HTWG Konstanz

UNIVERSITY OF APPLIED SCIENCES

Faculty Informatik

Project BlueOS

Development of a Realtime Embedded Operating System

provided by: Moritz Nagel 279162
Daniel Urbanietz 279184

submitted at: February 27, 2011

Abstract

As a part of a computer science team project, of the University of Applied Sciences Konstanz, we developed an operating system for AVR micro controllers, especially the ATmega series. Actually there are a lot of operating systems, but none of the ones, we tested, could satisfy our needs, especially, because there were heavy conflicts with additional libraries for the peripherals of the micro controller. That is the reason why we started this project.

The key idea of this project was, to create a framework, that contains the kernel and a lot of modules. This allows the programmer, to easily use the peripherals without the need of additional libraries, which could be incompatible to the kernel. Another important point was to make the system highly scalable, which makes it possible for the programmer to enable only the functionality the software really needs. In the minimal configuration the kernel can need less than 1kB of program memory, dependent on the used micro controller, so the BlueOS could also be used with ATtiny controllers.

Currently, there is only support for a few ATmega devices and ATxmega device, but it is easy, to port the OS to another classic AVR device.

Contents

List of Figures

List of Tables

Abbreviations

AVR Advanced Virtual RISC

CAN Controller Area Network

CRC Cyclic Redundancy Check

CRC8 8-Bit Cyclic Redundancy Check

CSMA/CR Carrier Sense Multiple Access/Collision Resolution

DIP Dual in-line package

DIP40 Dual in-line package with 40 pins

DMA Direct Memory Access

EEPROM Electrically Erasable Programmable Read-Only Memory

ESYS Embedded Systems

FIFO First In First Out

FET Field Effect Transistor

GND Ground

I^2C Inter IC Communication

IC Integrated Circuit

ICMP Internet Control Message Protocol

IP Internet Protocol

IPv4 Internet Protocol Version 4

IR Infra Red

ISR Interrupt Service Routine

JTAG Joint Test Action Group

LED Light Emitting Diode

MAC Media Access Control

MSL Maximum Segment Lifetime

NACK Not Acknowledge

NRZ Non-Return-to-Zero

OS Operating System

PCB Printed Circuit Board

SPI Serial Peripheral Interface

SLIP Serial Line Internet Protocol

TWI Two Wire Interface - Equal to I^2C

TCP Transmission Control Protocol

UART Universal Asynchronous Receiver Transmitter

UDP User Datagram Protocol

USART Universal Synchronous Asynchronous Receiver Transmitter

USB Universal Serial Bus

Chapter 1

Introduction

1.1 BlueRider Project

The BlueRider project was started in summer semester 2008. Its main intention was to equip a normal Carrera slotcar with avr microcontrollers and control it via Bluetooth. Our first prototype consisted of three ATmegas, one ATmega128 and two ATmega8. In the next step we produced a small series of ten cars.

These cars were used for an embedded systems laboratory. The students had to develop some programs for the cars. We have tested several operating systems for the ATmega platform. None of them was practicable because there were many bugs or the non-commercial version was too restricted. Therefore we decided to develop our own operating system, which is not only intended for the BlueRider platform. From the beginning the OS[1] should be highly scalable and flexible.

[1] Operating System

Chapter 2

Kernel

2.1 Summary

The BlueOS is a preemtive multitasking operating system, that is designed, to run on AVR[1] microcontrollers, that are produced by the Atmel Corporation.

2.2 Internal Description

2.2.1 Task Model

As the BlueOS is a multitasking system, tasks need to be represented by a logical unit. This logical unit is the task control block, which is a data structure, that contains all necessary informations needed for the scheduler and other system modules. It's most important informations are about the stack of the task. Therefore a pointer to a stack segment and the size of the stack are stored, so all tasks can have different stack sizes.

In figure 2.1 the stack of a task is shown. An important debug method are the magic numbers, at the beginning and the end of the stack. These numbers are usually a bitwise invert of the task ID, which are created, when the task is created. If the magic numbers are changed, the dispatcher recognizes it as stack overflow and stops the system, if the debug mode is enabled.

[1] Advanced Virtual RISC

Another important thing in the stack construction is the address of the `void blueOsFinalDestination()` function, which is the return function of the task. As the BlueOS allows tasks to return, a return address for the task function must be stored at the bottom of the stack. Studying the datasheet of the AVR CPU has shown (see [2]), that a function call does only store the return function on the stack. If a task returns, the CPU pops the return address from the stack and jumps there, so there are no additional informations needed on the stack. The CPU jumps to the function `void blueOsFinalDestination()`, which can be seen as function call. This function kills the current task, so the task is removed from the system. The return address of the task is stored in the higher addresses of the stack space of the task, because the AVR CPU decreases the stack pointer at a push operation and increases it at a pop operation.

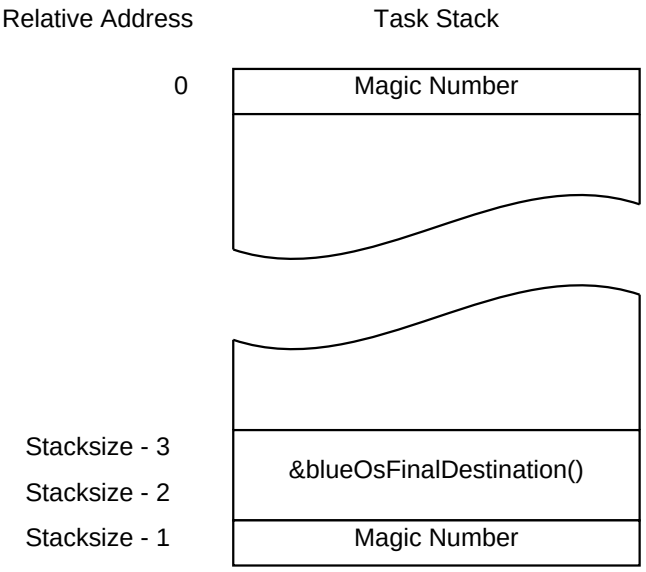

FIGURE 2.1: Stack Construction of a Task

The TCB also contains information about the father id and the holding ressources like semaphores. The father id points to the task which has created the task. Therefore the family structure of a task and its children can be followed up. If a task is killed by another task or has returned, the complete family structure is killed. That means all child-tasks are killed, too. Additionally all ressources occupied by this family structure are released immediately. This proceed makes it possible to have complex task family structures which can be deleted by only calling the kill function for father task. Another is important part of the TCB is the `BlueOsTaskHandle` which is a pointer to a TCB pointer. You can set this pointer as a parameter in the function blueOsTaskCreate(...) otherwise the parameter can be set to 0. But with the Handle it is easy implement a join mechanism for

tasks because the Handle is set to 0 when the task is killed. This could be very useful if one task "'shoots"' some jobs and has to wait until they are finished. Of course this functionality needs a lot of space in program memory and data memory. Therefore you can constitute that a task will never die in the config file.

2.2.2 Context Switching

The context switch is implemented inside a timer overflow interrupt service routine. When this ISR is called, the current program counter is pushed onto the stack, which is needed, to return from the ISR[2]. Then, all 32 general purpose registers and the status register are pushed onto the stack. Finally, the stackpointer is stored into the task control block of the current task.

If the debug mode is enabled, the stack of the current task it checked for overflows and underflows. Therefore, it simply verifies, if the stackpointer is inside the valid range of the stack. Also it checks some magic numbers, which are stored at the beginning and the end of the stack area of the current task. If the stack is damaged, or the stackpointer is invalid, the kernel panic function is called.

Now the ISR checks, if it was called by the software directly, or if it was called due to a timer overflow. In case of a timer overflow, the system time is incremented by one millisecond, because the timer overflows every millisecond. Also, the timer counter is reinitialized, to secure, that the next timer overflow is in one millisecond. If the kernel options `BLUE_OS_USE_DELAY` or `BLUE_OS_USE_USER_WATCHDOG` are enabled, the functions are called, that wakes up the tasks which are waiting for a delay or which calls the callback functions for the user watchdog.

After that, the scheduler is called, which simply selects the next task ID. The implementation of the scheduler can be selected in the configuration file. Then the current task is set to the next task, and the next stackpointer is set. Again, the stack is validated, like before, because the stack could be damaged by another task.

Finally, the registers filled with the values, which were stored into the stack before. The ISR returns with "reti()", which enables the interrupts and returns in only one single instruction, so the interrupts are enabled, even if the ISR was called by the software with deactivated interrupts. Now, the next task is running. (see [2])

[2]Interrupt Service Routine

2.2.3 Preemption

An important feature of the BlueOS kernel is, that it is preemptive. This was realized by placing the dispatcher into the system timer ISR, so it is executed every time the ISR is called. Therefore the kernel will preempt tasks, if the interrupts are enabled.

2.2.3.1 Software Interrupts

In an operating system it is often necessary to cause an hardware interrupt by software, e.g. is a timer interrupt needed, to manually yield to another task. There are many different approaches described in some Internet communities, but many of them are complex workarounds, because the AVR micro controller does not support to cause hardware interrupts by software. Studying the AVR datasheet [2] has shown, that an ISR call is exactly the same, as a function call, with the single difference, that the interrupts are deactivated. So, our implementation of the software interrupt disables the interrupts in the first step and then does a function call to the timer ISR.

2.2.4 Scheduling

There are currently two scheduling modes, which the kernel can be compiled for, either `BLUE_OS_SCHEDULER_PRIO_RR` or `BLUE_OS_SCHEDULER_OWN_IMPLEMENTATION`. `BLUE_OS_SCHEDULER_PRIO_RR` is the standard scheduler, it is a priority based scheduling. It compares the priorities of all active tasks and selects the one, which has the lowest value. If there are several tasks with the same priority, the scheduler uses round robin, to schedule these tasks.

If `BLUE_OS_SCHEDULER_OWN_IMPLEMENTATION` is selected as scheduler, the programmer can implement an own scheduling algorithm, e.g. a static scheduling, a scheduler with dynamic priorities, or whatever the programmer likes. Therefore, the function `uint8_t blueOsScheduleOwnImplementation()` must be implemented. This function represents the scheduler and is called every time, the dispatcher needs a task to dispatch. The return value of this function is the task ID of the task, that will be running next. It is important, that the returned value is valid. If the task ID does not exist, the system is in an undefined state. If e.g.

a sleeping task is selected, this task will be forced to run, so the task might think, it received a signal.

2.2.5 Synchronization

2.2.5.1 Critical Section

The critical section is the basic synchronization mechanism of the kernel. This mechanism consists of two functions: `uint8_t blueOsEnterCriticalSection()` and `void blueOsLeaveCriticalSection(uint8_t state)`. When the critical section is entered, the current interrupt state is temporary stored and the interrupts are deactivated, then the temporary interrupt state is returned. If a critical section is left, the interrupt state is restored with the supplied state argument. This implementation allows to use this mechanism, if the interrupts are already deactivated, so it can be used e.g. in ISRs, which may be needed, because the critical section is used in other kernel functions, e.g. the queue.

2.2.5.2 Signal/Wait

If `BLUE_OS_USE_SIGNAL` is defined, the signal/wait functionality is enabled. This functionality allows to set tasks to waiting mode, until another task activates it again by a signal. A signal is a data structure, which must be initialized with the function void `blueOsInitSignal(BlueOsSignal* signal)`. The data structure contains a linked list where all tasks can enqueue for a signal.

Tasks can wait for a signal, by using
`int8_t blueOsWait(BlueOsSignal* signal)`. Other tasks can either wake a single task with the function `int8_t blueOsSignalSingle(volatile BlueOsSignal* signal)`, or wake all tasks, by using `void blueOsSignalBroadcast(BlueOsSignal* signal)`. The only thing, that is done in this functions is, that the task state is changed between active and waiting, the scheduler knows, which tasks are allowed to be scheduled and which are not.

2.2.5.3 Semaphore

If `BLUE_OS_USE_SEMA` is defined, the semaphore support is enabled. The semaphore is based on the signal/wait mechanism. The main intention of the semaphore, is to protect resources. Every task can request a semaphore with the blocking function `int8_t blueOsAcquireSema(BlueOsSema *sema)`. It can be released with the function `int8_t blueOsReleaseSema(BlueOsSema *sema)`.

The data structure of the `BlueOsSema` contains a `BlueOsSignal`, the owner ID and a pointer to another semaphore. Every task has a pointer to a semaphore, therefore it is possible to build up a linked list of all semaphores which are assigned to that task. This is important to free all resources in case of a kill of that task. The acquire function checks the owner ID, and if the semaphore is occupied it waits on the signal. When the semaphore becomes free (return from wait), it will be inserted in the linked list of the calling task. The releasing function resets the owner ID and signals the signal variable. If the task is killed the function `void blueOsKill(int8_t taskId)` releases all semaphores of that task.

2.2.5.4 Event System

If `BLUE_OS_USE_EVENT` is defined, the Event System is enabled. This system provides functions to generate events. In contrast to the normal signal/wait mechanism, the event can transfer data to the listeners. The first byte of the data can be used as a mask. Therefore the event system can easy be used to deliver addressed messages between the tasks. It is also frequently used to abstract from hardware. For example the only difference between UART[3] and TWI[4] is the corresponding event variable. Every Task can listen to an event with the function `int8_t blue_os_eventListen(BlueOsEvent *event,uint8_t mask, uint8_t *data);`.

- `*event` is the pointer to the event.
- `mask` can be used to mask events. If mask is not 0 the first byte from data which has generated the event is compared to the mask.
- `*data` points to the buffer. It should be always big enough to hold all data structures from that event.

[3] Universal Asynchronous Receiver Transmitter
[4] Two Wire Interface - Equal to I^2C

2.2.6 System Tasks

2.2.6.1 Idle Task

The idle task exists in every configuration of the BlueOS, it can't be deactivated and runs with the lowest priority. To prevent cases with no active tasks, which can bring the scheduler into an undefined state, it is not allowed, to change the state of the idle task. This task has always the ID 0, so algorithms can easily detect the idle task. As the idle task is a real task, it also has it's own stack. A special feature of this stack is, that the kernel uses the stack to prevent stack overflows in other tasks.

In the standard case, the idle task always checks, if another task is active. As soon as another task is active, the idle task yields. When the system starts, the first executed task is the idle task, so this task initializes the kernel modules, such as the shell, or the TWI module. Initializations could also be done before the system starts, but then the system has no secure stack, which it has in the idle task. As soon as all initializations are done, the task always yields to another task.

When the idle task enters it's main loop, it always checks, if other tasks are active. In case of an active task, which could happen due to an interrupt, the idle task yields. Early implementations of this task did always yield in the main loop, independent of other task states. However, this implementation caused, that the interrupts were deactivated the most time, because the dispatcher and scheduler needed way more time than the call of yield. That heavily disturbed the realtime software.

If `BLUE_OS_IMPLEMENT_OWN_IDLE_TASK` is defined, the user can implement the function `void blueOsOwnIdleTask()`, which is called by the idle task. In this case, the standard implementation is disabled, so the own implementation must not return! It is advisable not to yield in every cycle of the idle task. Instead a solution, which is described above, should be chosen.

2.3 Libraries

2.3.1 Container

This library contains container structures and the corresponding access functions.

2.3.1.1 Queue

The Queue container is a FIFO[5] storage, that can't be deactivated in the system configuration, because this container is also used in some kernel functions. In the configuration, the maximum queue size can be set with the define `BLUE_OS_QUEUE_DATA_TYPE`. This define configures the default integer type for the information about the queue, so it can be defined as `uint8_t`, `uint16_t`, etc.

Queues are already synchronized, so they can be used to exchange data between tasks and ISRs. The data stored in the queue is in the `uint8_t` format. If larger sizes need to be enqueued, each byte must be pushed one by one. In this case, all needed push operations should be inside a critical section, to guarantee, that only complete blocks are stored inside the queue.

As other data structures inside the BlueOS, there needs to be allocated a memory segment which the queue structure uses to store its data. To create a queue, there also needs to be a queue object, which is of the type `BlueOsQueue`. Finally the queue needs to be initialized with the function `void blueOsQueueInit(...)`.

Every queue can have several callBack functions, whiches are executed when an element is inserted. Internally it is realized as a linked list of blueOsCallback objects. In order to guarantee fast queue operation, these callback function have to be as fast as possible (like ISR's) and under no circumstances blocking.

The queue API provides the following functions:

- blueOsQueueInit(...)
  initializes the Queue structure

[5] First In First Out

- blueOsQueueInitBlocking(...)
  initializes the Queue structure with signals for Enqueue and/or Dequeue operation. The corresponding function (enqueue and/or dequeue is blocking)

- blueOsQueueEnqueue(...)
  adds one element to the queue. If the queue is full, -1 is returned or the function blocks until one element is dequeued.

- blueOsQueueDequeue(...)
  removes one element to the queue. If the queue is empty -1 is returned or the function blocks until one element is enqueued.

- blueOsQueueFlush(...)
  flushes the queue's content.

- blueOsQueueAddCallBack(...)
  adds a callBack object to the callBack list from the queue object. the callBack must be initialized with a vaild function pointer. The function has to match this function definition:
  `typedef void (*CallBackFunc)(uint8_t* source, uint8_t value);`

- blueOsQueueRemoveCallBack(...)
  removes the callBack object from the queue object.

2.3.2 Utils

2.3.2.1 us Timer

The μs timer functions can be used, to read out the current system time. To use this functions, `BLUE_OS_US_TIMER` must be defined. `uint32_t blueOsGetMs()` is used, to read out the amount of milliseconds, since the system was started. Programmers, who use this time should consider, that the system time is stored in a 32-bit variable, so the milliseconds will overflow every 49.71 days.

If a more precise time is needed, `uint16_t blueOsGetUs()` can be used, to read out the number of microseconds since the last millisecond began. The precision of the microseconds depends on the frequency of the micro controller, e.g. a system with a clock of 16MHz has a precision of 4us. If another frequency is chosen, the

precision can be calculated the following way: $precision = \frac{prescaler}{f_{CPU}}$, where the standard prescaler of the timer is 64.

There is also the possibility to manually set the system time, what can be done with `void blueOsSetMs(uint32_t time)`. This function sets the milliseconds to the passed value.

2.4 Configuration

To configure the BlueOS, there is already a configuration file: `blue_os_config.h`. This file contains all configuration definitions, which are commented or uncommented. To enable a special configuration, simply uncomment the needed definition. In Table 2.1 a short description of the configuration possibilities is shown:

Configuration	Description
BLUE_OS_MAX_TASKS *number*	Number of user tasks, that maximally run at the same time.
BLUE_OS_SYSTEM_TASKS *number*	Number of system tasks.
BLUE_OS_SYSTEM_STACKSIZE *size*	Size of the stack, of the idle task.
BLUE_OS_SCHEDULER_PRIO_RR	Priority based round robin scheduler is selected.
BLUE_OS_SCHEDULER_OWN_IMPLEMENTATION	Scheduler is implemented by the user.
BLUE_OS_DEBUG	Enables kernel panics.
BLUE_OS_IMPLEMENT_PANIC_CALLBACK	Enables an implementable callback function tor the kernel panic.
BLUE_OS_DEBUG_SHELL	Enables debug output on the shell, in case of a kernel panic.
BLUE_OS_USE_SHELL	Enables the shell.
BLUE_OS_USE_SHELL_UART *number*	Defines the UART port of the shell.
BLUE_OS_USE_SHELL_BAUD_RATE *rate*	Defines the baud rate of the selected UART.
BLUE_OS_SHELL_RXBUFFER *size*	Size of the receive queue of the shell.
BLUE_OS_SHELL_TXBUFFER *size*	Size of the transmit queue of the shell.
BLUE_OS_USE_SHELL_VT100	Enables VT100 terminal emulation on the shell.
BLUE_OS_SHELL_VT100_LISTENER	Enables VT100 key events.
BLUE_OS_IMPLEMENT_OWN_IDLE_TASK	User can implement an own idle task
BLUE_OS_USE_SIGNAL	Enables signal/wait synchronization.
BLUE_OS_USE_SEMA	Enables semaphore synchronization.
BLUE_OS_USE_EVENT	Enables the event system.
BLUE_OS_US_TIMER	Enables μs timer functions.
BLUE_OS_USE_DELAY	Enables passive delay.
BLUE_OS_TWI	Enables the TWI system.
BLUE_OS_TWI_MASTER	Run TWI in master mode.
BLUE_OS_TWI_SLAVE	Run TWI in slave mode.
BLUE_OS_TWI_OWN_ADDR *address*	Sets the address of the TWI device.
BLUE_OS_TWI_BROADCAST	Enables Broadcast Receiver (slave only)
BLUE_OS_USE_USER_WATCHDOG	Enables watchdogs for tasks.
BLUE_OS_TASKS_NEVER_DEAD	Removes ability for tasks to return.
BLUE_OS_QUEUE_DATA_TYPE	Type of the indexes in the queue.
BLUE_OS_MEMORY	Enables Memory Management
BLUE_OS_MEM_SIZE	defines the available Memory size
BLUE_OS_MEMBLOCK_SIZE	defines the memory block size
BLUE_OS_DYNAMIC_TASK	defines the dynamic task ability (Memory Management must be enabled)

Table 2.1: BlueOS Configurations

Chapter 3

Modules

3.1 TWI

The two wire interface (TWI) is the Atmel equivalent to the I^2C[1] bus. It is designed to connect several controllers. According to the ATmega datasheet the ATmega series supports normal I^2C operation and a multimaster mode. We wanted to use the multimaster mode to easily connect several controllers which are using the blueOs. Multimaster means that every participant can be the master of the bus. Therefore a arbitration protocol is needed. This arbitration is hardware implemented inside the ATmega. Unfortunately there are some heavy issues in the TWI hardware. While normal transfer, there is a flow control between master and slave, realized with Acknowledgements. But the last packet (the stop condition) needs not to be acked. Therefore any other controller can begin a new transmission after that packet. The problem is that the twi hardware is stopped when a packet is received until the corresponding interrupt service routine is executed. Within this time the hardware is completely blind and the bus in not monitored. Actually the i2c standard defines a minimum time gap between a stop and a start condition. This gap is not implemented in the TWI hardware. Therefore a new transmission can be initiated while the last slave is working on its ISR. The last slave might want to initiate a new transmission, but because it doesn't know from the still running transmission it will produce a collision. We tried to work arround this issue with a small monitoring routine that checks whether the bus is free. But a controller, which has lost the last arbitration, will still send its start condition

[1] Inter IC Communication

directly after a stop condition is recognized. The only possibility to solve this problem would be a special sending task which is always polling the bus traffic. This approach would be very slow and unhandy. So we decided to discard the multimaster implementation. It looks like that the new XMega series has solved this problem with independent modules for twi sending and receiving. But at this point we didn't tested a multimaster TWI with Xmega's.

Now the TWI implementation consists of few functions which are realizing a normal TWI with only one master. You can activate the TWI in the config file. A slave can use the `blue_os_twiEvent` to wait for incoming data. The implementation is completely interrupt driven.

- `blue_os_twi_send(...)`
  sends a 5 byte frame. In case of a master it is sent directly. In case of slave a internal buffer is filled and the function is blocked until the master has read it.
- `blue_os_twi_receive(...)`
  receives a 5 byte frame. In case of a master it is received actively via a TWI read. In case of slave the function is blocked until the master has sent the data.

3.2 Shell

3.2.1 Normal Mode

If `BLUE_OS_USE_SHELL` is defined, the shell of the BlueOS is enabled. The shell has simple access functions for read/write and integer output. At the current version, it is only possible, to choose UART as input/output for the shell. With "`BLUE_OS_USE_SHELL_UART` *channel*", the UART output can be selected, if the micro controller has multiple UART devices. By the setting "`BLUE_OS_USE_SHELL_BAUD_RATE` *speed*" the baudrate of the UART device can be selected.

The shell is completely interrupt driven, so there are receive and transmit buffers, which are used to exchange date between tasks and the interrupt service routines. The sizes of these buffers can be configured with "`BLUE_OS_SHELL_RXBUFFER` *size*" and "`BLUE_OS_SHELL_TXBUFFER` *size*".

There is a function, which initializes the shell. This function is automatically called by the idle task, before the user tasks start, so no further initialization is needed. If there is the need, to restart the shell, the function `void blueOsInitShell()` can be used therefore.

3.2.2 VT100 Emulation

There is a comfortable VT100 API for the shell. You can activate it in the configfile. It provides a set of functions to easily create a text-based GUI via VT100. VT100 is an old terminal standard from DEC, developed in 1979. Today almost every terminal is able to display these VT100 sequences. Additionally a event for the VT100 system can be activated in the config file. This event distributes incoming characters and VT100 codes, therefore the event uses a 16bit wide container. Please see "'Event System"' for further information about events and handling them. In comparison to the normal shell read function it can be very useful because it distributes the incomming data to every task which is listening on that event.

3.3 Memory Management

3.3.1 Overview

The BlueOs provides a dynamic memory management. You can activate it in the config file. The dedicated memory size m and minimal block size b has to be specified. According to the chosen Algorithm an additional data structure for management is allocated. The memory management can only provide 256 blocks. Therefore you have comply with:

$\frac{m}{b} \leq 256$

3.3.2 Functions

- `int8_t blue_os_mem_init()`
  Initializes the management table.
- `int8_t blue_os_alloc(uint8_t** ptr, uint16_t bytes);`
  Allocates `bytes` bytes of memory. In case of fail the return value is -1 otherwise it is 0. Please consider that, dependent on the chosen algorithm, more memory space than `bytes` is allocated (see Algorithms).
- `void blue_os_free(uint8_t* ptr)`
  De-allocates the given memory space.
- `void blue_os_memChown(uint8_t* ptr, uint8_t taskId)`
  Changes the owner of the given memory space. That could be useful if the holding task is going to die and the memory space should be used by another task.
- `void blue_os_freeKill(uint8_t taskId)`
  In case of a task kill this funtction is used to de-allocating all memory spaces whiches this task is holding. This function is only internally used.
- `int8_t blueOsCreateDynamicTask(uint16_t stackSize, uint8_t priority, blueOsThread _functionPointer, BlueOsTCB** handle);`
  This function creates a task with the given function pointer. The stack is allocated with the memory management. Please consider that a tcb is

allocated to. The owner of the allocated memory space is the new task. Therefore this function is suited for shooting single jobs.

3.3.3 Algorithms

3.3.3.1 Buddy

The Buddy system is a very common Allocating algorithm. Its characteristic is that it can only allocate powers of two. The dedicated memory space has to be a power of two, too. So it can build up a memory tree with memory blocks. At the allocation the best fitting block will be devided by two until it is to small to hold the data. At de-allocation the block is marked as free and if its neighbour is free too, they will be merged. This proceed avoids any outer fragmentation of the memory. The inner fragmetation could be very high, if not powers of two were allocated.

The size of its management data structure is dependent on the maximum task count. A table which can contain all blocks is generated. Each entry represents a single block in the memory space, it contains the power of the memory area, the owner task id and a free bit.

task count	task id	power	free
< 8	3bit	4bit	1bit
≥ 8	8bit	4bit	1bit

TABLE 3.1: buddy table entry

Therefore the data consumption s for the management can be calculated in the following way.

$$s = \begin{cases} \frac{m}{b} & , taskcount < 8 \\ \frac{2m}{b} & , else \end{cases}$$

3.3.3.2 FirstFit

The FirstFit algorithm is very simple. At the allocation the first fitting free memory area will be marked as used. At de-allocation the area is marked as free. The area is merged to its surrounding areas, if they are free, too. This proceed avoids

any inner fragmentation except the residues from fixed block size. But outer fragmentation could increase very fast. In worst case it can enter a state with high fragmentation. The management table consists of the same entries as the buddy table. But its size is much higher. In case of the buddy we need only powers of two to describe the size, therefore four bits are enough. The first fit algorithm can allocate any number of blocks, so we need eight bit to describe the size. Presently one entry needs three bytes of memory. For the future it can be reduced to two bytes in case of maximal 127 tasks. The owner id and the free bit can be merged with a bit field. But this is not yet implemented.

3.3.3.3 RotateFirstFit

The RotateFirstFit Algorithm is similar to the normal FirstFit. The only difference is the allocation. At every allocation the position of the next free area after the allocated one is saved. At the next allocation, this position is used as the startpoint for the next fitting block search.

3.3.3.4 BestFit

The BestFit Algorithm searches the best fitting memory area. It avoids splitting big memory areas in case of small requests. This proceed seems to be the optimal allocation algorithm. Unfortunately it forwards the outer fragmentation. In comparison to the other algorithms it is the worst. But there might be a few use cases in which it could be optimal. Only for that reason, it is implemented. In general we advice you against this algorithm.

3.4 Networking

3.4.1 Overview

The networking of the BlueOS is a very simple implementation, wich is designed to use very low resources, to fit on AVR-devices. However, there is simple support for TCP/IP. In comparison to some other networking stacks for AVR-devices, this implementation is capable of actively sending data, without being polled. This is necessary, that several devices running BlueOS can communicate through the networking system.

Currently, the following protocols are implemented in the BlueOS:

- Network devices with IP[2]
  - SLIP[3]
- Higher level protocols
  - TCP[4]
  - UDP[5]
  - ICMP[6]

3.4.1.1 Networking Task

In the BlueOS all networking protocols are handled by one central networking task (`blueOsSocketsTask`). The reason to handle all protocols by one task is to reduce memory usage for task stacks and communication buffers. This task has a large stack size of 256 Byte by default, because some protocols need that stack size for calculations. If a larger stack size is needed, the constant `TASKSOCKETS_STACKSIZE` can be adjusted.

As soon as the networking task starts, it initializes the necessary networking devices. After the initialization, the main loop begins (see Figure 3.1). This loop

[2] Internet Protocol
[3] Serial Line Internet Protocol
[4] Transmission Control Protocol
[5] User Datagram Protocol
[6] Internet Control Message Protocol

consists of three steps. Firstly, the task is waiting for data. Therefore there is a signal, that is used by all networking devices to signalize that there was data received, or that there is active data to send. In the second step, all received data is processed, by the networking stack. Thirdly all active data is sent. All processed data is passed to the transmission routines of the networking devices. The networking stack automatically selects the correct networking device to send its data. With this design, the task only processes one packet at the same time, this keeps the memory usage low.

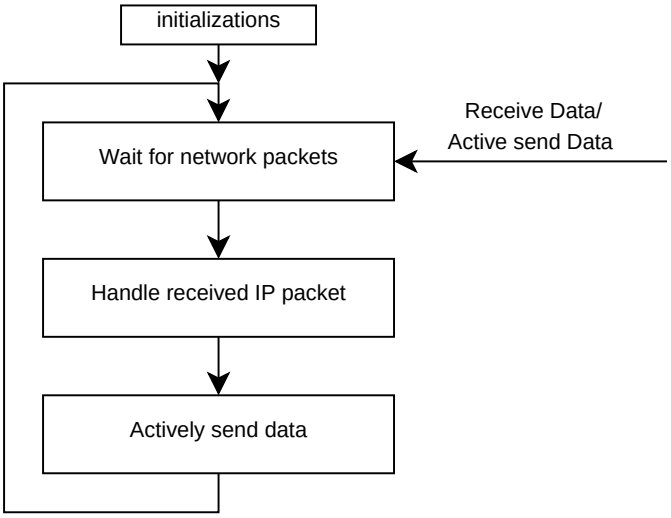

FIGURE 3.1: Principle of networking task operation

3.4.1.2 Networking Buffer

Network packets are processed like an production line. Therefore, the BlueOS has several networking buffers, which are stored in a global pool. This buffer is designed, to store a complete IP-packet. The idea is, that every module that currently uses such a buffer can use it alone without accesses of other modules at the same time.

So a networking buffer has a life cycle during its processing (see Figure 3.2). In the first step, a networking device gets an empty buffer of the global pool to receive data . The networking device uses this buffer, as long as the receive process takes. As soon as the data is received, the networking device enqueues the buffer to the networking pool for received IP-packets. Now the networking device must not care about the protocol processing and can again get a new empty buffer, to receive

the next packet. This solution guarantees a maximum efficiency of the networking device (Step 1&2 in Figure 3.2).

All received packets are processed by the networking task, so the higher level protocols are independent of the networking device. In some case the protocol may send an answer to the received packet, so the buffer, which was used to receive data is then used to transmit the response to that packet. The response is generated by the networking task. After that the buffer is enqueued to the correct networking device which is the same on which this packet was received. If active data is to send, the task gets a new empty buffer and enqueues it to the best networking device (Step 7&4 in Figure 3.2).

In case that there is no response to send to the packet, or the networking device has completely sent the data of a networking buffer, the used buffer gets stored back into the empty buffer pool (Step 5&6 in Figure 3.2).

To secure that the networking devices always can acquire buffers to receive data, there should be enough networking buffers in the system. At least there should be one for receiving and one for sending for each networking device, also there should be one buffer which can be processed by the networking task. However it is advisable, to initialize the system with some more networking buffers, because each sending queue of a networking device can store multiple buffers. In case of active sending most buffers are waiting in a sending queue, so there are less buffers free to receive data.

The amount of networking buffers inside the system is defined by `BLUE_OS_NETWORKING_BUFFER_C`
For simplicity, all networking buffers are of the same size, which can be configured by the define `BLUE_OS_NETWORKING_BUFFER_SIZE`. This buffer size must be large enough to store a complete IP packet. If however the received packet is larger than the buffer, the networking device must set the overflow flag that the networking task knows that is not complete.

3.4.2 Creating an own networking device

In the BlueOS networking system, a networking device must implement the supplied interface. The only function that needs to be implemented is a function to store networking buffers in the sending queue, this function is called by the networking task. It has the following structure: `void sendFunction(struct BlueOsNetworkingBu`

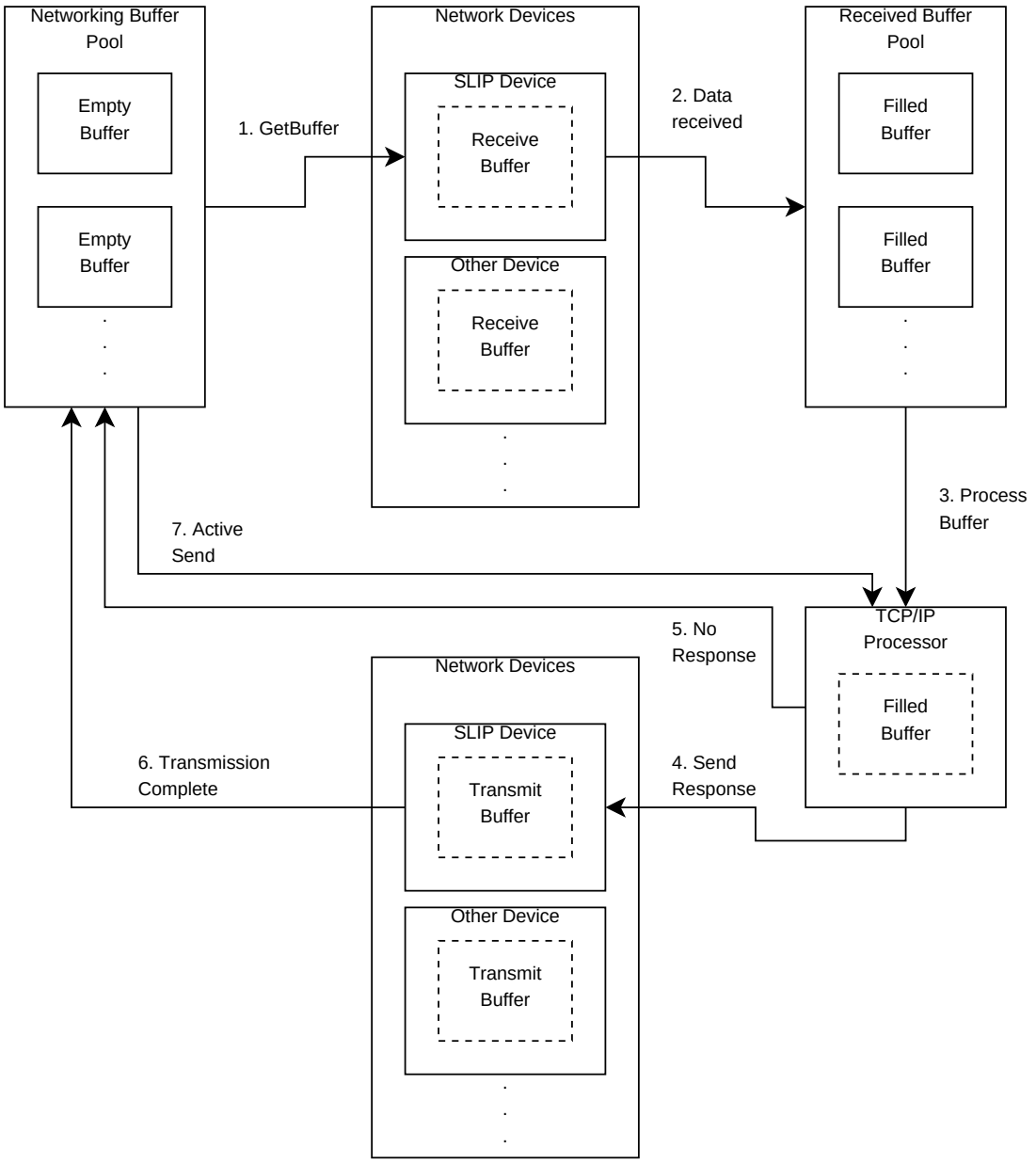

FIGURE 3.2: Life cycle of a networking buffer

When data is sent, networking buffers must be taken of this queue. The queue itself must be implemented by the networking device, therefore the networking buffer has a `next`-pointer, so it can be stored as linked list.

To implement the receiving routines, no interface must be implemented. This routines can acquire an empty networking buffer, by calling the function `blueOsNetworkingBufferPo` As soon as one IP-packet was received, the received networking buffer gets enqueued to the networking task by calling the function `blueOsSocketsReceivedData(...)`. After this function call, this buffer must not be used by the receiving routines, to receive a new packet, also a new networking buffer must be acquired.

To register the networking device in the system, a `BlueOsNetworkingDevice` structure must be created. Inside this structure, the `sendFunction` must be registered into the entry `enqueueFunction`. Also the IP-address and the subnetmask must be configured inside this structure. By calling the function `blueOsNetworkingAddDevice(.` the device gets registed into the system.

3.4.3 Serial Line Internet Protocol (SLIP)

SLIP is a very simple protocol, that is used, to transmit IP-packets over a serial line, it is described in the RFC 1055 (see [3]). The idea behind the protocol is that the IP-packet is simply send through the serial line, but there an end symbol (value is 192), which marks the packet. If there is actually a byte with the value 192, this byte is replaced by an escape sequence. An escape sequence is started with a byte value of 219, so the byte 219 is replaced by an escape sequence too. If the escape sequence is started, the next byte decides if it represents byte 192 (represented as 219, 220) or if it represents 219 represented as 219, 221).

There is already a reference implementation for this protocol. This code was modified and added to the BlueOS. It sends an end symbol at the start of the packet, to flush data, that was not correctly transmitted, so the new packed is received correctly. The implementation writes and reads data from UART. Therefore it uses the blocking read function, because the receive function will only return, if a complete packet was received.

To make it possible, to connect to a Windows SLIP server, or being a SLIP server for Windows clients, there special routines for server and client cases were created. The function `void blueOsSlipStartClient()` writes the string `"CLIENT"` to the serial line and waits, until the server answers with `"CLIENTSERVER"`, which a Windows server sends. If another server is contacted, this string won't be sent, so this function also returns, if the end symbol is received. On the other hand the function `void blueOsSlipStartServer()` starts the SLIP server in listening mode. It waits, until the string `"CLIENT"` was received and answers with the string `"CLIENTSERVER"`. However, if the end symbol is received, which is the case, when another client is connecting, this function is not returning a string and simply returning.

3.4.4 Internet Protocol (IP)

Packets, that are received or transmitted by the MAC[7] layer, where SLIP is located, can only be IP packets in the BlueOS. As this is the most common protocol that is used worldwide, there is no need of other protocols in this layer. To keep the code size small, only IPv4[8] is implemented and also doesn't have the full functionality.

3.4.4.1 Limitations

BlueOS serves the most important features of the IP, to enable a simple communication. What is not supported is packet fragmentation, type of service, flags and IP-options.

3.4.5 Internet Control Message Protocol (ICMP)

A simple version of the ICMP is implemented in the BlueOS. However this implementation is only for debugging and testing, so it only provides ping responses, to measure communication timings or to check, if the system is still running. This implementation does not validate incoming packets. It just checks, if the incoming packet has the ICMP-type 8, which is ping request, and then it sends the correct response.

3.4.6 Transmission Control Protocol (TCP)

The most interesting protocol which BlueOS provides is the TCP. Also this implementation is optimized for memory usage and speed. Of course TCP is not designed to transfer realtime data, it just secures that data is transfered completely and in the correct order. As TCP is designed to transfer streams instead of single packets, tasks access TCP sockets via queues, which are provided by the BlueOS. This simplifies the usage of the protocol.

[7] Media Access Control
[8] Internet Protocol Version 4

3.4.6.1 TCP Communication

When the BlueOS is starting and TCP is enabled, it calls the initialization function `void blueOsSocketsTcpInit()`. This function simply initializes some internal signals, that are needed that sending and receiving data does not need to wait actively.

If a TCP communication is wanted, the first thing to do is to create a socket, which can be done by the function `int8_t blueOsSocketsTcpCreate(...)` This function initializes the needed data structures for this socket. For this function it is important, that a receive buffer and a transmit buffer is passed. These memory segments are used to create the communication queues, that are needed for the communication. The queues, that are created are only half-blocking, that means that the receive queue only blocks on dequeueing and the sending queue only blocks on enqueueing.

The enqueueing in the receive queue must not block, because this is done by an ISR, in which interrupts are disabled. Blocking a task in this state could cause the stack of this task to overflow, if it's not large enough. Also the dequeueing of the sending queue must not block, because it is read out by the central networking task. If this task would block, no networking data could be transfered anymore. So, the networking task only sends data, that is currently in the sending queue.

As soon as the socket is initialized, there are two possibilities, either open the socket in listening mode, or actively connect to a remote socket. With the function `int8_t blueOsSocketsTcpListen(...)` the socket is opened in listening mode. At first, this function checks, if it is possible to open the given port. If all checks are valid, the socket is added to the list of open sockets and the task which opened the socket sleeps until a remote user connected to this port. If this function returns 0, then the connection is established and can now be used for communication.

To actively connect to a remote socket, there is the function `int8_t blueOsSocketsTcpConnect(...)`. Also this function checks, if the specified local port can be opened. If all checks are valid, this function sets the connect-flag in the socket structure. If this flag is set, the networking task actively sends a TCP-connect packet to the remote side. The user task then sleeps, until the connection is established.

There is also the possibility to configure the socket, this can be done by the function `int8_t blueOsSocketsConfigure(...)`. However there is currently only one configuration option. With the option `BLUE_OS_TCP_SEND_IMMEDIATELY` all data in the send queue will be send immediately. If this flag is not set, it will only be sent, if it is polled.

To communicate through an established connection, the function `int8_t blueOsSocketsGetStream(...)` can be used, to get pointers to the sending and receiving queue. For sending, simply data has to be enqueued to the sending queue. For receiving data, data can simply be dequeued of the receiving queue.

To close the connection, the function `int8_t blueOsSocketsTcpClose(...)` must be called. This function simply sets the close-flag of the socket data structure, which will lead the networking task to do the complete close procedure. This function returns immediately and does not wait until the connection is closed.

3.4.6.2 The TCP State Machine

The TCP standard has a complex state machine, that is needed, to establish connections and to close connections. Originally every socket has 11 states, but this state machine is optimized a bit, because only a very simple TCP is needed, so this implementation only hast 10 states (see Figure 3.3). The state of a socket is manipulated by several internal networking functions. In comparison to RFC 703 (see [4] page 23), this state machine merged the states "CLOSE WAIT" and "LAST-ACK", to simplify the implementation. Also there is no timeout of 2 MSL[9] in the state "TIME WAIT", sockets in this state act like closed sockets. Additionally to RFC 793, there is a connection from "FIN WAIT 1" to "TIMED WAIT", to speed up the closing procedure.

3.4.6.3 TCP Options

This very limited implementation of the TCP stack provides a few TCP options, listed here:

[9] Maximum Segment Lifetime

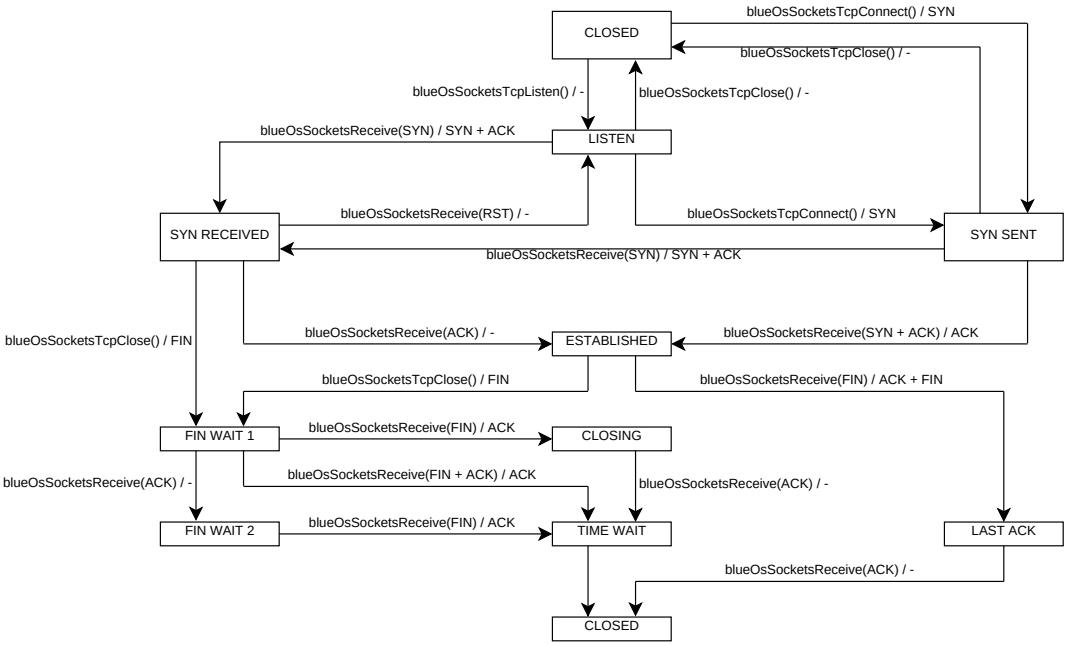

FIGURE 3.3: TCP Socket State Machine

TCP Option	Description
MSS	Maximum Segment Size. This option is configured while the connection is being established. It tells the remote socket, how large the maximum TCP packet is, that fits into the receiving buffer. This size is only TCP without IP headers and other headers and trailers!

3.4.6.4 Limitations

This implementation of the TCP is very simple, so it uses not much program memory and not much data memory. It can be used for simple communication only. For example, there is no flow control, so the tasks should be designed not to send too much data. Of course buffers can not overflow. Also this communication is not very fast, because of checksum calculations and other things, it should only be used for noncritical applications.

3.4.7 User Datagram Protocol (UDP)

Another protocol the BlueOS is providing, is the UDP. This protocol is defined in RFC 768 (see [5]). It is used for unsecured, packet-orientated communication, so packets can be lost or can arrive in different order. As this is a very simple protocol, it is completely implemented in the BlueOS.

3.4.7.1 UDP Communication

When the BlueOS is starting, it initializes the UDP system. Therefore only one signal is initialized, that is used to communicate with network devices.

As soon as the BlueOS is started, the user tasks can create UDP sockets. To do that, the function `int8_t blueOsSocketsUdpCreate(...)` is used. This function simply registers the receive buffer and send buffer of a task. The buffers provided to this function must be large enough, to fit the largest UDP data segment, that can be expected for the application. At this point should be mentioned that only one single packet is stored in the buffer. If the buffer is not read out fast enough, messages will be lost, e.g. only the last received packet is stored in the receive buffer, so it is the most actual one.

When the socket is initialized, a port can be opened for receiving data, by using the function `int8_t blueOsSocketsUdpOpen(...)`. This function checks, if the port can be opened and if all checks are valid, the port is registered in the open socket list. After that, data can be received on the specified port.

After opening the port, data can be exchanged. To receive data, the function `int16_t blueOsSocketsUdpRead(...)` can be used. This function checks, if the socket is open and if there is data in the receive buffer. If there was data received, it is copied into the specified buffer and the important informations of the sender are returned, e.g. IP address and remote port. To send data, the function `int16_t blueOsSocketsUdpWrite(...)` is used. Also this function checks if the socket is open and it checks, if there is still a sending process pending. If the send buffer is clear, the specified data is copied into the sending buffer, it will be send by the networking device the next time, the networking task is in active sending mode.

At the end of the communication, the socket can be closed by using the function `int8_t blueOsSocketsUdpClose(...)`. This function simply removes the socket from the open port list.

Chapter 4

License

Chapter 5

Limits

The BlueOS is not perfect. There are some issues, that have not been solved, yet:

- Using precompiled external libraries causes problems with the compiler, e.g. instructions are executed in the wrong order.
- Only one UART can be used as shell at the same time.
- Multimaster TWI is not supported due to ATmegas hardware issues
- There is no efficient way to abstract all UARTs with queues by default.

Chapter 6

Conclusion

We started this project because there were no free preemptive operating system available which would have matched our needs. Our first intention was to switch between several tasks. So we used a timer ISR to bend the stack. We realized that this relativ small ISR code represents a full preemptive kernel. We decided to continue this project. The development of the kernel itself was done in a few hours. Of course it had some bugs but in 99% of all cases it worked as it should. Exactly this last percent was the big problem. We experienced that the best design could have issues or implementation errors. Of course we made test cases for our implementation to reduce the errors. But unfortunately these cases hasn't covered constellations which we would never produce because of our own coding style. Therefore the blueOs had to be tested by a larger number of developers. Furtunately we had a lectureship in a laboratory for embedded system while developing the blueOs. The students in that course were the best test developers we could get. They have discovered a lot of errors and have mentioned a lot of enhancements. Over the time the blueOs was upgraded with more and more functionality. Simultaneously it becam evenmore stable. At this point we can say that our blueOs has grown up to powerful operating system. We implemented the common OS paradigms for synchronization and task model. So our system could serve nearly the same (basic) functionality as other more famous operating system like linux. But the development is not finished, there are still lots of ideas which could be implemented. Concluding it was a very interesting team project which has increased our skills about operating system enormously.

Chapter 7

Prospects

In future releases, support for further AVR devices will be added. Especially the ATxmega support will be advanced, because these devices have a lot more functionality.

Also there will be a kind of file system, that will allow it, to output the shell to any peripheral device. This would also allow to have multiple terminals.

Chapter 8

BlueRider - Software

8.1 Interface-Controller

8.1.1 Concept

Against the to the version 2, the new BlueRider has 4 controllers. The 4th controller is intended to work as the main interface to the external world. It is directly connected to bluetooth module. As a special feature it has a JTAG connection to each other controller. The software is almost a full Atmel JTAG MKII clone, therefore it can be used for programming and debugging. In addition it can forward the connected UARTs from every controller.

8.1.2 Manual

The Interface Controller GUI can be displayed with a VT100 terminal. It shows the states from each controller and voltages. The first voltage is the Internal voltage which can be measured internally. The second voltage is the External voltage which is measure at the capacitor parallel to the voltage regulator's input. While starting the GUI doesn't wait for a connected terminal, therefore you have press F4 in order to redraw the menu. You can select a controller by pressing the UP and DOWN keys. If ALL is selected the action is performed on each controller.

You can switch on the UART from another controller by pressing a F# key. Then the whole UART traffic is forwarded to the selected XMEGA. But the traffic is

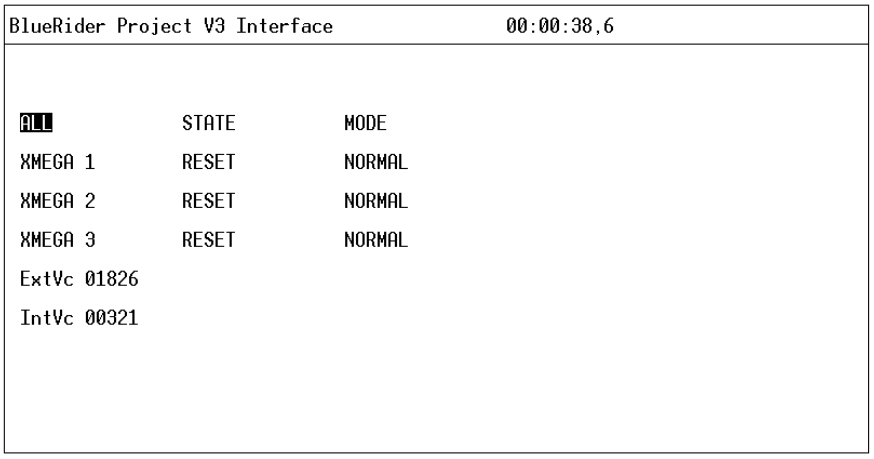

FIGURE 8.1: Interface Controller GUI

key	action	description
r	run	release controller from reset
q	stop	reset controller
p	program	starts the iHex programmer over XMODEM

TABLE 8.1: Interface actions

still monitored by the Interface controller to detect the F4 key, which will return to the interface controller's gui. Therefore the F4 key cannot be used by the other UARTs.

key	controller
F1	XMEGA 1
F2	XMEGA 2
F3	XMEGA 3
F4	XMEGA 4 (Interface Controller)

TABLE 8.2: Interface UARTS

8.1.3 Components

8.1.3.1 JTAG Implementation

The 4th controller has JTAG connnections to the other controllers. The JTAG interface is a common interface, mainly intended to test electrical circuits. It is standardized as IEEE 1149.1. The JTAG standard defines only the BoundaryScan capability. With boundary scans, a logic cell can be read out and manipulated. Therefore its easy to find a broken conductor path or a faulty soldered point,

because every pin of the device can be captured. The JTAG port consists of four wires, similar to the SPI Interface. There are wires for clock, dataIn, dataOut and testModeSelect. The last one would be the select wire from the SPI but it has more functionality. These four pins are also called test access port (TAP). Behind the interface, there is a state machine called TAP-controller.

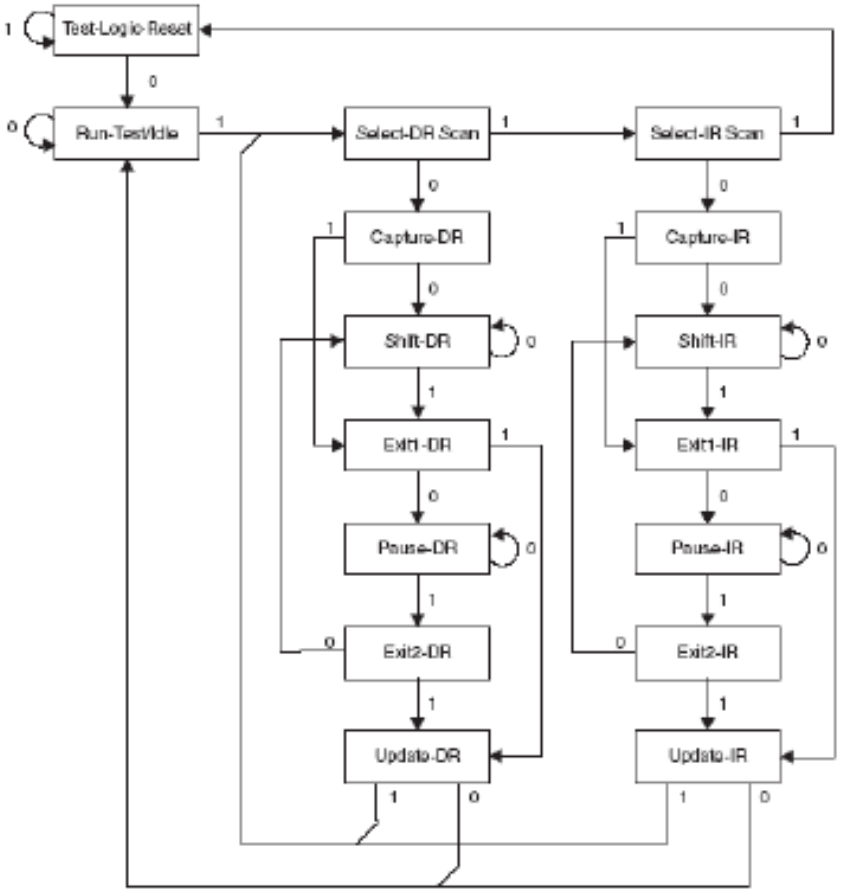

FIGURE 8.2: TAP Controller state diagram [1]

As you can see, there are mainly only two parts, one for the instruction register(IR) and another for data registers (DR). To fill or read them, there are shift states in the state machine. Normally the internal instruction register is 4bit wide at least for the XMEGA. The data register width is dependend on the executing instruction. While shifting read and write happens simultaneously. Therefore the read and write functions can be summarized to one. The implematation is completely made in software. It consists of a read/write-function an some others to manipulate the state machine.

Although the JTAG interface was originally developed to make boundary scans, the chip manufactors extended its capabilities to two other features, programming and debugging. Unfortunately there is no standard for them. Every chip or chip-series has its own implementation. Beside the standard instruction, standardized by IEEE 1149.1, the manufactors had to simply extend the JTAG instruction set.

8.1.3.2 PDI Implementation

The new XMEGA series from Atmel provides a JTAG and their proprietery PDI interface. The Program and Debug Interface (PDI) is a two pin interface, designed for external programming and on chip debuging. It communicates with a serial protocol on the data pin. The second pin is used for the clock signal. In principle it is a half-duplex synchronous UART. It uses 8 data bits, one even parity and two stopbits. Because of the half-duplex mode, there has to be a mechanism to avoid drive contentions at the data line. The pdi and the programmer have to take care that they don't drive the data line at the same time. Although the XMEGA supports a special bus-keeper mode, which would solve this problem, we looked for another way to connect to the PDI. Beside the hardware PDI, there is another possibility to connect to the PDI-controller. Atmel has extended their JTAG instruction. The new instruction is called PDICOM. While this instruction is seleted in the JTAG instruction register, the JTAG data register is connected to the 9bit pdi communication register. So we could connect via our JTAG implementation.

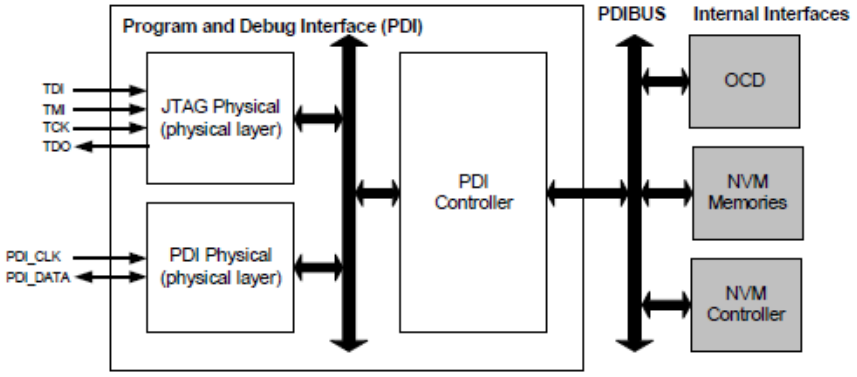

FIGURE 8.3: The PDI with JTAG and PDI physical and closely related modules (grey)[1]

The PDI-Bus is connected to all memories within the XMEGA including the SRAM space. Therefore it is able to control every component of the XMEGA.

The NVM-controller is reponsible for programming and reading the XMEGA's memories. Atmel has a very good documentation about programming. Therefore it was easy to implement the programming functions. First of all the PDI interface must be initialized with a special 8 byte NVM key. After that, all functions of the NVM-controller are ready.

The OCD-Controller is responsible for on-chip-debugging. Unfortunately there is no documentation about the OCD-System. An essential part of the new BlueRider, is the debug capability. Therefore we had to get these information on other ways. Our approach was to sniff the protocol, while an original Atmel JTAG MkII was working. Unfortunately we had no Logic-Analyser, so we had to built our own Analyser with an XMEGA device. The PDI works very fast with at least 1Mhz clock speed. Additionally we decided to sniff PDI over JTAG. The normal clock speed from the XMEGA is too slow for to capture the data. Therefore we had to overclock it. We figured out that our XMEGA works stabil until 48Mhz. With these speed we were able to record the traffic. We had no possibility to transfer the recorded traffic over UART because it is too fast. Therefore we had to save it in the local SRAM. But 8kByte are exceeded very fast. So we decided to save only the high level PDI instructions. We implemented a protocol decoder which filters the JTAG overhead. At this point we could record small protocol snippets from the traffic. It was still impossible to save a whole debug session into the local SRAM. But with the sniffed protocol snippets we were able to reconstruct large parts of the Protocol. In order to verify our reconstructed protocol, we implemented the JTAG MkII protocol. This protocol is completely published by Atmel. After that we could connect our clone directly to the AVR studio. Our reconstruction is working quite good. Main features like RUN, STOP, RUN-TO and several Breakpoints are working fine. We were a little bit unhappy because of the necesity of reverse engineering but it is completely impossible to integrate an original JTAG MKII into the BlueRider. At this point the sniffed protocol will not be explained further, because our intention was to create a debug platform within the BlueRider and not to violate any copyrights of Atmel.

At this point the debug capability is ported to the Interface controller. The MkII protocol is not ported, because our new UART speed is not supported by the AVR-Studio. Therefore the debug functions within the interface controller are not tested.

8.1.3.3 iHex decoder

We ported the iHex decoder implementation from BlueRider Bootloader implementation. Unfortunately after programming the last page was always corrupted. We assumed that page borders are never within an iHex-frame. So we checked the necesity for page writes only between the iHex frames. But we figured out that the "'avr-objcopy.exe"' always pastes a shorter iHex frame near to the end of file. This produces a page change whithin the iHex frame. We hope that this behavior is xmega dependend. But it could explain a lot of strange errors which occured while working with the BlueOs. Because the BlueRider Bootloader was used in all cases.

8.1.3.4 XMODEM

The XMODEM implementation from the BlueRider Bootloader was ported to the interface controller. The bootloader has used three ISR's two manage the XMODEM transfer. This was reduced to one task. The task is designed as a dynamic task, therefore the memory management must be activated. The XMODEM supports both normal and 1k-mode. It is designed to fit in the blueOs as an module for later use.

8.2 Main Controller

The main controller is reponsible for the motor control. It is connected to the engine control. It offers a small GUI which displays all variables of the BlueRider system. It produces the PWM's for the motor and the brake.

8.3 Sensor Controller

The sensor controller is ported from BlueRider v2. Nearly the whole code could be reused. Now it's working within the BlueOs, that means the old code has been distributed to several tasks.

Chapter 9

BlueRider - Hardware

9.1 Overview

Subsequent to the project BlueRider V2, there were a lot of ideas to upgrade the hardware, that was used there. Because the old system has very small micro controllers, the new idea was to use the new ATxmega controllers, which are more powerful than the old ATmega devices. Also the new hardware should fit better into the Carrera car, so it would be easier to build up BlueRider cars.

9.2 Hardware Description

The new hardware again consists of two circuit boards, one for engine control and one for the micro controllers. However now there is also planned a third board, which will contain all necessary sensors and sensor connections. All boards can be stacked via multi-pin connectors and the bottom board can be mounted into the Carrera car.

9.2.1 I/O - Board

The I/O board is the bottom board of the stack. It has the same size, as the original Carrera Digital hardware, so it can be mounted into the Carrera car easily. This circuit contains the power supply unit, which in the first instance secures the

electricity direction. The first instance is unsecured power, which is used, to drive the engine. The second instance is separated with a diode and stabilizes the power with a large capacitor, which is already used in version 2 of the BlueRider. The second instance also generates the 3.3V supply voltage for the logic. In comparison to version 2, this hardware uses other diodes, that have higher peak currents and a lower forward voltage, so this hardware works more efficiently.

Also this board has the engine control unit. The part to drive the engine is exactly the same as in version 2, it is simply a p-channel MOSFET (IRFR9024N) which is driven by a transistor. Different to the version 2 is the brake functionality. The new hardware also brakes the car by shorting the engine. However this is now done by a n-channel MOSFET (IRFR024N), which can drive large currents. This unit also has a logic, that prevents the hardware to shortcut and power the engine. Differently to version 2, now the brake signal is dominant over the drive signal, so the car can be stopped more easily.

As this is the bottom board of the stack, this board must contain the infra red communication system, that was developed in another project. This contains of an infrared LED, which is used to control the turnouts and the infrared receiver. The receiver was not changed to the bachelor thesis. It contains two infrared diodes, which are on the bottom of the car. This diodes generate a very low current, if they are illuminated, this current is amplified by an impedance amplifier. The amplified signal is now inverted and not clean, so there is a Schmitt trigger that inverts the signal and cleans the signal. The dimensioning of the parts was calculated in the bachelor thesis.

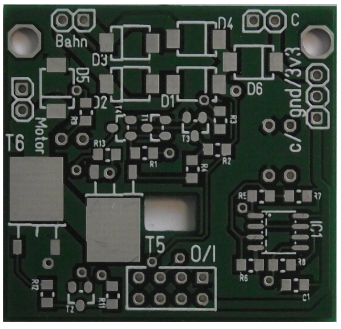

FIGURE 9.1: I/O Board Side 1

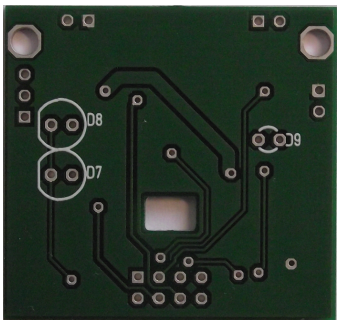

FIGURE 9.2: I/O Board Side 2

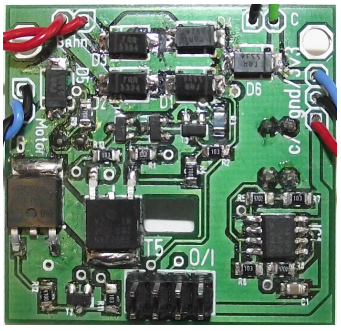

FIGURE 9.3: Populated I/O Board Side 1

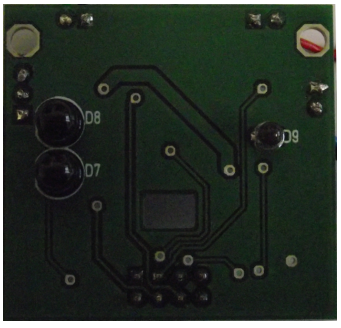

FIGURE 9.4: Populated I/O Board Side 2

9.2.2 Mainboard

The Mainboard is the middle board in the stack, it contains a connector, to connect to the bottom board and several connectors, to connect to the top board, which will connect sensors. This board is completely new designed. It now contains 4 ATxmega128 controllers, which are way more powerful than the controllers in version 2. There are 3 controllers, that do the same, as in version 2, and a new controller:

- Controller 1 (IC1) is the pilot controller, that can be programmed by the students.
- Controller 2 (IC2) is the main controller of the car, it controls the engine and brake signals.
- Controller 3 (IC3) is the sensor controller, it controls the infrared communication and odometry and other sensors.
- Controller 4 (IC3) is a master controller. It can reset all other controllers and has the Bluetooth connection. This controller is used to debug the other controllers. So this controller has JTAG connections to all other controllers.

Special attention was given to the wiring of the controllers, to guarantee a successful operation of all controllers, so the advises of Atmel were realized. Therefore a lot of capacitors and inductors were placed on the other side of the board.

In version 2 of the board it was a big problem that there was no baudrate clock used, so this board contains one baudrate clock (Q2) and one maximum clock (Q1). Every controller has a clock selection pad on the board, where the clock can be selected by shortcutting the corresponding pads. However, the usage of the new ATxmega controllers showed, that they have a very good internal clock and internal baudrate generator, so the controllers don't even need an external clock.

The version 2 of the BlueRider has a central IC bus, which connects all controllers. For compatibility the new board has this too, but it only can be used for slow data communication. So the new board also have UART connections to every other controller, because the ATxmega128 device has up to 8 UARTs. These connections can be used to directly fullduplex-communicate to every controller with high transfer rates.

To connect to the top board there are connected a lot of pins to the connectors, so there are A/D and D/A converters and serial communication pins, which can be connected at the top board.

As the used controllers have many pins and the board is very small and there are a lot of other very small components on this board, it was necessary to use 4-layer multilayer board, to successfully route all signals. The construction of the schematic and PCB layout was done with the Target 3001 editor.

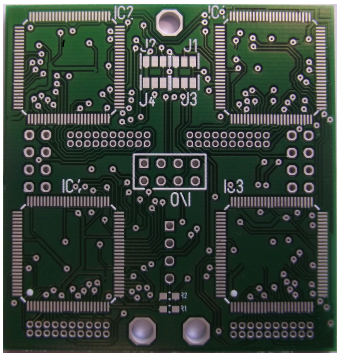

FIGURE 9.5: Mainboard Side 1

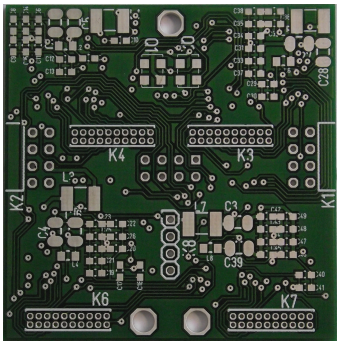

FIGURE 9.6: Mainboard Side 2

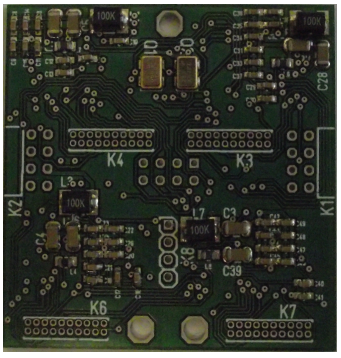

FIGURE 9.7: Populated Mainboard Side 1

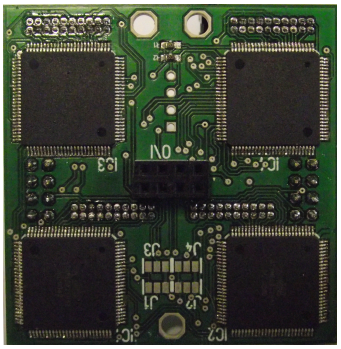

FIGURE 9.8: Populated Mainboard Side 2

9.2.3 Errata

9.2.3.1 I/O Board

Due to an error in the bachelor thesis, the impedance amplifier for the infrared signal is connected wrong, to correct this, the pins 5 and 6 of the operational amplifier (IC1) must be switched.

9.2.3.2 Mainboard

Unfortunately there was a fault in the Target 3001 database, so two pins of the ATxmega power supply are switched (pins 83 and 84). This results in a shortcut between the supply voltage and ground. To solve this Problem, the pins need to be removed of the board (see red markings on Figure 9.9).

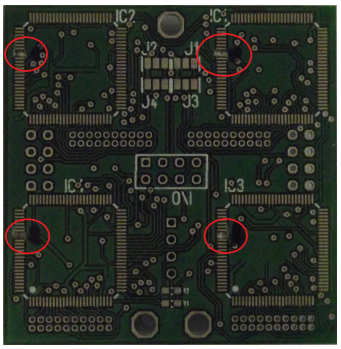

FIGURE 9.9: Mainboard that has removed the faulty pins

Bibliography

[1] Atmel. Xmega a manual. December 2009. URL http://www.atmel.com.

[2] Atmel. 8-bit avr microcontroller with 16/32/64k bytes in-system programmable flash. *atmel.com*, January 2010. URL http://www.atmel.com/dyn/resources/prod_documents/doc8011.pdf.

[3] J. Romkey. Rfc 1055 - nonstandard for transmission of ip datagrams over serial lines: Slip. *ietf.org*, June 1988. URL http://tools.ietf.org/html/rfc1055.

[4] University of Southern California. Rfc 793 - transmission control protocol. *ietf.org*, September 1981. URL http://tools.ietf.org/html/rfc793.

[5] J. Postel. Rfc 768 - user datagram protocol. *ietf.org*, August 1980. URL http://tools.ietf.org/html/rfc768.