UM1942
User manual

Linux software user manual for STreamPlug ST2100

Introduction

The STreamPlug ST2100 is a highly integrated SoC including an ARM®-based core, a wide set of peripherals and a PLC modem supporting the HomePlug™ AV standard. The full configuration of STreamPlug software is composed of three most important components as shown in *Section 1: STreamPlug full software architecture on page 10*: the STMicroelectronics® interface layer with the core scheduler, the system software and OK Linux®, (i.e.: a Linux kernel over the hypervisor provided by Open Kernel Labs, Inc. (OK Labs), now General Dynamics Broadband). The minimal configuration of the STreamPlug software includes the native Linux kernel running after the boot, without the core scheduler and hypervisor. This configuration is called also “native Linux” or “native” in following sections.

This document is not intended to be a tutorial on the Linux operating system or embedded software design/development. It only covers topics that are specific to use the STreamPlug Linux.

Contents

List of tables

List of figures

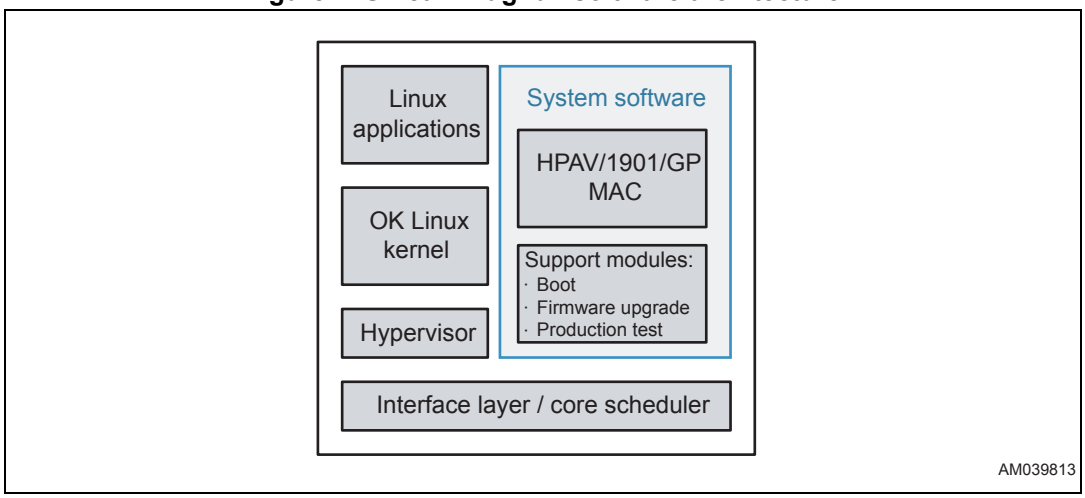
1 STreamPlug full software architecture

The interface layer with the core scheduler provides the necessary APIs to support the system software layer and the hypervisor. The system software provides the core software which implements the HPAV/1901/GP MAC as well as the supporting modules. The OK Linux consists of a collection of all the Linux (2.6.35.0) device drivers that control the specific hardware controllers embedded in the STreamPlug board and the virtualization technology provided by the OKL, (i.e.: the OKL4 Microvisor). Using the OKL technology to host a Linux guest OS confers the following benefits:

- Linux applications can run on the same processor side by side with legacy applications and legacy OSes.
- Concurrent support for two OS environments eliminates the need for either multiprocessor hardware or porting the legacy system to the Linux OS.
- Using “Secure HyperCell™ Technology”, OKL4 native cells can complement the Linux virtual machine (VM) by providing an execution environment with better real-time properties and stronger security.

OKL4 cells are well suited to hosting real-time OSes, easing implementation of latency-sensitive functions without sacrificing the rich ecosystem support available for the Linux.

Figure 1. STreamPlug full software architecture



2 Linux OS

The Linux supplied with the LSP, which is based on the kernel version 2.6.35.0, is licensed under the GPLv2 and distributed with the full source code.

Linux is an open source operating system running on all major processor architectures, including ARM processors. It is supported by a large group of engineers contributing back into the open source. This makes Linux a very dynamic and fast moving operating system. Key benefits of Linux on ARM:

- Complete scalable operating system providing a reliable multi-tasking environment
- Based on an open source model (GPL)
- Leverage a wide range of UNIX and open source applications
- Early availability on ARM processor-based platforms
- Used in many ARM technology-based designs including networking and wireless products
- Broad support through open discussion forums.

Please refer to http://Kernel.org for references. Public forums are available to review patches and information related to Linux development on ARM.

Linux support package (LSP)

STreamPlug LSP supports the following features of Linux:

- Based on Linux-2.6.35.0 version
- Virtual layer provided by the OKL4 between the kernel and HW (not present in the native configuration)
- All drivers integrated into standard Linux device model
- Where ever possible the drivers available from the kernel.org repository mainline have been reused

The LSP incorporates the STreamPlug specific set of drivers shown in *Table 1*.

Table 1. Linux support package

Section	Module
Platform section	Paravirtualized system clock
	Paravirtualized vector interrupt controller (VIC)
	Real-time clock (RTC) driver
Communication device drivers	GMAC Ethernet driver
	USB host
	USB device
	I^2C driver
	SPI driver
	UART driver
	CAN driver
	FIrDA® driver
	PCIe driver (root complex and endpoint)
	SATA driver
Non-volatile memory device drivers	FSMC NAND driver
	FSMC NOR driver
	Serial NOR Flash driver (SMI interface)
	USB mass storage support
	I^2C and SPI memory device support
Accelerators	JPEG driver
	General purpose DMA (DMAC) drivers
	C3 driver
Human interface device (HID) drivers Miscellaneous device drivers	CLCD, LCD panel support
	General purpose I/O (GPIO) driver
	AS GPIO (I/O and PWM) driver
	Watchdog (WDT) driver
Audio support	SPORT-I^2S driver, sound card device support
Video support	TS driver, camera capture support
Virtual devices support (these driver are available only in paravirtualized configuration)	KSP interface
	Misc regs access
	Vlog
	Flash memory shared access
	HPAV driver
	Image validate

3 Platform

This section describes the basic STreamPlug platform code and driver distributed in the standard machine specific layout of the Linux ARM architecture.

3.1 Platform description

The platform or the machine specific code is responsible for

- Initializing Virtual Interrupt Controller (or vector interrupt controller in the native configuration)
- Initializing the timer (clock source and clock event)
- Initializing static memory mapping if required by the system
- Defining the IO_ADDRESS and related macros so that the static memory can be used
- Providing the platform specific code for
  - Clock framework
  - Padmux framework
  - Initialization code for some specific controllers like FSMC and GPIO
  - Defining virtual IRQs in case of shared IRQs on the platform
- Providing system specific header files like those describing IRQ lines and base addresses of respective devices
- Platform specific drivers

3.1.1 Platform software overview

The machine specific code base is distributed among following directories:

- “arch/arm/plat-streamplug”- indicates all the STreamPlug SoCs
- “arch/arm/mach-streamplug - represents the STreamPlug family of boards

The platform is unique and when running in the full configuration it's paravirtualized by the presence of a hypervisor between FW and HW.

The “mach-streamplug” directory contains the following files:

- “clock.c” (machines clock framework)
- “dw_pcie.c” (PCIe functions for Synopsys DW controllers)
- “fsmc-nor.c” (FSMC - flexible static memory controller - interface for NOR Flash)
- “fsmc-sram.c” (FSMC - flexible static memory controller - interface for SRAM device)
- “pswrst_ctrl.c” (STreamPlug machines IP's software reset control source file)
- “miphy.c” (MiPHY™ routine source file)
- “padmux.c” (STreamPlug machines IP's padmux handling source file)
- “streamplug1x.c” (STreamPlug1x machines common source file)
- “streamplug1x_pcie_rev_350.c” (supports STreamPlug1x PCIe rev_350)
- “streamplug10.c” (STreasmPlug10 machine source file)
- “streamplug_devel_board.c” (STreamPlug devel. board source file)
- “streamplug_ksp_agent.c” (STreamPlug KSP interface controller)

The “plat-streamplug” directory contains the following files:

- “clcd.c” (CLCD configuration file)
- “clock.c” (clock framework for STreamPlug platform)
- “ipswrst_ctrl.c” (IP's software reset control for STreamPlug platform)
- “jpeg.c” (JPEG platform specific information file)
- “misc.c” (Misc platform routine source file)
- “padmux.c” (STreamPlug platform specific IP's padmux handling source file)
- “pll_clk.S” (PLL clock configuration for STreamPlug platform)
- “time.c” (STreamPlug platform: timer configuration file)
- “udc.c” (STreamPlug platform: USB device configuration file).

3.1.2 Platform kernel source and configuration

The Linux kernel running on the STreamPlug SoC was inherited from the open source basic software version 2.6.35. The STreamPlug chip is based on the ARM926 architecture. *Table 2* lists the branches added or modified within the ARM Linux tree in order to include this release support of the STreamPlug machine (see *Section 3.2.1*) and device drivers:

Table 2. Linux branches

File or folder	Status
/arch/arm/mach-STreamPlug	New
/arch/arm/plat-STreamPlug	New
/arch/arm/okl4-microvisor	New
/arch/arm/boot/Makefile	Modified in order to support the build of an ELF image compressed
/arch/arm/configs	Modified in order to include machine configurations. Paravirtualized kernel: okl4_hybrid_platform_streamplug_devel_board_defconfig Native Linux configuration: streamplug_devel_board_defconfig
/arch/arm/Kconfig	Modified in order to add the configuration for the STreamPlug platform
/arch/arm/Makefile	Modified in order to set the Linux entry offset to 0x00048000 and to add help comment for the ELF image

3.1.3 Platform configuration

The Linux kernel supports the following device drivers:

- Virtual Interrupt Controller
- Timers
- RTC
- Ethernet “Best Effort”
- USB host
- USB device (Ethernet, Zero and FS gadget) compiled as modules
- I^2C, with STreamPlug configured as master on I^2C bus
- SPI, with STreamPlug configured as master on SPI interface
- UART
- CAN
- FIrDA
- PCIe RC and EP
- SATA
- FSMC
- SMI
- JPEG (decoder and encoder)
- DMA
- C3
- CLCD
- GPIO
- AS GPIO (I/O and PWM)
- Watchdog
- SPORT Audio out & in
- TS
- Virtualised devices (full configuration)

3.2 Board support

The STreamPlug provides numerous possible functions, some of which are multiplexed with each other. These configurations are very much application dependent. Each board designed around the STreamPlug follows the application need to develop a board which exploits a particular kind of applications. Accordingly, software needs to configure the SoC to suit the board layout and let the user to use the desired functionality. Besides multiplexed functionality there are other board dependent configurations like usage of GPIOs which again must be handled in software.

3.2.1 Board registration

In ARM platforms each machine (board) is associated with a unique number called machine ID (MACH_ID). In order to have a one-to-one association, a new MACH_ID should be registered in the table of the all Linux ARM machine, supported by Russell King. This machine ID can be registered from the arm website: www.arm.linux.org.uk/developer/machines/?action=new.

Table 3 lists the machine ID defined into arch/arm/tools/mach-types for STreamPlug boards.

Table 3. STreamPlug machine ID

# machine_is_xxx	CONFIG_xxxx	MACH_TYPE_xxx	Number
STreamPlug	MACH_STREAMPLUG	STreamPlug	4011

3.2.2 Board compilation support

To compile the Linux, first setup these environment variables:

```
$  export CROSS_COMPILE=arm-none-linux-gnueabi--
$  export ARCH=arm
```

The generic commands to use for building the Linux kernel are the following ones:

```
$  make distclean -> to clean all obj files
$  make <configuration  file>->  to configure Linux Kernel for  the desired processor architecture
$  make <type of  image>   -> to build Linux Kernel elf  Image
```

The makefile specific for the STreamPlug machine is below “<linux root>/arch/arm”.

Note: Before building the Linux, install the toolchain provided by the CodeSourcery “arm-2009q1-203-arm-none-linux-gnueabi-”: www.sources.buildroot.net/arm-2009q1-203-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2.

Using:

```
$  make ARCH=arm distclean
```

will clean all *.objs and *.config files.

After cleaning the whole project, it's mandatory to build the configuration for the STreamPlug, before building the Linux kernel image.

In order to build the Linux kernel binary image for the STreamPlug environment, the Linux *.config file has to point to the configuration file specific for the STreamPlug chip; currently the configuration used are included into:

- “okl4_hybrid_platform_streamplug_devel_board_defconfig” for the development SoC board
- “streamplug_devel_board_defconfig” for the development SoC board (for native Linux)

below the “arch/arm/configs” folder.

The command line to setup the default configuration is:

```
$  make okl4_hybrid_platform_streamplug_devel_board_defconfig
```

Executing that command, the following traces will be displayed:

```
HOSTCC  scripts/basic/fixdep
HOSTCC  scripts/basic/docproc
HOSTCC  scripts/basic/hash
HOSTCC  scripts/kconfig/conf.o
HOSTCC  scripts/kconfig/kxgettext.o
SHIPPED scripts/kconfig/zconf.tab.c
SHIPPED scripts/kconfig/lex.zconf.c
SHIPPED scripts/kconfig/zconf.hash.c
HOSTCC  scripts/kconfig/zconf.tab.o
HOSTLD  scripts/kconfig/conf
#
#  configuration written to .config
#
```

In case a new configuration is needed, the following commands are available:

```
$  make menuconfig
```

or

```
$  make xconfig
```

or

```
$  make gconfig
```

using:

```
$  make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-  vmlinux
```

will compile the Linux kernel project and build the kernel image below “<linux root>” for the full configuration

or

```
$  make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-  elfImage
```

will compile the Linux kernel project and build the uncompressed kernel image wrapper in the ELF container below “arch/arm/boot” for the native configuration.

Both configurations support Ext2, JFFS2 and UBI file systems.

3.3 Pad multiplexing support

In order to decrease the number of pads, the STreamPlug design multiplexes some of the functionalities on the same I/O pins. So, which functionality has to be routed on a particular pad is board specific information that has to be activated at run-time according to the options that configure each peripheral, passed to the Linux at the startup via the command line.

Based upon the actual board layout and the functionality which it supports, the devices are listed into an array that is registered to the STreamPlug padmux framework.

From the STreamPlug machine architecture, below “arch/arm/mach-streamplug/padmux.c”:

```
static struct pmx_dev  *streamplug_pmx_dev_lookups[ ]  =
{
   &pmx_dev_arm_gpio1_info,
   &pmx_dev_arm_gpio2_info,
   &pmx_dev_pgc_info,
   &pmx_dev_dai_info,
   &pmx_dev_sport_info,
   &pmx_dev_ts_info,
   &pmx_dev_ark_gpio_info,
   &pmx_dev_clcd_info,
   &pmx_dev_uart1_info,
   &pmx_dev_uart2_info,
   &pmx_dev_sata_info,
   &pmx_dev_pcie_info,
   &pmx_dev_usb_info,
   &pmx_dev_eth_info,
   &pmx_dev_i2c_info,
   &pmx_dev_ssp_info,
   &pmx_dev_can_info,
   &pmx_dev_fsmc_info,
   &pmx_dev_firda_info
}

/* machine IP's  padmux  enabling */
void     init streamplug1x_padmux(void)
{
   int i;

   for (i  =  0; i <  ARRAY_SIZE(streamplug_pmx_dev_lookups);  i++)
      pmx_dev_enable(streamplug_pmx_dev_lookups[i]);
}
```

Padmux handling is enabled during the machine initialization phase.

For development board support (“arch/arm/mach-streamplug/streamplug_devel_board.c”) use:

```
static  void    init streamplug10_devel_board_init(void)
{
   ...

      streamplug1x_padmux();

   ...
}
```

3.3.1 Pad software overview

The devices options passed via the command line to the Linux kernel are parsed by the padmux handler that is in the STreamPlug “arch/arm/mach-streamplug/padmux.c”. Its purpose is to translate the string format of peripheral devices configuration passed by the command line into platform data that will be needed to enable the activation and configuration of a peripheral device by its own driver.

Multiplexed devices on the STreamPlug are abstracted through a structure “pmx_dev” defined as:

```
/*
   * struct pmx_dev:  device definition  structure
   *
   * name:  device name
   * platform:  device to be  register when  pmx is actived
   * modes:  device configuration array for different  modes  supported
   * mode_count:  size of modes array
   * is_active:  is peripheral active/enabled
   * selected:  number of peripheral mode selected at bootime
   */
struct pmx_dev  {
   char *name;
   struct pmx_dev_mode  **modes;
   u8  mode_count;
   u8  selected;
   void (*parse)(struct  pmx_dev  *pmx,  char *options);
};
```

where “pmx_dev_mode” is essentially an array of the register’s set and value which must be written into it in order to enable the device

```
/*
   * struct pmx_dev_mode: configuration structure  every group of modes of
a device
   *
   * name:  mode name
   * mux_regs:  array of mux registers, masks and valuesto enable the device
in
   * this  group of modes
   * mux_reg_cnt:  count  of  mux_regs elements
   */
struct pmx_dev_mode  {

   char *name;
   struct pmx_mux_reg  *mux_regs;
   u8  mux_reg_cnt;
   struct  platform_device *platform_dev;
   struct amba_device *amba_dev;
   struct  platform_device *platform_dev2;
   struct amba_device *amba_dev2;
};
```

where “pmx_mux_reg” is a simple structure defining the address where one must write a particular value to enable the device.

```
/*
   * struct pmx_mux_reg: configuration structure every group of modes of a
device
   *
   * reg: register of multiplexing
   * value: value to be  written
   */
struct pmx_mux_reg  {
   struct pmx_reg   *reg;
   u32  value;
};
```

For example consider enumeration of the multiplexed SSP device:

```
static struct pmx_reg   conf_gpio_g21 =  {
   .address  =  (void  *)( GetOffset (sMiscRegs,  conf_gpio_g21)),
   .mask  =  0x7,
   .offset  =  0,
   .size  =  4,
};

   static struct pmx_mux_reg  ssp_mode_regs[]  =  {
      { .reg  =  &conf_gpio_g21,  .value  =  0x0  },
```

```
};

static struct pmx_dev_mode  ssp_mode  =  {
   .name  =  "primary",
   .mux_regs  =  ssp_mode_regs,
   .mux_reg_cnt  =  ARRAY_SIZE(ssp_mode_regs),
   .amba_dev  =  &streamplug1x_ssp_device,
};

static struct pmx_dev_mode  *ssp_modes[]  =  {
   &ssp_mode,
};

void parse_ssp_options(struct  pmx_dev  *dev,  char *options)
{
   cs_gpio_pin =  simple_strtoul(options,  NULL,  10);
   printk(KERN_INFO  "%s  gpio_pin %d",     func    , cs_gpio_pin);
}

DECLARE_PMX_DEV(ssp,  ssp_modes,  1, parse_ssp_options);
```

where “DECLARE_PMX_DEV” is the macro purpose of which is generated the final device structure according to the optional parameters passed via the command line (in case of SSP options are the GPIO pin number for select CHIP-SELECT) that will be used to enable the correspondent device (i.e.: “pmx_dev_ssp_info” for SSP).

3.3.2 Pad kernel source and configuration

The STreamPlug platform padmux generic framework is implemented in “arch/arm/plat-streamplug/padmux.c”.

3.3.3 Pad usage

Table 4 lists all of the possible configurations to be passed to the ATAG command line in order to enable the desired set of peripherals.

Table 4. Command line options for padmux configuration

Peripheral	Values
CLCD	clcd=on:24bpp, if the CLCD is enabled with 24 bpp clcd=on:18bpp, if the CLCD is enabled with 18 bpp clcd=off, if the CLCD is disabled
PCIe bridge	– pcie=on:rc:1, if the PCIe is configured as a root complex with the MiPHY clock generated by the pll2 input clock – pcie=on:rc:2, if the PCIe is configured as a root complex with the MiPHY clock generated by the qfs4 input clock – pcie=on:rc:3, if the PCIe is configured as a root complex with the MiPHY clock generated by the external clock – pcie=on:ep:1, if the PCIe is configured as an endpoint with the MiPHY clock generated by the pll2 input clock – pcie=on:ep:2, if the PCIe is configured as an endpoint with the MiPHY clock generated by the qfs4 input clock – pcie=on:ep:3, if the PCIe is configured as an endpoint with the MiPHY clock generated by the external clock – pcie=off, if the PCIe is disabled
USB controller	– usb=on:device, if the USB is activated as a gadget – usb=on:host, if the USB is activated as a host – usb=off if the USB is not configured
Ethernet network controller	– eth=<on,off,rtos>:<primary.secondary>:<1,. . . ,3>:<Mac Address> Some examples: – eth=on:primary:1:00:80:40:AE:20:98, if the device driver is configured on low GPIOs groups, with the pll2 input clock as a PHY clock root, and with a default MAC address – eth=on:secondary: :00:80:40:AE:20:98, if the device driver is configured on high GPIOs groups, with the qfs4 input clock as a PHY clock root, and with a default MAC address – eth=on:primary:3:00:80:40:AE:20:98, if the device driver is configured on low GPIOs groups, with the external clock as a PHY clock root, and with a default MAC address – eth=off, if Ethernet is disabled – eth=rtos:primary, if the device driver is configured on high GPIOs groups and assigned to system FW
I^2C controller	– i2c=on, if the I^2C interface is enabled and configured – i2c=off, if the I^2C interface is disabled
Synchronous serial port	– ssp=on:<24,. . . ,39>, if the SPI interface is enabled and configured with a fixed CHIP-SELECT line – ssp=off, if the SPI interface is disabled The default value for number of GPIO used by the OK Linux to reserve the SPI CHIP-SELECT Line is 39. For STreamPlug OK Linux GPIOs the numbers reserved start from 24 to 39.
UART port 1	– uart1=on:primary, the UART1 enabled on the GPIO primary group – uart1=on:secondary, the UART1 enabled on the GPIO secondary group – uart1=off, the UART1 switched off – uart1=rtos:secondary, the UART1 enabled on the secondary group by system FW

Table 4. Command line options for padmux configuration (continued)

Peripheral	Values
UART port 2	– uart2=on:primary, the UART2 enabled on the GPIO primary group – uart2=on:secondary, the UART2 enabled on the GPIO secondary group – uart2=off, the UART2 switched off – uart2=rtos:secondary, the UART2 enabled on the secondary group by system FW
CAN network controller	– can=on:primary, if the device driver is configured on the low GPIOs group – can=on:secondary, if the device driver is configured on the high GPIOs group – can=off, if the device driver is disabled
FIrDA	– firda=on:1, if it supports only the SIR mode – firda=on:2, if it supports only SIR and MIR modes – firda=on:3, if it supports all SIR, MIR and FIR modes – firda=off, if the device driver is disabled
FSMC	– fsmc=on:<nand,sram,nor><0,1>:[initdone] Some examples: – fsmc=on:nand0:initdone, if the FSMC controller is enabled and configured for NAND Flash memory devices with 8-bit data width, while timing parameters are not changed (initialized by RTOS) – fsmc=on:nand1, if the FSMC controller is enabled and configured for NAND Flash memory devices with 16-bit data width – fsmc=on:nor1, if the FSMC controller is enabled and configured for Parallel NOR Flash memory devices with 16-bit data width – fsmc=off, if the FSMC is disabled
SATA	– sata=on:1, if the SATA is enabled and configured with the MiPHY clock generated by the pll2 input clock – sata=on:2, if the SATA is enabled and with the MiPHY clock generated by the qfs4 input clock – sata=on:3, if the SATA is enabled and with the MiPHY clock generated by the external clock – sata=off, if the SATA is disabled
SPORT	– sport=on, if the SPORT is enabled – sport=off, if the SPORT is disabled
TS	– ts=on, if the TS is enabled – ts=off, if the TS is disabled
AS GPIO	– ark_gpio=<on,off>:<nnnnnn,n=0,1,2> Some examples: – ark_gpio=on:110000, if the ARK_GPIO device driver is enabled with only the GPIOs group A and B enabled – ark_gpio=on:011221, if the ARK_GPIO device driver is enabled with AS GPIOs groups: – A disabled – B enabled – C enabled on the GPIO_GROUP 04 – D enabled on the GPIO_GROUP 10 – E enabled on the GPIO_GROUP 18 – F enabled on the GPIO_GROUP 13 – ark_gpio=off, if the AS GPIO is disabled

Table 4. Command line options for padmux configuration (continued)

Peripheral	Values
GP (ARM) GPIO	– arm_gpio1=on, if the GP gpio group 1 is enabled – arm_gpio2=on, if the GP gpio group 2 is enabled For the native Linux, they should be always on and they are set by default cmdline.
LINUX CONSOLE	– console=none if the Linux console is suppressed – console=<tty device>,<tty configuration> – console= ttyAMA0,115200n8, if the Linux console on ttyAMA0 with configuration: baudrate 115200 bps, flow control “none”, data size 8 bits (default) – console= ttyAMA1,115200n8, if the Linux console on ttyAMA1 (valid only if both UARTs to Linux) with default configuration.
SYSTEM (RTOS) CONSOLE	– rtosconsole=<uart port>,<uart configuration> – rtosconsole=uart1,115200n81 – rtosconsole=uart2,115200n81 (default) If not present no console is available on UARTs. In this case, the stpconsole application example can be used to access the RTOS console from the Linux.

Note: ***Important:*** *It is up to the user to ensure that no conflicting configurations are chosen. Normally this is taken care of in the board design, because the board has to be designed with a particular configuration option in mind and any conflicts must be resolved at the board design level. In the LSP support, only the devices supported by the board have to be enumerated in the manner described above. However, if there are any conflicting options chosen, the padmux initialization for the device mentioned last in the array is retained and the other device multiplexed with this particular device may not work.*

3.4 Clock framework

The clock framework defines programming interfaces to support software management of the system clock tree. This framework is widely used with system-on-chip (SOC) platforms to support various devices which may need custom clock rates. Note that these “clocks” don't relate to timekeeping or real-time clocks (RTC), each of which have separate frameworks.

3.4.1 Clock framework software overview

Clock framework support in the LSP is implemented around the Linux abstraction layer for clocks defined in “include/linux/clk.h”.

This abstraction only tends to declare and not define the APIs with their standard interfaces that must be implemented by the required platform. The platform is also expected to define the clock abstraction through “struct clk”. The struct clk is abstracted in “arch/arm/plat-streamplug/include/plat/clock.h”:

```
/**
   * struct clk - clock structure
   * @usage_count:  num of users who enabled this clock
   * @flags:  flags for clock properties
   * @rate:  programmed   clock rate in Hz
```

```
   * @en_reg:  clk enable/disable  reg
   * @en_reg_bit:  clk enable/disable  bit
   * @ops:  clk enable/disable  ops  - generic_clkops selected if NULL
   * @recalc:  pointer to clock rate recalculate function
   * @set_rate:  pointer to clock set rate function
   * @calc_rate:  pointer to clock get rate function for  index
   * @rate_config:  rate configuration information,  used  by  set_rate
   * @div_factor:  division factor to parent clock.
   * @pclk:  current  parent clk
   * @pclk_sel:  pointer to parent selection structure
   * @pclk_sel_shift:  register shift for selecting parent of this  clock
   * @children:  list for childrens or this  clock
   * @sibling:  node  for list of clocks having same parents
   * @private_data:  clock specific  private  data
   * @node:  list to maintain clocks linearly
   * @cl:  clocklook up  assoicated with this  clock
   * @dent:  object for  debugfs
   */
struct clk {
   unsigned int usage_count; unsigned int flags; unsigned long  rate;
unsigned int *en_reg;
   u8  en_reg_bit;
   const struct  clkops *ops;
   int (*recalc) (struct clk *clk,  unsigned long  *rate,
                                    unsigned long  prate);
   int (*set_rate)  (struct clk *clk,  unsigned long  rate); unsigned long
(*calc_rate)(struct  clk *, int index);
   struct rate_config rate_config;
   unsigned int div_factor;

   struct clk *pclk;
   struct pclk_sel *pclk_sel;
   unsigned int pclk_sel_shift;

   struct list_head children; struct list_head sibling; void *private_data;
   #ifdef  CONFIG_DEBUG_FS
   struct list_head node; struct  clk_lookup *cl; struct  dentry *dent;
   #endif
};
```

The following APIs are defined by the standard Linux CLK abstraction, which can be found in “include/linux/clk.h”:

- “clk_get” lookup and obtain a reference to a clock producer
- “clk_enable” inform the system when the clock source should be running
- “clk_disable” inform the system when the clock source is no longer required
- “clk_get_rate” obtain the current clock rate (in Hz) for a clock source. (This is only valid once the clock source has been enabled.)
- “clk_put” “free” the clock source
- “clk_round_rate” adjust a rate to the exact rate a clock can provide
- “clk_set_rate” set the clock rate for a clock source
- “clk_set_parent” set the parent clock source for this clock
- “clk_get_parent” get the parent clock source for this clock
- “clk_get_sys” get a clock based upon the device name
- “clk_add_alias” add a new clock alias

3.4.2 Clock framework kernel source and configuration

Above mentioned APIs and “struct clk” are implemented in following files:

- “arch/arm/plat-streamplug/include/plat/clock.h”
- “arch/arm/plat/streamplug/clock.c”

while the definitions and enumerations of each clock are defined in the following file: “arch/arm/mach-streamplug/clock.c”.

3.4.3 Clock framework internals

This section defines and provides an overview about the internal implementation of the STreamPlug clock framework. Users may not need to know all the details mentioned below. However, referring to this section can be beneficial to extend the clock framework or the rate tables in order to adapt to specific needs.

All of the clocks which can be used in the system are defined in: “arch/arm/mach-streamplug/clock.c”. For example the clk struct for the UART1 device is configured as follows:

```
/* uart1 clock */
static  struct clk uart1_clk =
{
   .flags  =  ENABLED_ONCE,
   .en_reg  =  (u32*  )( GetOffset (sMiscRegs,  low_speed_sub_clk_enb_reg)),
   .en_reg_bit  =  UART1_CLKENB_Pos,
   .pclk_sel  =  &uart_pclk_sel,
   .pclk_sel_shift  =  UART_CLKSEL_Pos,
   .recalc  =  &follow_parent,
};
```

Note that this clock object has:

- an enable/disable bit
- a multiple parent clock possible, defined through “uart1_pclk_sel”
- and that it just follows parent's clock rate without applying any divisor

In general a parent clock can be selected by writing corresponding “pclk_val to pclk_sel_reg” with a mask of “pclk_sel_mask “shifted by “pclk_sel_shift”. The user would not bother with these internal details because they are implementation details, the user may just call the “clk_set_parent” API. The “uart_synth_clk “and “pll3_usb_48m_clk” which are possible parents of the UART clock, can have their own parents and so on, in order to build the complete hierarchy.

An example to show the possible parents of the UART 1 is:

```
/* uart parent select structure  */
static struct pclk_sel uart_pclk_sel =
{
   .pclk_info  =  uart_pclk_info,
   .pclk_count  =  ARRAY_SIZE(uart_pclk_info),
   .pclk_sel_reg  =  (u32*  )( GetOffset (sMiscRegs,  gen_clk_cfg_reg)),
   .pclk_sel_mask  =  GEN_CLK_CFG_UART_CLKSEL_MASK,
};

static struct pclk_info uart_pclk_info[]  =
{
   {
     .pclk  =  &uart_synth_clk,
     .pclk_val  =  GEN_CLK_PLL1_VAL,
   },
   {
     .pclk  =  &pll3_usb_48m_clk,
     .pclk_val  =  GEN_CLK_PLL3_VAL,
   },
};
```

These structures define possible parents of the UART1 and how to select them, for example:

- “uart_synth_clk” can be selected by writing 1 onto the bit 4 “UART/UART2” of the “gen_clk_cfg_reg” miscellaneous register
- “pll3_usb_48m_clk” can be selected by writing 0 onto the bit 4 “UART/UART2” of the “gen_clk_cfg_reg” miscellaneous register

As an example to show one of the parents, “uart_synth_clk”:

```
/* uart synth clock */
static struct clk uart_synth_clk =
{
   .en_reg  =  (u32*  )( GetOffset (sMiscRegs,  uart_clk_synt_reg)),
   .en_reg_bit  =  SYNT_CLK_ENB_Pos,
   .pclk  =  &pll1_clk,
```

```
    .calc_rate  =  &aux_calc_rate,
    .recalc  =  &aux_clk_recalc,
    .set_rate  =  &aux_clk_set_rate,
    .rate_config  =  {aux_rtbl,  ARRAY_SIZE(aux_rtbl),  1},
    .private_data  =  &uart_synth_config,
};
```

Because this is a synthesizer which has divisors and multipliers, it can adapt to the required rate. For this, it defines its “calc_rate”, “recalc” and “set_rate” functions which can be used by the clock framework to set the desired clock rate. It also defines rate_config which actually points to a table (“aux_rtbl”) which contains entries of divisors values used to generate a given rate.

The “rate_config” is a simple structure defined as:

```
/**
    * struct rate_config - clk rate configurations
   * @tbls:  array of device specific  clk rate tables,  in  ascending order
of rates
    * @count:  size of tbls  array
    * @default_index:  default setting when  originally  disabled
    */
struct rate_config {
    void *tbls;
    u8  count;
    u8  default_index;
};
```

The “rate_config” points to a rate table (see example below) and a default index. The default index is used by the clock framework to fall back to a particular rate (pointed by index) when an invalid rate attempts to be programmed.

```
/* aux  rate configuration table,  in  ascending order of  rates  */
struct  aux_rate_tbl aux_rtbl[]  =
{
    /* For PLL1 = 332 MHz */
    {.xscale = 2, .yscale = 37, .eq = 0},/* 9  MHz */
    {.xscale = 1, .yscale = 8, .eq = 0},    /* 20.075 MHz */
    {.xscale = 1, .yscale = 4, .eq = 0},    /* 41.5 MHz */
    {.xscale = 1, .yscale = 2, .eq = 0},    /* 83  MHz */
    {.xscale = 1, .yscale = 1, .eq = 0},    /* 166  MHz */
};
```

Tables such as this may vary from the clock to clock for e.g.: the auxiliary synthesizer's table format is different from PLL to VCO and so on.

Using this rate table, the clock framework can program the synthesizer to generate desired frequency. The user can just simply call the “clk_set_rate” API.

In addition, please note that the entries in the above “aux_rtbl” should be sorted in ascending order according to the output clock rate generated assuming a constant parent clock rate.

3.4.4 Clock framework usage

The LSP tries to define and enumerate all possible clocks in the STreamPlug SoC. The following clocks are defined in: “arch/arm/mach-streamplug/clocks.c”:

```
/* array of all  streamplug 1x  clock  lookups */
static struct clk_lookup streamplug_clk_lookups[]  =  {
   /* root clks */
   { .con_id  =  "dummy_clk",.clk  =  &dummy_clk},
   { .con_id  =  "osc_32k_clk", .clk  =  &osc_32k_clk},
   { .con_id  =  "osc_24m_clk", .clk  =  &osc_24m_clk},
   /* clock  derived from 32  KHz osc  clk */
   { .dev_id  =  "rtc-streamplug", .con_id  =  "hclk",.clk  =  &rtc_clk},
   /* clock  derived from 24  MHz osc  clk */
   { .con_id  =  "pll1_clk", .clk  =  &pll1_clk},
   { .con_id  =  "pll2_clk", .clk  =  &pll2_clk},
   { .con_id  =  "pll3_usb_48m_clk",  .clk  =  &pll3_usb_48m_clk},
   { .dev_id  =  "wdt", .con_id  =  "hclk", .clk  =  &wdt_clk},
   /* clock derived from pll1  clk */
   { .con_id  =  "cpu_clk",.clk  =  &cpu_clk},
   { .con_id  =  "ahb_clk",.clk  =  &ahb_clk},
   { .con_id  =  "uart_synth_clk", .clk  =  &uart_synth_clk},
   { .con_id  =  "firda_synth_clk", .clk  =  &firda_synth_clk},
   { .con_id  =  "gpt0_synth_clk", .clk  =  &gpt0_synth_clk},
   { .con_id  =  "gpt1_synth_clk", .clk  =  &gpt1_synth_clk},
   { .con_id  =  "gpt2_synth_clk", .clk  =  &gpt2_synth_clk},
   { .dev_id  =  "uart1", .con_id  =  "hclk",.clk  =  &uart1_clk},
   { .dev_id  =  "uart2", .con_id  =  "hclk",.clk  =  &uart2_clk},
   { .dev_id  =  "dice_fir", .con_id  =  "hclk", .clk  =  &firda_clk},
   { .dev_id  =  "dice_ir",  .con_id  =  "hclk", .clk  =  &firda_clk},
   { .dev_id  =  "gpt0",  .con_id  =  "hclk", .clk  =  &gpt0_clk},
   { .dev_id  =  "gpt1",  .con_id  =  "hclk", .clk  =  &gpt1_clk},
   { .dev_id  =  "gpt2",  .con_id  =  "hclk", .clk  =  &gpt2_clk},
   /* clock derived from pll3  clk */
   { .dev_id  =  "designware_udc", .con_id  =  "hclk",.clk  =  &usbd_clk},
   { .con_id  =  "usbh_clk",.clk  =  &usbh_clk},
   /* clock  derived from ahb  clk */
   { .con_id  =  "ahbmult2_clk",  .clk  =  &ahbmult2_clk},
   { .con_id  =  "ddr_clk",.clk  =  &ddr_clk},
   { .con_id  =  "apb_lowsub_clk",.clk  =  &apb_lowsub_clk},
   { .con_id  =  "apb_bassub_clk",.clk  =  &apb_bassub_clk},
   { .con_id  =  "apb_appsub_clk",.clk  =  &apb_appsub_clk},
   { .con_id  =  "apb_armsub_clk",.clk  =  &apb_armsub_clk},
   { .dev_id  =  "i2c_designware.0",  .con_id  =  "hclk",.clk  =
&i2c_clk},
   { .dev_id  =  "dw_dmac", .con_id  =  "hclk",.clk  =  &dma_clk},
   { .dev_id  =  "jpeg-designware", .con_id  =  "hclk",.clk  =  &jpeg_clk},
```

```
    { .dev_id  =  "stmmaceth",.con_id  =  "ptp",  .clk  =  &ptp_clk},
    { .dev_id  =  "stmmaceth", .con_id  =  "eth",.clk  =  &mii_clk},
    { .dev_id  =  "stmmaceth",.con_id  =  "eth_phy",  .clk  =
&eth_phy_clk},
    { .dev_id  =  "stmmaceth", .con_id  =  "hclk",.clk  =  &gmac_clk},
    { .dev_id  =  "smi",.con_id  =  "hclk",.clk  =  &smi_clk},
    { .dev_id  =  "c3", .con_id  =  "hclk",.clk  =  &c3_clk},
    { .dev_id  =  "streamplug-sport",  .con_id  =  "hclk",.clk  =
&sport_clk},
    { .dev_id  =  "streamplug-ts.0", .con_id  =  "hclk",.clk  =  &ts_clk},
    /* clock  derived from apb  clk */
    { .dev_id  =  "ssp-pl022.0", .con_id  =  "hclk",.clk  =  &ssp_clk},
    { .dev_id  =  "streamplug_ark_gpio", .con_id  =  "hclk",.clk  =
&ark_gpio_clk},
    { .dev_id  =  "gpio1", .con_id  =  "hclk",.clk  =  &gpio1_clk},
    { .dev_id  =  "gpio2", .con_id  =  "hclk",.clk  =  &gpio2_clk},
    {.con_id  =  "clcd_synth_clk",.clk  =  &clcd_synth_clk},
    { .dev_id  =  "clcd",  .con_id  =  "clcdclk",  .clk  =  &clcd_clk},
    { .dev_id  =  "clcd",  .con_id  =  "hclk",  .clk  =  &amba_clcd_clk},
    {.con_id  =  "fsmc",.clk  =  &fsmc_clk},
    {.dev_id  =  "fsmc-nor",.clk  =  &fsmc_nor_clk},
    {.dev_id  =  "fsmc-sram",.clk  =  &fsmc_sram_clk},
    {.dev_id  =  "fsmc-nand",.clk  =  &fsmc_nand_clk},
    { .dev_id  =  "c_can_platform.1",  .clk  =  &can1_clk},
    { .dev_id  =  "c_can_platform.2",  .clk  =  &can2_clk},
    {.dev_id  =  "uport",.clk  =  &uport_clk},
    {.con_id  =  "dw_pcie",.clk  =  &pcie_clk},
    {.dev_id  =  "dw_pcie-rc",  .con_id  =  "hclk",  .clk  =  &pcie_rc_clk},
    {.dev_id  =  "dw_pcie-ep",  .con_id  =  "hclk",  .clk  =  &pcie_ep_clk},
    {.dev_id  =  "ahci",.clk  =  &sata_clk},
};
```

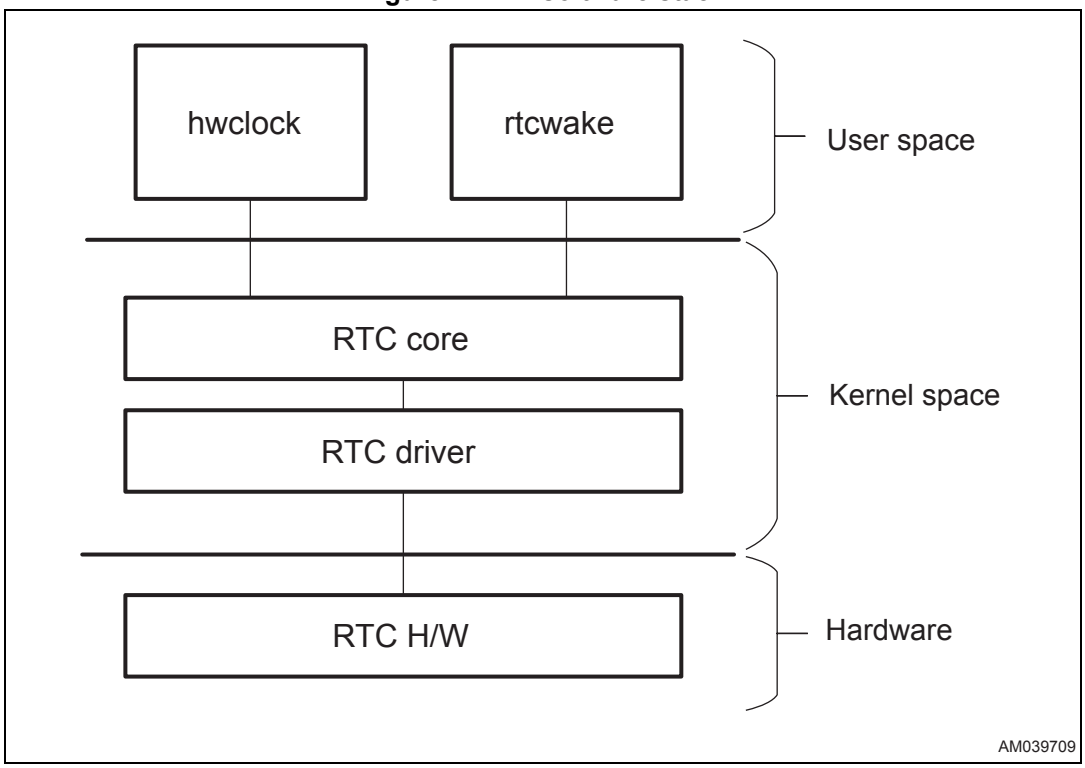
3.5 Real-time clocks (RTC)

The real-time clocks (RTC) is used to keep track of days, dates and time, including the century, year, month, hour, minutes and seconds.

3.5.1 RTC software overview

RTC support in the kernel is provided in the RTC framework. This is illustrated in *Figure 2*.

Figure 2. RTC software stack



3.5.2 RTC kernel source and configuration

The driver is implemented in “drivers/rtc/rtc-streamplug.c” and follows the Linux RTC class framework documented under: “linux-2.6/Documentation/rtc.txt”. *Table 5* lists the “Kconfig” options available for the RTC.

Table 5. RTC configurations

Configuration	Description
CONFIG_RTC_DRV_STREAMPLUG	Enables the STreamPlug RTC support

3.5.3 RTC platform configuration

There are no platform options available for this driver and the driver on its own functionally initializes the RTC hardware.

The RTC driver is enumerated in its respective CPU file “arch/arm/mach-streamplug/streamplug1x.c” which can be included by a corresponding board file to avail the support.

```
/* rtc device registration  */
static struct resource rtc_resources[]  =  {
   {
     .start  =  STREAMPLUG1X_ICM3_RTC_BASE,
     .end  =  STREAMPLUG1X_ICM3_RTC_BASE  +  SZ_4K - 1,
     .flags  =  IORESOURCE_MEM,
   }, {
     .start  =  STREAMPLUG1X_IRQ_BAS_SUBS_RTC,
     .flags  =  IORESOURCE_IRQ,
   },
};

struct platform_device  streamplug1x_rtc_device =  {
   .name  =  "rtc-streamplug",
   .id  =  -1,
   .num_resources  =  ARRAY_SIZE(rtc_resources),
   .resource  =  rtc_resources,
};
```

3.5.4 RTC usage

The hwclock is a shell utility for accessing the RTC clock. It is used to display the current time, set the hardware clock to a specified time, set the hardware clock to the system time, and set the system time from the hardware clock. The hwclock utility can be run periodically to insert or remove time from the hardware clock in order to compensate for a systematic drift (where the clock consistently gains or loses time at a certain rate if left to run).

```
# hwclock -h
hwclock: invalid option -- 'h'
BusyBox v1.18.4 (2015-01-27 15:55:04 CET) multi-call binary.

Usage: hwclock [-r|--show] [-s|--hctosys] [-w|--systohc] [-l|--localtime]
[-u|--utc] [-f FILE]

Query and set hardware clock (RTC)

Options:
        -r      Show hardware clock time
        -s      Set system time from hardware clock
        -w      Set hardware clock to system time
        -u      Hardware clock is in UTC
        -l      Hardware clock is in local time
```

```
        -f FILE Use specified device (e.g. /dev/rtc2)

# date 2015.02.02-03:34
Mon Feb  2 03:34:00 UTC 2015
# date 2015.02.02-15:34
Mon Feb  2 15:34:00 UTC 2015
# hwclock -r
Wed Dec 31 23:59:59 1969  0.000000 seconds
# hwclock -w
# hwclock -s
# reboot
...
# date
Mon Feb  2 15:36:26 UTC 2015
```

4 Communication drivers

The communication drivers provide the support to the STreamPlug interfaces like Ethernet, USB, I^2C and SPI.

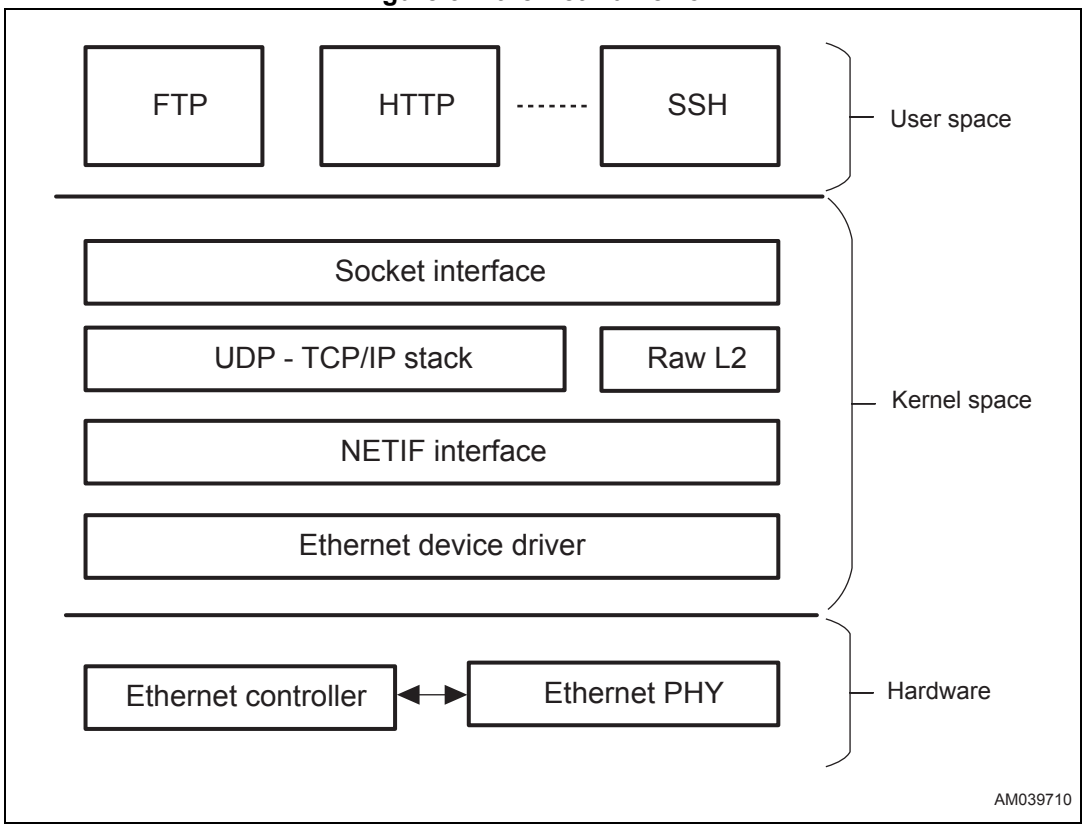
4.1 Gigabit media access controller (GMAC) - Ethernet

Ethernet is a family of standard technologies widely used in local area networks (LAN). The GMAC Ethernet controller is embedded into the STreamPlug architecture and it is hard configured to support only fast Ethernet. This section describes the STMMAC (GMAC) Ethernet driver.

4.1.1 GMAC software overview

The STMMAC Ethernet device driver sits on the top of the GMAC controller and interfaces with the Linux TCP/IP stack through the standard Linux network interface as shown in *Figure 3*.

Figure 3. Ethernet framework



The software overview section is broadly divided into two parts. The first part is more related to the STMMAC device driver core; while the second part is more related to the Ethernet PHY configurations that the user may need to handle through the driver.

STMMAC core

The following paragraphs cover the configurations that the user can do to the “stmmac” core at run-time, provide some more insights about transmission/reception data handling and give more information about key features of the driver.

Transmit process

The Xmit method is invoked when the kernel needs to transmit a packet. It sets the descriptors in the ring and informs the DMA engine that there is a packet ready to be transmitted. Once the controller has finished transmitting the packet, an interrupt is triggered so the driver will be able to release the socket buffers.

Receive process

When one or more packets are received, an interrupt is generated. The interrupts are not queued so the driver has to scan all the descriptors in the ring during the receive process. This is based on NAPI so the interrupt handler signals only if there is work to be done, and the related callback is scheduled at some future point. The incoming packets are stored, by the internal DMA controller, in a list of pre-allocated socket buffers in order to avoid using the memcpy.

PHY interface

The following paragraphs capture few tips and tricks that developers can use when porting to a new Ethernet physical device.

PHY abstraction layer

The physical abstraction layer provides a unified interface to a number of different physical layer (PHY) engines. For more information please see “Documentation/networking/phy.txt” within the kernel source tree.

PHY device driver

With the PHY abstraction layer, adding support for new PHYs is quite easy. In some cases, no work is required at all because a generic PHY driver is already provided.

The generic driver, by default, works without any interrupts (polling mode). This means that a timer will be used to periodically communicate between the PHY and the STMMAC in order to check the link status.

PHY platform setup

The driver setup information comes from specific platform structures. For example, “streamplug_devel_board.c”, as shown:

```
/* ethernet phy  device */
static struct plat_stmmacphy_data phy_private_data  =  {
   .bus_id  =  0,
   .phy_addr  =  -1,
   .phy_mask  =  0,
   .interface  =  PHY_INTERFACE_MODE_MII,
};

static struct resource  phy_resources =  {
   .name  =  "phyirq",
   .start  =  -1,
```

```
    .end  =  -1,
    .flags  =  IORESOURCE_IRQ,
};

struct  platform_device streamplug10_phy_device =  {
    .name  =  "stmmacphy",
    .id  =  -1,
    .num_resources  =  1,
    .resource  =  &phy_resources,
    .dev.platform_data  =  &phy_private_data,
};
```

4.1.2 GMAC kernel source and configuration

Table 6 lists the “Kconfig” options available for Ethernet:

Table 6. STreamPlug STMMAC configurations

Configuration	Description
CONFIG_NET	It enables networking support
CONFIG_NETDEVICES	It enables network device support
CONFIG_STMMAC_ETH	It enables STreamPlug Ethernet (Synopsys) support

The kernel sources related to Ethernet driver implementation are present in the “drivers/net/stmmac/” folder, which is spread across following files:

- “dwmac1000_core.c” carries GMAC core initializations
- “dwmac1000_dma.c” carries GMAC DMA initializations
- “dwmac100_core.c” carries MAC core initializations
- “dwmac100_dma.c” carries MAC DMA initializations
- “dwmac_lib.c” generic utility functions
- “enh_desc.c” enhanced descriptors handlers
- “norm_desc.c” normal descriptors handlers
- “stmmac_ethtool.c” ethtool support implementation
- “stmmac_main.c” main driver implementation
- “stmmac_mdio.c” MDIO implementation to access the PHY interface

4.1.3 GMAC platform configuration

The Ethernet driver expects several pieces of information from the platform. Refer to the driver's header file in “include/linux directory”. The data structure below (included into “arch/arm/mach-straemplug/streamplug1x.c”) provides the platform data passed to the STMMAC driver.

```
/* Ethernet device registration */
struct plat_stmmacenet_data ether_platform_data =  {
   .bus_id  =  0,
   .has_revmii  =  ETH_REVMII_ADDRESS,
   .has_gmac  =  1,
   .enh_desc  =  1,
   .pbl  =  8,
   .dev_addr  =  "00:80:e1:26:0a:5b",
};
```

while the data structure below, included into the files specific for each supported board, provides the details of the PHY specific data passed by the platform to the driver.

```
/* ethernet phy  device */
static struct plat_stmmacphy_data phy_private_data  =  {
   .bus_id  =  0,
   .phy_addr  =  -1,
   .phy_mask  =  0,
   .interface  =  PHY_INTERFACE_MODE_MII,
};
```

4.1.4 GMAC usage

The following Linux commands can be used to configure the Ethernet interface:

ifconfig

The “ifconfig” command allows the operating system to setup the network interfaces and the user to view information about the configured interfaces.

To configure the network IP address the following command should be used:

```
$  ifconfig eth0 192.168.1.1  netmask 255.255.255.0
```

The status of the configuration can be obtained with:

```
$  ifconfig  eth0

eth0Link encap:EthernetHWaddr 08:00:27:bd:c6:6e
inet  addr:192.168.1.1 Bcast:192.168.1.255 Mask:255.255.255.0
   UP BROADCAST  RUNNING  MULTICASTMTU:1500Metric:1
   RX Packets:0 errors:0  dropped:0 overruns:0 frame:0
   TX packets:0 errors:0  dropped:0 overruns:0 carrier:0 collisions:0
txqueuelen:32
   RX bytes:0 (0.0 B)TX bytes:  0  (0.0 B)
```

This shuts down the interface and reactivates it:

```
$  ifconfig  eth0 down
$  ifconfig eth0  up
```

To configure the MTU size:

```
$  ifconfig  eth0 down
$  ifconfig  eth0 mtu  <size>
$  ifconfig eth0  up
```

ethtool

The “ethtool” utility is used to display or change the Ethernet card settings. To setup the auto negotiation use:

```
$  ethtool -s  eth0  autoneg  on
```

To check the existing network configurations:

```
$  ethtool  eth0
```

To setup the forced speed 100, the full duplex mode (default):

```
$  ethtool -s  eth0  autoneg off speed  100  duplex full
```

To setup the forced speed 100, the half duplex mode:

```
$  ethtool -s  eth0  autoneg off speed  100  duplex half
```

To setup the forced speed 10, the full duplex mode:

```
$  ethtool -s  eth0  autoneg off speed  10  duplex full
```

4.2 Universal serial bus (USB) host

The universal serial bus (USB) is an industry standard to connect computers and electronic devices.

Linux provides two host control drivers (Linux EHCI and Linux OHCI). The architecture driver plugs into the USB host stack and allocates the basic resources for the USB host controller. The host side drivers for USB devices talk to the “usbcore” APIs. There are standard details of the API available. The details of the USB host APIs could be found online at the following address: www.Kernel.org/doc/htmldocs/usb.html.

4.2.1 USB host kernel source and configuration

To ensure proper USB device support some of the options in the kernel need to be enabled. *Table 7* lists the configuration options.

Examples in this document show configuration options for basic USB support as well as the commonly needed options, such as an USB mass storage device (most cameras and USB Thumb® drives).

Table 7. USB host configurations

Configuration	Description
CONFIG_USB_SUPPORT	This option adds core support for the USB bus.
CONFIG_USB	Enable this option if the system has the host side bus and will use USB devices. Also see the USB devices in “/proc/bus/usb”. Enabling this option is recommended.
CONFIG_USB_DEVICES	If this option is enabled, it will get a “file/proc/bus/usb/devices” which lists the devices currently connected to your USB bus or buses, and a file named “/proc/bus/usb/xxx/yyy” for every connected device, where *xxx* is the bus number and *yyy* the device number.
CONFIG_USB_EHCI_HCD	Enable this option to configure the host controller driver to support the USB2. EHCI is standard for USB 2.0 high speed host control hardware.
CONFIG_USB_OHCI_HCD	Enable this option to configure the USB host controller hardware for the OHCI specification. The OHCI is the standard for accessing USB 1.1 host controller hardware.
CONFIG_USB_STORAGE	Enable this option to connect a USB mass storage device to the host USB port. The option depends on SCSI support being enabled.
CONFIG_SCSI	Enable this option to use an SCSI hard disk, an SCSI tape drive, an SCSI CD-ROM or any other SCSI device under Linux. USB mass storage devices follow SCSI protocol, and hence this option should be enabled over USB mass storage devices.
CONFIG_USB_ACM	This driver supports USB modems and ISDN adapters which support the communication device class abstract control model interface.
CONFIG_NET	Required for enabling USB modem support
CONFIG_USB_USBNET	Multi-purpose USB networking framework
CONFIG_USB_NET_CDCETHER	This option supports devices conforming to the communication device class (CDC) Ethernet control model.
CONFIG_HID_SUPPORT	Options for various computer human interface device drivers.
CONFIG_HID	This option compiles into kernel the generic HID layer code (parser, usages, etc.), which can then be used by transport-specific HID implementation (like USB or Bluetooth®).

Using the following command it is possible to configure the Linux kernel:

```
make menuconfig

Device Drivers--->
SCSI  device support--->
(Although  SCSI  will be  enabled automatically  when  selecting USB Mass
Storage,we  need to enable disk support.)
---SCSI  support type (disk,  tape,  CD-ROM)
<*>SCSI  disk support
(Then  Move a  Level Back  and  Go into USB Support)
USB support--->
(This  is the root hub  and  is required for USB support.  If you'd  like to
compile this as  a  module,  it will be  called usbcore.)

<*>  Support for  Host-side  USB
(Enable  this option if your  system has  the host side bus  and  wants to
use  USB devices
and  also to see  your USB devices in /proc/bus/usb.  This is recommended.)

[*]USB device filesystem
(Select  at least one  of the HCDs.  If you  are unsure,  picking all  is
fine.)
--- USB Host   Controller Drivers
<*>  EHCI  HCD  (USB  2.0) support
<*>  OHCI HCD  support
(Moving  a  little further down,  we come to CDC  and  mass  storage.)
<*>  USB Modem (CDC  ACM)  support
<>  USB Printer  support
<*>  USB Mass  Storage support
```

For use with a USB keyboard, mouse, joystick, or any other input device, enable HID support. Go back one level to “Device drivers” and enable HID support as shown:

```
Device Drivers  --->
       [*]  HID  Devices --->
         <*>USB Human Interface  Device (full HID)  support
```

For use with a USB modem, enable the USB modem (CDC ACM) support as shown above along with the following supports:

```
Device Drivers  ---->
      [*]  Network device  support--->
         USB Network Adapters--->
           <*>  Multi-Purpose  USB Networking  Framework
```

4.2.2 USB host platform configuration

The USB host device driver has the following platform configuration:

```
/* usb  host device registration  */
static struct resource ehci_resources[]  =  {
   [0] =  {
      .start  =  STREAMPLUG1X_ICM4_USB_EHCI_BASE,
      .end  =  STREAMPLUG1X_ICM4_USB_EHCI_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   },
    [1] =  {
      .start  =  STREAMPLUG1X_IRQ_HIGH_SPEED_SUBS_USB_EHCI,
      .flags  =  IORESOURCE_IRQ,
   },
};

static struct resource ohci_resources[]  =  {
    [0] =  {
      .start  =  STREAMPLUG1X_ICM4_USB_OHCI_BASE,
      .end  =  STREAMPLUG1X_ICM4_USB_OHCI_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   },
    [1] =  {
      .start  =  STREAMPLUG1X_IRQ_HIGH_SPEED_SUBS_USB_OHCI,
      .flags  =  IORESOURCE_IRQ,
   },
};

static u64  ehci_dmamask  =  ~0;
static int usbh_id =  -1;

struct platform_device  streamplug1x_ehci_device =  {
   .name  =  "streamplug-ehci"
   .id  =  -1,
   .dev  =  {
      .coherent_dma_mask  =  ~0,
      .dma_mask  =  &ehci_dmamask,
      .platform_data  =  &usbh_id,
   },
   .num_resources  =  ARRAY_SIZE(ehci_resources),
   .resource  =  ehci_resources,
};

static u64  ohci_dmamask  =  ~0;
```

```
struct platform_device  streamplug1x_ohci_device =  {
   .name  =  "streamplug-ohci",
   .id  =  0,
   .dev  =  {
      .coherent_dma_mask  =  ~0,
      .dma_mask  =  &ohci_dmamask,
      .platform_data  =  &usbh_id,
   },
   .num_resources  =  ARRAY_SIZE(ohci_resources),
   .resource  =  ohci_resources,
};
```

4.2.3 USB host usage

A USB device can either use a custom driver or use one already present in the system. This is based on the concept of a device class and means that if a device belongs to a certain class, then the other devices of the same class can make use of the same device driver. Some of these classes are: the USB HID (human interface devices) class which includes input devices like keyboards and mice, the USB mass storage devices class which includes devices like pen drives, digital cameras, audio players, etc. and the USB CDC (communication devices class) which essentially includes USB modems and similar devices.

To enable the USB host support at run-time it is necessary to configure the Linux kernel command line using the options mentioned in *Table 4 on page 22*.

USB mass storage class

The USB mass storage standard provides an interface to a variety of storage devices, like hard disk drives and Flash memories. Plug-in the Flash memory into the available USB port and then type the following commands:

```
#  dmesg  | less
usb 1-1: new high speed USB device using streamplug-ehci and address 2
scsi0 : usb-storage 1-1:1.0
scsi 0:0:0:0: Direct-Access             USB DISK 2.0     PMAP PQ: 0 ANSI: 4
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] 15124992 512-byte logical blocks: (7.74 GB/7.21 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Mode Sense: 23 00 00 00
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Attached SCSI removable disk

#  mount  /dev/sda1  /mnt/

#  df -h
```

```
Filesystem                Size      Used Available Use% Mounted on
ubi0_0                    7.8M      3.5M      4.3M  45% /
tmpfs                    51.0M     32.0K     50.9M   0% /tmp
/dev/sda1                 7.2G      5.8G      1.4G  80% /mnt

/* Digital cameras can  be  accessed the  same way  as  memory  sticks.  */
```

The device is picked up as a USB 1.1 device and allocates an address. It also indicated which HCD is used.

USB communication device class (CDC)

The USB CDC class supports a lot of communication devices, including Ethernet. Compile and then boot-up the kernel with the options relevant to the USB Ethernet adapters enabled. The options are covered in the following configuration sections. Plug-in the USB Ethernet adapter, and console messages that are similar to the following will be displayed.

```
$hub  1-0:1.0: over-current  change   on  prot  1
usb  1-1:new high  speed   USB device using streamplug-ehci  and  address  2
usb-1.1: configuration #1  chosen from  1  choice
eth0:register  'asix'  at usb-STreamPlug  EHCI-1,  ASIX  AX88772  USB2.0
Ethernet,00;89:c8:3a:4c:0b
/* Type  in the following command and  check if the device has  been
recognized */
$  cat /proc/bus/usb/devices

 T:  Bus=01  Lev=01   Prnt=01 Port=00 Cnt=01   Dev#=  2  Spd=480  MxCh= 0
 D:  Ver=   2.00 Cls=ff(vendor)Sub=ff  Prot=00 MxPS= 64  #Cfgs=  1
 P:  Vendor=2001 ProdID=3c05  Rev=  0.01
 S:  Manufacturer= D-Link  Corporation
 S:  Product=DUB-E100
 S:  Serial Number=000001
 C:*  #Ifs=  1  Cfg#=   1  Atr=80 MxPwr=250mA
 I: If#=  0  Alt=  0  #EPs=  3  Cls=ff(vendor  specific) Sub=ff  Prot=00
Driver=asix
 E:  Ad=82(I)  Atr=03(Int.)  MxPS= 8  Ivl=128ms E:  Ad=81(I)  Atr=02(Bulk)
MxPS= 512  Ivl=0ms
 E:  Ad=03(O)  Atr=02(Bulk)  MxPS= 512  Ivl=0ms

/* The  functionality  could be  checked by  assigning the IP and  then test
a  simple ping operation.  */
$  ifconfig  eth0 192.168.1.11
eth0:  link up,  100Mbps,  full duplex,  lpa 0xcde1
th0:  linkup,  100Mbps,  full-duplex,  lpa 0xcde1
```

USB human interface device (HID) class

The USB HID class describes human interface devices such as keyboards and mice.

USB mouse

Compile and then boot-up the kernel with the options relevant to the USB mouse enabled. The options are shown in the configuration following paragraphs. Plug-in the USB mouse. Print output messages that are similar to the following will be displayed:

```
$ hub 1-0:1.0: over-current change on prot 1
usb 3-1:new low speed USB device using streamplug-ohci and address 3
usb-3.1: configuration #1 chosen from 1 choice
input: USB Optical Mouse as /class/input/input2
input: USB HID v1.11 Mouse[USB Optical Mouse] on usb-streamplug-ohci.0-
1
```

The following command can be used to check if the device has been recognized:

```
$ cat /proc/bus/usb/devices
T: Bus=03 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 3 Spd=1.5 MxCh= 0
D: Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1
P: Vendor=0461 ProdID=4d15 Rev= 2.00
S: Product=USB Optical Mouse
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=03(HID ) Sub=01 Prot=02 Driver=usbhid
E: Ad=81(I) Atr=03(Int.) MxPS=4 Ivl=10ms
```

USB keyboard

Compile and then boot-up the kernel with the options relevant to the USB keyboard enabled. The options are included in the following configuration paragraphs. Plug-in the USB keyboard. Print output messages that are similar to the following will be displayed.

```
$ hub 1-0:1.0: over-current change on prot 1
usb 3-1:new full speed USB device using streamplug-ohci and address 4
usb-3.1: configuration #1 chosen from 1 choice
input: Dell Dell Smart Card Reader Keyboard as /class/input/input3
input: USB HID v1.11 Keyboard [Dell Dell Smart Card Reader Keyoard]
on usb- streamplug-ohci.0-1

$cat /proc/bus/usb/devices
T: Bus=03 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 4 Spd=12 MxCh= 0
D: Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1
P: Vendor=413c ProdID=2101 Rev= 1.00
S: Manufacturer=Dell
S: Product=Dell Smart Card Reader Keyboard
C:* #Ifs= 2 Cfg#= 1 Atr=a0 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=03(HID ) Sub=01 Prot=01 Driver=usbhid
E: Ad=81(I) Atr=03(Int.) MxPS=8 Ivl=24ms
I:* If#= 1 Alt= 0 #EPs= 3 Cls=0b(scard) Sub=00 Prot=00
Driver=(none)
E: Ad=02(O) Atr=02(Bulk) MxPS=64 Ivl=0ms
E: Ad=82(I) Atr=02(Bulk) MxPS=64 Ivl=0ms
E: Ad=83(I) Atr=03(Int.) MxPS=8 Ivl=24ms
```

A simple method to verify if the STreamPlug chip is working as a USB host is to plug a USB mass storage device that has been formatted as FAT32, into the USB socket and verify from the kernel debug shell that USB signals are exchanged between the host and external device.

A portion of the Linux kernel log generated after the USB host device driver recognized the USB device is shown below.

```
usb 1-1: new high speed USB device using streamplug-ehci and address 2
scsi0 : usb-storage 1-1:1.0
scsi 0:0:0:0: Direct-AccessVBTMStore 'n' Go5.00 PQ: 0 ANSI: 0 CCS
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] 4028416 512-byte logical blocks: (2.06 GB/1.92 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through sd 0:0:0:0: [sda]
Assuming drive cache: write through
   sda:
sd 0:0:0:0: [sda] Assuming drive cache: write through sd 0:0:0:0: [sda]
Attached SCSI removable disk
```

After the USB device is recognized, the user may use the command mount to mount the filesystem (/dev/sd##) in the “folder /mnt”.

```
$ mount /dev/sda1 /mnt
```

Going into “/mnt” will show the filesystem which is on “/dev/sda1”:

```
$ ls /mnt
```

4.3 Universal serial bus (USB) device

The STreamPlug USB chip controller may run either as a USB host (master) or as a USB device (slave) and the mutually exclusive selection is done during the startup of the board.

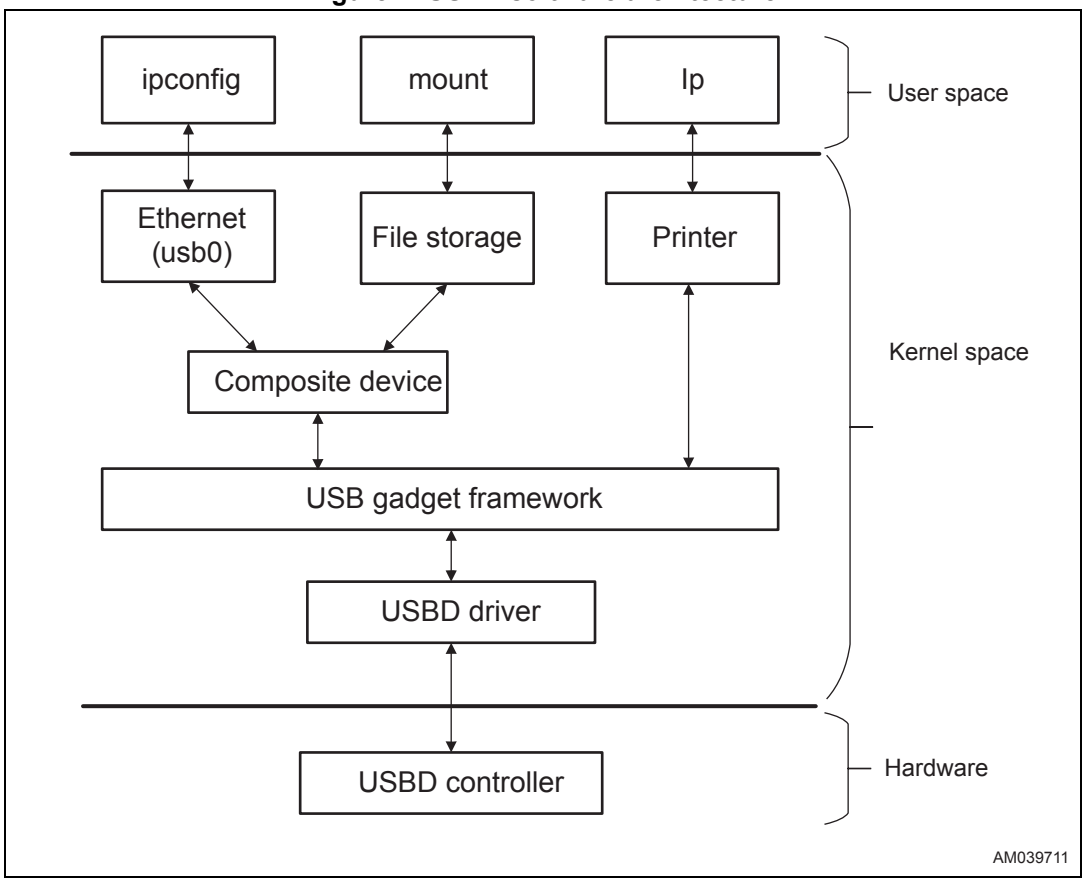
There is a wide variety of USB devices (USBD) available in the market. Examples of these devices are USB Ethernet adapters, USB audio devices, USB mass storage devices, USB printers, and so on. In the Linux USB world, these functions are called “gadgets”. The STreamPlug USBD can be used to build any of these functions. A device with multiple functions can also be built. Multifunctional printers, USB Ethernet plus mass storage are examples. These devices are generally known as “composite” devices.

4.3.1 USB device software overview

The USB device controller driver in the STreamPlug LSP supports the Linux USB gadget framework. This framework provides a flexible and easy interface for adding different USB slave devices. It also offers the facility to easily add multifunction USB composite devices.

Figure 4 shows the USB gadget framework most important components.

Figure 4. USBD software architecture



As shown in *Figure 4*, the gadget drivers can access the USB device driver either directly through the gadget framework or through the composite layer. The composite layer provides an interface where multifunctional devices (like audio and video) can be easily supported. It is preferable that USB gadget drivers which do not have composite features also interact through the composite layer. Please note that only one gadget driver at a time can exist in this framework using the gadget framework. Also remember that the composite layer is in an itself gadget drive. Therefore according to *Figure 4*, the printer and the composite layer cannot exist at the same time. One possibility is to build the printer gadget over the composite layer.

The remaining part of this document describes the composite layer interface. For detailed documentation on the gadget framework please refer to: www.linux-usb.org/gadget/.

4.3.2 USB device kernel source and configuration

The following files contain some of the source code part of the STreamPlug device driver for the USB gadget controller:

`arch/arm/mach-streamplug/ipswrst_ctrl.c`
`arch/arm/mach-streamplug/include/mach/generic.h`
`arch/arm/mach-streamplug/include/mach/streamplug10.h`
`arch/arm/mach-streamplug/clock.c`
`arch/arm/mach-streamplug/padmux.c`
`arch/arm/mach-streamplug/streamplug1x.c`
`drivers/usb/gadget/designware_udc.h`
`drivers/usb/gadget/designware_udc.c`
`drivers/usb/gadget/inode.c`
`drivers/usb/gadget/zero.c.`

The USB gadgets device driver is built into the following modules:

`g_zero.ko`
`g_ether.ko`
`gadgetfs.ko`

4.3.3 USB device platform configuration

A partial list of Linux kernel configuration options useful to configure the USB controller is present in *Table 8*.

Table 8. USB gadget Linux kernel configuration

Configuration	Description
CONFIG_USB	-
CONFIG_USB_SUPPORT	-
CONFIG_USB_DEVICE_CLASS	-
CONFIG_USB_GADGET	This enables USB gadget support in Linux kernel
CONFIG_USB_GADGETFS	-
CONFIG_USB_ZERO	This enables a test gadget driver (“zero”)
CONFIG_USB_ETH	-
CONFIG_USB_GADGET_DESIGNWARE	This enables STreamPlug USB device controller support
CONFIG_USB_DESIGNWARE	-
CONFIG_USB_TEST	This enables the USB test module for testing the zero gadget on the host side.
CONFIG_USB_GADGET_DUALSPEED	This enables dual (FULL and HIGH) speed support.

Other may be necessary and/or more fine-grained configurations may be needed by directly changing the source code. One of these cases is related to the FIFO configurations. The RxFIFO on the STreamPlug USBD can be configured for each endpoint. Keep in mind that total combined RxFIFO usage for all out endpoints should not exceed 2 KB. Similarly, total combined TxFIFO usage for all IN endpoints should be limited to 2 KB. To change this FIFO configuration, it is possible to edit the corresponding macro in the source code file.

An example is the configuration of the buffer length. The gadget drivers allocate a USB request and then submit it to the framework for transfer. The length of such transfer requests will determine the performance of the driver. Allocating a large buffer and hence a bigger buffer length will make CPU more free. The USB DMA would try to complete the transfer for the asked length and then interrupt CPU notifying the completion of the transfer.

The maximum buffer length is limited to 65535 bytes for an STreamPlug USB device.

USBD driver interface with Linux gadget layer

As mentioned above, the USB device controller driver supports the Linux gadget framework. For this, it exports certain device, endpoint specific routines and two functions for registering and unregistering to the framework.

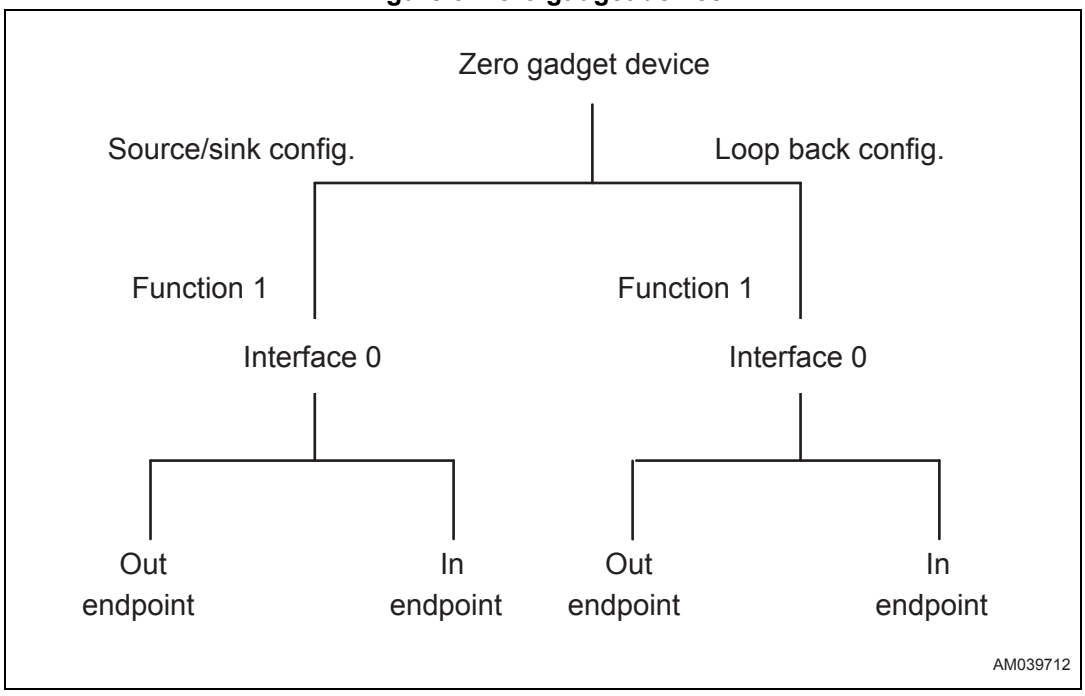
```
/* device specific  operations exported by  usbd  driver */
static  const struct usb_gadget_ops  dw_udc_dev_ops =  {
   .get_frame  =  dw_dev_get_frame,
   .wakeup  =  dw_dev_wakeup,
   .set_selfpowered  =  dw_set_selfpowered,
   .ioctl =  dw_ioctl,
};
/* endpoint specific  operations exported by  usbd  driver */
static struct usb_ep_ops  dw_udc_ep_ops =  {
   .enable  =  dw_ep_enable,
   .disable  =  dw_ep_disable,
   .alloc_request  =  dw_ep_alloc_request,
   .free_request  =  dw_ep_free_request,
   .queue  =  dw_ep_queue,
   .dequeue  =  dw_ep_dequeue,
   .set_halt  =  dw_ep_set_halt,
   .fifo_status  =  dw_ep_fifo_status,
   .fifo_flush  =  dw_ep_fifo_flush,
};
/* routine  exported by  usbd  driver for gadgets to register */
int usb_gadget_register_driver(struct  usb_gadget_driver *driver);

/* routine  exported by  usbd  driver for gadgets to un-register  */
int usb_gadget_unregister_driver(struct  usb_gadget_driver *driver);
```

The composite device layer registers to the gadget framework by calling the above APIs and exposes an interface which can be used by different functions (gadgets) to represent a composite device.

4.3.4 USB device usage

The composite device is designed in a such way that, the driver should first register to the composite layer. During registration, it passes some of the device related details (e.g.: device, string descriptor) to the composite layer. After registering, the composite device needs to add a configuration (multiple configurations are also possible) and then individual functions can add their interfaces. *Figure 5* shows a simple gadget driver (“zero gadget”) available with the STreamPlug LSP.

Figure 5. Zero gadget device



The illustrated gadget driver is built over a composite layer (although it is not a composite device) and is mainly used for testing the USB device controller. It provides two configurations: the first one has a source/sink function for generating/consuming USB packets, and the second one has a loop back feature. This example gadget driver is referred to the explanations given throughout this part of the document.

This driver can be found in the file:

`linux/drivers/usb/gadget/zero.c.`

Registering to the composite device

The un/registration to the composite device can be done with the following calls:

```
/* usb  composite gadget need  to fill following structure */
static struct usb_composite_driver zero_driver  =  {
   .name  =  "zero";
   .dev  =  &device_desc;
   .strings  =  dev_strings;
   .bind  =  zero_bind;  /* callback called on  successful registration */
   .unbind= zero_unbind,
   .suspend= zero_suspend,
   .resume= zero_resume,
};
/* Following are the APIs   for register/un-register  */
usb_composite_register(&zero_driver);
usb_composite_unregister(&zero_driver);
```

Adding configuration

Any composite device can have multiple configurations with multiple interfaces, each interface (or a group of interfaces) representing a unique function. The following API can be used to add a configuration:

```
static struct usb_configuration sourcesink_driver =  {
   .label  =  "source/sink",
   .strings  =  sourcesink_strings,
   .bind=  sourcesink_bind_config,  /* callback called during registration
to finish other configurations */
   .setup  =  sourcesink_setup,  /* callback to handle control  requests */
   .bConfigurationValue  =  3,
   .bmAttributes  =  USB_CONFIG_ATT_SELFPOWER,
   };

/* following function registered earlier,  is called during registration */
static int    init zero_bind(struct  usb_composite_dev *cdev)
{
   ...
   usb_add_config(cdev,  &sourcesink_driver);
   ...
}
```

Adding function

After adding configurations, the functions supported in each configuration need to be defined. Several functions as per the composite device design, can be added. The following mechanism can be used to add functions to configurations:

```
/* Following function registered earlier is called during registration */
static int sourcesink_bind_config(struct  usb_configuration *c)
{
   struct f_sourcesink*ss;
   intstatus;

   ss  =  kzalloc(sizeof  *ss,  GFP_KERNEL);
   if (!ss)
      return -ENOMEM;

   ss->function.name  =  "source/sink";
   ss->function.descriptors  =  fs_source_sink_descs;
   ss->function.bind  =  sourcesink_bind;
   ss->function.unbind  =  sourcesink_unbind;
   ss->function.set_alt = sourcesink_set_alt;
   ss->function.disable = sourcesink_disable;

   status =  usb_add_function(c,  &ss->function);
   if (status)
      kfree(ss);
   return status;
}
```

Initializing USB descriptors

There are some fields in standard USB descriptors that require inputs from the composite layer for initialization. In almost all descriptors some of these fields are indexed to string tables and to an interface number for the interface descriptors. Some helper routines are described below:

```
static int zero_bind(struct  usb_composite_dev *cdev)
{
   /* get next available string  index */
   id =  usb_string_id(cdev);
   if (id <  0)
   return id;
   strings_dev[STRING_MANUFACTURER_IDX].id  =  id;
   device_desc.iManufacturer  =  id;
   ...
}
static int sourcesink_bind(struct  usb_configuration *c,  struct
usb_function *f)
{
   ...
```

```
    /* allocate interface ID(s)  */
    id =  usb_interface_id(c,  f);
    if (id <  0)
    return id;
    source_sink_intf.bInterfaceNumber  =  id;
    ...
}
```

Data and control transfer

After completing the registering process, the gadget driver can handle setup requests through setup callbacks. In this way, other required endpoints can be configured and a transfer (of control or data) through the Linux gadget framework APIs can be initiated. More details on these APIs can be found in the references. *Table 9* summarizes these APIs and their purpose.

Table 9. Linux gadget framework API

Function	Description
struct usb_ep *usb_ep_autoconfig(struct usb_gadget *,struct usb_endpoint descriptor *)	Allocates a suitable free endpoint described by struct usb_endpoint_descriptor.
int usb_ep_enable(struct usb_ep *ep, const struct usb_endpoint_descriptor *desc)	Enables the endpoint ep, in order to be used for data transfer. The endpoint ep is described in struct usb_endpoint_descriptor.
struct usb_request *usb_ep_alloc_request(struct usb_ep *ep, gfp_t gfp_flags)	Allocates a request for USB transfer.
void usb_ep_free_request(struct usb_ep *ep, struct usb_request *req)	Frees the allocated request.
int usb_ep_disable(struct usb_ep *ep)	Disables the endpoint ep, so that it is not usable.
int usb_ep_queue(struct usb_ep *ep, struct usb_request *req, gfp_t gfp_flags)	Submits a transfer request on this endpoint (ep).
int usb_ep_set_halt(struct usb_ep *ep)	Halts a particular endpoint (ep).

USBD control

The APIs in *Table 10* may be used to configure and program the USB device.

Table 10. USB device control APIs

Function	Description
int usb_gadget_frame_number (struct usb_gadget *gadget)	Returns the current start of the frame number int usb_gadget_wakeup (struct usb_gadget enables the remote wakeup feature of USB *gadget) device.
int usb_gadget_set_selfpowered (struct usb_gadget *gadget)	The USB device is self-powered.
int usb_gadget_clear_selfpowered (struct usb_gadget *gadget)	The USB device is not self-powered but bus powered.
int usb_gadget_ioctl (struct usb_gadget *,unsigned code, unsigned long param)	Configures the USB device on configuration change. This API is STreamPlug specific and is mandatory to call on SET CONFIGURATION as it programs the controller accordingly “param” points to the function descriptors.

4.3.5 USB platform configuration

The STreamPlug USB gadget device driver manages the resources shown in the following code:

```
/* usb  device registration  */
static struct resource udc_resources[]  =  {
    [0] =  {
      .start  =  STREAMPLUG1X_ICM4_USBD_CSR_BASE,
      .end  =  STREAMPLUG1X_ICM4_USBD_CSR_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   },
    [1] =  {
      .start  =  STREAMPLUG1X_ICM4_USB_PLDT_BASE,
      .end  =  STREAMPLUG1X_ICM4_USB_PLDT_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   },
    [3] =  {
      .start  =  STREAMPLUG1X_IRQ_HIGH_SPEED_SUBS_USB_DEV,
      .end  =  STREAMPLUG1X_IRQ_HIGH_SPEED_SUBS_USB_DEV,
      .flags  =  IORESOURCE_IRQ,
   },
};

struct  platform_device streamplug1x_udc_device =  {
   .name  =  "designware_udc",
   .id  =  -1,
   .dev  =  {
```

```
        .coherent_dma_mask  =  0xffffffff,
    },
    .num_resources  =  ARRAY_SIZE(udc_resources),
    .resource  =  udc_resources,
};
```

4.3.6 USB platform usage

As shown above, there can be various user defined functions over the Linux gadget framework. Each of the functions (gadgets) exposes its own interface. For example, the USB Ethernet function exposes a netdev interface; the USB serial gadget exposes a tty interface and so on. This makes the usage of the USB gadgets very easy. Standard tools can be used for standard interfaces provided by these gadgets. The following files present in the Linux kernel documentation folder provide some usage examples:

```
* gadget_printer.txt  for usb  printer  device
* gadget_serial.txt  for usb  serial  device
```

The STreamPlug LSP provides a test gadget driver, “zero gadget”, to test the USB device controller. This gadget does not have any user interface. It just provides two configurations, “source and sink” and “loop back”, to support several test cases which can be executed from the USB host side. On the USB host corresponding to the zero gadget the “usbtest” driver supports several test cases to validate the USB through IOCTLs. A standard application “testusb” is available on the host side to execute desired test cases.

Please refer to the following link for details on this test setup: http://www.linux-usb.org/usbtest/, includes the binary image and the source code of an application to test the USB device gadget file system.

To enable the USB gadget device driver it is necessary to use the following Linux kernel parameter:

```
usb=on:device
```

in the XML configuration file for the OK Linux.

USB device Ethernet gadget

A simple application that can be performed in order to verify the functionality of the STreamPlug chip working as a USB device with the Ethernet gadget enabled is to try to establish a network communication with a remote machine such as it happens with the Ethernet “Best Effort” device driver. The application used is “iperf” which is provided as a Buildroot package.

In order to run the test, perform the following steps:

1. On the STreamPlug side, load the Ethernet gadget module:

```
$  modprobe   g_ether.ko
g_ether gadget:  using random  self ethernet  address g_ether gadget: using
random host ethernet address usb0:  MAC  4e:23:fc:67:21:49
usb0:  HOST MAC  16:a6:02:7c:6c:53
g_ether gadget:  Designware USB Device Controller  driver,  version:
Memorial Day  2008 g_ether gadget:  g_ether ready
registered  gadget driver  'g_ether'
```

2. Assign an IP address to the usb0 interface. For example:

```
$  ifconfig usb0  192.168.3.1  up
```

```
ADDRCONF(NETDEV_UP): usb0: link is not ready
```

On the host PC side:

1. Connect a USB cable.

```
[on STreamPlug console]:
g_ether gadget: high speed config #1: CDC Ethernet (EEM)
ADDRCONF(NETDEV_CHANGE): usb0: link becomes ready
```

2. Verify that the usb0 device is detected.

```
[on Host PC terminal]:
$ ifconfig usb0
usb0 Link encap:Ethernet HWaddr 96:f1:78:09:8e:53
inet6 addr: fe80::94f1:78ff:fe09:8e53/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1494 Metric:1
RX packets:6 errors:0 dropped:0 overruns:0 frame:0
TX packets:26 errors:0 dropped:0 overruns:0 carrier:0 collisions:0
txqueuelen:1000
RX bytes:384 (384.0 B) TX bytes:5713 (5.7 KB)
```

3. Assign an IP address to the usb0 interface. For example:

```
$ sudo ifconfig usb0 192.168.3.10 up
```

4. Check if the connection is up and running using the ping command on both sides.

On the STreamPlug side:

```
$ ping -c4 192.168.3.10
PING 192.168.3.10 (192.168.3.10): 56 data bytes
64 bytes from 192.168.3.10: seq=0 ttl=64 time=14.405 ms
64 bytes from 192.168.3.10: seq=1 ttl=64 time=4.534 ms
64 bytes from 192.168.3.10: seq=2 ttl=64 time=12.234 ms
64 bytes from 192.168.3.10: seq=3 ttl=64 time=20.256 ms

--- 192.168.3.10 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss round-trip
min/avg/max = 4.534/12.857/20.256 ms
```

On the host PC side:

```
$ping -c4 192.168.3.1
PING 192.168.3.1 (192.168.3.1) 56(84) bytes of data.
64 bytes from 192.168.3.1: icmp_req=1 ttl=64 time=15.0 ms
64 bytes from 192.168.3.1: icmp_req=2 ttl=64 time=31.2 ms
64 bytes from 192.168.3.1: icmp_req=3 ttl=64 time=6.29 ms
64 bytes from 192.168.3.1: icmp_req=4 ttl=64 time=21.4 ms

--- 192.168.3.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms rtt
min/avg/max/mdev = 6.290/18.482/31.220/9.106 ms
```

Finally, the “iperf” application may be used to evaluate the link bandwidth.

On the host PC side:

```
$iperf -s
```

On the STreamPlug side:

```
$iperf -c  192.168.3.10
```

The STreamPlug console will display:

```
------------------------------------------------------------
Client  connecting to 192.168.3.10, TCP port  5001
TCP window  size:  16.0 KByte   (default)
------------------------------------------------------------
[ 3] local  192.168.3.1 port 54883  connected with  192.168.3.10 port  5001
[ ID]  Interval Transfer Bandwidth
[ 3] 0.0-10.1 sec  14.0 MBytes  11.6 Mbits/sec
```

The output logs on the host PC side will show:

```
------------------------------------------------------------
Server listening on  TCP port  5001
TCP window  size:  85.3 KByte   (default)
------------------------------------------------------------
[ 4] local  192.168.3.10 port  5001  connected with  192.168.3.1 port  54883
[ ID]  Interval Transfer Bandwidth
[ 4] 0.0-10.2 sec  14.0 MBytes  11.5 Mbits/sec
```

```
The following is an example of console output while doing a gadget Ethernet
test:


STreamPlug  login:  root
#  modprobe   g_ether
g_ether gadget:  using random  self ethernet  address g_ether gadget: using
random host ethernet address usb0:  MAC  9a:98:08:e8:69:91
usb0:  HOST MAC  4a:61:f9:f4:cf:ed
g_ether gadget:  Designware USB Device Controller  driver,  version:
Memorial Day  2008 g_ether gadget:  g_ether ready
registered  gadget driver  'g_ether'
#  ifconfig
#  ifconfig usb0  192.168.3.1  up
#  ifconfig
usb0  Link encap:Ethernet  HWaddr 9A:98:08:E8:69:91
inet  addr:192.168.3.1 Bcast:192.168.3.255 Mask:255.255.255.0
inet6 addr:  fe80::9898:8ff:fee8:6991/64  Scope:Link
UP BROADCAST  RUNNING  MULTICAST MTU:1500  Metric:1
RX packets:68 errors:0  dropped:0 overruns:0 frame:0
TX packets:6 errors:0  dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:11947 (11.6  KiB)  TX bytes:504 (504.0 B)

#  ping 192.168.3.2
PING 192.168.3.2 (192.168.3.2):  56  data bytes
64  bytes from 192.168.3.2:  seq=0  ttl=64  time=3.520 ms
```

```
64 bytes from 192.168.3.2: seq=1 ttl=64 time=0.654 ms
64 bytes from 192.168.3.2: seq=2 ttl=64 time=0.660 ms
64 bytes from 192.168.3.2: seq=3 ttl=64 time=0.656 ms
64 bytes from 192.168.3.2: seq=4 ttl=64 time=0.660 ms
64 bytes from 192.168.3.2: seq=5 ttl=64 time=0.645 ms
64 bytes from 192.168.3.2: seq=6 ttl=64 time=0.664 ms
64 bytes from 192.168.3.2: seq=7 ttl=64 time=0.628 ms
64 bytes from 192.168.3.2: seq=8 ttl=64 time=0.645 ms

--- 192.168.3.2 ping statistics ---
9 packets transmitted, 9 packets received, 0% packet loss round-trip
min/avg/max = 0.628/0.970/3.520 ms
# iperf -c 192.168.3.2
------------------------------------------------------------ Client
connecting to 192.168.3.2, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[ 3] local 192.168.3.1 port 34838 connected with 192.168.3.2 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0-10.0 sec 34.6 MBytes 29.0 Mbits/sec
# rmmod g_ether
```

USB gadget FS

In order to test the USB device FS gadget, Run, on the STreamPlug, the application “gadgetfs” which may be found at the root level of the root FS “below/examples” folder.

Note: The host PC must have a Linux OS running, with a VM the test will not run.

1. Load the gadget FS module.

```
$ modprobe gadgetfs.ko
```

2. Verify the subfolder /gadget is present below the “/dev” folder, otherwise create it.
3. Mount the gadget FS module. Note that the message shows the driver is not associated anymore because the cable is not connected.

```
$ mount -t gadgetfs none /dev/gadget/
bind to driver nop --> error -120
```

4. Verify if the “designware_udc” device driver is correctly mounted and it is created below /dev/gadget.
5. Verify the USB cable is plugged on an appropriate connector only at the STreamPlug side.
6. Go into the “/example/gadgetfs” folder and run the application in background.

```
$ ./gadgetfs -v &
```

7. Verify it will display the following string:

```
$ /dev/gadget/designware_udc ep0 configured
$ serial="bvxryq1ex7ue0nw2yod9m9t5y2sib8939wzx4lo4y8g4h3m27peuxq1qi1z4p82"
```

Below the “/dev/gadget”, are the endpoints created ep1in, ep2out, ep3in:

8. Connect the cable to the PC USB connector.
9. On the host side, execute the command lsusb both as the normal user and superuser. Note that the host has correctly detected the USB device.

```
$ lsusb
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 002: ID 0525:a4a4 Netchip Technology, Inc. OKLinux-USB
user-mode bulk?' source/sink
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
$ sudo lsusb -v -s1:2
Bus 001 Device 002: ID 0525:a4a4 Netchip Technology, Inc. Linux-USB
user-mode bulk? source/sink
Device Descriptor:
bLength 18
bDescriptorType 1
bcdUSB2.00
bDeviceClass255 Vendor Specific Class
bDeviceSubClass 0
bDeviceProtocol 0
bMaxPacketSize0 64
idVendor0x0525 Netchip Technology, Inc.
idProduct0xa4a4 Linux-USB user-mode bulk source/sink
bcdDevice1.08
iManufacturer1 Analog Devices, Inc.
iProduct2 EZKIT-BF548
iSerial3 bNumConfigurations 1
Configuration Descriptor:
bLength 9
bDescriptorType 2
wTotalLength 39
bNumInterfaces 1
bConfigurationValue 3
iConfiguration4
GadgetFS Configuration bmAttributes0xc0
Self Powered
MaxPower 2mA
Device Qualifier (for other device speed):
bLength 10
bDescriptorType 6
bcdUSB2.00
bDeviceClass255 Vendor Specific Class
```

```
bDeviceSubClass 0
bDeviceProtocol 0
bMaxPacketSize0 64
bNumConfigurations  1
Device Status:  0x0000 (Bus  Powered)
```

10. On STreamPlug side, check that it will be displayed something like the following:

```
SUSPEND
CONNECT  high  speed
DISCONNECT
CONNECT  high  speed
SETUP 80.06 v0300   i0000  255
SETUP 80.06 v0302   i0409  255
SETUP 80.06 v0301   i0409  255
SETUP 80.06 v0303   i0409  255
SETUP 80.06 v0303   i0409  2
SETUP 80.06 v0303   i0409  128
```

4.4 I²C controller

The I²C is a multimaster serial single-ended 2-wire computer bus invented by Philips. It is used to attach low speed peripherals to a motherboard, embedded system, cellphone, or an other electronic device. It is a master-slave protocol, where communication takes a place between a host adapter (or a host controller) and client devices (or slaves).

4.4.1 I²C controller hardware overview

The I²C uses only two bidirectional lines, the serial data (SDA) and serial clock (SCL), pulled up with resistors. Typical voltages used are +5 V or +3.3 V, but systems with higher or lower voltages are also permitted. The I²C controller serves as an interface between the APB bus and the serial I²C bus. It provides master functions and controls all the I²C bus specific sequencing, protocol, arbitration and timing.

Features supported by I²C are:

- Two-wire I²C serial interface
- Two speeds:
  - Standard mode (100 Kbits/s)
  - Fast mode (400 Kbits/s)
- Master or slave I²C operation (LSP supports only the master mode)
- 7-bit or 10-bit addressing
- Slave bulk transfer mode
- Interrupt or polled-mode operation (LSP supports only the interrupt mode)

4.4.2 I²C controller software overview

Because the I²C works on the master/slave protocol, the communication takes a place between the host adapter (master) and client devices (slave).

The following terms are used for the I²C:

Bus

- Algorithm driver
- Adapter driver

Device

- Client driver

An algorithm driver contains the general code that can be used for the whole class of I²C adapters. Each specific adapter driver either depends on one algorithm driver, or includes its own implementation.

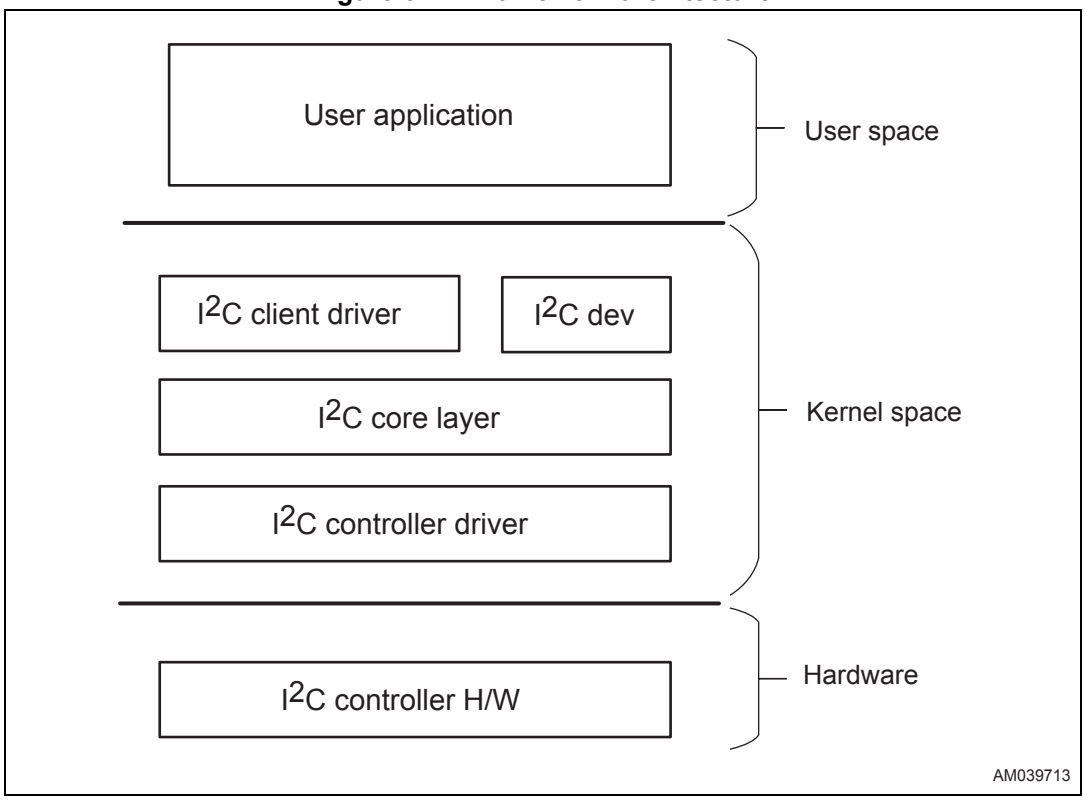
A client driver contains the general code to access some type of a device. Each detected device gets its own data in the client structure.

For a given configuration, it will need:

- A driver for the I²C bus,
- Drivers for I²C devices (usually one driver for each device), like the EEPROM, image sensor, etc.

Figure 6 illustrates the Linux I²C subsystem.

Figure 6. I²C framework architecture



4.4.3 I²C controller kernel source and configuration

The I²C kernel code is broken up into a number of logical blocks: the I²C core, I²C bus drivers and I²C client drivers.

I²C core

The I²C core is a code base consisting of routines and data structures available to host adapter drivers and client drivers. The core also provides a level of indirection that renders client drivers independent of the host adapter, allowing them to work even if the client device is used on a board that has a different I²C host adapter.

Busses

Busses are used for reading/writing to the slave device. The I²C busses drivers are provided in the following path: “drivers/i2c/busses/i2c-designware.c”.

I²C-dev

The I²C-dev allows communication with the user space. The I²C-dev code is provided in the following path: “drivers/i2c/i2c-dev.c”.

Table 11 lists the details corresponding to the layout of kernel configuration:

Table 11. I²C configurations

Configuration	Description
CONFIG_I2C	It enables I²C support
CONFIG_I2C_CHARDEV	It provides a character device for each I²C device present in the system, allowing the user to read and write directly from the I²C bus by ioctl() function.
CONFIG_I2C_DESIGNWARE	It enables the Synopsys designware I²C adapter (I²C hardware bus support). Only the master mode is supported.
CONFIG_I2C_HELPER_AUTO	It enables the “I²C algorithm” modules. These are basically software only abstractions of generic I²C interfaces.

4.4.4 I²C controller platform configuration

The optional platform data passed from machines for the I²C is as follows:

```
/* i2c device registration  */
static struct resource i2c_resources[]  =  {
   {
      .start  =  STREAMPLUG1X_ICM1_I2C_BASE,
      .end  =  STREAMPLUG1X_ICM1_I2C_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   }, {
      .start  =  STREAMPLUG1X_IRQ_LOW_SPEED_SUBS_I2C,
      .flags  =  IORESOURCE_IRQ,
   },
};

struct platform_device  streamplug1x_i2c_device =  {
   .name  =  "i2c_designware",
```

```
    .id  =  0,
    .dev  =  {
    .coherent_dma_mask  =  ~0,
},
    .num_resources  =  ARRAY_SIZE(i2c_resources),
    .resource  =  i2c_resources,
};
```

4.4.5 I²C controller usage

To enable the I²C support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

Access from user space

Usually, I²C devices are controlled by a kernel driver. But all devices on an adapter can be accessed from the user space, through the “/dev” interface, by loading the module “i2c-dev”.

Each registered I²C adapter gets an integer number, counting from zero (0). One can examine “/sys/class/i2c-dev/” to see what number corresponds to which adapter. The I²C device files are character device files with a major device number 89 and a minor device number corresponding to the number assigned as explained above. They should be called “i2c-%d” (i2c-0, i2c-1, . . . , i2c-10, . . .).

All 256 minor device numbers are reserved for “i2c”.

To access an I²C adapter from a C program, for example: “#include <linux/i2c-dev.h>”.

Note that there are two files named “i2c-dev.h” . One is distributed with the Linux kernel and is meant to be included from the kernel driver code, the other one is distributed with “i2c-tools” and is meant to be included from user space programs.

After deciding which adapter to access, inspect “/sys/class/i2c-dev/”. Adapter numbers are assigned somewhat dynamically, few assumptions about them can be made. They can even change from one boot to the next.

Next thing, open the device file, as follows:

```
#define DEVICE_FILE_NAME_I2C  "/dev/i2c-0"

int fd;

fd =  open(  DEVICE_FILE_NAME_I2C,  O_RDWR  );
if (fd <  0) {
   fprintf(stderr,  "FAILED  OPEN  Device i2c-0. Ret=%d\n",  fd);
   close(fd);
   exit(-1);
}
```

After the device is opened, specify the device address with which to communicate:

```
/* The  I2C  address */
#define I2C_SLAVE0x41
#define REGISTER_ADDR0x11
```

```
res = ioctl(fd, I2C_SLAVE, REGISTER_ADDR);
if (res < 0) {
   /* ERROR HANDLING; you can check errno to see what went wrong
*/
   fprintf(stderr, "FAILED I2C Set Address. Ret=%d\n", res);
   close(fd);
   exit(-1);
}
```

This completes the setup. The SMBus commands or the plain I^{2}C can now be used to communicate with a device. SMBus commands are preferred if the device supports them. Both are illustrated below.

Note that only a subset of the I^{2}C and SMBus protocols can be achieved by the means of read() and write() calls. In particular, so called combined transactions (mixing read and write messages in the same transaction) aren't supported. For this reason, this interface is almost never used by user space programs.

I^{2}C dev. interface

The following IOCTLs are defined for user space access:

“ioctl(file, I2C_SLAVE, long addr)”

Change the slave address. The address is passed in the 7 lower bits of the argument (except for 10-bit addresses, passed in the 10 lower bits in this case).

“ioctl(file, I2C_TENBIT, long select)”

Selects 10-bit addresses if selection not equals 0, selects normal 7-bit addresses if selection equals 0. Default 0. This request is only valid if the adapter has “I2C_FUNC_10BIT_ADDR”.

“ioctl(file, I2C_FUNCS, unsigned long *funcs)”

Gets the adapter functionality and puts it in *funcs.

“ioctl(file, I2C_RDWR, struct i2c_rdwr_ioctl_data *msgset)”

Do combine read/write transaction without stop in between. Only valid if the adapter has “I2C_FUNC_I2C”. The argument is a pointer to the following “struct”:

```
struct i2c_rdwr_ioctl_data {
   struct i2c_msg *msgs;/* ptr to array of simple messages */
   int nmsgs;/* number of messages to exchange */
}
```

The “msgs” pointer contains further pointers to data buffers. The function will write or read data to or from that buffers depending on whether the “I2C_M_RD” flag is set in a particular message or not. The slave address and whether to use the 10-bit address mode has to be set in each message, overriding the values set with the above ioctl's.

“ioctl(file, I2C_SMBUS, struct i2c_smbus_ioctl_data *args)”

These are not meant to be called directly. Instead, use the access functions below. Plain I^{2}C transactions can be done by using read(2) and write(2) calls. The address byte does not need to be passed. Instead, set it through the ioctl “I2C_SLAVE” before accessing the device.

SMBus level transactions (see <linux src>/Documentation/i2c/smbus-protocol for details) are done through the following functions:

s32 i2c_smbus_write_quick(int file, u8 value) , _s32 i2c_smbus_read_byte(int file)

All these transactions return -1 on failure; errno can be read to see what happened.

The write transactions return “0” on success; the read transactions return the read value, except for the “read_block”, which returns the number of values read. The block buffers need not be longer than 32 bytes. The above functions are all inline functions that resolve to calls to the “i2c_smbus_access” function that on its turn calls a specific ioctl with the data in a specific format. Read the source code to know what happens behind the screens.

Access in kernel space, through client driver

Usually, a single driver structure will be implemented and all clients instantiate from it.

Note: A driver structure contains general access routines, and should be zero initialized except for fields with data provided by the user. A client structure holds device specific information like the driver model device node, and its I²C address.

Following example is taken from “drivers/misc/eeprom/eeprom.c”, an I²C client driver to access EEPROMs.

Device creation

If it is known that an I²C device is connected to a given I²C bus, the device can be instantiated by simply filling an i2c_board_info structure with the device address and driver name and calling “i2c_new_device()”. This will create the device, the driver core will take care of finding the right driver and will call its “probe()” method. If a driver supports different device types, the type can be specified using the type field. IRQ and platform data can also be specified if needed.

For example, in “arch/arm/mach-streamplug/streamplug_devel_board.c”:

```
static  struct i2c_board_info   initdata i2c_board_info[]  =  {
   {
   .type  =  "eeprom",
   .addr  =  0x50,
   }
};
static  void    init i2c_init(void)
{
   i2c_register_board_info(0, i2c_board_info,
   ARRAY_SIZE(i2c_board_info));
}
```

Device detection

Sometimes is not known in advance which I²C devices are connected to a given I²C bus . This is for example the case of hardware monitoring devices on a PC's SMBus. In that case, the driver may be used to detect supported devices automatically. This is how the legacy model was working, and is now available as an extension to the standard driver model.

Simply define a detect callback which will attempt to identify supported devices (returning 0 for supported devices and “-ENODEV” for unsupported devices), a list of addresses to probe, and a device type (or class) so that only I²C buses which may have that type of the

device connected (and not otherwise enumerated) will be probed. For example, a driver for a hardware monitoring chip for which auto-detection is needed would set its class to “I2C_CLASS_HWMON”, and only I2C adapters with a class including “I2C_CLASS_HWMON” would be probed by this driver. Note that the absence of matching classes does not prevent the use of a device of that type on the given I2C adapter. All it prevents is auto-detection; explicit instantiation of devices is still possible.

This device detection mechanism is purely optional and not suitable for all devices.
A reliable way to identify the supported devices (typically using device specific, dedicated identification registers) is needed, otherwise misdetections are likely to occur and errors will occur.

Keep in mind that the I2C protocol doesn't include any standard way to detect the presence of a chip at a given address, let alone a standard way to identify devices. Even worse is the lack of semantics associated to bus transfers, which means that the same transfer can be seen as a read operation by a chip and as a write operation by another chip. For these reasons, explicit device instantiation should always be preferred to auto-detection where possible.

Device driver initialization

When the kernel is booted, or when an I2C driver module is inserted, some initializing has to be done. Fortunately, just registering the driver module is usually enough. The following example specifies the initialization procedure for an EEPROM device.

In the struct “i2c_driver”, below, the name field is the driver name, and must not contain spaces. It should match the module name (if the driver can be compiled as a module), although the “MODULE_ALIAS” can be used to add another name for the module. If the driver name doesn't match the module name, the module won't be automatically loaded (“HotPlug/ColdPlug”).

```
static const struct  i2c_device_id eeprom_id[]  =  {
   { "eeprom",  0  },
   { },
};


MODULE_DEVICE_TABLE(i2c,  eeprom_id);

static struct i2c_driver eeprom_driver =  {
   .driver  =  {
   .name=  "eeprom",
   },
   .probe=  eeprom_probe,
   .remove=  eeprom_remove,
   .id_table=  eeprom_id,
   .class=  I2C_CLASS_DDC | I2C_CLASS_SPD,
   .detect=  eeprom_detect,
   .address_list=  normal_i2c,
};

static int    init eeprom_init(void)
{
```

```
    return i2c_add_driver(&eeprom_driver);
}

static void    exit eeprom_exit(void)
{
    i2c_del_driver(&eeprom_driver);
}
```

All other fields are for callback functions.

Extra client data

Each client structure has a special data field that can point to any structure at all. This may be used to keep device specific data:

```
/* store the value */
void i2c_set_clientdata(struct  i2c_client *client,  void *data);

/* retrieve  the value */
void *i2c_get_clientdata(const  struct i2c_client *client);
```

I²C communication in kernel space

There are several functions that may be used to communicate with a device. They can be found in “include/linux/i2c.h”.

“int i2c_master_send(struct i2c_client *client, const char *buf, int count)”

These routines read and write some bytes from/to a client. The client contains the I²C address, it does not have to be included. The second parameter contains the bytes to read/write, the third contains the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64 Kbytes because “msg.len” is u16). The actual number of bytes read or written is returned.

“int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg, int num)”

This routine sends a series of messages. Each message can be a read or write, and they can be mixed in any way. When the transactions are combined, no stop bit is sent between transactions. The “i2c_msg” structure contains for each message the client address, the number of bytes of the message and the message data itself.

SMBus communication in kernel space

Following defines SMBus APIs to access I²C devices from kernel space.

“s32 i2c_smbus_read_byte(struct i2c_client *client)”, “s32 i2c_smbus_write_byte(struct i2c_client *client)”

All these transactions return a negative errno value on failure. The write transactions return 0 on success; the read transactions return the read value, except for block transactions, which return the number of values read. The block buffers need not be longer than 32 bytes.

```
s32  i2c_smbus_xfer(struct  i2c_adapter *adapter,  u16  addr,
unsigned short  flags,  char read_write,
u8  command,  int size,  union  i2c_smbus_data *data);
```

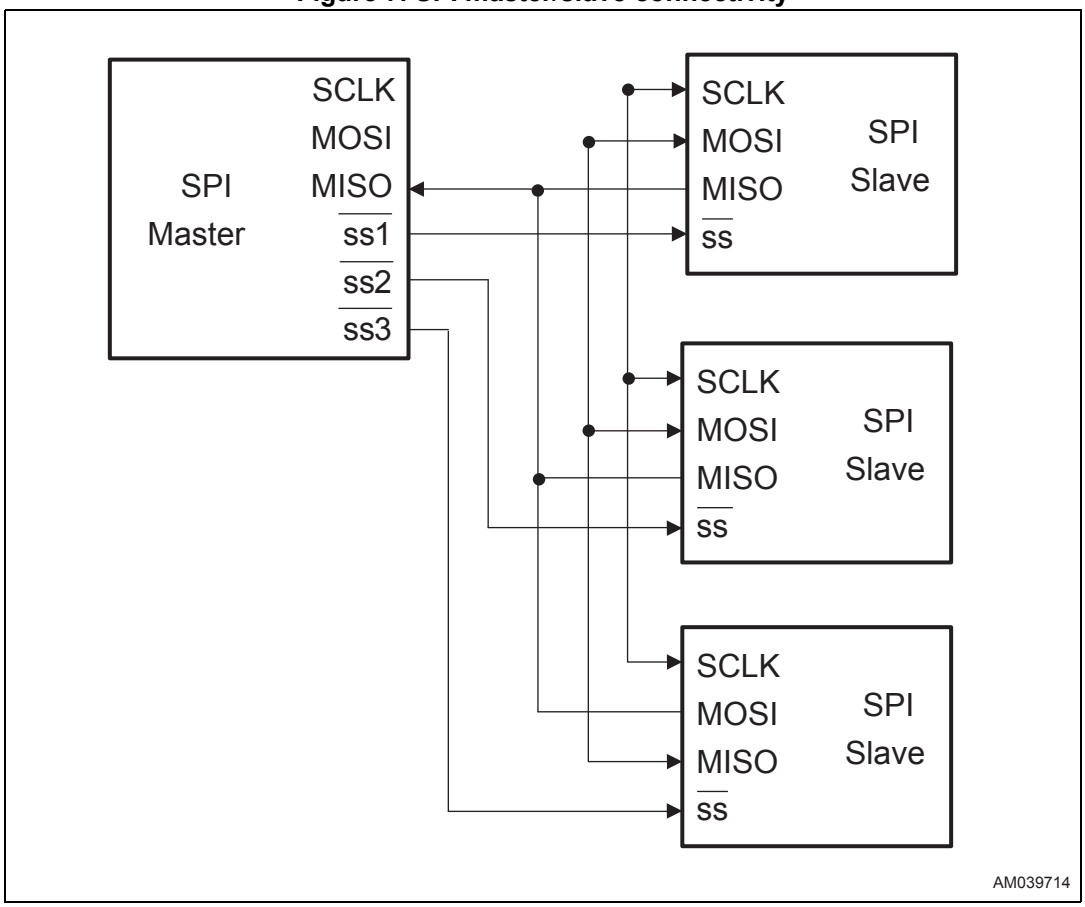
Read the file *“linux-2.6/Documentation/i2c/smbus-protocol”* for more information about the actual SMBus protocol.

All functions in the above subsection are implemented in using the following “s32 i2c_smbus_xfer( )” function which must not be used directly.

4.5 Serial peripheral interface (SPI) controller

The serial peripheral interface (SPI) bus is a synchronous serial data link standard named by Motorola that operates in the full duplex mode. Devices communicate in the master/slave mode where the master device initiates the data frame. Individual slave select (CHIP-SELECT) lines allow to multiple slave devices as shown in *Figure 7*. The SPI is used to connect microcontrollers to the sensors, memory and peripherals.

Figure 7. SPI master/slave connectivity



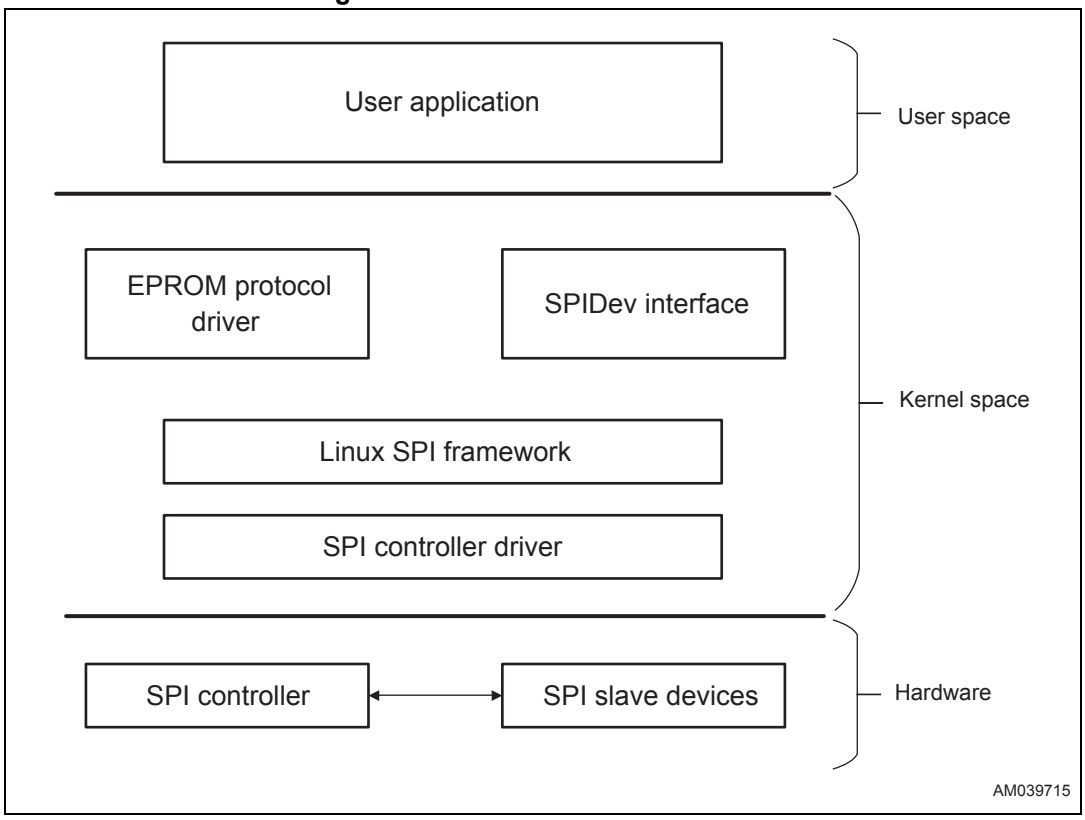
The SPI bus specifies four logic signals.

- SCLK - serial clock (output from master)
- MOSI - master output, slave input (output from master)
- MISO - master input, slave output (output from slave)
- SS - slave select (active low; output from master).

4.5.1 SPI software overview

The SPI framework present in Linux supports only the master side of the Motorola SPI interface. User applications can use the interface provided by the protocol drivers present in Linux. Protocol drivers use the standard calls provided by the SPI framework present in Linux. The SPI controller driver provides the interface to the SPI framework for accessing the SPI controller. The SPI controller transfers data to the SPI slave devices/memories connected to it according to the configuration provided by the SPI controller driver. *Figure 8* presents the SPI software system architecture.

Figure 8. SPI framework architecture



The Linux SPI framework defines two types of SPI drivers in Linux:

- Control drivers
- Protocol/slave drivers

Controller drivers configure SPI controllers. Their interface can be used for configuring the controller and transfer data over the SPI bus. They may or may not use DMA for data transfer with the slave device. The Linux SPI framework uses controller drivers for all its SPI related operations. The ARM PL022 controller driver can be found at “drivers/spi/amba-pl022.c”.

Protocol/slave drivers pass messages through the controller driver to communicate with a slave device on the other side of an SPI link. They are present above the SPI kernel framework and they provide the interface to the user applications present in the user space.

Currently two sample protocol drivers are present in the “drivers/spi” folder.

- FLASH protocol driver (“m25p80”, supported but not configured)
- General character interface driver, spidev (spidev is supported and configured by default).

The FLASH driver uses the SPI framework to communicate with a range of serial NOR chips. This driver presents a standard MTD interface to user applications. The MTD node for the SPI can be found by looking at the “/proc/mtd” file after booting Linux. This driver can be enabled through the CONFIG_MTD_M25P80 Kconfig option and is present in “drivers/mtd/devices/”.

The general character interface driver provides a character special device to the SPI controller. To access it, use the following calls: “open()”, “read()”, “write()” and “ioctl()”.

For a new interface, write a new protocol driver. For information on using the SPI framework, see the “spidev.c” files in the “drivers/spi” folder.

The SPI shows up in “sysfs” at several locations:

- /sys/devices/. . . /CTLR . . . physical node for a given SPI controller
- /sys/devices/. . . /CTLR/spiB.C . . . spi_device on bus “B”, CHIP-SELECT C, accessed through CTLR.
- /sys/bus/spi/devices/spiB.C . . . symlink to that physical . . . /CTLR/spiB.C device
- /sys/devices/. . . /CTLR/spiB.C/modalias . . . identifies the driver that should be used with this device (for hot plug/cold plug)
- /sys/bus/spi/drivers/D . . . a driver for one or more spi*.* devices.
- /sys/class/spi_master/spiB . . . symlink (or actual device node) to a logical node which could hold the class related state for the controller managing the bus “B”. All spiB.* devices share one physical SPI bus segment, with SCLK, MOSI, and MISO.

The Linux SPI framework provides APIs for registering and unregistering SPI slave drivers and transferring data over the SPI bus. These functions will be explained one by one with examples from the FLASH driver.

“int spi_register_driver(struct spi_driver *sdrv)”

The SPI slave driver must register itself with the SPI framework by calling this API, preferably from its “module_init()” routine. In this routine, the “struct slave_driver” is a structure which contains information about the SPI slave driver.

```
struct spi_driver {
   const struct  spi_device_id *id_table;
   int(*probe)(struct  spi_device *spi);
   int(*remove)(struct  spi_device *spi);
   void  (*shutdown)(struct  spi_device *spi);
   int(*suspend)(struct  spi_device *spi,  pm_message_t   mesg);
   int(*resume)(struct  spi_device *spi);
   struct  device_driverdriver;
};
```

Implementation of all these functions may not be required.

The following “spi_setup” API used to configure SPI controller slave drivers can change the previously saved struct “spi_device” with the new SPI configuration, like “bits_per_word”, “max_speed_hz”, mode, etc., and then call this API.

```
int spi_setup(struct  spi_device *spi)
```

The following API writes len (length) bytes of data present at the “buf” (buffer) address to the SPI slave device attached to the SPI controller.

```
int spi_write(struct  spi_device *spi,  const u8  *buf,  size_t len)
```

The following API reads “len” bytes of data from the SPI slave device attached to the SPI controller and saves data to the buf address.

```
int spi_read(struct  spi_device *spi,  u8  *buf,  size_t len)
```

The following API is used for using the full duplex mode of the SPI controller. It writes data to the slave device from the “txbuf” address and reads data to the “rxbuf” address from the slave device. The length of transfer is “len” for both Tx and Rx.

```
int spi_write_and_read(struct  spi_device *spi,  const u8  *txbuf,  u8
*rxbuf,  size_t len)
```

The following spi_unregister API is used to unregister a slave driver from the SPI framework. Preferably done from “module_exit()” routine of the slave driver.

```
void spi_unregister_device(struct  spi_device *spi)
```

Adding a new slave driver

This section will explain how a slave device driver should register itself with the SPI framework, using the “m25p80.c” driver as a basis for the example.

Registering slave driver

The SPI slave driver must be registered with the SPI framework. This is accomplished by calling:

```
spi_register_driver(&m25p_driver);
```

Unregistering the driver

The SPI slave driver must unregister itself when its module is removed. This is done by calling:

```
spi_unregister_driver(&m25p_driver).
```

Following is a registration part of the “drivers/mtd/devices/m25p80.c” driver:

```
static const struct  spi_device_id m25p_ids[]  =  {
   { "at25fs010",  INFO(0x1f6601,  0, 32  * 1024,4, SECT_4K)  },
   { "at25fs040",  INFO(0x1f6604,  0, 64  * 1024,8, SECT_4K)  },

   ......

/* ST Microelectronics -- newer   production may have feature  updates?'
*/
   { "m25p05",  INFO(0x202010,  0, 32  * 1024,2, 0)},
   { "m25p10",  INFO(0x202011,  0, 32  * 1024,4, 0)},
   { "m25p20",  INFO(0x202012,  0, 64  * 1024,4, 0)},
   { "m25p40",  INFO(0x202013,  0, 64  * 1024,8, 0)},
   { "m25p80",  INFO(0x202014,  0, 64  * 1024, 16, 0) },

   ......

   { },
};
MODULE_DEVICE_TABLE(spi,  m25p_ids);

static struct spi_driver m25p_driver =  {
   .driver  =  {
      .name  =  "m25p80",
      .owner  =  THIS_MODULE,
   },
   .id_table  =  m25p_ids,
   .probe=  m25p_probe,
   .remove=    devexit_p(m25p_remove),
};

static int   init m25p80_init(void)
{
   return spi_register_driver(&m25p_driver);
}
module_init(m25p80_init);

static void   exit m25p80_exit(void)
{
   spi_unregister_driver(&m25p_driver);
}
module_exit(m25p80_exit);
```

The kind of the interface provided to the user applications is slave driver dependent (sysfs, proc, dev, etc.). The struct spi_device is passed to the slave driver when the probe function of the slave is called from the SPI framework after the device registration. The structure in the slave driver must be saved and used for any communication with the SPI framework.

4.5.2 SPI kernel source and configuration

Following is the detail corresponding to the layout of the driver and kernel configuration:

The SPI controllers driver is present in “drivers/spi/amba-pl022.c”.

The SPI slave controller drivers are present in:

- “m25p80” slave driver: “drivers/mtd/devices/m25p80.c”
- “spidev” slave driver: “drivers/spi/spidev.c”

Table 12 lists the kernel configuration options associated with the SPI:

Table 12. SPI configurations

Configuration	Description
CONFIG_SPI	It enables SPI framework layer support
CONFIG_SPI_MASTER	It enables SPI_MASTER support
CONFIG_SPI_PL022	It enables AMBA™ PL022 controller support
CONFIG_SPI_SPIDEV	It enables SPIDEV slave support

4.5.3 SPI platform configuration

Once the slave/protocol driver is up, we must add/register a slave device with the SPI bus. The SPI controller driver and slave driver need some board specific data to work correctly. This data is present in the all SoC's board files, (i.e.: “streamplug_devel_board.c”). There are two types of platform information:

- “platform_data” which initializes “spi_device.platform_data”, the particular data stored there is slave driver specific.
- “controller_data” which is required by the SPI controller driver. This structure is controller driver specific and for adding a new slave device, it must be supplied.

```
DECLARE_SPI_CHIP_INFO(0,  flash,  spi0_flash_cs_control);

/* This will  define CHIP_INFO  structure for a  specific  spi slave */
#define DECLARE_SPI_CHIP_INFO(id,  type,  chip_select_control)\
static struct pl022_config_chip  spi##id##_##type##_chip_info =  {\
   .lbm  =  LOOPBACK_DISABLED,\
   .iface  =  SSP_INTERFACE_MOTOROLA_SPI,\
   .hierarchy  =  SSP_MASTER,\
   .slave_tx_disable  =  0,\
   .endian_tx  =  0,\
   .endian_rx  =  0,\
   .ctrl_len  =  SSP_BITS_8,\
   .data_size  =  SSP_DATA_BITS_8,\
   .com_mode  =  INTERRUPT_TRANSFER,\
```

```
   .rx_lev_trig  =  0,\
   .tx_lev_trig  =  0,\
   .clk_phase  =  SSP_CLK_FIRST_EDGE,\
   .clk_pol  =  SSP_CLK_POL_IDLE_LOW,\
   .cs_control  =  chip_select_control,\
};

static  struct spi_board_info   initdata spi_board_info[]  =  {
/* register m25p80  driver */
   {
      .modalias  =  "m25p80",
      .controller_data  =  &spi0_flash_chip_info,
      .platform_data  =  &spi_flash_info,
      .max_speed_hz  =  800000,
      .bus_num  =  0,
      .chip_select  =  0,
      .mode  =  0,
   }
};

/* Define chip select  routine  using GPIO_15  (gpio  line  39) as  default,
otherwise is
provided by  command line option */
DECLARE_SPI_CS_CONTROL(0,  flash,  cs_gpio_pin);

/* This will  define cs_control function for  a  specific  spi slave */
#define DECLARE_SPI_CS_CONTROL(id,  type,  gpio)\
   static  void spi##id##_##type##_cs_control(u32  control)\
   {\
      static int count,  ret;\
                              \
      if (unlikely(!count))  {\
         count++; \ ret =  spi_cs_gpio_request(gpio);\
      }           \
               \
   if (!ret)                                    \
      gpio_set_value(gpio,  control);             \
}

/* Definition of spi_cs_gpio_request()  is  present in <plat/spi.h>  */
static inline int spi_cs_gpio_request(u32  gpio_pin)
{
   int ret;

   ret =  gpio_request(gpio_pin,  "SPI_CS");
   if (ret <  0) {
```

```
            printk(KERN_ERR  "SPI:  gpio:%d request  fail\n",  gpio_pin);
            return ret;
        }
        else {
            ret =  gpio_direction_output(gpio_pin,  1);
            if (ret) {
                printk(KERN_ERR  "SPI:  gpio:%d direction set fail\n",
                    gpio_pin);
                return ret;
            }
        }
        return 0;
}
```

4.5.4 SPI usage

The user space and kernel space usage through the framework or directly (if the framework is not present).

4.6 Linux TTY framework

The TTY framework of the Linux kernel, serves as an intermediary layer between hardware device drivers and user applications to provide line buffering and management of input and output. The layer is purely software oriented and makes no direct communication with physical hardware. Instead, the TTY driver relies on an underlying device driver to communicate directly with the hardware.

4.6.1 Linux TTY framework software overview

The basic function of the TTY layer is to interface with the lower level device driver and insulate the higher level from the complexity of the hardware level. There are different types of TTY drivers: the console and serial port. The console driver is used at two different places in Linux. Firstly, at boot time it is used before the initialization of the serial TTY framework as it takes some time for the serial TTY framework to initialize during Linux boot-up. Secondly, after Linux boot-up, the console device sits in the lowest levels of Linux in order to bring critical information out of the system as soon as possible. It is not involved in all the complexity of TTY management.

4.6.2 Linux TTY framework kernel source

The tty source code in Linux is present in “drivers/char/tty_*” files.

4.6.3 Linux TTY framework usage

The following paragraphs briefly describe system calls which can be used to access and use serial devices in the TTY framework. The non-blocking mode is supported. When running in the blocking mode it may need to wait for the carrier.

System calls

TTY side interfaces:

open()

Called when the line discipline is attached to the terminal. No other call into the line discipline for this TTY will occur until it completes successfully. Returning an error will prevent the ldisc from being attached. If the “O_NONBLOCK” flag is specified and the open() call would result in the process being blocked for some reason it returns immediately. The first time the process attempts to perform I/O on the open descriptor it will block.

close()

This is called on a terminal when the line discipline is being unplugged. At the point of execution no further users will enter the ldisc code for this TTY.

hangup()

Called when the tty line is hung up. The line discipline should cease I/O to the TTY. No further calls into the ldisc code will occur. The return value is ignored.

write()

A process is writing data through the line discipline. Multiple write calls are serialized by the tty layer for the “ldisc”.

flush_buffer()

(Optional). May be called at any point between open and close, and instructs the line discipline to empty its input buffer.

chars_in_buffer()

(Optional). Reports the number of bytes in the input buffer.

set_termios()

(Optional). Called on “termios” structure changes. The caller passes the old termios data and the current data is in the TTY. It is called under the termios semaphore so it is allowed to sleep. Serialized only against itself.

read()

Moves data from the line discipline to the user. Multiple read calls may occur in parallel and the ldisc must deal with serialization issues.

poll()

Checks the status for the poll/select calls. Multiple poll calls may occur in parallel.

ioctl()

Called when an ioctl is handed to the TTY layer that might be for the ldisc. Multiple ioctl calls may occur in parallel.

compat_ioctl()

Called when a 32-bit ioctl is handed to the TTY layer that might be for the ldisc. Multiple ioctl calls may occur in parallel.

Example commands

The following paragraphs are example commands:

Getty

Getty opens a TTY port, prompts for a login name and invokes the “/bin/login” command. The getty command sets and manages terminals by setting up the speed, the terminal flags, and the line discipline.

Example:

```
/sbin/getty  9600  ttyAMA1
```

Stty

Stty is used to change and print the terminal line settings.

```
+/* List the attribute settings for a  terminal that has  a  user logged+
+  * on  it already.+
+  */+
stty -a  -F  /dev/ttyAMA0

+/* Disable modem control signals.  */+
stty clocal -F  /dev/ttyAMA0

+/*  Enable RTS/CTS  handshaking */+
stty crtscts -F  /dev/ttyAMA0

/* Set   the baud  rate of current terminal to 9600  baud.  */
stty
```

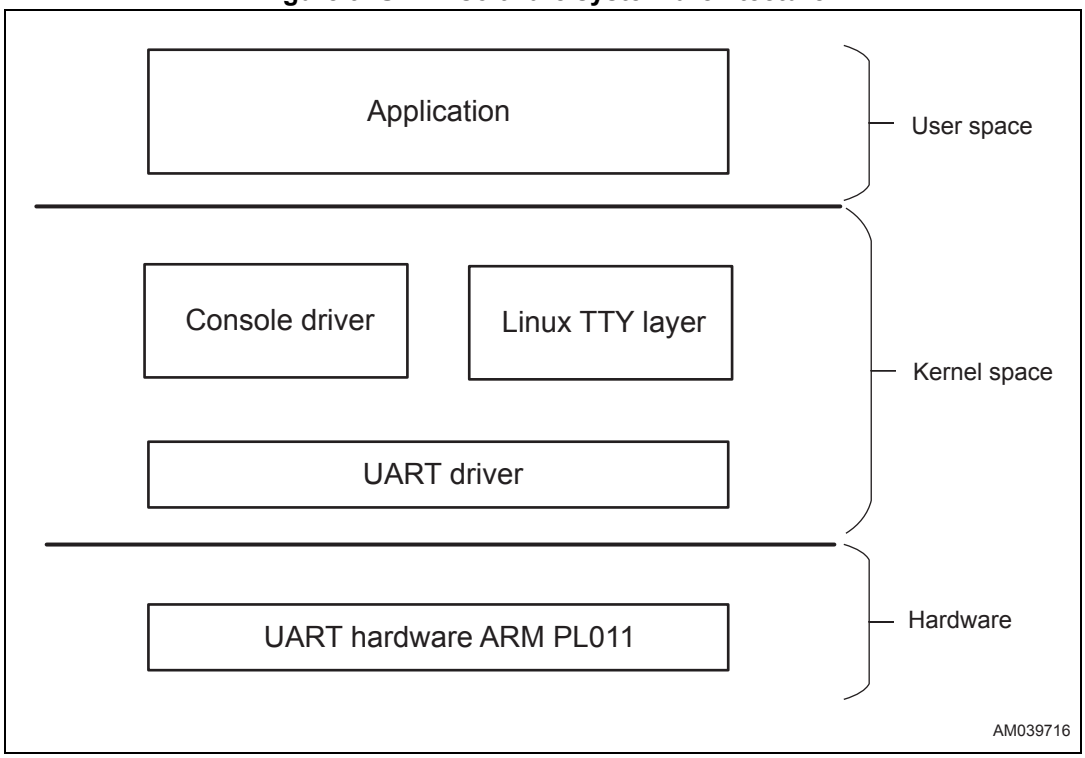
4.7 Universal asynchronous receiver/transmitter (UART)

The UART is a universal asynchronous receiver/transmitter that supports much of the industry standard 16C550 UART. Two UART ports are available on the STreamPlug. This section describes the integration of the STreamPlug UART device driver into the Linux kernel.

4.7.1 UART software overview

UART drivers support the TTY kernel layer. The I/O system calls start above top level line disciplines and finally ripple down to UART drivers through the TTY layer as shown in *Figure 9*. The data flow between the user space and the serial device driver, therefore, is mediated by the TTY layer, that implements functionalities that are common to all TTY-type devices.

Figure 9. UART software system architecture



There are different types of TTY drivers: the console and serial port. The console driver is used at two different places in Linux. First, at boot time it is used before the initialization of the serial TTY framework as it takes some time for the serial TTY framework to initialize during Linux boot-up. Secondly, after Linux boot-up, the console device sits in the lowest levels of Linux in order to bring critical information out of the system as soon as possible. It is not involved in all the complexity of TTY management. The serial ports are named ttyAMA0, ttyAMA1, etc. For each such serial port, there is a special file in the /dev (device) directory. The major number 204 is associated to the ttyAMA driver. For the UART layer the major number is 204 and the minor number ranges between 64 - 255.

4.7.2 UART kernel source and configuration

The following section contains details corresponding to the layout of the driver and kernel configuration.

The UART AMBA PL011 controller driver is present in "*drivers/serial/amba-pl011.c*".

The serial core controller is present in "*drivers/serial/serial-core.c*".

The platform data defining UART1 and UART2 controller configurations is present in *"arch/arm/mach-streamplug/streamplug1x.c"* as listed:

- "CONFIG_SERIAL_CORE" enables the UART TTY framework.
- "CONFIG_SERIAL_CORE_CONSOLE" enables the UART console framework.
- "CONFIG_SERIAL_AMBA_PL011" enables STreamPlug AMBA PrimeCell PL011 UART driver support for the TTY framework.
- "CONFIG_SERIAL_AMBA_PL011_CONSOLE" enables the STreamPlug UART driver support for the console framework.

4.7.3 UART platform configuration

This section lists the driver's platform interface and its possible configuration.

Driver configuration

Default device registration of the UART1/2 controller depends on the platform data passed from the boards ("*arch/arm/mach-streamplug/streamplug1x.c*").

```
/* uart1 device registration  */
struct amba_device streamplug1x_uart1_device =  {
   .dev  =  {
      .init_name  =  "uart1",
   },
   .res  =  {
   .start  =  STREAMPLUG1X_ICM1_UART1_BASE,
   .end  =  STREAMPLUG1X_ICM1_UART1_BASE  +  SZ_4K - 1,
   .flags  =  IORESOURCE_MEM,
   },
   .irq  =  {STREAMPLUG1X_IRQ_LOW_SPEED_SUBS_UART1,  NO_IRQ},
};

struct amba_device streamplug1x_uart2_device =  {
   .dev  =  {
      .init_name  =  "uart2",
   },
   .res  =  {
      .start  =  STREAMPLUG1X_ICM1_UART2_BASE,
      .end  =  STREAMPLUG1X_ICM1_UART2_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   },
   .irq  =  {STREAMPLUG1X_IRQ_LOW_SPEED_SUBS_UART2,  NO_IRQ},
};
```

4.7.4 UART usage

For usage please refer to the chapter on the Linux TTY framework.

4.8 Control area network (CAN)

The controller area network (CAN) bus is a bus designed to allow microcontrollers and devices to communicate with each other without a host computer. The STreamPlug CAN device driver is derived from the CAN bus driver for the “C_CAN” controller which is compliant to the CAN protocol version 2.0.

4.8.1 CAN software overview

The CAN device driver belongs to the Linux network stack and it is accessible through network interfaces. The entire stack can be summarized in the following layers and features:

- Application layer
- Object layer
  - Message filtering
  - Message and status handling
- Transfer layer
  - Fault confinement
  - Error detection and signaling
  - Message validation
  - Acknowledgment
  - Arbitration
  - Message framing
  - Transfer rate and timing
- Physical layer
  - Signal level and bit representation
  - Transmission medium

4.8.2 CAN kernel source and configuration

Table 13 lists the STreamPlug Linux kernel configuration which must be used to support the STreamPlug CAN bus.

Table 13. CAN Linux kernel configuration

Configuration	Description
CONFIG_CAN=y	Enable CAN support
CONFIG_CAN_RAW=y	
CONFIG_CAN_BCM=y	
CONFIG_CAN_DEV=y	
CONFIG_CAN_CALC_BITTIMING=y	
CONFIG_CAN_C_CAN=y	Enable C_CAN support
CONFIG_CAN_C_CAN_PLATFORM=y	Enable C_CAN platform

The key source code about the STreamPlug CAN device driver can be found in the following files:

```
drivers/net/can/c_can
drivers/net/can/c_can/c_can.h
drivers/net/can/c_can/c_can.c
drivers/net/can/c_can/c_can_platform.c
arch/arm/mach-streamplug/ipswrst_ctrl.c
arch/arm/mach-streamplug/include/mach/generic.h
arch/arm/mach-streamplug/include/mach/streamplug10.h
arch/arm/mach-streamplug/clock.c
arch/arm/mach-streamplug/padmux.c arch/arm/mach-streamplug/streamplug1x.c
```

An other related source code can be found also in the following folders:

```
include/linux/can
include/linux/can/platform
net/can
```

The hardware filtering mechanism is not enabled there, so all messages are considered good.

4.8.3 CAN platform configuration

The STreamPlug CAN device driver manages two resources: CAN1 and CAN2. The platform code for the driver is shown in the following paragraphs:

```
/* CAN1 device registration  */
static struct resource can1_resources[]  =  {
{
   .start  =  STREAMPLUG1X_ICM1_CAN1_BASE,
   .end  =  STREAMPLUG1X_ICM1_CAN1_BASE  +  SZ_4K - 1,
   .flags  =  IORESOURCE_MEM  | IORESOURCE_MEM_32BIT,
   }, {
   .start  =  STREAMPLUG1X_IRQ_LOW_SPEED_SUBS_CAN1,
   .flags  =  IORESOURCE_IRQ,
   },
};

struct  platform_device streamplug1x_can1_device =  {
   .name  =  "c_can_platform",
   .id  =  1,
   .num_resources  =  ARRAY_SIZE(can1_resources),
   .resource  =  can1_resources,
};

/* CAN2 device registration  */
static struct resource can2_resources[]  =  {
   {
      .start  =  STREAMPLUG1X_ICM1_CAN2_BASE,
      .end  =  STREAMPLUG1X_ICM1_CAN2_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM  | IORESOURCE_MEM_32BIT,
   }, {
      .start  =  STREAMPLUG1X_IRQ_LOW_SPEED_SUBS_FIRDA_CAN2,
      .flags  =  IORESOURCE_IRQ,
   },
};

struct  platform_device streamplug1x_can2_device =  {
   .name  =  "c_can_platform",
   .id  =  2,
   .num_resources  =  ARRAY_SIZE(can2_resources),
   .resource  =  can2_resources,
};
```

4.8.4 CAN usage

To enable the CAN support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

The CAN device driver can be tested connecting the board to a remote PC by a CAN to the USB bridge.

Two CAN interfaces are available for STreamPlug user application: CAN0 and CAN1. The file can_readme.txt available in the folder “/examples/can” describes a set of user space commands to test the CAN interfaces. In particular, it focuses on the following commands:

```
/sbin/ip  link set  <device> type can  bitrate <value>
ifconfig <device>  up
cansend <device>  <can_frame>
candump  <device> >  <outputfile>
```

The following is an example of a Linux session where the previous commands are used to setup the CAN bus and to receive and send simple messages.

```
STreamPlug  login:  root
#
#  cd  /examples/can
#  /sbin/ip  link set can0  type can  bitrate 125000
#  /sbin/ip  -details -statistics link show  can0
2: can0:  <NOARP,ECHO>  mtu  16  qdisc noop  state DOWN  qlen  10
link/can
can  state STOPPED  (berr-counter  tx 0  rx 0) restart-ms  0
bitrate 125000  sample-point  0.875
tq 1000  prop-seg  3  phase-seg1  3  phase-seg2  1  sjw  1
c_can:  tseg1 2..16 tseg2 1..8  sjw 1..4  brp 1..1024 brp-inc  1
clock  83000000
re-started  bus-errors  arbit-lost error-warn  error-pass  bus-off
0  0  0  0  0  0
RX:  bytes packets errors  dropped overrun mcast
0  0  0  0  0  0
TX:  bytes packets errors  dropped carrier collsns
0  0  0  0  0  0
#  ifconfig can0  up
c_can_platform c_can_platform.1:  can0:  setting BTR=0512 BRPE=0001
#  cansend can0  5A1#11.22.33.44.55.66.77.88
#  candump  can0
can0                 888   [8] 01  02  03  04  05  06  07  08
can0                 888   [8] 01  02  03  04  05  06  07  08
can0                 888   [8] 01  02  03  04  05  06  07  08
can0                 888   [8] 01  02  03  04  05  06  07  08
```

Other tests can be performed with the can utils utilities provided with the auxiliary filesystem.

4.9 Fast infrared data association (FIrDA)

The FIrDA IP is a Fast infrared controller that provides an interface to infrared wireless devices. It supports three IrDA™ modes, SIR, MIR and FIR. The transfer speed ranges from 2400 bps (SIR) to 4 Mbps (FIR). The standard mode, SIR, accesses the infrared port through a serial interface. The faster modes, MIR and FIR, require special support handling in Linux. In general, as reported by “irattach” manual page, Linux FIR support is not as stable and mature as SIR or MIR.

4.9.1 FIrDA software overview

The FIrDA Linux device driver belongs to the Linux network subsystem. The device may enable and be configured from the command line into the OK Linux XML cell configuration file. The FIrDA Linux device driver is initialized using platform data provided by the command line. In particular the QOS (Quality Of Service) parameters are set at driver initialization. The QoS parameters are used by IrLAP protocol during the negotiation phase with IrDA peers.

4.9.2 FIrDA kernel source and configuration

The most important source files for this device driver are:

`drivers/net/irda/dice_fir.c`

`include/linux/dice_fir.h`

However, other aspects of it are defined in the following files:

`arch/arm/mach-streamplug/ipswrst_ctrl.c`

`arch/arm/mach-streamplug/include/mach/generic.h`

`arch/arm/mach-streamplug/include/mach/streamplug10.h`

`arch/arm/mach-streamplug/clock.c`

`arch/arm/mach-streamplug/padmux.c`

`arch/arm/mach-streamplug/streamplug_devel_board.c`

`arch/arm/mach-streamplug/streamplug1x.c`

In order to support the FIrDA device driver it is necessary to enable the following options in *Table 14* within the STreamPlug Linux kernel configuration file.

Table 14. FIrDA Linux kernel configuration

Configuration	Description
CONFIG_IRDA=y	Add support for the IrDA™ protocols.
CONFIG_IRLAN=y	Add support for IrLAN protocol.
CONFIG_IRCOMM=y	Add support for the IrCOMM protocol.
CONFIG_DICE_FIR=y	Add support for FIR
CONFIG_IRDA_DEBUG_DEVICE=y	Activate debugging of IrDA device drivers

4.9.3 FIrDA platform configuration

The FIrDA device driver is characterized by the following STreamPLug FIrDA platform C structures:

```
/* Fast Irda Controller registration  */
static struct resource irda_resources[]  =  {
   {
      .start  =  STREAMPLUG1X_ICM1_FIRDA_BASE,
      .end  =  STREAMPLUG1X_ICM1_FIRDA_BASE  +  SZ_4K - 1,
      .flags  =  IORESOURCE_MEM,
   }, {
      .start  =  STREAMPLUG1X_IRQ_LOW_SPEED_SUBS_FIRDA_CAN2,
      .flags  =  IORESOURCE_IRQ,
   },
};

#if  defined (CONFIG_DICE_FIR)
static struct dice_fir_platform_data irda_platform_data;
#endif
struct platform_device  streamplug1x_irda_device =  {
#if  defined (CONFIG_DICE_FIR)
   .name  =  "dice_fir",
#elif  defined (CONFIG_DICE_IR)
   .name  =  "dice_ir",
#endif
   .id  =  -1,
   .num_resources  =  ARRAY_SIZE(irda_resources),
   .resource  =  irda_resources,
#if  defined (CONFIG_DICE_FIR)
   .dev.platform_data  =  &irda_platform_data,
#endif
};
```

4.9.4 FIrDA usage

To enable the FIrDA support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

A user space application, “irda_xfer” is provided in the STreamPlug filesystem (/examples/irda). It shall be used to test IrDA functionalities using two STreamPlug boards wire connected via the FIrDA I/Os. Since the IrDA is a network interface, the first step is the activation of the network IrDA interface using the ifconfig command.

```
$  ifconfig  irda0 up
```

Finally, a file transfer can be established by using run “irda_xfer”. It starts a discover procedure to determine whether there are any IrDA peers and upon the discovery it transfers the files indicated in the command argument between them.

To put side “A” in the listening mode, use:

```
$ ./irda_xfer -r
```

To send the file from the side B to side A, use :

```
$ ./irda_xfer -s</path/irdaInputFile>
```

An example of a session that sends a file through the IrDA interface is shown below.

```
# uname -a
Linux STreamPlug 2.6.35-vcpu-okl_streamplug+ #76 Mon Aug 27 14:48:15
CEST 2012 vcpuv5-el GNU/Linux
# pwd
/examples/irda
# cat hello_world.txt hello world
# ifconfig
irda0 Link encap:UNSPEC HWaddr C0-70-0B-A0-1D-00-00-00-00-00-00-00-00-00-
00-00
UP RUNNING NOARP MTU:2048 Metric:1
RX packets:81 errors:0 dropped:0 overruns:0 frame:0
TX packets:107 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:8
RX bytes:366 (366.0 B) TX bytes:866 (866.0 B) Interrupt:25
# ./irda_xfer -shello_world.txt
804: Using default DeviceIrDA: irda0
807: open socket
133: Waiting for discovery to finish.
122: getsockopt IRLMP_ENUMDEVICES ok, len=40
Discovered:
daddr: 81a3310a saddr: a00b70c0 name: Linux
Connected to 81a3310a mtu = 2039
date: 497
date: 62168263697000000
sent: FILE 12 32 14474676 3656803904 hello_world.txt
Received (5) ACK Y

Sent hello_world.txt, 12 bytes in 1 sec. 0.012 KBytes/sec last read:
Transport endpoint is not connected
#
```

While the following is the dump of a session to receive a file through the same interface.

```
# ifconfig
# ifconfig irda0 up
# ifconfig
irda0 Link encap:UNSPEC HWaddr
56-0A-1D-F4-1D-00-00-00-00-00-00-00-00-00-00-00
UP RUNNING NOARP MTU:2048 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:8
```

```
RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
Interrupt:25
#  ./irda_xfer  -r
804: Using default  DeviceIrDA:  irda0
807: open  socket
mtu=2039
FILE   12  32  14474676   3656803904  hello_world.txt
filename:  hello_world.txt
filesize:  12
Filemode:  32, Modified
fdate1:  14474676   =  0xdcddb4
fdate2:  3656803904  =  0xd9f66640
filedate:  62168263697000000
filedate:  497  =  Thu  Jan  1  00:08:17  1970
Received hello_world.txt,  12  bytes in 1  sec.  0.012 KBytes/sec
#  ls -altr
-rwxrwxrwx  1  1000  1000 625971  Jan  1  1970  irda_xfer
-rw-r--r--  1  1000  1000 1344  Jan  1  1970  irda_readme.txt
-rw-r--r--  1  1000  1000 1341  Jan  1  1970  irdaInputFile.txt drwxr-xr-x
10  1000  1000 0  Jan  1  1970  ..
-rw-r--r--  1  root root12  Jan  1  1970  hello_world.txt drwxr-xr-x  2
1000  1000 0  Mar  3  21:43 .
#  cat hello_world.txt hello  world
#
#  dmesg
bootconsole [early0]  enabled
Linux version 2.6.35-vcpu (xxxx@VirtualBox)  (gcc  version 4.3.3
(Sourcery  G++  Lite 2009q1-203) ) #9  Tue  Aug  28  09:36:05 CEST 2012
CPU: vCPUv5 [14069260] revision 0 (ARMv5TEJ) CPU:  VIVT  data cache,  VIVT
instruction cache Machine:  ST-STREAMPLUG-DEBUG
ATag  virq 4, "timer_tick"
ATag  microvisor_timer 2d,  4, "timer_microvisor_timer"
ATag  vclient 2e,  20, 6, "vserial_vtty0_vclient"
On node  0  totalpages:  8192
free_area_init_node:  node  0, pgdat 843e4eb8,  node_mem_map 847a4000
Normal zone:  64  pages  used  for memmap
Normal zone:  0  pages  reserved
Normal zone:  8128  pages,  LIFO  batch:0
OKL4:  vcpu_helper_page at  847e5000/01fff000
VMMU:paging_init:  VMMU:  Cache  management  handing is  possibly not
correct
(SDK-1545).
Built 1  zonelists in Zone  order,  mobility  grouping on.  Total pages:
8128
```

```
Kernel command line: console=vcon0,115200n8 clcd=off sata=off pcie=off
usb=on:host eth=off i2c=off ssp=off uart1=on:primary uart2=on:primary
can=off firda=on:3 fsmc=off sport=off ts=off
PID hash table entries: 128 (order: -3, 512 bytes)
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes) Inode-
cache hash table entries: 2048 (order: 1, 8192 bytes) Memory: 32MB =
32MB total
Memory: 24652k/24652k available, 8116k reserved, 0K highmem
Virtual Kernel memory layout:
vector : 0x01fff000 - 0x02000000 ( 4 kB)
fixmap : 0xfff00000 - 0xfffe0000 ( 896 kB)
DMA : 0xfb800000 - 0xfc000000 ( 8 MB)
vmalloc : 0x86800000 - 0xf0000000 (1688 MB)
lowmem : 0x84000000 - 0x86000000 ( 32 MB)
modules : 0x83000000 - 0x84000000 ( 16 MB)
.init : 0x84000000 - 0x8401d000 ( 116 kB)
.text : 0x8401d000 - 0x8439d000 (3584 kB)
.data : 0x843b4000 - 0x843e54c0 ( 198 kB)
Hierarchical RCU implementation.
Verbose stalled-CPUs detection is disabled. NR_IRQS:512
Console: colour dummy device 80x30
Calibrating delay loop... 164.24 BogoMIPS (lpj=821248)
pid_max: default: 4096 minimum: 301
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
VCPU support v1.0
NET: Registered protocol family 16
padmux: dev name usb
padmux: dev name uart1
padmux: dev name uart2
padmux: dev name firda
Serial: AMBA PL011 UART driver
uart1: ttyAMA0 at MMIO 0xd0000000 (irq = 26) is a AMBA/PL011
uart2: ttyAMA1 at MMIO 0xd0080000 (irq = 27) is a AMBA/PL011
bio: create slab <bio-0> at 0
vServices Framework 1.0 registering driver 0
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
NET: Registered protocol family 23
Switching to clocksource microvisor timer
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 1024 (order: 1, 8192 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
```

```
TCP: Hash tables configured (established 1024 bind 1024) TCP reno
registered
NET: Registered protocol family 1
Trying to unpack rootfs image as initramfs... Freeing initrd memory:
3724K
VFS: Disk quotas dquot_6.5.2
Dquot-cache hash table entries: 1024 (order 0, 4096 bytes)
JFFS2 version 2.2. (NAND) © 2001-2006 Red Hat, Inc.
msgmni has been set to 55
alg: No test for stdrng (krng)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
vServices Microvisor Transport v1.0
registering driver 0
OKL4 virtual console init
console [vcon0] enabled, bootconsole disabled
brd: module loaded
registering driver 0
st: Version 20081215, fixed bufsize 32768, s/g segs 256 smi smi: mtd
.name=w25q64_bank0 .size=800000(8M)
smi smi: .erasesize = 0x10000(64K)
Creating 1 MTD partitions on "w25q64_bank0":
0x000000000000-0x000000800000 : "Reserved"
smi smi: mtd .name=w25q64_bank1 .size=800000(8M)
smi smi: .erasesize = 0x10000(64K)
Creating 1 MTD partitions on "w25q64_bank1":
0x000000000000-0x000000800000 : "Auxiliary Root File System" smi smi:
mtd .name=w25q64_bank2 .size=800000(8M)
smi smi: .erasesize = 0x10000(64K)
Creating 1 MTD partitions on "w25q64_bank2":
0x000000000000-0x000000800000 : "Root File System"
CAN device driver interface
vs_ethernet_server_init - registering registering driver 0
usbcore: registered new interface driver asix
usbcore: registered new interface driver cdc_ether
usbcore: registered new interface driver cdc_eem
usbcore: registered new interface driver net1080
dice_fir dice_fir: DICE Fast IrDA probed successfully
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
streamplug-ehci streamplug-ehci: STREAMPLUG EHCI
streamplug-ehci streamplug-ehci: new USB bus registered, assigned bus
number 1
streamplug-ehci streamplug-ehci: irq 5, io mem 0xe1800000
streamplug-ehci streamplug-ehci: USB 0.0 started, EHCI 1.00
hub 1-0:1.0: USB hub found
```

```
hub 1-0:1.0: 1 port detected
ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
streamplug-ohci streamplug-ohci.0: STREAMPLUG OHCI
streamplug-ohci streamplug-ohci.0: new USB bus registered, assigned bus
number 2
streamplug-ohci streamplug-ohci.0: irq 5, io mem 0xe1900000
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 1 port detected
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
Loading designware_udc:
rtc-streamplug rtc-streamplug: rtc core: registered rtc-streamplug as
rtc0
i2c /dev entries driver
sp805-wdt wdt: registration successful
dw_dmac: DesignWare DMA Controller, 8 channels
TCP cubic registered
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
can: controller area network core (rev 20090105 abi 8)
NET: Registered protocol family 29
can: raw protocol (rev 20090105)
can: broadcast manager protocol (rev 20090105 t)
IrCOMM protocol (Dag Brattli)
802.1Q VLAN Support v1.8 Ben Greear <greearb@candelatech.com>
All bugs added by David S. Miller <davem@redhat.com> registering driver
0
rtc-streamplug rtc-streamplug: setting system clock to 1990-03-03
21:35:15 UTC (636500115)
Freeing init memory: 116K
net irda0: dice_fir_set_baudrate IrDA mode FIR -> SIR mode
net irda0: dice_fir_set_baudrate IrDA mode SIR -> FIR mode
```

Further IrDA utilities (provided by the buildroot for the auxiliary filesystem) or IrTTY device driver functionalities (SIR only) can be used for IrDA test purposes.

4.10 Peripheral component interconnect express (PCIe)

The PCIe is an important serial bus protocol which is commonly used for peripheral expansion. Gen1 operates at 2.5 Gbits/s. It is very similar to legacy PCI from the user's perspective.The upper software layer is the same as that of PCI. The lower layer has some PCIe-specific read/write configuration. When software boots, it determines which devices are connected downstream and at what speed (Gen1). Then it creates a map for the entire downstream device, which is further used by a device specific driver.

The STreamPlug PCIe controller is a dual mode controller, which can work as:

- Root complex (RC, host)
- Endpoint (EP, device)

4.10.1 PCIe software overview

Both the PCIe host and device controllers require initial PCIe modules configured.

The IP initialization (“arch/arm/mach-streamplug/streamplu1x.c”) code is shown below.

```
static int pcie_init(struct device *dev,  void     iomem  *mmio)
{
   int ret; set_pcie_reset_disable();
   set_pcie_clock_enable();
   set_ltssm_disable();
   set_uport_reset_disable();
   set_uport_clock_enable();
   set_pcie_core_reset_n_release();
   set_uport_reset_enable();
   set_uport_clock_disable();
   set_pcie_core_reset_n_setup();
   set_uport_reset_disable();
   set_uport_clock_enable();
   set_pcie_core_reset_n_release();

   ret =  miphy_pipew_completion();
   if (ret) {
      dev_err(dev,  "failed miphy   init pipew completion\n");
      goto err;
   }
```

```
   miphy_write(MIPHY_TX_DETECT_POLL_REG,  0x08);

   ret =  miphy_pll_idll_ready();
   if (ret) {
      dev_err(dev,  "failed pll/idll not ready\n");
      goto err;
   }

   return 0;

   err:
      return ret;
}
```

The initialization happens in the following order:

```
set_pcie_reset_disable()
```

The “arch/arm/plat-streamplug/misc.c” routine removes the PCIe controller from the reset state as shown below.

```
void set_pcie_reset_disable(void)
{
   u32  val;

   val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  expi_sub_swrst_reg))
);
   val &= ~EXPI_SUB_PCIE_SWRST_MASK;
   misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  expi_sub_swrst_reg))
);
}
```

set_pcie_clock_enable()

The “arch/arm/plat-streamplug/misc.c” routine configures the clock for the PCIe controller as shown below:

```
void set_pcie_clock_enable(void)
{
   u32  val;

  val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  expi_sub_clk_enb_reg))
);
   val |= EXPI_SUB_PCIE_CLKENB_MASK;
  misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  expi_sub_clk_enb_reg))
);
}
```

set_ltssm_disable()

The “arch/arm/plat-streamplug/misc.c” routine disables ltssm as shown below.

```
void set_ltssm_disable(void)
{
   u32  val;

   val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  pcie_ctr))  );
   val &= ~PCIE_CTR_LTSSM_ENB_MASK;
   misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  pcie_ctr))  );
}
```

set_uport_reset_disable()

The “arch/arm/plat-streamplug/misc.c” routine removes the PCIe UPort™ from the reset state as shown below.

```
void set_uport_reset_disable(void)
{
   u32  val;

   val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  expi_sub_swrst_reg))
);
   val &= ~EXPI_SUB_UPORT_SWRST_MASK;
   misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  expi_sub_swrst_reg))
);
}
```

set_uport_clock_enable()

The “arch/arm/plat-streamplug/misc.c” routine configures the clock for the UPort controller as shown below.

```
void set_uport_clock_enable(void)
{
   u32  val;

  val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  expi_sub_clk_enb_reg))
);
   val |= EXPI_SUB_UPORT_CLKENB_MASK;
  misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  expi_sub_clk_enb_reg))
);
}
```

set_pcie_core_reset_n_release()

The “arch/arm/plat-streamplug/misc.c” removes the PCIe core from the reset state as shown below.

```
void set_pcie_core_reset_n_release(void)
{
   set_pcie_pcie_core_reset_n_release();
   set_pcie_pipew_core_reset_n_release();
}
```

wherein,

set_pcie_pcie_core_reset_n_release()

The “arch/arm/plat-streamplug/misc.c” removes the core from the reset state as shown below.

```
void set_pcie_pcie_core_reset_n_release(void)
{
   u32 val;
val = misc_readl((u32* )( GetOffset(sMiscRegs, pcie_ctr)) );
   val |= PCIE_CTR_CORE_RST_N_MASK;
   misc_writel(val, (u32* )( GetOffset(sMiscRegs, pcie_ctr)) );
}
```

and

set_pcie_pipew_core_reset_n_release()

the “arch/arm/plat-streamplug/misc.c” removes the pipe wrapper from the reset state as shown below.

```
void set_pcie_pipew_core_reset_n_release(void)
{
   u32  val;

   val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  pcie_ctr))  );
   val |= PCIE_CTR_PIPEW_RTS_N_MASK;
   misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  pcie_ctr))  );
}
```

set_uport_reset_enable()

The “arch/arm/plat-streamplug/misc.c” puts the UPort controller into the reset state, as shown below.

```
void set_uport_reset_enable(void)
{
   u32  val;

   val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  expi_sub_swrst_reg))
);
   val |= EXPI_SUB_UPORT_SWRST_MASK;
   misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  expi_sub_swrst_reg))
);
}
```

set_uport_clock_disable()

The “arch/arm/plat-streamplug/misc.c”, switch off the UPort controller clock is shown below.

```
void set_uport_clock_disable(void)
{
   u32  val;

  val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  expi_sub_clk_enb_reg))
);
   val &= ~EXPI_SUB_UPORT_CLKENB_MASK;
  misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  expi_sub_clk_enb_reg))
);
}
```

set_pcie_core_reset_n_setup()

This code puts the PCIe core into the reset state, as shown below.

```
void set_pcie_core_reset_n_setup(void)
{
   u32  val;

   val =  misc_readl((u32*  )( GetOffset(sMiscRegs,  pcie_ctr))  );
   val &= ~(PCIE_CTR_CORE_RST_N_MASK  | PCIE_CTR_PIPEW_RTS_N_MASK);
   misc_writel(val,  (u32*  )( GetOffset(sMiscRegs,  pcie_ctr))  );
}
```

miphy_pipew_completion()

The “arch/arm/mach-streamplug/miphy.c” waits for the MIPHY Pipe Wrapper Configuration completed as shown below.

```
int miphy_pipew_completion(void)
{
   unsigned char val;
   unsigned int ucount=0;
   do  {
      val =  miphy_read(MIPHY_RX_BUFFER_REG);
      if (val ==  MIPHY_PIPEW_COMPL)  {
         return 0;
      }
      udelay(1);
      ucount++;
   } while (ucount  <=  (MIPHY_PIPEW_COMPL_TIMEOUT*100*1000));

   return 1;
}
```

After enabling the “TX_POLL bit from MIPHY” regs. space, it checks the MiPHY PII/iDII.

```
int miphy_pll_idll_ready(void)
{
   unsigned char val;
   unsigned int ucount=0;

   do  {
      val =  miphy_read(MIPHY_STATUS_REG);
      if ((val & MIPHY_PLL_IDLL_MASK)  ==  MIPHY_PLL_IDLL_MASK)  {
         return 0;
      } udelay(1); ucount++;
   } while (ucount  <=  (MIPHY_PLL_IDLL_RDY_TIMEOUT*100*1000));

   return 1;
}
```

PCIe root complex

The PCIe root complex device driver performs the following steps in order to complete the PCIe interface initialization:

- PCIe header type 0 configuration.
- Enabled the inbound transactions, for which two memory windows ranges have been opened within local DDR memory space:
  - From 0x00000000 to 0x00047FFF
  - From 0x10000000 to 0x7FFFFFFF
- Enabled the entire PCIe AHB space for outbound transactions:
  - From 0x40000000 to 0x4000FFFF; this memory window provides accesses in configuration space. According to the STreamPlug design architecture, the PCIe RC device driver supports connections until eight single function endpoints are connected by a PCIe switch. The current PCIe RC device driver will interface with only one single endpoint due to the absence of a switch. In that way the configuration space of the single endpoint will be found within the range [0x40000000-0x40000FFF].
- From 0x40010000 to 0x5FFFFFFF; this memory window provides 32-bit accesses in memory space (with or without prefetchable option) of the target endpoint.
- From 0x60000000 to 0xBFFFFFFF; the local base addresses of the low memory spaces defined for the root complex, have been set as:
  - BAR0 = 0x60000000 with size 512 Mbyte
  - BAR1 = 0x80000000 with size 1 Gbyte
- Enable the LTSSM machine state in order to negotiate the Link-up.
- INTA: HW interrupt RX handling is initialized; the PCIe INTA interrupt routine is attached to the VIC line 3.
- MSI: SW interrupt RX handling is initialized; the PCIe RC will retrieve the information about MSI from the target endpoint and set the MSI capabilities.
- Bus enumeration, in order to find the target endpoint in case the link is up.

PCIe target scanning, during that phase the endpoint will be configured according to its capabilities in order to perform memory transactions in both directions.

Note that:

- Only one endpoint will be supported when the STreamPlug is configured as a root complex
- “PCIE_LINK_REQ_RST_NOT_ITS” interrupt is left masked. Otherwise, it can cause the link disconnection.

PCIe endpoint

The PCIe endpoint device driver performs the following steps in order to complete the PCIe interface initialization:

- PCI header type 0 configuration
  - Enable bus master
  - Enable memory space access
  - Base addresses mask, to define the size of the three inbound windows:
- BAR0 mask, 32-bit memory access size 128 Mbyte
- BAR1 mask, 32-bit memory access size 2 Mbyte
- BAR2 mask, 32-bit memory access size 2 Mbyte
- Enabled the address translation:
  - The inbound transactions, for each BAR the following inbound window range has been defined
- Local DDR memory, from 0x00000000 to 0x08000000
- Lower peripheral interface, from 0xD0000000 to 0xD01FFFFF
- CLCD peripheral controller, from 0xFC200000 to 0xFC21FFFF
  - The outbound transactions; two memory windows ranges are defined to perform outbound transactions towards the root complex:
- From 0x40000000 to 0x47FFFFFF, it will be translated into target RC memory space starting from 0x00000000
- From 0x50000000 to 0x57FFFFFF, it will be translated into target RC memory space starting from 0x01000000

4.10.2 PCIe kernel source and configuration

The following details correspond to the layout of the driver and kernel configuration.

The PCI driver stack is in “drivers/pci”. This is common for all PCI and PCIe controllers.

Table 15 illustrates the kernel Kconfig option required to be enabled for the PCIe.

Table 15. PCIe configurations

Configuration	Description
CONFIG_PCI	Enable the PCI bus system.
CONFIG_PCIEPORTBUS	Enable PCI express port bus support.

These are required to enable kernel PCIe generic support, while the detailed options for the two possible configuration (the root complex and endpoint) are listed below.

PCIe root complex

The driver of the root complex is in “arch/arm/mach-streamplug”:

- “dw_pcie.c”
- “streamplug1x_pcie_rev_350”, for IP revision 3.50

Table 16 illustrates the kernel Kconfig options required to be enabled for the root complex.

Table 16. PCIe root complex configurations

Configuration	Description
CONFIG_DW_PCIE	Enable the support of the Synopsys designware PCIe dual mode controller.
CONFIG_STREAMPLUG_PCIE_REV350	Enable the ST STreamPlug PCIe Rev 3.50.
CONFIG_PCI_MSI	Enable the drivers to enable MSI (message signaled interrupts).

PCIe endpoint

The driver of the endpoint is in “drivers/misc streamplug1x_pcie_gadget.c”.

Table 17 illustrates the kernel Kconfig options required to be enabled for the endpoint.

Table 17. PCIe endpoint configurations

Configuration	Description
CONFIG_STREAMPLUG1X_PCIE_GADGET	Enable ST STreamPlug PCIe device support.

4.10.3 PCIe platform configuration

Due to mutual exclusive PCIe controllers, the root complex or endpoint platform configuration is set by padmuxing. The files containing the platform configurations for both types of supported PCIe controllers are in “arch/arm/mach-streamplug”:

- “streamplug1x.c”, where are defined the platform device structures
- “padmux.c”, where are defined the padmux option for configuring the PCIe controller as a host or a device.

```
static struct pmx_dev_mode  *pcie_modes[]  =  {
   &pcie_rc_mode,
   &pcie_ep_mode,
};

void parse_pcie_options(struct  pmx_dev  *dev,  char *options)
{
   pcie_clk_opt =  simple_strtoul(options,  NULL,  10);
   printk(KERN_INFO  "[M10]:  %s  pcie_clk_opt =  %d",     func     ,
pcie_clk_opt);
}

DECLARE_PMX_DEV(pcie,  pcie_modes,  PMX_DEV_DISABLE,  parse_pcie_options);
```

PCIe root complex

The PCIe platform device configuration is shown below.

```
struct platform_device  streamplug1x_pcie_host_device =  {
   .name  =  "dw_pcie-rc",
   .id  =  0,
   .dev  =  {
      .coherent_dma_mask  =  ~0,
      .dma_mask  =  &pcie_host_dmamask,
      .platform_data  =  &pcie_host_info,
   },
   .num_resources  =  ARRAY_SIZE(pcie_resources),
   .resource  =  pcie_resources,
};
```

and the “padmux” option to set the PCIe controller as a root complex is:

```
static struct pmx_mux_reg  pcie_rc_regs[]  =  {
   { .reg  =  &pci_sata_sel,  .value  =  0x0  },
   { .reg  =  &pci_device_type_sel,  .value  =  0x1  },
};

static struct pmx_dev_mode  pcie_rc_mode =  {
   .name  =  "rc",
   .mux_regs  =  pcie_rc_regs,
   .mux_reg_cnt  =  ARRAY_SIZE(pcie_rc_regs),
   .platform_dev  =  &streamplug1x_pcie_host_device,
};
```

PCIe endpoint

The PCIe platform device configuration is shown below.

```
struct  platform_device streamplug1x_pcie_gadget_device =  {
   .name  =  "dw_pcie-ep",
   .id  =  0,
   .dev  =  {
      .coherent_dma_mask  =  ~0,
      .dma_mask  =  &pcie_gadget_dmamask,
      .platform_data  =  &pcie_gadget_info,  //pcie_gadget0_id,
   },
   .num_resources  =  ARRAY_SIZE(pcie_resources),
   .resource  =  pcie_resources,
};
```

and the “padmux” option to set the PCIe controller as an endpoint is:

```
static struct pmx_mux_reg  pcie_ep_regs[]  =  {
   { .reg  =  &pci_sata_sel,  .value  =  0x0  },
   { .reg  =  &pci_device_type_sel,  .value  =  0x0  },
};
```

```
static struct pmx_dev_mode  pcie_ep_mode =  {
   .name  =  "ep",
   .mux_regs  =  pcie_ep_regs,
   .mux_reg_cnt  =  ARRAY_SIZE(pcie_ep_regs),
   .platform_dev  =  &streamplug1x_pcie_gadget_device,
};
```

4.10.4 PCIe usage

To enable the PCIe RC/EP support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*. Then PCIe devices/registers can be managed/monitored using any of the following utilities or commands:

- “lspci” is a utility for displaying information about all PCI buses in the system and all devices connected to them. Its details can be seen with man lspci.
- “setpci” is a utility for querying and configuring PCI devices. Its details can also be seen with man setpci.
- The PCI sysfs can be managed using the following commands: “cat/hexdump/echo”. It's possible to access to PCI sysfs of the both PCIe host and device, below “/sys/bus/pci/devices/0000\:0X\00.0/”, where *X* stays for a primary bus number at which the root complex and endpoint are connected.

4.11 Serial advanced technology attachment (SATA)

This chapter describes the functional behavior and software interface of the STreamPlug device driver for the serial advanced technology attachment (SATA) bus controller. The SATA, also called Serial ATA, is the evolution of the Parallel ATA (PATA), a computer bus interface to connect a host PC to physical storage devices such as hard disk drivers and optical drivers.

The STreamPlug SATA controller uses the advanced host controller interface (AHCI), which is a hardware mechanism (PCI class device) that allows software to communicate with Serial ATA devices. In other words AHCI acts as a data movement engine between the system memory and Serial ATA devices.

4.11.1 SATA software overview

The STreamPlug device driver has been inherited from the official Linux kernel AHCI platform controller which uses the libATA library. The libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI/ATA translation for ATA devices according to the T10 SAT specification. Features include power management, S.M.A.R.T., PATA/SATA, ATAPI, port multiplier, hot swapping and NCQ. For any more details have a look to the following Wikipedia page: https://en.wikipedia.org/wiki/LibATA.

Furthermore, some Linux specific information about libATA can be found in the following text document released together with the Linux kernel.

`Documentation/DocBook/libata.tmpl`

4.11.2 SATA kernel source and configuration

Table 18 lists source code files which have been modified to support the SATA on the STreamPlug boards:

Table 18. SATA source code files

File	Description
./drivers/ata/libahci.c	Set OOB timing according the RXOOB_CLK_FREQ to be used.
./drivers/ata/ahci.h	Add two vendor specific registers for the DWC SATA.

To enable the device driver it is necessary to enable the following Linux kernel configuration options as shown in *Table 19*.

Table 19. Linux kernel configuration for SATA support

Configuration	Description
CONFIG_ATA	Activate serial and parallel ATA support
CONFIG_ATA_VERBOSE_ERROR	Activate SATA verbose error reporting
CONFIG_SATA_AHCI_PLATFORM	Activate the AHCI SATA platform

4.11.3 SATA platform configuration

Configuration of the device/driver through platform data or inherently in the driver itself.

4.11.4 SATA usage

To enable the SATA support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

The following shows the log of the Linux kernel startup which detects an attached SATA device.

```
Linux version 2.6.35-vcpu-okl_streamplug (restellil@restellil-laptop) (gcc
version 4.3.3 ?' (Sourcery G++ Lite 2009q1-203) ) #32 Fri Feb 8 08:46:23
CET 2013
CPU: vCPUv5 [14069260] revision 0 (ARMv5TEJ) CPU: VIVT data cache, VIVT
instruction cache Machine: ST-STREAMPLUG
ATag virq 8, "timer_tick"
ATag microvisor_timer 39, 8, "timer_microvisor_timer"
ATag virq 9, "ksp_signal"
ATag ksp_agent 3a, 9, "ksp_ksp_agent"
ATag ksp_shared_mem fd100000, 6c00000, a00000, "shm_KSP_SHARED_MEMORY"
ATag vclient 3b, 20, a, "vserial_vtty0_vclient"
OKL4: vcpu_helper_page at 84c18000/01fff000
VMMU:paging_init: VMMU: Cache management handing is possibly not
correct (SDK-1545). Built 1 zonelists in Zone order, mobility grouping
on.Total pages: 21082
Kernel command line: console=vcon0,115200n8 root=/dev/mtdblock2
rootfstype=ext2,jffs2 ?' clcd=off sata=on:2 pcie=off usb=on:host
eth=off i2c=on ssp=on uart1=off uart2=off ?'
```

```
can=off  firda=off  fsmc=off  sport=off  ts=off
[M10]:  parse_sata_options sata_clk_opt  =  2
PID  hash  table entries:  512  (order:  -1, 2048  bytes)
Dentry cache  hash  table entries:  16384  (order:  4, 65536  bytes) Inode-
cache  hash  table entries:  8192  (order:  3, 32768  bytes) Memory:  83MB
=  83MB  total
Memory:  72868k/72868k available,  12124k  reserved,  0K  highmem
Virtual  Kernel memory  layout:
vector   : 0x01fff000 - 0x02000000     (   4  kB)
fixmap   : 0xfff00000 - 0xfffe0000     ( 896  kB)
DMA      : 0xef800000 - 0xf0000000     (   8  MB)
Vmalloc  : 0x89800000 - 0xe4000000     (1448  MB)
lowmem   : 0x84000000 - 0x89300000     (  83  MB)
modules  : 0x83000000 - 0x84000000     (  16  MB)
.init  : 0x84000000 - 0x84022000  ( 136  kB)
.text  : 0x84022000 - 0x8443b000 (4196  kB)
.data  : 0x84454000 - 0x84487f80  ( 208  kB)
Hierarchical RCU  implementation.
Verbose stalled-CPUs  detection is disabled. NR_IRQS:512
Console:  colour dummy device 80x30
Calibrating  delay loop...  164.24 BogoMIPS  (lpj=821248)
pid_max:  default:  4096  minimum:  301
Mount-cache  hash  table entries:  512
CPU:  Testing write buffer coherency:
ok
VCPU support v1.0
NET:  Registered protocol  family  16
padmux:  dev  name sata
padmux: dev  name usb
padmux: dev  name i2c
padmux:  dev  name ssp
 [STreamPlugDBG]:  miphy  input clock for sata is  qfs4
ENTER set_reference_clock
MiPHY Input clock provided by  QFS EXIT  set_reference_clock
Serial:  AMBA  PL011  UART driver
bio:  create  slab  <bio-0> at  0 vServices Framework  1.0
SCSI  subsystem initialized
usbcore:  registered new interface  driver  usbfs
usbcore:  registered new interface driver hub
usbcore:  registered new device driver  usb
Advanced  Linux Sound  Architecture  Driver Version 1.0.23.
NET:  Registered protocol  family  23
Switching to clocksource microvisor timer
NET:  Registered protocol  family  2
IP route cache  hash  table entries:  1024  (order:  0, 4096  bytes)
```

```
TCP established hash table entries: 4096 (order: 3, 32768 bytes)
TCP bind hash table entries: 4096 (order: 2, 16384 bytes)
TCP: Hash tables configured (established 4096 bind 4096)
TCP reno registered
NET: Registered protocol family 1
Trying to unpack rootfs image as initramfs... Freeing initrd memory:
6584K
VFS: Disk quotas dquot_6.5.2
Dquot-cache hash table entries: 1024 (order 0, 4096 bytes)
JFFS2 version 2.2. (NAND) © 2001-2006 Red Hat, Inc.
fuse init (API version 7.14)
msgmni has been set to 155
alg: No test for stdrng (krng)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
vServices Microvisor Transport v1.0
OKL4 virtual console init
console [vcon0] enabled
brd: module loaded
st: Version 20081215, fixed bufsize 32768, s/g segs 256
PIPEW COMPLETION!!!!!
MIPHY PLL LOCKED!!!!!
reset ff
reset 0
ahci ahci: forcing PORTS_IMPL to 0x1

HBA Capabilities - 0x6726ff80

HBA Capabilities after write 0 - 0x6726ff80

HBA Init after write 0x1 to PI register

PI register read - 0x1

Initializing HBA ... Done

TESTR register read - 0x0

OOB register read - 0x5080f19

Wrote OOBR register value=8204080c for 30 MHz

P0SCTL register read - 0x0
```

```
TESTR register  read - 0x0

OOB  register  read - 0x204080c

Wrote   OOBR  register value=8204080c for 30  MHz
ahci ahci:  AHCI  0001.0300 32  slots 1  ports 3  Gbps  0x1  impl platform
mode
ahci ahci:  flags:  ncq  sntf pm led clo only pmp pio slum  part ccc  apst
scsi0 : ahci
ata1:  SATA max UDMA/133  irq_stat 0x00400040,  connection status  changed
irq 3
smi  smi:  mtd  .name=w25q64_bank0  .size=800000(8M)
smi  smi:  .erasesize  =  0x10000(64K)
Creating 1  MTD  partitions on  "w25q64_bank0":
0x000000000000-0x000000800000  : "Reserved"
smi  smi:  mtd  .name=w25q64_bank1  .size=800000(8M)
smi  smi:  .erasesize  =  0x10000(64K)
Creating 1  MTD  partitions on  "w25q64_bank1":
0x000000000000-0x000000800000  : "Auxiliary Root   File System"
smi  smi:  mtd  .name=w25q64_bank2  .size=800000(8M)
smi  smi:  .erasesize  =  0x10000(64K)
Creating 1  MTD  partitions on  "w25q64_bank2":
0x000000000000-0x000000800000  : "Root  File System"
ssp-pl022  ssp-pl022.0:  ARM  PL022  driver,  device ID:  0x00241022
pl022:  mapped  registers from  0xd0100000 to  8987c000
m25p80  spi0.0: non-JEDEC  variant of m25p80
m25p80  spi0.0: m25p80  (1024 Kbytes)
Creating 1  MTD  partitions on  "w25x40":
0x000000000000-0x000000080000  : "External  SPI  Flash"
CAN device driver interface
usbcore:  registered new interface driver asix
 usbcore:  registered new interface driver cdc_ether
usbcore:  registered new interface driver cdc_eem
usbcore:  registered new interface  driver  net1080
ehci_hcd:  USB 2.0 'Enhanced'  Host   Controller (EHCI)  Driver
streamplug_ehci_hcd_drv_probe 0x89880000
streamplug-ehci  streamplug-ehci:  STREAMPLUG  EHCI
streamplug-ehci  streamplug-ehci:  new USB bus  registered,  assigned bus
number  1
streamplug-ehci  streamplug-ehci:  irq 5, io mem  0xe1800000
streamplug-ehci  streamplug-ehci:  USB 0.0 started,  EHCI  1.00
hub  1-0:1.0: USB hub  found
hub  1-0:1.0: 1  port detected
ohci_hcd:  USB 1.1 'Open'  Host   Controller (OHCI)  Driver streamplug-ohci
streamplug-ohci.0:  STREAMPLUG  OHCI
```

```
streamplug-ohci streamplug-ohci.0: new USB bus registered, assigned bus
number 2 streamplug-ohci streamplug-ohci.0: irq 5, io mem 0xe1900000
hub 2-0:1.0: USB hub found hub 2-0:1.0: 1 port detected Initializing
USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered. Loading designware_udc:
rtc-streamplug rtc-streamplug: rtc core: registered rtc-streamplug as
rtc0 i2c /dev entries driver
sp805-wdt wdt: registration successful
dw_dmac: DesignWare DMA Controller, 8 channels
No device for DAI AKCODEC
set_qfs2_clock - qfs_id = 5
ALSA device list:
No soundcards found. TCP cubic registered
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
can: controller area network core (rev 20090105 abi 8) NET: Registered
protocol family 29
can: raw protocol (rev 20090105)
can: broadcast manager protocol (rev 20090105 t) IrCOMM protocol (Dag
Brattli)
802.1Q VLAN Support v1.8 Ben Greear <greearb@candelatech.com> All bugs
added by David S. Miller <davem@redhat.com>
rtc-streamplug rtc-streamplug: hctosys: invalid date/time ata1: SATA
link up 1.5 Gbps (SStatus 113 SControl 300) ata1.00: ATA-8: KINGSTON
SS100S28G, 110512, max UDMA/100 ata1.00: 15649200 sectors, multi 16:
LBA48 NCQ (depth 31/32) ata1.00: configured for UDMA/100
scsi 0:0:0:0: Direct-AccessATAKINGSTON SS100S2 1105 PQ: 0 ANSI: 5 sd
0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] 15649200 512-byte logical blocks: (8.01 GB/7.46 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't
support DPO or FUA
sda:
sda1
sd 0:0:0:0: [sda] Attached SCSI disk
Freeing init memory: 136K
STreamPlug login: root
```

Please note that the device is enumerated by kernel as the SCSI device, "/dev/sda1" ("sda: sda1"). The user can print the partition table of the SATA driver using the sfdisk command as shown below.

```
$ sfdisk -ls
/dev/sda: 7824600

Disk /dev/sda: 974 cylinders, 255 heads, 63 sectors/track
Warning: The partition table looks like it was made
```

```
for C/H/S=*/246/40  (instead  of 974/255/63).
For   this listing I'll assume  that geometry.
Units =  cylinders of 5038080   bytes,  blocks of 1024  bytes,  counting
from 0

Device Boot   Start End  #cyls  #blocks Id  System
/dev/sda10+  1590- 1590- 7822336   7  NTFS
start:  (c,h,s)  expected (0,51,9) found (0,32,33)
end:  (c,h,s)  expected (1023,245,40) found (973,245,40)
/dev/sda20 -0 0  0  Empty
/dev/sda30 -0 0  0  Empty
/dev/sda40 -0 0  0  Empty total:  7824600   blocks
```

Then, at the user space level, commands are provided to mount the external SATA device and to access the corresponding filesystem. Therefore, the user may mount a FAT32 filesystem on the SATA device (“/dev/sda1”) into the folder “/mnt” just using one of the following command:

```
$  mount  /dev/sda1  /mnt
```

If the SATA drive is formatted as NTFS filesystem, use:

```
$  ntfs-3g  /dev/sda1  /mnt
```

The folder examples/sata, delivered with the LSP package, includes a “readme” file that explains a procedure to test the SATA bus interface.

5 Memory technology devices (MTD)

Memory technology devices (MTD) is a generic subsystem for handling memory technology devices under Linux. MTD provides an abstraction layer for raw Flash devices. It makes it possible to use the same API when working with different Flash types and technologies, e.g.: NAND, OneNAND, NOR, AND, ECC'd NOR, etc.

5.1 Linux MTD framework

MTD provides a generic interface between the device drivers and the upper layers of the system.

Device drivers do not need to know about the storage formats used, such as FTL, FFS2, etc. They only need to provide simple routines for read, write and erase. The presentation of the device's contents to the user in an appropriate form will be handled by the upper layers of the system.

The MTD system is divided into two types of the module: “users” and “drivers”. Drivers are the modules which provide raw read/write/erase access to physical memory devices. Users are like YAFFS or JFFS, they are the modules which use MTD drivers and provide a higher level interface to the user space. JFFS is a file system which runs directly on the Flash, and MTDBLOCK performs no translation.

The user space application can access the Flash device content using the mtdblock nodes (“/dev/mtdblockN”) and the mtdchar nodes (“/dev/mtdN”), either in the raw mode, for example using the MTD utils command, or in the logical mode, by mounting a file system (usually JFFS2) and accessing its files through open/read/write system calls.

MTD kernel configuration

Table 20 is the detail corresponding to the layout of the kernel configuration.

Table 20. MTD configurations

Configuration	Description
CONFIG_MTD	Enable memory technology devices.
CONFIG_MTD_CHAR	Provide a character device for each MTD device present in the system, allowing the user to read and write directly to the memory chips, and also use ioctl() to obtain information about the device, or to erase parts of it.

5.2 Accessing to MTD devices

5.2.1 Raw access from user space

MTD utils can be used to access on Flash devices via the MTD layer. A set of MTD utilities are available in the STreamPlug filesystem.

The MTD project provides a number of helpful tools for handling Flash such as:

- “mtd_debug”: gets info, read and write data or erase the specified MTD device.
- “flash_erase”: erases an erase block of Flash
- “flashcp”: copies data into MTD Flash
- “flash_info”: displays information about Flash devices
- “flash_lock”: lock Flash pages to prevent writing
- “flash_unlock”: unlock Flash pages to allow writing

Information about all MTD devices may be get using the “mtdinfo -a” command:

```
# mtdinfo -a
Count of MTD devices:           1
Present MTD devices:            mtd0
Sysfs interface supported:      yes

mtd0
Name:                           rootfs
Type:                           nor
Eraseblock size:                65536 bytes, 64.0 KiB
Amount of eraseblocks:          160 (10485760 bytes, 10.0 MiB)
Minimum input/output unit size: 1 byte
Sub-page size:                  1 byte
Character device major/minor:   90:0
Bad blocks are allowed:         false
Device is writable:             true
```

At the startup three devices are detected by Linux. They are the NOR Flash memories, through the SMI interface.

5.2.2 Raw access from kernel space

MTD devices can be directly accessed through MTD calls such as “mtd_read”, “mtd_erase” and “mtd_write” can be used to read, erase and write to MTD devices.

In this particular Linux distribution provides support for NAND, NOR Flashes and SRAM chips. MTD calls are mapped on specific functions for each different drivers (one for every device). Refer to *Section 5.3: Flexible static memory controller (FSMC) on page 112*.

The first MTD information is the mtd_info structure. It is retrieved by iterating through all registered MTD devices. This structure is defined in “include/linux/mtd/mtd.h”. This structure contains all necessary informations for the device configuration (the size of the whole device, the minimum size can be erased, etc.), without neglecting its main feature which consists in interfacing between the kernel space and user space.

The following code snippet shows a portion of the “mtd_info” structure.

```
/* following is  defined in 'include/linux/mtd/mtd.h'  */
struct  mtd_info {
   u_char type;
   uint32_t flags;
   uint64_t size;// Total size of the MTD

   /* "Major"  erase size for  the device.
   */
   uint32_t erasesize;
   /* Minimal writable  flash  unit  size.
   * Any  driver registering a  struct  mtd_info must  ensure a  writesize
of
   * 1  or larger.
   */
   uint32_t writesize;

   uint32_t oobsize;   // Amount of OOB data per block (e.g. 16)
   uint32_t oobavail;  // Available OOB bytes per block
   /*
    * If erasesize is a power of 2 then the shift is stored in
    * erasesize_shift otherwise erasesize_shift is zero. Ditto writesize.
    */
   unsigned int erasesize_shift;
   unsigned int writesize_shift;
   /* Masks based on erasesize_shift and writesize_shift */
   unsigned int erasesize_mask;
   unsigned int writesize_mask;

   // Kernel-only stuff starts here.
   const char *name;
   int index;

   /* ecc layout structure pointer - read only ! */
   struct nand_ecclayout *ecclayout;

   /* Data   for  variable erase regions.  If numeraseregions is zero,
   * it means  that  the whole device has  erasesize as  given above.
   */
   int numeraseregions;
   struct mtd_erase_region_info *eraseregions;

   /*
   * Erase is an  asynchronous operation.Device drivers  are supposed
   * to call instr->callback()  whenever the operation completes,  even
   * if it completes with  a  failure.
```

```
   * Callers are supposed to  pass   a  callback function and  wait for it
   * to be  called before writing  to the block.
   */
   int (*erase)  (struct mtd_info *mtd,  struct  erase_info *instr);

   /* This stuff for eXecute-In-Place */
   /* phys is optional and may be set to NULL */
   int (*point) (struct mtd_info *mtd, loff_t from, size_t len,
        size_t *retlen, void **virt, resource_size_t *phys);

   /* We probably shouldn't allow XIP if the unpoint isn't a NULL */
   void (*unpoint) (struct mtd_info *mtd, loff_t from, size_t len);

   /* Allow NOMMU mmap() to directly map the device (if not NULL)
    * - return the address to which the offset maps
    * - return -ENOSYS to indicate refusal to do the mapping
    */
   unsigned long (*get_unmapped_area) (struct mtd_info *mtd,
                 unsigned long len,
                 unsigned long offset,
                 unsigned long flags);

   /* Backing device capabilities for this device
    * - provides mmap capabilities
    */
   struct backing_dev_info *backing_dev_info;

   int (*read)  (struct mtd_info *mtd,  loff_t from,  size_t len, size_t
*retlen,  u_char *buf);
   int (*write) (struct mtd_info *mtd,  loff_t to,  size_t len,  size_t
*retlen,  const u_char *buf);
   int (*panic_write) (struct mtd_info *mtd, loff_t to, size_t len, size_t
*retlen, const u_char *buf);

   int (*read_oob) (struct mtd_info *mtd, loff_t from,
         struct mtd_oob_ops *ops);
   int (*write_oob) (struct mtd_info *mtd, loff_t to,
         struct mtd_oob_ops *ops);

   /*
    * Methods to access the protection register area, present in some
    * flash devices. The user data is one time programmable but the
    * factory data is read only.
    */
   int (*get_fact_prot_info) (struct mtd_info *mtd, struct otp_info *buf,
size_t len);
```

```
  int (*read_fact_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len,
size_t *retlen, u_char *buf);
   int (*get_user_prot_info) (struct mtd_info *mtd, struct otp_info *buf,
size_t len);
  int (*read_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len,
size_t *retlen, u_char *buf);
   int (*write_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t
len, size_t *retlen, u_char *buf);
   int (*lock_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t
len);

   /* kvec-based read/write methods.
      NB: The 'count' parameter is the number of _vectors_, each of
      which contains an (ofs, len) tuple.
   */
   int (*writev) (struct mtd_info *mtd, const struct kvec *vecs, unsigned
long count, loff_t to, size_t *retlen);

   /* Sync  */
   void (*sync)  (struct mtd_info *mtd);

   /* Chip-supported  device locking  */
   int (*lock) (struct mtd_info *mtd,  loff_t ofs,  uint64_t len);
   int (*unlock)  (struct mtd_info *mtd,  loff_t ofs,  uint64_t len);

   /* Power  Management  functions */
   int (*suspend)  (struct mtd_info *mtd);
   void (*resume)  (struct mtd_info *mtd);

   /* Bad  block management  functions */
   int (*block_isbad)  (struct mtd_info *mtd,  loff_t ofs);
   int (*block_markbad)  (struct mtd_info *mtd,  loff_t ofs);

   struct notifier_block reboot_notifier;  /* default mode before reboot */
/* ECC status information */
   struct mtd_ecc_stats ecc_stats;
   /* Subpage  shift (NAND)  */
   int subpage_sft;

   void *priv;

   struct module   *owner;
   struct  device dev;
   int usecount;

   /* If the driver is something smart,  like UBI,  it may need  to maintain
```

```
   * its own reference counting.  The  below functions  are only for
driver.
   * The  driver may register its callbacks.  These  callbacks are not
   * supposed to  be  called by  MTD  users */
   int (*get_device)  (struct mtd_info *mtd);
   void (*put_device)  (struct mtd_info *mtd);
};
```

After retrieving the “mtd_info” structure for the specific MTD device, reading (or writing) is relatively simple.

5.2.3 Access through file system from user space

The MTD partition can be mounted using a file system and then can be used. The NOR Flash partition is only supported by JFFS2. Therefore, make sure that a valid JFFS2 image is already present in the partition to avoid getting a lot of JFFS2 error messages.

```
$  mount  -t jffs2 /dev/mtdblock3  /mnt
$  cp  /tmp/file_name/mnt/
$  ls /mnt file_name
```

5.3 Flexible static memory controller (FSMC)

The FSMC can access a wide variety of memory. NAND and NOR Flashes and four SRAM chips are supported in this Linux distribution and their relative device drivers can be built as a module.

The following device driver modules are loadable using the modprobe command:

- “fsmc_nand.ko” for NAND Flash (located below “/lib/modules/2.6.35/Kernel/drivers/mtd/nand/)”
- “physmap.ko” for NOR Flash (located below “/lib/modules/2.6.35/Kernel/drivers/mtd/maps)”.

A NAND Flash example is:

```
#  modprobe   fsmc_nand.ko
```

As a result, a new MTD device is created. (i.e.: “mtd3”).

- # mtdinfo -m3 mtd3
- Name: External NAND Flash
- Type: NAND
- Erase block size: 16384 bytes, 16.0 KiB
- Amount of erase blocks: 4096 (67108864 bytes, 64.0 MiB) minimum input/output unit size: 512 bytes
- Subpage size: 512 bytes OOB size:16 bytes character device major/minor: 90:6
- Bad blocks are allowed: true
- Device is writable: true

In order to access to the memory device, it is necessary to get the device major and minor numbers. These numbers are used to create a special character file with the command “mknod”.

```
#  mknod  /dev/mtd_nand  c  90  6
```

Finally, FSMC functionalities may be tested using the provided MTD utilities.

5.3.1 NAND, FSMC

The NAND Flash is a non-volatile memory with a data access width of 8 or 16 bits. The read and write operations are done in pages (typically 512 or 2048 bytes) while the erase operation is done in erase blocks (the block size is typically of 16 Kbits or 64 Kbits). The NAND Flash is I/O mapped and requires a relatively complicated driver for any operation.

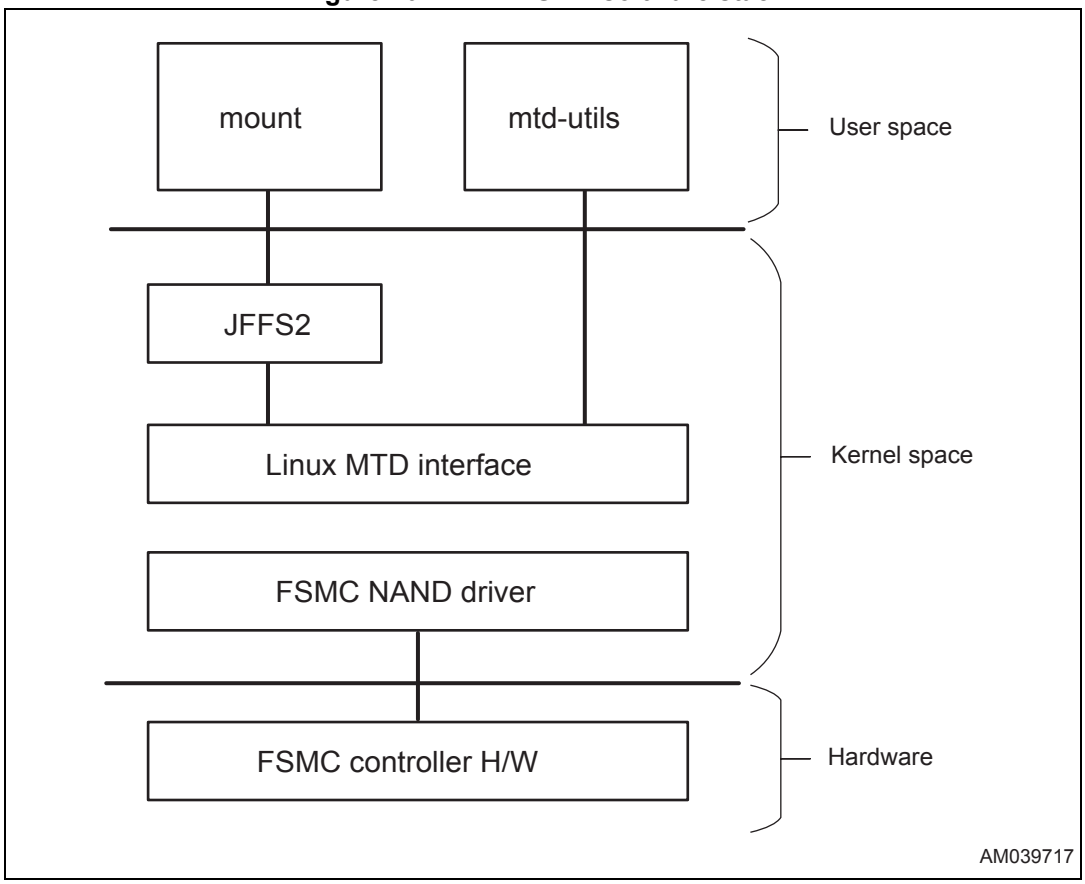
Nowadays, the NAND technology allows bigger size parts at a lower cost, but also with a lower reliability. The main issues with NAND technology are bit flipping and bad blocks. To correct bit flipping, the NAND controller and the driver use the error detection/correction code (EDC/ECC). The second issue requires the use of bad block management techniques.

The higher density, lower cost, faster write/erase times, and a longer rewrite life expectancy make the NAND Flash especially well suited for consumer media applications.

NAND, FSMC software overview

The NAND device driver layer sits between the FSMC HW controller and the Linux MTD interface of the SW stack. The NAND device driver provides all the necessary functions for a file system via the standard Linux MTD interface at the top layer and controls the functionality of the lower layer by using the available API as shown in *Figure 10*.

Figure 10. NAND FSMC software stack



Users can erase, read and write to the NAND devices through the standard MTD interface.

NAND, FSMC source and configuration

Table 21 lists the details corresponding to layout of the kernel configuration:

Table 21. FSMC NAND configurations

Configuration	Description
CONFIG_MTD	Provide the generic support for MTD drivers to register themselves with the kernel.
CONFIG_MTD_NAND	Enable support for accessing all types of NAND Flash devices.
CONFIG_MTD_NAND_FSMC	Enable support for NAND Flash chips on the STMicroelectronics flexible static memory controller (FSMC).

NAND, FSMC platform configuration

This section lists the driver's platform interface and its possible configuration. The default configuration of the FSMC controller depends on the platform data passed in the board definition under the machine folder. The platform configuration is implemented in the following routine:

```
/* set nand  device's  plat  data */

/* following is  defined in 'arch/arm/mach-
streamplug/streamplug_devel_board.c'  */

/* set nand  device's  plat  data */
fsmc_nand_set_plat_data(&streamplug1x_fsmc_nand_device,
      NAND_SKIP_BBTSCAN,
      FSMC_DEVICE_WIDTH8);

/* following is  defined in 'arch/arm/plat-streamplug/include/plat/fsmc.h'
*/

/* This function is used  to set platform data field  of pdev->dev */
static inline void fsmc_nand_set_plat_data(struct  platform_device *pdev,
      struct mtd_partition *partitions,  unsigned int nr_partitions,
      unsigned int options,  unsigned int width, unsigned int init_to_do)
{
   struct fsmc_nand_platform_data *plat_data;
   plat_data =  dev_get_platdata(&pdev->dev);

   if (partitions) {
      plat_data->partitions =  partitions;
      plat_data->nr_partitions =  nr_partitions;
   }

   plat_data->ale_off =  PLAT_NAND_ALE;
   plat_data->cle_off =  PLAT_NAND_CLE;

   plat_data->options =  options;
   plat_data->width =  width;
}
```

NAND, FSMC usage

To enable the FSMC NAND support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

The following utilities are specific for using the NAND Flash through the MTD interface:

```
nandtest - Perform Integrity Data test on the specified Nand mtd device.
nandwrite - Writes an input file to the specified Nand mtd device.
nanddump - Dumps the contents of a Nand mtd partition.
```

5.3.2 Parallel NOR, FSMC

Reading from the NOR Flash is similar to reading from a random access memory, the provided address and data bus are mapped correctly. Because of this, most microprocessors can use the NOR Flash memory as an execute in place (XIP) memory, meaning that programs stored in the NOR Flash can be executed directly from the NOR Flash without needing to be copied into the RAM first. The NOR Flash may be programmed in a random access manner similar to reading. Programming changes bits from a logical one to a zero. Bits that are already zero are left unchanged. Erasure must happen a block at a time, and resets all the bits in the erased block back to logical one. Typical block sizes are 64, 128, or 256 Kbytes.

Bad block management is a relatively new feature in NOR chips. In older NOR devices not supporting bad block management, the software or the device driver controlling the memory chip must correct for blocks that wear out, or the device will cease to work reliably.

The specific commands used to lock, unlock, program, or erase NOR memories differ for each manufacturer. To avoid needing unique driver software for every device made, special common Flash memory interface (CFI) commands allow the device to identify itself and its critical operating parameters.

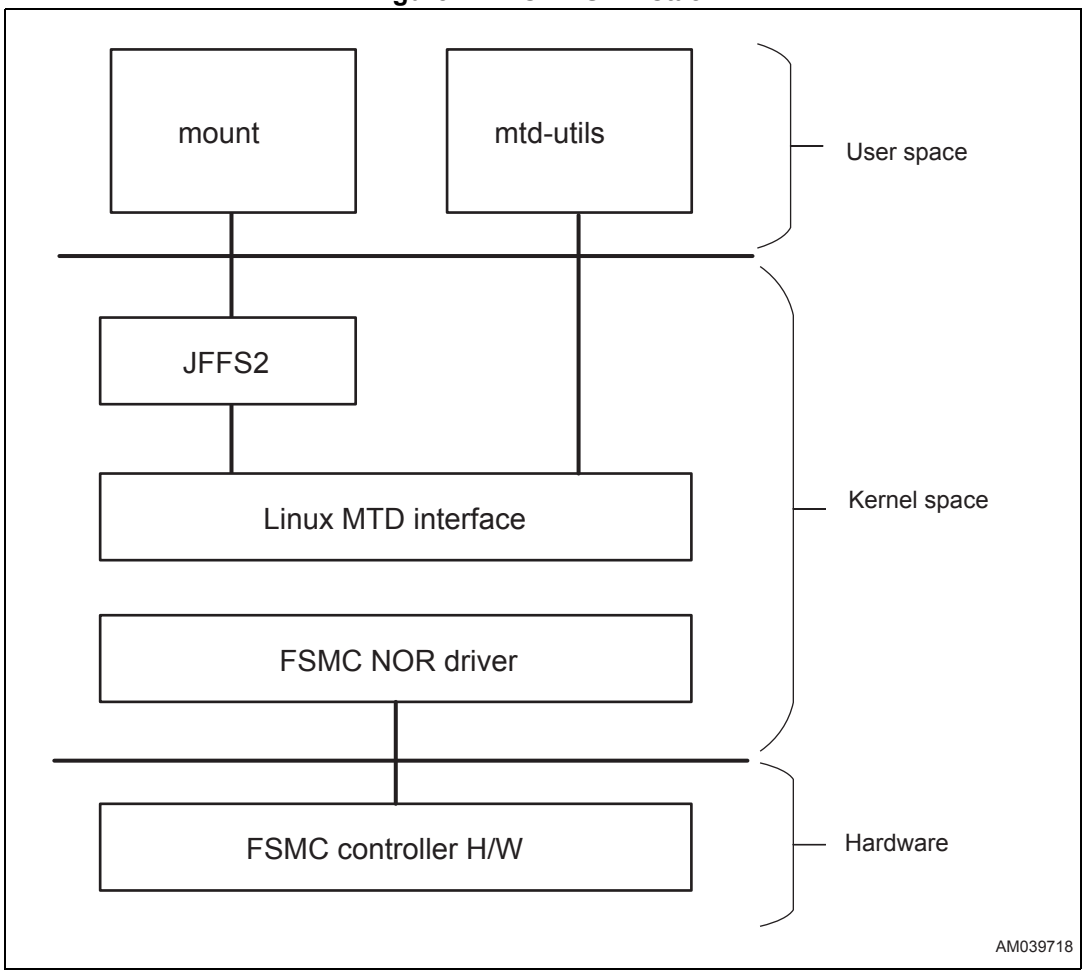
Besides using the NOR memory as a random access ROM, it can also be used as a storage device by taking advantage of random access programming. Some devices offer read-while-write functionality so that the code continues to execute even while a program or erase operation is occurring in the background. For sequential data writes, NOR Flash chips typically have slow write speeds, compared with the NAND Flash.

Parallel NOR, FSMC software overview

The NOR device driver sits on the top of the FSMC HW controller and provides all the necessary functions for a file system via the standard Linux MTD interface. The NOR device driver controls the functionality of the FSMC by using the API of the FSMC driver.

The user can access, read and write to the NOR devices through the standard MTD interface as illustrated in *Figure 11*.

Figure 11. NOR FSMC stack



Parallel NOR, FSMC source and configuration

The NOR Flash device driver provides a mapping driver which allows the NOR Flash and ROM driver code to communicate with chips which are mapped physically into the CPU's memory. The physical address and size and bus width of the Flash chips being used will need to be configured either statically with config options or at run-time.

Table 22 lists the details corresponding to the layout of the driver and kernel configuration.

Table 22. FSMC NOR configurations

Configuration	Description
CONFIG_MTD	Provide the generic support for MTD drivers to register themselves with the kernel.
CONFIG_MTD_PHYSMAP	Allow mapping of the NOR Flash in the physical memory.
CONFIG_STREAMPLUG_FSMC	Enable the AHB master interface to FSMC memories.

The FSMC physmap driver is present in “drivers/mtd/maps/physmap.c”.

Parallel NOR, FSMC platform configuration

This section lists the driver's platform interface and its possible configuration. Default configuration of the FSMC controller depends on the platform data passed.

```
/* set default physmap's  plat  data */

/* following is  defined in 'arch/arm/mach-
streamplug/streamplug_devel_board.c'  */

/* fsmc   NOR  partition info */
static  struct mtd_partition fsmc_nor_partition_info[]  =  {
   PARTITION("External  NOR  Flash",  0x00000000,  SZ_64M),
};

/* NOR  16  */
/* initialize fsmc   related data in fsmc   plat  data */
fsmc_init_nor_board_info(&streamplug1x_fsmc_nor_device,
      fsmc_nor_partition_info,
      ARRAY_SIZE(fsmc_nor_partition_info),
      FSMC_DEVICE_WIDTH16);

/* following is  defined in 'arch/arm/mach-streamplug/fsmc-nor.c'  */

void     init fsmc_init_nor_board_info(struct  platform_device *pdev,
      struct mtd_partition *partitions,
      unsigned int nr_partitions, unsigned int width)
{
   fsmc_nor_set_plat_data(pdev,  partitions,  nr_partitions,  width);
}

/* following is  defined in 'arch/arm/plat-streamplug/include/plat/fsmc.h'
*/

static inline void fsmc_nor_set_plat_data(struct  platform_device *pdev,
      struct mtd_partition *partitions,
```

```
        unsigned int nr_partitions, unsigned int width)
{
   struct physmap_flash_data *plat_data;
   plat_data =  dev_get_platdata(&pdev->dev);

   if (partitions) {
      plat_data->parts =  partitions;
      plat_data->nr_parts =  nr_partitions;
   }

   plat_data->width =  width;
}
```

Parallel NOR, FSMC usage

Refer to *Section 5.1: Linux MTD framework on page 107* for details on using the NOR Flash through the MTD interface both from the kernel and user space.

5.3.3 Static RAM (SRAM), flexible static memory controller

The SRAM (Static RAM) is a type of the RAM that stores data in transistor circuits. The static RAM is faster than the dynamic RAM and does not need to be continuously refreshed.

The family of the SRAM memory can be divided into:

- Asynchronous - independent of clock frequency, data in and data out are controlled by an address transistor.
- Synchronous - all timing are initiated by the clock edge. The address, data and control signal are associates with the clock signals.

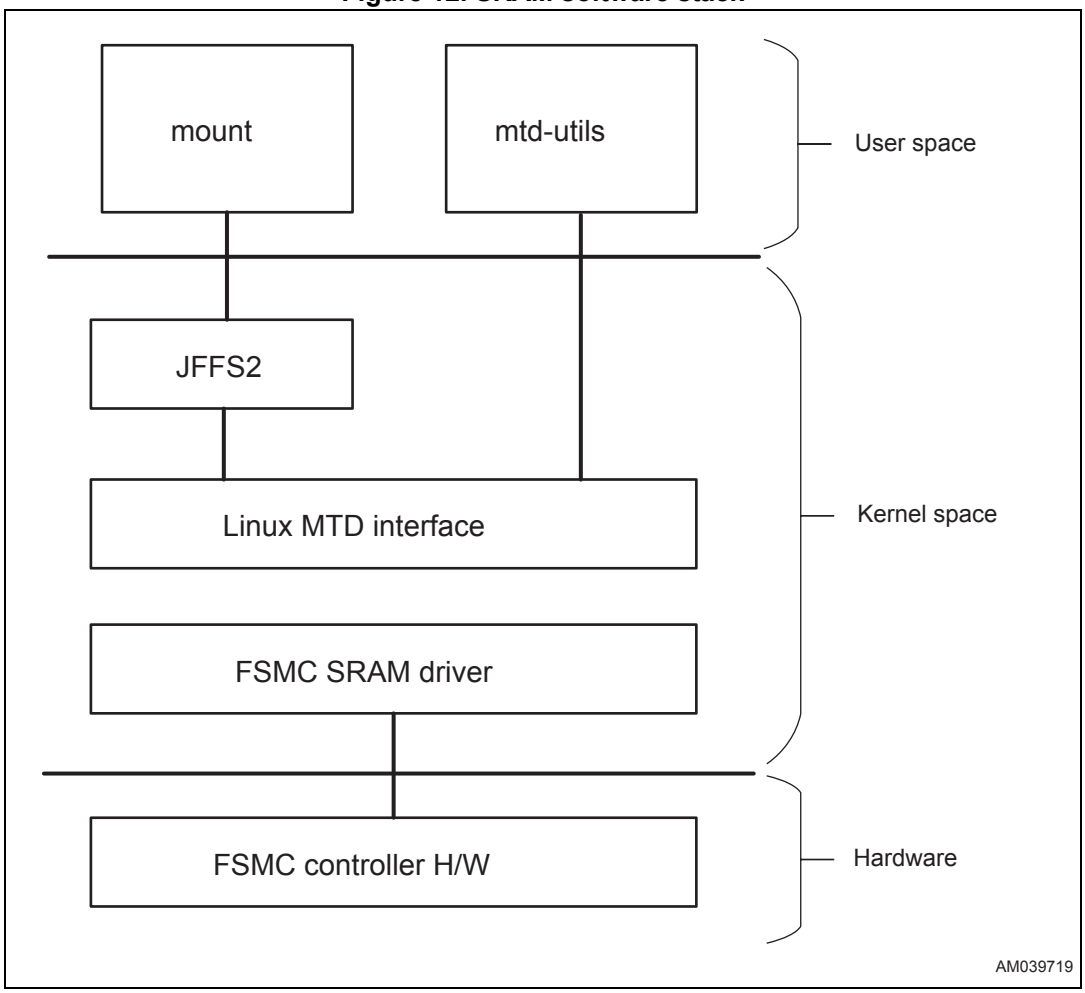
The FSMC device driver flexibility allows supporting the asynchronous static SRAM.

Asynchronous SRAMs are available from 4 kbytes to 64 Mbytes. The fast access time of the SRAM makes the asynchronous SRAM appropriate as a main memory for small cache less embedded processors used in everything from industrial electronics and measurement systems to hard disks and networking equipment, among many other applications. They are used in various applications like switches and routers, IP phones, to automotive electronics.

SRAM software overview

The SRAM device driver sits on the top of the FSMC HW controller and provides all the necessary functions for a file system via the standard Linux MTD interface as shown in *Figure 12*.

Figure 12. SRAM software stack



The FSMC device driver support is limited to asynchronous devices, but the controller also supports synchronous devices.

SRAM kernel source and configuration

Table 23 lists the details corresponding to the layout of the driver and kernel configuration:

Table 23. FSMC SCRAM configurations

Configuration	Description
CONFIG_MTD	Provide the generic support for MTD drivers to register themselves with the kernel.
CONFIG_MTD_PLATRAM	Map SRAM Flash for RAM areas described via the platform device system (MTD-RAM).
CONFIG_STREAMPLUG_FSMC	Enable the AHB master interface to FSMC memories.

SRAM platform configuration

This section lists the driver's platform interface and its possible configuration. Default configuration of the FSMC controller depends on the platform data passed.

```
/* following is  defined in 'arch/arm/mach-
streamplug/streamplug_devel_board.c'  */

/* fsmc   SRAM  partition info */
static  struct mtd_partition fsmc_sram_partition_info[]  =  {
   PARTITION("External SRAM Bank0", 0x00000000, SZ_1M),
   PARTITION("External SRAM Bank1", 0x01000000, SZ_1M),
   PARTITION("External SRAM Bank2", 0x02000000, SZ_1M),
   PARTITION("External  SRAM  Bank3",  0x03000000,  SZ_1M),
};

/* SRAM  16*/
/* initialize fsmc   related data in fsmc   plat  data */
fsmc_init_sram_board_info(&streamplug1x_fsmc_sram_device,
      fsmc_sram_partition_info,
ARRAY_SIZE(fsmc_sram_partition_info),FSMC_DEVICE_WIDTH16);

/* following is  defined in 'arch/arm/mach-streamplug/fsmc-sram.c'  */
void init fsmc_init_sram_board_info(struct  platform_device *pdev,
      struct mtd_partition *partitions,  unsigned int nr_partitions,
      unsigned int width)
{
   fsmc_sram_set_plat_data(pdev,  partitions,  nr_partitions,  width);
}

/* following is  defined in 'arch/arm/plat-streamplug/include/plat/fsmc.h'
*/
static inline void fsmc_sram_set_plat_data(struct  platform_device *pdev,
      struct mtd_partition *partitions,  unsigned int nr_partitions,
      unsigned int width)
```

```
{
   static struct platdata_mtd_ram *plat_data;
   plat_data =  dev_get_platdata(&pdev->dev);

   if (partitions) {
      plat_data->partitions =  partitions;
      plat_data->nr_partitions =  nr_partitions;
   }

   plat_data->bankwidth =  width;
}
```

SRAM usage

To enable the FSMC-SRAM support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

Please refer to *Section 5.1: Linux MTD framework on page 107* for details on using the SRAM Flash through the MTD interface from both - the kernel and user space.

5.4 Serial memory interface (SMI)

The serial NOR Flash is one of the primary method for booting the system. This section describes the SMI controller driver used to access serial NOR devices.

5.4.1 SMI hardware overview

The NOR Flash is a non-volatile memory which is memory mapped and has a standard serial (SPI) memory interface. The NOR Flash is well suited to be used as code storage because of its reliability, fast read operations, and random access capabilities.

Because the code can be directly executed in the place, the NOR Flash is ideal for storing the firmware, boot code, operating systems, and other data that changes infrequently. The NOR Flash memory has traditionally been used to store relatively small amounts of the executable code for embedded computing devices such as PDAs and cell phones.

The serial NOR Flash on the STreamPlug platform is driven by an SMI (serial memory interface) controller. The serial memory interface (SMI), acting as an AHB slave interface (32-, 16- or 8-bit), manages the clock, data access and status of the NOR Flash memory. The main features of SMI are:

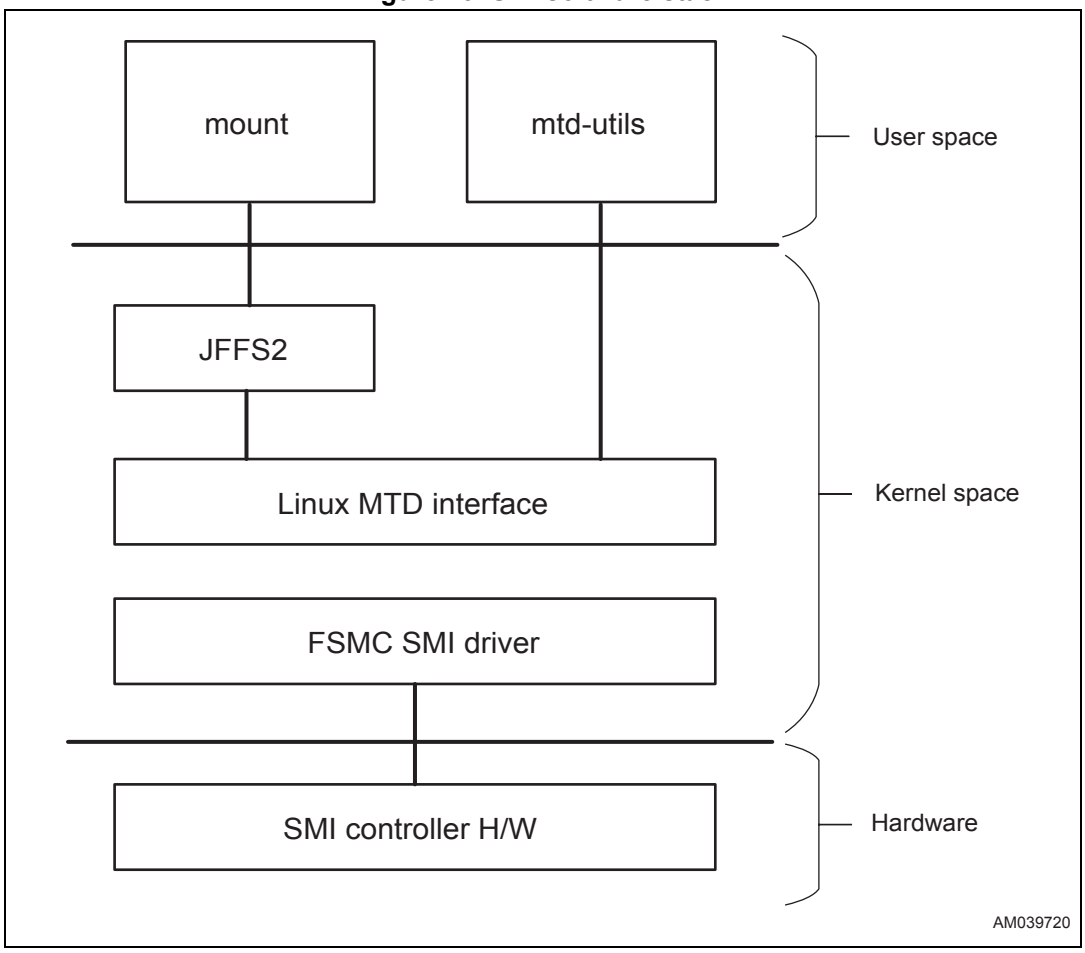
- Supports a group of the SPI-compatible Flash and EEPROM devices.
- The SMI clock signal (SMI_CLK) is generated by the SMI using the clock provided by the AHB bus.

5.4.2 SMI software overview

The SMI serial NOR device driver sits on the top of the SMI controller and provides all necessary functions for a file system via the standard Linux MTD interface.

The user can erase, read and write to the serial NOR devices through the standard MTD interface, as illustrated in *Figure 13*.

Figure 13. SMI software stack



5.4.3 SMI kernel source and configuration

Table 24 lists the details corresponding to the layout of the driver and kernel configuration:

The SMI controller driver is present in "*drivers/mtd/devices/streamplug_smi.c*".

The platform data defining NOR partitioning and controller configuration is present in "*arch/arm/mach-streamplug/streamplug_devel_board.c*".

Table 24. SMI configurations

Configuration	Description
CONFIG_MTD_STREAMPLUG_SMI	Enable SMI controller drivers
CONFIG_MTD_CMDLINE_PARTS	Enable dynamic partitioning based upon kernel command line arguments

5.4.4 SMI platform configuration

This section lists the driver's platform interface and its possible configuration.

SMI driver configuration

The default configuration of the SMI controller depends on the platform data passed from the boards ("*arch/arm/mach-streamplug/streamplug_devel_board.c*").

```
/* serial  nor flash specific board data */

static struct streamplug_smi_flash_info nor_flash_info[]  =
{
   {
   .name  =  "smi0",
   .fast_mode  =  1,
   .mem_base  =  FLASH_MEM_BASE_BANK0,
   .size  =  16  * 1024  * 1024,
   },
   {
   .name  =  "smi1",
   .fast_mode  =  1,
   .mem_base  =  FLASH_MEM_BASE_BANK1,
   .size  =  16  * 1024  * 1024,
   },
   {
   .name  =  "smi2",
   .fast_mode  =  1,
   .mem_base  =  FLASH_MEM_BASE_BANK2,
   .size  =  16  * 1024  * 1024,
   },
};

/* smi  specific board data */
```

```
static struct streamplug_smi_plat_data  smi_plat_data  =
{
   .clk_rate  =  40000000,/* used only in native configuration */
   .num_flashes  =  ARRAY_SIZE(nor_flash_info),
   .board_flash_info  =  nor_flash_info,
};

void     init smi_init_board_info(struct  platform_device *pdev)
{
   smi_set_plat_data(pdev,  &smi_plat_data);
}
```

The above snippet asks the SMI controller driver to configure its clock to 25 MHz to access the serial NOR Flash with partition info embedded in the cmdline.

The timing values apply to Linux native configuration only because in the full configuration the timing setup will be performed by RTOS.

A new NOR device can be added by replacing or extending the “board_flash_info” array supported by the corresponding “num_flashes”.

NOR Flash support

Serial NOR Flash devices accept a different set of commands over the serial interface for read, write and erase. This is enumerated through a structure in “*drivers/mtd/devices/streamplug_smi.c*”:

```
#define FLASH_ID(n,  es,  id,  psize,  ssize,  size)\
{  \
   .name  =  n,\
   .erase_cmd  =  es,\
   .device_id  =  id,\
   .pagesize  =  psize,\
   .sectorsize  =  ssize,  \
   .size_in_bytes  =  size\
}

static struct flash_device flash_devices[]  =  {
/* name - erase cmd - capacity.memorytype.manufacter  - psize - ssize - size
*/
   FLASH_ID("winbond  w25q128",  0xd8,  0x001840EF,  0x100,  0x10000,
0x1000000),
   FLASH_ID("winbond  w25q64",  0xd8,  0x001740EF,  0x100,  0x10000,
0x800000),
   FLASH_ID("st  m25p16", 0xd8,  0x00152020,  0x100,  0x10000,  0x200000),
   FLASH_ID("st  m25p32", 0xd8,  0x00162020,  0x100,  0x10000,  0x400000),
   FLASH_ID("st  m25p64", 0xd8,  0x00172020,  0x100,  0x10000,  0x800000),
   FLASH_ID("st  m25p128", 0xd8,  0x00182020,  0x100,  0x40000,
0x1000000),
   FLASH_ID("st  m25p05", 0xd8,  0x00102020,  0x80  , 0x8000   , 0x10000),
   FLASH_ID("st  m25p10", 0xd8,  0x00112020,  0x80  , 0x8000   , 0x20000),
```

```
   FLASH_ID("st  m25p20", 0xd8,  0x00122020,  0x100,  0x10000,  0x40000),
   FLASH_ID("st  m25p40", 0xd8,  0x00132020,  0x100,  0x10000,  0x80000),
   FLASH_ID("st  m25p80", 0xd8,  0x00142020,  0x100,  0x10000,  0x100000),
   FLASH_ID("st  m45pe10", 0xd8,  0x00114020,  0x100,  0x10000,  0x20000),
   FLASH_ID("st  m45pe20", 0xd8,  0x00124020,  0x100,  0x10000,  0x40000),
   FLASH_ID("st  m45pe40", 0xd8,  0x00134020,  0x100,  0x10000,  0x80000),
   FLASH_ID("st  m45pe80", 0xd8,  0x00144020,  0x100,  0x10000,  0x100000),
   FLASH_ID("sp  s25fl004", 0xd8,  0x00120201,  0x100,  0x10000,  0x80000),
   FLASH_ID("sp  s25fl008", 0xd8,  0x00130201,  0x100,  0x10000,
0x100000),
   FLASH_ID("sp  s25fl016", 0xd8,  0x00140201,  0x100,  0x10000,
0x200000),
   FLASH_ID("sp  s25fl032", 0xd8,  0x00150201,  0x100,  0x10000,
0x400000),
   FLASH_ID("sp  s25fl064", 0xd8,  0x00160201,  0x100,  0x10000,
0x800000),
   FLASH_ID("atmel  25f512", 0x52,  0x0065001F,  0x80  , 0x8000   ,
0x10000),
   FLASH_ID("atmel  25f1024"  , 0x52,  0x0060001F,  0x100,  0x8000   ,
0x20000),
   FLASH_ID("atmel  25f2048"  , 0x52,  0x0063001F,  0x100,  0x10000,
0x40000),
   FLASH_ID("atmel  25f4096"  , 0x52,  0x0064001F,  0x100,  0x10000,
0x80000),
   FLASH_ID("atmel  25fs040"  , 0xd7,  0x0004661F,  0x100,  0x10000,
0x80000),
   FLASH_ID("mac  25l512", 0xd8,  0x001020C2,  0x010,  0x10000,  0x10000),
   FLASH_ID("mac  25l1005", 0xd8,  0x001120C2,  0x010,  0x10000,  0x20000),
   FLASH_ID("mac  25l2005", 0xd8,  0x001220C2,  0x010,  0x10000,  0x40000),
   FLASH_ID("mac  25l4005", 0xd8,  0x001320C2,  0x010,  0x10000,  0x80000),
   FLASH_ID("mac  25l4005a", 0xd8,  0x001320C2,  0x010,  0x10000,
0x80000),
   FLASH_ID("mac  25l8005", 0xd8,  0x001420C2,  0x010,  0x10000,
0x100000),
   FLASH_ID("mac  25l1605", 0xd8,  0x001520C2,  0x100,  0x10000,
0x200000),
   FLASH_ID("mac  25l1605a", 0xd8,  0x001520C2,  0x010,  0x10000,
0x200000),
   FLASH_ID("mac  25l3205", 0xd8,  0x001620C2,  0x100,  0x10000,
0x400000),
   FLASH_ID("mac  25l3205a", 0xd8,  0x001620C2,  0x100,  0x10000,
0x400000),
   FLASH_ID("mac  25l6405", 0xd8,  0x001720C2,  0x100,  0x10000,
0x800000),
};
```

6 Accelerators

The STreamPlug includes some hardware accelerators to speedup the complex algorithm elaboration and data management. In particular there are:

- JPEG encoder/decoder
- DMA engine
- Cryptographic coprocessor (C3).

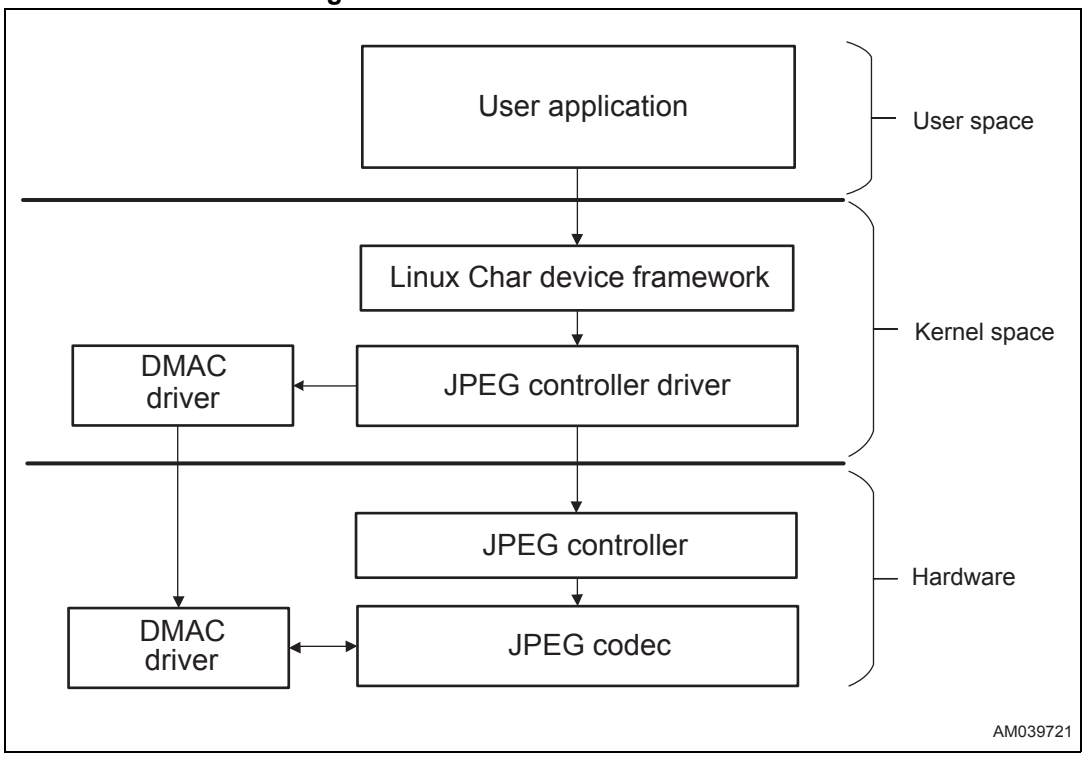
6.1 JPEG encoder/decoder

This section describes the JPEG encoder/decoder driver.

6.1.1 JPEG encoder/decoder software overview

The JPEG controller driver supports both JPEG encoding and decoding with/without header processing enabled. It acts as an interface between user level applications and the JPEG codec. The JPEG driver provides a char device interface to the user application and can be used from the user level only. The JPEG driver accepts (for encoding) and gives (for decoding) data in the MCU format. The overall JPEG codec software system architecture is shown in *Figure 14*.

Figure 14. JPEG software architecture



The JPEG driver exposes two device nodes to the user application: “jpegread” and “jpegwrite”. Input data may be written to the “jpegwrite” node and output data may be read from the “jpegread” node. Data may be written/read to or from the JPEG chunk by chunk. This means that input and output data buffers do not need to be very large. Small buffers

can be used again and again to write/read data to or from the JPEG. The following sections describe usage of the JPEG driver in detail. Note that the sections are in the sequence in which the JPEG driver is required to be programmed.

6.1.2 JPEG encoder/decoder kernel source and configuration

Table 25 lists the Linux kernel options related to the JPEG device driver.

Table 25. JPEG driver configuration options

Configuration option	Comment
CONFIG_DESIGNWARE_JPEG	This option enables the STreamPlug JPEG driver.
CONFIG_DW_DMAC	This option must be selected for JPEG operations. This will enable the DMA driver.

The STreamPlug JPEG device driver is composed by the following source code files:

- “drivers/char/designware_jpeg.c”
- “drivers/char/designware_jpeg.h”
- “arch/arm/plat-streamplug/jpeg.c”

6.1.3 JPEG encoder/decoder platform configuration

Configuration of the device/driver through platform data or inherently in the driver itself.

6.1.4 JPEG encoder/decoder usage

JPEG encoder/decoder kernel space

To access the JPEG driver specific data types in user applications, include “<linux/spr_jpeg_syn_usr.h>”. The JPEG device is allocated the major number dynamically. To obtain the major number of the JPEG device, run the following command after board boot-up:

```
$  cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
  9 st
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
```

```
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
161 ircomm
162 raw
180 usb
189 usb_device
204 ttyAMA
247 ubi0
248 bufV
249 ttyVS
250 ttyV
251 imageval
252 vlog
253 jpeg-designware
254 rtc

Block devices:
  1 ramdisk
259 blkext
  8 sd
 11 sr
 31 mtdblock
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
254 vblock
```

After obtaining the major number of the JPEG device, create JPEG nodes using the following commands:

```
$ mknod /dev/jpegread c major 0
$ mknod /dev/jpegwrite c major 1
```

In this example, the “major” is the major number allocated to the JPEG.

These user level nodes are used for any further interaction with the JPEG driver. The following steps illustrate how to do encoding/decoding with the JPEG.

Open JPEG nodes

After creating JPEG read and write nodes, the application should open them. Use the following system call to open JPEG nodes:

```
rfd = open("/dev/jpegread", O_RDWR | O_SYNC);
wfd = open("/dev/jpegwrite", O_RDWR | O_SYNC);
```

where:

- “O_RDWR” access permission is used to get read/write permissions.
- “O_SYNC” access permission is used for synchronous I/O. Any writes on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware. mmap function used later will give uncached memory if this flag is used, otherwise data consistency issues will occur.
- “rfd” and “wfd” are read and write file descriptors used to further communicate with JPEG nodes.

JPEG nodes can be opened by only one application at a time, thus they cannot be shared. On success, positive file descriptors are returned, otherwise on error, -1 is returned and errno is set appropriately. These file descriptors can be used for any further communication with the JPEG device.

Set source image size

The JPEG driver must be told in advance the size of the input data (required by hardware). Use the following system call to set the input data size:

```
ioctl(wfd, JPEGIOC_SET_SRC_IMG_SIZE, size);
```

where:

- “wfd” is the file descriptor of the jpegwrite node
- “JPEGIOC_SET_SRC_IMG_SIZE” is the ioctl command for setting input data size
- “size” is the size of the source image in bytes

Upon success, zero is returned, otherwise on error, -1 is returned and errno is set appropriately. After completion of processing, if the user needs to encode/decode another image, the user doesn't have to close JPEG nodes, de-allocate the buffer memory and open the nodes again. The user can simply call this function again with the size of new input data. This resets the complete JPEG system (software and hardware).

Set JPEG information

The JPEG can perform four types of operations. They are:

- Encoding with header processing (EWH): the output JPEG image will have a header as a part of the image.
- Encoding without header processing (EWOH): the output JPEG image will not have a header as a part of the image.
- Decoding with header processing (DWH): the input JPEG image will have a header as a part of the image.
- Decoding without header processing (DWOH): the input JPEG image will not have a header as a part of the image.

The JPEG header and compression table information is passed to the JPEG driver in all above cases, except DWH. In the DWH the JPEG codec extracts a header and table information from the input JPEG image. Before proceeding with encoding/decoding, provide header and table information to the JPEG driver (not required for DWH). Use the following system call to set JPEG info for the EWH and EWOH:

```
ioctl(wfd,  JPEGIOC_SET_ENC_INFO,  &enc_info);
```

where:

- “wfd” is the file descriptor of the jpegwrite node.
- “JPEGIOC_SET_ENC_INFO” is the ioctl command for setting JPEG encoding information.
- “enc_info” is the structure containing jpeg encoding information.

The “enc_info” structure is described below.

```
struct jpeg_enc_info {
   struct  jpeg_hdr hdr;/*  jpeg image   header */
   int hdr_enable;  /* header processing  enable/disable  */
   char qnt_mem[QNT_MEM_SIZE];  /* quantization memory  */
   char dht_mem[DHT_MEM_SIZE];  /* DHT memory  */
   char henc_mem[HENC_MEM_SIZE];  /* Huff enc  memory  */
};
```

The “jpeg_hdr” structure is described below.

```
struct  jpeg_hdr {
   u32  num_clr_cmp;  /* number  of color components minus   1. */
   u32  clr_spc_type;  /* number  of quantization tables in the output
stream.  */
   u32  num_cmp_for_scan_hdr;  /* number  of  components for scan   header
marker  segment
   minus   1.*/
   u32  rst_mark_en;/*  restart marker enable/disable  */
   u32  xsize;  /* number  of pixels per line  */
   u32  ysize;  /* number  of lines.  */
   u32  mcu_num;  /* this value defines the number  of minimum  coded
units to be  coded,
   minus   1  */
   u32  mcu_num_in_rst;  /* number  of mcu's  between two  restart markers
minus   1.*/
```

```
   struct mcu_composition mcu_comp[MAX_MCU_COMP];  /* represents MCU
composition */
};
```

The “mcu_composition” structure is described below.

```
struct mcu_composition {
   u32  hdc;   /  *hdc  bit  selects the Huffman table  for  the encoding of
the DC
                  * coefficient in the data units  belonging to the color
component */
   u32  hac;  /* hac  bit  selects the Huffman table  for  the encoding of
the AC
                * coefficients in the data units  belonging to the color
component */
   u32  qt;   /* QT indicates the quantization table to be  used  for  the
color
                *component */
   u32  nblock; /* nblock value is  the number  of data units (8 x  8
blocks of data)  of
                  * the color component contained in  the MCU  */
   u32  h; /* Horizontal Sampling factor for component */
   u32  v; /* Vertical Sampling factor for component */
};
```

Use the following system call to set JPEG info for DWOH:

```
ioctl(wfd,  JPEGIOC_SET_DEC_INFO,  &dec_info);
```

where:

- “wfd” is the file descriptor of the jpegwrite node.
- “JPEGIOC_SET_DEC_INFO” is the ioctl command for setting JPEG decoding information.
- “dec_info” is the structure containing jpeg decoding information.

The “dec_info” structure is described below.

```
struct jpeg_dec_info {
   struct  jpeg_hdr hdr;  /* jpeg image   header */
   int hdr_enable;/* header processing  enable/disable  */
   char qnt_mem[QNT_MEM_SIZE];/* quantization memory  */
   char hmin_mem[HMIN_MEM_SIZE];/* Huff min  memory  */
   char hbase_mem[HBASE_MEM_SIZE];  /* Huff base  memory  */
   char hsymb_mem[HSYMB_MEM_SIZE];  /* Huff symb  memory  */
};
```

Upon success, a zero is returned, otherwise on error, -1 is returned and errno is set appropriately. If this ioctl is not called before writing/reading data to/from the JPEG, then DWH (decoding with header processing) is performed by default.

Mapping memory for read and write

The JPEG driver needs to allocate buffers for storing the input and output data. To speed up the encoding/decoding process, the driver uses the “mmap()” Linux system call. This system call allocates physically contiguous memory for the JPEG driver and returns the virtual address of this memory to the user application. The user can then read and write to the virtual addresses and the same data is reflected in the driver buffers. This saves unnecessary data copy time between the kernel and user level. In order to further increase the performance of the encoding/decoding process, two buffers are used both for read and write operations. By having two buffers for read and write, we are actually parallelizing JPEG processing. By the time JPEG hardware reads/writes data from/to read/write buffer software has written/read data to/from an other buffer. Use the following system call to the map physical memory in virtual space:

```
void * mmap(void  *start,  size_t length,  int prot , int flags,  int fd,
off_t offset);
```

where:

- “fd” is the file descriptor of the jpeg read/jpeg write node.
- “length” is the total size of buffers (write or read) to allocate. The size should be multiple of the page size, i.e.: 4096 bytes. The maximum size that can be allocated or mapped at once is 4 Mbytes, this makes the size of each buffer (write or read) 2 Mbytes. A single call to “mmap” for the “jpegread/jpegwrite” node will allocate “length” amount of the memory and will return its base address. The user application should use this memory as two buffers of the same size.

Upon success, “mmap” returns a pointer to the mapped area. On error, the value “MAP_FAILED” [that is, (void *)], -1 is returned, and errno is set appropriately. The “mmap” function asks to map length bytes starting at offset “offset” from the file specified by the file descriptor “fd” into the memory, preferably at the address “start”. This “start” address is a hint only, and is usually specified as 0. The following parameters can be used to call this function for “rfd” and “wfd”.

```
mmap(0,  size,  PROT_READ  | PROT_WRITE,  MAP_SHARED,  file_desc,  0)
```

After one codec operation (encoding/decoding) has been completed, additional encode/decode operations may be performed without closing nodes of jpegread/jpegwrite or having to allocate/map the memory again. However, if the user needs to map the memory again for a different size, any previously mapped memory must first be unmapped.

Write source JPEG data

After mapping the memory for the write buffer, input data may be written to it (data length less than equal to the size of one write buffer). This can be done by writing or copying data directly to the mapped virtual memory addresses of write buffers.

Start encoding or decoding

Once input data is written to the write buffer, encoding/decoding can be started/resumed. Use the following system call to start and resume JPEG encoding/decoding with new input data:

```
ioctl(wfd,  JPEGIOC_START,  size);
```

where:

- “wfd” is the file descriptor of the jpegwrite node
- “JPEGIOC_START” is the command to start/resume jpeg encoding/decoding
- “size” is the amount of data written on the jpegwrite node, the memory mapped for each jpeg write buffer.

This is a blocking system call which unblocks or returns only when encoding/decoding with data supplied from the current write buffer is started or an error has occurred. On success, zero is returned otherwise, -1 is returned on error and errno is set appropriately. For example, if the total size of the input data is 15 Kbytes, and the size of each write buffer is 4 Kbytes, then the following steps are used to pass data to the JPEG:

1. Set the current write buffer to “write_buffer0”
2. Write 4 Kbytes of data into the current write buffer
3. Call “JPEGIOC_START ioctl” with the size equal to 4 Kbytes
4. If the current write buffer is buffer 0, toggle the write buffer to be buffer1. If it's set it to the buffer1, toggle it to the buffer0.
5. Follow steps *2.*, *3.* and *4.* two more times with a size of 4 Kbytes and one time with a size of 3 Kbytes.

Get encoded/decoded data

Once encoding/decoding is started, the output data must be copied from the read buffer. Use the following system call used to get the output data:

```
ioctl(rfd,  JPEGIOC_GET_OUT_DATA_SIZE,  &size);
```

where:

- “rfd” is the read node file descriptor.
- “JPEGIOC_GET_OUT_DATA_SIZE” is the command to get output data.
- “size” is the variable which will store the size of output data in bytes. This must always be equal to the size of the individual read buffer. If it is less than the size of the individual read buffer, end of encoding/decoding is indicated.

This is a blocking system call which unblocks or returns only when the output data size less than equal to the size of the individual read buffer is written to the current read buffer after encoding/decoding. On success, zero is returned otherwise, -1 is returned on error and errno is set appropriately. For example: the total size of output data is 15 Kbytes, the size of each write buffer is 4 Kbytes, then the user needs to do following steps to read data from the JPEG:

1. Set the current read buffer to the “read_buffer0”
2. Call “JPEGIOC_GET_OUT_DATA_SIZE ioctl”
3. Check the value of the parameter size if it is less than 4 kBytes/Kbits, wait until the end encoding/decoding is indicated
4. Read data from the current read buffer
5. If the current read buffer is the buffer0, toggle it the buffer1. If the read buffer is the buffer1, toggle it to the buffer0.
6. Follow steps *2.*, *3.*, *4.*, and *5.* until the time step *3.* doesn't indicate the end of encoding/decoding.

Writing input data to the write buffer and reading output data from the read buffer have to be done simultaneously. It is recommended to use different threads/processes for reading and writing. This will speed up the encoding/decoding process.

Get JPEG information

Once encoding/decoding is finished the user can read JPEG information (JPEG header information and compression tables). Use the following system call to get JPEG information:

```
ioctl(rfd,  JPEGIOC_GET_INFO,  &jpeg_info);
```

where:

- “rfd” is the read node file descriptor
- “JPEGIOC_GET_INFO” is the command to get jpeg information
- “jepg_info” is the instance of struct jpeg_info. After successful completion of this system call, it will contain information of the JPEG header and compression tables.

On success, zero is returned otherwise, -1 is returned on error and errno is set appropriately. Struct. “jpeg_hdr_info” is defined below:

```
     struct jpeg_info {
   struct jpeg_hdr hdr;/* jpeg image   header */
   char qnt_mem[QNT_MEM_SIZE]; /* quantization memory */
   char hmin_mem[HMIN_MEM_SIZE]; /* Huff min memory */
   char hbase_mem[HBASE_MEM_SIZE]; /* Huff base memory */
   char hsymb_mem[HSYMB_MEM_SIZE]; /* Huff symb memory */
   char dht_mem[DHT_MEM_SIZE]; /* DHT memory */
   char henc_mem[HENC_MEM_SIZE]; /* Huff enc memory */
};
```

The struct “jpeg_hdr” is already defined earlier in *Section : Set JPEG information on page 131*.

This ioctl is mainly useful for DWH (decoding with header processing enabled).

munmap

Once the application is finished with JPEG processing, it should unmap the memory that has been mapped for read and write buffers. Use the following system call to unmap memory:

```
munmap(adrs,  size);
```

where:

- “adrs” is the address of the mapped memory
- “size” is the size of the mapped memory

On success, a zero is returned otherwise, -1 is returned on error and errno is set appropriately.

Close

After unmapping the memory, JPEG nodes must be closed. Use the following system call to do this:

```
close(fd);
```

Close returns zero on success, or -1 if error occurred, errno is set appropriately. This function must be called both for read and write nodes.

JPEG codec usage

JPEG read and write are not synchronized enough, for example, one chunk of input data may produce output data varying in size. Due to this, write and read should be simultaneously made to the JPEG driver, otherwise the JPEG codec may be wasting time sitting idle. It is recommended to use two processes or threads to read and write data simultaneously from the JPEG driver. This will increase speed of JPEG processing. The following example is for encoding/decoding a JPEG image.

Below, input data is read by application from a file and is passed to the JPEG driver. After that, it is processed by the JPEG codec based no processing type (encoding/decoding), and output data is read by application again.

```
#include <sys/mman.h>
#include "include/linux/spr_jpeg_syn_usr.h"

struct jpeg_info jpeg_info;
volatile unsigned char *wbuf[2] = {NULL,  NULL},
*rbuf[2] =  {NULL,  NULL};
unsigned int wsize =  4*4096,  rsize =  4*4096;
unsigned int ssize =  XXX;/*  set size of input  data here */
int rfd,  wfd,  cur_rbuf =  1,  cur_wbuf =  1;
pid_t pid;
void jpegread()
{
   int size =  0, status;

   /* while encoding/decoding  is  not over */
   do
   {
      shuffle_buf(cur_rbuf);
```

```
        if((status = ioctl(rfd, JPEGIOC_GET_OUT_DATA_SIZE, &size)) != 0)
            return -1;

    /* Add code here for manipulating decoded data present in
rbuf[cur_rbuf] */
    }while(size == rsize);

    /* get jpeg info after encoding/decoding is over */
    ioctl(rfd, JPEGIOC_GET_INFO, &jpeg_info);

    /* unmap buf */
    munmap((char *)rbuf[0], 2*rsize);
    close(rfd);
}

void jpegwrite()
{
    uint size = 0, count=0;
    int wfd, status;

    /* open jpeg nodes */
    wfd = open("/dev/jpegwrite", O_RDWR|O_SYNC);
    if (wfd == -1)
        return -1;

    /* set src image size */
    ioctl(wfd, JPEGIOC_SET_SRC_IMG_SIZE, ssize);

    /* set jpeg info for DWOH */
    /*
    struct jpeg_dec_info dec_info;
    ioctl(wfd, JPEGIOC_SET_DEC_INFO, dec_info);
    */

    /* set jpeg info for EWH, EWOH */
    /*
    struct jpeg_enc_info enc_info;
    ioctl(wfd, JPEGIOC_SET_ENC_INFO, enc_info);
    */

   wbuf[0] = (unsigned char *)mmap(0, 2*wsize, PROT_READ | PROT_WRITE,
MAP_SHARED, wfd, 0);
    wbuf[1] = wbuf[0] + wsize;

    while(count < ssize)
```

```
    {
       size =  (ssize-count)  <  wsize?(ssize-count):wsize;
       count +=  size;

       shuffle_buf(cur_wbuf);
       /* Add  code  here to copy   size amount  of data on  wbuf[cur_wbuf]
*/

       if((status  =  ioctl(wfd,  JPEGIOC_START,  rd)) != 0)
          return -1;
    }
    munmap((char  *)wbuf[0],  2*wsize);
    close(wfd);
}

int main(void)
{
    /* open  jpeg nodes   */
    rfd =  open("/dev/jpegread",O_RDWR|O_SYNC);
    if (rfd ==  -1)
       return -1;

   rbuf[0] =  (unsigned  char *)mmap(0,  2*rsize,  PROT_READ  | PROT_WRITE,
MAP_SHARED, rfd,  0);
    if (rbuf[0] ==  NULL)
       return -1;
    rbuf[1] =  rbuf[0] +  rsize;

    pid =  fork();
    if (pid  ==  0)// child
    {
       jpegwrite();
       exit(0);
    }
    else if (pid  >  0)// parent
    {
       jpegread();
       wait(0);
    }
    else// failed to fork
    {
       printf("Can't  create child\n");
       exit(1);
    }
}
```

JPEG encoder/decoder user space

Both encoding and decoding can be tested with ready-to-use demonstration applications provided under the folder “/examples/jpeg”. Proof of the usage of the JPEG device driver can be obtained printing the number of raised IRQs with the following command:

```
$ cat /proc/interrupts
           CPU0
  5:          1        vIRQ  ehci_hcd:usb1, ohci_hcd:usb2
 16:          0        vIRQ  dw_dmac
 17:      12367        vIRQ  smi
 18:          0        vIRQ  rtc-streamplug
 24:          0        vIRQ  jpeg-designware
 26:         57        vIRQ  uart-pl011
105:       4906        vIRQ  microvisor timer
106:          0        vIRQ  okl4-ksp-agent
Err:          0
```

Decoder

In order to test the JPEG decoder device driver, an application is provided below the “/examples/jpeg” folder for decoding. Go into the “/example/jpeg” folder and run the following application in the background:

```
$  ./decode  <file  in>  <file  out>
```

An input file example (“lena.jpg”) is on the same folder. The *.mcu output file has to be passed as an input parameter to the Synopsys JDEM utility in order to generate the decoded image.

Encoder

In order to test the JPEG encoder device driver, an application is provided below the “/examples/jpeg” subfolder for encoding. Go into the “/example/jpeg” folder and run the application such as:

```
$  ./encode  <Q  table>  <Huffman  table>  <DHT table> <X size>  <Y size>
<MCU  format>
<restar marker distance>  <file  in>  <file  out>
```

For example:

```
$  ./encode  qetable.dat  htable.dat  dhttable.dat  128  128  1  1  lena.mcu
out.jpeg
```

6.2 Direct memory access (DMA)

All STreamPlug MPUs are equipped with a general purpose DMA controllers which provide several DMA channels that can be used to off load the CPU from some of the memory copying tasks. This section describes the details of the DMAC driver.

6.2.1 DMA hardware overview

Direct memory access (DMA) allows certain subsystems within the STreamPlug MPU to access the system memory for reading and/or writing independently of the CPU and to transfer data avoiding, in this way, an heavy CPU overhead.

The STreamPlug MPU uses the Synopsys designware DMA controller. It is connected to the AHB bus. Synopsys designware DMAC's main features are:

- AMBA 2.0-compliant
- DMA transfers
  - Peripheral-to-peripheral
  - Memory-to-peripheral
  - Peripheral-to-memory
  - Memory-to-memory
- Channels
  - Up to eight channels, one per source and destination pair
  - Unidirectional channels - data transfers in one direction only
- Interrupt generation on DMA transfer (multiblock) completion, block transfer completion, single and burst transaction completion.

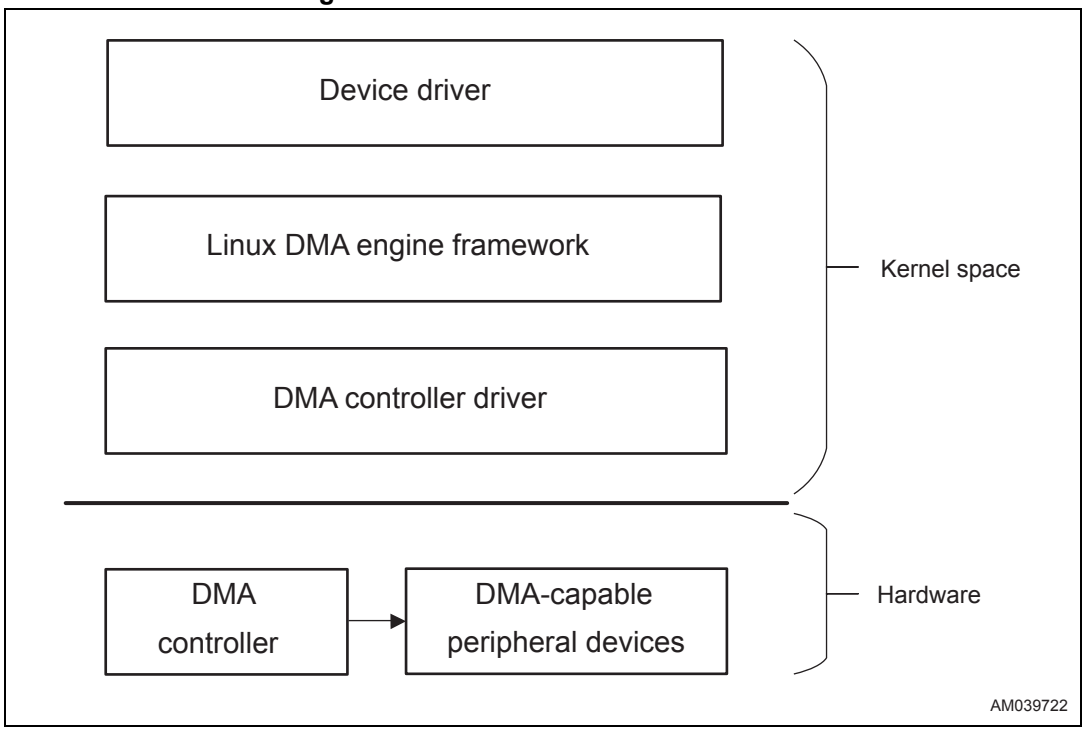
6.2.2 DMA software overview

The Linux DMA support has been organized in two different layers in order to provide abstraction to the user which can hide the internal implementation of DMA controller drivers.

The Linux DMA engine framework defines clear APIs and channel abstraction for the user to access underlying DMA hardware and it expects the underlying DMA controller driver to provide necessary callbacks to support this.

The overall DMA software system architecture is represented in *Figure 15*.

Figure 15. DMA framework architecture



The DMA framework present in Linux provides a simple interface to client drivers who wish to use the DMA. Clients just request DMA channels, transfer data on allocated DMA channels and finally free allocated DMA channels.

To increase the performance of the DMA driver, cache/memory consistency related issues are handled by client drivers for non-slave transfer, i.e.: transfers involving peripherals. They can synchronize data between the cache and DDR if the source or destination memory is cached. If memories are uncached there is no need for synchronization.

DMA engine API

The entire DMA engine abstraction is built around the concept of DMA channels which is abstracted as:

```
/**
* struct dma_chan  - devices supply DMA  channels,  clients use  them
* @device:  ptr to the dma device who supplies this  channel,  always !%NULL
* @cookie:  last  cookie value returned to client
* @chan_id:  channel ID  for sysfs
* @dev:  class device for  sysfs
* @device_node:  used  to add  this  to the device chan  list
* @local:  per-cpu  pointer to a  struct dma_chan_percpu
* @client-count:  how many clients  are using this  channel
* @table_count:  number  of  appearances in the mem-to-mem  allocation table
* @private:  private data for  certain client-channel  associations
*/
struct dma_chan  {
   struct dma_device *device;
   dma_cookie_t cookie;

   /* sysfs */
   int chan_id;
   struct dma_chan_dev   *dev;

   struct list_head device_node;
   struct dma_chan_percpu     percpu  *local;
   int client_count;
   int table_count;
   void *private;
};
```

The following example tries to list some of the common and the most used DMA engine framework APIs:

```
struct dma_chan  *dma_request_channel(dma_cap_mask_t  mask,  dma_filter_fn
filter_fn,  void *filter)
```

In order to do DMA transfers, the user must request a DMA channel. To request a channel “dma_request_channel()”, the API is used. A channel allocated via this interface is reserved to the caller, until the “dma_release_channel()” is called.

The “dma_filter_fn” parameter is defined as “typedef” bool (*dma_filter_fn)(struct dma_chan *chan, void *filter_param).

The “filter_fn” parameter is optional, but highly recommended for slave and cyclic channels as they typically need to obtain a specific DMA channel. When the optional “filter_fn” parameter is NULL, the “dma_request_channel()” simply returns the first channel that satisfies the capability mask. Otherwise, the “filter_fn” routine will be called once for each free channel which has a capability in the mask and return true when the desired DMA channel is found.

The following is only required for slave (involving peripherals) transfers.

```
int dmaengine_slave_config(struct dma_chan *chan, struct
dma_slave_config *config)
```

Most of the generic information which a slave DMA can use is in “struct dma_slave_config”. This allows the clients to specify the DMA direction, DMA addresses, bus widths, DMA burst lengths, etc. for the peripheral. If some DMA controllers have more parameters to be sent then they should try to include “struct dma_slave_config” in their controller specific structure. That gives flexibility to the client to pass more parameters, if required. “dma_slave_config” is defined as:

```
struct dma_slave_config {
   enum dma_data_direction direction;
   dma_addr_t src_addr;
   dma_addr_t dst_addr;
   enum dma_slave_buswidth src_addr_width;
   enum dma_slave_buswidth dst_addr_width;
   u32  src_maxburst;
   u32  dst_maxburst;
   bool device_fc;
};
```

The “dma_async_tx_descriptor” API can be used for DMA from the source to the destination memory and is used for non-slave usage.

```
struct dma_async_tx_descriptor *(*device_prep_dma_memcpy)(
   struct dma_chan *chan, dma_addr_t dest, dma_addr_t src,
   size_t len, unsigned long flags);
```

Similarly the “dma_async_tx_descriptor” API sets destination memory with a specific value.

```
struct dma_async_tx_descriptor *(*device_prep_dma_memset)(
   struct dma_chan *chan, dma_addr_t dest, int value, size_t len,
   unsigned long flags)
struct dma_async_tx_descriptor *(*device_prep_slave_sg)(
   struct dma_chan *chan, struct scatterlist *sgl,
   unsigned int sg_len, enum dma_data_direction direction,
   unsigned long flags);
```

For “slave_sg” usage one can prepare the DMA descriptor for a transfer through this API. The DMA on calling of this API prepares a list of scatter gather buffers for transfer from/to a peripheral.

6.2.3 DMA kernel source and configuration

The DMA controller device driver is composed by the Synopsys DMAC files: “drivers/dma/dw_dmac.c”, “drivers/dma/dw_dmac_regs.h” and “include/linux/dw_dmac.h”.

Table 26 lists the kernel configuration options related to STreamPlug DMA support.

Table 26. DMA configurations

Configuration	Description
CONFIG_DMADEVICES	Enable DMA ENGINE devices support
CONFIG_DW_DMAC	Enable SYNOPSYS designware DMA controller driver

6.2.4 DMA platform configuration

The optional platform data passed from machines for DMAC is as follows:

```
/* dmac device registration  */
static struct dw_dma_platform_data dmac_platform_data  =  {
   .nr_channels  =  8,
};

static struct resource dmac_resources[]  =  {
{
   .start  =  STREAMPLUG1X_ICM3_DMA_BASE,
   .end  =  STREAMPLUG1X_ICM3_DMA_BASE  +  SZ_4K - 1,
   .flags  =  IORESOURCE_MEM,
   }, {
   .start  =  STREAMPLUG1X_IRQ_BAS_SUBS_DMAC,
   .flags  =  IORESOURCE_IRQ,
   },
};

struct  platform_device streamplug1x_dmac_device =  {
   .name  =  "dw_dmac",
   .id  =  -1,
   .dev  =  {
      .coherent_dma_mask  =  ~0,
      .platform_data  =  &dmac_platform_data,
   },
   .num_resources  =  ARRAY_SIZE(dmac_resources),
   .resource  =  dmac_resources,
};
```

6.2.5 DMA usage

Below is a guide to device driver developers on how to use the DMA API of the DMA engine. The DMA usage consists of following steps:

1. Allocate a DMA slave channel
2. Set slave and controller specific parameters
3. Get a descriptor for transaction
4. Submit the transaction
5. Issue pending requests and wait for callback notification

Following are examples usage of the DMA engine for “memcpy”.

Common callback routine:

```
static  void dmatest_callback(void  *arg)  {
   int *done  =  arg;
   *done  =  1;
   printk(KERN_INFO  "DMA  xfer is complete");
}
```

Memcpy

```
void dma_memcpy()  {
   dma_cap_mask_t  mask;

   dma_cookie_t cookie;
   struct  dma_chan  *chan;
   struct dma_device *dmadev;
   struct dma_async_tx_descriptor *tx =  NULL;
   u8 *sbuf = 0xc0000000, *dbuf = 0xc8000000;
   dma_addr_t dma_srcs,  dma_dsts;
   int len =  0x1000,  done  =  0;
   enum dma_ctrl_flags flags  =  DMA_CTRL_ACK  | DMA_PREP_INTERRUPT  |
   DMA_COMPL_SKIP_DEST_UNMAP  | DMA_COMPL_SRC_UNMAP_SINGLE;

   dma_cap_zero(mask);
   dma_cap_set(DMA_MEMCPY,  mask);  chan  =  dma_request_channel(mask,
NULL,  NULL);

   dmadev  =  chan->device;

   dma_srcs =  dma_map_single(dmadev->dev,  sbuf,  len,  DMA_TO_DEVICE);
   dma_dsts =  dma_map_single(dmadev->dev,  dbuf,  len,  DMA_BIDIRECTIONAL);

   tx =  dmaengine_prep_memcpy(chan,  dma_dsts,  dma_srcs,  len,  flags);
   if (!tx) {
      dma_unmap_single(dmadev->dev,  dma_srcs,  len,  DMA_TO_DEVICE);
      dma_unmap_single(dmadev->dev,  dma_dsts,  len,  DMA_BIDIRECTIONAL);
   }
```

```
    tx->callback =  dmatest_callback;
    tx->callback_param =  &done;
    cookie =  tx->tx_submit(tx);
    if (dma_submit_error(cookie))  {
       printk(KERN_INFO  "Error in dma tx_submit\n");
       return;
    }

    dma_async_issue_pending(chan);
    while (!done)
       msleep(10);

    dma_unmap_single(dmadev->dev,  dma_dsts,  len,  DMA_BIDIRECTIONAL);
}
```

For the Synopsys DMA controller, this must be an instance of “struct dw_dma_slave”.

```
/**
* struct dw_dma_slave  - Controller-specific  information  about a  slave
*
* @dma_dev:  required DMA  controller  device
* @tx_reg: physical address of data register used for
*      memory-to-peripheral transfers
* @rx_reg: physical address of data register used for
*      peripheral-to-memory transfers
* @reg_width: peripheral register width
* @cfg_hi:  Platform-specific  initializer  for the CFG_HI register
* @cfg_lo:  Platform-specific  initializer  for the CFG_LO  register
* @src_master:  src master for  transfers  on  allocated channel.
* @dst_master:  dest master for  transfers  on  allocated channel.
* @src_msize: src burst size.
* @dst_msize: dest burst size.
* @fc: flow controller for DMA transfer
*/
```

```
Struct dw_dma_slave  {
   structdevice*dma_dev;
   dma_addr_ttx_reg;
   dma_addr_trx_reg;
   enum dw_dma_slave_widthreg_width;
   u32 cfg_hi;
   u32 cfg_lo;
   u8 src_master;
   u8 dst_master;
   u8src_msize;
   u8dst_msize;
   u8fc;
};
```

6.3 Channel controller coprocessor (C3)

The channel controller coprocessor (C3) is an hardware cryptographic coprocessor used to accelerate data intensive applications where computationally expensive algorithms must operate on medium to large memory buffers. Such applications can be found in the fields of security (data encryption, integrity check, etc.) and networking.

The most important features of the C3 are:

- Supported many cryptographic algorithms (AES, DES, TripleDES, SHA-1, SHA-256, MD5, etc.)
- Instruction driven by the DMA based programmable engine.

The C3 is a highly programmable DMA based hardware coprocessor that executes some instructions flows (programs) written in the memory by the host processor. These programs specify which operations must be performed and where to locate data buffers (input, output, parameters) in the memory. After being setup the C3 is completely autonomous and can perform an unlimited number of operations, until it hits an end of program instruction in which case it can signal the end of processing by the means of an interrupt request (if programmed to do so).

6.3.1 C3 software overview

The C3 device driver is composed by an API which allows to setup programs that the C3 processor can execute. The API is exposed through a set of Linux kernel symbols listed hereafter:

```
EXPORT_SYMBOL(count_sg_total); EXPORT_SYMBOL(count_sg);
EXPORT_SYMBOL(c3_unmap_sg_chain); EXPORT_SYMBOL(c3_unmap_sg);
EXPORT_SYMBOL(c3_map_sg); EXPORT_SYMBOL(c3_AES_CBC_encrypt);
EXPORT_SYMBOL(c3_AES_CBC_decrypt); EXPORT_SYMBOL(c3_AES_CBC_encrypt_sg);
EXPORT_SYMBOL(c3_AES_CBC_decrypt_sg);
EXPORT_SYMBOL(c3_AES_CTR_encrypt); EXPORT_SYMBOL(c3_AES_CTR_decrypt);
EXPORT_SYMBOL(c3_AES_CTR_encrypt_sg); EXPORT_SYMBOL(c3_AES_CTR_decrypt_sg);
EXPORT_SYMBOL(c3_SHA1_init); EXPORT_SYMBOL(c3_SHA1_append);
EXPORT_SYMBOL(c3_SHA1_append_sg); EXPORT_SYMBOL(c3_SHA1_end);
EXPORT_SYMBOL(c3_SHA1); EXPORT_SYMBOL(c3_SHA1_sg);
EXPORT_SYMBOL(c3_SHA1_HMAC_init); EXPORT_SYMBOL(c3_SHA1_HMAC_append);
EXPORT_SYMBOL(c3_SHA1_HMAC_append_sg); EXPORT_SYMBOL(c3_SHA1_HMAC_end);
EXPORT_SYMBOL(c3_SHA1_HMAC); EXPORT_SYMBOL(c3_SHA1_HMAC_sg);
```

6.3.2 C3 kernel source and configuration

The C3 device driver is composed by the following source code files:

```
drivers/char/c3/c3_driver_interface.c
drivers/char/c3/c3_driver_core.c drivers/char/c3/c3_cryptoapi.c
drivers/char/c3/c3_autotest.c drivers/char/c3/c3_mpcm.c
drivers/char/c3/c3_char_dev_driver_instructions.c
drivers/char/c3/c3_streamplug.c
drivers/char/c3/c3_char_dev_driver.c
drivers/char/c3/c3_registers.c
drivers/char/c3/c3_driver.mod.c
drivers/char/c3/c3_cryptoapi.mod.c
drivers/char/c3/c3_irq.c
arch/arm/mach-streamplug/ipswrst_ctrl.c
arch/arm/mach-streamplug/include/mach/streamplug10.h
arch/arm/mach-streamplug/clock.c
```

which are compiled into the following Linux kernel modules:

```
drivers/char/c3/c3_cryptoapi.ko
drivers/char/c3/c3_driver.ko
```

If the configuration listed in *Table 27* is adopted:

Table 27. C3 Linux kernel configuration

Configuration	Description
CONFIG_C3_DRIVER=m	C3 is a crypto accelerator.
CONFIG_C3_DRIVER_STREAMPLUG1x=y	C3 on STreamPlug1x.
CONFIG_AUTO_TEST=y	This setting enables default kernel level testing for different crypto algorithms.
CONFIG_C3_CRYPTOAPI_INTEGRATION=m	Module for integration with Linux CryptoAPI (kernel version 2.6.37). Support for offloading asynchronous AES (CTR, CBC), SHA-1, HMAC (SHA-1) operations to the C3 hardware.
CONFIG_CRYPTO=y	This option provides the core Cryptographic API.
CONFIG_CRYPTO_ALGAPI=y	This option provides the API for cryptographic algorithms.
CONFIG_CRYPTO_ALGAPI2=y	N/A
CONFIG_CRYPTO_AEAD=m	N/A
CONFIG_CRYPTO_AEAD2=y	N/A
CONFIG_CRYPTO_BLKCIPHER=y	N/A
CONFIG_CRYPTO_BLKCIPHER2=y	N/A
CONFIG_CRYPTO_HASH=y	N/A
CONFIG_CRYPTO_HASH2=y	N/A
CONFIG_CRYPTO_RNG=y	N/A
CONFIG_CRYPTO_RNG2=y	N/A
CONFIG_CRYPTO_PCOMP=y	N/A
CONFIG_CRYPTO_MANAGER=y	Create default cryptographic template instantiations such as CBC (AES).
CONFIG_CRYPTO_MANAGER2=y	N/A
CONFIG_CRYPTO_WORKQUEUE=y	N/A
CONFIG_CRYPTO_AUTHENC=m	Authenc: a combined mode wrapper for IPSec. This is required for IPSec.
CONFIG_CRYPTO_SEQIV=m	This IV generator generates an IV based on a sequence number by XORing it with a salt. This algorithm is mainly useful for CTR.
CONFIG_CRYPTO_CBC=y	CBC: a Cipher-Block Chaining mode. This block cipher algorithm is required for IPSec.
CONFIG_CRYPTO_CTR=m	CTR: a counter mode. This block cipher algorithm is required for IPSec.
CONFIG_CRYPTO_HMAC=m	HMAC: Keyed-Hashing for Message Authentication (RFC2104). This is required for IPSec.
CONFIG_CRYPTO_CRC32C=y	Castagnoli, et al.: cyclic redundancy check algorithm. Used by iSCSI for header and data digests and by others. See Castagnoli93. The module will be crc32c.
CONFIG_CRYPTO_MD5=y	MD5 message digest algorithm (RFC1321).

Table 27. C3 Linux kernel configuration (continued)

Configuration	Description
CONFIG_CRYPTO_AES=y	AES cipher algorithms (FIPS 197). AES uses the Rijndael algorithm. The Rijndael appears to consistently be a very good performer in both hardware and software across a wide range of computing environments regardless of its use in feedback or non-feedback modes. Its key setup time is excellent, and its key agility is good. Rijndael's very low memory requirements make it very well suited for restricted space environments, in which it also demonstrates an excellent performance. Rijndael's operations are among the easiest to defend against power and timing attacks. The AES specifies three key sizes: 128, 192 and 256 bits. For more information, see http://csrc.nist.gov/groups/ST/toolkit/index.html
CONFIG_CRYPTO_CAST5=y	The CAST5 encryption algorithm (synonymous with CAST-128) is described in RFC2144.
CONFIG_CRYPTO_DES=y	DES cipher algorithm (FIPS 46-2), and triple DES EDE (FIPS 46-3).
CONFIG_CRYPTO_ANSI_CPRNG=y	This option enables the generic pseudo random number generator for cryptographic modules. Uses the algorithm specified in ANSI X9.31 A.2.4. Note that this option must be enabled if CRYPTO_FIPS is selected.

6.3.3 C3 platform configuration

The C3 device driver defines the following platform configuration:

```
static struct device dev = {
   .init_name = "c3",
   .release = c3_release,
};
```

6.3.4 C3 usage

The C3 device driver module is tested only in kernel space using the following procedure. The autotest loads up automatically when the “c3_driver.ko” module is loaded.

```
$ modprobe c3_driver
```

The test can be stopped by removing the module using the “rmmod” command.

```
$ rmmod c3_driver
```

The C3 autotest log captured from the UART kernel console is shown below.

```
[C3 INFO] - Crypto Channel Controller (c) ST Microelectronics
[C3 INFO] - Driver version = 2.0
[C3 INFO] - Built on Feb 29 2012 at 11:44:22 [C3 INFO] - C3 device found (HID: ffff9000)
[C3 INFO] - SHA1 test [buffer size: 64] throughput : 2976 KBps [C3 INFO] - SHA1 test [buffer size: 128] throughput : 5898 KBps [C3 INFO] - SHA1 test [buffer size: 256] throughput : 11583 KBps [C3 INFO] - SHA1 test [buffer size: 512] throughput : 20897 KBps [C3 INFO] - SHA1 test [buffer size: 1024] throughput : 33573 KBps [C3 INFO] - SHA1 test
```

```
[buffer size: 2048] throughput : 48188 KBps [C3 INFO] - SHA1 test
[buffer size: 4096] throughput : 61686 KBps [C3 INFO] - SHA1 test
[buffer size: 8192] throughput : 71608 KBps

[C3 INFO] - SHA1_HMAC test [buffer size: 64] throughput : 2782 KBps
[C3 INFO] - SHA1_HMAC test [buffer size: 128] throughput : 5541 KBps
[C3 INFO] - SHA1_HMAC test [buffer size: 256] throughput : 10240 KBps
[C3 INFO] - SHA1_HMAC test [buffer size: 512] throughput : 18285 KBps
[C3 INFO] - SHA1_HMAC test [buffer size: 1024] throughput : 30029
KBps [C3 INFO] - SHA1_HMAC test [buffer size: 2048] throughput : 44618
KBps [C3 INFO] - SHA1_HMAC test [buffer size: 4096] throughput : 58514
KBps [C3 INFO] - SHA1_HMAC test [buffer size: 8192] throughput : 69482
KBps [C3 INFO] - SHA512 test [buffer size: 64] throughput : 6037 KBps

[C3 INFO] - SHA512 test [buffer size: 128] throughput : 11962 KBps [C3
INFO] - SHA512 test [buffer size: 256] throughput : 23486 KBps [C3
INFO] - SHA512 test [buffer size: 512] throughput : 45309 KBps [C3
INFO] - SHA512 test [buffer size: 1024] throughput : 84628 KBps [C3
INFO] - SHA512 test [buffer size: 2048] throughput : 150588 KBps

[C3 INFO] - SHA512 test [buffer size: 4096] throughput : 245269 KBps
[C3 INFO] - SHA512 test [buffer size: 8192] throughput : 357729 KBps
[C3 INFO] - SHA512HMAC test [buffer size: 64] throughput : 5423 KBps
[C3 INFO] - SHA512HMAC test [buffer size: 128] throughput : 10756
KBps [C3 INFO] - SHA512HMAC test [buffer size: 256] throughput : 21157
KBps [C3 INFO] - SHA512HMAC test [buffersize: 512] throughput : 40634
KBps

[C3 INFO] - SHA512HMAC test [buffer size: 1024] throughput : 76992
KBps [C3 INFO] - SHA512HMAC test [buffer size: 2048] throughput :
138378 KBps [C3 INFO] - SHA512HMAC test [buffer size: 4096] throughput
: 228826 KBps [C3 INFO] - SHA512HMAC test [buffer size: 8192]
throughput : 339917 KBps [C3 INFO] - DES_CBC test [buffer size: 64]
throughput : 3422 KBps

[C3 INFO] - DES_CBC test [buffer size: 128] throughput : 5378 KBps
[C3 INFO] - DES_CBC test [buffer size: 256] throughput : 10622 KBps
[C3 INFO] - DES_CBC test [buffer size: 512] throughput : 20078 KBps
[C3 INFO] - DES_CBC test [buffer size: 1024] throughput : 33032 KBps
[C3 INFO] - DES_CBC test [buffer size: 2048] throughput : 48530 KBps
[C3 INFO] - DES_CBC test [buffer size: 4096] throughput : 63209 KBps
[C3 INFO] - DES_CBC test [buffer size: 8192] throughput : 74540 KBps
[C3 INFO] - 3DES_CBC test [buffer size: 64] throughput : 2844 KBps [C3
INFO] - 3DES_CBC test [buffer size: 128] throughput : 5663 KBps [C3
INFO] - 3DES_CBC test [buffer size: 256] throughput : 10000 KBps [C3
INFO] - 3DES_CBC test [buffer size: 512] throughput : 16357 KBps

[C3 INFO] - 3DES_CBC test [buffer size: 1024] throughput : 24150 KBps
[C3 INFO] - 3DES_CBC test [buffer size: 2048] throughput : 31556 KBps
[C3 INFO] - 3DES_CBC test [buffer size: 4096] throughput : 37372 KBps
[C3 INFO] - 3DES_CBC test [buffer size: 8192] throughput : 41145 KBps
[C3 INFO] - AES128_CBC test [buffer size: 64] throughput : 3595 KBps
[C3 INFO] - AES128_CBC test [buffer size: 128] throughput : 6124 KBps
[C3 INFO] - AES128_CBC test [buffer size: 256] throughput : 10406 KBps
[C3 INFO] - AES128_CBC test [buffer size: 512] throughput : 20398 KBps

[C3 INFO] - AES128_CBC test [buffer size: 1024] throughput : 34362
KBps [C3 INFO] - AES128_CBC test [buffer size: 2048] throughput : 51979
KBps [C3 INFO] - AES128_CBC test [buffer size: 4096] throughput : 70378
KBps [C3 INFO] - AES128_CBC test [buffer size: 8192] throughput : 85244
KBps [C3 INFO] - AES256_CBC test [buffer size: 64] throughput : 3422
KBps
```

```
[C3  INFO]  - AES256_CBC test [buffer  size:  128] throughput  : 5400  KBps
[C3  INFO]  - AES256_CBC test [buffer  size:  256] throughput  : 10534  KBps
[C3  INFO]  - AES256_CBC test [buffer  size:  512] throughput  : 20480  KBps
[C3  INFO]  - AES256_CBC test [buffer  size:  1024] throughput  : 34711
KBps [C3  INFO]  - AES256_CBC test [buffer  size:  2048] throughput  : 53194
KBps [C3  INFO]  - AES256_CBC test [buffer  size:  4096] throughput  : 72882
KBps [C3  INFO]  - AES256_CBC test [buffer  size:  8192] throughput  : 89237
KBps [C3  INFO]  - [CDD]  Unregistering  character device driver
```

The autotest allows to check the performance of the following algorithms:

- DES/3DES (CBC)
- AES (CBC, CTR)
- Hash (SHA1/SHA512 w/HMAC).

7 Frame buffer drivers

The Linux frame buffer (“fbdev”) is a graphic hardware independent abstraction layer to show graphics on a computer monitor. The word frame buffer means a part of the video memory containing a current video frame, and the Linux frame buffer means “access method to the frame buffer under the Linux kernel”, without relying on system specific libraries such as “SVGALib” or another user space software.

Color liquid crystal display (CLCD)

The CLCD controller provides all the necessary control signals to interface directly to a variety of LCD panels. This section describes the driver for the CLCD controller available on the STreamPlug.

CLCD software overview

The CLCD device driver sits on the top of the CLCD controller and provides all necessary functions for a graphic application via the standard Linux frame buffer interface.

CLCD kernel source and configuration

Following is the detail corresponding to the layout of the driver and kernel configuration:

- The digital blocks CLCD driver is present in “drivers/video/amba-clcd.c”
- The platform data configuration is present in “arch/arm/plat-streamplug/clcd.c”.

The kernel configuration is listed in *Table 28*.

Table 28. CLCD configurations

Configuration	Description
CONFIG_FB_ARMCLCD	Enable the ARM® PrimeCell® PL110 color LCD controller
CONFIG_FB_ARMCLCD_SHARP_LQ043T1DG01	Enable the LCD device controller for the SHARP LQ043T1DG01 model. This is implementation of the “Sharp LQ043T1DG01, a 4.2" color TFT panel. The native resolution is 480 x 272.

CLCD clock source configuration

There are two clock domains in the LCD controller core:

- Bus clock (BCLK) domain:
  - Master and slave interfaces
  - Control and status registers
  - DMA controller
  - Write side of the palette two-port RAM
  - Write side of the input FIFO
  - Interrupt controller
- Pixel clock (PCLK) domain:
  - Read side of the Input FIFO
  - Read side of the palette two-port RAM
  - Pixel unpack
  - Timing and control unit
  - Output formatter

Within the pixel clock domain, there are two versions of PCLK:

- The internal pixel clock (PCLK), which serves as the on-chip clock for the DB9000 pixel pipeline logic
- The external pixel clock (LCD_PCLK), which serves as the off-chip clock to the LCD panel pixel clock input.

The controller's clock generator can derive PCLK and LCD_PCLK from 2 possible sources:

- The input bus clock (BCLK)
- The input pixel clock (PCLK_IN)

The “PCLK_IN” can be selected from different sources. The CLCD synthesizer is being one of them.

In the controller, the clock generator outputs are determined by the pixel clock timing register (PCTR) programming parameters.

In the STreamPlug, “BCLK” is the AHB clock (166 MHz) and “PCLK_IN” can be configured further from several sources in the SoC but outside the controller. The clock selection criteria in the driver is based on following strategy:

First determine if the required pixel CLK for panel can be generated by “BCLK”:

```
LCD_PCLK =  BCLK /(2 +  PCD)
```

If the PCD is not configurable, then select “PCLK_IN” as the source and call “clk_set_rate” to set the desired pixel clock for the panel.

CLCD allocate frame buffer memory

For a max. resolution of 800 x 400 pixels, the panel resolution requires 1.5 Mbytes of the contiguous memory for a frame buffer. During the setup, the routine “dma_alloc_writecombine” is used to allocate the memory for the frame buffer as shown below.

```
static int clcd_setup(struct  clcd_fb *fb)
{
   dma_addr_t dma;
   struct clk *clk;
   int ret =  0;

   clk =  clk_get(&fb->dev->dev,  "hclk");
   if (IS_ERR(clk))  {
      dev_err(&fb->dev->dev,  "failed  to get clcd clock\n");
      ret =  PTR_ERR(clk);
      goto out;
   }
   clk_enable(clk);

/* Detect  which LCD panel is  connected */
#ifdef  CONFIG_FB_ARMCLCD_SHARP_LQ043T1DG01
   fb->panel =  &sharp_LQ043T1DG01_in;
#endif
#ifdef  CONFIG_FB_ARMCLCD_SAMSUNG_LMS700
   fb->panel =  &samsung_LMS700_in;
#endif

   fb->fb.screen_base  =  dma_alloc_writecombine(&fb->dev->dev,  FRAMESIZE,
       &dma,  GFP_KERNEL);
   if (!fb->fb.screen_base)  {
      printk(KERN_ERR  "CLCD:  unable to map framebuffer\n");
      return -ENOMEM;
   }
   fb->fb.fix.smem_start  =  dma;
   fb->fb.fix.smem_len  =  FRAMESIZE;

   out:
      return ret;
}
```

CLCD platform data

Default configuration of the digital blocks CLCD controller depends on the platform data passed from the boards through functions defined in “arch/arm/plat-streamplug/clcd.c”. The CLCD main structure is shown below.

```
static struct clcd_panel sharp_LQ043T1DG01_in  =  {
   .mode  =  {
      .name  =  "Sharp  LQ043T1DG01",
      .refresh  =  0,
      .xres  =  480,
      .yres  =  272,
      .pixclock  =  KHZ2PICOS(9000),
      .left_margin  =  2,
      .right_margin  =  2,
      .upper_margin  =  2,
      .lower_margin  =  2,
      .hsync_len  =  41,
      .vsync_len  =  11,
      .sync  =  0, //FB_SYNC_HOR_HIGH_ACT  | FB_SYNC_VERT_HIGH_ACT,
      .vmode  =  FB_VMODE_NONINTERLACED,
   },
   .width  =  -1,
   .height  =  -1,
   .tim2  =  TIM2_IOE  | TIM2_CLKSEL | TIM2_BCD,
   .cntl  =  CNTL_LCDTFT  | CNTL_BGR,
   .bpp  =  32,
   .power_sleep  =  0,
};
```

CLCD usage

In order to test the CLCD display the following utilities are available:

- “fbv” - a frame buffer image viewer (provided as a buildroot package) that supports multiple image formats
- “clcd”, found in the folder “/example/jpeg”, that displays the result of hardware JPEG decoding on the CLCD.
- “gpm” - a tool to support mouse interactivity in Linux. GPM utilities are provided within mountable auxiliary FS.

To enable the CLCD support at run-time it is necessary to configure the Linux kernel command line using the options listed in *Table 4 on page 22*. To run “fbv” use the following command line:

```
$  fbv -k  -f "image  to display"
```

The two options adapt the image to the screen size (-k) and frame buffer color depth (-f), only in case LCD bpp is less than the image bpp.

To run the CLCD use the following command line:

```
$  ./clcd  "image  to display"
```

The program decodes the JPEG directly to the LCD frame buffer.

Some frame buffer options can be configured directly from the shell by accessing controls export through “sysfs”. Some useful options are:

- Control blinking cursor:

```
$ echo 0 > /sys/class/graphics/fbcon/cursor_blink # to disable
$ echo 1 > /sys/class/graphics/fbcon/cursor_blink # to enable
```

- clcd sleep mode:

```
$ echo 1 > /sys/class/graphics/fb0/blank # to enter
$ echo 0 > /sys/class/graphics/fb0/blank # to leave
```

To test the CLCD with the “gpm” utility it is necessary to connect a keyboard and a mouse through a USB hub to the board. Then the USB host and CLCD device driver have to be enabled in the Linux command line using the options listed in *Table 4 on page 22*. If a CLCD is connected and enabled, a shell login is automatically redirected on the CLCD panel.

By running the following command

```
$ gpm -m /dev/input/mouse0 -t imps2
```

GPM will stay in background as a daemon and the cursor on the CLCD screen will follow the movements done with the mouse. To stop it, apply the following command:

```
$ gpm -k
```

8 Miscellaneous devices

This section contains information on drivers which are not part of the other sections.

8.1 General purpose input/output (GPIO)

The general purpose input/output (GPIO) is a flexible software controlled digital signal. Each GPIO represents a bit connected to a particular pin, or “ball” on ball grid array (BGA) packages. Board schematics show which external device connects to which GPIOs. Drivers can be written generically, so that the board setup code passes such pin configuration data to drivers.

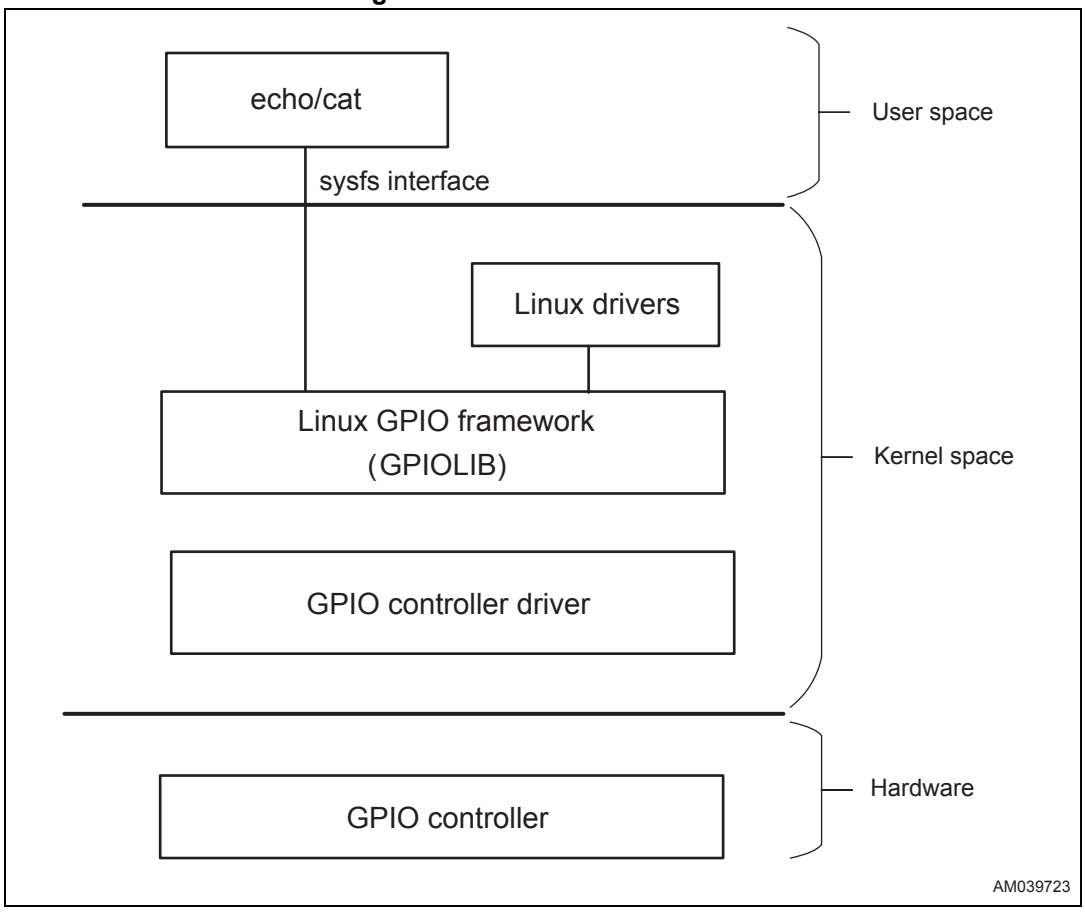
8.1.1 GPIO software overview

Each GPIO input/output can be controlled in the software mode through an APB interface. The APB interface generates read and write decodes for accesses to control, interrupt, and data registers. A read-only decode is provided to access the ID codes. The APB interface implements the storage elements for the data, data direction, mode control, interrupt interface, and identification registers. The GPIO can be accessed from two levels in Linux:

- From the user space using the “sysfs” interface
- From other kernel modules

The GPIO software system architecture is shown in *Figure 16*.

Figure 16. GPIO software stack



8.1.2 GPIO kernel source and configuration

The GPIO device driver is implemented by the following source files:

- The core features are implemented in “drivers/gpio/pl061.c”
- Common functions are in “drivers/gpio/gpiolib.c”

Table 29 lists the kernel configuration options related to GPIO support.

Table 29. GPIO configurations

Configuration	Description
CONFIG_GPIO_PL061	This enables support for the prime cell pl061 GPIO device.
CONFIG_ARCH_REQUIRE_GPIOLIB	Selecting this from the architecture code will cause the “gpiolib” code to always get built in.
CONFIG_GPIOLIB	This enables GPIO support through the generic GPIO library.
CONFIG_GPIO_SYSFS	This enables the “sysfs” interface for GPIOs.

8.1.3 GPIO platform configuration

The GPIO driver, both for pl061 and plgpios are implemented in the GPIO framework. Both these drivers must be informed about the platform details in order to work properly. All this information is purely SoC dependent and has already been provided from the corresponding SoC file to the driver while initialization and the user has not bother about them. The user can directly look into the source for more information on this aspect.

8.1.4 GPIO usage

The following GPIO user mode operations are allowed from the user space: the request, free, set and get direction, set and get value.

Request:

The user space may ask the kernel to export control of a GPIO pin to the user space by writing its number to this file “/sys/class/gpio/export”.

Example: export a node

The following line creates a “gpio24 node” for GPIO #24, if it is not requested by the kernel code:

```
#  echo  24  >  /sys/class/gpio/export
```

Free:

The user space may ask the kernel to take back control of a GPIO pin from the user space by writing its number to this file “/sys/class/gpio/unexport”.

Example: unexport a node

The following line removes the “gpio24” node exported using the “export” file:

```
#  echo  24  >  /sys/class/gpio/unexport
```

Set and get direction:

Once a GPIO pin is exported, the files “direction” and “value” appear under the “/sys/class/gpio/gpiopin/” folder. The direction of the GPIO can be set to OUT or IN by writing “out” or “in” on the “direction” file.

Example: setting direction in the OUT mode

It is possible to set the direction of the pin 32 to out by using the following command:

```
#  echo  "out" >  /sys/class/gpio/gpio32/direction
```

General purpose GPIOs are enumerated from 24 to 39.

The GPIO pin is a file created after exporting a GPIO pin (for example, gpio30, gpio32).

Set and get value:

The value of the GPIO can be configured, if the GPIO is configured in the OUT mode and its value can be read if the GPIO is configured in the IN mode. The value can be set by writing 1 or 0 in the “/sys/class/gpio/gpionr/value” file.

Example: setting GPIO pin 32

The pin 32 value can be set once the following command is used:

```
#  echo  1  >  /sys/class/gpio/gpio32/value
```

GPIO kernel mode

The following GPIO operations are allowed from kernel space: the request, set and get direction and value, and configure the GPIO for interrupt.

Request:

A GPIO pin can be requested by calling following function:

```
/*
* gpio:  is gpio pin number  to be  requested.
* label:  is a  string passed by  user as  a  unique identification  of
user.
*/

int gpio_request(unsigned  gpio,  const char *label);
```

The function “gpio_request()” will fail if an invalid GPIO pin number is used or the requesting GPIOs has already been claimed with the same function call. The return value is 0 if everything worked correctly otherwise a standard linux error is returned.

Free:

After using a previously requested GPIO pin, it must be set free. This can be done by using the following function:

```
/* Gpio:  is gpio pin number  already  requested */

void gpio_free(unsigned  gpio);
```

Passing an invalid GPIO number to “gpio_free()” will fail.

Set direction

After requesting a GPIO pin, its direction must be set. This can be done using following function:

```
/*
* gpio:  is gpio pin number.
* value:  is value to be  set,  0  or 1.
*/

int gpio_direction_output(unsigned  gpio,  int value);
```

The return value is zero for success, otherwise a negative number is returned. The return value must be checked because the get/set calls do not return errors and it is possible to have a wrong configuration. Setting the direction can fail if the GPIO number is invalid, or when that particular GPIO cannot be used in that mode.

For the value parameter, it is preferable to include and use the GPIO macros defined in the corresponding CPU's “gpio.h” file.

Set and get value

After the direction of the GPIO is set, its value can be read or written. A value can be written to a GPIO that is in the OUT mode and can read a value from a GPIO that is in the IN mode. This can be done using following functions:

```
/*
* gpio:  is gpio pin number.
* value:  is value to be  set,  0  or 1.
*/

void gpio_set_value(unsigned  gpio,  int value);

/* gpio:  is gpio pin number.  */

int gpio_get_value(unsigned  gpio);
```

The values are zero to mean “low” and nonzero to mean “high”.

To enable the GPIO support at run-time it is necessary to configure the Linux kernel command line using the options listed in *Table 4 on page 22*.

8.2 Application specific GPIO (AS GPIO)

The application specific GPIO is a flexible software controlled digital signal. Each GPIO represents a bit connected to a particular pin, or “ball” on ball grid array (BGA) packages. Board schematics show which external hardware connects to which GPIOs. Each AS GPIO works as a general purpose I/O GPIO. However, a subset of AS GPIOs are configurable as a pulse width modulator (PWM).

8.2.1 AS GPIO software overview

The operations of the GPIO pins are controlled by the registers, which can be, written/read by software through the APB interface. The pins can be used as dual function pins. The dual function may be either the PWM output, timer I/O and control, or the UART I/O and control. The dual use selection will be defined at a higher level except for the PWM function. The status of the pins can be captured into a register when commanded to do so by the software or by an external signal. The source of the capture signal is set in a register by the software. The GPIO pin can be used to generate interrupts when the pin is configured as an input. The interrupt generation is programmable. Interrupt can be generated when there is a change in polarity, on the rising/falling edge, or on both the edges of the input signal.

The GPIO module also contains pulse width modulators (PWM). The PWM is capable of operating as a continuous repetitive pulse generator, or a single pulse generator. The duration of high and low time for a repetitive pulse and the duration for a single pulse are programmable. The PWM can be enabled/disabled by writing to the PWM enable register. The polarity for a single pulse generator is also programmable. On reset the PWM output is set to low. The PWM operates on the APB clock (PCLK).

8.2.2 AS GPIO kernel source and configuration

The GPIO device driver is implemented by the following source files:

- The core features are implemented in “drivers/gpio/ark_gpio.c”
- Common functions are in “drivers/gpio/gpiolib.c”

Table 30 lists the kernel configuration options related to AS GPIO support.

Table 30. AS GPIO configurations

Configuration	Description
CONFIG_GPIO_ARK	This option enables support for AS GPIO.
CONFIG_ARCH_REQUIRE_GPIOLIB	Selecting this from the architecture code will cause the “gpiolib” code to always get built in.
CONFIG_GPIOLIB	This enables GPIO support through the generic GPIO library.
CONFIG_GPIO_SYSFS	This enables the “sysfs” interface for GPIOs.

8.2.3 AS GPIO platform configuration

The optional platform data passed from the machine for AS GPIOs is as follows:

```
/* gpio device registration  */
struct ark_gpio_platform_data ark_gpio_plat_data =  {
   .gpio_base=  (unsigned)-1,
   .irq_base=  STREAMPLUG1X_IRQ_ARK_GPIO_BASE,
   .enable_mask  =  ark_gpio_mask,
   .groups  =  STREAMPLUG_ARK_GPIO_GROUPS,
   .set_groups  =  STREAMPLUG_ARK_GPIO_SET_GROUP,
};

/* ark_gpio device registration */
struct  resource ark_gpio_resources[]  =  {
{
   .name  =  "ark_gpio",
   .start  =  STREAMPLUG1X_ICM2_ARK_GPIO_BASE,
   .end  =  STREAMPLUG1X_ICM2_ARK_GPIO_BASE  +  SZ_4K - 1,
   .flags  =  IORESOURCE_MEM,
   }, {
   .name  =  "ark_gpio_irq",
   .start  =  STREAMPLUG1X_IRQ_APP_SUBS_ARK_GPIO,
   .flags  =  IORESOURCE_IRQ,
   },
};

struct platform_device  streamplug10_ark_gpio_device =  {
   .name  =  "streamplug_ark_gpio",
   .id  =  -1,
```

```
    .num_resources  =  ARRAY_SIZE(ark_gpio_resources),
    .resource  =  ark_gpio_resources,
    .dev  =  {
    .platform_data  =  &ark_gpio_plat_data,
    },
};
```

8.2.4 AS GPIO usage

AS GPIOs are enumerated from 0 to 23. Only a subset of AS GPIOs is configurable in the PWM mode (GPIO [0:7]), the other ones works in I/O modality. Within the two subgroups of 4 GPIOs of GPIO [0:7], the first one handles dual PWMs, while the second one handles single PWMs.

I/O

The following GPIO operations are allowed from the user space: the request, free, set and get direction, set and get value, configure PWMs.

Request:

The user space may ask the kernel to export control of a GPIO pin to the user space by writing its number to this file “/sys/class/gpio/export”.

Example: export a pin

The following line creates a “gpio8 node” for GPIO #8, if it is not requested by the kernel code:

```
#  echo  8  >  /sys/class/gpio/export
```

Free:

The user space may ask the kernel to take back control of a GPIO pin from the user space by writing its number to this file “/sys/class/gpio/unexport”.

Example: unexport a pin

The following line removes the “gpio8” node exported using the “export” file.

```
#  echo  8  >  /sys/class/gpio/unexport
```

Set and get direction:

Once a GPIO pin is requested, the files “direction” and “value” can be found under the “/sys/class/gpio/gpio<pin number>/” folder. The direction of the GPIO can be set to OUT or IN by writing “out” or “in” in the “direction” file.

Example: setting direction in the OUT mode

The pin 16 can be set in the OUT mode with the following command:

```
#  echo  "out" >  /sys/class/gpio/gpio16/direction
```

Set and get value:

The value of the GPIO can be configured, if the GPIO is configured in the OUT mode and its value can be read if the GPIO is configured in the IN mode. The value can be set by writing 1 or 0 in the “/sys/class/gpio/gpio<pinnumber>/value” file.

Example: setting GPIO pin 16

The pin 16 value can be configured with the following command:

```
#  echo  1  >  /sys/class/gpio/gpio16/value
```

PWM configuration

When exporting a PWM GPIO the following files are created:

“pwm_mode”: to select to the output type, where 0 means a simple value mode, 1 is the PWM mode.

A set of parameters for the PWM configuration:

“pwm_enable”: to enable the PWM output, 1 means enable and 0 means disable.

“pwm_type”: to select between **“**continuous wave**”** and **“**one pulse one**”**: 0 -> means continuous, while 1 one pulse only.

“pwm_high”: to define the duration of the high phase in clock units, applicable only when the pwm_type is equal to 0.

“pwm_low”: to define the duration of the low phase in clock units, applicable only when the pwm_type is equal to 0.

“pwm_width”: to define the duration of the pulse width in clock units, applicable only when the pwm_type is equal to 1.

“pwm_polarity”: to define the polarity of the pulse, applicable only when the pwm_type is equal to 1.

“pwm_prescaler”: as the width of the pulses is configured in clock units a prescaler can be applied to the system one. The prescaler range is from 0 to 14, with the meaning listed in *Table 31*.

Table 31. AS GPIO PWM prescaler configurations

Value	Divider
0	8
1	16
2	32
3	64
4	128
5	256
6	512
7	1024
8	2048
9	4096
10	8192
11	16384
12	32768
13	65536
14	131072

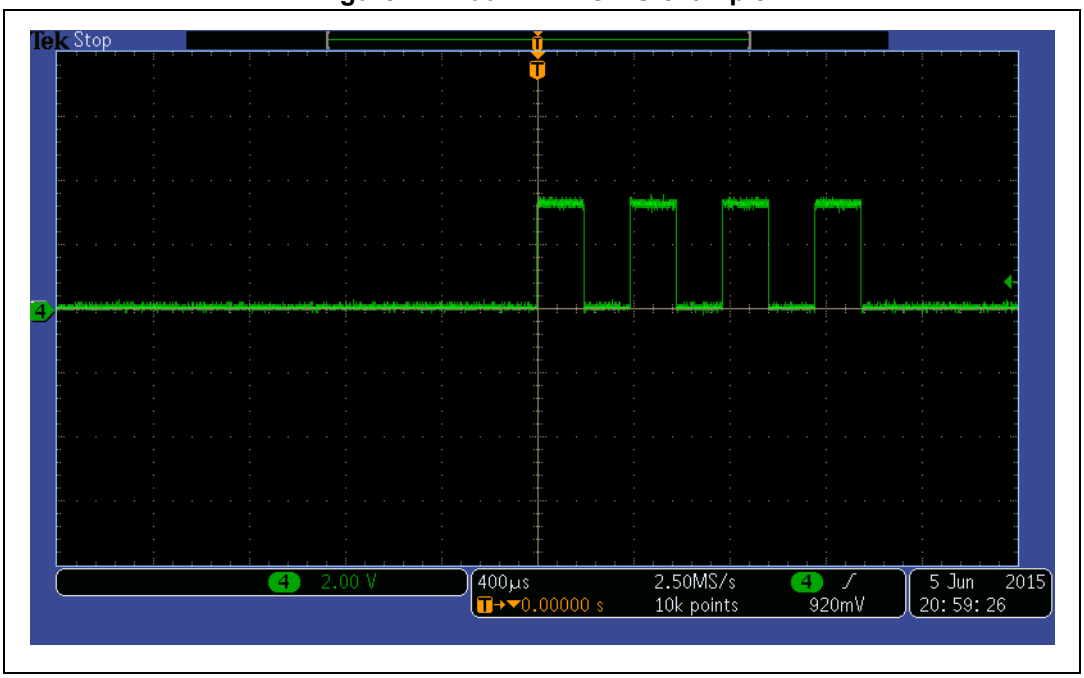
If exporting a dual PWM GPIO (0 - 3) two sets of PWM parameters are available, because the GPIO output is the combination of two PWMs.

Example: dual PWM configuration, pulse combined with continuous

```
# echo 0 > /sys/class/gpio/export
# echo out > /sys/class/gpio/gpio0/direction
# echo 1 > /sys/class/gpio/gpio0/pwm_mode
# echo 1 > /sys/class/gpio/gpio0/pwm_type
# echo 0 > /sys/class/gpio/gpio0/pwm2_type
# echo 1 > /sys/class/gpio/gpio0/pwm_polarity
# echo 10000 > /sys/class/gpio/gpio0/pwm_width
# echo 1000 > /sys/class/gpio/gpio0/pwm2_high
# echo 1000 > /sys/class/gpio/gpio0/pwm2_low
# echo 1 > /sys/class/gpio/gpio0/pwm2_enable ; echo 1 >
/sys/class/gpio/gpio0/pwm2_enable
```

The resulting pin output is captured in *Figure 17*.

Figure 17. Dual PWM GPIO example



PAD MUX configuration

To enable the AS GPIO support at run-time it is necessary to configure the Linux kernel command line using the options listed in *Table 4 on page 22*.

In particular, the value passed on the cmdline is “ark_gpio=on:nnnnnn” where “*nnnnnn*” is the muxing definition.

Each digit represents a group of 4 GPIOs, while the groups are identified by letters from *a* to *f*.

Some of them can be muxed onto different MFIOs, for example:

- Group a => MFIOs 32 - 35 (muxed selection option 1)
- Group b => MFIOs 28 - 31 (muxed selection option 1)
- Group c => MFIOs 16 - 19 (muxed selection option 1) or MFIOs 36 - 39 (muxed selection option 2)
- Group d => MFIOs 20 - 23 (muxed selection option 1) or MFIOs 40 - 43 (muxed selection option 2)
- Group e => MFIOs 48 - 51 (muxed selection option 1) or MFIOs 72 - 75 (muxed selection option 2)
- Group f => MFIOs 52 - 55 (muxed selection option 1) or MFIOs 76 - 79 (muxed selection option 2)

where *0* means the GPIOs are not exposed on pins.

For instance the following ATAG “ark_gpio=on:010011” means:

- AS GPIOs 0 - 3 (group a) off
- AS GPIOs 4 - 7 (group b) in MFIOs 28 - 31
- AS GPIOs 8 -1 1 (group c) off
- AS GPIOs 12 - 15 (group d) off
- AS GPIOs 16 - 19 (group e) in MFIOs 48 - 51
- AS GPIOs 20 - 23 (group f) in MFIOs 52 - 55

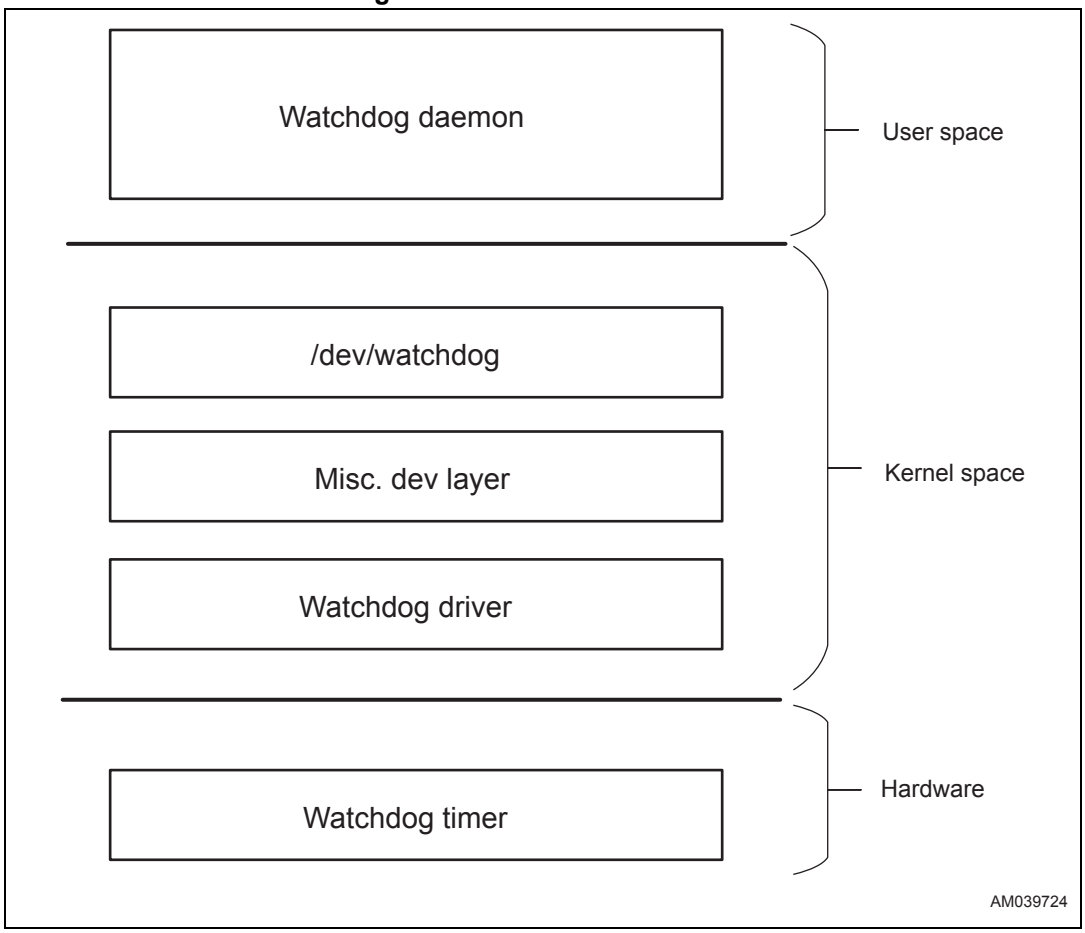
8.3 Watchdog timer (WDT) driver

A watchdog timer is a hardware device that triggers a system reset if its regularly generated interrupts are not acknowledged. The idea behind it is to have a reliable way to bring the system back from the hung state into the normal operation.

8.3.1 WDT software overview

The watchdog driver for the STreamPlug in the Linux support package is a part of the standard Linux watchdog framework. Watchdog drivers in Linux are based on the character device using the Misc device layer and provide a standard set of IOCTLs to the user. Through this interface the watchdog time can be configured, programed and refreshed. The standard Linux watchdog daemon can be used to configure and periodically pat (refresh) the driver in order to avoid system reset. A software crash or hang would thus prevent this pat from a happening and hence cause a system reset after timeout. *Figure 18* illustrates the watchdog framework.

Figure 18. WDT software stack



8.3.2 WDT kernel source and configuration

In the Linux source tree, the watchdog driver is present in the file: “drivers/watchdog/sp805_wdt.c”.

Table 32 lists the kernel configuration options affect the watchdog timer. These configurations can be selected through the “make menuconfig” interface in Linux.

Table 32. WDT Linux kernel configurations

Configuration	Description
CONFIG_WATCHDOG	This enables the watchdog support in the Linux kernel. Enabling this option means that even on closing watchdog the timer would remain active and would eventually reset.
CONFIG_WATCHDOG_NOWAYOUT	The default watchdog behavior is to stop the timer if the process managing it closes the file “/dev/watchdog”. If Y is used here, the watchdog cannot be stopped once it has been started.
CONFIG_ARM_SP805_WATCHDOG	This enables STreamPlug watchdog support.

Watchdog device driver interface with “Misc” device layer

As mentioned, the watchdog driver behaves as a character device, so normal file operations (open, close, ioctl, write) can be used to access its features. For this, the driver uses the “Misc” device layer and registers.

```
static  const struct file_operations streamplug_wdt_fops =  {
   .owner  =  THIS_MODULE,
   .write  =  streamplug_wdt_write,
   .unlocked_ioctl  =  streamplug_wdt_ioctl,
   .open  =  streamplug_wdt_open,
   .release  =  streamplug_wdt_release,
};

/* minor no.  is standard,  defined in miscdevice.h  */
streamplug_wdt_miscdev.minor =  WATCHDOG_MINOR;
streamplug_wdt_miscdev.name  =  "watchdog";
streamplug_wdt_miscdev.fops  =  &streamplug_wdt_fops;

/* register watchdog driver */
ret =  misc_register(&streamplug_wdt_miscdev);
```

Watchdog driver usage

The watchdog device driver provides a char device interface (“/dev/watchdog”) to the user. The standard file operations can be used to open and configure the watchdog device. The following sections explain how the watchdog device can be used.

Opening WDT

The watchdog timer is enabled as soon as it is opened by the user. The usual open call can be used to open the watchdog device.

```
---
char wdt_dev[] = "/dev/watchdog" int fd;

fd =  open(wdt_dev, O_RDWR);
if (fd <  0) {
   printf("Error in opening device\n");
}
---
```

Configuring WDT

IOCTL calls can be used to program and configure the watchdog timer. The following code snippet demonstrates the use of these IOCTLs.

```
int ret =  0;
int timeleft=0;
struct watchdog_info ident;
int  timeout =  45; /* in seconds */

/* to find  out supported options in  watchdog */
```

```
ret =  ioctl(fd,  WDIOC_GETSUPPORT,  &ident);

/* to set time out */
ioctl(fd,  WDIOC_SETTIMEOUT,  &timeout);

/* to find out how much time is left before reset */
ret =  ioctl(fd,  WDIOC_GETTIMEOUT,  &timeleft);

/* Refresh watchdog timer  at  every 10  secs   to prevent reset */
while (1) {
   ioctl(fd,  WDIOC,KEEPALIVE,  0);
   sleep(10);
}
```

Table 33 lists the standard “ioctl” calls supported by the STreamPlug watchdog driver.

Table 33. Watchdog IOCTLs

IOCTLs	Purpose
WDIOC_GETSUPPORT	The fields returned in the “ident” structure are: identity: A string identifying the watchdog driver “firmware_version”: the firmware version of the card if available. Options: A flags describing what the device supports.
WDIOC_KEEPALIVE	This “ioctl” does exactly the same thing as write to the watchdog device and hence refreshes the timer.
WDIOC_SETTIMEOUT	Set time out in seconds, after which reset would be generated (if WDT is not refreshed).
WDIOC_GETTIMEOUT	Query the current timeout.

For more information about Linux kernel support to the watchdog see the file:

`linux-2.6.35/Documentation/watchdog.txt`

8.3.3 WDT usage

A simple watchdog demonstration application is provided in the “/examples/watchdog/” folder to perform some operations on the watchdog peripheral.

The application uses the watchdog device file “/dev/watchdog.” If a such file is not present, the user may create it using the “mknod” command:

```
$  mknod  /dev/watchdog  c  10  130
```

The “*watchdog-demo”* using IOCTL provided by the ARM sp805 driver may be used to set and get the watchdog timeout, to check the last boot cause and to kick the watchdog.

Set the watchdog timeout.

The test procedure is:

1. Open both UART terminals on the PC host and plug cables on RS232 connectors on the board.
2. Run the “watchdog-demo” and verify at power-on that reset was made by “power-on reset” procedure:
   a) With the option “-i <interval time in sec>” the interval time may be changed (passing a decimal integer value) for the next watchdog.
   b) Without the option “-i <interval time in sec>” the interval time set to 60 s is considered the default.

With the “*watchdog-demo”* running, the user can keep the watchdog alive by selecting the option “w” or restarting the interval time with “i”. In this way the user will simulate the kernel behavior.

The user may perform two different tests with the “*watchdog-demo”*:

TEST 1: The user allows the watchdog to time out:

1. The user waits for a time equal to initial interval time.
2. The user will not close the “watchdog-demo” application and watch for OK Linux restarting.
3. At next login, the user will start the “watchdog-demo” and will verify the cause of the reset.

The following is an example of such a test:

```
Welcome  to OKL STreamPlug
STreamPlug  login:  root
#
#
#  mount  /dev/sda  /mnt/
#  cd  /mnt/examples/watchdog/
#  ./watchdog-demo  -i5
Set   watchdog interval to 5
Current  watchdog interval is 5
Last boot is  caused by  : Power-On-Reset
Use:
<w>  to kick through writing  over device file
<i>  to  kick  through IOCTL
<x>  to exit  the program

VMMU:  segment too  big  (80000000) for  index 0
Linux version 2.6.35-vcpu-okl_streamplug+  (developer@Kernel.org)  (gcc
version 4.3.3 ( ?'
Sourcery G++  Lite 2009q1-203) ) #13  Thu  Jul 25  14:35:57 CEST 2013
CPU:  vCPUv5 [14069260] revision 0  (ARMv5TEJ)
CPU:  VIVT  data cache,  VIV  ST-ATag  virq 6a,  "timer_tick"
ATag  microvisor_timer c2,  6a,  "timer_microvisor_timer"
ATag  virq 6b,  "oklinux_signal"
ATag  ksp_agent c3,  6b,  "oklinux_ksp_agent"
ATag  ksp_shared_mem  fd100000,  4800000, 1e00000,  "shm_KSP_SHARED_MEMORY"
```

```
ATag  Shared Buffer  40000000, 80000000, "pci_express"
ATag  vclient c4,  20, 6c,  "vserial_vtty0_vclient"
OKL4:  vcpu_helper_page at  84579000/01fff000
VMMU:paging_init:  VMMU:  Cache  management  handing is  possibly not
correct (SDK-1545). Built 1  zonelists in Zone  order,  mobility  grouping
on.Total pages:  11938
Kernel command line: console=vcon0,115200n8 root=/dev/mtdblock2
rootfstype=ext2,jffs2  clcd ?'
=off pcie=off sata=off usb=on:host eth=on:primary: i2c=off ssp=off
uart1=off uart2=off?'
can=off  firda=off  fsmc=off  sport=off  ts=off  ark_gpio=off
PID  hash  table entries:  256  (order:  -2, 1024  bytes)
Dentry cache  hash  table entries:  8192  (order:  3, 32768  bytes) Inode-
cache  hash  table entries:  4096  (order:  2, 16384  bytes) Memory:  47MB
=  47MB  total
Memory:  42468k/42468k available,  5660k  reserved,  0K  highmem
Virtual  Kernel memory  layout:
vector   : 0x01fff000 - 0x02000000     (   4  kB)
fixmap   : 0xfff00000 - 0xfffe0000     ( 896  kB)
DMA      : 0xef600000 - 0xf0000000     (  10  MB)
vmalloc  : 0x87000000 - 0xe9e00000     (1582  MB)
lowmem   : 0x84000000 - 0x86f00000     (  47  MB)
modules  : 0x83000000 - 0x84000000     (  16  MB)
.init   : 0x84000000 - 0x84026000     ( 152  kB)
.text   : 0x84026000 - 0x844ad000   (4636  kB)
.data   : 0x844c8000 - 0x844fbbe0    ( 207  kB)
Hierarchical RCU  implementation.
Verbose stalled-CPUs  detection is disabled. NR_IRQS:121
Console:  colour dummy device 80x30
Calibrating  delay loop...  164.65 BogoMIPS  (lpj=823296)
....
....
rtc-streamplug  rtc-streamplug:  rtc core:  registered rtc-streamplug  as
rtc0 i2c /dev  entries driver
Linux video capture interface:  v2.00
HM1355 driver  loaded
sp805-wdt  wdt:  registration  successful
dw_dmac:  DesignWare DMA  Controller,  8  channels
usbcore:  registered new interface driver usbhid
usbhid:  USB HID  core driver
No device for  DAI  AKCODEC
....
....
Welcome  to OKL STreamPlug
STreamPlug  login:  root
#
```

```
#
#  mount  /dev/sda  /mnt/
#  cd  /mnt/examples/watchdog/
#  ./watchdog-demo
Current  watchdog interval is 60
Last boot is  caused by  : Watchdog
Use:
<w>  to kick through writing  over device file
<i>  to  kick  through IOCTL
<x>  to exit  the program
```

TEST 2: the user will continue to refresh the watchdog:

1. The user may refresh before interval time expiring with one of the following options:
   a) “w”
   b) “i”
2. Run 'w' at least on time in order to refresh interval time to the value passed with 'i' or to default one
3. Do not close watchdog-demo and wait for OK Linux restart

The option “x” closes the “watchdog-demo”, leaving the watchdog control back to OK Linux kernel.

Interacting with watchdog via device file

The watchdog is automatically started. To stop the watchdog: write character “V” into “/dev/watchdog” to prevent stopping the watchdog accidentally and close the “/dev/watchdog” file.

To “kick” or to “feed” the watchdog any character can be written into the “/dev/watchdog” file.

Watchdog daemon

The watchdog is a daemon. It opens “/dev/watchdog” and keeps writing to it often enough to keep the kernel from resetting, at least once per minute. Each write delays the reboot time another minute. After a minute the watchdog hardware generates a reset. The watchdog can be stopped without causing a reboot if the device “/dev/watchdog” is closed correctly, unless the kernel is compiled with the “CONFIG_WATCHDOG_NOWAYOUT” option enabled.

The default timeout period can be programmed by passing an argument to the watchdog daemon in following manner:

```
$  watchdog -T  60
```

A “V” character writing causes a watchdog to stop. (See the “starting- stopping watchdog” point above).

9 Audio drivers

This section describes the drivers that can be used for audio.

SPORT controller

The DSP's serial port (SPORT) can be utilized for a direct connection to a DSP or an other device with a high speed serial interface. The SPORT controller is integrated interchip sound (I^2S) compliant that is an electrical serial bus interface standard used for connecting digital audio devices together. The controller provides a simple I^2S interface to industry standard audio components. It supports the standard I^2S frame format for transmitting and receiving audio data.

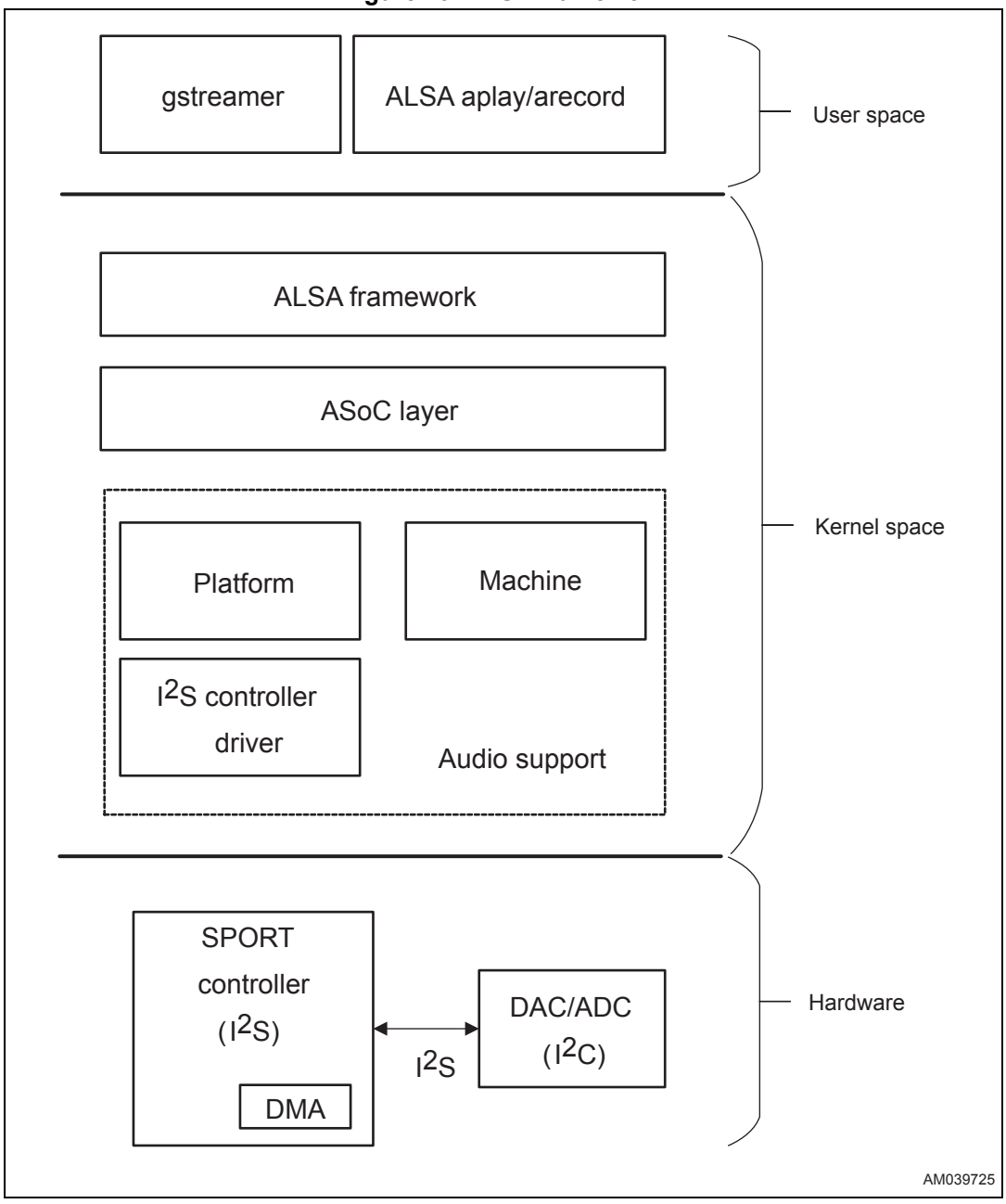
It features:

- Synchronous serial data transfer
- Dedicated transmit and receive data lines
- Supports full duplex devices for simultaneous data transfer in both directions
- Independent transmit and receive clocks. It provides an internally generated serial clock and frame sync signals in a wide range of frequencies, or accepts the clock and frame sync input from an external source.
- Perform interrupt driven, single word transfers to and from the on-chip memory, controlled by the processor.
- Execute DMA transfers to and from the on-chip memory where the SPORT interface can automatically receive or transmit an entire block of data.
- Three 32-bit word length, programmable frame “sync” for every transmitted or received word
- Can be configured to use early frame “sync”.

SPORT controller software overview

The I^{2}S driver is implemented within the “Advanced Linux Sound Architecture” (ALSA) framework shown in *Figure 19*. The ALSA provides suitable layers and APIs to support complex sound systems where it provides proper abstraction, so that each layer can be independent of others. As a consequence the applications dealing with audio remain immune to hardware and at the same time plugging new hardware is relatively easy.

Figure 19. ALSA framework



There is another layer of abstraction under the ALSA framework for embedded audio environment. The abstraction is known as the ALSA system-on-chip (ASoC) layer. The overall project goal of the ASoC layer is to provide better ALSA support for embedded system-on-chip processors and portable audio codecs.

The ASoC layer is designed to address these issues and provide the following features:

- Codec independence. It allows reuse of codec drivers on other platforms and machines.
- Easy I^2S/PCM audio interface setup between the codec and SoC. Each SoC interface and codec registers its audio interface. Capabilities with the core and codec are subsequently matched and configured when the application hardware parameters are known.

To achieve all this, the ASoC basically splits an embedded audio system into three components:

Codec driver

The codec driver is platform independent and contains audio controls, audio interface capabilities, the codec DAPM definition and codec I/O functions.

Platform driver

The platform driver contains the audio DMA engine and audio interface drivers.

Machine driver

The machine driver handles any machine specific controls and audio events. Complete STreamPlug audio support is within the ASoC framework. It has both the play and record feature.

SPORT controller kernel source and configuration

STreamPlug audio support is available below “sound/soc/streamplug/folder”.

- The I^2S controller driver is present in “sound/soc/streamplug/streamplug_i2s.c”.
- The platform data defining the I^2S PCM data format and PCM rate are present in “sound/soc/streamplug/streamplug-i2s.h”.
- The STreamPlug ASoC platform driver is present in “sound/soc/streamplug/streamplug_pcm.c”.
- STreamPlug ASoC machine implementation can be found in “sound/soc/streamplug/streamplug-sport.c”.

Table 34 lists the “Kconfig” options needed to enable the audio support over I^{2}S for the STreamPlug architecture.

Table 34. SPORT- I^{2}S configurations

Configuration	Description
CONFIG_SOUND	It enables ALSA sound system support
CONFIG_SND	It enables ALSA for SoC audio support
CONFIG_SND_PCM	It enables PCM for SoC audio support
CONFIG_SND_STREAMPLUG_SOC	It enables the ALSA SoC for the STreamPlug
CONFIG_SND_STREAMPLUG_SOC_I2S	It enables I^{2}S support
CONFIG_SND_STREAMPLUG_SOC_SPORT	It enables machine support
CONFIG_SND_STREAMPLUG_SOC_VB	it enables audio support
CONFIG_SND_SOC_AKCODEC	It enables the codec

SPORT controller platform configuration

In audio some essential parameters are required to play or record a song. These are the sample rate, sample format, number of channel, etc. These parameters are passed from the platform code. Platform data is used to pass the I^{2}S capability like the maximum channel, formats, rates, etc. This depends on the HW capability of the I^{2}S controller and the manner in which the platform intends to use it.

The following structure, “sport_platform_data”, is used to pass these capabilities which is defined in “include/linux/streamplug_sport_reg.h”.

```
struct sport_platform_data {
   #define PLAY(1 << 0)
   #define RECORD(1 << 1)
   unsigned int cap;
   int channel;
   u8 swidth;
};
```

In above structure, the fields are:

cap

It is used to configure the PCM capability and can be equal to:

- PLAY, or
- RECORD

channel

The maximum number of channels supported by the controller.

snd_fmts

Sound formats like ”SNDRV_PCM_FMTBIT_S16_LE”, etc. supported by the controller. Available formats are defined in “include/sound/pcm.h”.

snd_rates

Sampling rates like “SNDRV_PCM_RATE_48000”, etc. supported. These are defined in “include/sound/pcm.h/”.

play_dma_data

Configure the DMA channel for playing. This is DMA specific structure which can vary from the platform to platform. It will configure the TX line, transfer burst size, etc. Please refer to the DMA slave configuration section in *Section 6.2: Direct memory access (DMA) on page 139*.

capture_dma_data

Similar to “play_dma_data”, but for the capture interface.

bool (*filter)(struct dma_chan *chan, void *slave)

This is also DMA specific data which is called on the requesting DMA channel to validate the channel selection. Please refer DMA slave configuration section in *Section 6.2*. for details.

int (*i2s_clk_cfg)(struct i2s_clk_config_data *config)

The function is used for run-time audio clock configuration which is responsible to generate the correct reference, bit and word clock. These clocks depend on the sample rate, sample bit and number of the channel.

struct i2s_clk_config_data

This defines the clock related configuration data according to which the “i2s_clk_cfg” programs the required I^2S clocks.

The platform data passed to the “SPORT” controller driver is set below “arch/arm/mach-streamplug/streamplug1x.c”:

```
static struct sport_platform_data sport_data =  {
   .cap  =  PLAY | RECORD,
   .channel  =  1,
   .swidth  =  4,
};
```

Registration is set according to:

```
static struct resource sport_resources[]  =  {
{
   .name=  "sport",
   .start=  STREAMPLUG1X_ICM2_SPORT_BASE,
   .end       =  STREAMPLUG1X_ICM2_SPORT_BASE  +  SZ_4K - 1,
   .flags=  IORESOURCE_MEM,
   },
   {
   .name=  "sport_irq",
   .start=  STREAMPLUG1X_IRQ_APP_SUBS_TS_SPORT,
   .flags=  IORESOURCE_IRQ,
   }
};
```

```
struct platform_device  streamplug1x_sport_device =  {
   .name  =  "streamplug-sport",
   .id  =  -1,
   .dev  =  {
   .coherent_dma_mask  =  ~0,
   .platform_data  =  &sport_data,
   },
   .num_resources  =  ARRAY_SIZE(sport_resources),
   .resource  =  sport_resources,
};

struct  platform_device streamplug_pcm_device =  {
   .name  =  "streamplug-pcm",
   .id  =  -1,
};
```

SPORT controller usage

There are standard utilities available in the Linux kernel for play, capture and control interfaces:

- “aplay”, “alsaplay” and “play” to play audio files
- “arecord” to record audio into a file
- “alsacntl” to control interfaces features like the master volume control, L/R volume control and ADC gain

Another user space application that performs both audio OUT/IN is the GStreamer that is based also on the ALSA framework. Basic command for play an audio file is:

```
$  aplay *.wav
```

while to record audio it is possible to use the following command:

```
$  arecord -r 48000  -f S16_LE  foo.wav
```

where: “*-r”* is for the rate and *“-f* “is for the format.

ALSA has its own “proc filesystem” tree (“/proc/asound”) where many useful information can be found. The most useful are:

- “/proc/asound/card0”, card0 directory exists for the sound card the system knows about the PCM devices area available on the card, there will be directories such as “pcm0p” or “pcm0c” (the latest char is for “p = playback”, “c = capture”). They hold the PCM information for each PCM stream.
- “/proc/asound/cards” lists the card specific files
- “/proc/asound/pcm” lists the allocated pcm streams.
- “/proc/asound/version” lists the version and date the ALSA subsystem module (or kernel) was built.

10 Video drivers

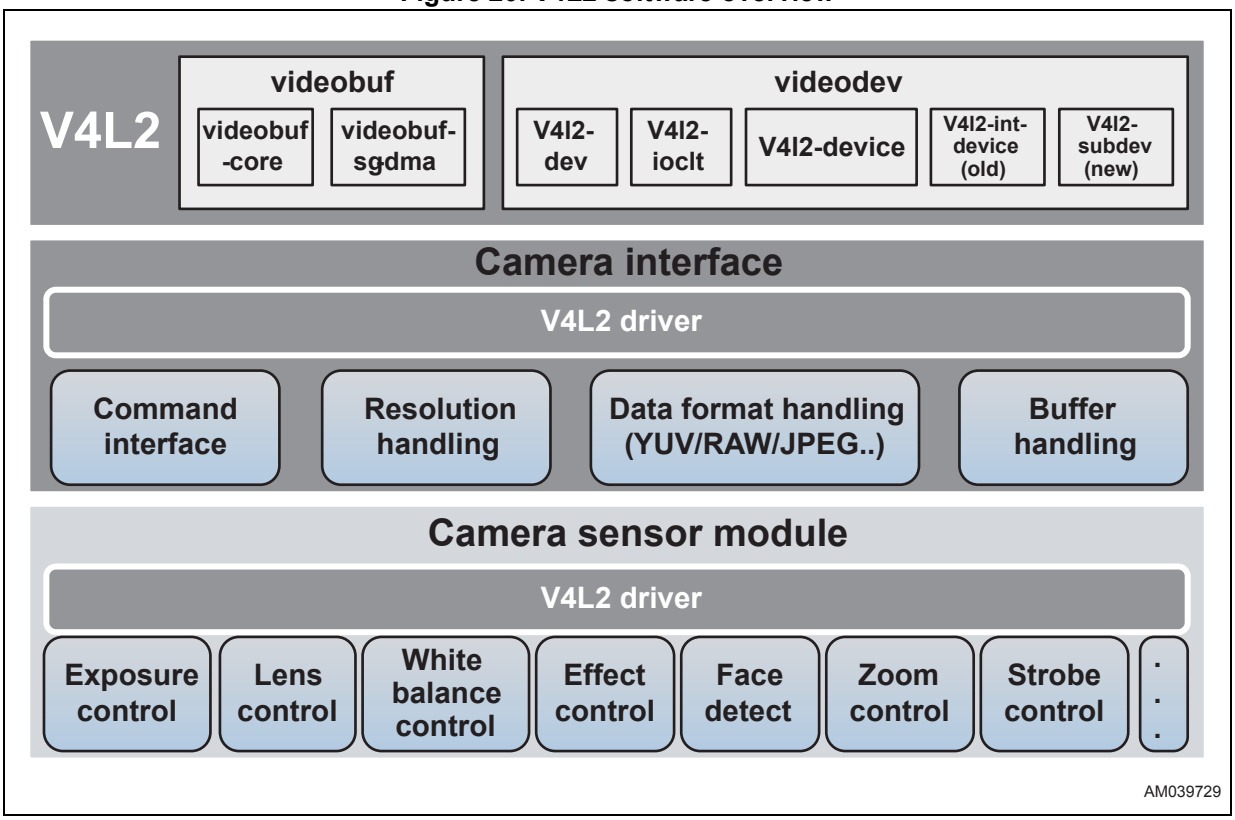
This section describes the drivers that can be used for a video.

10.1 Video for Linux Two framework

The “Video For Linux Two” is the second version of the “Video For Linux API”, a kernel interface for the analog radio, video capture and video output drivers. This section talks only about the V4L2 interface for video capture devices. (For the details of the V4L2 interface with other devices like analog radio, please refer to www.linuxtv.org/downloads/v4l-dvb-apis/).

Figure 20 illustrates the V4L2 subsystem.

Figure 20. V4L2 software overview



Details of the V4L2 framework can be found as well in the standard Linux documentation “Documentation/video4linux/”.

Programming a V4L2 device

Programming a V4L2 device consists of these steps:

1. Opening the device
2. Changing device properties, selecting a video input, video standard, picture brightness, etc.
3. Negotiating a data format
4. Negotiating an input/output method
5. The actual input/output loop
6. Closing the device

In practice, most steps are optional and can be executed out of order (depending on the V4L2 device type). To open and close V4L2 devices applications use the “open()” and “close()” functions, respectively. Devices are programmed using the “ioctl()” function as explained in the following sections. This section provides the details of only those IOCTLs which are required to capture streaming data from a simple video capture device. An example of such standard V4L2 capture application can be seen at http://linuxtv.org/downloads/v4l-dvb-apis/capture-example.html.

For a detailed discussion of all V4L2 IOCTLs please refer to http://www.linuxtv.org/downloads/v4l-dvb-apis/.

Opening and closing the driver

A video capture device can be opened using an “open()” call from the application with the device name and mode of operation as parameters. The application can open the driver in either blocking mode or non-blocking mode as shown in the code snippet below:

```
/* open  a  video capture device in  blocking mode */
fd_blocking =  open("/dev/video0",  O_RDWR);

/* open  a  video capture device in  non-blocking  mode */
fd_nonblocking =  open  ("/dev/video0",  O_RDWR  | O_NONBLOCK);
```

The application can call the “close()” function with the respective file handle to close a specific device as shown in the code snippet below:

```
/* closing a  video capture device as  per the mode */
close (fd_blocking);

close (fd_nonblocking);
```

Video buffer management

A V4L2 driver allows two different types of memory allocation modes for allocating video buffers:

- Driver-buffer mode (“MMAP I/O” method), for the MMAP I/O method, the application requests memory from the driver by calling “VIDIOC_REQBUFS ioctl”. In this method, the maximum number of buffers is limited to “VIDEO_MAX_FRAME” (which is usually set to 32).
- User-buffer mode (user pointer I/O method), for the user pointer method, the application needs to allocate physically contiguous memory using some other mechanism in the user space and then provide a pointer to this memory to the video capture driver.

This section only presents the MMAP I/O method (for details of user pointer I/O method please refer to standard Linux documentation “Documentation/video4linux/videobuf”). Below are the major steps the application needs to perform for the buffer allocation using the MMAP I/O method.

Allocating video buffers using MMAP I/O method:

The “ioctl” used by the user space application to allocate video buffers is “VIDIOC_REQBUFS”. This “ioctl” takes the following arguments:

- Pointer to the instance of “v4l2_requestbuffers” structure
- Buffer type (set to” V4L2_BUF_TYPE_VIDEO_CAPTURE” for capture devices)
- Number of buffers desired, and
- Memory type (set to “V4L2_MEMORY_MMAP”).

The following code snippet depicts how to use the “VIDIOC_REQBUFS ioctl”:

```
struct  v4l2_requestbuffers reqbuf;  /* buffer request  parameters */

reqbuf.count  =  numbuffers;
reqbuf.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.memory  =  V4L2_MEMORY_MMAP;

ret =  ioctl(fd , VIDIOC_REQBUFS,  &reqbuf);

if(ret)  {
   printf("cannot allocate memory\n");
   close(fd);
   return -1;
}
```

Note: It is important to know that this “ioctl” can be called only once from the application. “Numbuffers” must have a value equal or greater than 2.

Mapping the kernel space video buffer address to user space:

Mapping the kernel space video buffer to the user space can be done via “mmap”. The buffer size and physical address of the buffer can be used to get the user space address. The following code snippet depicts how to use the “mmap”:

```
/* allocate buffer by  VIDIOC_REQBUFS  */
...

/* query the buffer  using VIDIOC_QUERYBUF  */
...

/* addr holds the user space   address of the video buffer  */
int addr;

addr =  mmap (NULL  /* start anywhere */, buf.length,
   PROT_READ  | PROT_WRITE  /* required */, MAP_SHARED  /* recommended  */,
   fd,  buf.m.offset);

/* buffer.m.offset  is same as  that returned by  VIDIOC_QUERYBUF  */
```

Not all video capture devices use the same kind of buffers. In fact, there are (at least) three common variations:

1. Buffers which are scattered in both the physical and (kernel) virtual address spaces. (Almost) all user space buffers are like this, but it makes great sense to allocate kernel space buffers this way as well when it is possible. Unfortunately, it is not always possible; working with this kind of the buffer normally requires hardware which can do scatter/gather DMA operations.
2. Buffers which are physically scattered, but which are virtually contiguous; buffers allocated with “vmalloc()”, in other words. These buffers are just as hard to use for DMA operations, but they can be useful in situations where DMA is not available but virtually-contiguous buffers are convenient.
3. Buffers which are physically contiguous. Allocation of this kind of the buffer can be unreliable on fragmented systems, but simpler DMA controllers cannot deal with anything else.

“Videobuf” can work with all three types of buffers, but the driver author must pick one at the outset and design the driver around that decision. Depending on which type of buffers is being used, the driver should include one of the following files:

```
media/videobuf-dma-sg.h                 /* Physically scattered */
media/videobuf-vmalloc.h                /* vmalloc()  buffers      */
media/videobuf-dma-contig.h             /* Physically  contiguous */
```

The driver's data structure describing a V4L2 device should include a “struct videobuf_queue” instance for the management of the buffer queue, along with a “list_head” for the queue of available buffers. There will also need to be an interrupt safe spin lock which is used to protect (at least) the queue. The following “videobuf_queue_ops” are simple callbacks to help the “videobuf” deal with the management of buffers:

```
struct videobuf_queue_ops {
   int (*buf_setup)(struct  videobuf_queue *q, unsigned int *count,
unsigned int *size);
   int (*buf_prepare)(struct  videobuf_queue *q, struct  videobuf_buffer
*vb,
      enum v4l2_field field);
   void (*buf_queue)(struct  videobuf_queue *q, struct  videobuf_buffer
*vb);
   void (*buf_release)(struct  videobuf_queue *q, struct  videobuf_buffer
*vb);
};
```

These callbacks must be implemented by the video capture driver (for details please refer to “Documentation/video4linux/videobuf”).

V4L2 IOCTL handling

This “ioctl” is used to identify video capture devices compatibility with the V4L2 specification and to obtain information about individual hardware capabilities.

Query capabilities (VIDIOC_QUERYCAP)

Capabilities of a video capture device, for example, can be “V4L2_CAP_VIDEO_CAPTURE” and “V4L2_CAP_STREAMING”. The details of this “ioctl” and the mechanisms to use are depicted in the snippet below:

```
struct v4l2_capability capability;
ret =  ioctl(fd,  VIDIOC_QUERYCAP,  &capability);
if(ret)  {
   printf("Cannot  do  QUERYCAP\n");
   return -1;
}

if(capability.capabilities  & V4L2_CAP_VIDEO_CAPTURE)  {
   printf("Capture  capability is supported\n");
}

if(capability.capabilities  & V4L2_CAP_STREAMING)  {
   printf("Streaming  is supported\n");
}
```

where “VIDIOC_QUERYCAP” is the “ioctl” name.

Format enumeration (VIDIOC_ENUM_FMT)

This “ioctl” is used to enumerate the information of the format (horizontal pitch/pixel length, pixel format, etc.) that are supported by underlying the video capture device. The details of this “ioctl” and the mechanisms to use are depicted in the snippet below:

```
struct v4l2_fmtdesc fmt;
int i = 0;

while(1) {
   fmt.index = i;
   ret = ioctl(fd, VIDIOC_ENUM_FMT, &fmt); /* ioctl name:
VIDIOC_ENUM_FMT */
   if(ret) {
   break;
   }

   printf("description = %s\n",fmt.description);
   if(fmt.type == V4L2_CAP_VIDEO_CAPTURE)
   printf("Video Capture type\n");
   if(fmt.pixelformat == V4L2_PIX_FMT_UYVY)
   printf("V4L2_PIX_FMT_UYVY\n");
   i++;
}
```

Set format (VIDIOC_S_FMT)

This “ioctl” is used to set the format for the underlying video capture device. The driver validates the parameters sent as arguments to this ioctl call. It returns an error if parameters are not valid; otherwise, it configures these parameters. The driver calculates the bytes per line and the image size based on the hardware capabilities and the application can retrieve the same using the “VIDIOC_G_FMT” [Get format (VIDIOC_G_FMT)] ioctl. The details of this ioctl and the mechanisms to use are depicted in the snippet below:

```
struct v4l2_format fmt;

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = height; fmt.fmt.pix.width = width;
fmt.fmt.pix.field = V4L2_FIELD_NONE;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
   perror("VIDIOC_S_FMT\n");
   close(fd);
   return -1;
}
```

Get format (VIDIOC_G_FMT)

This “ioctl” is used to get the current format from the underlying video capture device. The driver provides format parameters in the structure pointer passed as an argument. The details of this ioctl and the mechanisms to use are depicted in the snippet below:

```
struct  v4l2_format fmt;
fmt.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;
ret =  ioctl(fd,  VIDIOC_G_FMT,  &fmt);
if(ret)  {
   perror("VIDIOC_G_FMT\n");
   close(fd);
   return -1;
}
```

Try format (VIDIOC_TRY_FMT)

This “ioctl” is used to validate a specific format for the underlying video capture device. The capture driver does know hardware changes for this ioctl. It just checks if it can support the requested format. The driver returns an error if parameters are not valid. The details of this ioctl and the mechanisms to use are depicted in the snippet below:

```
struct  v4l2_format fmt;

fmt.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.pixelformat  =  V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height  =  height; fmt.fmt.pix.width  =  width;
fmt.fmt.pix.field  =  V4L2_FIELD_NONE;

ret =  ioctl(fd,  VIDIOC_TRY_FMT,  &fmt);
if(ret)  {
   perror("VIDIOC_TRY_FMT\n");
   close(fd);
   return -1;
}
```

Query crop capabilities (VIDIOC_CROPCAP)

This “ioctl” is used to query the crop capabilities of the underlying capture device. Applications use this to query the cropping limits, the pixel aspect of the image, and to calculate the scale factor. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
struct v4l2_cropcap cropcap;

memset(&cropcap,0,sizeof  (cropcap));
ret =  ioctl(fd,  VIDIOC_CROPCAP,  &cropcap);
if(ret)  {
   perror("VIDIOC_CROPCAP\n");
   close(fd);
   return -1;
}
```

Set crop (VIDIOC_S_CROP)

To change the cropping rectangle applications initialize the type and “struct v4l2_rect” substructure named “c” of a “v4l2_crop structure” and call the “VIDIOC_S_CROP” ioctl with a pointer to this structure. The driver first adjusts the requested dimensions against hardware limits, i.e.: the bounds given by the capture/output window and it rounds to the closest possible values of the horizontal and vertical offset, width and height. Secondly, the driver adjusts the image size (the opposite rectangle of the scaling process, source or target depending on the data direction) to the closest size possible while maintaining the current horizontal and vertical scaling factor. Finally the driver programs the hardware with the actual cropping and image parameters. “VIDIOC_S_CROP” is a write-only ioctl, it does not return the actual parameters. To query them applications must call “VIDIOC_G_CROP” and “VIDIOC_G_FMT”. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
/* display half of the image   area starting  at 0,0 */

struct  v4l2_crop crop;
crop.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;
crop.c.height  =  image_height/2;
crop.c.width  =  image_width/2;
crop.c.top  =  0;
crop.c.left  =  0;

ret =  ioctl(fd,  VIDIOC_S_CROP,  &crop);
if(ret)  { perror("VIDIOC_S_CROP\n");
   close(fd);
   return -1;
}
```

Get crop (VIDIOC_G_CROP)

This “ioctl” is used by the user space application to get the current crop rectangle bounds. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
struct  v4l2_crop crop;
crop.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;

ret =  ioctl(fd,  VIDIOC_G_CROP,  &crop);
if(ret)  {
   perror("VIDIOC_G_CROP\n");
   close(fd);
   return -1;
}

printf ("top =  %d\n",crop.c.top);
printf("left =  %d\n",crop.c.left);
printf ("height =  %d\n",crop.c.height);
printf("width  =  %d\n",crop.c.width);
```

Queue a video buffer (VIDIOC_QBUF)

This “ioctl” is used by the user space application to place a video buffer in the video buffer queue. The application has to specify the buffer type (“V4L2_BUF_TYPE_VIDEO_ CAPTURE”), buffer index, and memory type (“V4L2_MEMOR”) at the time of queuing. The driver adds the buffer at the tail of the video buffer queue. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
struct v4l2_buffer buf;

buf.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.type  =  V4L2_MEMORY_MMAP;
buf.index  =  0;

ret =  ioctl(fd,  VIDIOC_QBUF,  &buf);
if(ret)  {
   perror("VIDIOC_QBUF\n");
   close(fd);
   return -1;
}
```

DeQueue a video buffer (VIDIOC_DQBUF)

This “ioctl” is used by the user space application to dequeue a video buffer from the video buffer queue. The application has to specify the buffer type (“V4L2_BUF_TYPE_VIDEO_CAPTURE”), and memory type (“V4L2_MEMORY_MMAP”) at the time of dequeuing. The driver provides the latest video buffer processed at its end. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
struct v4l2_buffer buf;

buf.type  =  V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.type  =  V4L2_MEMORY_MMAP;
buf.index  =  0;

ret =  ioctl(fd,  VIDIOC_DQBUF,  &buf);
if(ret)  {
   perror("VIDIOC_DQBUF\n");
   close(fd);
   return -1;
}
```

Stream on (VIDIOC_STREAMON)

This “ioctl” is used by the user space application to start video capture functionality. If streaming is already started, this ioctl call returns an error. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
int ret;

ret =  ioctl(fd,  VIDIOC_STREAMON,  NULL);
if(ret)  {
   perror("VIDIOC_STREAMON  \n");
   close(fd);
   return -1;
}
```

Stream off (VIDIOC_STREAMOFF)

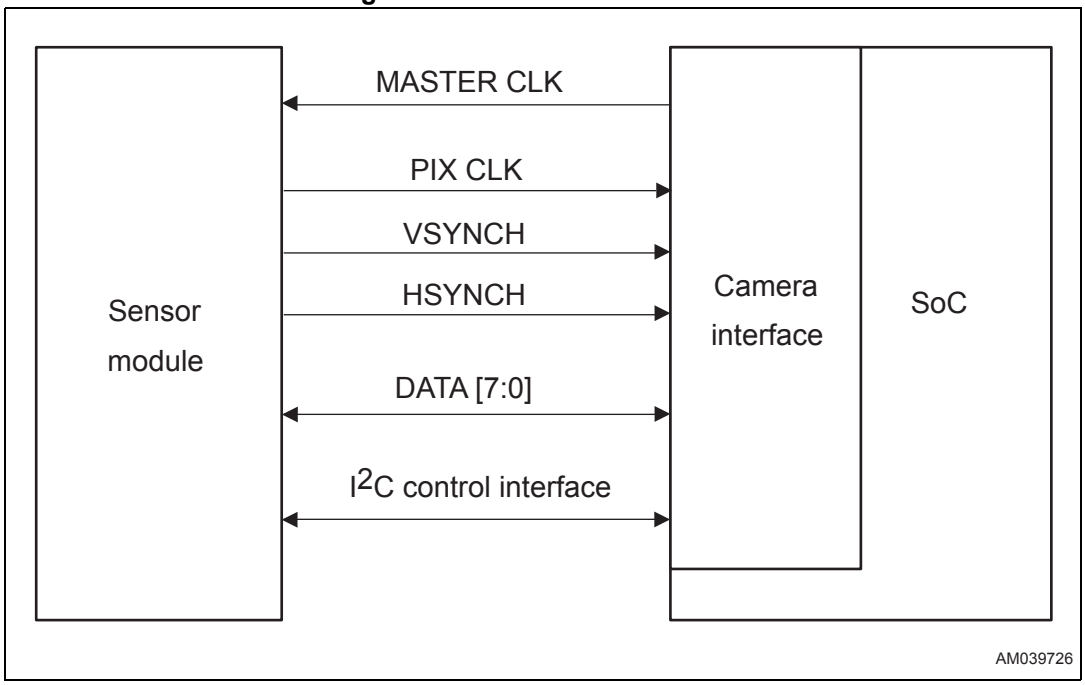
This “ioctl” is used by the user space application to stop video capture functionality. If streaming is not yet started, this ioctl call returns an error. Details of this ioctl and mechanism to use the same are depicted in the snippet below:

```
int ret;

ret =  ioctl(fd,  VIDIOC_STREAMOFF,  NULL);
if(ret)  {
   perror("VIDIOC_STREAMOFF  \n");
   close(fd);
   return -1;
}
```

10.2 SoC-Camera framework

Usually the external on-board sensor is connected to the SoC via I^2C bus which acts as the control path whereas an 8-bit parallel interface between the sensor and SoC is used as the data transfer path. There are also some additional signals like a pixel clock, HSYNC and VSYNC which are used to signify valid data on the data bus. *Figure 21* illustrates such a connection.

Figure 21. SoC-Camera interface



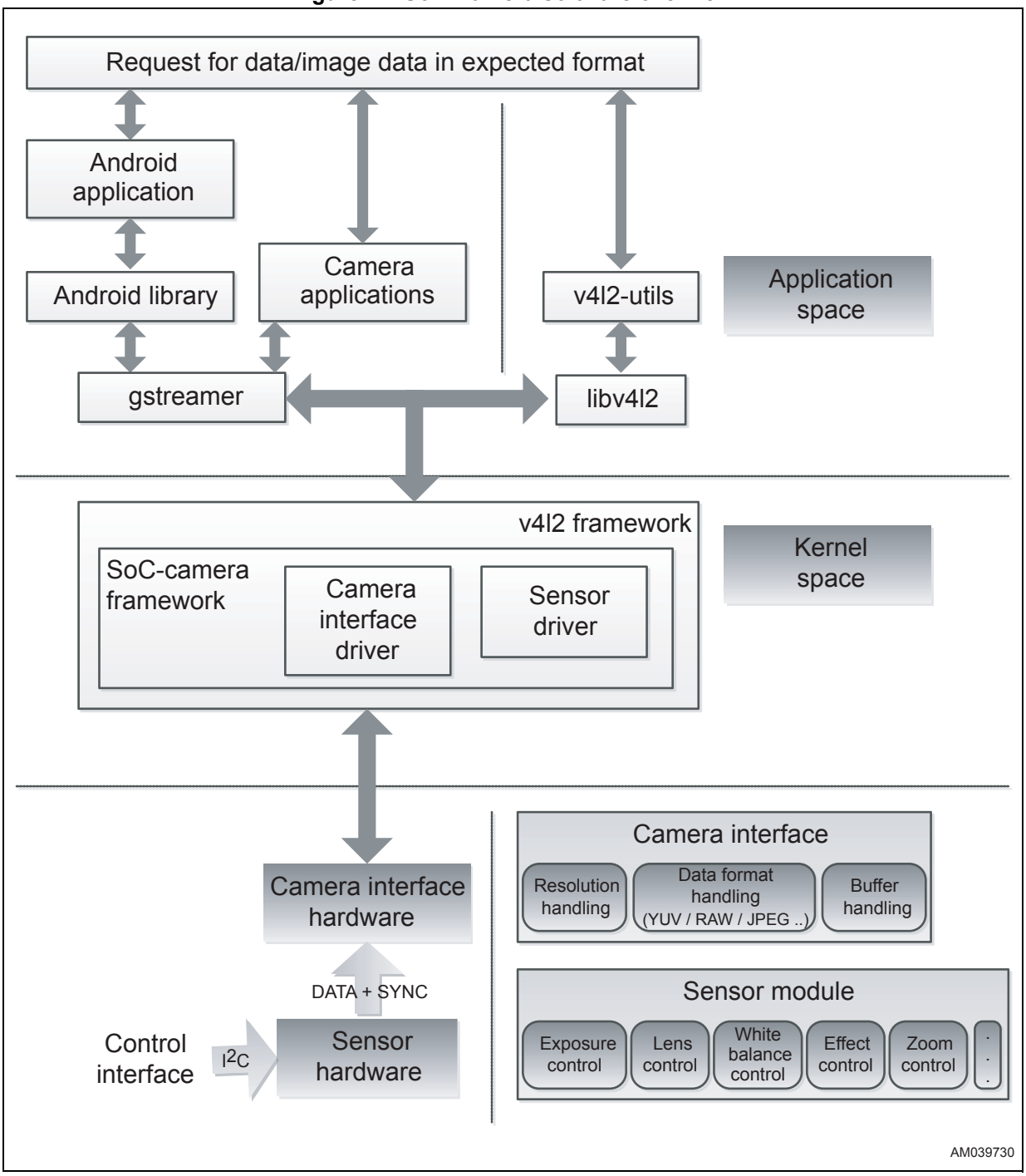
For a camera connection like the one described above, Linux provides a framework called “Video for Linux 2” (V4L2) which supports a wide variety of video capture devices. Using the V4L2 framework a user space application can access and configure a video capture device. However, due to the broadness of the V4L2 framework, a new framework called “SoC-Camera” framework was devised which is a subset of the V4L2 and provides a unified API between camera host drivers and camera sensor drivers. It implements a V4L2 interface to the user (currently only the “mmap” method is supported).

This subsystem has been written to connect drivers for system-on-chip (SoC) video capture interfaces with drivers for CMOS camera sensor chips to enable the reuse of sensor drivers with various hosts. The subsystem has been designed to support multiple camera host interfaces and multiple cameras per interface, although most applications have only one camera sensor.

For the details of the SoC-Camera framework and how user space applications use the same please refer to Linux documentation at “Documentation/video4linux/soc-camera.txt”.

Figure 22 depicts the SoC-Camera subsystem data/control flows:

Figure 22. SoC-Camera software overview



The STreamPlug camera is only supposed to handle one camera on its transport stream (TS) interface.

10.2.1 Camera interface

The camera interface module acts a simple video capture device when interfaced with an external image sensor.

Camera host API

A host camera driver is registered using the following function:

```
soc_camera_host_register(struct  soc_camera_host *);

where the parameter may be a struct like the following:

static struct soc_camera_host streamplug_soc_camera_host  =  {
   .drv_name  =  STREAMPLUG_CAM_DRV_NAME,
   .ops  =  &streamplug_soc_camera_host_ops,
};
```

All camera host methods are passed in a struct soc_camera_host_ops:

```
static struct soc_camera_host_ops streamplug_soc_camera_host_ops  =  {
   .owner=  THIS_MODULE,
   .add=  streamplug_camera_add_device,
   .remove=  streamplug_camera_remove_device,
   .suspend=  streamplug_camera_suspend,
   .resume=  streamplug_camera_resume,
   .set_crop=  streamplug_camera_set_crop,
   .get_formats=  streamplug_camera_get_formats,
   .put_formats=  streamplug_camera_put_formats,
   .set_fmt=  streamplug_camera_set_fmt,
   .try_fmt=  streamplug_camera_try_fmt,
   .init_videobuf=  streamplug_camera_init_videobuf,
   .reqbufs=  streamplug_camera_reqbufs,
   .poll=  streamplug_camera_poll,
   .querycap=  streamplug_camera_querycap,
   .set_bus_param=  streamplug_camera_set_bus_param,
};
```

where:

- ".add" and ".remove" methods are called when a sensor is attached to or detached from the host, apart from performing host internal tasks they shall also call sensor driver's ".init" and ".release" methods respectively.
- ".suspend" and ".resume" methods implement host's power management functionality and it's their responsibility to call respective sensor's methods.
- ".try_bus_param" and ".set_bus_param" are used to negotiate physical connection parameters between the host and the sensor.
- ".init_videobuf" is called by SoC-Camera the core when a video device is opened.

Further video-buffer management is implemented completely by the specific camera host driver. The rest of the methods are called from respective V4L2 operations.

Camera API

Sensor drivers can use “struct soc_camera_link”, typically provided by the platform and used to specify to which camera host bus the sensor is connected and provide platform “.power” and “.reset” methods for the camera. The “soc_camera_device_register()” and “soc_camera_device_unregister()” functions are used to add a sensor driver to or remove one from the system. The registration function takes a pointer to “struct soc_camera_device” as the only parameter. This struct can be initialized, for example, as follows:

```
/* link  to driver operations */
icd->ops =  &mt9m001_ops;
/* link  to the underlying physical (e.g.,  i2c) device */
icd->control =  &client->dev;
/* window  geometry */
icd->x_min =  20;
icd->y_min =  12; icd->x_current =  20;
icd->y_current =  12; icd->width_min =  48;
icd->width_max =  1280; icd->height_min =  32; icd->height_max =  1024;
icd->y_skip_top =  1;
/* camera  bus  ID,  typically obtained from platform  data */
icd->iface =  icl->bus_id;
```

The struct “soc_camera_ops” provides “.probe” and “.remove” methods, which are called by the SoC-Camera core, when a camera is matched against or removed from a camera host bus, “.init”, “.release”, “.suspend”, and “.resume” are called from the camera host driver as discussed above. Other members of this struct provide respective V4L2 functionality.

The “struct soc_camera_device” also links to an array of “struct soc_camera_data_format”, listing pixel formats, supported by the camera.

10.2.2 V4L2 subdev API

Camera drivers are interfaced to the SoC-Camera core and to host drivers over the V4L2-subdev API, but do not return any results. Therefore all camera drivers shall reply to “.g_fmt()” requests with their current output geometry. This is necessary to correctly configure the camera bus. The “.s_fmt()” and “.try_fmt()” drivers have to be also implemented. The sensor window and scaling factors have to be maintained by camera drivers internally. According to the V4L2 API all capture drivers must support the “VIDIOC_CROPCAP ioctl”, hence we rely on camera drivers implementing “.cropcap()”. If the camera driver does not support cropping, it may choose to not implement “.s_crop()”, but to enable cropping support by the camera host driver at least the “.g_crop” method must be implemented.

User window geometry is kept in “.user_width” and “.user_height” fields in “struct soc_camera_device” and used by the SoC-Camera core and host drivers. The core updates these fields upon successful completion of a “.s_fmt()” call, but if these fields change elsewhere, e.g.: during “.s_crop()” processing, the host driver is responsible for updating them.

10.3 Video transport stream (TS)

The video transport stream (TS) is a byte-wide parallel port that provides the I/O interface to peripheral devices. This interface is typically used to interface to external devices that either generate or input 8-bit data streams, such as MPEG encoders and decoders. The interface can be configured as an input or as an output. The interface can either be the master of the TS clock, or it can use an externally generated clock for the bus. The TS interface is a unidirectional interface.

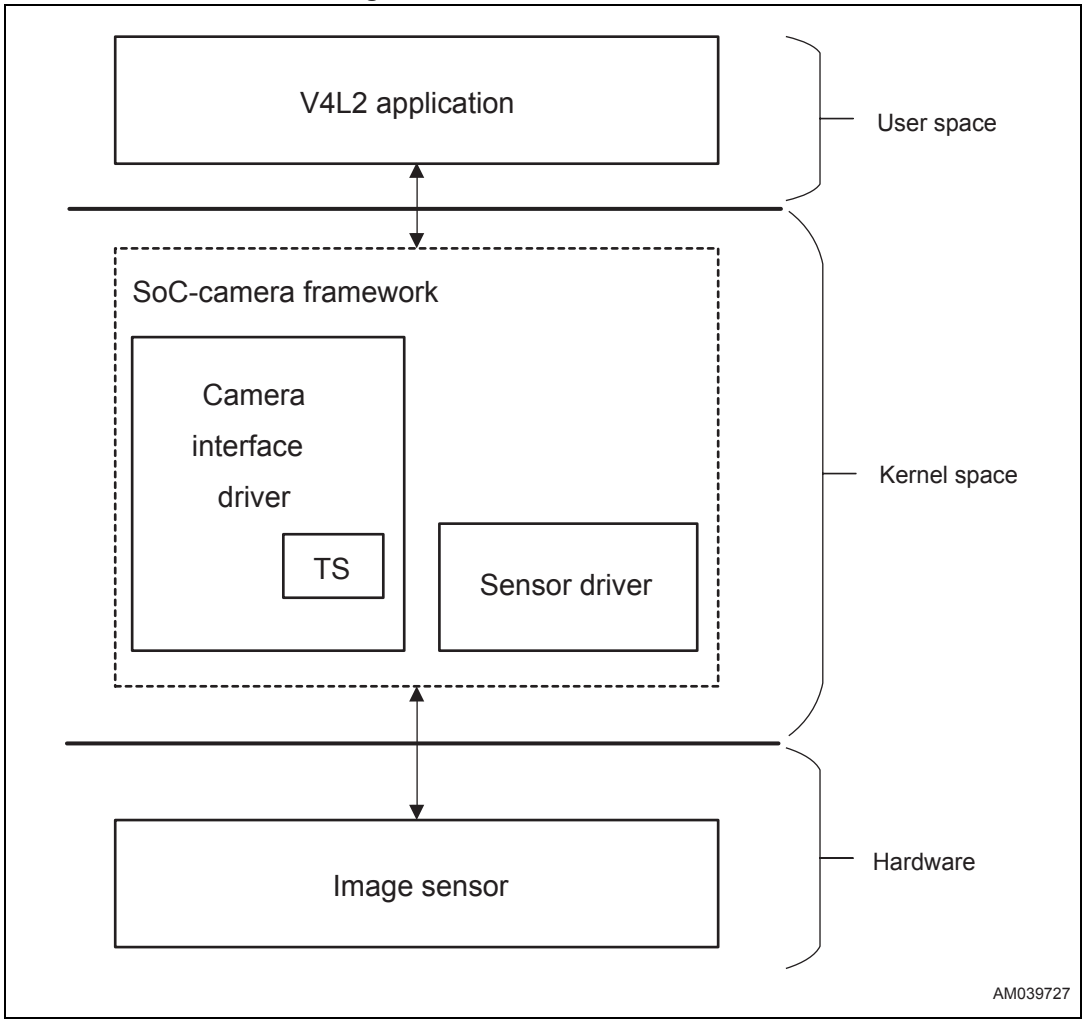
As a transmitter the TS interface can control the flow of data. As a receiver the TS interface has no flow control. The receiver must be prepared to receive all data with no method of slowing down the interface or recapturing missed data.

In the STreamPlug the TS has been used to transfer the data from the image sensor to the V4L2 framework.

10.3.1 TS software overview

Figure 23 illustrates how the TS driver is embedded into the camera interface to transfer data from the image sensor to the video for the Linux two framework (V4L2).

Figure 23. TS software overview



10.3.2 TS kernel source and configuration

The most important files containing the source code for the TS device driver are following:

```
arch/arm/plat-streamplug/misc.c
arch/arm/mach-streamplug/ipswrst_ctrl.c
arch/arm/mach-streamplug/include/mach/generic.h
arch/arm/mach-streamplug/include/mach/streamplug10.h
arch/arm/mach-streamplug/clock.c
arch/arm/mach-streamplug/padmux.c
arch/arm/mach-streamplug/streamplug1x.c
drivers/media/video/streamplug_camera.c
drivers/media/video/hm1355.c
include/media/hm1355.h
include/linux/streamplug_ts_reg.h
```

The “HM1355” is an image sensor used only for a demonstration purpose.

The Linux kernel must be built with the options listed in *Table 35* in order to enable the support for the V4L2, the SoC-Camera and the TS device driver.

Table 35. TS Linux kernel configuration options

Configuration	Description
CONFIG_MEDIA_SUPPORT	Enable framework for media support
CONFIG_VIDEO_DEV	Enable video devices
CONFIG_VIDEO_V4L2_COMMON	Enable common V4L2 support
CONFIG_VIDEO_MEDIA	Enable video media
CONFIG_VIDEO_V4L2	Enable V4L2
CONFIG_VIDEOBUF_GEN	Enable video buffer
CONFIG_VIDEOBUF_DMA_CONTIG	Enable video buffer with DMA
CONFIG_VIDEO_CAPTURE_DRIVERS	Enable video capture device drivers
CONFIG_VIDEO_ADV_DEBUG	
CONFIG_VIDEO_HELPER_CHIPS_AUTO	
CONFIG_VIDEO_IR_I2C	Enable I^2C for video devices
CONFIG_SOC_CAMERA	Enable SoC-Camera framework
CONFIG_SOC_CAMERA_HM1355	
CONFIG_VIDEO_STREAMPLUG	Enable STreamPlug specific video support
CONFIG_V4L_USB_DRIVERS	
CONFIG_USB_VIDEO_CLASS	
CONFIG_USB_VIDEO_CLASS_INPUT_EVDEV	

10.3.3 TS platform configuration

The platform data associated to the TS are in “arch/arm/mach-streamplug/streamplug1x.c”.

```
/* camera  interface 0  device registration  */
static int soc_camera_set_bus_param(struct  soc_camera_link *link,
   unsigned long  flags)
{
   return 0;
}

static unsigned long  soc_camera_query_bus_param(struct  soc_camera_link
*link)
{
   return 0;
}

static  void soc_camera_free_bus(struct  soc_camera_link *link)
{
}

static struct i2c_board_info soc_camera_i2c[]  =  {
   {
   I2C_BOARD_INFO("hm1355",  0x24),
   },
};

static struct streamplug_camera_pdata  camera_pdata =  {
   .flags=  HM1355_FLAG_VFLIP | HM1355_FLAG_HFLIP | \ HM1355_FLAG_8BIT,
   .mclk_10khz  =  HM1355_MCLK_12MHZ,
   .pclko_10khz  =  HM1355_PCLKO_18MHZ
};

static struct soc_camera_link iclink[]  =  {
{
   .bus_id=  0, /* Must  match  with the camera  ID  */
   .board_info=  &soc_camera_i2c[0],
   .i2c_adapter_id=  0,
   .query_bus_param=  soc_camera_query_bus_param,
   .set_bus_param=  soc_camera_set_bus_param,
   .free_bus=  soc_camera_free_bus,
   .module_name=  "hm1355",
   .priv=  &camera_pdata,
   },
};

static struct platform_device soc_camera[]  =  {
```

```
    {
        .name=  "soc-camera-pdrv",
        .id  =  0,
        .dev=  {
              .platform_data  =  &iclink[0],
        },
    },
};

static struct resource streamplug_camera_resources[]  =  {
{
    .start  =  STREAMPLUG1X_ICM2_TS_BASE,
    .end  =  STREAMPLUG1X_ICM2_TS_BASE  +  SZ_4K - 1,
    .flags  =  IORESOURCE_MEM,
    },
    {
    .name=  "ts_irq",
    .start=  STREAMPLUG1X_IRQ_APP_SUBS_TS_SPORT,
    .flags=  IORESOURCE_IRQ,
    }
};

/* camera  interface 0  device registeration  */
struct platform_device  streamplug1x_ts_device =  {
    .name  =  "streamplug-ts",
    .id  =  0,
    .dev  =  {
        .coherent_dma_mask  =  ~0,
        .platform_data  =  &camera_pdata,
    },
    .num_resources  =  ARRAY_SIZE(streamplug_camera_resources),
    .resource  =  streamplug_camera_resources,
};
```

10.3.4 TS usage

The TS device driver can be tested using a user space application like GStreamer to capture data from, for example, the image capture sensor peripheral. In Linux, such type of peripherals are accessed through the SoC-Camera framework. As the SoC-Camera framework is a subset of the V4L2 which provides a unified API between camera host drivers and camera sensor drivers, applications that are written using standard the V4L2 APIs and IOCTLs can directly work with the drivers written in the SoC-Camera framework, with the only limitation being that only the MMAP I/O method can be used. For details of the V4L2 framework and how to write user space applications to access video capture devices using the V4L2, please refer to previous sections.

To enable the TS support at run-time it is necessary to configure the Linux kernel command line using one of the options listed in *Table 4 on page 22*.

Once the Linux kernel has finished the startup phase, some parameters of the V4L2 framework can be configured using the “v4l2-dbg” utilities provided with the root filesystem. Then the GStreamer tool can be found either in the auxiliary filesystem or in an external one like a USB key or an SATA disk.

The following are the steps necessary to capture a stream of images from the image sensor:

1. Mount the USB device (“mount /dev/sdX /mnt”).
2. Export the following environment variables:

```
export PATH=:$PATH:/mnt/<gstreamer  folder>/usr/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/mnt/<gstreamer  folder>/usr/lib
export GST_PLUGIN_PATH=/mnt/<gstreamer  folder>/usr/lib:/mnt/<gstreamer
folder>/usr/lib/  ?'
gstreamer-0.10
export GST_PLUGIN_SCANNER=/mnt/<gstreamer  folder>/usr/libexec/gstreamer-
0.10/gst-plugin-  ?'
scanner
```

3. [Optional] mount the SATA device if the data have to be stored on the SATA disk.
4. Configure the image sensor. These values set the minimum delay between 2 successive frames according to the target device:

```
v4l2-dbg  --set-register=16  0x01 v4l2-dbg  --set-register=17  0x4F
```

Such values must be chosen depending on the peripheral used to save the image data stream. *Table 36* lists some of them.

Table 36. Image sensor delay parameter

Frame size	DDR	USB	SATA
VGA (648 x 488)	0x014F	0x08FF	0x18FF
FULL FRAME (1022 x 728)	0x014F	0x18FF	0x18FF
FULL FRAME (1022 x 808)	0x014F	0x18FF	0x18FF
FULL FRAME (1022 x 1032)	0x014F	0x18FF	0x28FF

Run the GStreamer application in the background mode using one of the following command:

```
- Data stream stored on DDR:
gst-launch v4l2src device=/dev/video0 num-buffers=100 ! 'video/x-raw-yuv,width=648,height=488, framerate=20/1' ! filesink
location=/tmp/<FileName>.raw &

- Data stream stored on USB device:
gst-launch v4l2src device=/dev/video0 num-buffers=100 ! 'video/x-raw-yuv,width=648,height=488, framerate=20/1' ! filesink
location=/mnt/<FileName>.raw &

- Data stream stored on SATA:
gst-launch v4l2src device=/dev/video0 num-buffers=100 ! 'video/x-raw-yuv,width=648,height=488, framerate=20/1' ! filesink
location=/sata/<FileName>.raw &
```

11 Virtualized devices

The “Kernel Support Package” (KSP) is a software layer which allows the virtualized Linux kernel to access through an SoC interface the hardware components (timers, interrupts, etc.) and it is active only in the full configuration.

The KSP is composed by five interface classes:

- Boot interface which provides the entry point of a boot image, performs CPU and platform initializations before starting the kernel.
- System interface which provides generic functionalities for cache management, idle handling and so on.
- Interrupt controller which provides interrupt decoding and management using the platform interrupt controller. The KSP is required to decode incoming interrupts, manage masking and unmasking of interrupts, and maintain interrupt registration data on behalf of the kernel.
- Device drivers which provided device drivers needed by the kernel (serial console, timer, VIC, etc.)
- Platform extensions.

Access to lower hardware configuration, or the shared information between different OSs is guaranteed by some APIs, provided by the SoC interface.

The KSP agent API is an object that mediates access to the KSP procedure call “syscall”. It provides:

- A unique user identifier to enable secure identification of the caller
- A virtual interrupt line which supports signaling events from the SoC interface to the Linux kernel interface
- Support for reading a memory space of Linux kernel shared with the SoC interface.

The KSP agent interface is primarily intended for exporting services which otherwise would be difficult or impossible to implement in a guest OS.

The following paragraphs describe a list of device drivers “virtualized” that take advantage of the KSP agent API interface to access to the hardware and/or to share resources with other embedded systems.

- The KSP interface controller is a device driver that handles the accesses of the various read/write to lower layers and/or the signaling coming from abstract layers to be dispatched to the target device.
- The “Misc” manager is not a properly device driver. In order to configure the systems during the machine initialization phase, the accesses to basic miscellaneous registers are performed by calling read/write APIs provided by the KSP interface controller.
- Virtual log (“Vlog”) is a virtual console to the RTOS.
- Virtual Flash controllers are the device drivers that perform the SMI and FSMC NAND Flash controllers, were updated in order to support a mechanism of accesses to the memories shared with the RTOS, regulated by KSP interface call procedures.
- Virtual Ethernet (“VEth”), a device driver that exposes the HPAV modem to the Linux kernel as an Ethernet device, in order to integrate it in the Linux standard network stack.
- “Image Validate” device (“imageval”) is a device driver, it is used to notify to the RTOS that the image is valid or not during the boot sequence. It is possible invalidate the firmware image in each time.

11.1 KSP interface controller

In order to support the KSP agent at the Linux kernel, a device driver was implemented at machine levels. It handles the signaling coming from the KSP interface and provides some primitives used by the virtualized drivers to access to lower layers than the Linux kernel.

11.1.1 KSP software overview

The software implementing the KSP interface controller is in:

`arch/arm/mach-streamplug/streamplug_ksp_agent.c.`

In order to register the KSP interface to the STreamPlug machine:

`arch/arm/mach-streamplug/streamplu1x.c`

the *streamplug_ksp_intf_registration* is called during machine initialization.

During the probe sequence, the KSP interface controller tries to:

- Allocate space for its own data structures:

```
struct streamplug_ksp_dev {
 struct  platform_device *pdev;
 /* spin lock */
 spinlock_t lock;
};

kspdev =  kzalloc(sizeof(*kspdev),  GFP_KERNEL);
```

- Recover the KSP agent unique identifier from platform data, provided by the KSP interface at start-up. The KSP agent is the key to access to the lower KSP interface.

```
ksp_agent = (struct okl4_tag_ksp_agent *)pdev->dev.platform_data;
```

- Register the virtual interrupt line assigned by the KSP interface, in order to handle the signaling coming from below

```
err = request_irq(ksp_agent->virq, streamplug_ksp_agent_int_handler, 0,
pdev->name, kspdev) ;
```

where the “streamplug_ksp_agent_int_handler” is the IRQ handler specific for the KSP agent. When interrupt arises, the handler will have to ask to the SoC interface for the payload that includes the information of signaling in order to dispatch it to the correspondent supported virtualized device drivers.

```
static irqreturn_t streamplug_ksp_agent_int_handler(int irq, void
*dev_id)
{
   struct streamplug_ksp_dev *dev = dev_id;
   okl4_channel_secondary_status_t payload = 0;
   unsigned int i;
   unsigned long flags;

   spin_lock_irqsave(&dev->lock, flags);
```

A Linux kernel panic error is generated, during the startup sequence, in case of misalignment of the version number between the KSP interface and the RTOS.

```
payload = streamplug_ksp_get_irq_payload();
printk(KERN_DEBUG "%s: payload %x", func , (u32)payload);
if (!payload){
  printk(KERN_DEBUG "%s: Failed collect irq action ret=%x", func ,
(int)payload);
   } else {
      if(payload & (1 << (STREAMPLUG1X_VIRQ_SHUTDOWN -
STREAMPLUG1X_VIRQ_BASE)))
      {
         printk(KERN_WARNING "The system is going down NOW!");

         /* Send signals to every process _except_ pid 1 */
         sys_kill(-1, SIGTERM);
         printk(KERN_WARNING "Sent SIG%s to all processes", "TERM");

         sys_kill(-1, SIGKILL);
         printk(KERN_WARNING "Sent SIG%s to all processes", "KILL");

         // power off
         Kernel_power_off();
      }
      for (i=0; i<STREAMPLUG1X_NUM_DEV_VIRQS; i++) {
         if (payload & (1 << i)){
```

```
            printk(KERN_DEBUG  "%s:  irq bit %d,  virq %d",     func    , i,
(int)( ?'
             STREAMPLUG1X_VIRQ_BASE  +  i));
             generic_handle_irq((STREAMPLUG1X_VIRQ_BASE  +  i));
          }
       }
    }

    spin_unlock_irqrestore(&dev->lock,  flags);

    return IRQ_HANDLED;
}
```

where “STREAMPLUG1X_VIRQ_BASE” is the base of the virtual interrupt lines list assigned statically to each virtualized device driver (Vlog, Virtual Flash Controller, VEth). The definitions are in the “arch/arm/mach-streamplug/include/mach/irqs.h” and their assignment in the platform data structures of virtualised device drivers.

Below is a list of functions implemented in order to send commands to the KSP interface. Each command is sent to lower layers using the API provided by the KSP agent “_okl4_sys_ksp_procedure_call”. The list of commands supported are in the “arch/arm/mac-streamplug/include/mach/streamplug_ksp_agent.h”.

- “streamplug_ksp_get_irq_payload” performs the “GET_IRQ_PAYLOAD” in order to know which interrupt lines have to be dispatched at the kernel layer. Used by the KSP interface controller driver.
- “streamplug_ksp_vlog_open” performs the “VLOG_OPEN” command in order to notify to the RTOS the open of the Vlog Read/Write procedure. It is used by the Vlog device driver.
- “streamplug_ksp_vlog_close” performs the “VLOG_CLOSE” command in order to notify to the RTOS the termination of Read/Write procedures. It is used by the Vlog device driver.
- “streamplug_ksp_vlog_read” performs the “VLOG READ” command in order to read a character from the remote buffer. It is used by the VLog device driver.
- “streamplug_ksp_vlog_write” performs the “VLOG WRITE” command in order to write a character into the remote buffer. It is used by the VLog device driver.
- “streamplug_ksp_misc_read_reg” performs the “MISC_REG_READ” command in order to obtain the value of a certain “Misc” register of which offset is passed to the KSP procedure call. It is used by “Misc”.
- “streamplug_ksp_misc_read_reg” performs the “MISC_REG_WRITE” command in order to set a “Misc” register of which offset is passed to the KSP procedure call. It is used by “Misc”.
- “streamplug_ksp_flash_mutex_handler” performs the request/release of “Mutex” by Linux kernel in order to access to Flash memories when they are shared with remote embedded systems. Used by Flash controller drivers.
- “streamplug_ksp_veth_generic_cmd” - it's a generic function that performs all the commands towards the KSP interface for the Virtual Ethernet device driver.

11.1.2 KSP kernel source and configuration

Table 37 lists the kernel configuration options associated with the KSP interface controller.

Table 37. KSP agent controller configurations

Configuration	Description
CONFIG_KSP_AGENT_ENABLED	It enables support of the KSP interface controller

11.1.3 KSP platform configuration

The platform data associated to the KSP interface controller are in “arch/arm/mach-streamplug/streamplug1x.c”.

```
struct  platform_device streamplug1x_ksp_agent_device =  {
   .name  =  "okl4-ksp-agent",
   .id  =  -1,
   .dev  =  {
      .platform_data  =  &okl4_ksp_agent,
   },
};
```

where “okl4_ksp_agent” is a global structure that provides the capabilities of the KSP agent associated to the Linux kernel.

11.2 Miscellaneous register access (Misc)

According to the peripheral configuration specified in the Linux kernel command line a set of miscellaneous registers have to be accessed during the startup procedure. The access is not performed directly into the miscellaneous register memory space, but through the KSP agent interface.

Misc software overview

During the machine initialization phase every time an access to a miscellaneous register is required, the correspondent function is called. The functions are provided by the KSP interface controller into “arch/arm/mach-streamplug/streamplug_ksp_agent.c”. For writing procedures use:

```
void streamplug_ksp_misc_write_reg(unsigned  int value,  unsigned int *reg)
```

and for reading procedures use:

```
unsigned int streamplug_ksp_misc_read_reg(unsigned  int *reg).
```

11.3 Virtual log

The “Virtual log” (VLog) is a device driver that virtualizes the UART interface of the RTOS.

11.3.1 Virtual log software overview

VLog is a simple char driver, implemented below the “drivers/char/streamplug_vlog.c” file.

During the probe phase, it will have to register two different IRQ handlers, read and write, each of them is associated to one interrupt line statically assigned by its own platform data. Both of them will be notified to the VLog device driver by the KSP agent that will dispatch the interrupt request to the correspondent handler according to the payload it recovers from the lower KSP module.

READ

The procedure with which a user space application can ask to the Vlog driver to read is:

```
ssize_t dev_read(struct file *fil, char *buff, size_t len, loff_t *off)
{
   int size = 0;
   int i = 0;
   int left = 0;

   pr_debug("%s enter\n",__func__);
   pr_debug("Read flag: %x \n", read_flag);
   spin_lock(&sdev->lock);
   if(read_flag)
   {

      read_flag = 0;
      spin_unlock(&sdev->lock);
      size = streamplug_ksp_vlog_read(buff, len, &left);
      pr_debug("bytes left %d \n", left);

      // if there are bytes into the buffer, set the flag again
      if(left)
         read_flag = 1;

   }
   else
      spin_unlock(&sdev->lock);
   pr_debug("%s exit\n",__func__);

   return (ssize_t)size;
}
```

An interrupt for the reading procedure arises when at least one character is available in the buffer of the RTOS.

The VLog driver may access to the buffer where data are stored until this interrupt is detected. Each read request coming from the user space is rejected with 0 bytes returned. After its detection, at the first request from the user space, the VLog driver recovers char by char from the buffer and forwards them to the user space application that will display them in “stdout”. The buffer is handled by the soc module of the KSP interface, so the Vlog driver will have to call the “streamplug_ksp_vlog_read” provided by the KSP agent controller, in order to access it.

The KSP interface for the VLOG READ byte command will return the number of characters still available within the remote buffer. The procedure will continue until no data will be available within the buffer and VLOG driver will send a 0 to the user space.

WRITE

The procedure with which a user space application can ask to the Vlog driver to write is:

```
ssize_t dev_write(struct file *fil, const char *buff, size_t len, loff_t
*off)
{
   ssize_t ret = 0;
   int size = (unsigned int)len;

   pr_debug("%s enter\n",__func__);
   if(write_flag)
   {

      // there isn't enough space, stop to write
      if(!streamplug_ksp_vlog_write(buff,size))
         write_flag = 0;

   }
   pr_debug("write_flag %x", write_flag);
   pr_debug("%s exit\n",__func__);
   return len;
}
```

Every time the VLog driver receives a write request from user space applications, it will forward the character passed from “stdin” to the remote buffer calling the procedure provided by the KSP interface controller “streamplug_ksp_vlog_write”. The KSP interface for the VLOG WRITE byte command will return the number of bytes the remote buffer may still accept. When the buffer is full, no more data have to be sent by the VLog driver that will have to wait for arising the interrupt for “Write” procedure dispatched by the KSP interface controller. In that case the VLog driver may continue to send data to the remote buffer.

11.3.2 Virtual log kernel source and configuration

Table 38 lists the kernel configuration options associated with the KSP interface controller.

Table 38. Virtual log configurations

Configuration	Description
CONFIG_STREAMPLUG_VLOG	It enables the virtual log driver support

11.3.3 Virtual log platform configuration

The platform data associated to the KSP interface controller are in “arch/arm/streamplug1x.c”.

```
/* vlog device registration */
static struct resource vlog_resources[] = {
   {
      .start = STREAMPLUG1X_VIRQ_VLOG_TO_RTOS,
      .flags = IORESOURCE_IRQ,
   }, {
      .start = STREAMPLUG1X_VIRQ_VLOG_FROM_RTOS,
      .flags = IORESOURCE_IRQ,
   }
};

struct platform_device streamplug_vlog_device = {
   .name = "streamplug-vlog",
   .id = 0,
   .num_resources = ARRAY_SIZE(vlog_resources),
   .resource = vlog_resources,
};
struct platform_device  streamplug_vlog_device =  {
   .name  =  "streamplug-vlog",
   .id  =  0,
   .num_resources  =  ARRAY_SIZE(vlog_resources),
   .resource  =  vlog_resources,
};
```

11.3.4 Virtual log usage

The steps to test the VLog device drivers are:

1. Verify if the device is correctly registered:

```
$ cat /proc/devices | grep vlog
```

2. Create the correspondent node:

```
$ mknod /dev/vlog c 'cat /proc/devices | grep vlog | awk '{print $1}'' 1
```

3. To run use the utilities provided by the Linux kernel, and wait for characters in “stdin” in the background:

```
$ tail -f /dev/vlog &
```

4. Send characters into “stdout”:

```
$ echo "<option>" > /dev/vlog
```

In addition, a small application named “stpconsole” is available in the example set and in the default root filesystem to expose the RTOS console in Linux. This application permits the access to the RTOS interface when the UARTs are not available because needed by the application.

11.4 SMI/FSMC NAND memory shared access

The Flash controllers device drivers (SMI and FSMC NAND) have been modified in order to support the shared access by two different software components such, for example, the Linux kernel and the RTOS.

The Flash controllers allow the shared access through the use of a Mutex mechanism. This means that whenever a component accesses the Flash, it first has to obtain the corresponding Mutex. If this is not possible then it has to wait until the Mutex is freed from the other component.

The Flash controller has a virtual interrupt line assigned to, in order to be notified through the KSP interface controller that one of the components need to access to the Flash.

In the STreamPlug the Flash controllers have been modified to allow the shared access to the corresponding Flashes by the Linux kernel and the RTOS. However, only the Linux kernel Flash controller get notified of the access request by the RTOS.

11.4.1 SMI/FSMC NAND software overview

In order to be notified by the KSP agent interface driver to locked/unlocked the Flash memory by remote systems, the SMI/FSMC NAND Flash controllers have to register their own virtual interrupt lines assigned statically and stored into platform data. For both controllers, the two IRQ handlers associated will have to handle the state of memory (reserved to Linux kernel or RTOS), and the state of the driver when interrupt is detected.

The IRQs handling is the same for both SMI and FSMC NAND Flash controllers.

Flash memory status

As default, the Flash memory is reserved to the Linux kernel system. When the request interrupt arrives, the state of memory changes from reserved to RTOS. When the release interrupt occurs, the state comes back to reserved to Linux. Different actions are taken according to the current state of the driver such as: if there are no accesses ongoing, the state of the controller is IDLE and in case of a request from the ROTS state the Mutex is

immediately released, while in case of release indication the procedure to acquire the Mutex is started. If there is an access in memory ongoing, nothing is performed until the procedure ends, but internal flags are updated to force Flash memory area status change and to force the Mutex release of the request.

If a Mutex request or Mutex release is already ongoing, nothing is performed until the procedure ends, but internal flags are updated to force Flash memory area status change and to force the Mutex release of the request.

Mutex request/release

When the Flash memory status is reserved to the Linux kernel, no accesses to the KSP interface are performed in order to acquire/release the Mutex. This is done by calling the “streamplug_ksp_flash_mutex_handler” with the appropriate command provided by the KSP interface controller.

In case of Flash memory status is RTOS, the Flash controller has always to wait for a release indication and acquire the Mutex before the access to the memory.

11.4.2 SMI/FSMC NAND kernel source and configuration

The support of the virtual interrupt handler by the SMI/FSMC NAND is subordinated by enabling the “CONFIG_KSP_AGENT_ENABLED” option.

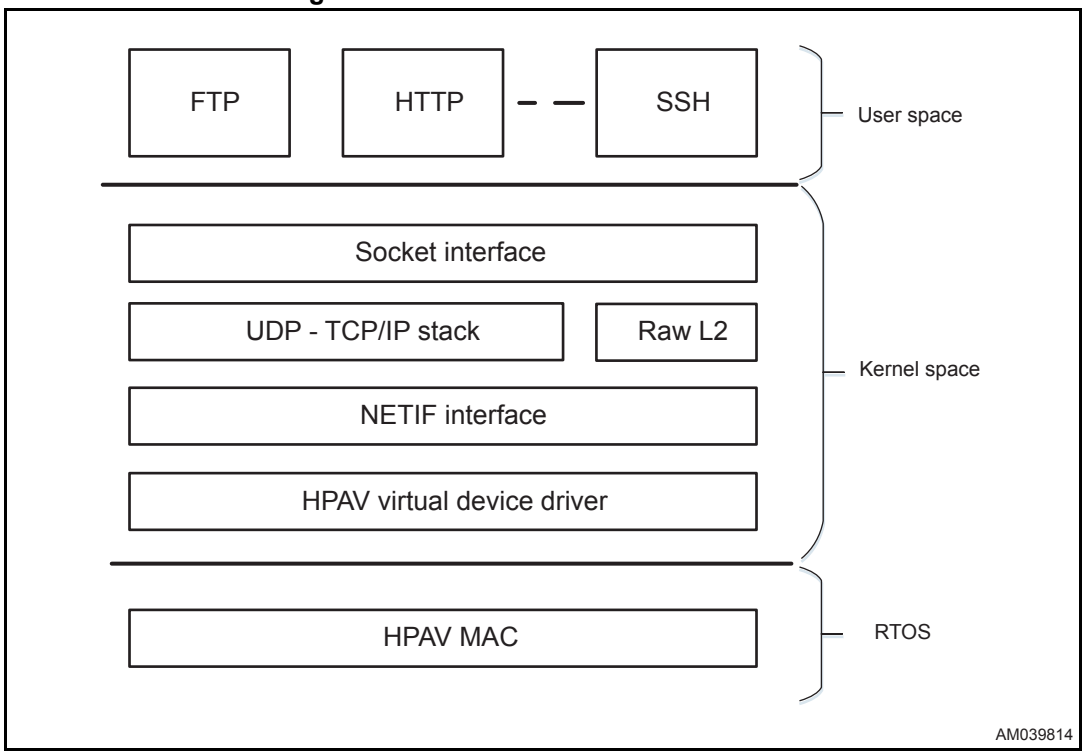
11.4.3 SMI/FSMC NAND platform configuration

The new interrupt lines were added and statically assigned to the resources of both SMI and FSMC NAND.

11.5 HomePlug AV (HPAV) driver

The HomePlug AV (HPAV) driver is a virtualized device driver that exposes the HPAV modem to the Linux kernel as an Ethernet device, in order to integrate it in the Linux standard network stack, as illustrated in *Figure 24*.

Figure 24. HPAV stack software overview



11.5.1 HPAV software overview

The structure of the HomePlug AV device driver reflects the generic Ethernet device driver structure. Linux supports some standard “ioctl” commands to configure network devices at the user space level. In the future more “ioctl” commands will be provided to extend existing features.

The main difference is the new interaction with the KSP interface at the kernel space level. The KSP interface interacts with the HPAV device driver via:

- Interrupts
- Commands

In both cases, there's a strict interaction with the KSP agent interface controller that provides the capabilities to send commands and the dispatcher to notify the signaling via interrupt routine of messages receptions.

During the probe phase, all the platform data of KSP info are recovered in order to register the interrupt to the KSP interface and acquire a shared memory buffer for memory operations. As for the other virtualized device driver, the HPAV has to register its own virtual interrupt, the line of which is passed through platform data by the machine definition.

This is performed during the device open function shown below:

```
...
set_irq_chip_and_handler_name(dev->irq,  &stmmac_ksp_chip,
stmmacvirt_interrupt,  "vstmmac");
set_irq_flags(dev->irq,  (IRQF_VALID|IRQF_SHARED));
set_irq_chip_data(dev->irq,  dev);
...
```

where the “stmmacvirt_interrupt” is the virtual IRQ handler:

```
static irqreturn_t stmmacvirt_interrupt(int  irq,  struct  irq_desc *desc)
{

   struct net_device *dev = desc->chip_data;
   struct stmmac_virt_priv *priv = netdev_priv(dev);

#if defined(STMMAC_VIRT_XMIT_DEBUG) || defined(STMMAC_VIRT_RX_DEBUG)
   printk(KERN_DEBUG "%s: interrupt occurred virq %d, desc %x", __func__,
irq, desc);
#endif
   if (unlikely(!dev)) {
      pr_err("%s: invalid dev pointer\n", __func__);
      return IRQ_NONE;
   }
   stmmacvirt_dma_interrupt(priv);

   return IRQ_HANDLED;
}
```

11.5.2 HPAV kernel source and configuration

Table 39 lists the kernel configuration options associated with the KSP interface controller.

Table 39. Virtual Ethernet configurations

Configuration	Description
CONFIG_STMMAC_VIRT_ETH	It enables the support of HomePlug AV driver

11.5.3 HPAV platform configuration

The platform data associated to HPAV are in “arch/arm/mach-streamplug/streamplug1x.c”.

```
static struct resource virteth_resources[]  =  {
   {
   .start  =  STREAMPLUG1X_VIRQ_ETH,
   .flags  =  IORESOURCE_IRQ,
   },
};
```

```
/* virtual phy  device */
static struct plat_stmmacvirtphy_data virtphy_private_data =  {
   .bus_id  =  0,
   .phy_addr  =  -1,
   .phy_mask  =  0,
   .interface  =  PHY_INTERFACE_MODE_MII,
};

struct platform_device  streamplug1x_virtphy_device =  {
   .name  =  "stmmacvirtphy",
   .id  =  -1,
   .dev.platform_data  =  &virtphy_private_data,
};
/* Virtual  ethernet device registration  */
struct plat_stmmacenet_data virtether_platform_data =  {
   .bus_id  =  0,
   .has_revmii  =  0,
   .has_gmac  =  0,
   .enh_desc  =  0,
   .pbl  =  8,
   .dev_addr  =  "01:80:e1:26:0a:5b",
};

static u64  virteth_dma_mask =  ~(u32)  0;

struct platform_device streamplug1x_virteth_device =  {
   .name  =  "stmmacvirteth",
   .id  =  -1,
   .num_resources  =  ARRAY_SIZE(virteth_resources),
   .resource  =  virteth_resources,
   .dev  =  {
      .platform_data  =  &virtether_platform_data,
      .dma_mask  =  &virteth_dma_mask,
      .coherent_dma_mask  =  ~0,
   },
};
```

11.6 Image validate device driver

Linux notifies to RTOS that the image is valid or not during the boot sequence and specifies the kernel version. It is also possible to set a user application version. Both versions are 16 bytes long.

11.6.1 Image validate device driver software overview

Image validity is a simple char driver, implemented below the ”drivers/char/streamplug_image_validity.c” file.

WRITE

During the write operation a KSP agent is sent to set the user version.

The procedure with which a user space application can ask to the image validity driver to write is:

```
ssize_t dev_iv_write(struct file *fil, const char *buff, size_t len, loff_t
*off)
{
   if (!buff)
      return 0;

   if (len > USER_IMAGE_VER_SIZE)
      len = USER_IMAGE_VER_SIZE;

   memcpy(idev->version, buff, len);
   if (!streamplug_ksp_set_image_version_handler(idev->version, len))
      return len;
   else
      return 0;
}
```

READ

The procedure with which a user space application can ask to the image validity driver to read is:

```
ssize_t dev_iv_read(struct file *fil, char *buff, size_t len, loff_t *off)
{

   if (len > USER_IMAGE_VER_SIZE)
      len = USER_IMAGE_VER_SIZE;
   memcpy(buff, idev->version, len);

   return (ssize_t)len;
}
```

This operation permits to know the value of the image version and it is used to check the result of write procedure.

IOCTL

This function is used to perform two operations: “SET_IMAGE_STATUS” and “GET_IMAGE_STATUS”.

During the “SET_IMAGE_STATUS” operation a KSP agent is sent to notify to RTOS the validity of the image (0 and 1 are the only values allowed, 0 for an invalid image and 1 for a valid one). The image value is invalid by default and if the driver receives a new request to update a valid value, it issues the KSP, but if the value is already set valid, the driver doesn't issue the KSP agent. The “SET_IMAGE_STATUS” operation is a call at least of the Linux initialization process (at the end of “/etc/inittab”).

The “GET_IMAGE_STATUS” operation permits to know the value of image validity and it is used to check the result of the “SET_IMAGE_STATUS” operation.

```
static long dev_iv_ioctl(struct file* fil, unsigned int cmd, unsigned long
arg)
{
   long ret=0;
   u32  status = arg;

   switch (cmd) {

      case SET_IMAGE_STATUS:
         if ((status == IMAGE_GOOD) || (status == IMAGE_NO_GOOD))
         {
            if (idev->value != status)
            {
               printk(KERN_DEBUG "%s  value = %u\n",__func__, status);
               idev->value = status;
               ret = streamplug_ksp_set_image_validity_handler(status);
            }
         }
         else
            ret = -EINVAL;
         break;
      case GET_IMAGE_STATUS:
         printk(KERN_DEBUG "%s value is %d\n",__func__, idev->value);
         ret = (long) idev->value;
         break;

      default:
         ret = -EINVAL;
   }
   return ret;
}
```

11.6.2 Image validate device driver kernel source and configuration

Table 40 lists the kernel configuration options associated with the KSP interface controller.

Table 40. Image validity configuration

Configuration	Description
CONFIG_STREAMPLUG_IMAGE_VALIDITY	It enables the image validity driver support
CONFIG_EARLY_SET_IMAGE_GOOD	It enables the early set image good

When the “CONFIG_EARLY_SET_IMAGE_GOOD” is enabled the drivers are compatible with the old file systems. The kernel calls the KSP agent to notify to RTOS that the image is valid. The image value is good by default and if the driver receives some new request returns always success.

11.6.3 Image validate device driver platform configuration

The platform data associated to the KSP interface controller are in “arch/arm/mach-streamplug/streamplug1x.c”.

```
/* image validity device registration */
struct platform_device streamplug_iv_device = {
   .name = "streamplug-image-validity",
   .id = 0,
}
```

11.6.4 Image validate device driver usage

The steps to test the image validity device drivers are:

1. Verify if the device is correctly registered:

```
$ cat /proc/devices | grep imageval
```

2. Create the correspondent node, if it does not exist:

```
$ mknod /dev/imageval c 'cat /proc/devices | grep imageval | awk
'{print $1}'' 1
```

In addition, a small application named “stpimagevalidate” is available in the example set and in the default root file system to be used after the Linux initialization or to invalidate the firmware image. The option of this application are:

- Usage: stpimagevalidate [options]
- “*-i* “ sets the image as invalid
- “*-v* “sets the image as valid
- *“-n <version>”* sets the image version
- *“-s”* shows image information
- “*-h”* displays this usage information.

Appendix A Acronyms

Table 41 contains a list of acronyms used within the document.

Table 41. List of acronyms

Acronym	Definition
AMBA	Advanced microcontroller bus architecture
ADC	Analog-to-digital converter
ARM	Advanced RISC machine
AVB	Audio Video Bridging
BSP	Board support package
C3	Channel controller coprocessor
CAN	Controller area network
CPU	Central processing unit
DDR	Double data rate SDRAM
DHCP	Dynamic host configuration protocol
DMA	Direct memory access
DWC	Designware cores
EEPROM	Electrically erasable programmable read only memory
EP	PCIe endpoint device
FS	File system
FSMC	Flexible static memory controller
FW	Firmware
GPIO	General purpose input/output signal
HID	Human interface device
I^2C	Inter-integrated circuit
I^2S	Inter-IC sound
IP	Internet protocol
IPs	Intellectual Properties
JFFS2	Journaling Flash file system version 2
JPEG	Joint photographic experts group
JTAG	Joint test action group
KSP	Kernel support packages provided by OKL Microvisor
MFIO	Multifunction input/output signal
MTD	Memory technology device
NFS	Network file system
OOB	Out-of-band

Table 41. List of acronyms (continued)

Acronym	Definition
PC	Personal computer
PCIe	Peripheral component interconnect express
RevMII	Reverse media independent interface
RC	PCI-e root complex device
RTC	Real-time clock
SATA	Serial advanced technology attachment
SCP	Secure copy Linux command
SDRAM	Synchronous dynamic random access memory
SDK	OKL4 SDK
SMI	Serial Management Interface
SoC	System-on-chip
SOC	Definition of board the chip is mounted on
SPI	Serial Peripheral Interface bus
SPORT	Serial port
TAG	Tagged List
UART	Universal asynchronous receiver/transmitter
USB	Universal serial bus
VIC	Vectored interrupt controller

Revision history

Table 42. Document revision history

Date	Revision	Changes
26-Nov-2015	1	Initial release.