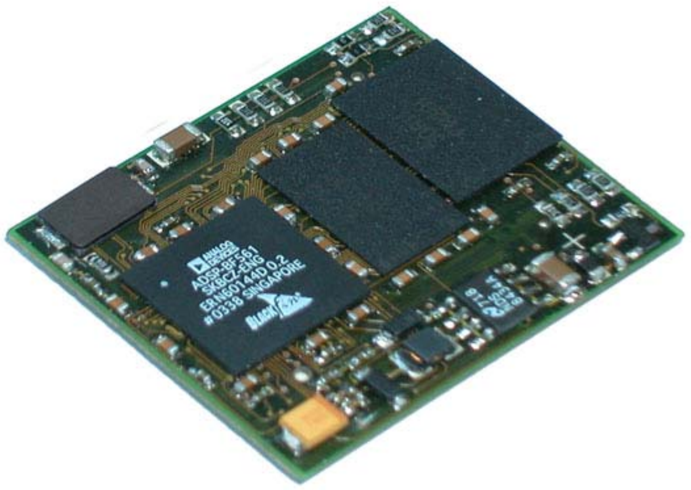

BLACKSheep V0.5.x

Software Manual

www.tinyboards.com

Maximum Power at Minimum Size

Contact

Bluetechnix Mechatronische Systeme GmbH
Waidhausenstr. 3/19
A-1140 Vienna
AUSTRIA/EUROPE
office@bluetechnix.at
http://www.bluetechnix.at
http://www.tinyboards.com

Version 2.2

2005-01-10

Document No. 099-0001-02

Table of Contents

Information
For further information on technology, delivery terms and conditions and prices please contact Bluetechnix (http://www.bluetechnix.at).

Warnings
Due to technical requirements components may contain dangerous substances

The Core Boards and Development systems contain ESD (electrostatic discharge) sensitive devices. Electro-static charges readily accumulate on the human body and equipment and can discharge without detection. Permanent damage may occur on devices subjected to high-energy discharges. Proper ESD precautions are recommended to avoid performance degradation or loss of functionality. Unused core boards and development boards should be stored in the protective shipping package.

1 Introduction

The VDK BLACKSheep is a multithreaded framework for the Analog Devices Blackfin processor family that includes driver support for a various types of hardware extensions. It bases on the VDK micro kernel included within the VDSP++ development environment. It also contains a full featured TCP/IP stack based on the "lwip" stack including the standard berkley socket API. It is in continuous development, and you can found the newest version on the "Bluetechnix" website (http://www.tinyboards.com).

1.1 Overview

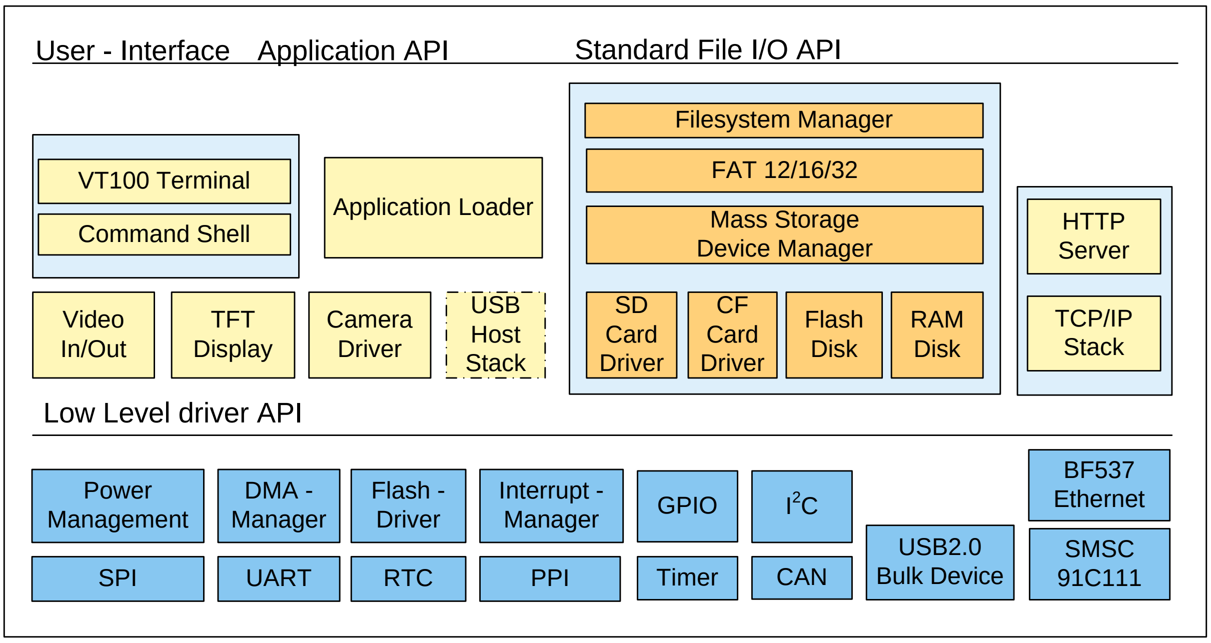

Figure 1.1: Overview of the BLACKSheep framework

1.2 BLACKSheep Components

The BLACKSheep software package includes driver for

- SPI
- PPI
- UART
- GPIO
- Timer
- Real Time Clock
- I^{2}C
- Video input/output
- ethernet chip SMSC91C1111
- internal MAC of the BF537
- SD Card
- Compact Flash Card

- Ram Drive
- FAT 12/16/32 file system
- Omnivision camera modules
- Hitachi color display
- USB 2.0 device
- AD1836 audio codec

With the package you get the Go Ahead web server port for the BLACKSheep. In future versions we plan to add also support for ftp and email.

1.3 Target Environment

The BLACKSheep was developed for the Blackfin core modules CM-BF533, CM-BF537E/U, TCM-BF537 and CM-BF561 provided by Bluetechnix. Some parts of the BLACKSheep code uses peripheral interfaces located on the DEV-BF5xx and EVAL-BF5xx boards also provided by Bluetechnix.

1.4 Application

You can use the drivers and other packages from the BLACKSheep within your own application. The application can be a VDK kernel project as well as a single threaded stand alone application without the need of any VDK specific function call. (All low level drivers works also in a single threaded stand alone application, they don't need VDK functionality. The TCP/IP stack needs a multithreaded environment.)

It is also possible to use the BLACKSheep as a framework where you can embed the functionality needed by your application.

1.5 Portability

If you want to port the VDK BLACKSheep drivers to other multithreaded operating system environments you have to provide support for semaphores and protection of critical sections. Replace the semaphore calls and calls to adi_entercriticalsection and adi_exitcriticalsection with your environment specific function calls for protecting critical sections. The most part of the low level drivers does not need operating specific calls, so they are easerly portable to other operating systems.

1.6 Development Environment

The entire BLACKSheep was developed with the VDSP++4.0 development environment provided by Analog Devices. It is recommended to use the VDSP++ to include BLACKSheep code into your own applications, but the BLACKSheep is written in ANSI C, so it is possible to compile the code also on other platforms. Currently a successful compilation of the BLACKSheep code can only be guaranteed on VDSP++ 4.0.

1.7 Documentation

Bluetechnix does not provide documentation for the lwip stack, the go ahead web server port or the VDK kernel and services API. For documentation regarding this software packages please contact Analog Devices or the appropriate software developer or refer to the official developers websites. (For the VDK kernel and services: www.analog.com and for lwip: http://www.sics.se/~adam/lwip/).
Bluetechnix only supports the BLACKSheep specific modifications and needed initialisation calls for this packages.

This document gives an overview of the BLACKSheep drivers and functionality. Refer also to the appropriate doxygen documentation to get more detailed information about the source code of each module.

2 Module Specification

2.1 Overview

Figure 1.1 shows the basic architecture of the BLACKSheep software and how the packages interact.
The lowest level driver acts as a hardware abstraction layer and provides a common interface to the upper layers that not depends on the Blackfin processor type. Not all drivers use the same interface. The APIs depends on the driver type.

2.2 Low Level Driver

The low level driver uses some managers from the system services provided by Analog Devices. They came with the VDSP++ development environment. So you have to initialize the proper manager prior to use any of the low level drivers.
The system services used by the BLACKSheep are:

- interrupt manager
- dma manager
- power management service

Refer to the “Device_Drivers_and_System_Services_Manual.pdf” to learn more about the ADI system services.

2.2.1 Initialisation of the system services

The following section shows how to initialize the system services to work correctly with the BLACKSheep driver framework.

Interrupt manager:

The following call is needed for a correct initialisation of the ADI interrupt manager

```
static u8 intmgr_storage[(ADI_INT_SECONDARY_MEMORY * 8)];
unsigned long response_count;
unsigned long   critical_reg;

// initialize the interrupt manager
adi_int_Init( intmgr_storage, sizeof(intmgr_storage), &response_count, &critical_reg);
```

The following functions of the ADI interrupt service are used by the VDK BLACKSheep drivers:

adi_int_Init
adi_int_CECHook
adi_int_SICWakeup

adi_int_SICEnable
adi_int_SICGetIVG
adi_int_Close

DMA manager:

The DMA manager does not need any special general initialisation. The initialisation for each required channel is done by the driver itself.

Power management service:

The BLACKSheep uses the power management service from Analog Devices to set the optimal core and system clock frequency.

Initialize example:

```
ADI_PWR_COMMAND_PAIR PowerTable [] = {
{ADI_PWR_CMD_SET_PROC_VARIANT, (void*)ADI_PWR_PROC_BF537SKBC600 },
{ ADI_PWR_CMD_SET_PACKAGE, (void*)ADI_PWR_PACKAGE_LQFP },
{ ADI_PWR_CMD_SET_VDDEXT,  (void*)ADI_PWR_VDDEXT_330 },
{ ADI_PWR_CMD_SET_CLKIN, (void*)CLKIN },
{ ADI_PWR_CMD_END, NULL},
};

// initialize power management
result = adi_pwr_Init(PowerTable);
if (result != ADI_DEV_RESULT_SUCCESS) {
        return result;
}

//set optimal frequencies
result = adi_pwr_SetFreq(0, 0, ADI_PWR_DF_ON);
if (result != ADI_DEV_RESULT_SUCCESS) {
        return result;
}
```

Each driver is implemented as single threaded version as well as reentrable version to be used in a VDK kernel environment. To enable the modifications needed by the VDK you have to define the **_USE_VDK_** macro within your project settings. If enabled, some drivers need support for semaphores so enable the semaphores within the project kernel settings.

2.2.2 Processor specific dependencies

Each driver that uses processor specific registers or hardware specific entities is divided into to parts, the processor independent and the processor dependent part. The files associated to the processor dependent part are named according to the following naming convention: **_global.h and *_global.c*. This files are located in the *driver/src/bf5xx* directory of the BLACKSheep source tree. All processor independent files are located in the

driver/src/common directory. To realize a project including a low level driver is sufficient to add the processor independent file and the appropriate **_global* file of the processor you use to your project and the driver should compile with the correct settings for your target core processor.
In some cases the code is gated by pre-processor macros to ensure that the code will be compiled with the correct settings for the processor you use. The VDSP++ automatically defines a pre-processor macro named *__ADSPBF5xx__* depending on the target processor. If you use another IDE to compile the BLACKSheep code you have to define such pre-processor macro by yourself.

The processor specific files fills some global structures containing the addresses of the registers. The processor independent part of the driver just refers to this structures to access the registers. No address of any register is hardcoded within the processor indipendent part.

2.2.3 GPIO

Files associated: *gpio_global.c, gpio_global.h, GPIOconfig.c, GPIOconfig.h*

Directories: *driver/src/common, driver/src/bfxxx*

Dependencies: interrupt manager

The GPIO driver allows to set a gpio as input or output, to set and clear the GPIO and to read in the voltage level (high or low). It provides also a function to set a GPIO as interrupt source, with the possibility to call a personal callback function if an interrupt occurs.

Example:

Sets the GPIO PF13 as output and to logical level high.

```
gpio_becomeOutput (_PF13);
gpio_set (_PF13);
```

GPIO expander:

If you want to be able to use other GPIOs than that of the Blackfin you have the possibility to add GPIO expander to the GPIO driver. The GPIO expander driver has to implement the following 5 functions: *set, clear, becomeInput, becomeOutput*, and *readFlag*.

The following initialisation sequence adds a GPIO expander driver to the generic GPIO driver:

```
gpio_AddIOExpander (     BANK-ID,
                         becomeInput,
                         becomeOutput,
                         set,
                         clear,
                         toggle,
                         readFlag,
```

0);

BANK-ID is a free selectable ID greather then 0x20. The BANK-ID must be unique for each GPIO expander added to the driver.

2.2.4 Timer

Files associated: *gpTimerConfig.c, gpTimerConfig.h, timer_global.c*

Directories: *driver/src/common, driver/src/bfxxx*

Dependencies: interrupt manager

The timer driver allows to configure each timer by setting its config-, period- and with-register. Each timer can generate an interrupt with the possibility to call a user specified callback function.

Functions:

```
T_GP_TIMER_INST *timer_gp_setup (    unsigned short pa_nTimerNr,
                                     unsigned short pa_nConfig,
                                     unsigned long pa_nWidth,
                                     unsigned long pa_nPeriod,
                                     bool bPositivePulse,        //not used anymore
                                     unsigned long pa_nSystemClk,
                                     T_TIMER_CALLBACK pa_fnCallback)
```

The setup function returns a handler that identifies the timer. Each function call that follows has to use this handle for referencing the timer.

```
void timer_enable (T_GP_TIMER_INST *pTimerHndl)
void timer_disable (T_GP_TIMER_INST *pTimerHndl)
void timer_set_period (T_GP_TIMER_INST *pTimerHndl, unsigned long pa_nValue)
void timer_set_width (T_GP_TIMER_INST *pTimerHndl, unsigned long pa_nValue)
```

2.2.5 UART

Files associated: *UARTconfig.c, UARTconfig.h, uart_global.c, uart_global.h*

Directories: *driver/src/common, driver/src/bfxxx*

Dependencies: interrupt manager

The UART driver is implemented as a buffered input, output character device. The buffer size can be determined by setting the appropriate parameter in the “uart_open” function. If you use a processor with more than one UART like the BF537, you can open each UART with a specific baudrate, parity, number of stop bits, number of data bits and input and output buffer size.

The buffers are allocated dynamically in the heap section true a „malloc“ call, so be sure to have enough space in the heap section defined by your linker description file. If the memory allocation failes the uart_open function returns zero.

The first call of uart_putChar writes directly to the hardware transmit register to initiate the transfer, until the UART is transmitting bytes each further call of uart_putChar writes to the transmit buffer.

Error conditions:

If uart_open failes the function returns zero.
A call to uart_putChar with a full transmit buffer results in an error returned by the function.

If there are hardware errors on the UART interface or the UART receives data when the transmit buffer is full, the UART interrupt handler calls a common error function.

Initialisation example:

The following example shows a correct initialisation for the fisrt UART with 57600 baud, no parity, 1 stopbits, 8 databits:

```
UARTs_init ();
T_UART_HANDLE hUART = uart_open(0, sclk, 1024, 1024, 0);
uart_setMode(hUART, 57600, UART_NONE, 8, 1);
```

For transfering data you can use the following functions:
uartPutchar : writes a byte to the uart
uartGetchar: gets a byte from uart buffer
uartWritestring: writes a null terminated string to the uart

2.2.6 SPI

Files associated: *SPIsimpleio.c, SPIsimpleio.h, spi_global.c, spi_global.h*

Directories: *driver/src/common, driver/src/bfxxx*

Dependencies: none

Currently the SPi interface driver is implemented as a simple input output system without buffer handling. That means that writes and reads to or from the SPI goes directly to the appropriate hardware register. The SPI driver supports a transfer size of 8 or 16 bits, selectable by a parameter of the „spi_setup“ function.
Currently only the master mode is supported. A buffered version of the driver, supporting the slave mode will be released soon.

The handling of the SPISSEL signal, the slave select signal to determ witch slave is enabled to receive or transmit data, has do be done by the upper level driver or the application, calling the “spi_select” and “spi_deselect” functions prior and after writing or reading in data.

By writing a word to the SPI there is automatically one byte ready for reading, because of the nature of the SPI communication. If this data is not read by a call of spiReaddata, the data will be overwritten on the next transfer.

Initialisation example

The initialisation of the SPI is done by a call to „spi_setup“ „spi_open“ for each connected slave device. In a future buffered version you can determ the transmit and receive buffer size as a parameter in the „spi_open“ function.
The following example shows a correct initialisation to setup the SPI as master, with a SCK frequency of 20Mhz and a maximum of two slaves with 8 bit data width.

```
spi_setup(2, 20000000, 0x0000, 0x0800, 0x0000, 0x0000, sclk);
```

2.2.7 PPI

Files associated: *PPIconfig.c, PPIconfig.h, ppi_global.c*

Directories: *driver/src/common, driver/src/bfxxx*

Dependencies: interrupt manager

The PPI driver provides functions to set the PPI and associated DMA registers. There are to functions to configure the PPI interface: *ppi_itu656_setup* and *ppi_gp_setup*.

ppi_itu656_setup:

Configures the PPI for ITU656 input mode.

ppi_gp_setup:

Allows to set each of the PPI and the associated DMA registers independently. That means that you are completely free in configuring the PPI by setting the appropriate values of the registers.

Example:

The example sets the ppi register for receiving an ITU-R656 4:2:2 stream with a resolution of 320 x 240 pixel and then waits until one frame is captured.

```
ppi_setup_itu656 (
                cPPI,
                PPI_SINGLE_SHOT,
                PPI_MEMORY_WRITE,
                (unsigned long)YUVbuffer,
                PPI_FRAMEWISE_INT,
                (T_PPI_CALLBACK)PPIHandler,
                1,
                2,
                PPI_DMA_16_BIT,
                PPI_RECEIVE_ACTIVE_ONLY,
                PPI_16BIT_PACKING,
                PPI_FIELD_BOTH,
                PPI_SKIP_NONE,
                nXsize,
                nYsize,
                0);
g_bCapturingFinished = false;
ppi_enable_itu656(cPPI);                          // enable the ppi
unsigned long nTimeout = 250;                     // 250 ms timeout
while (!g_bCapturingFinished && nTimeout--) {     // wait until captured
        Sleep (1);                                // wait for 1 ms
        }
ppi_disable_itu656 (cPPI);                        // disable the ppi
ppi_close (cPPI);                                 // close the ppi
```

The global variable *g_bCapturingFinished* is set by the callback function *PPIHandler*.

2.2.8 Flash

Files associated: *flash.c*, *flash.h*

Directories: *driver/src/common*

Dependencies: gpio driver

The files contain a series of functions to manipulate the flash mounted on the CM-BF5xx. The most important are routines for erasing sectors, erasing the entire device, programming a 16 bit value into the flash and checking if the entire flash is empty. It provides also a function for reading the flash.

To use the FLASH memory you have to enable the appropriate asynchronous memory bank. In case of the CM-BF561 this is only bank 0, already enabled at boot time. In case of a CM-BF533 these are the bank 0 and the bank 1 (bank 0 is enabled on boot time). Be aware to enable the banks for 16 bit accesses in case of the CM-BF561.

To specify the configuration of the flash you have to set some pre-processor macros so that the flash driver works correctly. The example shows the settings for the CM-BF533:

```
#define FLASH_INTEL_STRATA              // intel strata compatible flash
#define FLASH_START_ADDRESS             0x20000000
#define FLASH_NUM_SECTORS               16
#define FLASH_SECTOR_SIZE               0x20000
#define FLASH_NOF_BANKS                 1
#define FLASH_BANK_SIZE                 0x200000        //2Mb
#define FLASH_BANK_SELECT0              0
#define FLASH_BANK_SELECT1              0
#define FLASH_BANK_SELECT2              0
#define FLASH_BANK_SELECT3              0
```

The macros FLASH_BANK_SELECTx specifies witch GPIO signals are connected to the flash to access more than 2Mb of flash memory. For this purposes the GPIOs acts as bank select signals. The above macros are declaring a flash with a size of 2Mbytes with 16 sectors of 128kb each, without any bank switching.

2.2.9 I²C

Files associated: *i2c_framing_layer.c, i2c_framing_layer.h, i2c_hal.c, i2c_timing_layer.c, i2c_timing_layer.h, timers.c, timers.h*

Directories: *driver/src/common/i2c*

Dependencies: interrupt manager

The I²C driver implements an I²C compatible interface with either 100kbit or 400kbit of clock speed. If a BF537 processor is used the driver uses the on chip TWI unit to perform the I²C interface. On the other Blackfin processors which doesn't have a TWI interface in hardware the I²C protocol is emulated by to GPIOs, one for SCL and the other for SDA. Which GPIO is used for which signal can be specified at runtime by the appropriate parameter in the *I2CInit* function. For this reason the *I2CInit* function looks different for the BF537 and the other Blackfin processors:

- for the BF537:

I2CInit (unsigned long pa_lFrequency, unsigned long pa_lSystemClk)

- for other Blackfin processors without TWI interface:

I2CInit(unsigned long pa_lFrequency, unsigned char pa_SCLpin, unsigned char pa_SDApin, unsigned char pa_cTimer, 0)

The I²C driver can be compiled either as blocking or no blocking. If compiled as blocking the read and write functions waits until the operation completes. The blocking or no blocking option is selectable by the pre-processor macro I2C_BLOCKING.

2.2.10 RTC

Files associated: *extRTCconfig.c, extRTCconfig.h, intRTCconfig.c, intRTCconfig.h, RTCconfig.c, RTCconfig.h*

Directories: *driver/src/common/rtc*

Dependencies: none

On the BF533 and the BF537 the internal RTC is supported. On the BF561 currently no RTC is available. The driver provides functions to read and write the time and the date. The file RTCconfig.c includes the common interface to all RTCs. Depending if an internal or external

RTC is used the files *extRTCconfig* or *intRTCconfig* must be included in the project. Currently no external RTC is supported so the appropriate functions in *extRTCconfig* are empty. The choice if an internal or an external RTC is used currently is done at compile time by pre-processor macros. If an BF561 is used the compiler will automatically call the functions of the external RTC, if the other Blackfin types are used, the compiler uses the functions for the internal RTC. Prior to have access to the internal RTC they must be initlaized by calling the *int_rtc_Setup* function.

Prototype declarations:

int int_rtc_Setup (void);
*int int_rtc_getDate(unsigned char *pa_pnWeekDay, unsigned char *pa_pnDay, unsigned char *pa_pnMonth, unsigned short *pa_pnYear);*
int int_rtc_setDate(unsigned char pa_nDay, unsigned char pa_nMonth, unsigned short pa_nYear);
*int int_rtc_getTime(unsigned char *pa_pnHour, unsigned char *pa_pnMinute, unsigned char *pa_pnSecond);*
int int_rtc_setTime(unsigned char pa_nHour, unsigned char pa_nMinute, unsigned char pa_nSecond);

2.2.11 SCCB

Files associated: *sccbConfig.c, sccbConfig.h*

Directories: *driver/src/common*

Dependencies: gpio driver

The SCCB driver implements the 2-wire SCCB protocol used by the Omnivision camera modules to read and write the configuration register. The SCCB is similar to the I²C interface. The two signals *clock* and *data* are emulated with GPIOs. Therefore the SCCB driver needs the GPIO driver to work properly. The SCCB driver provides functions to open and close the SCCB interface as well as functions to read and write registers within an Omnivision camera chip with a certain device address.

The associated functions are:

```
T_ERROR_CODE sccb_open (      T_SCCB_HANDLE *pa_phSccb,
                              T_GPIO_MASK pa_sio_cMask,
                              T_GPIO_MASK pa_sio_dMask,
                              T_GPIO_MASK pa_pwdnMask,
                              unsigned char pa_cWriteAddr,
                              unsigned long pa_nCoreClk)

T_ERROR_CODE sccb_writeByte(  T_SCCB_HANDLE pa_hSccb,
                              unsigned char pa_cDevAddr,
                              unsigned char pa_cSubAddr,
                              unsigned char pa_cData)

unsigned short sccb_readByte( T_SCCB_HANDLE pa_hSccb,
```

unsigned char pa_cDevAddr,
unsigned char pa_cSubAddr,
*T_ERROR_CODE *pa_errCode)*

T_ERROR_CODE sccb_close(T_SCCB_HANDLE pa_hSccb)

The function sccb_open returns a handler *pa_hSccb* that identifies the SCCB interface. Each following call to the interface has to be identified by this handle.

2.2.12 Omnivision Camera module

<u>Files associated:</u> *CAMconfig.c, CAMconfig.h, CAMdefault.c, CAMdefault.h*

<u>Directories:</u> *driver/src/common*

<u>Dependencies:</u> SCCB driver

The Omnivision camera driver uses the SCCB driver to communicate with the camera module. There is one setup function that initializes the camera register according to the subaddress-value pairs defined in the *pa_pConfigParames* array. This array can be initialized manually or with predefined values from the *CAMdefault.c*. This file contains several functions that initialize this array depending on the required framerate, resolution and data format. Currently not for all possible setups default initialisation functions are provided, but the user is free to realize his oen array with register subadress-value pairs according to the initialisation he needs. Each entry in the array consists of an integer value with the following format: 0xSAVA where SA is the subaddress and VA the value written to the register.

The *cam_setup* function returns a handler that identifies the camera. He contains the device ID of the camera (0x7660 for OV7660 and 0x7648 for OV7648). Currently the modules OV7648 and OV7660 are supported. Each of them has a maximal resolution of VGA (640x480).

2.2.13 TFT-Display

<u>Files associated:</u> *TFTconfig.c, TFTconfig.h*

<u>Directories:</u> *driver/src/common/tft*

<u>Dependencies:</u> GPIO driver, PPI driver, Timer driver

The TFT-Display driver currently supports only the Hitachi TX09D50VM1CCA and compatible, but it can be adapted to other display types with similar interface with a few modifications.

The driver uses 4 timers to create the timing needed by the display. One timer just creates the 5,33Mhz pixel clock and can be omitted by using an external 5,33Mhz oscillator. One of the other timers creates the HSYNC signal and the other two the signal DTMG. Each timer outputs the waveform on his associated timer pin. For creating the correct DTMG timing the two pins of the associated timer has to be connected over an AND-gate. The output of the

AND-gate then provides the correct DTMG-timing. The timer used for the display timing can be specified at runtime with the appropriate parameters in the setup function.

Another timer is used to create the PWM-signal for the background led so it is possible to change the brightness of the display.

The other pins needed y the display as the PCI input are emulated by GPIOs.

The TFT-display driver initializes the PPI and DMA. Which PPI channel is used can be specified with a parameter of the setup function. Of course currently only using the BF561 a second PPI can be used. As the BF533 has not enough timer, the display driver doesn't support this processor type.

The user is responsible for allocating enough memory for the framebuffer. This memory is given to the display driver by the *nFrameBuffer* parameter. As the Hitachi display uses the RGB565 format and has a resolution of 320 x 240 a memory size of 320 x 240 x 2 = 153600 bytes is needed. The framebuffer can be either in landscape or in portrait format. If it is in landscape format the TFT_ROTATE_FRAME pre-processor macro has to be defined because the Hitachi display is in portrait format.

The *pa_bBitReverse* parameter specifies if the display data-pins are connected bit reversed or not.

Prototypes of the functions:

```
T_ERROR_CODE TFTsetup (        unsigned char pa_cPPI,
                               unsigned long pa_nCoreClk,
                               unsigned long pa_nSystemClk,
                               unsigned short pa_nXsize,
                               unsigned short pa_nYsize,
                               unsigned char pa_BytePerPixel,
                               unsigned char pa_cPWMtimer,
                               unsigned char pa_cDTMGshiftTimer,
                               bool pa_bCreateDCLK,
                               unsigned char pa_cDCLKtimer,
                               T_GPIO_MASK pa_cPCIflag,
                               unsigned char *nFrameBuffer,
                               bool pa_bBitReverse,
                               void *pa_Reserved);
void TFTsetBrightness (unsigned short pa_cBrightness);
void TFTsetPixel (unsigned short x, unsigned short y, unsigned char pa_cRed, unsigned char
pa_cGreen, unsigned char pa_cBlue);
void TFTgetPixel(unsigned short x, unsigned short y, unsigned char *pa_pcRed, unsigned
char *pa_pcGreen, unsigned char *pa_pcBlue);
void TFTblendPixel(unsigned short x, unsigned short y, unsigned char pa_cRed, unsigned
char pa_cGreen, unsigned char pa_cBlue, unsigned char pa_cAlpha);
unsigned long TFTgetFramebuffer (void);
```

Setup example:

```
// allocate memory for the display frame buffer
unsigned char *pcFrameBuffer = (unsigned char *)malloc
(XSIZE*(YSIZE+TFT_Y_BLANKING_LINES) * BYTES_PER_PIXEL);

// setup tft-display
T_ERROR_CODE erResult = TFTsetup (   PPI,
                                     CORE_CLK,
                                     SYSTEM_CLK,
                                     XSIZE,
                                     YSIZE,
                                     BYTES_PER_PIXEL,
                                     LED_PWM_TIMER,
                                     DTMG_SHIFT_TIMER,
                                     CREATE_DOT_CLOCK,
                                     DCLK_TIMER,
                                     PCI_FLAG,
                                     pcFrameBuffer,
                                     BIT_REVERSE,
                                     0);
```

2.2.14 SMSC Ethernet driver

Files associated: *eth_conf.h, io_sprt.h, lan91c111.c, lan91c111.h*

Directories: *driver/src/common/smsc*

Dependencies: none

The driver for the SMSC Ethernet chip is a port of the driver from Analog Devices, so refer to the appropriate documentation of Analog Devices.

2.2.15 Internal MAC of BF537

Files associated: *ADI_ETHER_BF537.c EMAC2_regbits.h*

Directories: *driver/src/bf537/mac*

Dependencies: interrupt manager

The driver for the internal MAC of the BF537 is a port of the driver from Analog Devices, so refer to the appropriate documentation of Analog Devices.

2.2.16 Video In, Video Out

Directories: *driver/src/common/analog_video*

Dependencies: PPI driver, DMA manager, Interrupt manager

Refer to the appropriate documentation in the *driver/src/common/analog_video/doc* directory.

2.2.17 USB 2.0 device

Directories: *driver/src/common/smsc*

Dependencies: interrupt manager, dma manager, GPIO driver

The BLACKSheep supports the NETPLX2272 USB device chip. The driver is a port from the Analog Devices driver that cames with the VDSP++ development environment. Please refer to the appropriate documentation from Analog Devices.

2.3 Device manager

Files associated: *MSDmanager.c, MSDmanager.h*

Directories: *BLACKSheep/common/msd*

The device manager manages the low level driver of all devices where file systems can be located like SD-card, CF-card and RAM-drive called block devices. The device manager also controls calls to character- or streaming devices.

The device manager allows to register or un-register block or streaming devices and to perform hardware independent read or write operations. The function *msd_info* provides information about the device.

Example:

The example registers an SD-card to the device manager.

```
// initializing  sd-card
T_SD_HANDLE hSDcard = sd_open (pa_cSDcardSlot);
// registering sd-card at the device manager
msd_register((void*)hSDcard, "sd0", 0, dev_sd_blockRead, dev_sd_blockWrite,
dev_sd_info);
```

Prior to using any functionality of the device manager he has to be initialized by calling the *msd_initManager* function.

2.4 File System

The files system driver hides the specific architecture of the device and file system management from the user. From a users point of view all devices have a common interface. The user can call standard io-functions as *fopen*, *fclose*, *fread*, *fwrite* to access the devices. Each devices is addressed by a path. The path consists of the partition name followed by a colon and the path in Microsoft Windows style.

Naming convention:

The style of the path naming is associated to Microsoft Windows. A possible path on the sd-card could be: *sd0:\samplepath1\samplepath2.*

2.4.1 File System Manager

Files associated: *fsmgr.c*, *fsmgr.h*

Directories: *BLACKSheep/common/fs*

The file system manager manages different file systems. Currently only a FAT driver is available. Each file-system has to be registered at the file system manager to be known at the system. The example shows how to register a file-system to the file system manager and how a partition is mounted into the system.

Example:

```
// registering the fat file system
fs_register("fat", fat_getFileFunctions() );

// mounting a sd-card partitionformattet with the FAT file system
// The partition name is sd0
fs_mountPartition("sd0", "sd0", "fat");
```

Prior to use any functionality of the file system manager he has to be initialized by calling the *fs_setup* function.

2.4.2 FAT driver

Directories: *BLACKSheep/common/fs, BLACKSheep/common/fs/fat*

The FAT driver is a full featured driver for FAT12/16/32 based on the Free DOS FAT driver. Please refer to the Free DOS FAT driver documentation.

The FAT driver supports long file names for FAT32 but also the long file name extension for FAT16.

2.5 Storage Devices (Block Devices)

2.5.1 RAM-drive Driver

Files associated: *RDconfig.c, RDconfig.h, devRD.c, devRD.h*

Directories: *BLACKSheep/common/msd*

The RAM-drive driver emulates a RAM-drive. The memory needed for the device is allocated from heap. So be sure to have enough space in heap. The driver implements the following functions:

Prototypes:

T_RD_HANDLE rd_open(unsigned long pa_nNofSectors);
*T_ERROR_CODE rd_readSingleBlock(T_RD_HANDLE pa_hRD, unsigned long pa_nBlockAddr, unsigned char *pa_cData);*
*T_ERROR_CODE rd_writeSingleBlock(T_RD_HANDLE pa_hRD, unsigned long pa_nBlockAddr, unsigned char *pa_cData);*
void rd_close(T_RD_HANDLE pa_hRD);

Interface to the device manager:

To provide a common interface to the device manager the file *devRD.c* contains the 3 common interface functions for the device manager:
*signed long dev_rd_blockRead(void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*signed long dev_rd_blockWrite(void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*T_ERROR_CODE dev_rd_info(void *pa_pDevHandle, T_DEVICE_INFO *pa_pstDeviceInfo);*

2.5.2 SD-Card Driver

Files associated: *SDcard.c, SDcard.h, devSD.c, devSD.h*

Directories: *BLACKSheep/common/msd*

The SD-card driver uses the SPI driver to communicate with the SD-card, so currently only the SPI mode of the SD-card is supported. The driver does all low level communication with the SD-card providing an interface to the user, so that the user can communicate with the SD-card. As the SD-card driver uses the SPI-interface be sure to have initialized the SPI-driver prior to call any function of the SD-card driver (see section 2.2.6). The following example shows how to initialize the SD-card:

Example:

T_SD_HANDLE hSDcard = sd_open (pa_cSDcardSlot);

The only parameter needed by the *sd_open* function is the SPI slave select signal which is connected to the SD-card. The function returns a handle which identifies the SD-card and is used by other further calls to the SD-card driver.

Interface to the device manager:

Similar to the RAM-drive driver also the SD-card driver has defined three functions as a common interface for the device manager. The functions from this interface are defined in the files *devSD.c*. The three methods are:

*signed long dev_sd_blockRead (void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*signed long dev_sd_blockWrite (void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*

*T_ERROR_CODE dev_sd_info (void *pa_pDevHandle, T_DEVICE_INFO *pa_pstDeviceInfo);*

As you can see the functions differs only in the function name. The parameters are the same.

2.5.3 CF Card Driver

Files associated: *IDEconfig.c, IDEconfig.h, devIDE.c, devIDE.h*

Directories: *BLACKSheep/common/msd*

The CF-card driver supports CF-cards in true IDE mode. Hence the files and functions associated with the CF-card have "ide" as prefix. For this reason the CF-card driver should also working with mass storage devices supporting the IDE-ATA mode as harddisks or similar devices. The example shows how to initialize the CF-card driver and how to open a device:

Prototypes:

```
T_IDE_HANDLE ide_open( T_ADDRESS pa_nSlotBaseAddress,
                       unsigned short pa_nAddressShift,
                       bool pa_bAsMaster,
                       T_GPIO_MASK pa_gpioResetFlag,
                       T_GPIO_MASK pa_gpioIntReq
                       );
```

pa_nSlotBaseAddress:	Baseaddress of the CF-card device
pa_nAddressShift:	Address multiplier, signals to the driver witch address pin is connected as word selector.
pa_bAsMaster:	Defines if device is master or slave
pa_gpioResetFlag:	GPIO flag which is connected to the RESET pin of the CF-card
pa_gpioIntReq:	GPIO flag which is connected to the nIRQ pin of the CF-card

Example:

```
T_IDE_HANDLE hIDE = ide_open(0x20200300,
                             4,
                              true,
                             _PF12,
                             _PF13);
```

Interface to the device manager:

For the interface to the device manager applies the same as for the other block devices. The fnctions are defined in *devIDE.c.* The prototypes are:

```
signed long dev_ide_blockRead(void *pa_pDevHandle, unsigned long pa_nBlockAddr,
                        unsigned long pa_nBlockCount, unsigned char *pa_pcData);
signed long dev_ide_blockWrite(void *pa_pDevHandle, unsigned long pa_nBlockAddr,
```

*unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*T_ERROR_CODE dev_ide_info(void *pa_pDevHandle, T_DEVICE_INFO *pa_pstDeviceInfo);*

2.6 Streaming Devices (Character Devices)

The streaming devices are similar to the block devices. The main difference is that not only an entire block can be written to the device but also single characters. Another difference is the T_DEVICE_INFO block that contains important information about the device. The meaning of each entry in the T_DEVICE_INFO structure differs between block devices and character devices. Please refer to the appropriate code documentation to get more information about the meaning of each entry in the T_DEVICE_INFO structure.

Such as the block devices also character based devices provides a common interface to the device manager consisting of the three functions read, write and info. To be consistent in the names they are also named blockRead and blockWrite although only single characters are written or read from the device instead of entire blocks.

2.6.1 UART

<u>Files associated:</u> *devUART.c, devUART.h*

<u>Directories:</u> *BLACKSheep/common/msd*

The UART streaming device driver uses the low level UART driver to communicate over the on-chip UART. Similar to the block devices the file *devUART.c* provides the three functions needed by the device manager to communicate with the character device.

Interface to the device manager:

*signed long dev_uart_blockRead(void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*signed long dev_uart_blockWrite(void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*T_ERROR_CODE dev_uart_info(void *pa_pDevHandle, T_DEVICE_INFO *pa_pstDeviceInfo);*

2.6.2 IO-buffer

<u>Files associated:</u> *IOBUFconfig.c, IOBUFconfig.h, devIOBUF.c, devIOBUF.h*

<u>Directories:</u> *BLACKSheep/common/msd*

The io-buffer device implements either a FIFO or a FILO buffer, selectable at runtime by a parameter in the initialisation function. The buffer size can be specified at runtime by a setup parameter.

Prototype declarations:

T_IOBUF_HANDLE iobuf_open (unsigned char pa_cBufferType,
unsigned char pa_cBufFullHandle,
unsigned short pa_nBufferSize);
T_ERROR_CODE iobuf_writeData (T_IOBUF_HANDLE pa_hDevice, unsigned char pa_nData);
*unsigned char iobuf_readData (T_IOBUF_HANDLE pa_hDevice, T_ERROR_CODE *pa_cError);*
unsigned short iobuf_close(T_IOBUF_HANDLE pa_hDevice);
unsigned short iobuf_getBytesInBuffer (T_IOBUF_HANDLE pa_hDevice);

pa_cBufferType:	Specifies the buffer type
pa_cBufFullHandle:	Specifies how to handle a buffer full condition
pa_nBufferSize:	Size of the buffer in byte

Example:

The example shows how to initialize the io-buffer as FIFO buffer.

T_IOBUF_HANDLE hIObuf = iobuf_open(IOBUF_FIFO, IOBUF_BLOCK_RW, 512);

The macros are declared in the file *IOBUFconfig.h.*

Interface to the device manager:

*signed long dev_iobuf_blockRead(void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*signed long dev_iobuf_blockWrite(void *pa_pDevHandle, unsigned long pa_nBlockAddr, unsigned long pa_nBlockCount, unsigned char *pa_pcData);*
*T_ERROR_CODE dev_iobuf_info (void *pa_pDevHandle, T_DEVICE_INFO *pa_pstDeviceInfo);*

2.7 IO-Manager

Files associated: *iomgr.c, iomgr.h*

Directories: *BLACKSheep/common/stdio*

Since the BLACKSheep is based on the VDK micro kernel from Analog Devices he can operate in a multithreaded environment. To arbitrate accesses to devices for each running thread the IO manager was created. So each thread gets his own channel for stdin and stdout. The assignation of the io channels to the threads is managed by the io-manager
The io manager has to be initialized by calling the *io_setup* function.

If the user creates a thread that needs access to a character device as standard input or standard output the user has to assign two streams to the thread, one as input and one as output by calling the *io_setThreadIO* function. Done this, a call to functions like *printf* or *getchar* are redirected to the appropriate input or output stream.

Prototype declarations:

T_ERROR_CODE io_resetThreadOutput(VDK_ThreadID pa_idThread, FILE
**pa_OutputStream);*
*T_ERROR_CODE io_setThreadIO(VDK_ThreadID pa_idThread, FILE *pa_InputStream,*
*FILE *pa_OutputStream);*
T_ERROR_CODE io_removeThreadIO (VDK_ThreadID pa_idThread);
*FILE *io_getThreadInput (VDK_ThreadID pa_idThread);*
*FILE *io_getThreadOutput (VDK_ThreadID pa_idThread);*

2.8 TCP/IP stack

The TCP/IP stack is based on the lwip stack. The stack was ported by Analog Devices to support the Blackfin processor family. Please refer to the appropriate documentation from Analog Devices that comes with the VDSP++ development environment.

2.9 Error handling

Each function that can causes errors returns a common error type T_ERROR_CODE. Two special error conditions are defined in the include file datatypes.h (see section 2.11.2): ERR_NONE for no errors and ERR_GENERIC. Special error values are defined in the config-files of each driver.

2.9.1 Hardware errors

Each driver enables the hardware error interrupts. So if a hardware error occurs the driver specific error handling function is called.

2.10 Include Directories

There are two special directories witch should be added to each project as additional include directories: *driver/include* and *blacksheep/include*.

2.11 Important include files

2.11.1 environment.h

Directory: *project directory*

The environment.h included file contains the more important hardware specific macros like the flash configuration and the CLKIN frequency, the GPIOs for the signalling led, the CS for the SD-card and the CF-card configuration. Each project should have his own environment.h file. Hence the file should be in the directory where the project file is.

2.11.2 datatypes.h

Directory: *driver/include*

This file contains a few type definitions for the most common byte oriented types and the definition for the error codes.

Common error codes:

ERR_GENERIC	generic error
ERR_NONE	no error

2.11.3 stdio.h

Directory: *BLACKSheep/include*

The stdio.h contains important macros that assign the standard io functions to the BLACKSheep specific one, identified by a “bs” as prefix. Including this file common calls to the io functions like fopen, fclose, fwrite and so one can be done. The pre-processor replaces such calls with the BLACKSheep specific file io functions like bs_fopen, bs_fclose. The calling parameters are the same as the standard functions corresponding to the ANSI C standard. If you have such ANSI calls (fopen, fclose …) within your project and you don’t include the stdio.h file the linker of the VDSP++ uses the standard io libraries from Analog Devices to resolve such symbols. Be aware that this io libraries works only with the JTAG device from Analog Devices.

2.12 Precompiled libraries

The BLACKSheep framework contains precompiled libraries for several drivers. They are located in the *driver/lib* directory. Be aware that some driver libraries works only with the development or evaluation boards provided by Bluetechnix because the uses hardware specific units.

3 Revision History

2004 12 20	First release V1.0 of the Document
2005 05 03	Revision 1.1
2006 03 03	Changes for BLACKSheep 0.5.1 Revision 2.0
2006 03 03	Revision 2.1
2006 11 14	Revision 2.2

A List of Figures and Tables