Lab Protocol

Code mobility in Networked Embedded System

NES

Group 4

abstract: The lab protocof contains the final project documentation. We present the introductory part of the project and all necessary organization details in the chapter 1. The requirements are stated in chapter 2. Chapter 3 provides the reader with unambiguous specification, Implementation details are represented in chapter 4.

January 29, 2013

Igor Pelesić, Matrikelnumber 0006828
Konstantin Selyunin, Matrikelnumber 1228206
Miljenko Jakovljević, Matrikelnumber 0426673

Contents

1 Project Outline

1.1 Organization

The roles and responsibilities for the project are represented as follows:

- **Project manager:** Konstantin Selyunin [S]
  - Defining tasks
  - Internal organization
  - Control meeting deadlines
  - Agent assember language: Development and Implementation
  - Adaptation of drivers
- **System architect:** Igor Pelesić [P]
  - Defining and reviewing technical aspects
  - Designing communication protocol
  - Adaptation of drivers
  - Platform: Design and Implementation
- **Zigbee communication:** Miljenko Jakovljević [J]
  - Designing board-to-board communication using zigbee
  - Presentation for workshop 1: communication part

1.2 Project Description

The purpose of the project is to design, implement and evaluate code mobility platform on Embedded system engineering board [2]. Our goal is to develop the system that allows users to build and execute simple agent program on top of hardware ESE platform. To achieve the goal we have developed three layered software: agent layer, platform layer, communication layer. Our goal is to show that code mobility concepts that are successfully used on much higher abstraction level are applicable for the embedded applications. During the project we have developed and implemented infrastructure that allows developer of agent program do not be aware of the hardware services presented on the given platform.

1.3 Definitions, Acronyms, and Abbreviations

By *code mobility* we mean the capability of code to change the location where it is executed.

Strong code mobility is the ability to allow migration of both code and execution state to the destination, *weak* code mobility allows code transfer but it does not involve the transfer of the execution state.

Platform is a component that provides corresponding hardware services to

1.4 Background

The research has been done to use code mobility in distributed environment [1] and various application has been developed including [3] web application platform that allows people without major programming experience to develop the application as work-flow specification in graphical form. The use of code mobility is to "move the knowledge close to the resources" [4] and enable higher flexibility of accessing remote resources.

1.5 Workpackages

In the following section we describe workpackage deliverables for our project:

1.5.1 WP1 Documentation

1.5.1.1 Requirements and Specification

Before the development and implementation, clear requirements should be defined. In this deliverable we define user roles, global requirements for the project, functional and non-functional requirements.

1.5.1.2 Presentation for Workshop 1

In the first workshop we introduce to the audience the general overview of our project and specification. For this deliverable we have done self-contained presentation, which inroduce all necessary concepts, our goals and approach. The goal for preparing the presentation is to convey a message of our project to audience, assuming no prior knowledge of code mobility concepts. We introduce milestones and time plan, as well as project management concepts to achieve the goal.

1.5.1.3 Presentation for Workshop 2

In the second workshop we present the results of our work. We do the test application for using the code mobility on the board. We discuss our major design decisions that have beed made during the design and implementation phases.

1.5.1.4 Lab protocol

The lab protocol will consist of outline of our project, the requirements and specification for the project. In addition it contains precision description of Agent language, low-level assember-like language that support code mobility syntax. Desciption of the API and structure of our software.

Workpackage 1. Documentation			
Responsible:	Konstantin Selyunin, Igor Pelesić	Start date:	07.11.2012
Deliverables:	D1.1 Requirements and Specification	Finish date:	28.01.2013
	D1.2 Presentation for Workshop 1	Estimated Effort:	180 hours
	D1.3 Presentation for Workshop 2	Interdependencies:	all
	D1.4 Lab protocol		

1.5.2 WP2 Adaptation of drivers

The platform will provide access to hardware for mobile agents. During this deliverable drivers for the following peripherials should be adapted or otherwise implemented:

1. **Bargraph:** Port A of nodes 0 and 1 is connected to the led bargraph. The driver should display encoded in binary number a value from the range `0 ... 255` on the bargraph.

2. **Heater:** Two heating resistors on the node 2 could be controlled by PWM signal. Driver that provide setting a duty cycle should be implemented. To control PWM PIN of the microcontroller timers should be configured and appropriate mode of the PWM should be selected.

3. **Cooler:** The cooling fan is also controlled by PWM signal. The same aproach as for the heating should be used here. Controlling the speed of the fan should be done by setting up the duty cycle of the PWM signal.

4. **Temperature sensor:** Three temperature sensors are connected to the bottom of the sink with I2C interface. The driver should read data from all sensors and return the average.

5. **Led matrix display:** Led matrix display with 6 segments of 5 by 7 each is connected to the node 3. The driver should provide API for writing single character and arrays of characters to the led matrix.

6. **TFT display:** Node 2 is connected to 640 by 360 TFT display. The driver should provide the following capabilities: set the cursor to the position on the display, set font and background colors and print arrays of characters on the display.

Workpackage 2. Adaptation of drivers			
Responsible:	Igor Pelesić , Konstantin Selyunin	Start date:	15.11.2012
Deliverables:	D2.1 driver implementation	Finish date:	12.12.2012
		Estimated Effort:	50 hours
		Interdependencies:	

1.5.3 WP3 Agent language tool

To design mobile agents special language that supports constructs for mobility is required. In this deliverable we design and implement the low-level assember-like language. The Agent language should provide access to the hardware as well as have syntax for expressing code-mobility concepts.

Workpackage 3. Agent language tools			
Responsible:	Konstantin Selyunin	Start date:	06.12.2012
Deliverables:	D3.1 agent language tool	Finish date:	21.12.2012
		Estimated Effort:	40 hours
		Interdependencies:	D1.1

1.5.4 WP4 Platform Communication

Protocol needs to provide environment for communication between platforms and transfering code. During this deliverable communication protocol that fulfil aforementioned requirements should be implemented. The main purpose of the project is to implement main code mobility concepts so we do not restrict ourselves to fulfil real-time requirements. CSMA/CA protocol will suit for our purpose, so we propose to implement communication using this protocol. One of the main goals for possible future work is to make agents and message transfer real-time.

Workpackage 4. Communication			
Responsible:	Igor Pelesić	Start date:	10.12.2012
Deliverables:	D4.1 CSMA/CA communication protocol	Finish date:	21.12.2012
		Estimated Effort:	60 hours
		Interdependencies:	D1.1

1.5.5 WP5 Platform

Platform supports concurrent execution of mobile agents as well as provides means for transfering agent code and messages. The main challenges in this deliverable are to implement priority based scheduler, execution layer and communication layer. It is of paramount importance that each platform support only hardware that is physically connected to dedicated μC, to save memory. It should be done during compile time.

Workpackage 5. Platform			
Responsible:	Igor Pelesić	Start date:	21.12.2012
Deliverables:	D5.1 Platform. Scheduler	Finish date:	15.01.2013
	D5.2 Platform. Execution layer	Estimated Effort:	120 hours
	D5.3 Platform. Communication layer	Interdependencies:	

1.6 Milestones and timeplan

For successful completion of our project, the following deadlines should be met:

- 22.11.2012 Clear defined requirements and specification
- 06.12.2012 Workshop 1: presentation of project outline, specification and requirements. Discussion of challenges, possible fallacies and pitfalls.
- 15.12.2012 Avaliability of Agent language tool
- 21.12.2012 Completion of communication protocol (D4.1)
- 23.01.2013 Avaliability of platform (D5.1), completion of implementation work.
- 29.01.2013 Documentation of the work in the lab protocol
- 29.01.2013 Demo application for workshop 2.
- 31.01.2013 Workshop2: Presentation of results. (D1.3)

1.7 Gantt diagramm

To represent interdependencies between tasks and sequence of execution of all tasks in our project we use Gantt diagram.

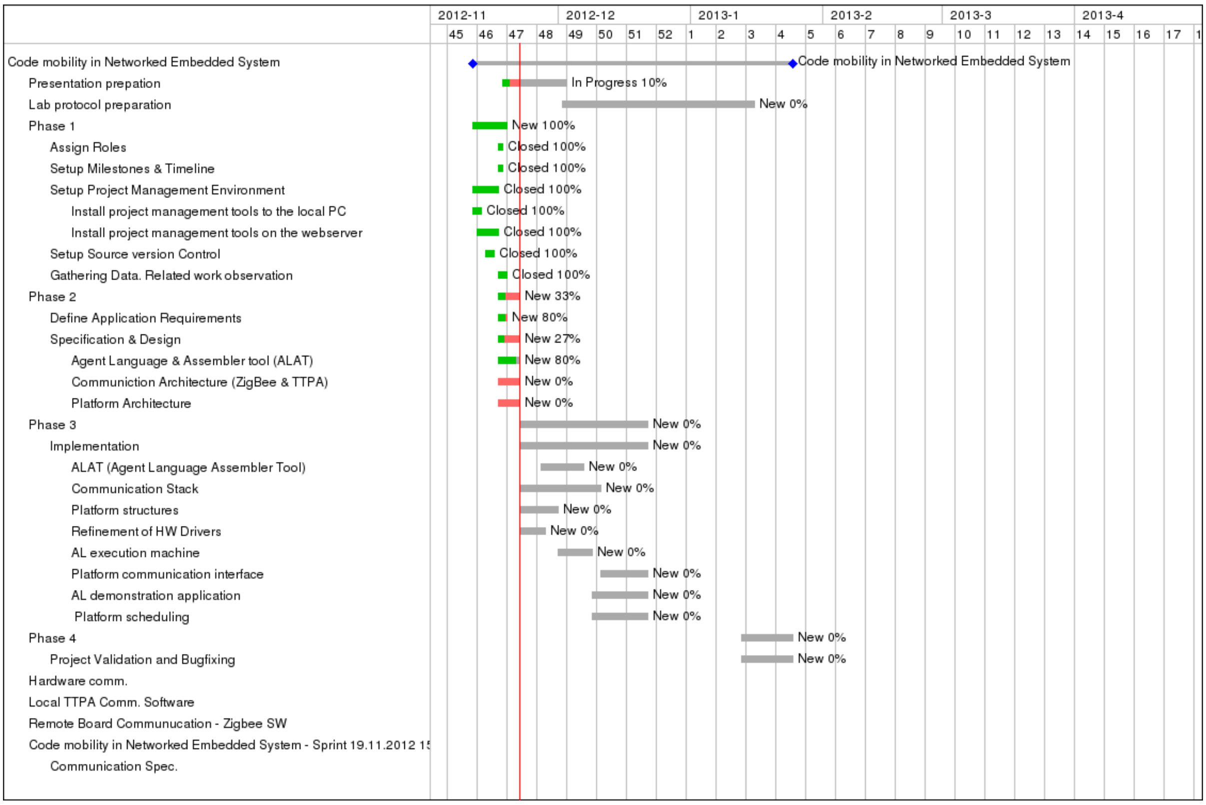

Figure 1.1: Gantt Diagram of the Project

2 Requirements

This section lists all requirements that should meet the project with respect to user roles of code mobility application. Defining requirements in a rigorous way will help us to exercise realistic validation scenarios.

2.1 User roles

R_UR_1 Application Developers (Tasks: Create control application in agent language, debug, test, prepare deployment packages)

R_UR_2 Application Consumers (Tasks: Deploy control application on target system, fill valuable bug reports)

R_UR_3 Plattform Developers (Tasks: Maintenance, Extensions, Porting to another target board)

R_UR_4 Application Designers (Tasks: Design control application)

2.2 Global Requirements:

2.2.1 Application Development requirements:

R_AD_1 The App Developer should be enabled to instantiate up to 4 agents on a single node, which are running concurrently.

R_AD_2 The App Developer should be allowed to configure the execution scheduling of the agents via a prioriatization of the agents.

R_AD_3 The platform should provide a simple agent programming language to the App Developer in which the agents of an application can be developed.

R_AD_4 The agent language should provide the App Developer with the possibility to reproduce its code on another node or on another board.

R_AD_5 The agent language should provide the App Developer with the possibility to communicate with another agent on the same board.

R_AD_6 The agent language should provide the App Developer with the possibility to access the node hardware.

R_AD_7 The agent language should provide the App Developer with the possibility to implement loops.

R_AD_8 The agent language should provide the App Developer with the possibility to compare variables.

R_AD_9 The agent language should provide the App Developer with the possibility to perform addition, subtraction, multiplication and division on variables.

R_AD_10 The agent language should provide the App Developer with the possibility to perform delays in the execution of code.

R_AD_11 The platform should allow debugging of agents executions.

R_AD_12 The platform should provide means for the creation of easily installable deployment packages.

2.2.2 Application Consumers requirements:

R_AC_1 The platform should provide means to deploy the agent software on the target boards easily.

R_AC_2 A tracing mechanism should be provided in order to ease the process of fault detection and to allow valuable bug descriptions.

2.2.3 Application Designers requirements:

R_A_DES_1 A description of the platform possibilities and limitations should be provided.

R_A_DES_2 The platform should provide means for reducing the overall complexity of a system, by allowing encapsulation of different tasks.

R_A_DES_3 The platform should provide configurable inter agent communication facilities.

R_A_DES_4 The platform should provide means to enable standby scenarios by allowing dynamical code reproduction.

R_A_DES_5 The platform should provide means for strong mobility, where an agent and its execution state are transferred to a new node or board and the execution on the new destination is started from the memorized state.

R_A_DES_6 A description of a platform should provide a list of all available services

2.3 Non-functional requirements

R_NF_1 The platform should be open to extensions i.e adding new hardware.

R_NF_2 The agent language should be extendable.

R_NF_3 Scalability

R_NF_4 Documentation

R_NF_5 A platform tracing mechanism should be provided which allows for more efficient bug-fixing.

2.4 Low-Level Requirements

2.4.1 Communication protocol

R_LL_CP_1 Protocol must provide means to avoid collisions on the bus

R_LL_CP_2 Protocol must provide means to check correctness of the data sent

2.4.2 Drivers

R_LL_DRV_1 Drivers shall deliver access for the platform to hardware by means of API

R_LL_DRV_2 The cooler driver must provide means to set up the duty cycle of the fan in range 0 (turn off) to 100 (full speed).

R_LL_DRV_3 The heater driver must provide functions to set the dissipated power of the heating resistors in range: 0 (turn off) to 100 (max power dissipation).

R_LL_DRV_4 Temperature driver must provide means to read temperature from all three sensors with precision of 1/8 of degree Celcius.

R_LL_DRV_5 Let matrix driver must provide means to display char arrays on the led indicators.

R_LL_DRV_6 TFT display driver must provide means to set background color of a display, position cursor to the desired location, set the font and background color and print array of characters on the display

3 Specification and design

3.1 General

The following figure depicts the general outline of the code mobility project.

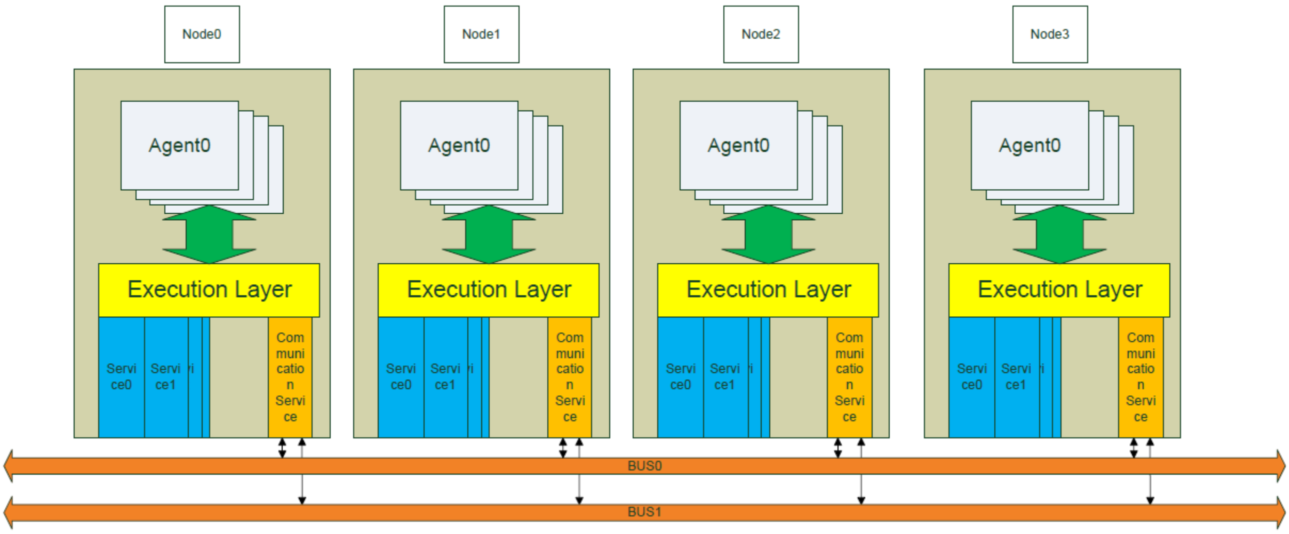

Figure 3.1: Overview

On each of the 4 nodes, which can be found on the ESE board, a virtualization platform will be deployed. This virtualization platform will be able to execute up to 4 agents concurrently. The agents will be programmed in a simplistic assembler like agent language. On the platform there will be an execution layer which is able to execute the agent language. The agents will be able to access the hardware attached to a node via services which are provided by the virtualization platform. Additionally the agents can reproduce themselves to another node or even board. Within the platform a scheduler will be responsible for providing execution time to each of the agents according to their priority.

3.2 Virtualization Platform

The main task of the virtualization platform is to interprete the agent language commands of the agents and to provide them access to the hardware attached to a node via well defined interfaces. Additionally the platform should allow the concurrent execution of the agents. Therefore some basic means for code and data protection for the agent memory is required. This is achieved by assigning each of the agents an own memory segment and not allowing any other agent to access any other memory but its own. If some collaboration between the agents is required this must be requested via the communication service. The metadata of an agent as its code and memory segment will be stored in a structure that is shown in the figure below.

Every agent has a unique id within the virtualization platform. Additionally a priority and a status for the scheduler are stored. Assigning these values for an agent lies within the scope of an agent developer. Reproducing an agent on a virtualization platform where the agent's id

```
typedef struct {

        uint8_t id;
        uint8_t priority;
        agent_status_t status;

        uint32_t status_flag;
        uint16_t pc;

        int16_t regs[REG_MAX];

        uint16_t code_len;
        uint16_t regstr_len [STR_REG_MAX];

        uint16_t* code;

        char** reg_str;

        volatile char* rec_msg_content;
        volatile uint16_t rec_msg_len;

} agent_t;
```

Figure 3.2: Agent structure

is already used will result in a denial of the reproduction by the platform. Every agent has 13 numerical general purpose registers used for the execution of the agent language. Additonally there are 3 char general purpose registers. The result of every agent language command will be written to the accumulator. There is also a program counter which is used for the execution of the agent and the numerical agent language representation is stored as well. The agent structure also contains a buffer for receiving messages from other agents.
In order to reproduce the agent on another board or node the agent's structure needs to be serialized and transmitted via the communication layer.
Additionally the virtualization platform has to provide the agent developer with some means to deploy the agent executable to the virtualization platform, during compilation of the platform. During the initialization of the platform all deployed agents should be instantiated on the given platform.

3.3 Execution Layer

The execution layer is responsible for the execution of an agent which is written in the agent language and later translated to agent opcodes. The agent language provides means for:

- storing values to the general purpose registers
- comparing the contents of the general purpose registers
- performing basic mathematical functions like addition, subtraction, multiplication and division
- a jump operation

- reproduction and cloning functions
- sleep, delay and terminate functions
- functions to access the hardware attached to a node

If a function of the agent language returns a value, this value will be stored in the accumulator, where it can be used later on for further operations e.g. comparison etc.
The basic workflow of the execution layer as soon it is called by the scheduler is to read the next agent language opcode (all agent opcodes have a fixed length) as identified by the program counter, to decode it and to perform the function which is described by the opcode. Eventually the program counter value is changed and the control is returned back to the scheduler.

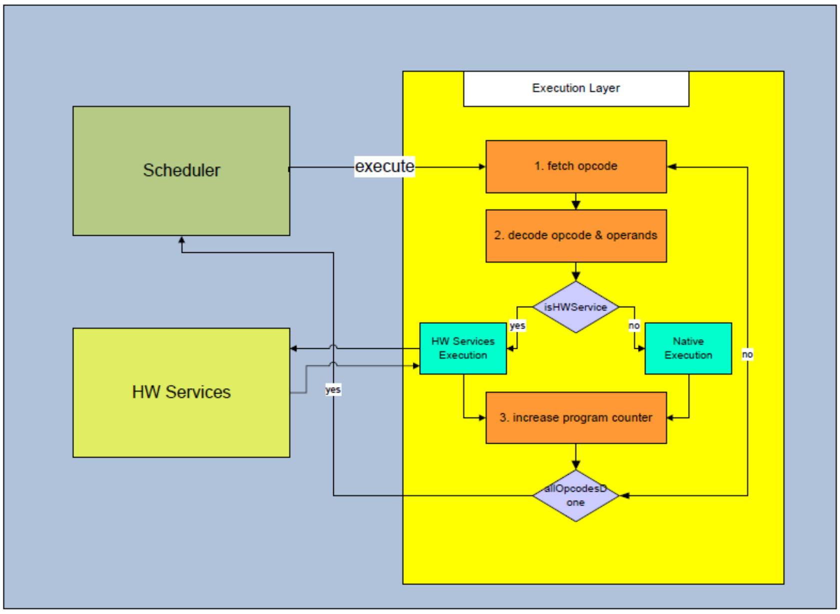

Figure 3.3: Execution Layer

The execution layer is called by the method execute which takes the following input parameters:

- Pointer to a specific agent structure
- Number of opcodes to execute

3.4 Hardware Services

The virtualization platform provides access to the hardware attached to a node via according hardware device drivers. The methods of the device drivers are made public to the execution layer which in turns allows the agents to access these methods. As the hardware supported by a node differs from node to node the virtualization platform should be able to discover during its initialization which hardware is supported on the node where it's running.
This will be achieved by defining a global set of function pointers within the virtualization platform. This set should contain all possible methods of all available device drivers. During initialization the platform will assign the according function pointer to a method provided by the device driver if the device is supported, otherwise the according function pointer will stay null.

```
/**
        Function : execute_agent
                excutes the next opcodes of an agent
        Returns :
                the amount of successfully executed opcoded
        Parameters :
                agent : agent status structure
                opcode: amount of opcodes to be executed
**/
uint8_t execute_agent(agent_t *agent, uint8_t opcode_size);
```

Figure 3.4: Executing agent language opcode

```
typedef struct {

        void (*set_bargraph)(uint8_t value);

        uint32_t (*clk_get_time)(void);

        void (*set_cooler)(uint8_t duty_cycle);

        void (*DISPLAY_drawBg)(uint16_t rgb);

        void (*DISPLAY_drawDot)(uint8_t row, uint8_t col,
                                uint16_t rgb, uint8_t grid);

        void (*DISPLAY_draw_char)(uint8_t x, uint8_t y,
                                uint16_t font_color, uint16_t bg_color,
                                uint8_t pixel_size, char c);

        void (*heater_set)(uint8_t duty_cycle);

        void(*button0_callback)(void);
        void(*button1_callback)(void);

        uint16_t (*therm_get_temp)(uint8_t name);

        void (*dotmatrix_send)(char *data);

} drivers_t;
```

Figure 3.5: Device drivers methods

The platform detects all the supported drivers on a specific node by inspection of the linked drivers. All drivers linked will be initialized during the setup of a platform and the methods provided by the drivers will be stored in the global function pointer table. Choosing this approach we would reach some form of modularity which would allow us to exchange the device drivers without necessity to change the platform code.
If an interaction with a device driver is blocking, then the calling agent will be put to status blocking unless there is an answer from the device driver.

3.4.1 Device drivers

3.4.1.1 Cooler

The device driver for the cooler should be initialized with a function:

- `void init_cooler(void)` This function should configure the timer and set PWM mode. After executing init function cooler should stay off.
- `void set_cooler(uint8_t duty_cycle)` This function sets the duty cycle of the PWM-signal, which controls the speed of the fan and the cooling effect.
- `required components` 1 timer for PWM signal

3.4.1.2 Heater

The device driver for the heating registers should be initialized with a function:

- `void heater_init(void)`

  This function should configure the timer and set PWM mode. After executing init function heater should stay off.
- `void heater_set(uint8_t duty_cycle)`

  This function sets the duty cycle of the PWM-signal, which controls the dissipated power (0 - no heating, 100 - max power dissipation)
- `required components` 1 timer for PWM signal

3.4.1.3 Temperature sensor

The temperature sensor driver should be initialized with the following function:

- `void therm_init(void)`

  This function should initialize temperature sensors connected to I2C bus.
- `uint16_t therm_get_temp(uint8_t name)`

  This function returns the value of the temperature in degrees Celcius.

3.4.1.4 Led matrix

The led matrix driver should be initialized with the following function:

- `void init_dotmatrix(void)`
- `void dotmatrix_send(char *data)`

  Using this function we send the first six characters to the led matrix.

3.4.1.5 Bargraph

With the following function we initialize LED bargraph, connected to the port A of nodes 0 or 1

- `void bargraph_init(void)`
- `void set_bargraph(uint8_t value)`

  This function is used to display the corresponding `value` on the bargraph.

3.4.1.6 TFT display

The following function is used to initialize TFT display that is connected to the node 2 of the ESE board:

- `void DISPLAY_init(void)`
- `void DISPLAY_drawBg(uint16_t rgb)`

  This function is used to draw the background of the display. RGB color could be defined using the following macro: `RGB(R[0..255], G[0..255], B[0..255])`.

- `void DISPLAY_string(uint8_t x, uint8_t y, uint16_t font_color, uint16_t bg_-color, uint8_t pixel_size, char *string)`

  The following function is used to display char array on the display, starting from the position `x, y` with corresponding RGB values of font and background. The size of the font could be changed by setting the size of the basic drawing pixel.

3.5 Communication Layer

The agents should be able to communicate with other agents on the same node or on the same board. Therefore the agent language provides means to request the sending or receiving of a message.
The sending function is blocking the further execution of the agent until the message is sent. When an agent wants to send a message this message is proceeded to the communications service which takes care of the actual transmission. While the sending procedure is ongoing the further execution of the sending agent is blocked. As soon as the communication service signalizes a successful message transmission or a failure the result of the sending function is written to the accumulator and the agent will be made available for further execution.
When an agent sends a message to another agent, the receiving platform stores the content to the receiver agent structure. The receiving agent is able to retrieve the last message from its buffer. However only one message can be stored within the receiving agent structure and the next message will overwrite the content and possible the id of the last message.
The communication service provides no guarantees that sending of a message will succeed; it works on a best effort approach. Therefore the agent developer has to make sure by reading the return value of a sending operation whether the message was successfully sent or not and should initialize a retransmission in case of failure.
Every message sent should be identified by an id, in order to allow the transmission of messages with different semantics.
The receiver of a message should be identified via the node number (0..3) where the receiving agent is currently expected to be running and the receiver agent id. As the ids of agents are within the scope of the agent developer she has to make sure, that the correct receiving agent is addressed. Additionally a multicast message could be supported by allowing omitting the node address which should result in sending the message to all agents identified by the provided id.

3.6 Scheduler

The main task of the scheduler which is part of the virtualization platform is to identify the next agent to be executed and to utilize the execution layer to perform the execution of the according agent. The decision which agent to be chosen should be made on a static priority based scheduling policy.
Every agent is assigned a priority (0..254) by the agent developer which is stored within the agent structure. The highest priority is 254 and the lowest priority is 0. Based on the priorities

of the currently running agents the scheduler creates a static list by which the order of the agent execution is defined. The scheduler instructs the execution layer to execute exactly priority + 1 opcodes for a given agent. Eventually the control returns to the scheduler and the next agent from the list is picked. The list is iterated cyclically.

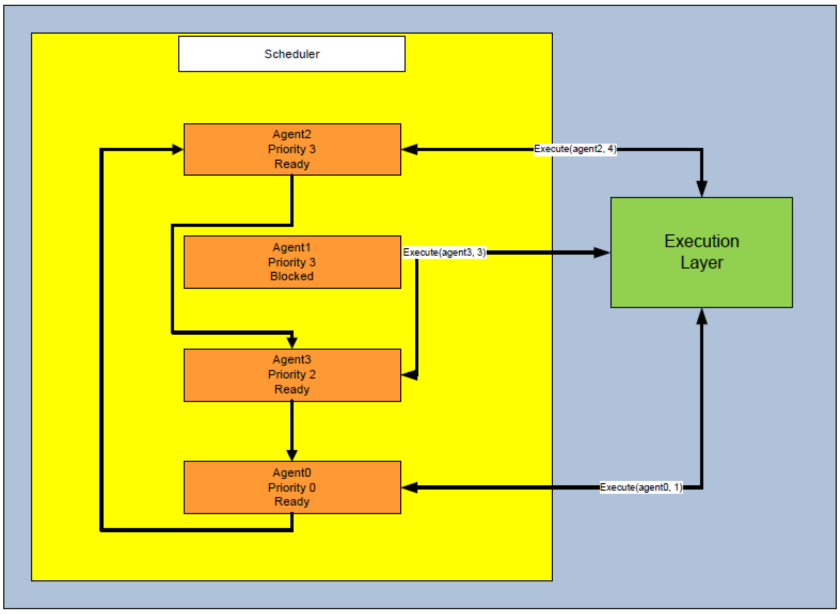

Figure 3.6: Scheduler

If an agent which is to be scheduled next is currently blocked, then its execution should be omitted.
As an agent can reproduce it self to another node or board or clone itself within the same platform the scheduling list requires adaptation as soon as new agent is deployed on a platform. Whenever a given platform is the destination end point of a reproduction respectively cloning operation the scheduler needs to update its scheduling list before proceeding with further executions.

3.7 Agent language

3.7.1 Agent language (Assembler level)

To develop a mobile agent *Agent language* will be used. Language is tied to agent internal structure and support necessary operation for code mobility and message exchange. While writing the program for agent user should not be aware of hardware services presented on a given platform, but have common knowledge about all available services and what operations are allowed to do with the services (have list of services and available operations).

It is the responsibility of the platform to provide required service to the agent (perform measurement, IO operation) or to manifest an error if the service is not available on the given platform. All current variables are allowed to store only in registers of agent structure (Fig. 3.2).

We propose to use the following principle: every opcode should be 16 bits long, that will lead to more simple procedure during decoding of the executable on the platform side. Another principle is that `reg_0` is used as `accumulator`: the results of all computations, comparisons an messages received by the agent will be put to this register. This will lead to more compact code on the platform side.

The language supports the following groups of operations: arithmetic, control flow, code mobility, message exchange, access to hardware services. Every agent has 16 registers in its internal structure: 13 for holding 16-bit numerical values and 3 for holding character strings.

To achieve the following twofold goal: keep the length of the every opcode 16 bits as well provide a capability to directly write values to the 16-bit registers we propose to split every 16-bit register into `high` and `low` parts, that will be used in `ldl` and `ldh` commands. Pictorially we represent it as follows.

The following table reporesents registers of agent structure as well as their corresponding addresses.

Addressing registers of agent structure			
General purpose registers		**Character registers**	
Register	rrrr	Char Register	rrrr
reg_0	0000	reg_str_0	1101
reg_1	0001	reg_str_1	1110
reg_2	0010	reg_str_2	1111
reg_3	0011		
reg_4	0100		
reg_5	0101		
reg_6	0110		
reg_7	0111		
reg_8	1000		
reg_9	1001		
reg_10	1010		
reg_11	1011		
reg_12	1100		

3.7.1.1 Arithmetic operations of agent assembly language

Addition

Add the content of `reg_d` and `reg_r` (or `value`) and put the result into `reg_0`.

`add reg_d, reg_r`

Syntax	Operands	Program counter	Flags
add reg_d, reg_r	reg_0 $\leq$ reg_d $\leq$ reg_12, reg_0 $\leq$ reg_r $\leq$ reg_12	PC = PC + 1	C

Operation

reg_0 $\leftarrow$ reg_d + reg_r

16-bit opcode:

		reg_d	reg_r
0000	0011	dddd	rrrr

`add reg_r, value`

Syntax	Operands	Program counter	Flags
add reg_r, value	reg_0 $\leq$ reg_r $\leq$ reg_12, 0x00 $\leq$ value $\leq$ 0xFF	PC = PC + 1	C

Operation

reg_0 $\leftarrow$ reg_r + value

16-bit opcode:

	reg_r	value	
0011	rrrr	vvvv	vvvv

Subtraction

Subtract `reg_s` (or `value`) from `reg_m` and put the result into `reg_0`.

`sub reg_m, reg_s`

Syntax	Operands	Program counter	Flags
`sub reg_m, reg_s`	`reg_0` ≤ `reg_m` ≤ `reg_12`, `reg_0` ≤ `reg_s` ≤ `reg_12`,	`PC = PC + 1`	`C`

Operation

`reg_0` ← `reg_m - reg_s`

16-bit opcode:

		reg_m	reg_s
0000	0110	mmmm	ssss

`sub reg_m, value`

Syntax	Operands	Program counter	Flags
`sub reg_m, value`	`reg_0` ≤ `reg_m` ≤ `reg_12`, `0x00` ≤ `value` ≤ `0xFF`	`PC = PC + 1`	`C`

Operation

`reg_0` ← `reg_m - value`

16-bit opcode:

	reg_m	value	
0110	mmmm	vvvv	vvvv

Division

Divide `reg1` by `reg2` (or value) and put the result into `reg_0`.

`div reg_d, reg_r`

Syntax	Operands	Program counter	Flags
`div reg_d, reg_r`	`reg_0` ≤ `reg_d` ≤ `reg_12`, `reg_0` ≤ `reg_r` ≤ `reg_12`,	`PC = PC + 1`	`C`

Operation

`reg_0` ← `reg_d / reg_r`

16-bit opcode:

		reg_d	reg_r
0000	1001	dddd	rrrr

`div reg_d, value`

Syntax	Operands	Program counter	Flags
`div reg_d, value`	`reg_0` ≤ `reg_d` ≤ `reg_12`, `0x00` ≤ `value` ≤ `0xFF`	`PC = PC + 1`	`C`

Operation

`reg_0` ← `reg_d / value`

16-bit opcode:

	reg_d	value	
1001	dddd	vvvv	vvvv

Multiplication

Multiply `reg1` and `reg2` (or `value`) and put the result into `reg_0`.

`mul reg_d, reg_r`

Syntax	Operands	Program counter	Flags
`mul reg_d, reg_r`	`reg_0` $\leq$ `reg_d` $\leq$ `reg_12`, `reg_0` $\leq$ `reg_r` $\leq$ `reg_12`	`PC = PC + 1`	`C`

Operation

`reg_0` $\leftarrow$ `reg_d * reg_r`

16-bit opcode:

		reg_d	reg_r
0000	1100	dddd	rrrr

`mul reg1, value`

Syntax	Operands	Program counter	Flags
`mul reg_d, value`	`reg_0` $\leq$ `reg_d` $\leq$ `reg_12`, `0x00` $\leq$ `value` $\leq$ `0xFF`	`PC = PC + 1`	`C`

Operation

`reg_0` $\leftarrow$ `reg_d * value`

16-bit opcode:

	reg_d	value	
1100	dddd	vvvv	vvvv

3.7.1.2 Control flow operations and comparison in agent assembly language

Jump if greater

Jump to `offset` in code segment of agent structure if the value of `reg_0`. is 1.

`jmpgr offset`

Syntax	Operands	Program counter
`jmpgr offset`	`-128` $\leq$ `offset` $\leq$ `+127`	`PC = PC + offset +1` if `reg_0 = 1`, `PC = PC + 1` otherwise

16-bit opcode:

		offset	
1111	0011	vvvv	vvvv

Jump if equal

Jump to `offset` in code segment of agent structure if the value of `reg_0` is 0.

`jmpeq offset`

Syntax	Operands	Program counter
`jmpeq offset`	`-128` $\leq$ `offset` $\leq$ `+127`	`PC = PC + offset +1` if `reg_0 = 0`, `PC = PC + 1` otherwise

16-bit opcode:

		offset	
1111	0110	vvvv	vvvv

Jump if less
Jump to `offset` in code segment of agent structure if the value of `reg_0` is -1.

`jmpls offset`

Syntax	Operands	Program counter
`jmpls offset`	`-128` $\leq$ `offset` $\leq$ `+127`	`PC = PC + offset +1` if `reg_0 = -1`, `PC = PC + 1` otherwise

16-bit opcode:

		offset	
1111	1100	vvvv	vvvv

Comparison
Compare `reg1` and `reg2` (or `value`).

`compare reg_d, reg_r`

Syntax	Operands	Program counter
`compare reg_d, reg_r`	`reg_0` $\leq$ `reg_d` $\leq$ `reg_12`, `reg_0` $\leq$ `reg_r` $\leq$ `reg_12`	`PC = PC + 1`

Operation
`reg_0` $\leftarrow$ `1` if (reg_d - reg_r $>$ 0)
`reg_0` $\leftarrow$ `0` if (reg_d - reg_r $=$ 0)
`reg_0` $\leftarrow$ `-1` if (reg_d - reg_r $<$ 0)

16-bit opcode:

		reg_d	reg_r
0000	1010	dddd	rrrr

`compare reg_d, value`

Syntax	Operands	Program counter
`compare reg_d, value`	`reg_0` $\leq$ `reg_d` $\leq$ `reg_12`, `0x00` $\leq$ `value` $\leq$ `0xFF`	`PC = PC + 1`

Operation
`reg_0` $\leftarrow$ `1` if (reg_d - value $>$ 0)
`reg_0` $\leftarrow$ `0` if (reg_d - value $=$ 0)
`reg_0` $\leftarrow$ `-1` if (reg_d - value $<$ 0)

16-bit opcode:

	reg_r	value	
1010	rrrr	vvvv	vvvv

3.7.1.3 Code mobility operations of agent assembly language

Move code
Move agent structure to platform that possess required service

`move service`

Syntax	Operands	Program counter
`move service`	`service_0` $\leq$ `service` $\leq$ `service_255`	`PC = PC + 1`

Operation
Serialize and transmit agent structure to the platform that possess required service

16-bit opcode:

	reg_r	service	
1111	0001	ssss	ssss

Clone code

Replicate agent structure on the given platform

`clone`

Syntax	Program counter
`clone`	`PC = PC + 1`

16-bit opcode:

1111	0010	0000	0000

Die

Destroy agent structure and free corresponding memory

`die`

16-bit opcode:

1111	0100	0000	0000

3.7.1.4 Message exchange

Send Message exchange between agents

`sendmsg reg, agent, platform`

Syntax	Operands	Program counter
`sendmsg reg, agent, platform`	`platform_0` ≤ `platform` ≤ `platform_3`	`PC = PC + 1`
	`agent_0` ≤ `agent` ≤ `agent_3`	
	`reg_0` ≤ `reg` ≤ `reg_12`	
	`reg_str_0` ≤ `reg` ≤ `reg_str_2`	

Operation

Send value of the register `reg` to the agent `aa` on the platform `pp`

16-bit opcode:

		register	agent	platform
1111	1000	rrrr	aa	pp

Receive

Pull message from `platform` to `register`.

`pullmsg reg`

Syntax	Operation	Program counter
`pullmsg`	`reg_0` ≤ `reg` ≤ `reg_12`	`PC = PC + 1`
	`reg_str_0` ≤ `reg` ≤ `reg_str_2`	

Operation

`reg` ← message

16-bit opcode:

			rrrr
1111	1010	0000	0000

3.7.1.5 Store, move and wait operations

Store

Store `value` in `h`-part of `reg_d`

`ldh reg_d, value`

Syntax	Operands	Program counter
`ldh reg_d, value`	`reg_0` $\leq$ `reg_d` $\leq$ `reg_12`, `0x00` $\leq$ `value` $\leq$ `0xFF`	`PC = PC + 1`
Operation		
`reg_d_h` $\leftarrow$ `value`		

16-bit opcode:

	reg_d	value	
1101	dddd	vvvv	vvvv

Store `value` in `l`-part of `reg_d`

`ldl reg_d, value`

Syntax	Operands	Program counter
`ldl reg_d, value`	`reg_0` $\leq$ `reg_d` $\leq$ `reg_12`, `0x00` $\leq$ `value` $\leq$ `0xFF`	`PC = PC + 1`
Operation		
`reg_d_l` $\leftarrow$ `value`		

16-bit opcode:

	reg_d	value	
0100	dddd	vvvv	vvvv

Push `char` value in the `str_reg`

`storecr reg_str, char`

Syntax	Operands	Program counter
`storecr reg_str, char`	`reg_str_0` $\leq$ `reg_str` $\leq$ `reg_str_2`	`PC = PC + 1`
Operation		
`reg_str` $\Leftarrow$ `value`		

16-bit opcode:

	reg_str	value	
1011	rrrr	vvvv	vvvv

Clear the `str_reg`

`clr reg_str`

Syntax	Operands	Program counter
`clr str_reg`	`reg_str_0` $\leq$ `reg_str` $\leq$ `reg_str_2`	`PC = PC + 1`
Operation		
`clear str_reg`		

16-bit opcode:

			rrrr
0000	0010	0000	rrrr

Move

Move value from reg_r to reg_d

`mv reg_d, reg_r`

Syntax	Operands	Program counter
`mv reg_d, reg_r`	reg_0 $\leq$ reg_d $\leq$ reg_12, reg_0 $\leq$ reg_r $\leq$ reg_12	PC = PC + 1

Operation

reg_d $\leftarrow$ reg_r

16-bit opcode:

		reg_d	reg_r
0000	1101	dddd	rrrr

Wait

Wait for ms

`wait delay_ms`

Syntax	Operands	Program counter
`wait delay_ms`	0 $\leq$ delay_ms $\leq$ 0xFF	PC = PC + 1

16-bit opcode:

		delay	
0000	0101	dddd	dddd

Assign priority value

Assign priority of the agent to value in range 0..3

`priority value`

Syntax	Operands	Program counter
`priority value`	0 $\leq$ value $\leq$ 3	PC = PC + 1

Operation

priority $\leftarrow$ value

16-bit opcode:

		priority	
0000	1000	pppp	pppp

3.7.1.6 Access to hardware services

Set

Set service to reg or value

`setservice service_id, reg`

Syntax	Operands	Program counter
`setservice service_id, reg`	service_0 $\leq$ service_id $\leq$ service_255, reg_0 $\leq$ reg $\leq$ reg_12	PC = PC + 1

16-bit opcode:

	reg	service_id	
0111	rrrr	ssss	ssss

Get
Put corresponding value from the service to the `reg_0`.

`getservice service_id`

Syntax	Operand	Program counter
`getservice service_id`	service_0 $\leq$ service_id $\leq$ service_255	`PC = PC + 1`

16-bit opcode:

		service_id	
0000	0111	ssss	ssss

3.8 Communication Architecture

The communication architecture is designed to support communication between nodes on the same development board as well as between boards.

3.8.1 Hardware

The communication on the board is carried out over two serial bus channels. One of them is to be used for a distributed control application running on nodes 0-3. Another bus is dedicated for code mobility between nodes 0-4.

Access to the bus is controlled by separate UART modules on each micro-controller. The bit rate is constrained by the maximum value of 2 Mbps according to the manual.

Node 4 functions as a gateway to another board. It is a bridge between the local and the wireless zigbee network.

3.8.2 Code Mobility

Code mobility between nodes includes local mobility on the same board and remote mobility between different boards. Executable agents generally have larger volume than control data. Sending at regular time intervals is not assumed, thus communication is aperiodic. A simple protocol based on message acknowledgment can be used.

There are two use cases: a) local mobility: destination is one of the nodes 0-3. b) remote mobility: destination is the gateway node 4. The gateway is to contain a zigbee stack implementation to enable access to the personal area network.

3.8.3 Addressing Scheme

Simple local addressing requires unique identifiers for each node. For remote communication, board addresses have to be compatible with the configuration of the zigbee network. Since, each node will have a static number of agent execution environments, the address has to contain its identifier as as well.

3.8.4 Communication Interface

The interface for accessing the communication system is given below in Figures 3.7 through 3.9.

```
struct frame{
        unsigned dst_node:4;            //destination node
        unsigned dst_board:4;           //destination board
        unsigned dst_agent:4;           //destination agent
        frame_id_t frame_id;            //source of a message
        unsigned frame_length:16;       //length of frame payload in
            bytes
        unsigned index:16;              //index for buffering
        struct frame *next_frame;       //next frame to be sent
        char *data;                     //payload
};
```

Figure 3.7: Message Structure

```
/**
   Function: recv_handler
             Reassembles a complete frame from received packets
   Parameters: msg_length        length of the current packet
               msg_body          payload of the current packet
 */
void recv_handler(uint8_t msg_length, uint8_t *msg_body);
```

Figure 3.8: Message Receiving

```
/**
   Function: send_message
             Sends message over communication interface
   Parameters: frame
               frame to be sent
   Returns: amount of sent packets

*/
uint8_t send_message(frame_t frame);
```

Figure 3.9: Message Sending

4 Implementation

4.1 Platform

4.1.1 Initialization

The platform initialization depends on the settings of a nodes makefile and on the data provided by the Application developer. The settings of a nodes makefile influence the set of hardware services available to the specific platform. When a node is linked to some hardware drivers supported by the platform e.g. bargraph, the according makefile will compile the platform code with a C preprocessor setting -DBARGRAPH. The platform will only support those drivers for which C preprocessor defines where made, by inspecting the defines and only assigning the driver function pointers supported. This allows for simple adaptation and extension of the platform by changing the drivers linked to the platform. Additionally by choosing this approach a smaller size of the executable is achieved.

```
# put platform specific hardware drivers to be supported by this node
OBJ-ESEL-MDEP-$(MNAME)-y += protocol0 bargraph
```

Figure 4.1: Platform makefile

The makefile snippet from the figure shown above will result in a compilation of the platform with the setting -DPROTOCOL0 -DBARGRAPH.

The initialization code of the platform checks for these defines and only registers and initializes those drivers supported as shown in the figure below.

```
#ifdef BARGRAPH
        bargraph_init();
        platform.drivers.set_bargraph = set_bargraph;
#endif

#ifdef PROTOCOL0
         protocol_init(platform.id, recv_handler);
#endif
```

Figure 4.2: Platform drivers initialization

Additionally the agents to be executed need to be initialized on the platform by the application developer. This is achieved by providing C macros to the application developer which need to be filled with proper data. The C macros offered by the platform are shown in the figure below.

The AGENT_INIT macro needs to be defined by the application developer in order to instantiate an agent. As its input parameters it requires the agent id, agent priority and a binary string

```
#include "platform.h"

PLATFORM_CONFIGURATION()
{
  AGENTS_CONFIGURATION(){
      // agent id, agent prio, agent_code
      AGENT_INIT(0x02, 0x02, 000001110000000111110
      0011010001000001010001111001000000000000001111
      1001111111011),
  },

  BOARD_ID(0x00)
};
```

Figure 4.3: Platform agent initialization

representing the agent code, which is delivered by the platform assembler tool (asm_agent). During platform initialization the binary string is converted to a binary representation in order to reduce the actual code size. Up to 4 agents can be initialized. All configured agents are assigned the status ready.

Additionally the application developer is able to initialize the board id, required for inter board communication via the BOARD_ID macro.

4.1.2 Execution

After successfull platform intialization the scheduler iterates through the configured agents i.e. those with status ready and forwards them to the execution layer to be executed via the method execute_agent shown in figure 4.4 on page 30.

The execution layer fetches the next opcode for the considered agent, decodes the according and finally executes the specific opcode. Eventually the program counter is increased and the next opcode gets executed. The execution of an agent is stopped as soon as the desired amount of opcode has been executed or if an agent was put to a different status than ready.

The decoding of the agent opcodes is performed by analyzing the 8 bit opcode header of the total 16 bit opcode as exemplarily shown in figure 4.5 on page 31.

Finally the opcode gets executed and agent configuration structure is updated as shown in figure 4.6 on page 31.

After all opcodes of an agent have been executed or the according agent was stopped the scheduler looks for the next agent with status ready to be executed.

4.1.3 Communication

The communication layer provides means to send and receive messages via the USART serial bus. The lower level implementation of the CSMA/CA protocol allows up to 15 bytes of payload to be transferred with a single message. Due to this limitation an upper layer protocol is introduced which allows greater messages to be exchanged between nodes.

```
uint8_t execute_agent(agent_t *agent, uint8_t opcode_size) {

  uint8_t opcodes_done = 0;

  while (opcodes_done < opcode_size) {
    //1. fetch next opcode
    uint16_t opcode = agent->code[agent->pc];

    //2. decode opcode
    opcode_t dec_opcode = decode_opcode(opcode);

    //3. execute opcode
    execute_opcode(agent, dec_opcode);

    //4. increase program counter
    if (agent->status == ready) {
      if (agent->pc < agent->code_len - 1 || agent->pc == 0xffff) {
        agent->pc += 1;
      } else {
        agent->status = stopped;
        break;
      }
      opcodes_done += 1;
    } else {
      return opcodes_done;
    }

  }

  return opcodes_done;
}
```

Figure 4.4: Platform agent execution

This protocol works with frames, where a frame is split into a sufficient amount of packets which are transmitted via the serial bus sequentially. In order to increase data throughput 2 types of packages were introduced: start packages and data packages.

The start packages always initialize the sending of a new frame and contain all the necessary data to succesfully address the destination of the packet and inform the receiver about specific frame settings i.e. frame id and frame length. Figure 4.1.3 on page 32 shows the layout of start packages.

The data packages are only used when a frame payload is greater than the 15 byte which can be sent within a single packet. These data packages identify the frame to which the belong and are able to transmit more payload data within a package. Figure 4.2 on page 32 shows the layout of data packages.

```
uint8_t nibble1 = NIBBLE1(opcode);
uint8_t nibble2 = NIBBLE2(opcode);

switch (nibble1) {
//0000
case 0:
  switch (nibble2) {

  //clr reg_str
  //0000 0010 0000 rrrr
  case 2:
    result.id = CLEAR;
    result.reg1 = NIBBLE4(opcode);
    break;

  //add reg_d, reg_r
  //0000 0011 dddd rrrr
  case 3:
     result.id = ADD_R;
     result.reg1 = NIBBLE3(opcode);
     result.reg2 = NIBBLE4(opcode);
    break;
```

Figure 4.5: Platform agent opcode decoding

```
case JMP_G:
  PRINTF("jmpgr offset:%d\n", opcode.value);
  if (agent->regs[REG_ACC]==1) {
    agent->pc = agent->pc + opcode.value;
  }
  break;

case JMP_E:
  PRINTF("jmpeq offset:%d\n", opcode.value);
  if (agent->regs[REG_ACC]==0){
    agent->pc = agent->pc + opcode.value;
  }
  break;

case JMP_L:
  PRINTF("jmpls offset:%d\n", opcode.value);
  if (agent->regs[REG_ACC]==-1){
    agent->pc = agent->pc + opcode.value;
  }
  break;
```

Figure 4.6: Platform agent opcode execution

dst_node	packet len
start_type	src board
src_node	frame id
packet id hi	
packet id low	
dst board	dst agent
frame length hi	
frame length low	
data	
...	
crc	

Table 4.1: Start Package

dst_node	packet len
start_type	src board
src_node	frame id
packet id hi	
packet id low	
data	
...	
crc	

Table 4.2: Data Package

The receiving platform of the communication reassembles the received packets into a single frame prior to informing the according agent about this event.

4.1.4 Code Mobility

In order to provide means for code mobility a localization service is introduced which allows identifying the hardware supported by a specific node. This is achieved by a static array storing the addresses of the nodes supporting a specific hardware as shown in figure 4.7 on page 32. This localization is only valid for the current ESE board and requires adaptation when porting the platform to another board.

```
uint8_t service_locations[MAX_SERVICE][MAX_NODES] = {
        {NODE0_ID, NODE1_ID, INVALID, INVALID},   //BARGRAPH
        {NODE1_ID, INVALID, INVALID, INVALID},    //THERMOMETER
        {NODE1_ID, INVALID, INVALID, INVALID},    //COOLER
        {NODE1_ID, INVALID, INVALID, INVALID},    //HEATER
        {NODE3_ID, INVALID, INVALID, INVALID},    //LED
        {NODE2_ID, INVALID, INVALID, INVALID},    //LCD
        {NODE0_ID, NODE1_ID, NODE2_ID, NODE3_ID}  //BUTTONS
};
```

Figure 4.7: Service localization

After the address of the receiving board has been identified, the agent is serialized via the serialiaze_agent method. A frame containing a code mobility message is marked by a code mobility header and trailer (0x55).

When a complete frame has been received by a platform it checks whether this is a data message or code mobility message by inspecting the first(header) and last(trailer) byte of the received message. If a code mobility message was received the platform deserializes the agent, increments its program counter by 1 and instantiate this very agent within the platform so its considered for execution during the next scheduling round.

```
if (GET_MOBILITY_HEADER(current->data) == MOBILITY_BYTE &&
        GET_MOBILITY_END(current->data, current->frame_length - 1)
        == MOBILITY_BYTE){
  //code mobility message

  for (i = 0; i < AGENT_MAX; i++){
    if (platform.agents[i].status == stopped){
      agent_t agent = deserialize_agent(current->data);
      agent.id = id;
      agent.regs[REG_ACC] = 0;
      agent.pc+= 1;
      platform.agents[i] = agent;
      break;
    }
  }
```

Figure 4.8: Agent deserialization

In order to allow to distinguish whether the agent was moved or is the initiatior of the moving, the receving platform writes a 0 to accumulator of the received agent, whereas the sending platform writes the amount of sent packets to the accumulator of the sending agent.

4.2 Agent language assembler tool

Agent language assemler tool provides the means to convert agent program with `.ma` extension (which stands for "mobile agent") into binary code, that could be executed on the platform. After the compilation it generates two files: listing of the program for the debugging purposes and binary file with `.bin` entension.

For the implementation of Agent language assembler tool we use Python programming language.

The implementation of Agent language assembler tool is basically two pass assember, block diagram of which is shown in in the figure below.

This tool performs two passes over source file. In the first pass it reads the entire source file, looking for labels in the source code and identifying opcodes. All labels, mnemonics, operands are collected and are put to the symbol table. No instructions are assembled during this pass and symbol table contain labels, mnemonics and operands. As every instruction has fixed size it is clear enough how to determine the offset in the branching instruction. But we should point out that during the execution of each instruction programm counter increments by 1, so the `final_offset = relative_offset - 1`.

One of the most important issues is to correctly assemble labels of the branching instructions. There can be two problems with labels: *multiple-defined labels* and *invalid labels*. Example of multiple-defined labels error is as follows:

```
GOTO:  add reg_0, reg_4
       ...
       ...
GOTO:  setservice temp, reg_0
       ...
       ...
       pullmsg reg_4
       compare reg_4, reg_0
       jmpls GOTO
```

Figure 4.9: Multiple labels error

For the

5 Validation

We have performed validation on different levels for different part of our project. The main components on which validation was performed are Agent language tool, virtualization platform, communication protocol, as well as the final validation of overall system.

The validation of Agent language based on the unit testing methodology, since inputs and outputs for this part can be clearly and unambiguously defined.

The platform was validated by series of tests, based on the use cases in our specification (see sec.3).

The communication protocol was the most elusive part to validate. Based on extensive testing in different use scenarios we could establish a safe margin for maximum of data transmission rate. For the validation of the protocol we used tools for serial communication between microcontroller and PC, such as `xxd`.

In order to test the services provided by the aforementioned software components together we have implemented application for controlling temperature. First of all we test the interaction between agent language tool and execution platform by compiling the platform for x86 target applying our debug platform.

The second step was to perform functional validation of the platform by executing the application code. The code itself in C-format has been previously validated on microcontroller simulator for the target environment.

5.1 Overall system validation

Overall system validation			
	description	expected result	result
1	platform initialization		8.11.2012
defining requirements, and specification (ALAT, communication architecture, platform)	11.11.2012 - 30.11.2012	29.11.2012	-1 days
presentation for workshop 1	30.11.2012 - 5.12.2012	5.12.2012	0 days
implementation phase (total)	6.12.2012 - 21.12.2012	21.01.2013	+30 days
implementation phase (Agent language tool)	6.12.2012 - 12.12.2012	17.01.2013	+5 days
implementation phase (communication protocol)	6.12.2012 - 11.12.2012	10.01.2013	-1 days
implementation phase (refinement of drivers)	03.12.2012 - 07.12.2012	07.12.2012	-1 day
implementation phase (platform, total)	08.12.2012 - 27.12.2012	23.01.2013	+26 days
implementation phase (platform, scheduling)	08.12.2012 - 15.12.2012	15.12.2012	0 days
implementation phase (platform, hardware middle layer)	15.12.2012 - 21.12.2012	10.01.2013	+20 days
implementation phase (platform, communication layer)	22.12.2012 - 27.12.2012	23.01.2013	+26 days
validation phase	13.01.2013 - 24.01.2013	26.01.2013	+2 days
workshop 2 presentation	23.01.2013 - 29.01.2013	29.01.2013	0 days
lab protocol documentation	12.11.2012 - 23.01.2013	29.01.2013	+5 days

5.2 Driver validation

5.3 Platform validation

5.4 Agent language tool-Platform validation

During the design of agent program and execution of an agent two reverse actions are performed:

5.5 Validation of communication protocol

5.6 Test cases

6 Results, future plans and expenditure of work

6.1 Platform

As the size of flash memory in μC is enough limited (128 kB), one of our goals was to reduce the size of firmware as small as possible, in order to keep the platform extendable for adding new features. Every platform that corresponds to a node on the ESE board supports only dedicated hardware drivers, this approach is achieved during compilation time.

6.2 Communication

We have implemented CSMA/CA protocol which allows up to 15 bytes of payload in a single message. To transfer payload more that 15 bytes we introduce frames which provide us with higher level of abstraction. We transfer agent code using the aforementioned frames. Before moving the code, the platform serialize agent and then send it as messages over the communication medium.

As a result we have communication protocol that supports transferring agent code and messages between agents from one board to another.

During the implementation phase we spent two times more then it was planned trying to make board to board communication using zigbee and tiny os.

What we have figured out is the following: it is possible to transfer the data from one board to another using tinyOS and zigbee stack, it works great. The main pitfall here is the communication between zigbee node and serial bus. TinyOS is a modular software and provides means to access hardware by adding components to the kernel and access the hardware by means of interfaces of this components.

For accessing the serial port the following components could be used: `UARTByte, UARTStream,` ... which provide different level of abstraction: from sending a single byte to arrays via UART, but none of this components send anything to the serial in practice. TinyOS website provides information that in most of the cases components are platform specific, so we come to conclusion that probably serial communication using tinyOS does not work due to incompatibility of the platform.

6.3 Agent language

We have developed the Agent language, defined executable opcodes and implemented assembler tool for converting agent program into binary code.

The main decision here was to keep the size of all opcodes constant, so that it lead to more simple assemble procedure and more simple decoding on the platform side.

As it is of prior importance to provide user with the feedback about the program written, we manifest all the possible errors in the agent program. Agent language tool generates listing file that shows the results of assemble and binary executable code of agent that could be inserted in a desired platform.

According to the specification, for mobile agent developer it is of no importance by what means to access the hardware. She could write, for instance,

```
getservice temp
```

and while executing this command platform will access temperature sensors on the ESE board via I2C interface, compute average of all three sensors and put the resulting value to the `reg_0` of agent. By using this approach we go one level up in the abstraction ladder hierarchy.

But there is always space for improvement. As one of the future goals one can consider to implement high level language, that will be compiled into the designed assembler language.

6.4 Drivers

While some drivers have already been implemented and provided with ESE library, to use all the required drivers in our project it is required to attach the drivers to the platform.

During this step some side condition has occurred: for example, because of simulataneous using of the same timer by two drivers the platform has been restarted.

6.5 Validation

6.6 Time expenditure

We started our work on October, 29th and end three month later, on 29th of January. Because of the problems with zigbee implementation and porting of the protocol to bus1, memory leaks, non-reproducable the whole project needed longer time then estimated.

According to the table of planned and real dates the implementation phase last one month longer then it was planned.

It is necessary to take into account that essentially from that month 2 weeks was Christmas holidays and due to the illness of our colleague [J] it is required more time that it was supposed to.

It is worth to mention that despite the force-majeure we succeeded to complete our project on time.

Planned and real dates			
	planned dates	real end date	
preliminary tasks (assigning roles, gathering data, setting up environment)	29.10.2012 - 10.11.2012	8.11.2012	-2 days
defining requirements, and specification (ALAT, communication architecture, platform)	11.11.2012 - 30.11.2012	29.11.2012	-1 days
presentation for workshop 1	30.11.2012 - 5.12.2012	5.12.2012	0 days
implementation phase (total)	6.12.2012 - 21.12.2012	21.01.2013	+30 days
implementation phase (Agent language tool)	6.12.2012 - 12.12.2012	17.01.2013	+5 days
implementation phase (communication protocol)	6.12.2012 - 11.12.2012	10.01.2013	-1 days
implementation phase (refinement of drivers)	03.12.2012 - 07.12.2012	07.12.2012	-1 day
implementation phase (platform, total)	08.12.2012 - 27.12.2012	23.01.2013	+26 days
implementation phase (platform, scheduling)	08.12.2012 - 15.12.2012	15.12.2012	0 days
implementation phase (platform, hardware middle layer)	15.12.2012 - 21.12.2012	10.01.2013	+20 days
implementation phase (platform, communication layer)	22.12.2012 - 27.12.2012	23.01.2013	+26 days
validation phase	13.01.2013 - 24.01.2013	26.01.2013	+2 days
workshop 2 presentation	23.01.2013 - 29.01.2013	29.01.2013	0 days
lab protocol documentation	12.11.2012 - 23.01.2013	29.01.2013	+5 days

Spent time per workpackage			
workpackage	planned time	spent time	difference
D1.1 Requirements and Specification	60	58	-2
D1.2 Presentation for Workshop 1	20	24	+4
D1.3 Presentation for Workshop 2	20	20	0
D1.4 Lab protocol	80	90	+10
D2.1 driver implementation	50	40	-10
D3.1 agent language tool	40	43	+3
D4.1 CSMA/CA communication protocol	60	57	-3
D5.1 Platform. Scheduler	20	14	-6
D5.2 Platform. Execution layer	30	43	+13
D5.3 Platform. Communication layer	70	113	+43

7 Conclusion

In our project we have focused on implementing and validating code mobility environment on the ESE board. As a testing application we have developed a distributed control application, where all actions are executed by agents, that reside on the each platform of the ESE board, Testing application demonstrates all the capabilities of developed environment: after the start of the application mobile agent is transfered from node 0 to node 1, where then this agent fetch temperature value of heating element from the platform, compares it with setpoint and adjust it by setting cooling service. Meanwhile the agents on the nodes 2 and 3 are responsible for accessing TFT display and led matrix.

The testing application illustrates the following implementation results: the ability to transfer code from one node to another including state and values of all registers, transferring messages between agents, execute several agents on the current platform.

For project management we used e-mail, personal meetings and skype to get the work done. We installed and deployed to the public cloud Heroku open source bugtracker (redmine). This web application could be used for composing Gantt diagram, defining and assigning tasks and documenting current results of the project, as well as planning future work. Redmine is great helper for project administrator since it is possible to assign roles and tasks for each members and monitor the current working process online.

The course was extremely helpful for various reasons. First of all, we master our skills in development of embedded system applications and communication between nodes. We get acquainted ourselves with new code mobility concept, implemented simple assembler-like language, sufficient for writing useful mobile agents. The most hard and time consuming part of the project was implementing virtualization platform that support scheduler, provides access to hardware services, and capable of transferring agents between platforms.

It is worth to mention that working in a group was extremely helpful for mastering project management skills. Defining plan and meeting the deadlines is worth to practice for real day-to-day experience. Despite all the difficulties with the third guy [J], we succeeded to get work done in time.

The workshop days was of great advantage for us to master our presentation skills as well as to discuss possible fallacies and pitfalls of our project and intended solution.

Bibliography

[1] E. Bartocci, D. Cacciagrano, N. Cannata, F. Corradini, E. Merelli, L. Milanesi, and P. Romano. An agent-based multilayer architecture for bioinformatics grids. *EEE transactions on Nanobioscience*, 6(2):142–148, 2007.

[2] Jürgen Galler. A modular software package for the embedded systems engineering board. Bachelor thesis, Faculty of Informatics, A-1040 Wien Karlsplatz 13, 2011.

[3] Flavio De Paoli, Antonella Di Stefano, Andrea Omicini, and Corrado Santoro, editors. *Proceedings of the 7th WOA 2006 Workshop, From Objects to Agents (Dagli Oggetti Agli Agenti), Catania, Italy, September 26-27, 2006*, volume 204 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[4] Paolo Romano, Ezio Bartocci, Guglielmo Bertolini, Flavio De Paoli, Domenico Marra, Giancarlo Mauri, Emanuela Merelli, and Luciano Milanesi. Biowep: a workflow enactment portal for bioinformatics applications. *BMC Bioinformatics*, 8(S-1), 2007.

A Source code

A.1 Platform

A.1.1 platform.h

```c
/*
 * platform.h
 *
 * Created on: Dec 8, 2012
 * Author: igor
 */

#ifndef PLATFORM_H_
#define PLATFORM_H_

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "global.h"
#include "bargraph.h"
#include "thermometer.h"
#include "cooler.h"
#include "heater.h"
#include "DISPLAY.h"
#include "protocol0.h"
#include "ledmatrix.h"
#include "pushbutton.h"

#define AGENT_MAX 4
#define OPCODE_LEN 16
#define STR_REG_MAX 3
#define REG_MAX 13

#define MAX_SERVICE 7
#define MAX_NODES 4
#define INVALID 0xff

#define NODE0_ID 0x04
#define NODE1_ID 0x03
#define NODE2_ID 0x02
#define NODE3_ID 0x01

#define PLATFORM_CONFIGURATION() \
platform_config_t platform_config =

#define AGENTS_CONFIGURATION() \
.agents_conf =

#define AGENT_INIT(agentid, agentprio, agentcode) \
```

```c
{ .id=agentid, \
.active = 1, \
.prio = agentprio, \
.code = #agentcode }

#define PLATFORM_ID(id) \
.platform_id = id

#define BOARD_ID(id) \
.board_id = id

typedef struct {

  uint8_t id;
  uint8_t active;
  uint8_t prio;
  char *code;

} agent_config_t;

typedef struct {
  agent_config_t agents_conf[AGENT_MAX];
  //uint8_t platform_id;
  uint8_t board_id;
  uint8_t frame_id;
} platform_config_t;

extern platform_config_t platform_config;

typedef struct {

  void (*set_bargraph)(uint8_t value);

  uint32_t (*clk_get_time)(void);

  void (*set_cooler)(uint8_t duty_cycle);

  void (*DISPLAY_string)(uint8_t x, uint8_t y, uint16_t font_color, uint16_t bg_color, uint8_t pixel_size, char *string);
  void (*DISPLAY_drawBg)(uint16_t rgb);

  void (*heater_set)(uint8_t duty_cycle);

  void(*button0_callback)(void);
  void(*button1_callback)(void);

  uint16_t (*therm_get_temp)(uint8_t name);

  void (*dotmatrix_send)(char *data);

} drivers_t;

enum agent_status {
  stopped, ready, blocked
};
typedef enum agent_status agent_status_t;

#define REG_ACC 0
```

```c
#define OVERFLOW 32
#define OVERFLOW_MASK 0x80000000
#define ERROR 0x000000FF
#define SET_ERROR(flag, errno) (flag |= (errno & ERROR))

typedef struct {

  uint8_t id;
  uint8_t priority;
  agent_status_t status;

  uint32_t status_flag;
  uint16_t pc;

  int16_t regs[REG_MAX];

  uint16_t code_len;
  uint16_t regstr_len [STR_REG_MAX];

  uint16_t* code;

  char** reg_str;

  volatile char* rec_msg_content;
  volatile uint16_t rec_msg_len;

  volatile uint8_t sem;
} agent_t;

typedef struct {
  volatile agent_t agents[4];
  drivers_t drivers;
  uint8_t id;
} platform_t;

extern volatile platform_t platform;
extern uint8_t service_locations[MAX_SERVICE][MAX_NODES];
extern volatile uint8_t button0_pressed;
extern volatile uint8_t button1_pressed;

void init_drivers(void);
void init_agents(void);
void reset_agent(uint8_t id);
uint8_t clone_agent(agent_t *agent);
void platform_init(void);
void run_platform(void);

void buttoncallback0(void);
void buttoncallback1(void);


#endif /* PLATFORM_H_ */
```

A.1.2 platform.c

```c
/*
* platform.c
```

```
 *
 * Created on: Dec 8, 2012
 * Author: igor
 */

#include "platform.h"
#include "hw_layer.h"
#include "scheduler.h"
#include "comm_layer.h"

#include "util/delay.h"

volatile platform_t platform;
volatile uint8_t button0_pressed;
volatile uint8_t button1_pressed;

uint8_t service_locations[MAX_SERVICE][MAX_NODES] = {
      {NODE0_ID, NODE1_ID, INVALID, INVALID},    //BARGRAPH
      {NODE1_ID, INVALID, INVALID, INVALID},   //THERMOMETER
      {NODE1_ID, INVALID, INVALID, INVALID},   //COOLER
      {NODE1_ID, INVALID, INVALID, INVALID},   //HEATER
      {NODE3_ID, INVALID, INVALID, INVALID},   //LED
      {NODE2_ID, INVALID, INVALID, INVALID},   //LCD
      {NODE0_ID, NODE1_ID, NODE2_ID, NODE3_ID}       //BUTTONS
};

void init_drivers(void){

#ifdef BARGRAPH
  bargraph_init();
  platform.drivers.set_bargraph = set_bargraph;
#endif

#ifdef PROTOCOL0
   protocol_init(platform.id, recv_handler);
#endif

#ifdef TIMER2

#endif

#ifdef CLOCK

#endif

#ifdef HEATER
  heater_init();
  platform.drivers.heater_set = heater_set;
  //heater_set(50);
#endif


#ifdef DISPLAY
  DISPLAY_init();
  platform.drivers.DISPLAY_string = DISPLAY_string;
  platform.drivers.DISPLAY_drawBg = DISPLAY_drawBg;
  /*DISPLAY_string(20, 150, RGB(30,238,30), RGB(0,0,0), 2, "Agents ↩
             :");
```

```
  DISPLAY_string(20, 130, RGB(30,238,30), RGB(0,0,0), 2, "Agent message :");
  DISPLAY_string(20, 110, RGB(30,238,30), RGB(0,0,0), 2, "Agent state :");
  DISPLAY_string(20, 110, RGB(30,238,30), RGB(0,0,0), 2, "Platform status:");*/
#endif

#ifdef THERMOMETER
  therm_init();
  platform.drivers.therm_get_temp = therm_get_temp;
#endif


#ifdef PUSHBUTTON
  platform.drivers.button0_callback = buttoncallback0;
  platform.drivers.button1_callback = buttoncallback1;
  init_pushbutton0(platform.drivers.button0_callback);
  init_pushbutton1(platform.drivers.button1_callback);
  button0_pressed = 0;
  button1_pressed = 0;
#endif

#ifdef COOLER
  init_cooler();
  platform.drivers.set_cooler = set_cooler;
  //set_cooler(50);
#endif

#ifdef LEDMATRIX
  init_dotmatrix();
  platform.drivers.dotmatrix_send = dotmatrix_send;
  //platform.drivers.dotmatrix_send=dotmatrix_send_scrolling_text;
  //dotmatrix_enable_scrolling();

  //char buf[] = "ABCD";
  //dotmatrix_send(buf);
  //platform.drivers.dotmatrix_send(buf);
  /*uint16_t val = 0xff;
  dotmatrix_send_int(val, 0);
  dotmatrix_send_comma(1);
  dotmatrix_send_int(val, 3);*/

#endif

}


void buttoncallback0(void){
  button0_pressed = 1;
}

void buttoncallback1(void){
  button1_pressed = 1;
}

void init_agents(){

```

```
  uint8_t i = 0;

  for (i=0; i < AGENT_MAX; i++) {

    if (platform_config.agents_conf[i].active == 1) {
      uint8_t id = platform_config.agents_conf[i].id;
      platform.agents[id].id = id;
      platform.agents[id].status = ready;
      platform.agents[id].priority = platform_config.agents_conf[i].prio;

      if (platform.agents[id].reg_str == 0) {
        platform.agents[id].reg_str = (char**) malloc(STR_REG_MAX * sizeof(char*));
        platform.agents[id].reg_str[0] = (char*) malloc(1);
        platform.agents[id].reg_str[1] = (char*) malloc(1);
        platform.agents[id].reg_str[2] = (char*) malloc(1);
      }

      size_t len = strlen(platform_config.agents_conf[i].code);
      platform.agents[id].code = (uint16_t*) malloc((len / OPCODE_LEN) * sizeof(uint16_t));
      uint16_t ind = 0;
      char opcode[OPCODE_LEN];
      while (ind < len / OPCODE_LEN) {
        strncpy(opcode,(platform_config.agents_conf[i].code + (ind * OPCODE_LEN)), OPCODE_LEN);
        platform.agents[id].code[ind] = strtol(opcode, NULL, 2);
        ind += 1;
      }
      platform.agents[id].code_len = ind;
    }
  }

}

void reset_agent(uint8_t id){

  agent_t *agent = (agent_t*)&(platform.agents[id]);
  agent->id = 0;
  agent->status = stopped;
  agent->priority = 0;
  memset(agent->regs, 0, REG_MAX * sizeof(int16_t));

  if (agent->reg_str != 0){

    if (agent->reg_str[0] != 0) {
      free(agent->reg_str[0]);
      agent->reg_str[0] = 0;
    }

    if (agent->reg_str[1] != 0) {
      free(agent->reg_str[1]);
      agent->reg_str[1] = 0;
    }

    if (agent->reg_str[2] != 0) {
      free(agent->reg_str[2]);
      agent->reg_str[2] = 0;
```

```c
    }

    free(agent->reg_str);
    agent->reg_str = 0;

  }

  memset(agent->regstr_len, 0, sizeof(uint16_t) * STR_REG_MAX);

  if (agent->code != 0){
    free(agent->code);
    agent->code = 0;
  }
  agent->code_len = 0;
  agent->pc = 0;
  agent->status_flag = 0;

/*  if (agent->rec_msg_id != 0){
    free(agent->rec_msg_id);
    agent->rec_msg_id = 0;

  }
*/
  if (agent->rec_msg_content != 0) {
    free(agent->rec_msg_content);
    agent->rec_msg_content = 0;
    agent->rec_msg_len = 0;
  }

}

uint8_t clone_agent(agent_t *agent){
  uint8_t i = 0;
  uint8_t result = 1;

  for (i = 0; i < AGENT_MAX; i++){
    if (platform.agents[i].status == stopped){
      reset_agent(i);

      platform.agents[i].id = i;
      platform.agents[i].status = ready;
      platform.agents[i].priority = agent->priority;

      if (platform.agents[i].reg_str == 0) {
        platform.agents[i].reg_str = (char**) malloc(STR_REG_MAX * sizeof(char*));
        platform.agents[i].reg_str[0] = (char*) malloc(agent->regstr_len[0]);
        platform.agents[i].reg_str[1] = (char*) malloc(agent->regstr_len[1]);
        platform.agents[i].reg_str[2] = (char*) malloc(agent->regstr_len[2]);
      }

      memcpy(platform.agents[i].reg_str[0], agent->reg_str[0], agent->regstr_len[0]);
      platform.agents[i].regstr_len[0] = agent->regstr_len[0];

```

```
      memcpy(platform.agents[i].reg_str[1], agent->reg_str[1], agent->regstr_len[1]);
      platform.agents[i].regstr_len[1] = agent->regstr_len[1];

      memcpy(platform.agents[i].reg_str[2], agent->reg_str[2], agent->regstr_len[2]);
      platform.agents[i].regstr_len[2] = agent->regstr_len[2];

      platform.agents[i].code_len = agent->code_len;
      platform.agents[i].code = (uint16_t*) malloc( agent->code_len * sizeof(uint16_t));
      memcpy(platform.agents[i].code, agent->code, agent->code_len * sizeof(uint16_t));

      platform.agents[i].pc = agent->pc + 1;
      platform.agents[i].status_flag = agent->status_flag;

      memcpy(platform.agents[i].regs, agent->regs, REG_MAX * sizeof(int16_t));
      platform.agents[i].regs[REG_ACC] = 0;
      result = 0;

      if ( agent->rec_msg_content!= NULL){
        memcpy(platform.agents[i].rec_msg_content, agent->rec_msg_content, agent->rec_msg_len);
        platform.agents[i].rec_msg_len = agent->rec_msg_len;
      }
      break;
    }
  }

  return result;
}

/*
* Initialize the platform with the provided configuration and
* setup all requested drivers.
 */
void platform_init(void) {

#ifdef NODE0
  platform.id = NODE0_ID;
#elif NODE1
  platform.id = NODE1_ID;
#elif NODE2
  platform.id = NODE2_ID;
#elif NODE3
  platform.id = NODE3_ID;
#endif

  init_drivers();
  init_agents();
  sei();
}

void run_platform(void) {

  schedule_next();
```

```c
  //_delay_ms(1000);
}

int main(void) {


  platform_init();

  /*if (platform.drivers.dotmatrix_send != NULL){
    char a1[] = "3";
    platform.drivers.dotmatrix_send(a1);
    _delay_ms(2000);
  }*/

  while (1){
    run_platform();
  }

  return 1;
}
```

A.1.3 scheduler.h

```c
/*
* scheduler.h
*
* Created on: Dec 10, 2012
* Author: igor
*/

#ifndef SCHEDULER_H_
#define SCHEDULER_H_

#include "platform.h"
#include "exe_layer.h"

extern volatile uint8_t last_agent;

void schedule_next(void);

#endif
```

A.1.4 scheduler.c

```c
/*
 * scheduler.c
 *
 * Created on: Dec 10, 2012
 * Author: igor
 */
#include "scheduler.h"

volatile uint8_t next_agent_id = 0;

void schedule_next(void) {

```

```
  uint8_t first_agent_id = next_agent_id;
  uint8_t all_blocked = 0;

  // consider next agent
  agent_t *next_agent = (agent_t*) &(platform.agents[next_agent_id]);

  // search for an unblocked agent
  while (next_agent->status != ready) {
    if (next_agent_id < 3) {
      next_agent_id += 1;
    } else {
      next_agent_id = 0;
    }

    if (first_agent_id == next_agent_id){
      all_blocked = 1;
      break;
    }

    next_agent = (agent_t*) &(platform.agents[next_agent_id]);
  }

  if (all_blocked){
    return;
  }

  // execute the next opcodes for the agent
  uint8_t opcodes_done = execute_agent(next_agent, next_agent->priority +
      1);

  // schedule next agent
  next_agent_id += 1;
}
```

A.1.5 hw_layer.c

```
/*
 * scheduler.c
 *
 * Created on: Dec 10, 2012
 * Author: igor
 */
#include "scheduler.h"

volatile uint8_t next_agent_id = 0;

void schedule_next(void) {

  uint8_t first_agent_id = next_agent_id;
  uint8_t all_blocked = 0;

  // consider next agent
  agent_t *next_agent = (agent_t*) &(platform.agents[next_agent_id]);

  // search for an unblocked agent
  while (next_agent->status != ready) {
    if (next_agent_id < 3) {
```

```
        next_agent_id += 1;
      } else {
        next_agent_id = 0;
      }

      if (first_agent_id == next_agent_id){
        all_blocked = 1;
        break;
      }

      next_agent = (agent_t*) &(platform.agents[next_agent_id]);
    }

    if (all_blocked){
      return;
    }

    // execute the next opcodes for the agent
    uint8_t opcodes_done = execute_agent(next_agent, next_agent->priority + 1);

    // schedule next agent
    next_agent_id += 1;
}
```

A.1.6 hw_layer.c

A.1.7 exe_layer.c

A.1.8 comm_layer.c

A.1.9 hw_layer.c

A.2 Agent assembler tool

A.2.1 asm_agent

A.3 Platform

A.4 Drivers