coreboot boot structure
LA-UR-06-7928

New coreboot group

March 13, 2008

Abstract

This is the new coreboot boot architecture.

1 Introduction

The new coreboot boot architecture depends on CAR, with payloads appearing as files in a LAR archive. The device tree is defined by a device tree blob (DTB) and all the activities flow from that. For now, the DTC will produce a standard coreboot v2 device tree; this will, we hope, be improved. romcc is gone.

Required attributes of a CPU for coreboot v3:

- Supports CAR

Required platform attributes:

2 Goal

2.1 Design Goals

- All components are seperate modules.
- The strict seperation of normal/fallback does not exist anymore. Any module can be available several times.
- Commonly used code is shared. There is _one_ implementation of printk, and one implementation for each compressor.
- Under construction, things have changed recently.

2.2 Features

3 FLASH layout

Shown in 1 is the layout of the whole FLASH. Note that we can kill the buildrom tool, since the FLASH code is now a LAR archive. Note that the linker script will now be very simple. The initram is roughly what is in auto.c, although the early hardware setup from auto.c is now in the pre-initram, so that we have serial output and other capabilities. The FLASH recovery is interesting: in hopeless cases, the serial port can be used to load a new flash image, and allow a successful boot from a totally hosed machine. VPD includes data such as the MAC address, instance of the motherboard, etc. The DTB can be modified by the flashrom tool, and hence a platform can be customized from a binary FLASH image. Each LAR file has a checksum attached to the end, so that the system can verify that the data is uncorrupted. We now build at least four targets for a platform:

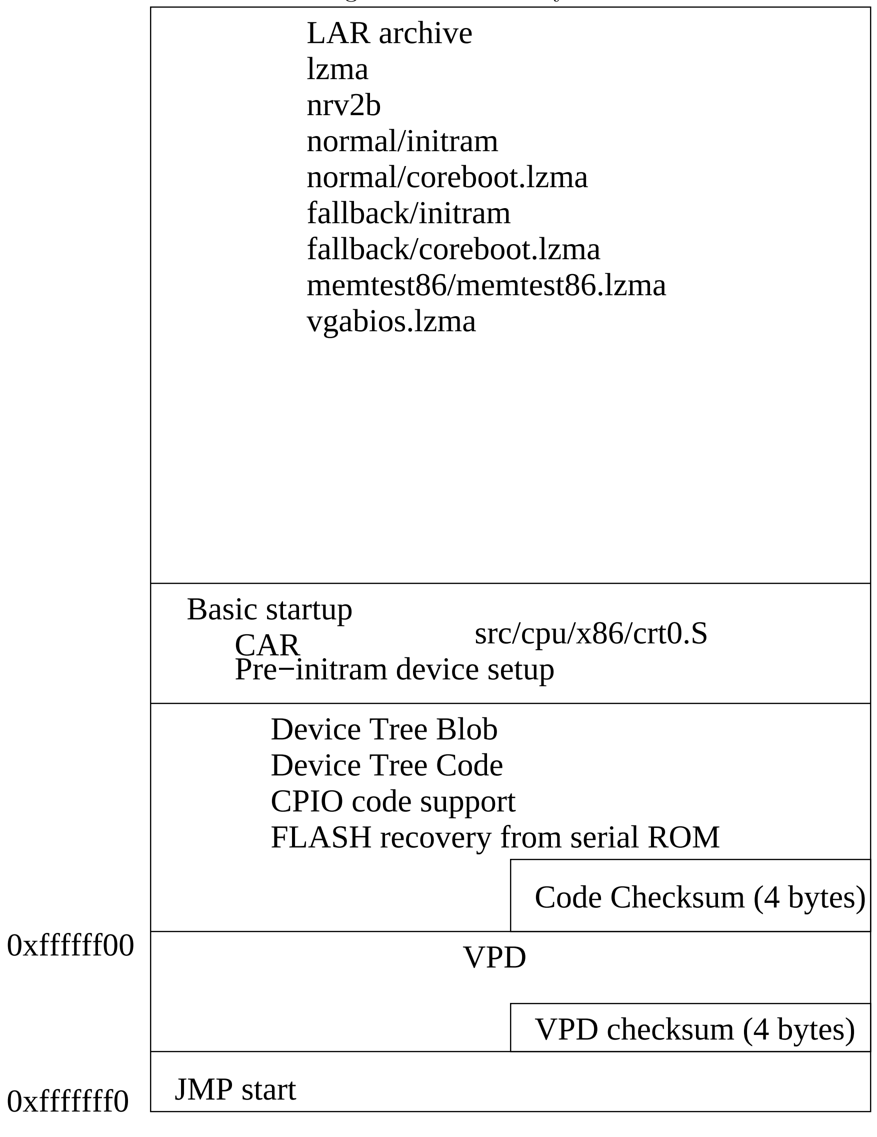
- Basic startup and CAR (in most cases, same for all processors of a given type)
- Pre-initram device setup (large FLASH, serial port, etc.)
- initram
- Traditional coreboot RAM code (LAR, etc.)
- Load payload and start it

4 Introduction to the coreboot device tree

4.1 Purpose and function

The coreboot device tree is probably the single most important concept in coreboot, and, in V2, was the least understood part of the software. The device tree provides a tremendous amount of capability to the software. The initial tree, which almost always will be an incomplete representation of the hardware (consider pluggable devices), is created by the configuration tool – in V3, the device tree compiler, or DTC. The initial tree is statically allocated at compile time. At runtime, hardware must be probed, and the tree must be filled in, extended, and even restructured as the hardware discovery process occurs. The restructuring mainly consists of adding nodes, and reordering nodes in the tree to accomodate these new nodes. The tree represents devices as nodes and busses as edges (called links) between the nodes. Some devices can bridge a bus to another; these are intermediate nodes in the tree. Other devices do not bridge busses; these are leaf nodes. And, of course, a bridge device might exist with nothing on the other side – this device will of course also be a leaf node.

Figure 1: FLASH layout



At build time, the programmer can specify, via a configuration file, hardware that is likely to be on the mainboard, although it is not required to be present (e.g. a 2-cpu system might have only one CPU installed). Also, hardware that can be probed, and that does not need any special configuration, can be left out of the configuration file and left to be discovered dynamically, even if it is known to be on the board. At run time, the software must determine what hardware exists, and modify the tree to accord to reality. This modification can include deletion of nodes, including bridge nodes; and even deletion of edges in the graph. The software must also be able to add new nodes and edges, as bridges and devices are found.

Finally, once the tree is built, the device tree software must allocate resources to each device. Resources, in this context, are for the most part address space ranges, in memory or I/O space. A given device will require a certain range of addresses and, still worse, might require that those addresses be fixed at a certain value (such as a superio which is hardwired to address 0x4e). The software must allocate the resources to devices, and, for a bridge, must allocate the resources to the bridge that are held by all the devices on the other side of the bridge.

The process of resource allocation must take into acount the fact that addresses at a bridge may have special alignment requirements (e.g. the 4096 byte alignment for memory address ranges); that memory addresses on a device must be aligned to the size of the region (e.g. a 512KiB region must be aligned to 512KiB); and other restrictions dependent on the device or bridge. Another issue is the use of *subtractive address ranges*, which define an address range that is picked up by default if no other resource claims it. *Subtractive address ranges* are typically used for legacy PC address ranges.

4.2 Device tree structures

There are three primary objects which are used to manage the coreboot device tree: devices, links, and drivers. Devices and links are generic structures: drivers, on the other hand, are specialized. We describe these parts, and their relationship, below.

These structures are linked together in the static device tree. The static device tree, a set of initialized C structures, is created by the device tree compiler, in the file build/statictree.c, using as input the dts file in the mainboard directory. This tree defines the hardware that is known to exist on the mainboard. At run time, the static tree is elided with dynamically determined information, and can even be restructured to some extent (e.g., the static tree has a device at 0:4.0; if a dynamic device is found at 0:3.0, it will be placed in the tree “in front of” the 0:4.0 device).

Each device has a void * which can be filled in, via the dts file, with non-generic, i.e., device-specific, information.

What’s the difference between generic and non-generic information? As an example, the PCI configuration address or “path” of a device is generic; there is a path of some sort for every device in a system. But, not all devices have iden-

Figure 2: The files in the i440bx directory.

```
[rminnich@q ~]$ ls ~/src/bios/coreboot-v3/northbridge/intel/i440bxemulation/
config.h i440bx.c i440bx.h Kconfig Makefile
```

tical capabilities. Some PCI devices have IDE controllers, others do not; some can drive the legacy PC XBUS, others can not; and so on. In coreboot V1, we attempted to create devices that were the union of all possible devices, but creating such a union proved to be impossible, as new devices with new capabilities came out. So, in V2, we split off the device-specific information into a seperate structure. The generic device structure is defined in include/device/device.h; the device-specific structures are defined in the source directory for a given device, always in a file named config.h, e.g. src/northbridge/intel/i440gx/config.h.

For an analogous structure, see the Linux kernel Virtual File System (VFS) code. A VFS is a generic file system, with a generic structure, which can be extended via individual file system structures.

More on device source directories, configuration structure and config.h file

The generic code for the device tree is contained in the device directory. The code for a given device is contained in a single directory, which is not shared with any other device. The naming convention is <device-type>/<vendor>/<device-name>/filename. The config.h file contains configuration information for a given device.

Devices, in some cases, have special control registers that need to be set. In a few cases, generic code can handle these operations: see device/pci_device.c. Device-specific functions for controlling the device and its settings are found in the device-specific directory. All the configuration variables for controlling a device must be defined in a single structure; to reiterate, that structure is defined in the file config.h. It is one structure, instead of a set of variables, because it must be instantiated and initialized by the device tree compiler (dtc), and a pointer to it is held in the generic device structure.

We show an example of a specific device, below. The device is the i440bx emulation.

i440bx.h contains manifest constants defining registers, bits in registers, and so on.

Config.h defines the structures and declarations that allow the static device tree to compile. We show it below.

The file contains an extern declaration, pointing to a set of operations for the chip (needed to get statictree.c to compile); and the chip-specific structure, containing the control variable ramsize.

The Kconfig and Makefile are for the Kbuild system.

Figure 3: config.h for the i440bx

```
extern struct chip_operations northbridge_intel_i440bxemulation_ops;
struct northbridge_intel_i440bx_config
{
/* The various emulators don't always get 440BX right. So we are
 * going to allow users to set the RAM size via Kconfig.
 */
 int ramsize;
};
```

4.3 Bus

Busses, defined in include/device/device.h, connect parent devices to child devices. Busses are attached to a device, and have child devices attached to them.

4.4 Generic device structure and code

Generic devices are defined in include/device/device.h. Devices:

- have a path
- are attached to a bus
- have sibling devices
- have a vendor and device ID
- have a class and hdr type
- have several booleans, describing state, including enabled, initialized, resources have been read, and on the mainboard
- have a rom address, if a rom is attached to them (e.g. VGA)
- have a set of up to MAX_RESOURCES (currently 12) resources. The resources are built into the structure and are not dynamically allocated. Functions to manage the resources attached to a device are found in device/device_util.c
- have links, which are usually empty in the case of everything but a bridge
- have a set of device operations – these are per-device-type
- have a set of chip operations, per chip-type
- have a chip information structure, which is per-chip instance

Path A path names a way of accessing a device. These are defined in include/device/path.h. The path structure is in essence a case-variant type (struct which includes a type and a union of all possible path types).

Device Resources Resources describe memory and I/O address ranges, IRQs, and DRQs. They are define in include/device/resource.h. There can be variations of a resource which include things like prefetchable, cacheable, and so on.

4.5 How are devices created? Via static and dynamic constructors

In V2, there was no formal model for creating and/or allocating device structures. There was not even a formal convention or way of thinking about such operations; this lack of rigor, to some extent, was a result of our limited understanding of how to think about the problem, which in turn was a result of the revolution in design which followed on the release of the Opteron, with its multiple busses per socket, integrated north bridge, ability to site a non-Opteron device in an Opteron socket, on-chip HyperTransport router, and so on.

We learned a lot with V2, and that knowledge underlies the architecture of the V3 device tree. We have introduced a standardized device id, and are using the notion of C++ constructors to unify our thinking.

The device id is very similar to the existing path structure. It is defined in include/device/device.h, and is a standard C case-variant type with a tag and a union.

The device tree compiler is the static constructor. It reads the dts file in the mainboard directory and creates the initial device tree, with the devices and busses already set up.

The dynamic constructor is part of the device tree code. There is a set of default constructors, but each device can have its own private constructors if needed. The constructor structure is simple: it is a standard device id, and a pointer to a device_operations structure.

The purpose of a dynamic constructor is to take a standard device path and a device id, and create a device structure, with as much information filled in as possible. At minimum the device id, path, and device operations are filled in.

5 Boot Process

The boot process consists of a number of independent, seperately compiled components. Unlike V2, we are not using ld scripts to glue these components together, since the overall bugginess of the various tools (as and ld in particular) made use of ldscripts very hard to maintain.

By design, the seperate components can be individually replaced without replacing any other component. This design implies that functions such as print are duplicated in the code. If this duplication causes problems we can revisit this decision.

5.1 Stage 0: Basic startup (ASM, PIC) and CAR (ASM, PIC) in arch/{architecture}

The Stage 0 code is a binary blob that (on x86) lives in 8192 bytes at the top of memory. This code comprises a jump vector to get from reset to the start of the stage 0 code. The stage 0 code is responsible for any steps needed to make the processor behave properly, such as flushing TLBs, clearing paging bits, and so on. Stage 0, on the x86, enables segments but not paging; on other platforms, stage1 might also set up an initial page table. Stage 0 at minimum switches to a protected mode of some sort, and on x86, at minimum switches on 32-bit mode. Stage 0 then sets up Cache-As-Ram (CAR) so that stage 1 can be be written in C, and use functions.

This code "begins life" executing in real mode, at 0xf000:fff0. The code does initial setup, transitions to 32-bit mode, and then falls into the CAR code.

The files are named for the type of target. The current code, named stage0_i586.S, is designed for a generic i586. The file to assemble is determined from the CONFIG_STAGE0 makefile variable, which is set in the mainboard Kconfig file. Please note, there are NO code files included. The assembly code for early startup rarely changes. To give an example, much of the Stage 0 code was written in 1999/2000, and has changed little since. The CAR code has been unchanged since it was written two years ago.

CAR is a standard cache-as-ram assembly source for the architecture. It is actually included in the stage0_*.S file; but we maintain a distinction in the stage nomenclature for now. This code sets up cache-as-ram, zeros a memory area, sets up a stack, and then calls stage 1.

Config variables:

1. CONFIG_CARTEST. Test the CAR once it is enabled.
2. CONFIG_ROMSIZE. Size of the ROM part.

5.2 Stage 1: C, in arch/{architecture}

Initial entry point for stage 1 is arch/{architecture}/stage1.c:stage1_main().

printk() may be used from anywhere in C code, but it will output characters only after the welcome banner has been printed.

1. POST(0x02). (C)
2. Stop secondary processors. (C)
3. Preboot hardware, as from auto.c (C). hardware_stage1()
4. Initialize serial port. (C)

5. Initialize USB debug port. (C)
6. Print welcome banner to celebrate working printk() output. (C)
7. Enable ROM. (C)
8. Decide whether we can proceed or must recover from serial port. Criteria? (C)
9. Checksum the top flash "boot area", if it is bad then ... recover from serial port (C, PIC). We can definitely reflash LAR archive, but NOTE: reflashing the boot block is tricky ... (C)
10. Examine the flash. Look in DTB option node, normal property for directory named by the boot type (e.g. 'normal', "fallback', etc.). (C)
11. In that directory, execute 'initram'. If that fails, execute 'initram' in any other directory. If that fails as well, recover from serial port. (C, PIC)
12. Make sure that in '/', there is a decompressor for each compression type used. (C)
13. In the directory named by the boot type, execute 'stage2'. (C)
14. In that directory, look for a file 'payload', and load all of 'payload/segment%d'. (C)
15. Jump to the entry point of 'payload/segment0'. (C)

5.3 Stage 2: Device tree

Run the standard device tree code. This code runs in 6 phases. The device tree, as set up by dts, has two ways it can be traversed.

The first is the hierarchy formed by busses and devices. Devices have up to MAX_LINKS links, which are initialized as part of the process of creating the static tree. These links point to busses. A bus has a child device, a device associated with it (e.g. a PCI bridge device), and other attributes described elsewhere. Some operations, such as enumeration, require that the tree be traversed in the hierarchy represented by the bus/device relationship. This traversal starts at the root device, and for each link, follows those busses to the other devices.

The second is a simple traversal, via linked list, of all the devices. This faster, less complex traversal, is performed when there is no need to be concerned with the bus/device relationship.

Figure 4: The dts for the emulation/qemu target

```
/{ config="mainboard,emulation,qemu-i386";
 ~ ~ ~ ~cpus { ...};
%%
struct mainboard_emulation_qemu_i386_config root = { .nothing = 1, };
```

Phase 1 – very early setup This phase existed to make printk work and has been obsoleted.

The simple traversal (forall devices) is used for this phase.

Post codes:

- Entry: 0x20
- Exit: 0x2f

Phase 2 – preparation for bus scan These are functions that are required before any PCI operations of any kind are run.

The simple traversal (forall devices) is used for this phase.

Post codes:

- Entry: 0x30
- Exit: 0x3f

Phase 3 – bus scan This phase is typical of all the phases that do a hierarchical traversal. It starts at the root device (i.e. the mainboard), which uses the distinguished function *dev_root_phase3*. Some root devices have special setup requirements, and there is a way to call this special setup code. If the dts has specified a configuration for the root device, it is possible to set up an enable_dev function. In other words, for any device, it is possible to call some 'preparation' code for that device. We show an example of such a function below, from the QEMU mainboard. First, we show the dts, to show how the chip operations can be enabled.

The dts has been shortened for readability. Note the use of the 'config=' in the root. It specifies that there is an initialized structure after the %% in the dts file. The structure is at the bottom. The dts generates the code shown below.

The code after the %% is reproduced exactly. The dts generates a generic device struct, and initializes the .chip_ops struct member to point to the mainboard_emulation_qemu_i386_ops structure.

When phase 3 is run, the code checks the chip_ops structure member and, if it is non-zero, checks the chip_ops->enable_dev pointer and, if it is non-zero, calls it.

Figure 5: Code generated for the dts

```
struct mainboard_emulation_qemu_i386_config root = { .nothing = 1, };
struct device dev_root = {
     .path =  { .type = DEVICE_PATH_ROOT },
    .chip_ops = &mainboard_emulation_qemu_i386_ops,
.
.
.
};
```

Figure 6: What the mainboard file looks like with enable_dev

```
static void enable_dev(struct device *dev){
    printk(BIOS_INFO, "qemu-i386 enable_dev done\n");
}
struct chip_operations mainboard_emulation_qemu_i386_ops = {
   .name = "QEMU Mainboard",
      .enable_dev = enable_dev
};
```

The mainboard code is shown below. The enable_dev function will be called in phase 3, *before* any other enumeration is done for that device.

Root_dev_phase3, which is called with the root *device*, calls dev_phase3, for each device attached to the root device. The devices are, in fact, bridge devices, i.e. the device attached to a bus. Dev_phase3, in turn, checks the bus device to see if it is a non-NULL pointer, if is enabled, if it has ops and a phase 3 ops; if so, the functions calls the bus device's phase 3 op to kick off scanning of busses.

The phase 3 op is different for each type of bus. For the root bus, which is statically configured, the phase 3 operation walks the set of statically initialized pointers for the root device; for the (e.g.) PCI device, which is much more dynamic, the code does actual probing.

Some busses require a reset operation after scanning. The dev_phase3 code will scan its subordinate busses, and then test all the busses to see if a reset is needed. If so, for each bus that needs a reset, a reset is performed, and *the bus scanning operation is repeated* until a reset is no longer needed.

To sum up, the operation for phase 3, bus scanning, is as follows

- The root device is the starting point for bus scanning
- After some initial setup, including an optional call to the chip_ops->enable_dev method for the root device, the dev_phase3 function is called with the root

device as the parameter.

- The dev_phase3 function, after checking that the bus has the ability to be scanned (i.e. the device has an ops->phase3 pointer), scans the bus by calling the phase3 function for the bus.
- If scanning results in a need for a reset, the reset(s) are performed on the links that need it, and *the scan operation is repeated.* This cycle continues until no resets are needed.

The per-device phase 3 operation for a bus has a mutually recursive relationship with dev_phase3. The per-device function is called with the pointer to the device that was passed into dev_phase3. The per-device phase 3 iterates over the set of child links (i.e. busses) that are attached to the device and, for each link, checks the chip_ops of the child link device for each link, and determines whether to call the enable_dev for each child link device. The one quite non-intuitive action that some of these functions take is to enable the child link device, whether the child link device is enabled or not in the configuration. This enable is done in order to ensure that child busses are properly enumerated, whether they are enabled or not.

Once the child link devices have been properly examined and (for some busses) set up for enumeration, the per-device phase 3 operation iterates over the child link devices one more time and calls dev_phase 3 for each child link device. This final loop completes the enumeration for this level of the hierarchy.

At the end of the per-device phase 3 operation, the structure of the physical device tree has been completely determined, including both the static devices and any dynamic devices, such as cards plugged into PCI slots. For each level of the tree, the structure which define the devices, and busses, have been filled in, and the presence or absence of devices has been determined. At the end of this pass, it is possible to determine what resources each device will need, and to allocate those resources as needed.

Post codes:

- root_dev_phase3 entry: 0x40
- After enable_dev is tested and potentially called: 0x41
- dev_phase3 entry: 0x42
- dev_phase3 entry: 0x4e (note: since this is a recursive function, the post codes can cycle from 4e to 43 and back again)
- root_dev_phase3 exit: 0x4f

Phase 4 The point of phase 4 is to determine what resources are needed for each device; to allocate those resources; and to configure the devices with those resource values. Resources are determined in one of two ways. Some devices, if present, have static resource requirements (e.g. superio parts, which have a fixed requirement for two I/O addresses). Other devices have resource requirements that can be determined by reading registers (such as Base Address Registers in PCI) and are hence dynamic.

A similar mutual recursion is employed, starting again at the root. The root devices phase 4 ops are called with the root device as a parameter. For each link on the device, and for each type of resource that is needed to be determined, the compute_allocate_resource function is called. This function takes a bus, resource, mask, and type as a parameter. As busses as scanned, and resources are read, the mask is applied ot the resource and compared to the type, so as to select the type of resource desired.

Once the reading of resources is done, the root device has IO resources as resource 0, and mem resources as resource 1.

After the generic resource reading has been done, there is one special case, for VGA, which overrides the standard hierarchical traversal. If VGA console is enabled, the bridges must be configured in such a way as to pick a "primary" VGA device. Once the resources have been enumerated, a function called allocate_vga_resource is called. This function traverses the devices in non-hierarchical order, and selects one of them as the VGA device for the so-called "compatibilty chain". Once this device is selected, the function walkss the tree from the device to the root, enabling the VGA CTL bit in each bridge.

Once this phase has been done, all the memory and IO resources have been enumerated and allocated to each device, and to each bridge, in the system. This phase is easily the most complex of all the phases in stage 2.

Post codes:

- Entry: 0x50
- Exit: 0x5f

Phase 5

Post codes:

- Entry: 0x60
- Exit: 0x6f

Phase 6 Post codes:

- Entry: 0x70
- Exit: 0x7f

5.4 Stage 3: elf boot

WARNING: you can not load any elf segment in the range 0 to 0x1000. That is our stack.

1. Each file has a four-byte checksum at the end. Check the checksum for each one. (C)

2. If all the tests pass, run each one, in order, decompressing those which need it. The last one might not return. If the checksum fails, If the test fails, use the backup property in the option node to find a backup. initram is (C, PIC) as it must execute in place. The coreboot payload will be uncompressed to RAM, and is in C, but need not be PIC.

5.5 Stage 4

6 The static tree (This part needs to be updated, once the other stages are done)

The static tree is generated from the DTS. Shown is a sample DTS, for QEMU. Note that we don't fill out all properties of each node, e.g. the northbridge. The sum total of all properties is found in the dts for that node in the source directory, i.e. src/northbridge/intel/440bx/440bx.dts (is this name ok? Or just chip.dts?)

6.1 How DTC will compile the DTS

There are two pieces to the static tree. The first is the tree itself. As in v2, the user does not see the structures and types that define this tree; the user does define the structure of the tree by the way they lay out the config file. Sibling, child, and parent references are defined by the use of reserved names (sibling, child, and parent, unsurprisingly) and the use of & to define what the sibling, child, and parent keywords are referring to.

The second part of the tree is the per-chip and per-device information. As in v2, each device or chip can define a structure which defines per-device information. These structures are called config structures, and define per-instance configuration of a chip. A survey of all the v2 structures shows that for almost all such config structures, almost all use int, unsigned long and unsigned int, char, and array of char types. However, for superio parts, the config structures in almost all cases contain structure declarations. We could in theory resolve the superio issue as follows: define the superio struct as having links, much as our other structures do now:

```
struct superio {
    void *links[8];
};
```

Figure 7: Sample DTS

```
/{
        model = "qemu";
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "emulation-i386,qemu";
        cpus {
                #address-cells = <1>;
                #size-cells = <0>;
                emulation,qemu-i386@0{
                        name = "emulation,qemu-i386";
                        device_type = "cpu";
                        clock-frequency = <5f5e1000>;
                        timebase-frequency = <1FCA055>;
                        linux,boot-cpu;
                        reg = <0>;
                        i-cache-size = <2000>;
                        d-cache-size = <2000>;
                };
        };
        memory@0 {
                device_type = "memory";
                reg = <00000000 20000000>;
        };
        /* the I/O stuff */
        northbridge,intel,440bx{
                associated-cpu = <&/cpus/emulation,qemu-i386@0>;
                southbridge,intel,piix4{
                        superio,nsc,sucks{
                                uart@0{
                                        enabled=<1>;
                                };
                        };
                };
        };
        chosen {
               bootargs = "root=/dev/sda2";
           linux,platform = <00000600>;
           linux,stdout-path="/dev/ttyS0";
        };
        options {
               normal="normal";
               fallback="fallback";
   };
};
```

Figure 8: How we get from the mainboard DTS to C

Then initialize them:

```
struct superio superio {
    .links = {&pc_keyboard, &com1, &com2, 0};
}
```

In our opinion, this is asking for trouble. We currently, in the superio code, can catch stupid errors in usage that would be lost were we to go to this void * based approach. In fact, we can argue that we ought to be adding stronger type checking to the tree, not taking it away. As of this version of the document, the handling of the superio is not defined.

Note that we are going to need an unflatten tool to generate the device tree. The steps are as follows:

- Compile time creation of the C structures.
- Run-time filling in the blanks with data about real hardware.
- Runtime generation of the OFW device tree.

The DTS is defined per each mainboard. It uses elements which are actually defined elsewhere – for example, if the user references the Intel 440BX northbridge, the DTC must pull in northbidge/intel/440bx/dts to get the full set of definitions. Call the full DTS the base DTS; call the DTS mentioned in the mainboard DTS the instance DTS. Each member of the DTS from the base DTS must be initialized in some manner so we can infter type and default values. The instance can define some, all, or none of the values. The DTC will create a C file with structure declarations and initializations in it.

We show how this looks in

7 Makefile targets

7.1 lzma

This is for creating the coreboot.lzma file.

7.2 initram

This is for creating initram. The actual files used can be defined in any Makefile that is part of this build. Typically, the files are defined in the northbridge Makefile.

7.3 coreboot_ram

This is the code that runs in RAM. This is almost always hardwaremain(). This code is almost always defined by the mainboard Makefile.

7.4 payload

This is what we boot. Almost always this is FILO, Etherboot, Linux kernel, Open FirmWare, and so on.

7.5 coreboot.lar

This is the "file system" that contains the lzma, initram, coreboot_ram, and payload targets.

7.6 jumpvector

This is the jumpvector. Jumpvector is entered at power on reset (POR) or hard or soft reset.

7.7 vpd

This contains information that a payload can use to find out about the mainboard.

8 Conclusions

This is great stuff.

9 Appendix A: Issues

- On most non-x86 architectures, the bootblock is at the start of the flash, not at the end. The general structure of the flash layout can stay the same on such systems, just flipped upside down.
- Move over to the Xorg version of x86emu/biosemu, drop the one we have now as it is not complete enough. (Ron is not so sure about this, since we have done our own bug-fixes to x86emu)

10 Comments from Peter Stuge

- Ridiculous and error-prone to require commands in three dirs for a build. (Edit targets/foo/bar/Config.lb, run ./buildtarget foo/bar in targets and finally cd targets/foo/bar/baz to make.) (Deps fail on reconfig, I've gotten the wrong payload a couple of times causing annoying extra reboots/hotswaps/flashes.)

- Flash ROM size needs to affect one option, and one option only. Maybe even autodetect it for those building on the target. All other sizes can and MUST be derived from this value. Also: What about option ROMs? Should we aim to produce a ready-to-use lb-2.0-epia.rom and require a correct (how carefully do we check?) vgabios.rom in order to build with VGA support - or just dump a half- finished product in the user's lap and require them to finish the puzzle on their own? Licensing issues? Is "cat" considered "linking"?
- Any redundancy in the config/build process should be removed. I must not need to type the target name more than once. Brings me to..
- Global vs. local builds - pros/cons with kernel style (global) build (always produces arch/x/*Image) and coreboot v2 style build (produces target/x/y/z/coreboot.rom for each target) Either way the config/build system must be consistently either global or local.
- Support for target variants? Same mobo with/without certain parts populated. Perhaps just sets of default options that can be pre-selected as a base config and then still allow user to change whatever they want. (Kconfig has just one variant per arch, right?)
- ..basically we want a system that is able to do very complex detailed configurations but that's also able to hide all the details behind "512KiB EPIA-MII 6000E without CF addon" (hypothetical example)
- Some boards will require more from the user, but when possible a config and build should be dirt simple.
- One idea is a kind of iterative config with increasing resolution per iteration. Novice users with a known-good board need only complete the first iteration: flash size, board name and board variant if any. Further iterations are optional and allow increasingly specific settings. Think fdisk normal/expert mode.
- Payload. I say something must be included in the coreboot tree or trivially added to a tree by download or command. FILO is candidate for inclusion. What's up with FILO(EB) and FILO(CB) ? Merge them? Make EB default payload? FILO? memtest86? All about making a usable product. memtest86 would have to be explicitly selected in expert mode in favor of the default option that would be able to load an OS. Doesn't matter much if it's only Linux right now because that's the most likely boot candidate for early CB adopters.
- Payload config. Long/tedious for EB, simple default for boards with onboard LAN, what to do otherwise? Tricky for FILO. (e.g. EPIA-MII CF boot requires IDE+!PCI, !PCI requires !USB or build fails) filesystems, devices, etc.

- Kernel payload and payload utilities - where to get mkelfImage? I had to look hard. Should it be downloaded on demand? Perhaps after the user chooses her payload? Think cygwin installer that downloads selected packages. Maybe a bad idea.
- Consistent terminology - the payload seems to have many names in the decompression code. ;)

11 Case study: new port

This is a case study of installing the amd norwich board into V3. For each commit, I'll record what we did.

11.1 319

- Index: mainboard/Kconfig – had to modify this to include the amd directory
- Index: mainboard/amd/Kconfig – had to add and modify this for the norwich
- Index: mainboard/amd/norwich/Kconfig – created by modifying the qemu target.

11.2 320

- Index: southbridge/Kconfig – had to modify this to include the amd options
- Index: southbridge/amd/Kconfig – had to modify this to include the cs5536 options
- Index: southbridge/amd/cs5536/Makefile – created by modifying the Intel piix4 target.
- Index: southbridge/amd/Makefile – had to modify this to include the amd directory
- Index: southbridge/Makefile– had to add and modify this for the cs5536

11.3 321

- Index: northbridge/Kconfig – had to modify this to include the amd options
- Index: northbridge/amd/Kconfig – had to modify this to include the cs5536 options

- Index: northbridge/amd/geodelx/Makefile – created from scratch – we're getting the hang of this
- Index: northbridge/amd/Makefile – had to modify this to include the geodelx
- Index: northbridge/Makefile – had to add and modify this for adding amd

11.4 322

We had some errors and made some changes for building.

- Index: mainboard/amd/Kconfig
- Index: mainboard/amd/norwich/Makefile

11.5 R323

The real work begins. To this point it has been more directory structure and Kconfig.

- Create the file arch/x86/amd_geode_lx.h. This file contains definitions for the chip. This is from V2, and had a few mods for V3 conventions. Also, there were complaints about spacing etc. and these got fixed.
- Create the file arch/x86/stage0_amd_geodelx.S. This contains the CAR code for the CPU. We took the qemu target and pulled out the "CAR" part and pulled in the v2 car code for the LX. The CAR part really gets sandwiched in between startup stuff at the beginning of the file and the jmp vector at the end. It begins at the DCacheSetup label and ends before the leave_DCacheSetup label. This builds.

Next we bring in the initram.c from v2. This is coreboot-v2/src/mainboard/amd/norwich/cache_as_ram_auto.c. It will not build in V3, as the includes are wrong. We fix these and, while we are at it, change the entry point to be called main(). This file will be a standalone file in the coreboot Lightweight Archive (LAR), and hence needs to have main() as the entry point.

We pretty much get a ton of errors once include is fixed. Why? Because the old code was started from romcc, and included lots of .c files, since that is how romcc worked. The new code is going to be linked. We took all the .c includes out, so now we get this:

```
/home/rminnich/src/bios/coreboot-v3/mainboard/amd/norwich/initram.c: In function 'spd_read_byte': /home/rminnich/src/bios/coreboot-v3/mainboard/amd/norwich/initram.c:30: er
```

Now what we have to do is start building initram in the familiar way, via linking it with other .o files so it can become a true standalone program.

Hold on. What are we doing here? We need to create an initram file for the LAR. This initram file is going to set up DRAM. Coreboot supplies a skeleton function, which we show below, and the programmer needs to supply some functions for their chipsets, so that this function can work.

What code is needed? A few things. The northbridge code must supply register set functions. The southbridge or superio must supply smbus read functions. The basic sdram setup is found in lib/ram.c, and is dead simple:

```
void ram_initialize(int controllers, void *ctrl) {
  int i;
       /* Set the registers we can set once to reasonable values. */
        for (i = 0; i < controllers; i++) {
                printk(BIOS_INFO,
                       "Setting registers of RAM controller %d\n", i);
                ram_set_registers(ctrl, i);
        }
        /* Now setup those things we can auto detect. */
        for (i = 0; i < controllers; i++) {
                printk(BIOS_INFO,
                  "Setting SPD based registers of RAM controller %d\n", i);
                ram_set_spd_registers(ctrl, i);
        }
        /* Now that everything is setup enable the RAM. Some chipsets do
         * the work for us while on others we need to it by hand.          */
        printk(BIOS_DEBUG, "Enabling RAM\n");
        ram_enable(controllers, ctrl);
        /* RAM initialization is done. */
        printk(BIOS_DEBUG, "RAM enabled successfully\n");
}
```

Ram_initialize is a core function of initram. When it is called, RAM is not working; after it is called, RAM is working. This function in turn calls functions in the northbridge code (or, in some cases, CPU code; it depends on the part). The basic idea is that this code is called with a pointer to an opaque type (ctlr *), and an int indicating how many controllers, dram slots, or whatever “things” there are, where a “thing” is totally chipset dependent. The lib/ram.c code sets hardcoded settings, then sets dynamic settings by querying the SPD bus, then enables the RAM. This basic cycle has been refined now for eight years and has worked well on many systems.

The northbridge code has to provide three basic functions. The first function, ram_set_registers, sets up basic parameters. It will be called for each of the ram “things”, where, as described above, “thing” can be just about anything, depending on the chipset. The function will be called with the ctlr pointer, and in index in the range 0..controllers-1. The second function, ram_set_spd_registers, is called to tell the northbridge code that it should do spd setup for “thing” i. Finally, the northbridge-provided enable function is called.

Any or all of these functions may be empty. In the common case, they all do something. These functions, in turn, may require other functions from other chipset parts. The most important, and common, set of functions reads SPD values using the SMBUS. The mainboard must configure, as part of stage2, a file to be compiled which provides these functions. The simplest function is called smbus_read_byte(unsigned device, unsigned address). This function should do any needed initialization, to keep life simple for the northbridge code, and then read from SMBUS device 'device' at address 'address'. Typically, the device address range is 0xa0 up to 0xa8. The address depends on the DRAM technology.

All of the coreboot code that is run after this point uses the device tree; none of the initram code uses the device tree. The reason is simple: the device tree lives in RAM. This bootstrap code is intentionally simple and does not use the device tree.

We will start by providing SMBUS functions. The SMBUS for this board is supported on the AMD CS5536 chip. The file we create will be in southbridge/amd/cs5536/smbus_initram.c.

The revision numbers skip a bit here, since others are also working on V3. We start with revision 339.

R339

Get the old code:

```
    cp coreboot-v2/src/southbridge/amd/cs5536/cs5536_early_smbus.c southbridge/amd/cs5536/s
```

Then we need to set up the mainboard Makefile to include this file in the mainboard stage2. This is pretty easy: add the .o for this file to the INITRAM_OBJ in the mainboard Makefile:

```
INITRAM_OBJ=$(obj)/stage0.init $(obj)/stage0.o $(obj)/mainboard/$(MAINBOARDDIR)/initram.o $(obj)/southbridge/amd/cs5536/smbus_initram.o
```

Now we get lots more errors, so off we go to fix them!

R334 The errors were mainly changes to printk. Also, we had to copy the old cs5536.h from the V2 tree. This file is only used for the southbridge part, so, following coreboot include rules, cs5536.h does not go in include; rather, it goes in the `southbridge/amd/cs5536` directory. We fix the includes so that they have proper names, and add Doxygen comments, which as of June 1, 2007, are required for all commits. Also, all functions save smbus_read_byte are static, as only that one function is needed externally; smbus_read_byte has a static, first_time, which is checked and, if set, smbus_read_byte will initialize the smbus hardware via smbus_enable.

At this point, we have smbus support for the chipset, an essential first step.

R345 We bring across the cs5536_early_setup.c, and make the few changes needed to get it to compile.

Now it is time to fix the compilation errors. First, we need to get the pll_reset.c function from V2. This function requires few changes. Mainly, they involve removing artifacts of romcc, such as a lack of include directives, and use of simple print functions instead of printk.