BRO - BeagleboaRd OS

The origin documentation can be found on GitHub: bro-fhv.github.io/docs/

List of contents

Goals

This is the documentation of our operating system for a resource limited system. The basic goal was to implement a bare metal OS on a BeagleBone.

The basic points are:

- a monolitic kernel
- applications in PHP and C
- an event driven application layer
- use network for communication via TCP and HTTP
- HTML/CSS/JS (client-side) will be delivered to the client and communicate via AJAX with the application

Ultimately we should be able to control DMX headlights with our applications via a browser.

Current State

The current state of the project contains:

- a hardware abstraction layer
- drivers for LEDs, GPIO, Serial Port, Timer
- memory management
- file system
- elf loader
- process scheduler
- library for client applications to communicate with the OS
- Ethernet component with UDP

Todo

- porting of LWIP to use TCP/IP which is needed for PHP
- porting of PHP
- implementation of the DMX driver
- refactoring of the HAL

Team

- Thomas Gaida
- Stefan Lässer
- Johannes Schwendinger
- Johannes Wachter
- Michael Zangerle

References

BeagleBoard:

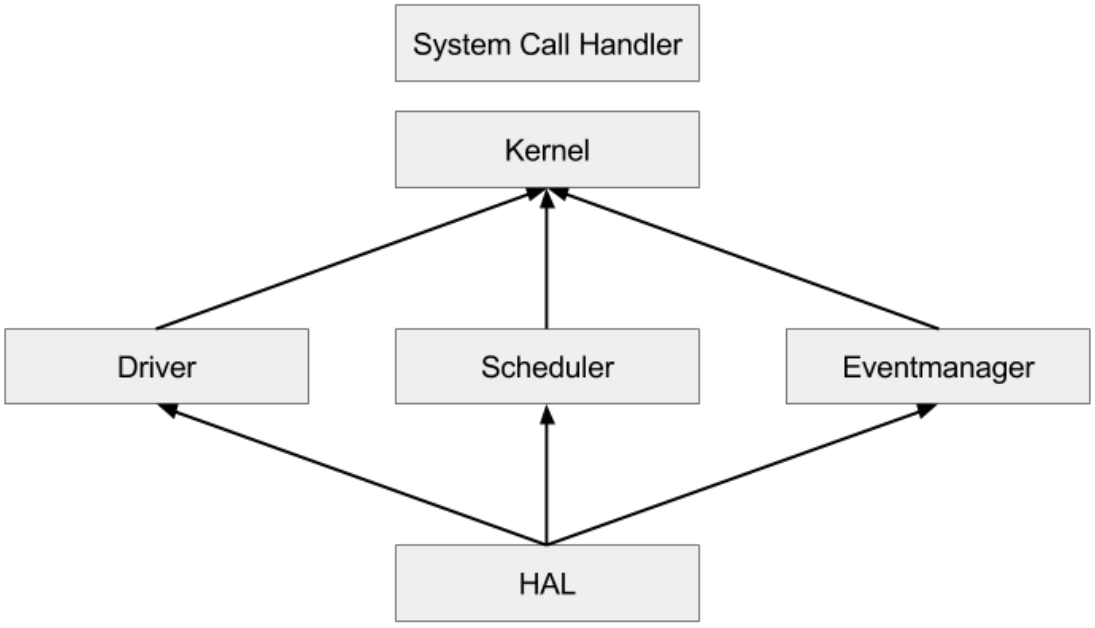
- http://beagleboard.org/

University of Applied Sciences Vorarlberg

- http://www.fhv.at/

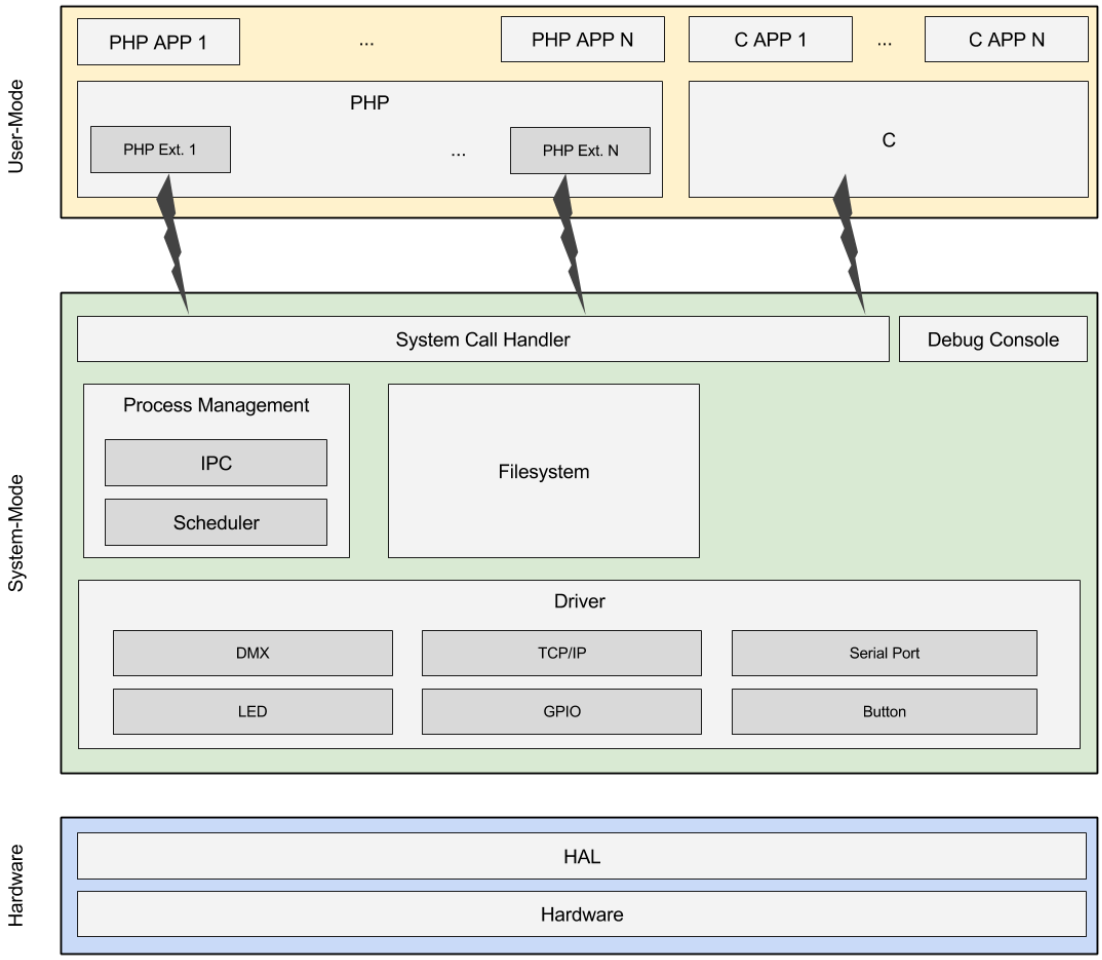
Project Structure

- Application: each application = one repository
  - PHP-Extensions
  - C-Lib
  - PHP-Apps
  - HTTP-Webserver
- Kernel: each block in architecture = one repository
  - File-System
  - Event-Manager
  - Scheduler + Process Management
  - Driver: each driver as folder
- GPIO
- HAL: one repository -> components as folder



The Architecture

Below you can find an overview of the basic structure of our planned operating system.

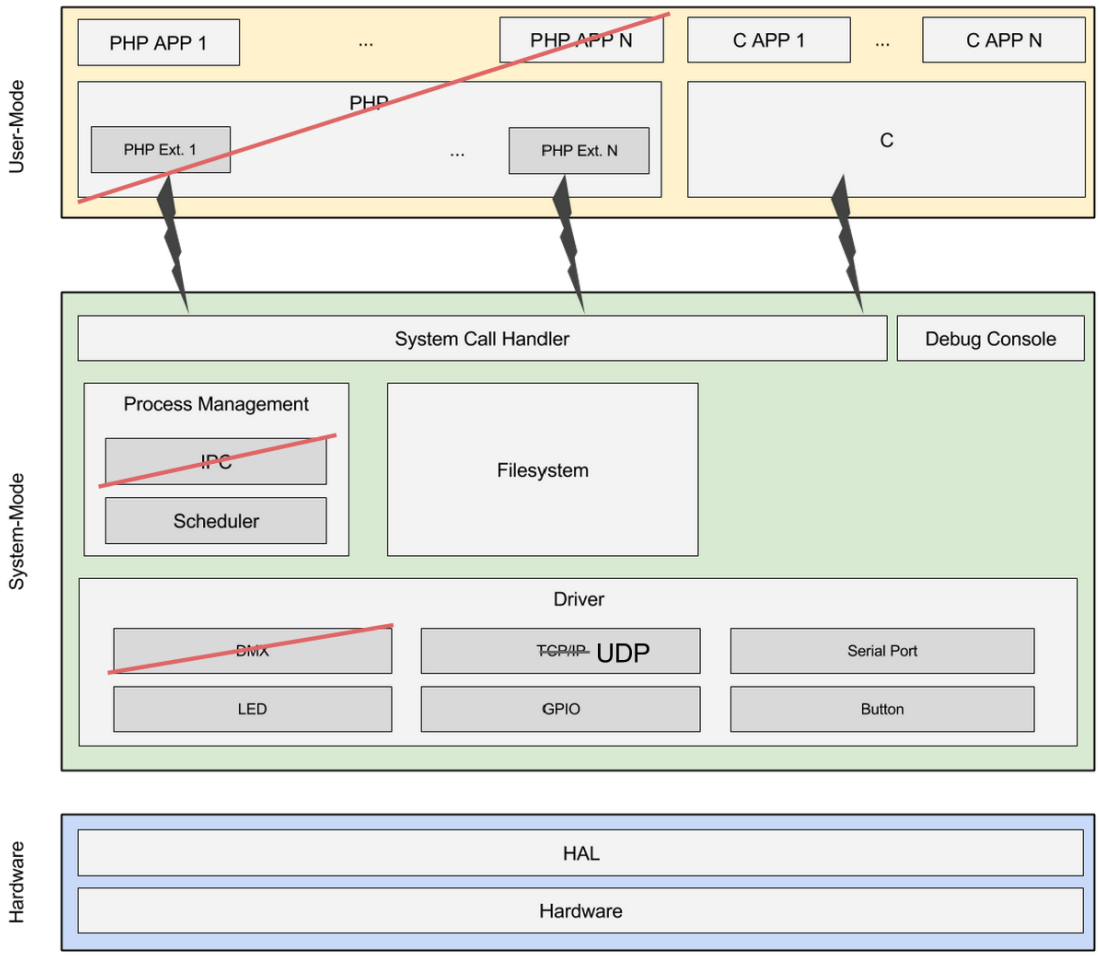


Modifications

During development we faced various problems. One of these problems was the porting of LWIP on our system. This specific problem was caused by an unaligned address access and we were not able to solve it. Ultimately we found a solution (partial reimplementation of LWIP code) and got it working so we could at least use UDP.

Unfortunately to run PHP application we would have needed TCP/IP.

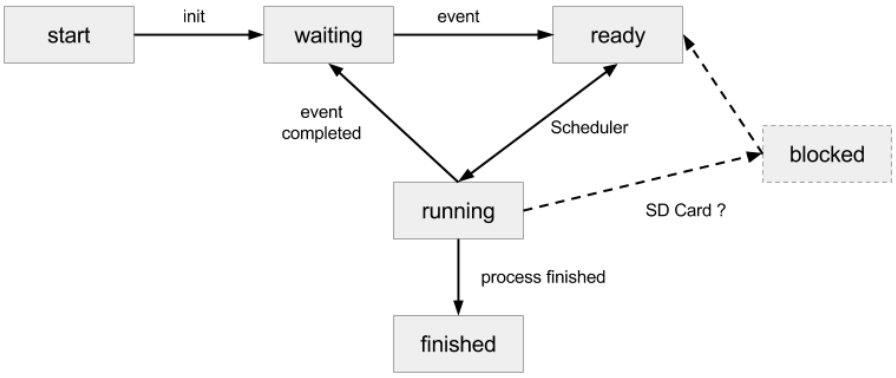
Below you find an updated illustration of our current operating system structure.



Process-Management

Process Lifecycle

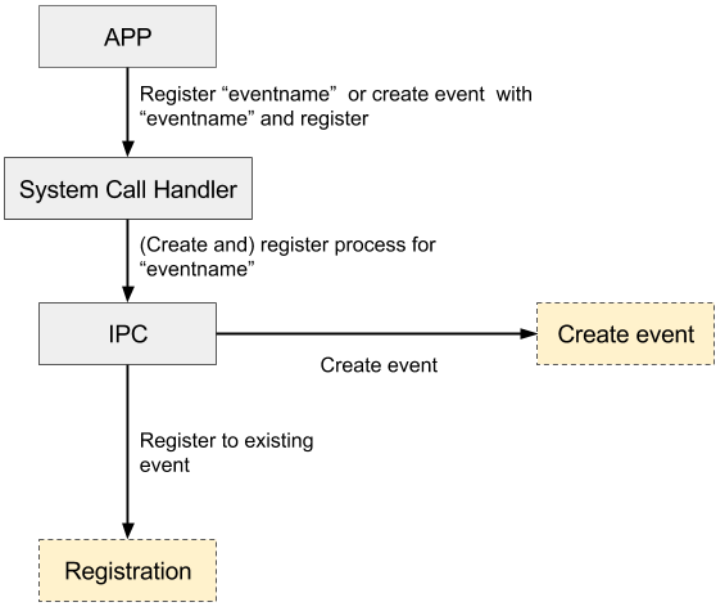
Below you can find an overview over the basic lifecycle of a process.



Process-Event-Management

Below you can find an overview over the basic functions concerning the event management:

- creating
- register listener
- deregister listener
- emit
- register emitter
- deregister emitter



API

```
bool events_create(char event_name[], int count, processID pid[]);


bool events_register_listener(char event_name[], processID pid);

bool events_deregister_listener(char event_name[], processID pid);


bool events_emit(char event_name[], processID pid, void* data);

bool events_register_emitter(char event_name[], processID pid);

bool events_deregister_emitter(char event_name[], processID pid);
```

The function to create events offers the possibility to limit the number of processes listening and/or to limit the events to specific process ids.

Scheduler

Below you can find an overview over the basic functions concerning the scheduler:

- Process management (start, stop, wait, kill)
- Process switching

```
void SchedulerRunNextProcess(Context* context);

void SchedulerStartProcess(processFunc func);

Process* SchedulerCurrentProcess(void);

void KillProcess(processID);

void loadProcessFromElf(uint32_t length, uint8_t* data);
```

The following code segment represents a process.

```
/*
 * Process structure
 */
typedef struct {
    processID id;
    processState state;
    processFunc func;
    programCounter pc;
    registerCache reg[REG_COUNT];

    /* Control Process Status Register */
    cpsrValue cpsr;

    uint32_t* masterTable;

} Process;
```

Problems

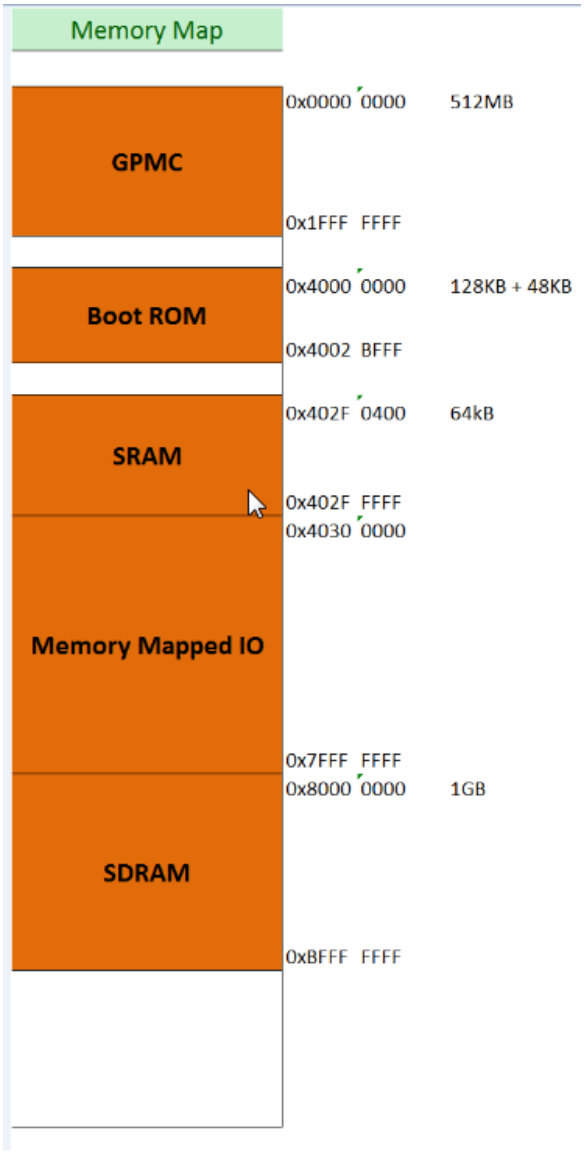
Because we assigned a wrong parameter to the following function (wrong REG_COUNT) we had some major problems when we switched processes. Because of this bug we lost the process context which resulted in some problems when loading an elf file.

```
memcpy(context->reg, gThreads[gRunningThread].reg, REG_COUNT);
```

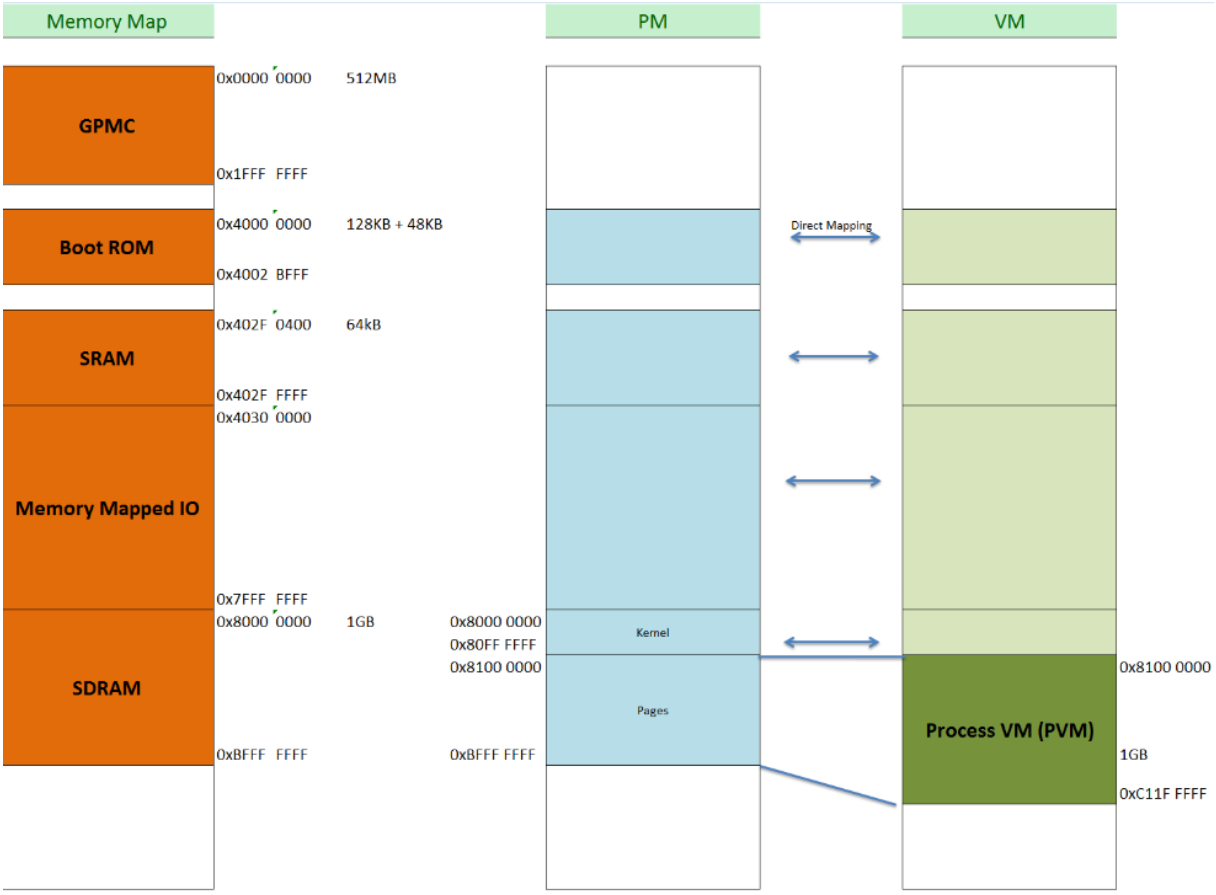
Memory Management Unit

Memory Model

The here given illustration shows the memory model of our BeagleBone board. You can find the full description in the AM335x Technical Reference Manual.



The below shown graphic shows the memory concept of our MMU. It shows how we map the physical memory to your virtual memory.

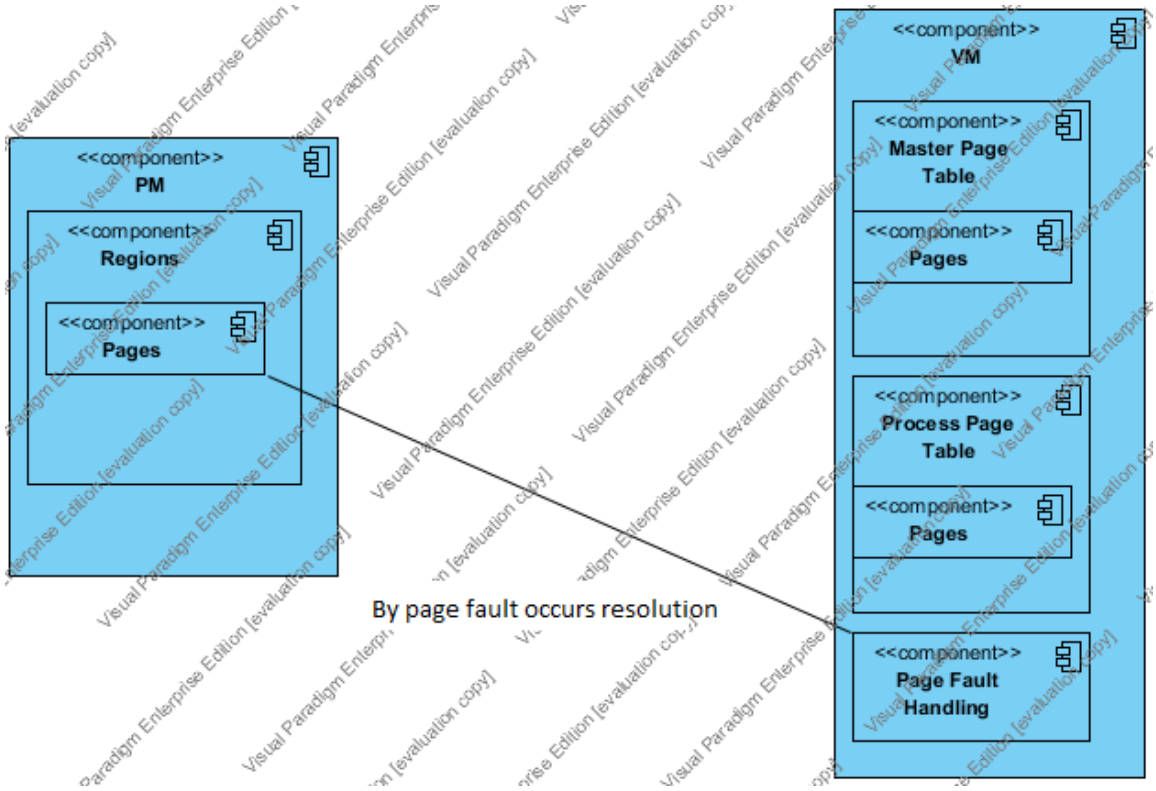


Moreover you can read information about the memory sizes and page numbers listed in the table here.

Memory informations			
Kernel	16MB	16777216	Byte
Process virtual memory	1GB	1073741824	Byte
Page size	64kB	65536	Byte
Page table entries		16384	Entries
Entry size	4Byte	4	Byte
Page table size	64kB	65536	Byte
Maximal processes	32		
Page table size	2MB	2097152	Byte

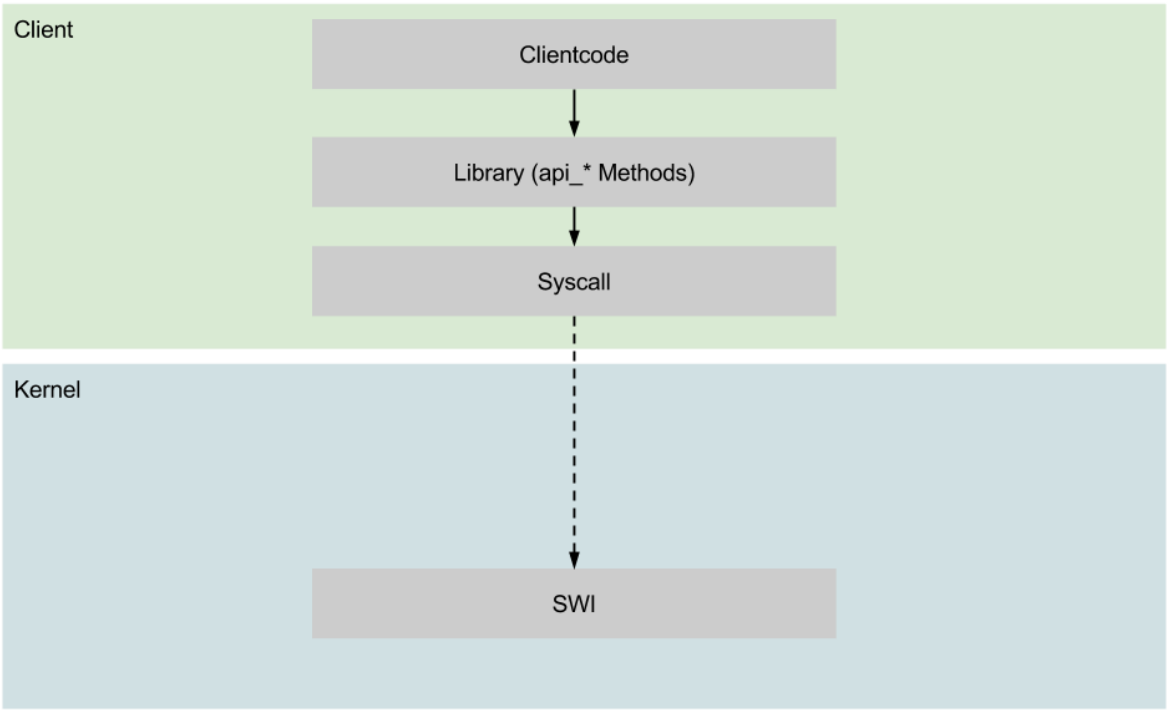
As you can see we use large pages with 64kb. Our process virtual memory has a size of 1 GB. This allows us to save 16384 pages. With this memory model we are able to manage up to 32 processes. We do not think that we need more than 32 processes for our small operating system.

The next graphic gives information about the components of the physical memory and virtual memory. It shows that the physical memory consists of different regions. Each of these regions has pages. The virtual memory consists of a master table and process tables. With the virtual memory we also handle the logic of page faults.



The Client

Below you can find an overview over the basic workflow of a client/application program. The client program uses the api_* methods of the library which itself calls the syscall method and results in a software interrupt in our OS.



Software Interrupt

The processes loaded by elf use software interrupts to communicate with the operating system. Therefore we implemented a small library for our so called client. This library provides a console output and access/control to the onboard LEDs, GPIO and Ethernet (limited to UDP). Below you can find an example for printing text onto the console.

```
/**
 * Print function alias for standard printf
 */
void lib_print(const char* format, ...) {
    char parsed[API_PRINTF_MAXLENGTH];
    va_list args;
    va_start(args, format);
    vsprintf(parsed, format, args);
    va_end(args);

    SyscallArgData data;
    data.swiNumber = SYSCALL_PRINTF;
    data.arg1 = (uint32_t) & parsed;
    Syscall(&data);
}
```

The abstraction layer provided by the library uses the following struct to communicate with the OS and supports five arguments and two return values.

```
typedef struct SyscallArg
{
    uint32_t swiNumber;
    uint32_t arg1;
    uint32_t arg2;
    uint32_t arg3;
    uint32_t arg4;
    uint32_t arg5;
    uint32_t result;
    uint32_t result2;
} SyscallArgData;


#pragma SWI_ALIAS(Syscall, 1)
extern void Syscall(SyscallArgData* data);
```

The software handler of the operating system processes the incoming syscalls caused by the software interrupts and forwards them to the target components of the OS.

```
void SwiForward(SyscallArgData* data) {


    uint32_t result = -1;
    void* resultPointer=NULL;


    switch (data->swiNumber) {


    case SYSCALL_PRINTF:
        SwiStdioPrintf((const char*) data->arg1);
        break;
    case SYSCALL_LED_ON_0:
        SwiLed0On();
        break;
        ...
```

Timer

The following section is explaining the API and the problems we were faced during implementation of the timer functionality. In the API section we also go into detail and explain some important stuff that needs to be known when using the timer of our operating system. Last you can find information about which timer were already used.

API

The API offers the opportunity to easily configure and enable a specific timer by just using the following enum. So the user of a timer does not need to have knowledge about the hardware address of the timer he/she wants to use.

Here you can see the enum:

```
typedef enum {
    Timer_TIMER1MS = 1,
    Timer_TIMER2,
    Timer_TIMER3,
    Timer_TIMER4,
    Timer_TIMER5,
    Timer_TIMER6,
    Timer_TIMER7
} Timer;
```

The first method that needs to be called is TimerConfiguration. As parameter you need to pass:

- the Timer (use the enum) you want to configure
- the milliseconds
    - which specify the amount of time after which the first interrupt should be raised
    - and how much time should pass between each interrupt
- the InterruptRoutine, which is called when the interrupt occurs.

In the InterruptRoutine you can do the stuff you want. The interrupt flags and so on were reset automatically.

Here you can see the API for the configuration:

```
typedef void (*InterruptRoutine)(void);


int32_t TimerConfiguration(Timer timer, uint32_t milliseconds, InterruptRoutine
routine);
```

To start, pause, resume, stop and reset a specific timer you can use the following methods:

```
int32_t TimerEnable(Timer timer);

int32_t TimerPause(Timer timer);

int32_t TimerContinue(Timer timer);

int32_t TimerStop(Timer timer);

int32_t TimerDisable(Timer timer);

int32_t TimerReset(Timer timer);
```

If a timer is started it runs until it is stopped or paused.

Our system also provides the opportunity to use a lightweight delay timer. The TimerDelaySetup method is called in the main of the OS - so you don't need to call it again.

```
void TimerDelaySetup();
```

To use the delay timer you use TimerDelayDelay. This method offers you a one-shot delay. When the passed milliseconds are over, an interrupt occurs and the timer stops automatically. This timer is processed as blocking operation - so when the delay timer is started no other code will be executed.

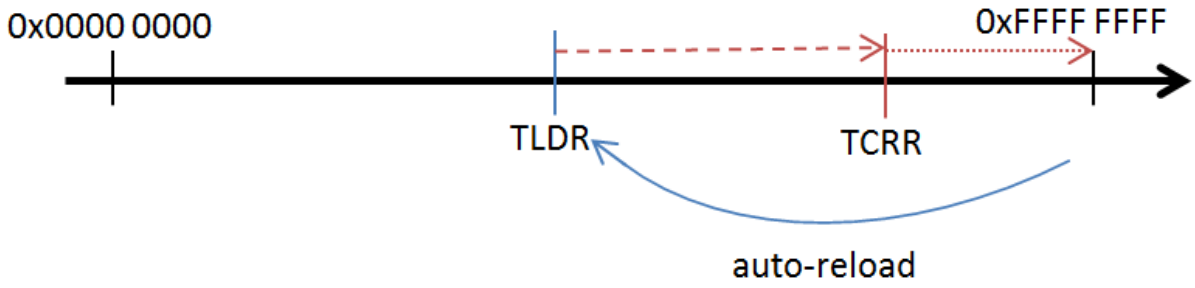
```
void TimerDelayDelay(uint32_t milliSec);
```

Used clock

For every timer we use the high frequency system input clock with 32 KHz as clock source.

Problems

The only problem we were faced to was to realize the trigger of the interrupt every time the set amount of time has passed. The following image gives a clue how we have realized this. The "timer counter register" (=TCRR) is increased automatically and the starting value is equal to the value of the "timer load register" (=TLDR). The value of the TLDR is used as reset value of the TCRR after an overflow has occurred (means TCRR value > 0xFFFF FFFF). In this case the TCRR value is set to the value of the TLDR and the interrupt is triggered (=overflow-interrupt).

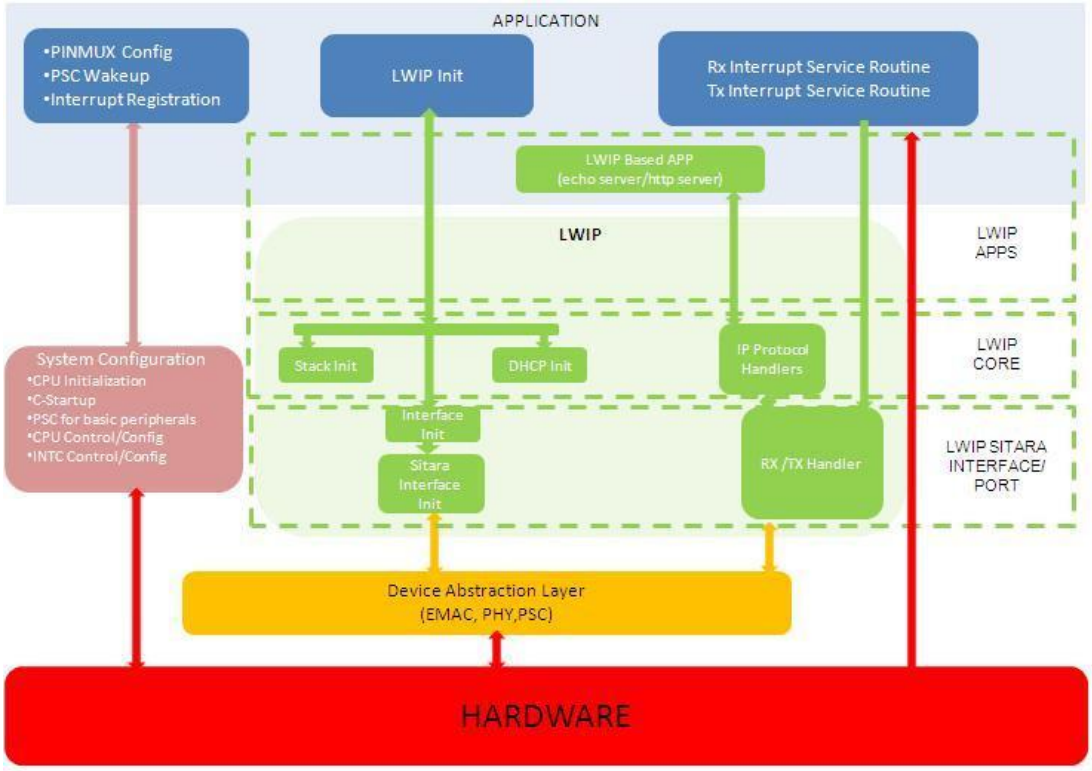


Which timers were used

- Timer2 is used for the scheduler
- Timer7 is used for DelayTimer
    - DelayTimer is used for the network configuration
    - DelayTimer is used for the SD card initialization

Networking

This project uses LWIP for basic networking tasks. The porting is inspired by Startaware.



Problems

The biggest problems in this component were some unaligned accesses. These were caused by unpacked structs in the porting part of LWIP.

Solution

The solution for this problem was to reimplement the UDP and IP part by our own which was accomplished by defining structs for each protocol header and cast the data package. This prevented the unaligned accesses.

Example: UDP header contains the IP and Ethernet header:

```
typedef struct {
    uint8_t destMacAddr[MAC_ADDR_LENGTH];
    uint8_t srcMacAddr[MAC_ADDR_LENGTH];
    uint8_t type[TYPE_LENGTH];
} eth_header_t;


typedef struct {
    uint8_t ethHeader[14];


    unsigned version :4;
    unsigned ihl :4;
    uint8_t tos;
    uint16_t tot_len;
    uint16_t ident;
    uint16_t flags_offset;
    uint8_t ttl;
    uint8_t protocol;
    uint16_t checksum;
    uint8_t srcIp[IP_ADDR_LENGTH];
    uint8_t destIp[IP_ADDR_LENGTH];
} ip_header_t;
```

```
typedef struct {
    uint8_t ethHeader[14];
    uint8_t ipHeader[20];

    uint8_t srcPort[2];
    uint8_t destPort[2];
    uint8_t len[2];
    uint8_t checksum[2];

    uint8_t data1;
    uint8_t data2;

    uint8_t dataRestStart;
} udp_header_t;
```

Current State

In the current state it is possible to receive and send packages with UPD. An application can start listing for a specific port and can poll for incoming data.

Outlook

For a real usage of LWIP it would be important to fix the porting of LWIP and use the whole library. That would raise the security and the reliability.

After that the IPC should be implemented to asynchronously inform the application for incoming packages.

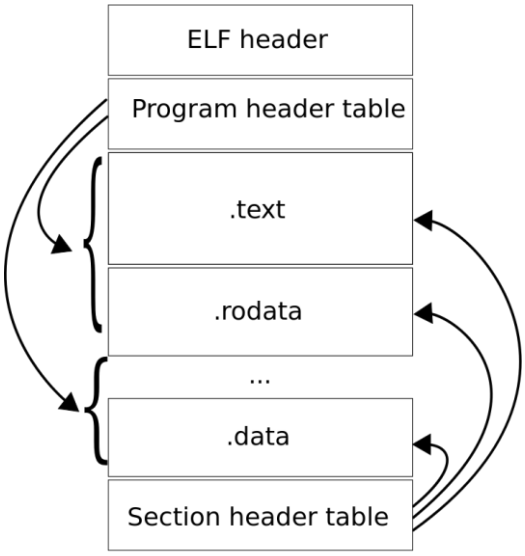
Code Example

```
void main() {

    BroUdpInit(PORT);


    while (1) {

        if (BroUdpHasData(PORT)) {

            printf("echo\n");


            upd_package_t* package = BroUdpGetData(PORT);

            BroUdpSendData(package->sender, PORT, package->data, package->len);


            free(package->data);

            package->data = NULL;

            package->len = 0;

        }

    }

}
```

Elf Loader

To load new processes we use ELF files which were loaded from SD card.

Below you can see the structure of an ELF file.



An ELF file has two views: the program header shows the segments used at run-time, whereas the section header lists the set of sections of the binary.

To load the elf file we read its header to get positions and size of the program headers and the entry point of the program. Then we iterate through all program headers, copy their data into memory and map the virtual addresses to the newly allocated memory.

Example for ELF initialization

Below you find an example for the initialization of an elf file.

```
// load elf
startFileSystem();
FILINFO fi;

if (f_stat("BRO_UDP.out", &fi) == FR_OK) {
    uint8_t* dataBuff = malloc(fi.fsize);
    getElfFile(dataBuff, fi.fsize, "BRO_UDP.out");

    // start a process
    loadProcessFromElf(0, dataBuff);
    free(dataBuff);
}
```

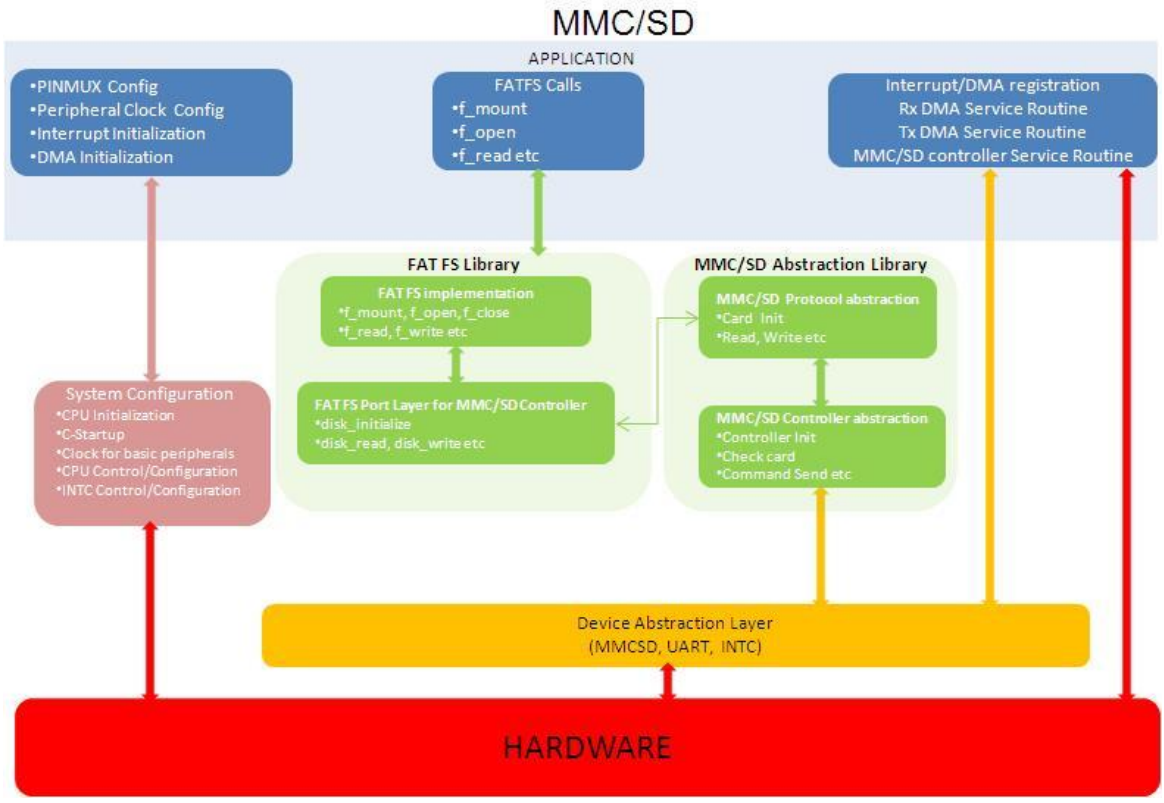
SD-Card

This project uses FatFs as file system. It is a generic FAT file system module for small embedded systems. FatFs Homepage

Features supported

The whole SD part is directly built up on the StarterWare MMCSD Driver Because of this, the following features are available:

- Support for SD v2.0 standard
- Support for Standard Capacity and High capacity cards
- Support for Standard Speed and High Speed cards
- DMA mode of operation



Changes

In order to work with the BRO OS some minor changes were necessary. The whole interrupt had to be reconfigured to use the already implemented interrupt driver of the BRO OS. Additionally the same was done with the timers and clocks as well as UART.

Current State

In the current state it is possible to read File from an SD Card. This is done with the elf loading.

Outlook

For a real usage of SD card functionality some method it is necessary to implement methods that can handle file reading and writing on a higher level.

Code Example

The following code example shows how initializing is done

```
/**
 * Inits file system and checks for mounted sd
 */
int startFileSystem(void)
{    volatile unsigned int i = 0;
    volatile unsigned int initFlg = 1;
    /* Initialize console for communication with the Host Machine */
    /* Configure the EDMA clocks. */
    EDMAModuleClkConfig();
    /* Configure EDMA to service the HSMMCSD events. */
    HSMMCSDEdmaInit();
    /* Perform pin-mux for HSMMCSD pins. */
    HSMMCSDPinMuxSetup();
    /* Enable module clock for HSMMCSD. */
    HSMMCSDModuleClkConfig();
```

```
    /* Basic controller initializations */
    HSMMCSDControllerSetup();
    /* Initialize the MMCSD controller */
    MMCSDCtrlInit(&ctrlInfo);
    MMCSDIntEnable(&ctrlInfo);
    while(1)
    {
        if((HSMMCSDCardPresent(&ctrlInfo)) == 1)
        {
            if(initFlg)
            {
                HSMMCSDFsMount(0, &sdCard);
                initFlg = 0;
            }
            return 1;
        }
        else
        {
            //errorhandling code
        }
    }
}
```

BRO-Network Client

Below you can see our network client. Over this client you can chat with other user in the network. It works over LAN and Wireless-LAN. Also each chat user can switch on the four LEDs of the beagle bone board with a simply checkbox activation click.

On our beagle bone runs a chat server in our operation system. Every time the server receives a message, we broadcast it back to all users in the network. If a user switch on a LED we invoke our LED-Driver over our syscall routine.

Solution configuration

See GitHub: bro-fhv.github.io/docs/ -> Configuration