AVR32744: AVR32 AP7 Linux Custom Board Support

8-bit AVR® Microcontrollers

Application Note

Features

- Adding a new custom board code to the Linux kernel
  - Boot sequence considerations
  - Adding peripherals to the system
- Adding a new board to the kernel configuration system
  - Menu entries
  - Custom kernel configuration

1 Introduction

This application note describes how the Linux® kernel must be expanded in order to add support for a new custom board.

Because the kernel source changes over the time, a close look at the source code is mandatory to verify that the configurations described in this document are still valid.

Adding support for a custom board makes it easier to maintain changes across kernel releases. A proper board code can also be submitted upstream in order to be added to the official kernel.

Rev. 32106A-AVR32-07/08

2 Custom board setup code

The code for a new custom board should be placed into a sub-directory beneath arch/avr32/boards/, which reflects the board name.

Examples of already existing boards for the AVR®32 architecture in the Linux kernel are the STK®1000 (arch/avr32/atstk1000/) and ATNGW100 (arch/avr32/atngw100/). These examples can be used to form a new custom board.

2.1 Configuration sequence

The main task for the board setup code is to configure the on-chip peripherals and register them with the system. Depending on when in the boot process a module is needed it has to be initialized earlier or later. Also if modules have dependencies within each other the initialization sequence is important. Currently most of the board-specific initialization code is gathered in a setup.c (for the STK1000 the code is spread to other files because this board supports different devices).

2.1.1 Early board setup

The serial console must be initialized very early in the boot process to be able to provide the boot messages. The function prototype can be found in include/asm/arch/init.h and should be implemented like this:

```
void __init setup_board(void)
{
    /* Put early board setup code (like the console) here */
}
```

More details upon the USART setup are described in a later chapter.

2.1.2 Main board setup code

The main initialization is done by specifying a function (or functions) which will be called during the main boot process. Where in the boot process this function is called can be specified by “initcall” macros that are defined in include/linux/init.h. Another situation where an “initcall” is needed is described in the next chapter. The main board setup code should be placed in a function that is defined like the following:

```
static int __init myboard_init(void)
{
    /* Put main board code here */
}
postcore_initcall(myboard_init);
```

In this function all peripheral initializations and device registering with the Linux kernel should be done. In other words put all code that is not especially reserved for other places here.

2.1.3 Late board setup

The flash initialization is done later in the boot process and is therefore not done in the initializations described in the previous chapter. The reason for that is because the SMC flash initialization needs a pre-initialized SMC controller.

The flash setup code is usually gathered in a flash.c file but can also be added to the setup.c file. According to the initialization in the previous chapter a function must be defined so that it is called during the boot process but at a later stage. The function should look like this.

```
static int __init myboard_flash_init(void)
{
    /* Put flash initialization code here */
}
device_initcall(myboard_flash_init);
```

2.2 Clock setup

Linux needs to know the available clock sources in order to calculate all other system clocks. Therefore these values have to be provided in the board setup code. Currently the clocks are specified in arch/avr32/mach-at32ap/at32ap700x.c but this will probably be moved sooner or later to the board setup code because it is board specific.

The boot loader sets up the main system clock and bus speeds and the kernel reads out these values. In order to change the main clock and bus speeds the boot loader has to be recompiled with the new settings.

2.3 Module configuration

To ease the setup of a board the peripheral initialization, code is already available in the AVR32 kernel code. All prepared functions are available by including the header file include/asm/arch/board.h. The according function bodies are in arch/avr32/mach-at32ap/at32ap700x.c.

Sometimes the prepared functions may not serve the actual needs and have to be adapted. For example an adaptation is necessary if the pin multiplexing should be different or unused peripheral pins should be used for other things. In such cases a custom initialization is needed. This should rather be done in the board code than in the original code, which would result in an incompatibility with already existing boards.

Pin numbers that can be used together with the prepared functions can be calculated by using the macros in the file include/asm/arch/at32ap700x.h or in a similar file if another derivate is used. To select I/O line 0 on the peripheral I/O (PIO) controller B for instance, use the following code to get the pin number for the at32_select_gpio function:

GPIO_PIN_PB(0)

For the I/O lines on the other PIO controllers A, C and so forth, use GPIO_PIN_PA(), GPIO_PIN_PC() …

The following chapters give a short overview how to use these prepared functions or reference documentation where this is described.

2.3.1 Main system modules

The first setup function that should be called in the main board setup code is:

```
void at32_add_system_devices(void)
```

This is mandatory because it adds all basic devices such as the interrupt controller, system manager, real time clock and others.

2.3.2 GPIOs

The application note “AVR32408: AVR32 AP7 Linux GPIO driver” from the Atmel web-site describes the needed steps to reserve and configure pins in the board setup code.

2.3.3 LCD controller

Documentation and examples for the configuration of the LCD controller is available in the application note ”AVR32416: AVR32 AP7 Linux LCD Panel Customization”.

2.3.4 USART

The first part of the USART initialization should be placed into the early board setup code as described in chapter 2.1.1. This is the assignment of USART modules to device numbers. The function

```
void at32_map_usart(unsigned int hw_id, unsigned int line);
```

takes as arguments the USART module number `hw_id` (numbering as described in the device datasheet) and a number “line” for the device that it should be assigned to. In addition it will configure the pins (by default only RXD and TXD) for the USART module. For example the following call

```
at32_map_usart(1, 0);
```

assigns the USART1 module to the device 0 and will appear in the /dev directory as ttyS0. If more signals are needed like handshaking signals the source code must be edited or an own implementation of the function needs to be done. Calling the function

```
void at32_setup_serial_console(unsigned int usart_id);
```

sets the device number that should be used as serial console. It will also be used to print the boot messages.

The second part of the USART initialization should be done in the other setup function that is called later in the boot process. The previously preconfigured USART module has to be registered as a platform device in order to load a driver for it. This is done with the function

```
struct platform_device *at32_add_device_usart(unsigned int id);
```

The function argument selects the device number for which a platform device should be registered and the id conforms to the line id's given with the at32_map_usart function.

2.3.5 SPI

The application note “AVR32743: AVR32 AP7 Linux Kernel Module Application Example” describes how a SPI module can be configured and added as a platform device.

The application note “AVR32401: Interfacing DataFlash® in Linux with the AVR32” describes how a DataFlash device can be connected over the SPI interface.

2.3.6 TWI

TWI modules can be added by calling the function

```
struct platform_device *at32_add_device_twi(unsigned int id);
```

The `id` specifies the TWI module number and starts with the number 0 for the first device. The device datasheet lists available TWI modules.

Another possible solution is to use GPIOs as TWI interface. More information about this is available in the application note “AVR32083: AVR32 AP7 Linux TWI Driver”.

2.3.7 Ethernet

The Ethernet address is provided by the boot loader through tags that are passed to the Linux kernel at start up. This is similar to the kernel boot command line. To get the Ethernet address the tags have to be parsed. After that the hardware can be initialized with the address. The needed functions and data to do this can be copied from one of the existing boards.

```
struct eth_addr {
        u8 addr[6];
};
static struct eth_addr __initdata hw_addr[2];
```

Also the two functions `set_hw_addr` and `parse_tag_ethernet` must be copied. The macro

__tagtable(ATAG_ETHERNET, parse_tag_ethernet);

is needed in order to add the `parse_tag_ethernt` functions to the tag parsing process.

The copying of the functions, declarations and definitions may not be needed in later releases of the kernel because they may be moved to the other prepared setup functions.

To add an Ethernet device to the system the following function has to be called:

```
struct platform_device * at32_add_device_eth(unsigned int id,
    struct eth_platform_data *data);
```

The `id` specifies the Ethernet module number and starts with the number 0 for the first device. The `eth_platform_data` is defined as

```
struct eth_platform_data {
        u32     phy_mask;
        u8      is_rmii;
};
```

Setting `is_rmii` to 1 will configure the Ethernet device for the Reduced Media Independent Interface (RMII). The `phy_mask` can be used to mask out everything but the correct address during the auto detection on the management interface. This is normally not needed.

Next step is to configure the Ethernet address by calling:

```
static void set_hw_addr(struct platform_device *pdev);
```

The parameter should be the return value of the `at32_add_device_eth` function.

2.3.8 USB

Following function can be used to add a USB controller to the system.

```
struct platform_device * at32_add_device_usba(unsigned int id,
    struct usba_platform_data *data);
```

The `id` specifies the USB module number and starts with the number 0 for the first device. The `usba_platform_data` is optional (if not used pass in a null pointer instead) and is defined like this:

```
struct usba_platform_data {
    int vbus_pin;
};
```

By using this structure a VBUS signal pin can be specified. The pin number is a GPIO line connected to the VBUS signal from the USB connector. Use the macro GPIO_PIN_NONE to mark unused pins in this structure.

2.3.9 MCI

The MultiMedia Card interface is added by following function to the system:

```
struct platform_device * at32_add_device_mci(unsigned int id,
    struct mci_platform_data *data);
```

The `id` specifies the MCI module number and starts with the number 0 for the first device. The data structure is defined as:

```
struct mci_platform_data {
        int detect_pin;
        int wp_pin;
};
```

In this structure selections can be made to specify pins that should be used as detect and write protect pins. The pin numbers are the GPIO lines connected to the detect signal and write protect signal from the MMC/SD-card socket. Use the macro GPIO_PIN_NONE to mark unused pins in this structure.

2.3.10 AC97 Controller

To add the AC97 controller use following function:

```
struct platform_device *at32_add_device_ac97c(unsigned int id);
```

The `id` specifies the AC97C module number and starts with the number 0 for the first device.

2.3.11 ABDAC

The ABDAC can be added to the system by using following function:

```
struct platform_device *at32_add_device_abdac(unsigned int id);
```

The `id` specifies the ABDAC module number and starts with the number 0 for the first device.

2.3.12 PWM

PWM channels can be added with the following function:

```
struct platform_device *at32_add_device_pwm(u32 mask);
```

The mask parameter specifies the channels that should be added. Each bit in the mask represents a channel but not all channels may be available on the device. The mask 0x3 will add channel 0 and 1 to the system. The datasheet lists available channels and the respective pins that are used.

2.3.13 PS/2 Module (PSIF)

All information regarding a system extension with the PS/2 module(s) is available in the application note “AVR32415: AVR32 AP7 Linux PS/2 keyboard and mouse”.

2.3.14 SSC

A SSC device is added by calling:

```
struct platform_device * at32_add_device_ssc(unsigned int id,
    unsigned int flags);
```

The `id` specifies the SSC module number and starts with the number 0 for the first device. The flags parameter is used to select specific pins of the SSC module that should be used. Possible values are defined as macros.

Table 2-1. Possible values for the flags parameter

Macro	Description
ATMEL_SSC_TK	Configures TX_CLOCK pin
ATMEL_SSC_TF	Configures TX_FRAME_SYNC pin
ATMEL_SSC_TD	Configures TX_DATA pin
ATMEL_SSC_TX	Configures TX_CLOCK, TX_FRAME_SYNC and TX_DATA pins
ATMEL_SSC_RK	Configures RX_CLOCK pin
ATMEL_SSC_RF	Configures RX_FRAME_SYNC pin
ATMEL_SSC_RD	Configures RX_DATA pin
ATMEL_SSC_RX	Configures RX_CLOCK, RX_FRAME_SYNC and RX_DATA pins

2.3.15 CF/IDE

A CompactFlash device is added to the system by following function:

```
struct platform_device * at32_add_device_cf(unsigned int id,
    unsigned int extint, struct cf_platform_data *data);
```

The `id` specifies the CF module number and starts with the number 0 for the first device. The `extint` parameter chooses the external interrupt line to which the IDE device is connected. The datasheet lists available external interrupt lines. Use 0 for the first line, 1 for the second … The last parameter is a data structure that is defined as follows:

```
struct cf_platform_data {
    int     detect_pin;
    int     reset_pin;
    int     vcc_pin;
    int     ready_pin;
    u8      cs;
};
```

The `cs` value specifies the used chip select line. Valid values for the AP7000 and AP7001 are 4 and 5 because these chip select lines are dedicated to the CompactFlash controller. See datasheet for more details. The other structure members can be used to configure optional pins. Use the macro GPIO_PIN_NONE to mark unused pins in this structure.

The application note "AVR32115: ATA Driver for AVR32" describes how to connect parallel ATA devices to the AVR32 using the built-in CompactFlash controller, the External Bus Interface (EBI) and a simple EBI to ATA adaptor. This application note describes an example adaptor card for the STK1000 development board.

To hook up IDE devices the function

```
struct platform_device * at32_add_device_ide(unsigned int id,
    unsigned int extint, struct ide_platform_data *data);
```

can be used. The `id` specifies the IDE module number and starts with the number 0 for the first device. The `extint` parameter chooses the external interrupt line to which the IDE device is connected. The datasheet lists available external interrupt lines. Use 0 for the first line, 1 for the second … The last parameter is a data structure that is defined as follows:

```
struct ide_platform_data {
        u8      cs;
};
```

The `cs` value specifies the used chip select line. Valid values for the AP7000 and AP7001 are 4 and 5 because these chip select lines are dedicated to the CompactFlash controller. See datasheet for more details.

2.3.16 SMC/Flash

As previously mentioned in chapter "2.1.3 Late board setup" the flash initialization has to be done later in the boot process. The configuration of the SMC for a flash device needs three steps. All necessary structures and function prototypes related to the SMC configuration are available in the header file include/asm/arch/smc.h.

The first step is to set proper timings for the flash device. This can be done by calling the function

```
void smc_set_timing(struct smc_config *config,
    const struct smc_timing *timing);
```

with the pre-configured arguments. All values in the `smc_timing` and `smc_config` structure are documented in include/asm/arch/smc.h and are directly related to the register values in the device datasheet.

After the timings are in place the configuration must be set by calling the following function:

```
int smc_set_configuration(int cs, const struct smc_config *config);
```

The `config` parameter should contain the same structure that was used earlier in the timing setup. The `cs` value selects a chip select line. Valid values are described in the datasheet.

The last step is to add a flash device as a platform device to the system. To form the platform device data the flash partitions must be specified in the structure `mtd_partition` that can be found in include/linux/mtd/partitions.h. For an 8 MiB flash this could look like:

```
static struct mtd_partition flash_parts[] = {
        {
                .name           = "u-boot",
                .offset         = 0x00000000,
                .size           = 0x00020000,    /* 128 KiB */
                .mask_flags     = MTD_WRITEABLE,
        },
        {
                .name           = "root",
                .offset         = 0x00020000,
                .size           = 0x007d0000,
        },
        {
                .name           = "env",
                .offset         = 0x007f0000,
                .size           = 0x00010000, /* 64 KiB */
                .mask_flags     = MTD_WRITEABLE,
        },
};
```

More information about the structure members can be found in the header file include/linux/mtd/partitions.h. It is important to place the partition boundaries on a multiple of an erase sector size. The U-Boot environment partition must be at least the size specified in the boot loader (usually 64 KiB). Also the partition for the boot loader must be big enough to fit the image (at the moment 128KiB is enough for U-Boot). The boot loader has to be placed at the beginning of the flash in order to boot. The rest of the flash can be used to store the root file system or other data. To make a partition readonly for the Linux system set the MTD_WRITEABLE flag.

Next step is to tie the mtd-partitions to a `physmap_flash_data` structure that can be found in include/linux/physmap.h. For the above partitions the structure would be initialized like this:

```
static struct physmap_flash_data flash_data = {
        .width          = 2,
        .nr_parts       = ARRAY_SIZE(flash_parts),
        .parts          = flash_parts,
};
```

The `width` value specifies the interface width of the flash device in bytes. The specified partitions are tied to `parts`. To fit this all together two more structures are needed. The memory address range that will be used for the flash must be set in a resource structure.

```
static struct resource flash_resource = {
        .start          = 0x00000000,
        .end            = 0x007fffff,
        .flags          = IORESOURCE_MEM,
};
```

All above structures can now be gathered in a single structure as a platform device.

```
static struct platform_device flash_device = {
        .name           = "physmap-flash",
        .id             = 0,
        .resource       = &flash_resource,
        .num_resources  = 1,
        .dev            = {
                .platform_data = &flash_data,
        },
};
```

To add the platform device to the system call:

```
platform_device_register(&flash_device);
```

3 Adding a custom board to the Linux kernel configuration system

3.1 Menu entry for the board

In order to make a custom board visible as a menu entry in the Linux kernel configuration system the file arch/avr32/Kconfig needs to be edited.

```
choice
        prompt "AVR32 board type"
        default BOARD_ATSTK1000

config BOARD_ATSTK1000
        bool "ATSTK1000 evaluation board"

config BOARD_ATNGW100
        bool "ATNGW100 Network Gateway"
        select CPU_AT32AP7000
endchoice
```

Similar to the above configuration options for the ATSTK1000 and the ATNGW100 a new entry is needed here for a new board. This entry could look like this:

```
config BOARD_MYBOARD
        bool "MYBOARD My own custom board"
        select CPU_AT32AP7000
```

The “select” is optional but can be useful to automatically set a device type. This approach should be used for all boards that use a fixed device type. For the ATSTK1000 this is not useful because different devices can be plugged onto the board. Valid device types are listed in the same file. Currently are CPU_AT32AP7000, CPU_AT32AP7001 and CPU_AT32AP7002 available.

If the board has its own configuration options an additional entry is needed. This entry could look like this:

```
if BOARD_MYBOARD
source "arch/avr32/boards/myboard/Kconfig"
endif
```

Basically the lines above add the “Kconfig” file in the custom board directory to the configuration system. All additional configuration options in the board specific “Kconfig” file will be available now when the board is selected in the menu.

3.2 Build entry

The Makefile in arch/avr32/ needs an entry that adds the path to the board code to the build.

An entry like the following is needed:

```
core-$(CONFIG_BOARD_MYBOARD)   += arch/avr32/boards/myboard/
```

Make sure that “BOARD_MYBOARD” reflects the name of the configuration option described in chapter 3.1 and keep the “CONFIG_” before the name. Also the path should point to the correct board.

3.3 Makefile for custom board

The board needs also a Makefile that adds the files that should be compiled into the kernel to the build. A simple Makefile that adds the board and flash set up code (if separated into two files) would look like this

```
obj-y  += setup.o flash.o
```

and should be placed in arch/avr32/boards/myboard/Makefile.

3.4 Custom kernel configuration

The Linux kernel needs to be configured properly before a build. The current configuration is stored in the file “.config” in the Linux kernel root source directory. (The leading “.” makes this file not visible by default; use e.g. “ls –a” to list it)

An easy and effectively way to configure the kernel is to load a prepared configuration to “.config”. This can be done by hand (just copying the configuration file to “.config”) or more general by using the Linux configuration framework. The Linux configuration framework can be used by running

make ARCH=avr32 myboard_defconfig

with “myboard_defconfig” replaced by a valid configuration. This will copy the configuration from arch/avr32/configs/ to “.config”. The “ARCH=avr32” definition selects the architecture resulting in a search path for the myboard_defconfig configuration of arch/avr32/configs/. Without the definition the configuration file would be searched for in arch/x86/configs/, if developing on a x86 architecture.

Following steps are needed to add a new board configuration.

1. Create a valid Linux kernel configuration for the custom board.
2. Copy the “.config” file to arch/avr32/configs/ and rename it to the desired configuration name but keep “_defconfig” at the end of the name.
3. To load the configuration run “make ARCH=avr32 myboard_defconfig” by using the chosen configuration name instead of “myboard_defconfig”.

4 References

- Related application notes from the Atmel web-site http://www.atmel.com/dyn/products/app_notes.asp?family_id=682
    - AVR32408: AVR32 AP7 Linux GPIO driver
    - AVR32115: ATA Driver for AVR32
    - AVR32743: AVR32 AP7 Linux Kernel Module Application Example
    - AVR32401: Interfacing DataFlash in Linux with the AVR32
    - AVR32415: AVR32 AP7 Linux PS/2 keyboard and mouse
    - AVR32083: AVR32 AP7 Linux TWI Driver
- Linux kernel documentation
    - Documentation/ folder in the Linux kernel sources
    - Linux kernel configuration system

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
Avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature