LinuxBIOS on AMD64

written by Stefan Reinauer <stepan@suse.de>
in courtesy of SUSE LINUX AG, Nuremberg

Table of Contents

1. Abstract

This document targets porting LinuxBIOS to new Motherboards and creating custom firmware images using LinuxBIOS. It describes how to build LinuxBIOS images for the AMD64 platform, including hypertransport configuration and pertinent utilities. If you are missing information or find errors in the following descriptions, contact Stefan Reinauer <stepan@suse.de>

2. What is LinuxBIOS?

LinuxBIOS aims to replace the normal BIOS found on PCs, Alphas, and other machines with a Linux kernel that can boot Linux from a cold start. The startup code of an average LinuxBIOS port is about 500 lines of assembly and 5000 lines of C. It executes 16 instructions to get into 32-bit mode and then performs DRAM and other hardware initializations required before Linux can take over.

The projects primary motivation initially was maintenance of large clusters. Not surprisingly interest and contributions have come from people with varying backgrounds. Nowadays a large and growing number of Systems can be booted with LinuxBIOS, including embedded systems, Desktop PCs and Servers.

3. Build Requirements

To build LinuxBIOS for AMD64 from the sources you need a recent Linux system for x86 or AMD64. SUSE Linux 8.2 or 9.0 are known to work fine. The following toolchain is recommended:

- GCC 3.3.1
- binutils 2.14.90.0.5
- Python 2.3
- CVS 1.11.6

NOTE: Later versions should also work. Prior versions might lead to problems.

4. Getting the sources

The latest LinuxBIOS sources are available via CVS. The CVS repository is maintained at SourceForge.net (the project name is "FreeBIOS"). You can get the entire source tree via CVS:

```
% cvs -d:
pserver:anonymous@cvs.freebios.sourceforge.net:/cvsroot/freebios login
```

Hit return when you are asked for a password. Then checkout (or update) the freebios source tree as follows:

```
% cvs -z3
-d:pserver:anonymous@cvs.freebios.sourceforge.net:/cvsroot/freebios co
freebios2
```

Once the source tree is checked out, it can be updated with:

```
% cvs update -Pd
```

Due to recent problems with SourceForge's CVS infrastructure we set up a snapshot site that keeps hourly source trees of the last four days. It is available at http://snapshots.linuxbios.org/. Due to the major structural enhancements to LinuxBIOS, AMD64 support is only available in the `freebios2` tree. This tree reflects (as of November 2003) LinuxBIOS version 1.1.5 and will lead to LinuxBIOS 2.0 when finished. Most x86 hardware is currently only supported by the LinuxBIOS 1.0 tree.

5. LinuxBIOS configuration overview

To support a large variety of existing hardware LinuxBIOS allows for a lot of configuration options that can be tweaked in several ways:

- Firmware image specific configuration options can be set in the image configuration file which is usually found in `freebios2/targets/<vendor>/<motherboard>/`. Such options are the default amount of output verbosity during boot-up, image size, use of fall-back mechanisms, firmware image size and payloads (Linux Kernel, Bootloader...) The default configuration file name is `Config.lb`, but LinuxBIOS allows multiple configurations to reside in that directory.
- Motherboard specific configuration options can be set in the motherboard configuration file placed in `freebios2/src/mainboard/<vendor>/<motherboard>`. The motherboard configuration file is always called `Config.lb`. It contains information on the on-board components of the motherboard like CPU type, northbridge, southbridge, hypertransport configuration and SuperIO configuration. This configuration file also allows to include add-on code to hook into the LinuxBIOS initialization mechanism at basically any point.

This document describes different approaches of changing and configuring the LinuxBIOS source tree when building for AMD64.

6. Building LinuxBIOS

One of the design goals for building LinuxBIOS was to keep object files out of the source tree in a seperate place. This is mandatory for building parallel LinuxBIOS images for several distinct motherboards and/or platforms. Therefore building LinuxBIOS consists of two steps:

- creating a build tree which holds all files automatically created by the configuration utility and the object files
- compiling the LinuxBIOS code and creating a flashable firmware image.

The first of these two steps is accomplished by the `buildtarget` script found in `freebios2/targets/`. To build LinuxBIOS for instance for the AMD Solo Athlon64 motherboard enter:

```
% cd freebios2/targets
% ./buildtarget amd/solo
```

This will create a directory containing a Makefile and other software components needed for this build. The directory name is defined in the firmware image specific configuration file. In the case of AMD's Solo motherboard the default directory resides in `freebios2/targets/amd/solo/solo`. To build the LinuxBIOS image, do

```
% cd amd/solo/solo
% make
```

The LinuxBIOS image filename is specified in the firmware image specific configuration file. The default filename for AMD's Solo motherboard is `solo.rom`.

7. Programming LinuxBIOS to flash memory

The image resulting from a LinuxBIOS build can be directly programmed to a flash device, either using a hardware flash programmer or by using the Linux flash driver devbios or mtd. This document assumes that you use a hardware flash programmer. If you are interested in doing in-system software flash programming, find detailed information:

- http://www.openbios.org/development/devbios.html (/dev/bios)
- http://www.linux-mtd.infradead.org/ (Memory Technology Device Subsystem MTD)

8. LinuxBIOS configuration

The following chapters will cope with configuring LinuxBIOS. All configuration files share some basic rules

- The default configuration file name in LinuxBIOS is `Config.lb`.
- All variables used in a configuration file have to be declared in this file with `uses VARNAME` before usage.
- Comments can be added on a new line by using the comment identifier # at the beginning of the line.
- LinuxBIOS distinguishes between statements and options. Statements cause the LinuxBIOS configuration mechanism to act, whereas options set variables that are used by the build scripts or source code.
- Default configuration values can be set in the motherboard configuration files (keyword `default`)
- Option overrides to the default configuration can only be specified in the build target configuration file `freebios2/targets/<vendor>/<mainboard>/Config.lb` (keyword `option`)

8.1 Common Configuration statements

- `uses`

  All local configuration variables have to be declared before they can be used.

  Example:

  ```
  uses ROM_IMAGE_SIZE
  ```

  NOTE: Only known configuration variables can be used in configuration files. LinuxBIOS checks `freebios2/src/config/Options.lb` to see whether a configuration variable is known.

- `default`

  The default statement is used to set a configuration variable with an overridable default value. It is commonly used in motherboard configuration files.

  Example:

  ```
  default ROM_IMAGE_SIZE=0x10000
  ```

  It is also possible to assign the value of one configuration variable to another one, i.e.:

  ```
  default FALLBACK_SIZE=ROM_SIZE
  ```

Also simple expressions are allowed:

```
default FALLBACK_SIZE=(ROM_SIZE - NORMAL_SIZE)
```

If an option contains a string, this string has to be protected with quotation marks

```
default CC="gcc -m32"
```

- `option`

  The `option` statement basically behaves identically to the `default` statement. But unlike `default` it can only be used in build target configuration files (`freebios2/targets/<vendor>/<mainboard>`). The `option` statement allows either to set new options or to override default values set with the `default` statement in a motherboard configuration file. Syntax and application are the same as with `default`.

8.2 Firmware image specific configuration

LinuxBIOS allows to create different firmware images for the same hardware. Such images can differ in the amount of output they produce, the payload, the number of sub-images they consist of etc.

The firmware image specific configuration file can be found in `freebios2/targets/<vendor>/<motherboard>`.

8.2.1 Firmware image specific keywords

In addition to the above described keywords the following statements are available in firmware image specific configuration files:

- `romimage`

  the `romimage` definition describes a single rom build within the final LinuxBIOS image. Normally there are two romimage definitions per LinuxBIOS build: normal and fallback.

  Each `romimage` section needs to specify a mainboard directory and a payload. The mainboard directory contains the mainboard specific configuration file and source code. It is specified relatively to `freebios2/src/mainboard`. The payload definition is an absolute path to a static elf binary (i.e Linux kernel or etherboot)

```
romimage "normal"
  option USE_FALLBACK_IMAGE=0
  option ROM_IMAGE_SIZE=0x10000
  option LINUXBIOS_EXTRA_VERSION=".0-Normal"
  mainboard amd/solo
  payload /suse/stepan/tg3--ide_disk.zelf
end
```

`buildrom`

The `buildrom` statement is used to determine which of the LinuxBIOS image builds (created using `romimage`) are packed together to the final LinuxBIOS image. It also specifies the order of the images and the final image size:

```
buildrom ./solo.rom ROM_SIZE "normal" "fallback"
```

8.2.2 Firmware image configuration options

In addition to the definitions described above there are a number of commonly used options. Configuration options set in the firmware image specific configuration file can override default selections from the Motherboard specific configuration. See above examples about “option” on how to set them.

- `CC`

  Target C Compiler. Default is “$(CROSS_COMPILE)gcc”. Set to “gcc -m32” for compiling AMD64 LinuxBIOS images on an AMD64 machine.

- `CONFIG_CHIP_CONFIGURE`

  Use new chip_configure method for configuring (non-pci) devices. Set to 1 for all AMD64 motherboards.

- `MAXIMUM_CONSOLE_LOGLEVEL`

  Errors or log messages up to this level can be printed. Default is 8, minimum is 0, maximum is 10.

- `DEFAULT_CONSOLE_LOGLEVEL`

  Console will log at this level unless changed. Default is 7, minimum is 0, maximum is 10.

- `CONFIG_CONSOLE_SERIAL8250`

  Log messages to 8250 uart based serial console. Default is 0 (don't log to serial console). This value should be set to 1 for all AMD64 builds.

- `ROM_SIZE`

  Size of final ROM image. This option has no default value.

- `FALLBACK_SIZE`

  Fallback image size. Defaults to 65536 bytes. This does not include the fallback payload.

- `HAVE_OPTION_TABLE`

  Export CMOS option table. Default is 0. Set to 1 if your motherboard has CMOS memory and you want to use it to store LinuxBIOS parameters (Loglevel, serial line speed, ...)

- `CONFIG_ROM_STREAM`

  Boot image is located in ROM (as opposed to CONFIG_IDE_STREAM, which will boot from an IDE disk)

- `HAVE_FALLBACK_BOOT`

  Set to 1 if fallback booting is required. Defaults to 0.

The following options should be used within a `romimage` section:

- `USE_FALLBACK_IMAGE`

  Set to 1 to build a fallback image. Defaults to 0

- `ROM_IMAGE_SIZE`

  Default image size. Defaults to 65535 bytes.

- `LINUXBIOS_EXTRA_VERSION`

  LinuxBIOS extra version. This option has an empty string as default. Set to any string to add an extra version string to your LinuxBIOS build.

8.3 Motherboard specific configuration

Motherboard specific configuration files describe the onboard motherboard components, i.e. bridges, number and type of CPUs. They also contain rules for building the low level start code which is translated using `romcc` and/or the GNU assembler. This code enables caches and registers, early mtrr settings, fallback mechanisms, dram init and possibly more.

NOTE: The `option` keyword can not be used in motherboard specific configuration files. Options shall instead be set using the `default` keyword so that they can be overridden by the image specific configuration files if needed.

8.3.1 Motherboard specific keywords

The following statements are used in motherboard specific configuration files:

- `arch`

  Sets the CPU architecture. This should be i386 for AMD64 boards. Example:

  ```
  arch i386 end
  ```

- `cpu`

  The cpu statement is needed once per possibly available CPU. In a one-node system, write:

  ```
  cpu k8 "cpu0" end
  ```

- `driver`

  The `driver` keyword adds an object file to the driver section of a LinuxBIOS image. This means it can be used by the PCI device initialization code. Example:

  ```
  driver mainboard.o
  ```

- `object`

  The `object` keyword adds an object file to the LinuxBIOS image. Per default the object file will be compiled from a .c file with the same name. Symbols defined in such an object file can be used in other object files and drivers.

  ```
  object reset.o
  ```

- `makerule`

  This keyword can be used to extend the existing file creation rules during the build process. This is useful if external utilities have to be used for the build. LinuxBIOS on AMD64 uses `romcc` for it's early startup code placed in `auto.c`.

To tell the configuration mechanism how to build `romcc` files, do:

```
makerule ./auto.E
        depends "$(MAINBOARD)/auto.c"
        action  "$(CPP) -I$(TOP)/src $(ROMCCPPFLAGS) $(CPPFLAGS) \
                               $(MAINBOARD)/auto.c > ./auto.E"
end
makerule ./auto.inc
        depends "./auto.E ./romcc"
        action  "./romcc -mcpu=k8 -O ./auto.E > auto.inc"
end
```

Each `makerule` section contains file dependencies (using the `depend` keyword) and an action that is taken when the dependencies are satisfied (using the `action` keyword).

- `mainboardinit`

  With the `mainboardinit` keyword it's possible to include assembler code directly into the LinuxBIOS image. This is used for early infrastructure initialization, i.e. to switch to protected mode. Example:

  ```
  mainboardinit cpu/i386/entry16.inc
  ```

- `ldscript`

  The GNU linker ld is used to link object files together to a LinuxBIOS ROM image. Since it is a lot more comfortable and flexible to use the GNU linker with linker scripts (`ldscripts`) than to create complex command line calls, LinuxBIOS features including linker scripts to control image creation. Example:

  ```
  ldscript /cpu/i386/entry16.lds
  ```

- `dir`

  LinuxBIOS reuses as much code between the different ports as possible. To achieve this, commonly used code can be stored in a seperate directory. For a new motherboard, it is enough to know that the code in that directory can be used as is. LinuxBIOS will also read a Config.lb file stored in that directory. This happens with:

  ```
  dir /pc80
  ```

- `config`

  This keyword is needed by the new chip configuration scheme. Should be used as:

  ```
  config chip.h
  ```

- `register`

  The register keyword can occur in any section, passing additional parameters to the code handling the according device. Example:

  ```
  register "com1" = "{1, 0, 0x3f8, 4}"
  ```

- `northbridge`

  The `northbridge` keyword describes a system northbridge. Some systems, like AMD64, can have more than one northbridge, i.e. one per CPU node. Each northbridge is described by the path to the northbridge code in LinuxBIOS (relative to `freebios2/src/northbridge`), i.e. `amd/amdk8` and a unique name (i.e "mc0")
  Example:

  ```
  northbridge amd/amdk8 "mc0"
    [..]
  end
  ```

- `southbridge`

  To simplify the handling of bus bridges in a LinuxBIOS system, all bridges available in a system that are not northbridges (i.e AGP, IO, PCI-X) are seen as southbridges.

  Since from the CPUs point of view any southbridge is connected via the northbridge, a southbridge section is declared within the northbridge section of the north bridge it is attached to.

  Like the northbridge, any other bridge is described by the path to it's driver code, and a unique name. If the described bridge is a hypertransport device, the northbridge's hypertransport link it connects to can be specified using the `link` keyword.

  Example:

  ```
  northbridge amd/amdk8 "mc0"
    [..]
    southbridge amd/amd8111 "amd8111" link 0
      [..]
    end
    [..]
  end
  ```

- `pci`

  The `pci` keyword can only occur within a `northbridge` or `southbridge` section. It is used to describe the PCI devices that are provided by the bridge. Generally all bridge sections have a couple of `pci` keywords. The first occurrence of the `pci` keyword tells LinuxBIOS where the bridge devices start, relative to the PCI configuration space used by the bridge. The following occurences of the `pci` keywords describe the provided devices. Adding the option `on` or `off` to a PCI device will enable or disable this device. This feature can be used if some bridge devices are not wired to hardware outputs and thus are not used.

Example:

```
northbridge amd/amdk8 "mc1"
        pci 0:19.0
        pci 0:19.0
        pci 0:19.0
        pci 0:19.1
        pci 0:19.2
        pci 0:19.3
end
```

or

```
southbridge amd/amd8111 "amd8111" link 0
        pci 0:0.0
        pci 0:1.0 on
        pci 0:1.1 on
        pci 0:1.2 on
        pci 0:1.3 on
        pci 0:1.5 off
        pci 0:1.6 off
        pci 1:0.0 on
        pci 1:0.1 on
        pci 1:0.2 on
        pci 1:1.0 off
        [..]
end
```

- `superio`

  SuperIO devices are basically handled like brigdes. They are taking their driver code from freebios2/src/superio/. They don't provide PCI, but ISA PnP devices. Normally they are connected to a southbridge. If this is the case, the superio section will be a subsection of the southbridge section of the southbridge it is connected to.

  Example:

```
superio NSC/pc87360 link 1
     pnp 2e.0
     pnp 2e.1
     pnp 2e.2
     pnp 2e.3
     pnp 2e.4
     pnp 2e.5
     pnp 2e.6
     pnp 2e.7
     pnp 2e.8
     pnp 2e.9
     pnp 2e.a
     register "com1" = "{1, 0, 0x3f8, 4}"
     register "lpt" = "{1}"
end
```

8.3.2 Motherboard specific configuration options

The following options are commonly used in motherboard specific configuration files. They should be set using the `default` keyword:

- `HAVE_HARD_RESET`

  If set to 1, this option defines that there is a hard reset function for this mainboard. This option is not defined per default.

- `HAVE_PIRQ_TABLE`

  If set to 1, this option defines that there is an IRQ Table for this mainboard. This option is not defined per default.

- `IRQ_SLOT_COUNT`

  Number of IRQ slots. This option is not defined per default.

- `HAVE_MP_TABLE`

  Define this option to build an MP table (v1.4). The default is not to build an MP table.

- `HAVE_OPTION_TABLE`

  Define this option to export a CMOS option table. The default is not to export a CMOS option table.

- `CONFIG_SMP`

  Set this option to 1 if the mainboard supports symmetric multiprocessing (SMP). This option defaults to 0 (no SMP).

- `CONFIG_MAX_CPUS`

  If CONFIG_SMP is set, this option defines the maximum number of CPUs (i.e. the number of CPU sockets) in the system. Defaults to 1.

- `CONFIG_IOAPIC`

  Set this option to 1 to enable IOAPIC support. This is mandatory if you want to boot a 64bit Linux kernel on an AMD64 system.

- `STACK_SIZE`

  LinuxBIOS stack size. The size of the function call stack defaults to 0x2000 (8k).

- `HEAP_SIZE`

  LinuxBIOS heap size. The heap is used when LinuxBIOS allocates memory with malloc(). The default heap size is 0x2000, but AMD64 boards generally set it to 0x4000 (16k)

- `XIP_ROM_BASE`

  Start address of area to cache during LinuxBIOS execution directly from ROM.

- `XIP_ROM_SIZE`

  Size of area to cache during LinuxBIOS execution directly from ROM

- `CONFIG_COMPRESS`

  Set this option to 1 for a compressed image. If set to 0, the image is bigger but will start slightly faster (since no decompression is needed).

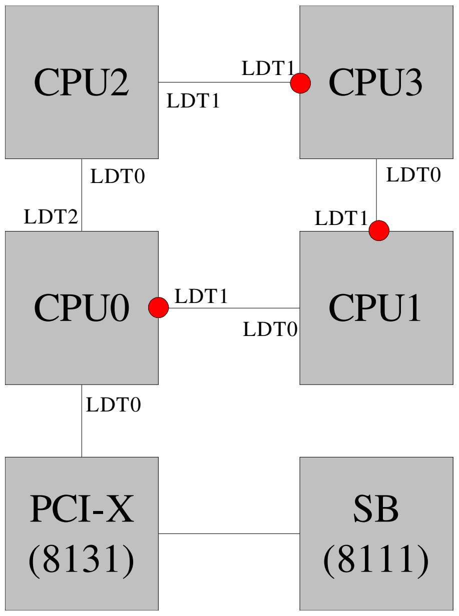
9. Tweaking the source code

Besides configuring the existing code it is sometimes necessary or wished to tweak certain parts of LinuxBIOS by direct changes to the code. This chapter covers some possible enhancements and changes that are needed when porting LinuxBIOS to a new motherboard or just come handy now and then.

9.1 Hypertransport configuration

Before LinuxBIOS is able to activate all CPUs and detect bridges attached to these CPUs (and thus, see all devices attached to the system) it has to initialize the coherent hypertransport devices.

The current algorithms to do coherent hypertransport initialization are not fully automatically evaluating the hypertransport chain graph. Therefore the code needs to be adapted when porting LinuxBIOS to a new AMD64 motherboard. An example arrangement of hypertransport devices looks like this:



Each hypertransport device has one to three hypertransport links that are used for device interconnection. These links are called LDT[012], or accordingly UP, ACROSS, DOWN. Communication between the hypertransport devices can be freely routed, honoring the physical wiring. Teaching the coherent hypertransport initialization algorithm this wiring happens in two steps.

1. Setting outgoing connections

   The algorithm needs to know which outgoing port of a CPU node is connected to the directly succeeding node. This is done in `freebios2/src/mainboard/<vendor>/<mainboard>/auto.c` with a number of preprocessor defines (one define for two-node systems, three defines for four-node systems).

   The ports in question are flagged with a circle in the graph for illustration. For the example graph above (all outgoing connections are realized using `LDT1`/`ACROSS`) the defines are:

   ```
   #define CONNECTION_0_1 ACROSS
   #define CONNECTION_0_2 ACROSS
   #define CONNECTION_1_3 ACROSS
   ```

2. Describing routing information between CPUs.

   There are basically three different message types for hypertransport communication:

   1. request packages
   2. response packages
   3. broadcast packages

   These three message types are routed using different hypertransport ports. The routing information is written to the AMD K8 routing table. In an N-node system this routing table consists of 3*N*N entries , one for each message type and for each possible CPU->CPU communication. For simplicity LinuxBIOS keeps the 3 routing entries for each CPU->CPU communication in one machine word.

   The routing table of each node looks like this:

   ```
   /* Routing Table for Node i
    *
    * F0: 0x40, 0x44, 0x48, 0x4c, 0x50, 0x54, 0x58, 0x5c
    *  i:    0,    1,    2,    3,    4,    5,    6,    7
    *
    * [ 0: 3] Request Route
    *     [0] Route to this node
    *     [1] Route to Link 0
    *     [2] Route to Link 1
    *     [3] Route to Link 2
    * [11: 8] Response Route
    *     [0] Route to this node
    *     [1] Route to Link 0
    *     [2] Route to Link 1
    *     [3] Route to Link 2
    * [19:16] Broadcast route
    *     [0] Route to this node
    *     [1] Route to Link 0
    *     [2] Route to Link 1
    *     [3] Route to Link 2
    */
   ```

The routing table is passed to the coherent hypertransport initialization algorithm by defining a function called `generate_row()` in `auto.c`:

```
    static unsigned int generate_row
                        (uint8_t node, uint8_t row, uint8_t maxnodes)
```

This function is trivial if there is only one CPU in the system, since no routing has to be done:

```
static unsigned int generate_row
                    (uint8_t node, uint8_t row, uint8_t maxnodes)
{
        return 0x00010101; /* default row entry */
}
```

On a two node system things look slightly more complicated. Since the coherent hypertransport initialization algorithm works by consecutively enabling CPUs, it contains routing information for driving the system with one node and two nodes:

```
static unsigned int generate_row
                 (uint8_t node, uint8_t row, uint8_t maxnodes)
{
        uint32_t ret=0x00010101; /* default row entry */
        static const unsigned int rows_2p[2][2] = {
                { 0x00050101, 0x00010404 },
                { 0x00010404, 0x00050101 }
        };

        if(maxnodes>2) maxnodes=2;

        if (!(node>=maxnodes || row>=maxnodes)) {
                ret=rows_2p[node][row];
        }

        return ret;
}
```

Systems with four nodes have to contain routing information for one, two and four node setups:

```
static unsigned int generate_row
                    (uint8_t node, uint8_t row, uint8_t maxnodes)
{
        uint32_t ret=0x00010101; /* default row entry */

        static const unsigned int rows_2p[2][2] = {
                { 0x00030101, 0x00010202 },
                { 0x00010202, 0x00030101 }
        };
        static const unsigned int rows_4p[4][4] = {
                { 0x00070101, 0x00010202, 0x00030404, 0x00010204 },
                { 0x00010202, 0x000b0101, 0x00010208, 0x00030808 },
                { 0x00030808, 0x00010208, 0x000b0101, 0x00010202 },
                { 0x00010204, 0x00030404, 0x00010202, 0x00070101 }
        };

        if (!(node>=maxnodes || row>=maxnodes)) {
                if (maxnodes==2)
                        ret=rows_2p[node][row];
                if (maxnodes==4)
                        ret=rows_4p[node][row];
        }
        return ret;
}
```

9.2 DRAM configuration

Setting up the RAM controller(s) is probably the most complex part of LinuxBIOS. Basically LinuxBIOS serially initializes all RAM controllers in the system, using SPD-ROM (serial presence detect) data to set timings, size and other properties. The SPD data is usually read utilizing the I2C SMBUS interface of the southbridge.

There is one central data structure that describes the RAM controllers available on an AMD64 system and the concerned devices:

```
struct mem_controller {
        unsigned node_id;
        device_t f0, f1, f2, f3;
        uint8_t channel0[4];
        uint8_t channel1[4];
};
```

Available motherboard implementations and CPUs create the need to add special setup code to RAM initialization in a number of places. LinuxBIOS provides hooks to easily add code in these places without having to change the generic code. Whether these hooks have to be used depends on the motherboard design. In many cases the functions executed by the hooks just carry out trivial default settings or they are even empty.

Some motherboard/CPU combinations need to trigger an additional memory controller reset before the memory can be initialized properly. This is, for example, used to get memory working on pre-C stepping AMD64 processors. LinuxBIOS provides two hooks for triggering onboard memory reset logic:

- static void memreset_setup(void)
- static void memreset(int controllers, const struct mem_controller *ctrl)

Some motherboards utilize an SMBUS hub or possibly other mechanisms to allow using a large number of SPD-ROMs and thus ram sockets. The result is that only the SPD-ROM information of one cpu node is visible at a time. The following function, defined in `auto.c`, is called every time before a memory controller is initialized and gets the memory controller information of the next controller as a parameter:

```
static inline void activate_spd_rom
                     (const struct mem_controller *ctrl)
```

The way SMBUS hub information is coded into the `mem_controller` structure is motherboard implementation specific and not closer described here. See `freebios2/src/mainboard/amd/quartet/auto.c` for an example.

LinuxBIOS folks have agreed on SPD data being the central information source for RAM specific information. But not all motherboards/RAM modules feature a physical SPD ROM. To still allow an easy-to-use SPD driven setup, there is a hook that abstracts reading the SPD ROM ingredients that are used by the memory initialization mechanism:

```
static inline int spd_read_byte(unsigned device, unsigned address)
```

This function, defined in auto.c, directly maps it's calls to `smbus_read_byte()` calls if SPD ROM information is read via the I2C SMBUS:

```
static inline int spd_read_byte(unsigned device, unsigned address)
{
        return smbus_read_byte(device & 0xff, address);
}
```

If there is no SPD ROM available in the system design, this function keeps an array of SPD ROM information hard coded per logical RAM module. It returns the “faked” SPD ROM information using device and address as indices to this array.

9.3 IRQ Tables

Motherboards that provide an IRQ table should have the following two variables set in their Config.lb file:

```
default HAVE_PIRQ_TABLE=1
default IRQ_SLOT_COUNT=7
```

This will make LinuxBIOS look for the file `freebios2/src/mainboard/<vendor>/<motherboard>/irq_tables.c` which contains the source code definition of the IRQ table. LinuxBIOS corrects small inconsistencies in the IRQ table during startup (checksum and number of entries), but it is not yet writing IRQ tables in a completely dynamic way.

NOTE: To get Linux to understand and actually use the IRQ table, it is not always a good idea to specify the vendor and device id of the actually present interrupt router device. Linux 2.4 for example does not know about the interrupt router of the AMD8111 southbridge. In such cases it is advised to choose the vendor/device id of a compatible device that is supported by the Linux kernel. In case of the AMD8111 interrupt router it is advised to specify the AMD768/Opus interrupt controller instead (vendor id=`0x1022`, device id= `0x7443`)

9.4 MP Tables

LinuxBIOS contains code to create MP tables conforming the Multiprocessor Specification V1.4. To include an MP Table in a LinuxBIOS image, the following configuration variables have to be set (in the mainboard specific configuration file `freebios2/src/mainboard/<vendor><mainboard>/Config.lb`):

```
default CONFIG_SMP=1
default CONFIG_MAX_CPUS=1 # 2,4,..
default HAVE_MP_TABLE=1
```

LinuxBIOS will then look for a function for setting up the MP table in the file `freebios2/src/mainboard<vendor>/<mainboard>/mptable.c`:

```
void *smp_write_config_table(void *v, unsigned long * processor_map)
```

MP Table generation is still somewhat static, i.e. changing the bus numbering will force you to adopt the code in `mptable.c`. This is subject to change in future revisions.

9.5 POST

LinuxBIOS has three different methods of handling POST codes. They can be triggered using configuration file options.

- Ignore POST completely. No early code debugging is possible with this setting.

  Set the configuration variable `NO_POST` to 1 to switch off all POST handling in LinuxBIOS.
- Normal IO port 80 POST. This is the default behavior of LinuxBIOS. No configuration variables have to be set. To be able to see port 80 POST output, you need a POST expansion card for ISA or PCI. Port 80 POST allows simple debugging without any other output method available (serial interface or VGA display)
- Serial POST. This option allows to push POST messages to the serial interface instead of using IO ports. NOTE: The serial interface has to be initialized before serial POST can work. To use serial POST, set the configuration variable `CONFIG_SERIAL_POST` to the value 1.

9.6 HDT Debugging

If you are debugging your LinuxBIOS code with a Hardware Debug Tool (HDT), you can find the source code line for a given physical EIP address as follows:

Look the address up in the file `linuxbios.map`. Then search the label `Lxx` in the file `auto.inc` created by `romcc`. The original source code file and line number is mentioned in `auto.inc`.

9.7 Devices and Device Drivers

With only a few data structures LinuxBIOS features a simple but flexible device driver interface. This interface is not intended for autonomously driving the devices but to initialize all system components so that they can be used by the booted operating system.

Since nowadays most systems are PCI centric, the data structures used are tuned towards (onboard and expansion bus) PCI devices. Each driver consists of at least two structures. The `pci_driver` structure maps PCI vendor/device id pairs to a second structure that describes a set of functions that together initialize and operate the device:

```
static void adaptec_scsi_init(struct device *dev)
{
      [..]
}

static struct device_operations lsi_scsi_ops  = {
        .read_resources   = pci_dev_read_resources,
        .set_resources    = pci_dev_set_resources,
        .enable_resources = pci_dev_enable_resources,
        .init             = lsi_scsi_init,
        .scan_bus         = 0,
};
static struct pci_driver lsi_scsi_driver __pci_driver = {
        .ops    = &lsi_scsi_ops,
        .vendor = 0xXXXX,
        .device = 0xXXXX,
};
```

By separating the two structures above, `M:N` relations between compatible devices and drivers can be described. The driver source code containing above data structures and code have to be added to a LinuxBIOS image using the driver keyword in the motherboard specific configuration file `freebios2/src/mainboard/<vendor>/<mainboard>/Config.lb`:

```
driver lsi_scsi.o
```

9.8 Bus Bridges

Currently all bridges supported in the LinuxBIOS2 tree are transparent bridges. This means, once the bridge is initialized, it's remote devices are visible on one of the PCI buses without special probing. LinuxBIOS supports also bridges that are non-transparent. The driver support code can provide a scan_bus function to scan devices behind the bridge.

9.9 CPU reset

When changing speed and width of hypertransport chain connections LinuxBIOS has to either assert an `LDTSTOP` or a reset to make the changes become active. Additionally Linux can do a firmware reset, if LinuxBIOS provides the needed infrastructure. To use this capability, define the option `HAVE_HARD_RESET` and add an object file specifying the reset code in your mainboard specific configuration file `freebios2/src/mainboard/<vendor>/<mainboard>/Config.lb`:

```
      default HAVE_HARD_RESET=1
      object reset.o
```

The C source file `object.c` (resulting in `object.o` during compilation) shall define the following function to take care of the system reset:

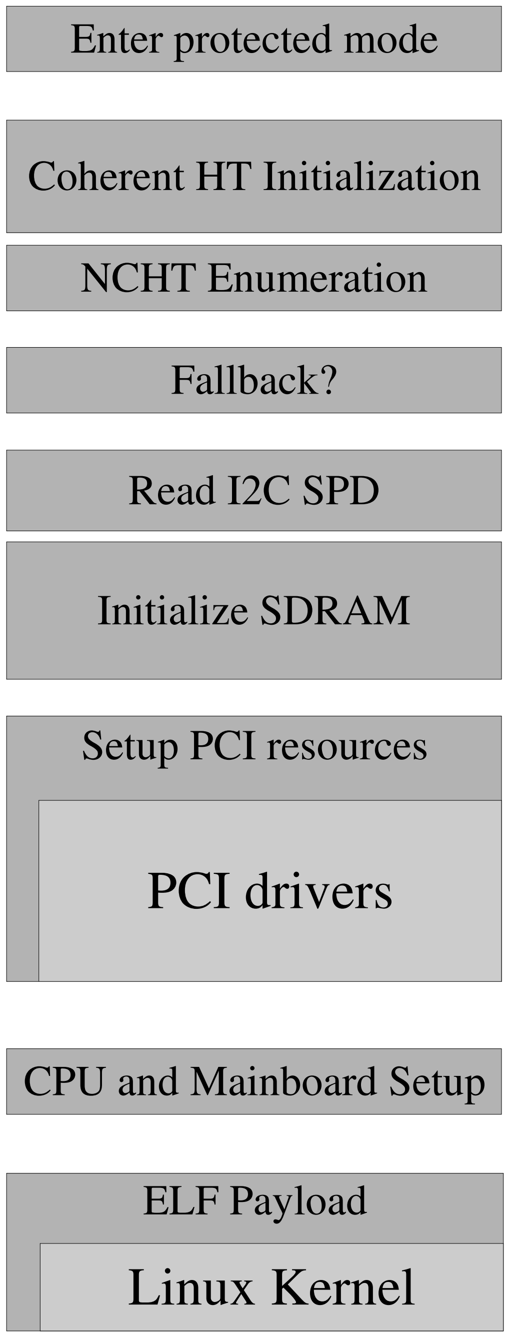
```
      void hard_reset(void);
```

See `freebios2/src/mainboard/arima/hdama/reset.c` for an example implementation.

10. LinuxBIOS Internals

This chapter covers some of the internal structures and algorithms of LinuxBIOS that have not been mentioned so far.

10.1 Code Flow



10.2 Fallback mechanism

LinuxBIOS provides a mechanism to pack two different LinuxBIOS builds within one LinuxBIOS ROM image. Using the system CMOS memory LinuxBIOS determines whether the last boot with a default image succeeded and boots a failsafe image on failure. This allows in-system testing without the risk to render the system unusable.

See `freebios2/src/mainboard/arima/hdama/failover.c` for example code. The fallback mechanism can be used with the `cmos_util`.

10.3 (Un-)Supported Standards

LinuxBIOS supports the following standards

- Multiprocessing Specification (MPSPEC) 1.4
- IRQ Tables
- Elf Booting

However, the following standards are not supported until now, and will probably not be supported in future revisions:

- ACPI
- APM

10.4 LinuxBIOS table

LinuxBIOS stores information about the system in a data structure called the LinuxBIOS table. This table can be read under Linux using the tool `lxbios` from the Lawrence Livermore National Laboratory.

Get more information about `lxbios` and the utility itself at
http://www.llnl.gov/linux/lxbios/lxbios.html

10.5 ROMCC limitations

ROMCC, part of the LinuxBIOS project, is a C compiler that translates to completely rommable code. This means the resulting code does not need any memory to work. This is one of the major improvements in LinuxBIOS V2, since it allows almost all code to be written in C. DRAM initialization can be factored and reused more easily among mainboards and platforms.

Since no memory is available during this early initialization point, romcc has to map all used variables in registers for their time being. Same applies for their stack usage. Generally the less registers are used up by the algorithms, the better code can be factored, resulting in a smaller object size. Since getting the best register usage is an NP hard problem, some heuristics are used to get reasonable translation time. If you run out of registers during compilation, try to refactor your code.

10.5 CMOS handling

LinuxBIOS can use the motherboard's CMOS memory to store information defined in a data structure called the `CMOS table`. This information contains serial line speed, fallback boot control, output verbosity, default boot device, ECC control, and more. It can be easily enhanced by enhancing the CMOS table. This table, if present, is found at `freebios2/src/mainboard/<vendor>/<mainboard>/cmos.layout`. It describes the available options, their possible values and their position within the CMOS memory. The layout file looks as follows:

```
#start-bit length  config config-ID    name
[..]
392        3       e      5            baud_rate
[..]
#config-id   value    human readable description
5            0         115200
5            1          57600
5            2          38400
5            3          19200
5            4           9600
5            5           4800
5            6           2400
5            7           1200
```

To change CMOS values from a running Linux system, use the cmos_util, provided by Linux Networks as part of the LinuxBIOS utilities suite. Get it at

ftp://ftp.lnxi.com/pub/linuxbios/utilities/

10.6 Booting Payloads

LinuxBIOS can load a payload binary from a Flash device or IDE. This payload can be a boot loader, like FILO or Etherboot, a kernel image, or any other static ELF binary.

To create a Linux kernel image, that is bootable in LinuxBIOS, you have to use `mkelfImage`. The command line I used, looks like follows:

```
objdir/sbin/mkelfImage -t bzImage-i386 --kernel /boot/vmlinuz   \
      --command-line="console=ttyS0,115200 root=/dev/hda3"      \
      --initrd=/boot/initrd --output vmlinuz.elf
```

This will create the file `vmlinuz.elf` from a distribution kernel, console redirected to the serial port and using an initial ramdisk.

10.6.1 Kernel on dhcp/tftp

One possible scenario during testing is that you keep your kernel (or any additional payload) on a different machine on the network. This can quickly be done using a DHCP and TFTP server.

Use for example following /etc/dhcpd.conf (adapt to your network):

```
subnet 192.168.1.0 netmask 255.255.255.0 {
          range 192.168.1.0 192.168.1.31;
            option broadcast-address 192.168.1.255;
}


ddns-update-style ad-hoc;

host hammer12 {
          hardware ethernet 00:04:76:EA:64:31;
            fixed-address 192.168.1.24;
              filename "vmlinuz.elf";
}
```

Additionally you have run a `tftp` server. You can start one using `inetd`. To do this, you have to remove the comment from the following line in `/etc/inetd.conf`:

```
tftp dgram udp wait root /usr/sbin/in.tftpd in.tftpd -s /tftpboot
```

Then put your kernel image `vmlinuz.elf` to `/tftpboot` on the `tftp` server.

11. Glossary

- payload

  LinuxBIOS only cares about lowlevel machine initialization, but also has very simple mechanisms to boot a file either from FLASHROM or IDE. That file, possibly a Linux Kernel, a boot loader or Etherboot, are called payload, since it is the first software executed that does not cope with pure initialization.

- flash device

  Flash devices are commonly used in all different computers since unlike ROMs they can be electronically erased and reprogrammed.

12. Bibliography

12.1 Additional Papers on LinuxBIOS

- http://www.linuxnetworx.com/products/linuxbios_white_paper.pdf
- http://www.linuxbios.org/papers/
- http://www.lysator.liu.se/upplysning/fa/linuxbios.pdf
- http://portal.acm.org/citation.cfm?id=512627

12.2 Links

- Etherboot: http://www.etherboot.org/
- Filo: http://te.to/~ts1/filo/
- OpenBIOS: http://www.openbios.org/