM nor a full USB stack. It requires additional hardware, like the Ajays NET20DC USB Debug Device, and drivers for the device for the target to send the logging information to. The debug mode is not yet supported by Intel FSP for debug information.

Summary

The coreboot firmware philosophy is about building up with small, fast, target-specific needs. The developers have created a framework to build on and do not make assumptions about the users' needs. Intel FSP and coreboot together allow system designers to customize their solutions down to the smallest details. We look forward to what the next generation of coreboot developers will bring.