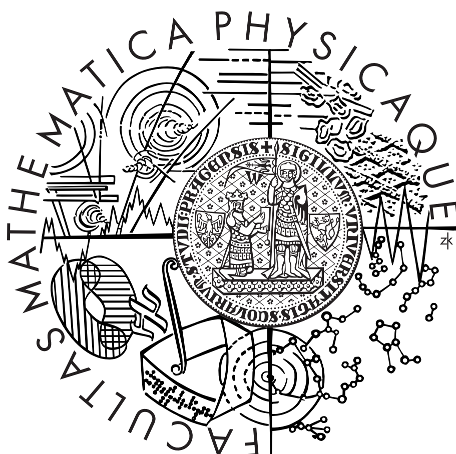
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jiří Královec

Punch Press Simulator

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Informatics
Specialization:	Software Systems

Prague 2015

I would like to express my thanks to my supervisor doc. RNDr. Tomáš Bureš, Ph.D. for guiding me through this Master thesis. Also, I would like to thank my family for their support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, 30th July 2015 signature of the author

Název práce: Punch Press Simulator

Autor: Jiří Královec

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato práce se zabývá návrhem a realizací výukové podpory pro předměty zabývající se systémy reálného času. Záměrem práce je vytvoření platformy, prostřednictvím které budou mít studenti možnost seznámit se s jednotlivými aspekty tvorby software specifickými pro systémy reálného času. Realizovaná platforma sestává ze tří částí – řadiče, řízeného stroje a jeho vizualizace. Stroj reprezentuje konkrétní průmyslový stroj, v tomto případě se jedná o punch press. Řadič umožňuje řízení stroje a vizualizace zobrazuje uživateli jeho aktuální stav. V rámci praktické výuky mají studenti možnost vytvářet programy pro řadič, který daný stroj řídí, vizualizace pak průběžně poskytuje vizuální informaci o stavu stroje a o případném chybném řízení. Platforma je realizovaná jako hardware-in-the-loop simulace, to znamená, že procesor a zařízení řadiče jsou skutečné hardwarové komponenty, ale daný průmyslový stroj je pouze softwarová simulace. Zvolené řešení, kdy reálný průmyslový stroj je nahrazen jeho simulací, má nízké jak pořizovací náklady, tak náklady na provoz a zároveň eliminuje možnost poškození drahého zařízení v případě chybného řídícího programu.

Klíčová slova: systémy reálného času, simulace, zpětnovazební řízení, výuka

Title: Punch Press Simulator

Author: Jiří Královec

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: This work tries to remedy the practical part of teaching development of software for real-time systems. It does so by creation of a platform on which students can practically learn aspects of development of software for real-time systems. The resulting platform consists of a plant, a visualizer and a controller. The plant represents an industrial machine, the visualizer displays the current state of the plant. The controller drives the plant. Students learn by developing a program for the controller. The resulting platform is realized as a hardware-in-the-loop simulation – the controller's processor and devices are real hardware, and the plant is a simulated device. The platform has a low cost, low space requirements and it is not easily breakable.

Keywords: real-time systems, simulation, feedback control, teaching

Contents

1 Introduction **2**
1.1 Goal . . . 2
1.2 Content Summary . . . 3
1.3 Example . . . 4

2 Analysis **5**
2.1 Requirements . . . 5
2.2 Approaches to the Solution . . . 6
2.3 Software-In-The-Loop Simulation . . . 6
2.3.1 Interpretation . . . 7
2.3.2 Relation to Requirements . . . 7
2.4 Processor-In-The-Loop Simulation . . . 8
2.4.1 Relation to Requirements . . . 9
2.5 Hardware-In-The-Loop Simulation . . . 10
2.5.1 Plant on the Development Machine . . . 10
2.5.2 Plant on a Dedicated Machine . . . 10
2.5.3 Relation to Requirements . . . 11
2.6 No Simulation . . . 12
2.6.1 Use of Toy Machines . . . 12
2.6.2 Relation to Requirements . . . 12
2.7 Solution Strategy . . . 13
2.7.1 Further Analysis Of The Selected Solution . . . 13
2.7.2 Total costs . . . 14
2.8 Detail Goals . . . 15

3 Plant-Controller Interface **16**
3.1 Base Specification . . . 16
3.2 Technical Analysis . . . 17
3.2.1 Plant Restart . . . 17
3.3 Protocol over SPI . . . 18
3.3.1 Example . . . 19
3.3.2 Timing . . . 20
3.3.3 SPI Configuration . . . 20

4 Plant-Visualizer Interface **22**
4.1 User Interface Analysis . . . 22
4.2 Technical Analysis . . . 22
4.3 Messaging Protocol . . . 23
4.4 Messages . . . 25

5 Hardware Connection **28**
5.1 Usage . . . 28
5.2 Schema . . . 28

6 Plant 31
6.1 Technical Analysis . . . 31
6.2 Architecture . . . 33
6.3 Simulator Module . . . 34
6.3.1 Simulator . . . 35
6.3.2 Main . . . 37
6.3.3 User-space Interface . . . 38
6.4 High Priority Interrupts . . . 38
6.4.1 Spurious Interrupt Requests . . . 39
6.4.2 Testing . . . 40
6.5 SPI Driver Module . . . 40
6.6 SPI Controller . . . 41
6.7 PRU Driver Module . . . 41
6.8 Visualizer Interface . . . 42

7 Visualizer 44
7.1 Architecture . . . 44
7.2 Usage . . . 44

8 Base Controller Project 47
8.1 Dependencies . . . 47
8.2 Basic Usage . . . 47

9 Evaluation 49
9.1 Controller Program . . . 49

10 Related Work 52
10.1 Educational Platforms . . . 52
10.2 Hardware-In-The-Loop Simulators . . . 53

11 Conclusion 54

Bibliography 56

List of Abbreviations 59

A Contents of the Attached Disc 60

B Plant Interface Specification 62

1 Introduction

Use of real-time systems is widespread today and numbers of these systems are growing. We meet them in everyday life in cars, cell phones, point-of-sale terminals, washing machines etc. and they are equally prolific in industrial use driving robots, whole factory sections, rockets, ships or planes. These systems deal with many kinds of problems different from the problems of non-real-time systems. The software of real-time systems has to deal with specific problems as well (timing requirements – reaction to events in a strictly limited time interval; event driven programs; multitasking – cooperation of several subtasks in order to complete the main task; dealing directly with hardware, often specialized hardware; incorrect programs damaging machinery or lives; feedback control of dynamic systems). Because of the specifics, the development of software for real-time systems requires specific approaches.

Many universities have specialized courses aimed at teaching development of software for real-time systems. Practical exercises are the typical problem all such courses must face. They are necessary so that students may try out and thus adopt a variety of specific techniques. To be as close as possible to the real world development, we would like students to learn on exercises that are close to industrial problems.

Some universities use real industrial robots as a learning tool in real-time systems courses. Main drawbacks of this solution are high cost of acquisition and maintenance. The space necessary to place such a machine in is usually also limiting. It is usually impossible to have an industrial robot for each student because of the high costs and space requirements. Additionally, in order to control the machine students must understand the fundamental electromechanical properties of it. This is another drawback, it is time consuming and turns attention away from the main purpose of the course.

Another case, used by courses, is not using any additional hardware at all. These courses run students' programs on ordinary desktop computers and the programs control only simulated hardware. This solution has the advantage of not requiring much maintenance and every student can run a simulation on his development machine. Major drawback here is that students are far away from the real devices. Neither of these widely used solutions is ideal. We would like to have a more balanced setup. The setup should be low cost but, give students the necessary hands-on experience and allow them to focus on software development.

1.1 Goal

The goal of this thesis is to help with practical teaching of embedded and real-time systems. We want students to get a hands-on experience with programming hardware commonly used for real-time applications and usage of devices (timers, bus controllers). Also, we want them to learn feedback control of dynamic systems with real-time requirements on a real-life industrial example. To achieve the goal,

we design and implement a platform that will enable it. We want to be able to get the platform for each student in a class.

The platform will be realized as a hardware-in-the-loop simulation, meaning that it will contain a real processor and devices, but the machine that will be controlled (so called plant) will be simulated. The details will be discussed more later in Chapter 2. The controlled plant will be based on a punch press, an industrial machine used to cut holes in sheet metal. The resulting platform will consist of inexpensive hardware (*BeagleBone Black* board, *stm32f4-discovery* board, hardware connection between the boards and connection to a desktop computer) and software (free software and our software). Only a common desktop computer with no additional hardware will be required in order to use the platform. The platform will thus be easily used in school laboratories or with students' own computers.

1.2 Content Summary

Text of the thesis excluding the introduction and the conclusion is divided into following parts.

- *Chapter two – Analysis* – analyzes available implementation strategies and compares them.
- *Chapter three and four* design interfaces between individual hardware parts of the solution, plant and controller, and plant and visualizer respectively. The chapters analyze possible hardware interconnections, select one, and design a protocol based on it.
- *Chapters five through seven* deal with the implementation of individual parts of the solution. They contain technical background, description of the solution and implementation decisions. Additionally, their purpose is to allow for extension of the platform by others or reuse of it in other work.
- *Chapter eight – Base Controller Project* – describes the project prepared for students to allow them to start developing without time consuming selection of appropriate tools and libraries beforehand.
- *Chapter nine – Evaluation* – contains evaluation of the created platform.
- *Chapter ten – Related Work* – summarizes other works that deal with the same or similar problems, and compares them with this work.

At the end of this text are attached two appendices. Appendix A describes the content of the attached disc, which contains implemented programs and programs and libraries that we reused. Appendix B contains a punch press interface specification, from which we derive in Chapter 3.

1.3 Example

One example of a specialized real-time course is the *Embedded and Real-Time Systems* at *Faculty of Mathematics and Physics at Charles University in Prague*. Current laboratory classes of the *Embedded and Real-Time Systems* course are divided into four assignments, each of which is further divided into smaller exercises.

- During the first assignment elaboration students learn to use a real-time operating system *RTEMS* and to apply different scheduling approaches using it.
- The second assignment builds on the knowledge gained in the first assignment (students again use the *RTEMS* operating system in the same environment) and teaches feedback control of a dynamic system by creating a controller for a punch press machine.
- The third assignment contains the same feedback control problem as the second assignment. This time students solve it by model driven development in the *Simulink* program.
- During the fourth assignment elaboration students use *LEGO Mindstorms* kit to build a simple plotter device.

The third assignment is focused on modeling rather than programming. In the fourth exercise students run their programs on a real toy hardware of *LEGO Mindstorms*. However during elaboration of the other two assignments students only run their programs in an emulator of *i386* processor with simulated hardware devices. The platform created during elaboration of this thesis helps to create an environment for solution of these assignments, which will be much closer to the real-life environment than the current setup is. Using the platform students will create programs for a real processor designed for use in embedded and real-time systems and the programs will control real hardware devices. Students will deploy and test the resulting programs on the processor and run them in real-time.

2 Analysis

The purpose of this chapter is to analyze possible strategies to implement the desired platform. The desired platform is specified in Chapter 1. We first specify the requirements we want the solution to cover, then we enumerate possible solution strategies. Finally we choose the strategy used for the implementation. The strategy that will be used in the implementation is then analyzed further for the purpose of the particular implementation. The implementation itself is described in the following chapters.

2.1 Requirements

In Chapter 1 we outlined why and what we want to accomplish. However, for the purpose of analysis, we need to describe our goals in more detail. We specify the goals as a set of requirements that the result should fulfill. Using the requirements we will be able to judge the different proposed strategies to select the most suitable one. The requirements follow below.

- *Low cost* – the resulting platform should have low cost of acquisition and maintenance costs. We want to be able to have an instance of the platform for each student in a class and easily replace anything that gets broken.
- *Low space requirements* – the result should not occupy a lot of space. A typical computer laboratory does not have enough space to host an industrial robot for each student. We would also like students to have an opportunity to take the platform home, to test and use it themselves.
- *Portability* – the development platform we aim at is a desktop computer with a *Linux* based operating system, but the platform should be easily portable to other desktop platforms as well (*Windows*, *Mac OS X*).
- *Multiple learning steps* - we would like to allow students to go through multiple learning steps on the same platform.
- *Durability* – the hardware of the platform should be tough to break or at least easily and cheaply repairable, we expect students to make mistakes while using it.
- *Hands-on experience with hardware* – students should get close to hardware (low level programming, control of real devices).
- *Run in real-time* – students' programs should run in real time as real life real-time programs do.
- *Testing* – we want to be able to easily (automatically) test that the behavior of the controller programs made by students is correct. This is useful both for students and for teachers when they are grading students' works.

- *Debugging* – debugging a real-time program is different from debugging a usual computer program. We want students to experience the differences to non-real-time systems.
- *RTOS* – usage of real-time operating system on the controller has to be possible. (Even though we do not use real-time operating system in this work, we want to be able to easily add it later.)

2.2 Approaches to the Solution

Following sections from 2.3 to 2.6 each correspond to one possible implementation strategy. The proposed solutions are divided into a controller, a plant and a visualizer. The plant represents an industrial machine to be controlled, the visualizer displays the current state of the machine (and is not necessary when the state of the machine is directly observable). The controller is the part of the system which controls the plant. Programs of students run on the controller. The controller consists of a processor and devices (timers, floating point coprocessors, *I/O*, etc.). The controller program has to communicate with the plant through devices.

Both hardware and software of the plant, the visualizer and the connection of all the parts will be provided by this thesis. The controller will be provided as a hardware only, software for the controller will be created by students that will use the platform.

We want students to get a hands-on experience with real hardware, but that does not necessarily mean that we have to use real hardware only. It is possible to have parts (or the whole) of the solution realized in software or hardware and both types have some advantages and disadvantages. Different proposed implementation strategies differ in different parts of the solution realized by hardware or software. The following sections are ordered by how much of hardware is employed in the solution.

2.3 Software-In-The-Loop Simulation

Software-in-the-loop simulation means that we simulate all parts that the controller program sees, the devices, the plant and the visualizer. It is depicted in Figure 2.1. Controller programs are written as ordinary programs to be run as a normal process inside an operating system, usually the development machine. To emulate the environment of a real-time system, the controller program uses a special library. It is up to the library to make the system look real-time like. The library can emulate devices directly connected to the controller and the behavior of a direct processor usage, for example interrupts.

The library can also simulate the real-time features on a higher level and behave like a whole real-time operating system with drivers rather than a raw processor

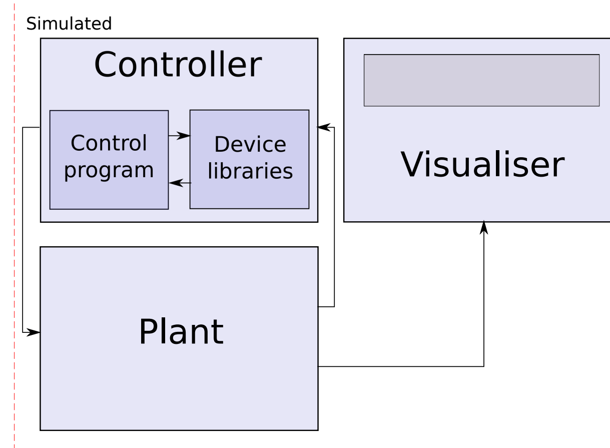


Figure 2.1: Software-in-the-loop Simulation Diagram

and devices. Such higher level library would not have to exactly emulate behavior of the low level devices.

The plant and the visualizer are simulated in this setup, they are another libraries or programs connected to the main library. Not simulating the plant and visualizer would be also possible, but would not bring any advantages. The hardware connection of the plant to the development machine would probably be non-trivial, and probably would also require alterations of the operating systems of the development machine. Both of these would significantly harm portability. Additional disadvantages (cost, space requirements, durability) of usage of a non-simulated plant are further described in the Section 2.6.

2.3.1 Interpretation

Instead of running the controller program directly we can interpret it. Interpreting gives us more control over the program and allows us to run the program in a simulated time instead of running it according to the operating system. That allows us to emulate devices more easily and be able to run in real-time from the point of view of the controller program. However, we need an additional software component, an interpreter that will interpret the program. Running in simulated time is not what we aim at.

2.3.2 Relation to Requirements

This design is the least demanding on additional hardware. We only need a development machine to run it. Hence, it does not have any additional hardware costs or space requirements and no maintenance costs. If only free software and software we create is used, it does not have any additional costs per student. The solution is durable as it has no physical parts that can wear out or be physically broken.

Portability to other platforms depends on the software used. In comparison to other strategies, portability is worse as all the components of the platform are

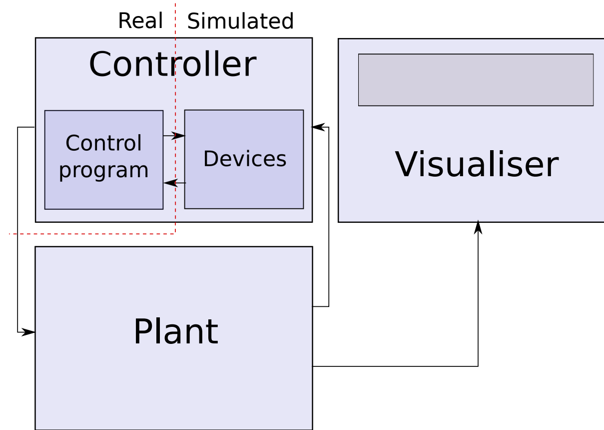


Figure 2.2: Processor-in-the-loop Simulation Diagram

software components and all of them have to be ported if we move to another development platform. Since no additional hardware is necessary, it is easy to change the platform. For example it is possible to extend the development environment by extending the devices library. It is hard, if not impossible, to mimic the behavior of existing complex hardware devices exactly, because the devices are only a library and, for example, the library cannot immediately react to changes on a certain location in memory as some real devices do. As all the hardware is simulated, it is under our control, hence we can easily check that it is used correctly and check all the results the controller program creates.

Main problem is, that students get no hands-on experience with real hardware. The controller program runs as a process in an operating system and thus it does not have any contact with direct hardware interaction and run in real-time. Debugging is the same as in other non-real-time programs.

An interpreted program has the additional advantage that from the point of the controller program, it might seem that it runs in real-time. Implementation of devices is easier because we are no longer limited by what the operating system offers, but are allowed to use the interpreter as well. Additional software component is necessary and other disadvantages of the software-in-the-loop simulation remain the same.

2.4 Processor-In-The-Loop Simulation

Processor-in-the-loop simulation is similar to the software-in-the-loop simulation. It is depicted in Figure 2.2. The devices, plant and visualizer are still realized as software components. The processor of the controller is a real hardware intended for usage in real-time systems. The controller program is run directly on the processor as a sole user of it instead of running as a process in an operating system and sharing resources with other processes. A real-time operating system can also be utilized and be part of the controller program. The program can also be run in a virtual machine emulating the processor, instead of a real processor, in this case all the previous statements still apply.

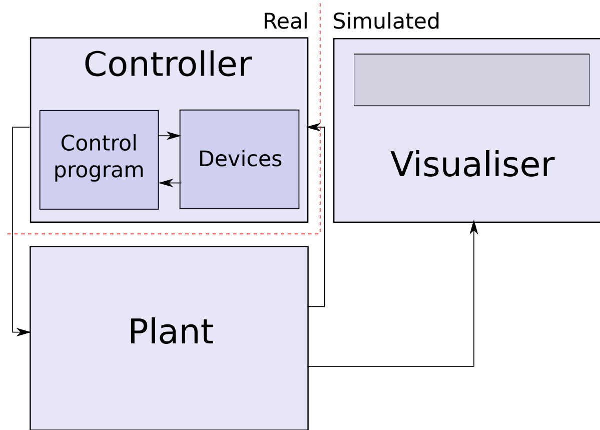


Figure 2.3: Hardware-in-the-loop Simulation Diagram

2.4.1 Relation to Requirements

In this design not much hardware is necessary, a processor and a programmer for the processor are sufficient. We thus still have a low hardware costs and expect to have the same software cost as in the case of the software-in-the-loop simulation. The addition of the processor and a programmer does not have any significant impact on the space used, they are both smaller than a common development machine. Durability of the solution is worse than in the software-in-the-loop case as we use additional hardware, but it is still very high. The programmer or the processor usually cannot be broken by incorrect controller programs.

The program is compiled for and run directly on a processor with the highest privileges. Students thus experience the features of a real processor, for example interrupts, running in real-time, different behavior in different modes and constraints caused by the lack of standard libraries and operating system. It is also possible to utilize a real-time operating system, which usually contains many concepts from the real-time systems theory. If we use a real processor they also get a hands-on experience with low-level debugging.

Processor-in-the-loop approach eliminates some of the disadvantages of the software-in-the-loop approach, but some remain. The main one being non-existence of real devices.

When a virtualized processor is used we need no additional hardware. We thus have almost all the advantages of the software-in-the-loop approach with better durability and no space requirements. The only additional component we need is the emulator. There are free emulators available, capable of emulating real-time processors (for example *QEMU*). Programming of the processor and debugging of the controller program are usually easier and students do not get to use hardware programmers and debuggers.

2.5 Hardware-In-The-Loop Simulation

In comparison to the processor-in-the-loop simulation, hardware-in-the-loop simulation exchanges another simulated component of the system with a real one. The whole controller part including devices is real, this also means that the connection with the plant must be done in hardware. The plant is simulated and the visualizer is a software component. Only the plant and the visualizer are software components. This approach is depicted in Figure 2.3.

Addition of hardware brings us additional costs of the processor, devices (today usually in one chip with processor), a programmer and a debugger for the processor. The plant has to have a hardware interface that the controller uses. The plant must run in real-time, because the controller and devices run in real-time. There are two options for the platform of the plant, either we run the plant on the development machine or we devote a dedicated machine to it.

2.5.1 Plant on the Development Machine

Nowadays, the development machine is typically a powerful desktop computer with enough computational power to host the plant simulation. The problem is connection of the a controller to it and reaction to the controller's signals in time. Reaction times within the range of microseconds are not easily achievable with non-real-time operating systems usual for development machines. Connection of the controller presents another problem. Even though the development machine is usually able to communicate over buses used in real-time systems, it might not be possible for us to utilize them. They are usually already in use and possible additional usage by the controller program (that can be improper) may cause the development machine to not work properly. A direct connection would thus not be possible, we would have to create (or use an existing) an interface board as a bridge between the controller and the development machine. The interface board would then allow us to have a plant interface that is common in real-time systems, insulate the controller from the development machine, so that it cannot affect it in the wrong way, and have real-time response to the controller's signals.

With the plant simulation on the development computer, the visualizer is just another software component on the same computer. There are many ways to connect two software components on the same computer and the visualizer can display the plant's state with low latency.

2.5.2 Plant on a Dedicated Machine

Having the plant on a dedicated machine presents additional costs but allows us to use a suitable hardware and have full control over it. Suitable hardware must contain the necessary buses, inputs and outputs that the controller needs. Also, we need a full control over them to allow for quick reactions to the controller stimuli.

With the plant running on a dedicated machine we have two options regarding

the visualizer's machine. If the plant's platform is capable of running both the plant and the visualizer and enables connection to a display, we may run both components on the same machine as in the previous case. In this case the solution needs an additional display. The other option is to run the visualizer on the development machine. This requires to design and implement an additional connection between the plant and the development machine and additional software that will transfer the plant's state to the visualizer. The solution is thus more complex.

2.5.3 Relation to Requirements

The main benefit of this solution, with respect to the previous one, is that students get a hands-on experience with hardware. Students program a real controller, that uses only real devices and the simulated plant is accessible only through real devices. All the hardware the controller program uses directly is real. Depending on the precision of the simulation it might not be possible for the controller program to tell the difference between hardware-in-the-loop simulation and no simulation.

The controller program runs in real-time and students also get experience with low-level debugging. The solution is still durable as it has no moving parts, that can wear out quickly, and cannot be broken by inappropriate controller programs. The plant cannot be broken by an inappropriate controller program, because it is simulated. Extensibility highly depends on the exact hardware and software configuration used. We cannot check that the controller uses its close devices correctly, but we still have the plant under full control and can check that it is being used correctly. The solution still can have a low hardware price starting at several hundreds of dollars (price of a cheap dedicated computer and a display). It has a low space requirements with the thing requiring the most space being an additional display, if we decide to use it.

The "plant on the development machine" option has a bad portability. It requires additional hardware connected to the development machine through fast buses and quick reactions to the controller's stimuli. That would certainly require alterations of the operating system, which have bad portability.

With the plant running on a dedicated machine, the portability is significantly better. When using another development machine we use the same plant. If we have a dedicated display connected directly to the plant and run the visualizer on it, then it requires no porting. If we run the visualizer on the development machine, then we have to port the connection and the visualizer application to the new machine. The connection can be done using common buses (USB, serial port), because we do not require fast communication and portability thus mostly depends on the visualizer application.

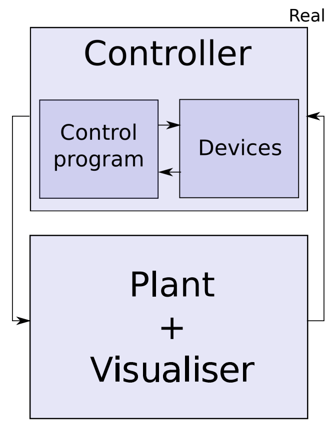


Figure 2.4: No Simulation Diagram

2.6 No Simulation

The last approach we consider is not using any simulation at all, it is depicted in Figure 2.4. In this case the controller and the plant are both hardware components, the visualizer might or might not be a hardware component. In most cases the plant is also the visualizer for example it is the case for industrial robots, where we can directly see and measure results produced by their work. That is why both the plant and visualizer are represented as one block in the diagram. In some cases a special visualizer is necessary, for example in the case of driving an electrical motor we cannot directly see how fast it is, but we can use sensors to measure the speed and then display it.

2.6.1 Use of Toy Machines

Instead of using a real world industrial machine as a plant, we might use a toy imitation of such a machine. It would, however, mean that we have to design, create and then also maintain it ourselves.

2.6.2 Relation to Requirements

This solution strategy gives students the best hands-on experience with real hardware as both the controller and the plant are hardware components. Implementation of the solution is the simplest as it does not require us to design and implement additional software components. We only have to create a connection between the controller and the plant.

Automatic testing of controller programs is dependent on the plant used. Porting to another development platform is easier than in other cases as it only requires controller development tools to be portable, no other connection to the development machine or a software for it is necessary. Durability of the solution is significantly lower than in the previous cases as we expect the plant to have moving parts, which wear out faster than electronic components and can be damaged

Table 2.1: Considered controller boards and their prices

Name	*Processor*	*Approximate price*
stm32f4 discovery	32bit ARM Cortex-M4F	$15
TIVA C LAUNCHPAD	32bit ARM Cortex-M4F	$14
PIC Clicker	8bit PIC, PIC18F47J53	$19
ARDUINO UNO R3	8bit AVR, ATmega328	$23

by improper controller programs.

Main disadvantage of this solution are costs. Costs necessary to acquire the plant, maintain it and repair it, if improperly used, are high. Each plant also usually takes up a lot of space. Hence it is unlikely that we would be able to get one for each student or that students would be able to take the platform home.

To control a real plant, understanding its electrical and mechanical properties is necessary. This requires a lot of additional knowledge students has to get, that is time consuming, and it strays the attention away from the main aim, which is learning real-time systems software development. Thus, despite of its advantages, this approach is not very suitable for teaching real-time systems.

Usage of toy machines lowers the costs involved and the space necessary for the plant, however other disadvantages remain the same.

2.7 Solution Strategy

From the enumerated strategies we chose the hardware-in-the-loop simulator with a plant running on a dedicated machine and a visualizer on the development machine. This option satisfies our requirements the most.

We have to acquire additional hardware, but the hardware is low cost, see Section 2.7.1 for more details. It is thus not a problem to have the setup for each student. It is possible to make the solution durable and prohibit damaging of the hardware by inappropriate controller programming. It is of course not possible to prevent all physical damage. The solution will have no parts that easily wear out or can be broken by a bad controller program, therefore we get low maintenance cost.

2.7.1 Further Analysis Of The Selected Solution

To verify the low cost of the solution we choose the suitable controller and plant hardware in this section. An obvious candidates for a controller board are *ARM* based boards with processors aimed at embedded and real-time systems. We might also use a board with an *AVR* or *PIC* processors, they would probably be sufficient as controllers but are not powerful enough to host a real-time operating system. Some of the possible boards are listed in Table 2.1, for comparison also include *PIC* and *AVR* platforms.

Table 2.2: Considered plant boards and their prices

Name	*Approximate price*
Raspberry Pi (Model B+)	$25
Raspberry Pi 2	$35
PandaBoard	$185
Intel Galileo Gen 2	$158
Intel Minnow	$99
UDOO	$99
BeagleBoard-XM	$149
BeagleBone Black	$53

From these we chose the *stm32f4 discovery*. It is a cheap board, its processor contains a lot of devices and the board contains additional devices (LEDs, button). It has enough *I/O* pins and supports many types of buses and other peripherals. Using the chosen board, students can go through multiple learning steps (for example: blink a LED on the controller board; use a timer to blink the LED with a specified period; read the state of the plant; react to button push; drive the plant using feedback control).

The plant board we need a sufficiently powerful board, that will be able to connect to the controller board at signal level. The processor of the board has to react to signals in real-time and allow the plant simulation to run with sufficient speed. The boards we considered for this purpose are listed in Table 2.2.

We chose the *BeagleBone Black* board, it is not the cheapest board but we consider it to be cheap. It is a board with a powerful processor with many unused *I/O* pins and several modules for buses that are also unused on the board. Many of the *I/O* pins and buses are exposed on the connectors of the board. Also manual for the processor is freely available to anyone, this is not the case of some other boards, for example *Raspberry Pi*. We need to use the devices on the processor directly, hence we will need the manual during implementation.

We have only considered boards that are both cheap and with performance oriented processors. More power boards such as Intel-based boards were disqualified due to their high prices and the fact that these boards also do not have their *I/O* pins or *SPI* buses easily accessible or already use them for other purposes.

2.7.2 Total costs

In the solution only free software and software that we create will be used, software costs thus will be zero. The selected boards together cost around $68, the necessary components to create the platform according to Chapter 5 (cables, resistors, transistor, capacitors, integrated circuit) cost under $10. Additional component, that might be necessary in order to connect the platform to a development machine, is a *USB-SerialPort* converter with price around $10. The resulting price will thus be under $100 per student.

The strategies presented in Section 2.3 and Section 2.4 would have lower hardware

costs, expected to be close to zero, but they offer no or almost no hands-on experience with real hardware and connected problems. The strategy presented in Section 2.6 offers the same hands-on experience with the controller hardware and offers a real plant, but at much higher costs, which usually prevent schools from having the solution for every student. The selected solution thus has the best performance to price ratio.

2.8 Detail Goals

With the design of the solution selected we can now specify more precisely the steps necessary for its implementation. List of the detail goals follows, each of the next steps must of course fulfill the earlier specified requirements that will be affected by the implementation most notably portability, durability and testing requirements.

- *Design the protocol used for communication between the plant and the controller* – selection of hardware used for communication and a design of a protocol on top of it.
- *Design the protocol used for communication between the plant and the visualizer* – selection of hardware used for communication and a design of a protocol on top of it.
- *Connect individual parts of the solution* – design particular hardware connection of individual parts of the solution, which processor/board pins will be connected, how and why.
- *Implement the plant* – design and implementation of the software component of the plant.
- *Implement the visualizer* – design and implementation of the software component of the visualizer.
- *Evaluate the whole solution* – evaluate the whole solution by connection of all parts and implementation of a controller for the plant.
- *Prepare students' environment* – prepare an environment that will be used by students. Students will use the environment so they can start development quickly, with prepared and validated tools.

3 Plant-Controller Interface

Before we start implementing the solution, we have to decide how different parts of it will interface each other. The most important is the interface of the plant used by the controller, the interface will be used by students in their programs. This chapter designs the interface and serves as a reference for implementation of the plant and a detailed user's guide for controller implementation.

The plant is a punch press, an industrial machine used to cut holes in a sheet metal. We would like the interface, through which the controller uses the plant, to resemble an interface of an existing punch press, but primarily we want it to be suitable for teaching. That is why we base our interface on an existing one, already in use in the *Embedded and Real-Time Systems* course at *Faculty of Mathematics and Physics at Charles University in Prague*. The interface we base on, together with full behavior of the machine, is described in [1] (also available in Appendix B).

3.1 Base Specification

The punch press machine as defined by [1] consists of a working area, motors and a head. The working area is a rectangular plane of a fixed size see Figure 3.1 with a strip of safe zones around it. The position of the head can be affected by two motors, one for each axis. The motors are controlled by setting their power. When the head is stopped, it can punch a hole in the working area. The controller observes state of the plant trough reading its sensors – quadratic encoders, safe zones, head state, error. The controller drives the plant by setting its actuators – motor power, interrupt enabling.

In the existing interface, communication with the plant is done using memory mapped registers and a dedicated interrupt request. The dedicated interrupt is used to inform the controller that the visible (for the controller) state of the plant changed. The memory mapped registers allow reading of the sensors of the plant and setting the actuators of the plant.

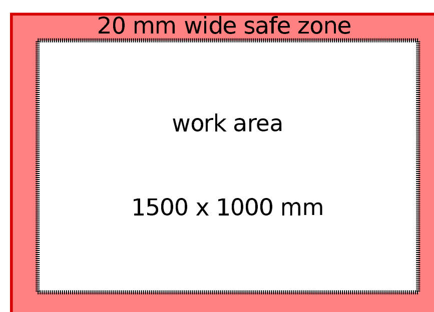


Figure 3.1: Punch press working area

3.2 Technical Analysis

On the plant's processor, communication using memory mapped registers and dedicated interrupt requests are usually used only by devices that are part of the processor. It is not easily possible to add another memory mapped device and a dedicated interrupt. Further more, it is not a common method to connect two separate hardware devices, hence we do not use it. There are other connection options that are commonly used to connect two devices. The connection options we have considered follow below.

- *UART* – provides point-to-point communication. It has problems with synchronization when sending multiple byte messages, slower than other considered options.
- *I2C* – is a master-slave bus. It is fast, a master can communicate with any slave, enables multiple masters.
- *SPI* – is a master-slave bus. It is fast and single master bus.
- *GPIO* – is suitable for transmitting single bit data.For longer messages requires too many pins or an additional protocol, therefore not commonly used to transmit complex information.
- *USB* – not suitable for real-time applications, it is unnecessarily complex.
- *Ethernet* – not suitable for real-time applications, it is unnecessarily complex.

As a replacement for the base interface interrupt we chose a single line of a *GPIO*. The level of the signal won't carry any information, a level change (edge) of the signal will mean a change in the plant's visible state (visible by the controller). This will allow the controller to listen to the signal line and detect any change.

We chose *SPI* as the main means for communication, which will replace the memory mapped registers of the base interface. The exact protocol used is described in Section 3.3.

3.2.1 Plant Restart

In the process of development of a controller program for the plant, we expect the controller program to contain errors. The program might cause breakage of the simulated plant. We want the controller to always start running with a working plant, not affected by any previous activity. The specification of the base interface does not cover this problem. To do this automatically without any interaction from the controller program, we use the *NRST* pin of the plant's board (*stm32f4-discovery*). The pin is high when the board's processor is running, and low when it is not. Hence, we detect the processor's start by detecting a rising edge on the pin and we detect that the processor has stopped by detecting a falling edge. We prefer to reset the plant when the processor stops, this way there is more time to perform the plant's reset before the controller is running again.

3.3 Protocol over SPI

SPI (*Serial Peripherial Interface*) is a de-facto standard and does not have any official specification, more details can be found for example in [2]. It is a master-slave bus, with one master and one or more slaves. The master initiates and controls the communication. It is a synchronous serial communication interface, the communication is full-duplex or half-duplex. The master selects a slave by a corresponding signal and then periodically sends pulses over the clock signal. On every pulse one bit of a word is both sent and received, received only or sent only by the master, depending on the configuration. When the master sends a bit, the slave receives it, and vice versa. After one word is sent, another words may be sent in the same manner with the same slave selected, or the slave might be deselected.

While the boards selected as plant and controller boards allow us to send 8 or 16 bit long words over *SPI*, we choose 8 bit words to allow usage of more kinds of controllers. *SPI* sends one word at a time, with 8 bit words it is not possible to read and write all the sensors and actuators in a single word transfer. We thus have to design a more complex protocol.

In our case, the controller is the *SPI* master and the plant is the *SPI* slave. As we need both to read the state of plant and set its actuators, we use the *SPI* in full-duplex mode, one word is always sent to and from the slave at the same time. We define the protocol as a set of commands. The controller sends commands and the plant obeys them. A command may have a parameter. If a command has a parameter, then the controller always has to send it immediately after the command. A command may also have a return value. If a command has a return value the controller receives it in the immediately following word. The value read by the controller after anything else than a command with a return value is undefined. All commands, parameters and return values are always sent as one word.

The controller always has to send a valid command or a parameter to the previous command, otherwise the behavior of the plant is undefined. Definitions of valid commands follow. Each of the following items begins with the numeric value of the command and a name in brackets followed by a description.

- 0 (*nop*) – Command with no effect, enables retrieval of a return value of the previous command without affecting the plant. Has no parameters nor return value.
- 1 (*encoders*) – Command for reading the quadratic encoders, which encode the position of both of the plant's motors. Using the encoders the controller is able to detect that the head has moved and the direction of the movement. Each encoder change means movement of a quarter of a millimeter. Both the encoders have two bits. If the motor is moving forward they generate the sequence 00, 01, 11, 10, if a motor is moving backwards they generate the reverse of the sequence. The return value of the command is described in Table 3.1.

Table 3.1: Depiction of *encoders* command return value.

7	6	5	4	3	2	1	0
-	-	-	-	Y encoder		X encoder	

- 2 (*errors*) – Command for reading the error status of the plant. It has no parameters. The return value of the command says whether the punching head is down, whether the plant is broken (fail signal) and for each safe zone whether the head is in it. Safe zone value is true if the center of the head is in the corresponding safe zone. The fail signal is set when the center of the head gets outside of the working area and safe zones or the punching constraints are not satisfied. When the fail signal is set, the plant is broken and a recovery is impossible. The return value of the command is described in Table 3.2.

Table 3.2: Depiction of *errors* command return value.

7	6	5	4	3	2	1	0
-	-	bottom safe zone	top safe zone	right safe zone	left safe zone	fail	head down

- 3 (*punch*) – Starts a punching sequence. After a punching sequence is completed, a punch is created. It has no parameters or a return value. To issue this command, another punching sequence must not be in progress, otherwise it results in a failure of the plant. The punching head must be still during the whole punching sequence, otherwise it results in a failure of the plant. For the head to be still its speed has to be lower than 0.0001 m/s (in each axis).

- 4 (*pwrx*) – Command for setting power to the x axis motor. It has one parameter and no return value. The parameter represents the power to be set, it is from range [-128, 127].

- 5 (*pwry*) – Works the same as the *pwrx* command, but sets power for the y axis motor.

- 6 (*irq-en*) – Enables or disables signaling of the changes by the plant. It has one parameter and no return value. The parameter can be zero or one, zero disables the signaling, one enables it.

3.3.1 Example

Figure 3.2 illustrates the protocol usage on an example. The example moves the head into the left safe zone and performs a punch there. It is written in a pseudocode based on the *C* language. The *sleep* function performs waiting for given number of seconds. The *spi* function sends the given number and returns the number that was received. The *select_plant* function lowers the slave select line of the plant.

```
// lower the slave select line of the plant
select_plant();
// move the head into the left safe zone
spi(4); // pwrx command
spi(-20);
spi(2);
while (!(spi(2) & 0x4))
   ;
spi(4); // pwrx command
spi(0);

// wait until the head is stopped
spi(1); // encoders command
unsigned int enc = spi(0) & 0x3;
while (1) {
   sleep(10); // sleep for one second
   spi(1);
   unsigned int enc2 = spi(0) & 0x3;
   if (enc == enc2)
      break;
   enc = enc2
}

spi(3); // start punching
```

Figure 3.2: Example code listing

3.3.2 Timing

During communication with the plant over *SPI*, certain timing requirements have to be satisfied. When the timing requirements are not satisfied, the behavior of the plant is undefined. Satisfaction of the timing requirements is up to the master as the slave has no means to influence it. The timing requirements are explained graphically in Figure 3.3, times *T1*, *T2* and *T3* all has to be at least 1 microsecond. *T1* is the time interval between lowering the slave select line and the start of transmission of the first word. *T2* is the time interval between the end of one word and the start of the next one. *T3* time interval stars when the slave select is pulled up and ends when it is lowered down. The timing requirements are inferred from the plant's implementation.

3.3.3 SPI Configuration

The *SPI* communication has several parameters. To allow both parts to exchange words successfully, we have to specify these parameters so that they are the same on both of them. The slave select is active low. The clock signal is active low and data are latched on even edges of the clock. In a word transmission the most significant bit is sent first. Maximum clock frequency is 48 MHz.

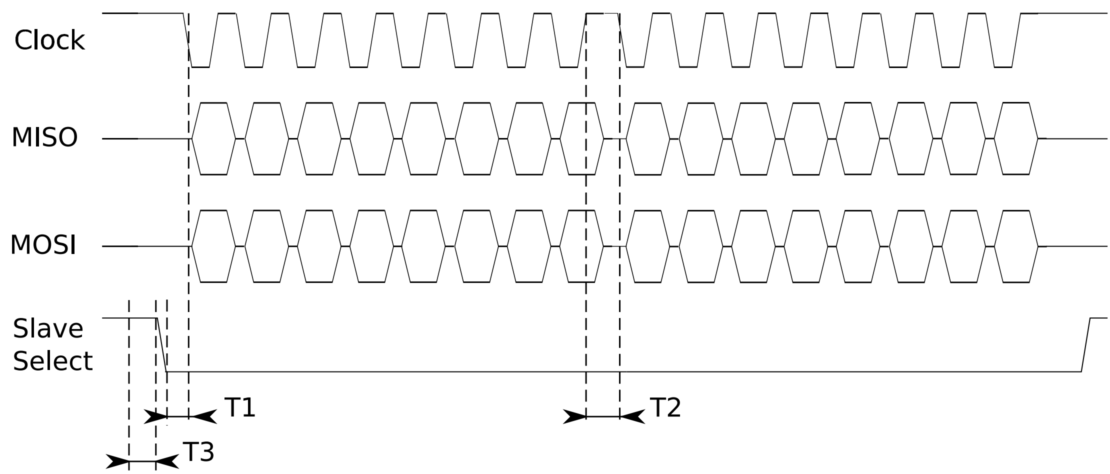


Figure 3.3: SPI Timing

4 Plant-Visualizer Interface

As the plant is a simulated device it does not have any real machine parts. The state of the plant is thus not directly observable using human senses or measurable. However, we need the state of the plant to be easily observable so that students are able to see what the plant does under influence of their controller programs. We also want to be able to log the behavior of the plant, at least punched holes and errors, to check that the holes were punched correctly and to check that the controller program is correct by both students and teachers who grade students' programs.

4.1 User Interface Analysis

We have decided to display the state of the plant on the development computer. We have two options, display the state in a textual form or in a graphical form. Using only textual output would be easier and would allow us to print the state on a console, and then display the state on any computer with a console emulator. We would not have to create additional program and we could connect to the plant in several ways (*ssh* connected through *Ethernet*; serial terminal program connected through *serial port*). But a textual form of the state would be hard to read, hence we decided to use a graphical form. This way we will have to create an additional program that will display the state graphically and design a communication protocol for communication between the plant and the visualizer.

4.2 Technical Analysis

First we select a suitable hardware connection of the plant and the visualizer. On top of the hardware connection we design the communication protocol. We want the hardware connection to be available on most of the desktop computers and easy to setup. The list below contains considered options.

- *USB* – it is fast and available on every development machine. However, it requires an operating system driver on the development machine side. The driver is not easily portable to other systems and might need an update together with system updates.

- *Ethernet* – it is fast and requires connection either directly to the development machine or the local area network. Connection directly to the development machine is problematic as it usually has only one *Ethernet* connector and thus would be disconnected from the local network in order to connect the plant. Disconnecting the network connection is not usually possible. Connection of the plant to the local area network is also not easy. Problematic is the address of the plant, we need to know it in order to connect to the plant, and it has to be different for different networks and

different for each plant on the same network (and depends on the *DHCP* if used).

- *Serial port* – it is, slower but available on many development machines. Adapters providing serial connection over *USB* adapter are usable on machines without *serial port*. It requires access to the file representing the port (on *Unix* systems) or *COM* port (on *Windows*) and no additional setup.

We have decided to use the *serial port*. It is slower than other options but we do not expect to transfer a lot of data. It has the advantage of working everywhere and almost out of the box.

4.3 Messaging Protocol

Serial port provides a full-duplex, synchronous or asynchronous communication between two devices. It allows us to transfer one byte at a time through an unreliable channel. A byte sent through the channel may be received corrupted, or not at all, but *serial port* offers no way to check it. In addition to the lines used for asynchronous byte transfers, *serial port* also contains several other lines used for flow control and synchronization (the lines can also can be used for other purposes). We do not use the additional lines, using them might have timing issues (too high latencies) when a virtual *serial port* is used (using a *USB-serial* adapter). Hence we use asynchronous communication over the *serial port*.

If we send multiple bytes over the *serial port*, the recipient gets the data sent one byte at a time in the same order they were sent. The recipient thus sees it as a byte stream, where bytes may be missing or corrupted. We need to be able to find a message in the stream, if there is one, receive the message and check its correctness. We also want to be able to check that the recipient received the message. We were not able to find a suitable library that would solve this problem for us, and so we designed our own protocol. Its description follows. Message format is depicted in Table 4.1.

Table 4.1: Depiction of the message format. Upper row of the tables contains order of a byte in the message, lower row describes content of the corresponding byte.

0	1	2	3
0xA5	checksum	data length	
4	5	6	(6 + data lenght)
message type	message number	data	

The first byte of the message always contains the same value A5 hexa, it is used for detection of start of the message. To make the detection of the start of the message reliable, the first byte is the only byte of the message with the most significant bit set to one. All following bytes have the most significant bit set to zero. Hence, it is only allowed to use lower seven bits of them. This feature does not constraint us, the higher level protocol we use sends seven bit *ASCII*

characters as data. We could as well solve this using for example byte stuffing and allow usage all 8 bits of each byte, but we do not need the 8th bit and it would make the protocol more complex.

The second byte contains a checksum of the message, the checksum is used to check correctness of the message. The checksum is computed as sum of all the bytes of the message, excluding the first detection byte and the checksum byte, modulo 128 to fit it into the seven bytes. The checking method is simple and only guarantees detecting a single bit error, but we do not expect many damaged bytes and do not send any critical messages.

The third byte and fourth byte contain the length of the payload of the message. The number is in the big endian format. The first byte contains the higher seven bits and the second one contains the lower seven bits of the length. The maximum length of a payload of a message is 16384 bytes.

The sixth byte and seventh byte contain the message type and the message number respectively. From the eight byte follows payload of the message, if there is one. The message type and message number are used together. The message number is used to detect the correct response for a previously sent message. If a recipient of a message responds to it, the recipient has to use the same message number as the original message had, and has to use a correct response message type. Possible message types are enumerated below.

- *echo* (1) – used for messages that check reachability of the communication counterpart. The message may contain an arbitrary payload. Upon receiving this type of message the recipient responds with a message that has the same message number and payload and type *echo response*.
- *echo response* (2) – a message type used for a response to the *echo message*.
- *data* (3) – a message type used for standard communication, the recipient can respond by a message with type *response*.
- *response* (4) – a message type used to respond to a *data* message. The message may contain arbitrary payload and has to have the same message number as the message it is responding to.

It is up to the higher level whether the *response* messages are used. The *response* message may or may not contain data. When we get a correct *response* message, we are sure that our message was received correctly, otherwise we do not know. There is no way to make sure that a message was delivered exactly once, but the higher level protocol we use does not need this feature. Detecting that a message was not delivered is done using timeout, if we do not get a correct response after the specified timeout, we suppose the other side did not get the message.

Serial port has several parameters that are adjustable and both sides of communication have to know them in advance and have the same setup, otherwise the communication is not possible. These parameters are speed, number of bits per byte, parity, number of start bits and number of stop bits. We choose speed 115200 Bd/s, which is usually the maximum allowed speed for desktop computer *serial ports*. Eight bits per byte, no parity, one start bit and one stop bit.

4.4 Messages

In the Section 4.3 we have created a service that can send a message over *serial port*. In this subsection we create the protocol used to transfer plant's state to the visualizer on top of the service. We have only one requirement for the resulting protocol, we need to be able to update the state with sufficient frequency to display the plant's movement smoothly. The frequency should be around 20 Hz.

We design the protocol as a master-slave communication protocol. The visualizer is the master, and the plant is the slave. The visualizer sends commands and receives slave's responses. The plant does not send anything on its own, it only waits for commands and sends responses. There are several commands to discover the plant's state and to configure the plant. All the commands and responses fit into one message of the lower level messaging protocol, hence we do not need any message splitting before message dispatch and joining after message reception.

We have decided to use messages in a text format, in *ASCII* encoding. In comparison with binary format, text format requires more bytes to transfer the same amount of information. However, text format is human readable and with the usage of only single byte characters, we do not have to deal with endianess. The easy readability helps with testing and debugging. There are several usable existing formats with free libraries which we can make use of. Additionally, because of the usage of *ASCII* characters, which only uses seven bits from each byte, we are able to make the lower level messaging protocol simpler (see Section 4.3 for details).

There are several text formats which we can use, for example *XML*, *JSON* or *YAML*. We chose the *JSON* format because it is concise, easy to read and there are many free libraries for different programming languages that enable *JSON* generating and parsing.

We define a message as one *JSON* object. The plant sends command messages, to which the visualizer responds. Every command message contains a string property named *command* containing the name of the command. Each command then may or may not have other parameters specified as additional properties. Responses to all commands are also one *JSON* object, but otherwise do not have a common format. The list below describes allowed commands.

- *status* – has no other arguments. The response is the plant's status in the form depicted in Figure 4.1.
- *session* – has no other arguments, response is the current plant's session identifier. The session identifier changes when the plant is reset to enable visualizer to detect the reset. The response is in the form depicted in Figure 4.2.
- *initpos* – sets the initial position of the punch press's head. Has two additional parameters x and y which contain the position in nanometers. The command does not provoke a response, a verification of correct reception of the command is possible using the *session* command.

```
{
  "pos": { x: <x posision in nm>, y: <y position in nm> },
  "vel":
  {
    x: <x velocity in micrometers per second>,
    y: <y velocity in micrometers per second>
  },
  "punch":
  {
    x: <x position of the last punch in nm>,
    y: <y position of the last punch in nm>
  },
  "num": <number of the last punch>,
  "fail": <whether the simulated plant is damaged>
}
```

Figure 4.1: Format of the return value of the *status* command response.

```
{
  "id": "<string representing the current session>",
  "rnd": <whether the initial head position is random>,
  "ip":
  {
    x: <x axis initial head position>,
    y: <y axis initial head position>
  }
}
```

Figure 4.2: Format of the return value of the *session* command response.

- *random_initpos* – sets the initial position of the punch press's head to be randomly generated every time. The command does not provoke a response, a verification of correct reception of the command is possible using the *session* command.

- *power* – sets the current motor power of the plant. Has two additional parameters x and y which contain the motor power for each axis. The command has no response.

- *velocity* – sets the current velocity of the head of the plant. Has two additional parameters x and y which contain the velocity, in micrometers per second, for each axis. The command has no response.

- *position* – sets the current position of the punch press head. Has two additional parameters x and y which contain the position in nanometers. The command has no response.

We expect the visualizer to frequently issue the *status* command to update the main status information. The frequency shall be around 20 Hz. The 20 Hz is enough to detect all the punches the plant does, because each punch lasts 100 ms. It is also enough to clearly see the movement of the head. The *session* command shall be issued less often to detect that the plant was reset. When reset of the plant is detected, the visualizer's state shall be reset as well. The *initpos* command serves to help controller's programmers with controller debugging by allowing them to set the initial plant's head position. Commands *power*, *velocity* and *position* are only useful for debugging of the plant and visualizer together.

To verify the protocol enables visualizer state update with frequency 20 Hz, we compute the maximum update frequency possible with the protocol. The setup of the serial line used (115200Bd/s, 1 start bit, 1 stop bit, no parity) gives us a raw data speed of 11520 bytes per second. For each update we have to send the update command and the state as a response. The *status* command is 18 bytes long and the response can always fit into 154 bytes. The lower level messaging protocol has overhead of six bytes for each message. To execute one update, we have to serially transfer at most 184 bytes. One update will thus last less than 0.016 s. Using this protocol we are thus able to execute the update up to 60 times per second, which is enough. In this computation we neglected the time necessary to create and process messages, but they are expected to be very low.

5 Hardware Connection

This chapter describes the hardware connections of the plant and the visualizer, and the plant and the development computer, which together form the created platform.

5.1 Usage

When powering the platform up, the order in which the individual parts are powered up is important. The plant's board (*Beaglebone Black* board) has to be connected to the power first. After the plant is powered, the controller (*stm32f4-discovery* board) may be powered as well. This is because the plant board's processor manual [2] specifies that no power may be connected to the processor's pins unless the processor is not powered up (although it was experienced that doing so does not necessarily damage the plant). To remedy this we could add transistor before each pin of the plant to shielded it when the plant board is not powered, but we did not find it necessary. The time at which the development computer is powered up or connected to the plant is not important.

5.2 Schema

Figure 5.1 and Figure 5.2 contains connection schemes of different parts of the created platform. The pin labels on the controller and the plant in the schemes consist of two parts, that are divided by a dash. The first part contains the pin according to manual of the respective board ([3] for the plant, [6] for the controller). The second part describes how the pin is used.

Figure 5.1 contains the connection diagram of the plant board and the development machine, where the visualizer runs. On the right side of the diagram, there is the plant as a box, on the other side is the *DB9* connector usually used on desktop computers for connection to a *serial port*. In between is *MAX3232* integrated circuit, it is a fast voltage level converter. On the plant's side the *TTL* voltage levels are used (concretely 0 – 0.8 V for zero, 2.0 – 3.3 V for one) but desktop computers use the *RS-232* voltage levels on *serial ports* (-15 – -3 V for zero, 3 – 15 V for one), *MAX3232* converts between them. For its function *MAX3232* needs capacitors *C1* through *C5*, they all have value 0.1 uF. Power for the *MAX3232* converter is taken from the *Vcc* pin on the plant.

The *DB9* connector has to be plugged into the development machine to a port accessible to the user who will run the visualizer program. On the plant the *UART4* peripheral is used for serial connection, but any other *UART* peripheral could be used as well.

Figure 5.2 contains the connection diagram of the plant and the controller. The plant is represented by a box on the left side, controller by a box on the right

Figure 5.1: Connection schema of the plant and the development machine

side. The plant and the controller are connected through six lines. Using the *reset line*, the plant detects resets of the controller. It is connected to the *NRST* pin of the controller, which is low when the controller is held in reset and high otherwise. To hold the *reset line* low when the controller is not present or is not powered, the line has a pull-down resistor connected to it. The plant detects the start of the controller's program by detecting rising edge on the line. On the *IRQ line*, the plant creates an edge (changes logical value) when an observable change in its state happens. An observable change is a change visible through the controller interface. The edge generation on the *IRQ line* can be enabled or disabled through the plant-controller interface. The *IRQ line* is the only line that is in output mode, other lines are either in input mode (*reset detect*, *MOSI*, *CLK*, *slave select*) or in open drain mode (*MISO*). The *IRQ line* is thus the only one on which the controller is able to create a higher current flow than is allowed. A too high current flow could damage the plant physically. To remedy this we added a shielding transistor to the line, with the transistor in place the current from the *IRQ line* is not affected by the controller (a software driven *GPIO* pin of the *Beaglebone Black* cannot be set to open drain mode).

The remaining lines are used for *SPI* connection. The resistors *R1*, *R2* and *R3* have values 100 kΩ, 10 kΩ and 100 kΩ respectively. The pull-up resistors *R4* through *R7* are software setup on-the-chip resistors of the plant's processor. Power for the shielding transistor is taken from the *Vcc* pin of the plant.

On the controller side, the controller can use other pins to connect the *IRQ line*, and other peripheral modules for *SPI* communication. Only the *NRST* pin has to stay as it is. On the plant other *GPIO* lines may be used, or the other *SPI* peripheral.

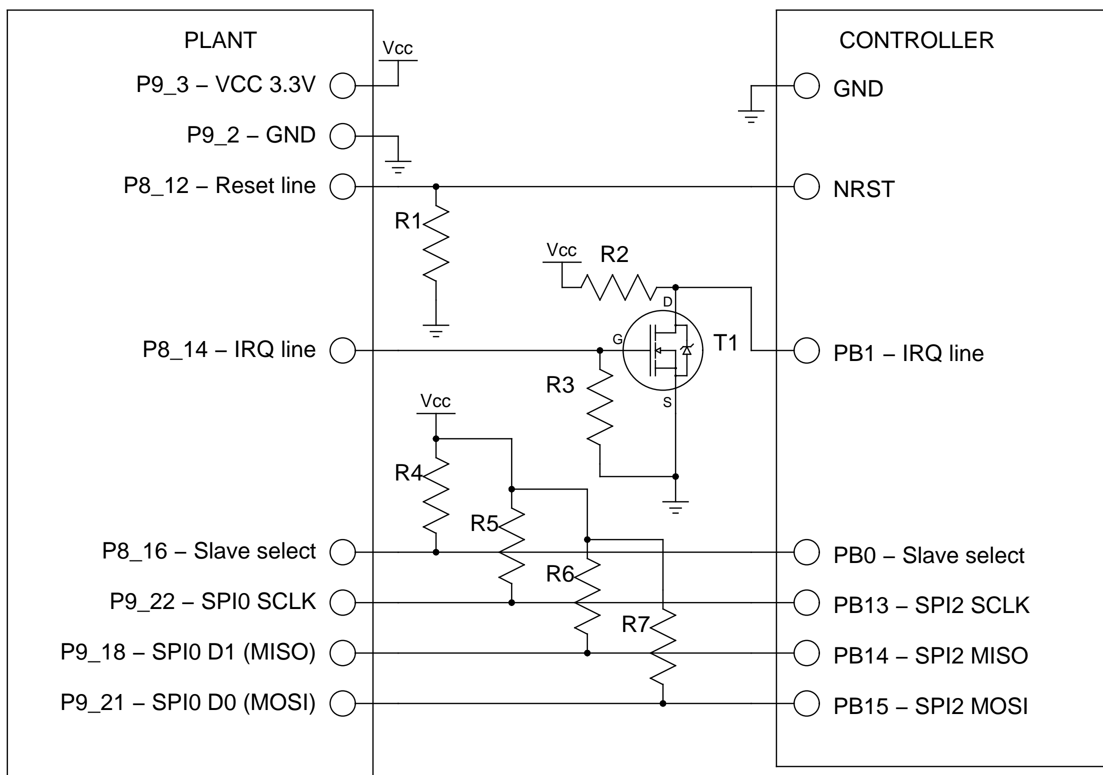


Figure 5.2: Conncetion schema of the plant and the visualizer

6 Plant

In this chapter we describe the implementation of the plant for the platform. First, we analyze the implementation options and describe the overall architecture, then we describe the individual components of the architecture. For each of the components we have its own section.

The plant is based on the *Beaglebone Black* minicomputer (board selection is discussed in Section 2.7.1). The *Beaglebone Black* board is based on the *Texas Instruments AM3359 Sitara* processor. The processor has a single main *ARM Cortex-A8* core of *ARMv7-A* architectute, additionaly, the processor contains two other processor cores, which together form a single peripheral called the *Programmable Real-Time Unit*. The processor features many other peripherals. Many pins of the processor are exposed on the pin headers of the board and so are easily accessible, making the board suitable for our kind of applications. For more information about the board design see the *Beaglebone Black* manual [3]. For more information about the processor see [2], the *ARMv7-A* architecture is described in [4].

6.1 Technical Analysis

First we need to select the software platform that will be used for the development. Further analysis and development depend on the selected platform. The options we have considered are in a list below.

- *Create own platform* – creating a platform anew would be possible since the documentation of the hardware we use is almost entirely accessible, but it would require a lot of additional work in comparison to other options.
- *StarterWare* – a hardware abstraction layer, supports the processor, not the *Beaglebone Black* board directly.
- *SYS/BIOS* – a real-time operating system, supports the processor, not the *Beaglebone Black* board directly.
- *Linux* – an operating system, contains drivers for almost all *Beaglebone Black* hardware. *Linux* images directly for *Beaglebone Black* are accessible and updated. It is supported directly by the authors of *Beaglebone Black*.
- *FreeBSD* – an operating system, it does not support as much of the *Beaglebone Black* hardware as *Linux*. It has several known issues running on *Beaglebone Black*.

From these options we selected *Linux*, concretely *Debian Wheezy* distribution with *Linux* kernel version 3.8. This setup is supported by the *BeagleBone Black* developers. The kernel version used is a version specially patched to support all

of the *BeagleBone Black* hardware. From now on we will also refer to the *Linux kernel* as the kernel.

To be able to communicate with the controller we need to be able to communicate over *SPI* and read and write values to the processor's pins. *GPIO* functionality of the processor is fully supported and can be utilized through the kernel's *GPIO* subsytem. The processor contains two *SPI* modules with support for slave mode and many general purpose input/output pins, that are available. Unfortunately the kernel's *SPI* spubsystem does not support *SPI* in slave mode (nor does *SPI* driver for the processor).

To communicate with the visualizer a *serial port* is necessary, the processor contains several *UART* peripherals that enable asynchronous serial communication, which is what we need. The *Linux* kernel supports the *UART* peripherals and they are accessible from user-space as *serial ports* exploitable using standard *Unix* API (they are also accessible from the kernel).

Additionally to the main core, the processor contains two much simpler processors. These processors have their own memory (separate for program and data), but also have access to all other modules on the chip and to the main memory. We will use them to handle *SPI* communication with the controller.

The plant will be simulated using a continuous simulation, that will be periodically updated. We would like the update of the simulation to be executed as often as possible and with low jitter. To do so, we use a precise timer and run the update of the simulation in the timer interrupt handler. The processor contains several high precision timers, which the kernel does not utilize and are thus available for our needs. The timers are available through special kernel driver.

We want to use the most of the processor's execution time for the plant's simulator and have low latency and jitter. Hence we decided to run the update directly in a timer interrupt handler in the kernel. To utilize the processor more, we would like for the update to preempt any other running task. We have considered several options to reach the goal, the options follow below.

- *Stock Linux kernel* – stock *Linux* kernel runs interrupt handlers with interrupt requests disabled, an interrupt handler is thus not preemptible. We cannot break this feature as handlers may rely on it.
- *Fast Interrupt Request* – security versions of the processor contain feature the *Fast Interrupt Request* feature (*FIQ*). The *FIQ* enables to set an interrupt line to be *FIQ*, then the interrupt is processed in the *FIQ* handler, instead of the normal interrupt handler. That would be easily exploitable, because the *Linux* kernel does not use the *FIQ* handler and the handler would not be preemptible by normal interrupt requests. Unfortunately the processor of the *Beaglebone Black* is only a general purpose version, which does not feature *FIQ*.
- *Prioritization of interrupts* – another feature usable for this purpose is interrupt prioritization. This feature enables to set a priority of each interrupt request. An interrupt request with a higher priority is able to preempt han-

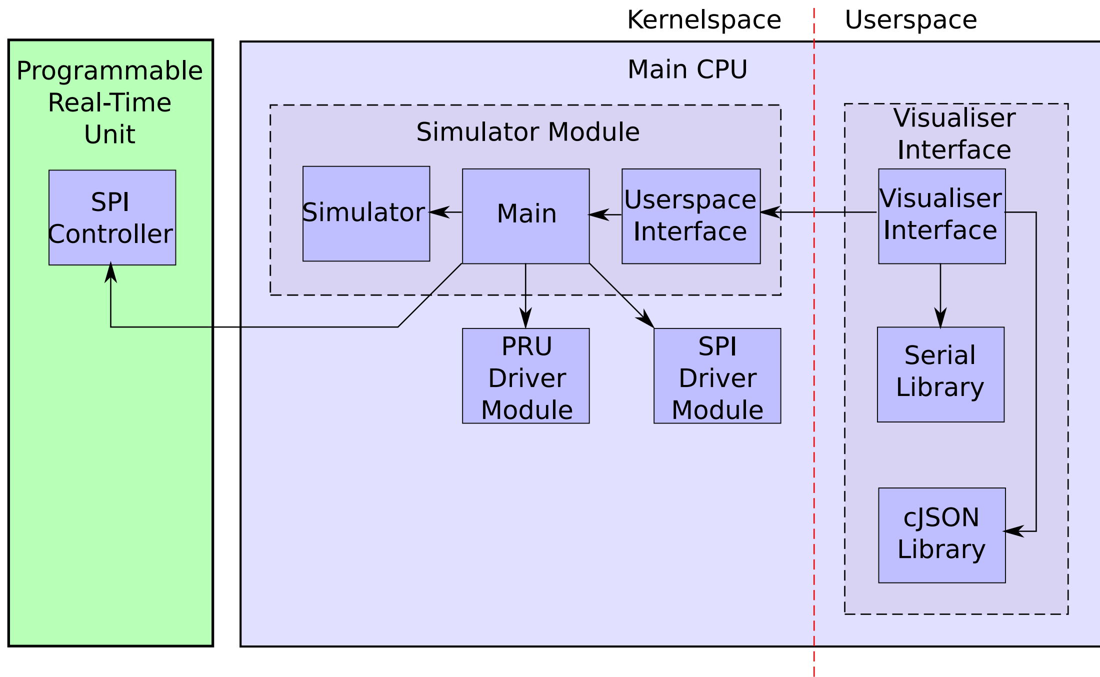


Figure 6.1: Architecture of the plant – depicts individual components of the architecture, hierarchy of the components, and on which processor and in which mode the components run. An arrow between two components means that one component directly uses another one (at which the arrow points).

dling of an interrupt with lower priority. However *Linux* does not support interrupt prioritization, we would have to implement it ourselves.

- *PREEMPT_RT* – *Linux* kernel patch. It enables task prioritization in *Linux* and contains other real-time extensions. Some of the processor's peripherals do not work with this patch applied.

- *Xenomai* – framework cooperating with the *Linux* kernel, brings real-time tasks.

We have decided to use the *Prioritization of interrupts* option. The selected option will require some kernel alterations but it will give us ultimate control over the implementation and we will be able to keep every other functionality of the kernel working.

6.2 Architecture

The plant's software architecture is depicted in Figure 6.1. The figure shows individual components as boxes, an arrow between two boxes means one component uses another one.

The plant uses the main processor core and also one of the cores of the *Programmable Real-Time Unit* (*PRU*). The *PRU* core hosts the *SPI Controller* component that realizes communication with the controller. The main core hosts the

main part of the plant. The main part of the plant is separated into two parts, kernel-space part and user-space part. These two parts communicate with each other through a device file.

The kernel-space part realizes the simulator and a part of the communication interface for the controller. It consists of three kernel modules (the *Simulator* module, the *PRU Driver* module and the *SPI Driver* module). The *Simulator* module is the main component of the kernel-space part. The module runs the simulator, provides interface to the user-space program and communicates with the *SPI Controller* component running on the *PRU* core. The *Simulator* module uses the other modules mainly for hardware initialization.

The *SPI Driver* module realizes a driver of the processor's *SPI* peripherals. The main module uses it to get a handle of the *SPI* peripheral, enable it and configure it. The *Simulator* module and the *SPI Driver* module do not further use the *SPI* peripheral themselves, it is further used only by the *SPI Controller* component running on the *PRU* core. The *PRU Driver* module realizes a driver of the processor's *PRU* peripheral. The driver is an open-source program, which we reused. The *Simulator* module uses the *PRU Driver* module to get a handle of the *PRU* peripheral, initialize the peripheral and load the program of the *SPI Controller* component to it.

The user-space part consists of one program. The program realizes communication with the visualizer through an interface defined in Chapter 4. The program consists of three components (the *Visualizer Interface*, the *Serial Messaging* library and the *cJSON* library). The *Visualizer Interface* component is the main part realizing the interface. The main part then uses the *Serial Messaging* library and the *cJSON* library. The *Serial Messaging* library enables communication over *serial port* using messages as defined in Section 4.3. The *cJSON* library is an open-source library, which we reused. The *cJSON* library enables creation and parsing of data in *JSON* format.

The *SPI* Controller component realizes *SPI* communication with the controller as defined by Section 3.3. The component uses the *SPI* peripheral directly (not through the *SPI Driver* module), it does not initialize or configure the peripheral. The component communicates with the *Simulator* module through shared memory. The program of the component is written in the *PRU* assembly language.

As a main programming language we use the *C*. *C* is suitable for low-level programming. It is also the main language of the *Linux* kernel. Small parts of the code are written in an assembly language.

6.3 Simulator Module

The main software part of the plant is the Simulator kernel module, it controls operation of kernel part of the plant. Parts of the module are described in the following subsections.

6.3.1 Simulator

The simulator component embodies the simulator of the punch press machine (the machine is described in Appendix B). Through its API it enables reading the state of the simulator, setting its inputs and updating the simulation.The simulation is continuous and runs in real-time. The update method has the time elapsed from previous update as its parameter, upon invocation it computes the new state of the simulation for the current time in response to the previous state of the simulation, the inputs and the time elapsed from previous update invocation. Because of the necessity of running in real-time, the processor time we have to update the simulation in is limited. The simulation thus has to be sufficiently simple.

Instead of writing the simulator anew we reused an existing one with permission of its author. The existing simulator is part of a *QEMU* device driver, used in the course *Embedded and Real-Time Systems* at *Faculty of Mathematics and Physics at Charles University in Prague*. It fulfills the specification on which we based the plant-controller interface and is written in *C*, it is thus easily reusable for us.

Event though the reused simulator fulfills the specification and is written in *C*, it was not possible to use it as is. To allow running the simulator inside the *Linux* kernel it was separated from the driver. Also, it was rewritten to only use fixed point integer arithmetics instead of floating point, on which the computation was previously based.

The *Linux* kernel does not support direct usage of floating point coprocessor in kernel code. Floating point computations are not necessary in the Linux kernel code and it would require save and restoration of the processor's floating point context on a kernel entry and exit respectively, among other reasons. We have several options to remedy this. To utilize floating point instructions we could save the state of floating point coprocessors we use before our computations, and then restore the state afterwards. The version of the *Linux* kernel used (3.8) does not, however, contain support for this on a processor with *ARMv7-A* architecture. The *Linux* kernel has support for this since version 3.12 in the *kernel_mode_neon* feature. Additionally, we would have to make sure that our code cannot raise any floating point exceptions as they are expected to only arise in user-space code. Another option is to exploit the compiler and set it to emulate the floating point operations using only integer instructions, however it is much slower than usage of floating point instructions. Last option, and the one we chose, is not to use floating point code at all and replace it with a fixed point code. This requires us to rewrite the existing floating point code, but the result will be faster than previous solutions and easily portable to other platforms.

The internal working of the simulator puts constraints on the frequency of updates of the simulation. In order to explain the constraints and inner workings of the simulator, we describe the main part of the update computation, which is update of movement of the plant's head, here.

Inputs of the plant (that the user of the simulator sets) are power to motors and requests to punch, outputs (that the user of the simulator can only read) are quadratic encoders of motors and safe zone statuses and state of the punching

$$friction = 200$$

$$velocity_decrease = \frac{(friction \cdot update_time)}{1000}$$

$$motor_force = (power \cdot 2500 - velocity) \cdot 5$$

$$random_force = \frac{(motor_force \cdot random_integer)}{1000000}$$

$$total_force = motor_force + random_force$$

$$velocity = max(velocity + \frac{(total_force \cdot update_time)}{1000000} - velocity_decrease, 0)$$

$$position = position + \frac{velocity \cdot update_time}{1000}$$

Figure 6.2: Computation of an update of movement for one axis of the simulated machine

head, inner state (not visible by the user of the simulator) is the position and the velocity of the head. The update function first updates velocities and positions in both axes. Error statuses and quadratic encoder statuses are then set according to the new position. State of the punching head is updated last, it comprises of starting or ending a punch and setting errors, if punch is requested during another punch or punching happens when the head is not slow enough. Simplified computation of new velocity and head position in one axis is shown in Figure 6.2, *update_time* is the time elapsed from the previous update, *random_integer* is a random integer value in range [0, 1000), other values come from the state of the simulation. To make the equations simpler, they only take into account nonnegative values of velocity and power.

The position updating puts a constraint on the duration elapsed between two updates which we pass to the update function. The duration must be small enough so that the update does not move the head more than a quarter of one millimeter in one axis. A quarter of a millimeter is the distance after which quadratic encoders change and we have to inform the controller about encoder change. To compute the maximal *update_time* we can use, and therefore the time elapsed between two updates, we start with the aforementioned equations. From them we compute the maximal speed the punch press can reach, it is 280.04 mm/s. For further computations we round the value up to 300 mm/s. The maximal frequency of change of a quadratic encoder thus is 1200 Hz. To be sure that we do not move the head more than a quarter of a millimeter (and thus do not skip a quadratic encoder value) during one update, we have to update the simulation at most each 833 microseconds. We verified by experiment that this update time is feasible, with further experimenting update duration of 83 microseconds was reached. Other aspects of the simulator do not put constrains on update time.

Communication with User-space

To communicate with the user-space part of the plant, *JSON* messages are used. To process incoming messages and generate outgoing messages requires access to the inner status of the simulator, processing and generation of these messages it is thus part of the simulator. Part of the message parsing is also a simple *JSON* parser, because the kernel does not contain any support for *JSON* format and porting a whole *JSON* processing library into it would be excessive. *JSON* formatting is done simply using *printf*.

Testing

The simulator will run in kernel mode and is update in an interrupt handler. Programs running in kernel are not as easy to debug as user-space programs. User-space debuggers cannot be used, and debuggers suitable for kernel debugging require additional hardware and non-trivial setup. Without additional kernel setup, printing during interrupt handling is disabled. User interaction (passing input, when the program is running) with kernel programs is also non-trivial.

We need to be able to test the simulator extensively, and we would like to do it more easily. Hence we decided to test it in user-space, instead of kernel-space. For this purpose we created a separate application called the tester. It was possible to run the simulator in user-space, because it does not have any dependencies on the kernel code. The tester is an interactive application that allows us to update the simulation (once or multiple times), set inputs of the simulation and print its state by simple commands. Also we are able to use all the classic debugging methods.

6.3.2 Main

The Main part of the Simulator module takes care of update of the simulator, initializes used peripherals, resets the simulator on reset of the controller and informs the controller about changes in the plant's state.

Two *GPIO* lines are used through the *GPIO* subsystem. On the first line the reset of the controller board is detected, the simulator is reset upon the controller reset detection. This is so that the controller always starts its operation on a "clean" plant. On the second line a level is changed each time a change in the plant's state, which is visible through the controller's interface, happens. The *SPI Driver* module is used to initialize the *SPI* peripheral. The *PRU Driver* module is used to initialize the *PRU* peripheral, load the program of the *SPI Controller* component into it and communicate with the *SPI Controller* (read inputs for simulator and write simulator's output).

Update of the simulator is executed regularly in a timer interrupt handler. One of the processor's timers is used to issue the interrupt regularly. The timer is not used through any generic *Linux* timer interface, it is used through a *dmtimer* driver, which is specific to one processor class and enables direct usage of the

timer. The timer interrupt has a higher priority than any other interrupt, to be as regular and as frequent as possible. In the timer interrupt handler, first the plant's actuators are updated from the *PRU* memory, then the simulator's update is invoked, and finally the plant's sensors are set in the *PRU* memory and an interrupt is issued to the controller through corresponding *GPIO* line if need be.

6.3.3 User-space Interface

To transfer current plant's state to the visualizer a user-space application is used (called *Visualizer Interface*). Hence we need an interface which will allow the *Visualizer Interface* application to read the simulator's state. We use a device file for this purpose, it is a character device file. The user-space application has to open the file and then it can read the plant's state or write a command. Every single write to the file is taken as a command. The commands have a form of *JSON* objects, each received command is forwarded to the simulator, which executes it. If the command is successfully accepted and executed, the number of bytes that the user has written is returned, otherwise an error code is returned. Every single read from the device file reads the state of the plant. If the buffer passed, when reading the state, is not big enough, only a part of the state is read, the next read again starts reading of the state from start (instead of continuing where the previous read left it off). The state is read in a *JSON* format as generated by the simulator. Return value is the number of bytes read.

The file to be used by the user-space application is not created by the kernel module itself, it has to be created from user-space by the *mknod* command.

6.4 High Priority Interrupts

The processor of the *BeagleBone Black* supports prioritization of interrupts, which we want to utilize. However the *Linux* kernel does not support it, we thus have to alter the kernel. The interrupt prioritization is enabled by the interrupt controller, the controller allows setting a priority for each interrupt and a global interrupt threshold. When an interrupt request is signaled by a peripheral the interrupt controller propagates it to the processor only if it has a higher priority than is the priority of the current interrupt threshold. The *Linux* kernel only allows two interrupt states, interrupts enabled and interrupts disabled (and masking individual interrupts). Interrupt disabling is done through the *Program Status Register*. By letting the interrupts enabled in the *Program Status Register* and disabling them using the interrupt threshold, we can preserve the behavior of interrupt enabling/disabling for all the standard kernel interrupts and allow interrupts with higher priority to preempt the standard interrupt handlers. We use a higher priority for special interrupts, to which we want to react faster. Interaction with the kernel inside the higher priority interrupts has to be very limited, as the kernel is not prepared for that.

To accomplish the outlined goal we have fulfilled several subgoals. The first of

them is alteration of the interrupt controller driver, which allows to set interrupt priorities, and alteration of the interrupt handler, which allows interrupts with higher priorities to preempt interrupt handlers with lower priorities. The second subgoal is alteration of the functions that the kernel uses to enable/disable interrupts. Instead of disabling all interrupts, these functions will set the interrupt threshold to disable all standard *Linux* interrupts. Enabling of interrupts will still allow all (non-masked) interrupts, but again will do it by setting the interrupt threshold. The third subgoal is saving the current thread's interrupt threshold when the thread taken off the processor (by interrupt, exception or voluntarily) and restoration of the threshold when the thread is run again. The interrupt threshold is now part of the thread's context.

Fourth step is taking care of special cases – early boot and the *cpu_idle* process. On early boot, the interrupt controller driver is not initialized and we thus cannot set interrupt priorities, instead we have to enable and disable interrupts using the *Program Status Register*. The *cpu_idle* process runs when no other process wants to run and it may call the *wfi* instruction. It calls the *wfi* instruction with interrupts disabled, but interrupts have to be disabled using the *Program Status Register*, not by priority threshold. If interrupts were disabled using the priority threshold and the *wfi* instruction called, the processor would never wake up.

Additional problem that may arise when using higher priority interrupts is stack overflow. The kernel uses the kernel stack of the current thread when handling an interrupt. We kept the behavior the same and so when an interrupt handler is interrupted by a higher priority interrupt, the new handler uses the same stack. The kernel stack of a thread has a fixed size and on the *ARM* architecture it cannot be easily set to any other size. Users of the higher priority interrupts should take this into account and use as little of the stack as possible.

6.4.1 Spurious Interrupt Requests

When a processor interrupt happens, it might be so called *Spurious* interrupt (see the processor's manual for detailed information). The *Spurious* interrupt means that the interrupt controller register, which should contain the number of the interrupt when an interrupt handler is invoked, does not contain a correct value. The fact that the current interrupt is *Spurious* is signaled in one of the registers. The *Spurious* interrupt may happen for several reasons. This means that we cannot service the current interrupt with the highest priority as we do not know which one it is.

In order to remedy the situation, the interrupt controller must be acknowledged the same way as if a normal interrupt has happened, then the interrupt controller recomputes the interrupt that currently has the highest priority and sets the correct values into registers. The *Linux* kernel deals with spurious interrupt by restarting the interrupt handler. This may significantly prolong the latency of handling a *Spurious* interrupt. We remedy this behavior by acknowledging the interrupt controller and waiting in the interrupt handler, until it signals a correct interrupt.

6.4.2 Testing

To verify that interrupts with higher priority are allowed to interrupt lower priority handlers, we created a testing module. The module utilizes two timers and their interrupt handlers. The first timer signals an interrupt with lower frequency, the interrupt handler has a lower priority and it takes a while to complete. The second timer signals interrupt with a sufficiently higher frequency and the interrupt has a higher priority. We measure how many times the higher priority interrupt handler interrupts the lower priority one. Using this method, it was verified that the higher priority interrupt can be interrupted by the lower priority one (but not vice versa).

Additionally, the testing module measures the maximal latency of the higher priority interrupt handler (how long it takes to start the interrupt handler when the interrupt is signaled). This is done using the same timer which generates the higher priority interrupts. When the timer overflows, it generates an interrupt request and is loaded with a predefined value, we read the timer value when the interrupt handler starts and convert it to latency. Using this method, the maximal latency we measured was 11 microseconds.

When we used the same method to measure latency of a timer interrupt handler without usage of higher priority interrupts, the maximal latency we measured was more than 2000 microseconds. In this case we used one timer.

6.5 SPI Driver Module

SPI is the main means for communication with the controller, according to the plant specification from Chapter 3. The plant is to be slave to the controller. The processor has an *SPI* peripherals allowing *SPI* communication in both master and slave mode, which we want to utilize. The *Linux* kernel, however, does not support slave mode for *SPI* and so even though it contains an *SPI* driver for the processor, the driver does not support slave mode. Therefore we created a new *SPI* driver with support for slave mode (and slave mode only).

The driver is realized as a separate kernel module loadable and removable at runtime. The interface of the driver allows enabling and disabling of the *SPI* peripheral, configuring the *SPI* peripheral, read of the receiving register and its status, write to the transmitter register and read of its status and management of interrupts of the peripheral (enable/disable interrupt, get/clear interrupt status, register interrupt handler). The driver does not support master mode, DMA or fast data transfers using the FIFO feature of the SPI peripheral. The driver's interface is only reachable from kernel mode by other kernel modules, no user-space interface is provided. The driver does not do any locking, it expects each device to be used by one user only, and it expects the user to assure mutual exclusion where necessary.

The driver utilizes several kernel subsystems. To power the peripheral up, the driver uses the runtime power management subsystem and the clock management subsystem. As the driver is a device driver, it uses the platform driver

infrastructure to identify devices which it drives and get information about the device (interrupt number, address range). To setup the pins of the *SPI* peripheral correctly, the driver uses the pin-control subsystem.

6.6 SPI Controller

Using the *SPI Driver* module, we could implement the *SPI* part of the plant's interface. This kind of implementation would, however, take processor time away from the simulator update. The *SPI* interrupt handler would have a higher priority, than the simulation update interrupt, in order to have short waiting times between two words sent over *SPI*. Moreover, we cannot control how much the controller uses the *SPI* and it is capable of using it very often and that could result in the *SPI* interrupt handler taking too much of processor time and not enough processor time for the simulator. Because of this we have decided to utilize a processor from the PRU peripheral to handle the *SPI* communication.

The *PRU* peripheral contains two processors, its own data and program memory, an interrupt controller and has access to all the peripherals of the processor, including the *SPI* module. We still use the *SPI* driver module to get the *SPI* peripheral as its sole user, power it up (we have to use *Linux* kernel API for that and cannot do it from the *PRU* peripheral), configure and enable the *SPI* peripheral. We thus do not need to reimplement such code on the *PRU*.

The program of the *SPI Controller* handles the communication itself. It checks the slave select line to reset the communication on its falling edge. If the slave select line is down, it communicates with the controller through protocol from Chapter 3.

The program of the *SPI Controller* has to communicate with the *Simulator* kernel module, which executes the simulator update. The communication is done using two words in the *PRU*'s memory, which is whole accessible from the main processor core as well. The first word contains the plant's sensors, it is written by the *Simulator* module after the simulator's update to contain the latest values. The *SPI Controller* program reads it, when it has to send sensor values to the controller. The second word contains plant's actuators. The *SPI Controller* program writes it according to received controller's messages. It is read by the *Simulator* module before the simulator update takes place.

The program of the *SPI Controller* is written in a *PRU* assembly language. Part of the attached software is the *pasm* assembler, which can be used to compile the code. The *pasm* assembler is an open source program, which we reused.

6.7 PRU Driver Module

To power up the *PRU* peripheral, initialize it, load a program into it and run the program we need a driver. The *Linux* kernel in version 3.8 contains a *PRU* peripheral driver, but the driver only has a user-space interface and does not

support communication with the module.

Instead of the driver in the 3.8 *Linux* kernel, we reused a different open source driver. The driver enables using the module and communicating with it (using shared memory or interrupts) through kernel interface. To be able to use the driver in the 3.8 kernel we had to update it as it was written for an older kernel version. We removed some unused function attributes, made it understand device trees, made it use the power management subsystem to power up the module before it is used and other smaller changes. We compile the driver as a separate module.

6.8 Visualizer Interface

Communication with the visualizer is done using a separate program. The program was made as user-space application so it can take advantage of user-space libraries, easier debugging and testing, and because it does not require direct access to kernel features. However, the program requires access to the state of the simulator. It accesses state of the simulator through user-space interface of the *Simulator* module, which is a device file. The communication protocol used is briefly described in Section 6.3.3.

The application can run in two modes, the daemon mode and the interactive mode. The application is expected to normally run in the daemon mode, the interactive mode is useful for testing and debugging of the simulator, which runs in kernel. In the daemon mode, the application opens a given *serial port*, through which it communicates with the *Visualizer*. It reads commands from the *serial port*, executes them using the *Simulator* module interface and writes back replies. The communication protocol used is defined in Chapter 4. In the interactive mode, the application does not use a *serial port* at all. Instead, it reads user commands from the standard input and uses standard output for replies. The commands allow to print out the simulator's state and set inputs and some parameters of the simulator.

The application accepts tree command line options:

- *-i* – this option makes the application run in an interactive mode, otherwise it runs in a daemon mode.
- *-d <filename>* – this option allows specifying a device file different from the default one. The device file is used for communication with the Simulator module.
- *-s <filename>* – this option allows specifying a *serial port* different from the default one. The *serial port* is used for communication with the visualizer.

To parse and create *JSON* messages, used in communication with both the visualizer and the kernel module, the application uses a *cJSON* library. The *cJSON* library is an open source library, which we reused. For low level messaging over *serial port*, as defined in Section 4.3, a separate library (*Serial Messaging* library)

was created. The main functions the library provides is sending a message over *serial port*, receiving a message and waiting for a message to come.

7 Visualizer

This chapter describes the *Visualizer* application implementation, its architecture and usage. The application's purpose is to graphically display the current state of the plant in real-time and to help with debugging and verification of correctness of controller programs. Communication with the plant is performed using *serial port*, the protocol used is described in Chapter 4. The main platform for the visualizer is a personal computer with a *Linux* based operating system, which we expect to be the most often used development computer. The application, however, should be easily portable to other platforms as well (*Windows*, *Mac OS X*).

7.1 Architecture

As a programming we chose *Java*, specifically *Java 8* with *Standard Edition* environment. *Java* allows for quick development and easy portability as it is officially supported on all the most used operating systems (*Windows*, *Linux*, *Mac OS X*).

The architecture of the application is shown in Figure 7.1. The picture shows application's modules and libraries as boxes and arrows between the boxes show which module directly uses another module. The application is divided into three modules (*Visualizer*, *PlantState*, *SerialCom*). Names of the modules correspond to names of main classes in the modules, each module is contained in one *Java* package. The *SerialCom* module realizes the low level messaging protocol over a *serial port*, defined in Section 4.3. The *PlantState* module encapsulates information about the current state of the plant and allows to update the current known state (read it from the plant). It uses the *SerialCom* module for communication with the plant. The *Visualizer* module initializes the whole application and visualizes the plant's state represented by an instance of the *PlantState* class.

To make the development faster and the resulting application portable we reused several existing libraries, all of the used libraries are open source. The *JavaFx* library is used to create the graphical user interface in the *Visualizer* module. The *jSSC* library for communication over a *serial port* is used by the *SerialCom* module. A library for *JSON* format parsing and generating is used by the *PlantState* module, the library is implementation of the *JSR 353* specification – *Java API for JSON Processing*.

7.2 Usage

The application can be executed using the *run* command in its directory. It is necessary to make sure that the path to the *Java 8* virtual machine and the *classpath* (a path where the *Java Virtual Machine* looks for libraries) are set properly in it. The application has four command line options, their description

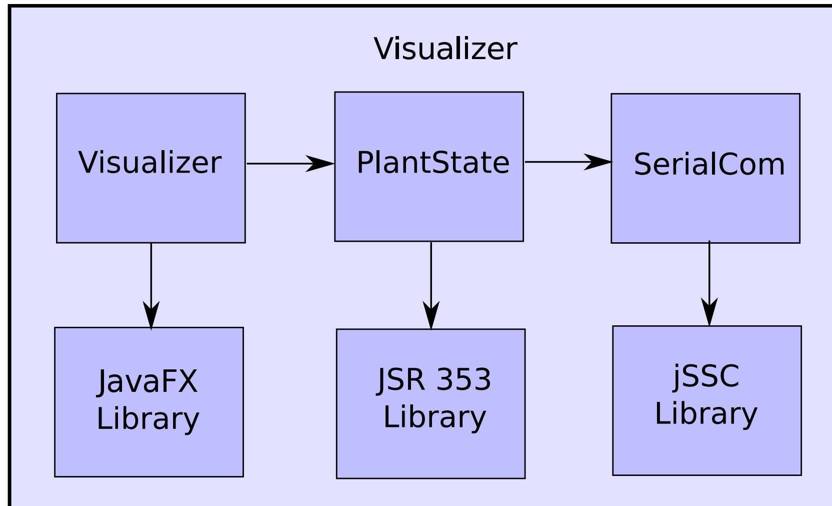


Figure 7.1: Architecture of the *Visualizer* – depicts individual components of the architecture and hierarchy of components. An arrow between two components means that one component directly uses another one (at which the arrow points).

follows below.

- *-i <input file>* – sets the path to the file containing positions of punches that the controller program is supposed to perform. The application then shows the expected punches graphically and checks whether they are punched correctly. The input file has to be an *ASCII* encoded text file with one punch defined on each line. One punch is defined by its x and y positions in this order, the positions have to be separated by a comma. The positions have to be floating point numbers specifying a number of millimeters from the top left corner of the punch press work area. If the *-i* option is not entered, no input file is used and no punch checks are performed.
- *-l <output file>* – sets path to the log file. The application then writes information about errors, performed punches to the log file and also periodically writes the head's position. The default log file is the console output.
- *-s <serial device>* – sets the serial device used for communication with the plant. The default value of the serial device is */dev/ttyUSB0*.
- *-h* – prints description of command line options of the application and available *serial ports*.

The application's user interface is depicted in Figure 7.2. The main part of the application's window in the top left shows the punch press's working area, the head, and the planned and performed punches. Below the main part, there is a slider, which sets zoom of the main part. On the bottom of the window, there is a status bar, which contains information about the current state of the plant and whether the visualizer is connected to the plant. On the right side is a panel with punches and a panel with initial position. The panel with punches shows positions of planned an performed punches. The panel with initial position displays setup of the initial position, whether it is random or has a fixed value.

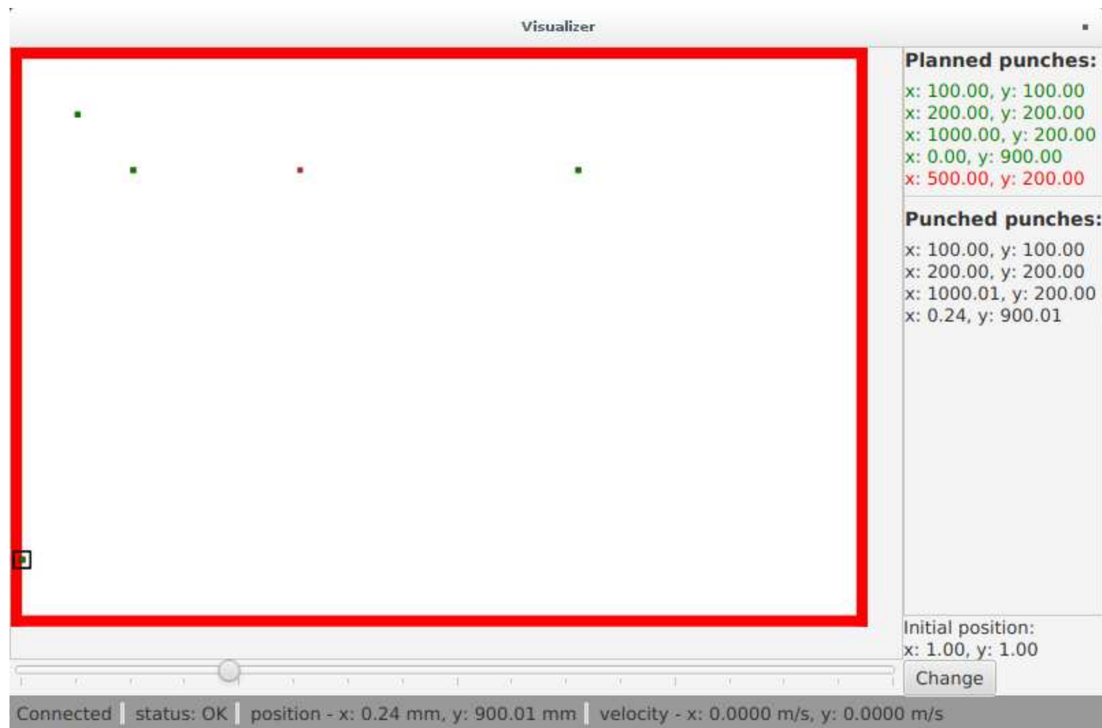


Figure 7.2: A screenshot of the *Visualizer* application

The panel also contains a button that shows window, in which it is possible to set the initial position.

The application has to run at most once for each *serial port*. If multiple instances of it use the same *serial port*, their communication with the plant becomes intermingled and the applications will not be able to run correctly.

8 Base Controller Project

This chapter describes the created base of a controller project its dependencies and usage. The project is supposed to help students start controller development by providing everything that is necessary to start developing, so that students do not have to look for suitable development tools themselves, which is time-consuming. It is tested on a Linux development machine, but should be easily portable to other platforms (*Windows*, *Mac OS X*) as all its dependencies have versions for different platforms. It comprises of an example program and libraries, makefile, an *Eclipse IDE* project and.

The project contains two libraries, the *CMSIS* library which is a standard library from *ARM* and a processor specific library from the *ST Microelectronics* company which provides a hardware abstraction layer with drivers for devices included on the processor of the controller board.

8.1 Dependencies

Before the project may be compiled and the resulting program written into the processor's memory several things have to be prepared.

The necessary dependencies of the project are the *GNU GCC* and *Binutils* for ARM, the *openocd* and the *GNU make*. Additionally, an *Eclipse IDE* is necessary to open the *Eclipse IDE* project.

Path to all the necessary programs have to be set correctly in the makefile file of the project. For each program that is used in the compilation directly, there is a variable at the start of the makefile.

8.2 Basic Usage

Further development of the base project is possible either from the *Eclipse IDE* or from a command line. The project can be imported to an *Eclipse IDE* workspace (by importing its folder) as is. The imported project can be compiled, or cleaned from *Eclipse IDE* in a standard way. One thing that cannot be easily done from the *Eclipse IDE* is addition of new files to the project. New source files have to be added directly to the makefile of the project. In the makefile, the list of files to be compiled is listed in a variable named *SRCS*. Figure 8.1 shows the *Eclipse IDE* with the project open in it. Additionally, programming of the controller's processor and debugging cannot be done from the *Eclipse IDE* alone, *make burn* and *make openocd* commands (described below) have to be used for this purpose.

From command line, working with the project is done through the program *make*. Whole setup of the project is in a makefile. The following commands can be used.

- *make proj* – compiles the project.

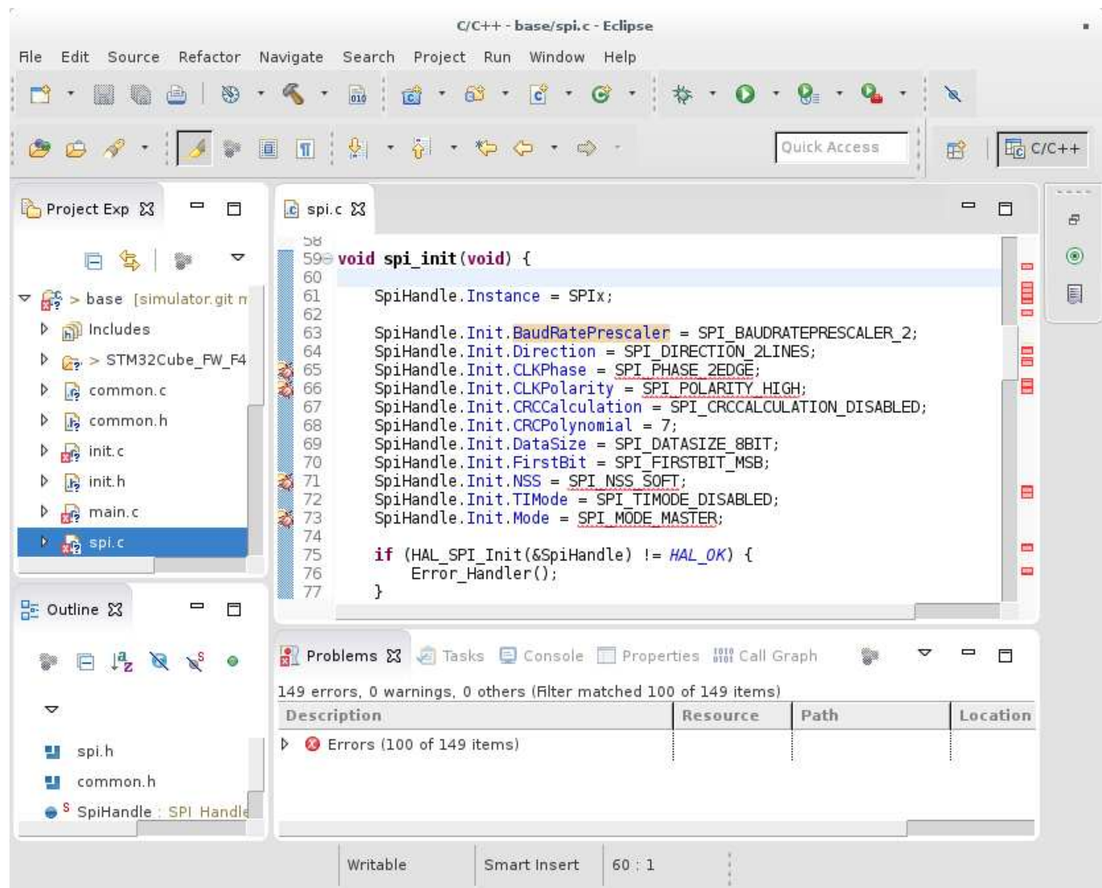


Figure 8.1: *Eclipse IDE* with the base project open

- *make burn* – writes the compiled program into the processor using *openocd*.
- *make openocd* – starts the *opneocd* in a debugging mode, then it is possible to connect the *gdb* interactive debugger to it or start debugging using the *Eclipse IDE*.
- *make clean* – removes all the files created by compilation.

9 Evaluation

To verify correctness of the created platform, we need to test it. We already tested most of the parts of it individually. Some parts, however, were not testable individually and we need to test the platform as a whole.

To test the whole solution we first connect the individual parts (plant, controller, visualizer) and power them up according to Chapter 5. A photography depicting the wired up hardware setup is in Figure 9.1. Then we run the corresponding programs (the plant's simulation and the visualizer) and create a program for the controller, that will drive the plant to punch a certain pattern. Finally, we verify that controller has driven the plant to create the expected pattern using the visualizer.

9.1 Controller Program

The created controller program is based on the base project described in Chapter 8. The punch press's head start is at a random position when the controller program starts and the controller has to reset the head to zero position first. This way we know the origin of the working area, and we can use quadratic encoders to the measure distance of the head from the origin in both axes. To accomplish this the program utilizes feedback control (for more information see [7]). To reach the zero position a simple on/off control is used. Moving the head to predefined punching positions is realized using proportional feedback control. In order to communicate with the plant, the program has to utilize the *SPI* peripheral of the processor, and react to level changes on the *IRQ line*. Time measurement is also required to detect, that the punch press's head has stopped and we can perform a punch.

Using the aforementioned algorithm the controller program can drive the plant to punch holes in a selected pattern. After the controller program was created and deployed to the controller, the selected pattern was punched correctly. We thus conclude that the platform works as expected. The result of the controller program driving the plant as seen by the visualizer is displayed in Figure 9.2.

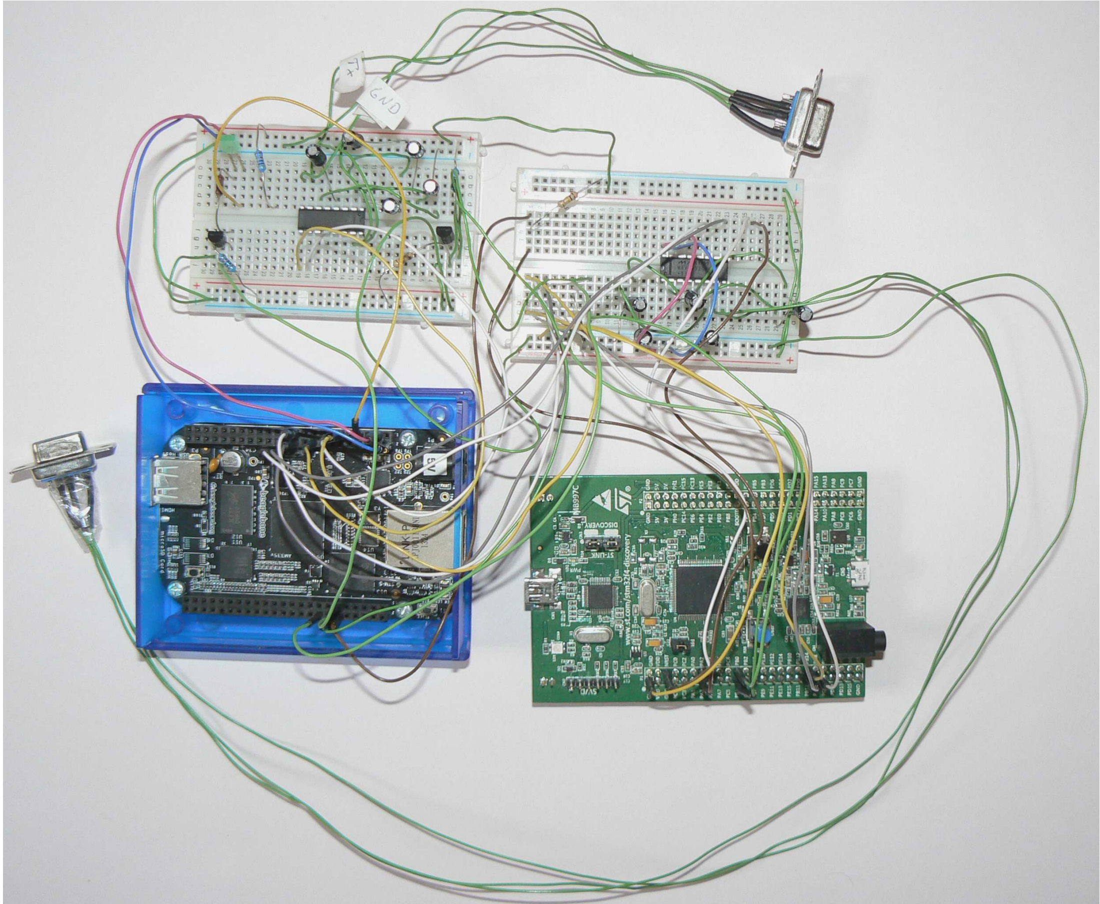


Figure 9.1: Photo of the hardware setup used – the setup contains additional LED and *MAX3232*, which serve for debugging (by lighting the LED and printing to a terminal connected through a serial port respectively

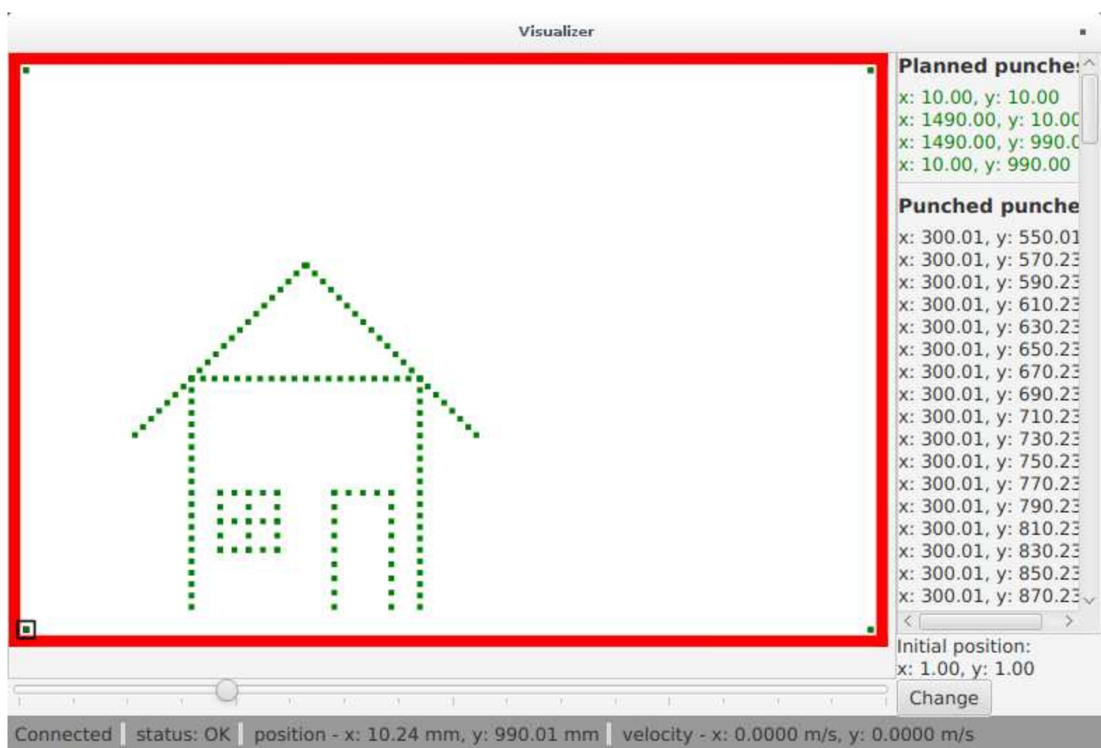


Figure 9.2: Screenshot of the visualizer application

10 Related Work

This chapter presents other works (by other authors), that deal with similar problems as this thesis. It is divided into two sections, each of which describes works of a similar type.

10.1 Educational Platforms

Each of the following paragraphs describes one educational platform, at the start of each paragraph is the name of the platform and a reference to an article about it.

A Platform for Real-Time Control Education with LEGO Mindstorms [8] – an education platform based on the *Lego Mindstorms* platform. Students learn by creation of controller programs for robots created from the *Lego Mindstorms*. The programs for the controller are created by a combination of modeling in the *Simulink* program and writing code in the *Ada* programming language.

A Teaching Platform for Embedded Systems Engineering [9] – an education platform. Consists of four microprocessor nodes connected into a network through a serial bus. Each of the nodes has access to different devices (real-time clock, thermal control path, microphone, displays, potentiometers, LED bargraphs) and through common serial bus they can all access a *Zigbee* wireless communication bridge. The platform is mainly focused on the thermal control path, on which students learn feedback control, and real-time networking. The thermal control path consists of a heat sink, resistors for heating the heat sink, a fan for cooling the heat sink and digital temperature sensors to measure temperature of the heat sink.

Teaching Platform for Lessons of Embedded Systems Programming [10] – an education platform consisting of *M68EVB908GB60* microcontroller evaluation kit, panel PC (*Advantech PPC-L60T*), *Advantech ADAM* modules and possibly other modules. The platform focuses on teaching real-time systems programming on both the microcontroller evaluation kit and the panel PC.

A Didactic Platform for Practical Study of Real Time Embedded Operating Systems [11] – an education platform. Firmware is based on the *FreeRTOS* operating system. Exercises are aimed at studying embedded operating systems (multitasking, interprocess communication, process synchronization). The platform consists of a main controller connected through *SPI* to an *I/O* module and through *RS485* bus to one or more identical execution modules. All hardware modules of the platform are custom made and are based on *AVR* microcontrollers.

10.2 Hardware-In-The-Loop Simulators

There are many companies that offer products that support hardware-in-the-loop simulation. Their products (mainly software for creation of simulators and hardware to run the simulation on), however, are costly and require powerful and costly hardware to run the resulting simulation on. Below is a list of several companies that offer hardware-in-the-loop simulation products. Each item of the list contains name of the company, the product and a link to their websites respectively.

- *Maplesoft*, *MapleSim* – software for creation of simulations, http://www.maplesoft.com/products/maplesim/
- *MathWorks*, *Simulink Real-Time* – software for creation of simulations, http://www.mathworks.com/products/simulink-real-time/
- *Concurrent*, *SIMulation Workbench* – software for creation of simulations, https://www.ccur.com/real-time/products/simulation-workbench/
- *RTDS Technologiew*, *RTDS Simulator* – both software and hardware for real-time power system simulation, https://www.rtds.com/
- *dSPACE*, *dSPACE Simulator* – hardware for a simulator, http://www.dspace.com/en/pub/home/products/hw/simulator_hardware.cfm

11 Conclusion

This thesis has pursued improving of teaching of real-time systems. Specifically, the practical part of teaching software development for real-time systems. Mainly, we intended for each student to get a hands-on experience with programming real-time systems hardware, learn feedback control of dynamic systems and reach these goals at low cost.

To reach the goal we have created a platform, on which students can learn during courses. The platform consists of a plant, a visualizer and a controller. The controller is connected to the plant and hosts the students' programs that drive the plant. It is based on the *stm32f4-discovery* development board. The plant represents a punch press, an industrial machine to be controlled by the controller using feedback control. The punch press machine is simulated in software, but has a hardware interface, through which it is driven by the controller. The plant runs on the *Beaglebone Black* development board. The visualizer is an application that displays the current state of the plant. It runs on the development computer of a student and is connected to the plant using serial port. Students learn on the platform by developing a program for the controller, the created program controls the plant.

The created platform fulfills the objectives of the thesis. It is suitable for teaching as it is low cost, has low space requirements and is durable (uses a simulated machine that cannot be physically damaged). The platform is aimed to be used together with a *Linux* based PC, but should be easily portable. Using the platform students can go through multiple learning steps (for example: blink a LED on the controller board; use a timer to blink the LED with a specified period; read the state of the plant; drive the plant using on-off feedback control; drive the plant using proportional feedback control; drive the plant using proportional and derivative feedback control).

The platform uses only software that we have created and freely available software. The visualizer runs on the development computer and requires no additional hardware. The plant and the controller both run on low cost development boards. The cost of the platform is thus under $100 (see Section 2.7.2 for more details).

To avoid time consuming search for suitable development tools and libraries, the controller base project was created. It consists of libraries that simplify hardware usage, a makefile to easily compile and debug the project, and a sample code. Students are expected to base their controller programs on the base project.

The resulting platform could be extended in several ways. The punch press simulator used does not take into account all physical aspects, that the punch press machine could have. The simulator could be replaced with another one that simulates the punch press more accurately.

The plant's platform could be used to host simulator of another machine in order to have more diverse assignments for students. The current plant teaches mainly feedback control, another plant could be aimed at other specifics of real-time

systems programming (a machine that would require cooperation of multiple tasks at the same time to be controlled; a machine that would require the integrating part of the PID controller to be used).

Unfortunately, the current base project does not utilize a real-time operating system. A more sophisticated base project would utilize one. Real-time operating systems usually contain many aspect from the real-time systems theory (for example different scheduling strategies) and are thus a good platform to learn to use them and test them. Furthermore, as the more powerful chips required to run an real-time operating system are getting cheaper, the real-time operating systems are more widely used.

Bibliography

[1] Bureš, Tomáš. *Specification of a punch press* [online]. [cit. 2015-07-29]. Available on: http://d3s.mff.cuni.cz/ bures/ers-files/lectures/Punch Press Specification.pdf

[2] Texas Instruments. *AM335x Sitara Processors Technical Reference Manual* [online]. [cit. 2015-07-29]. Available on: http://www.ti.com/lit/ug/spruh73l/spruh73l.pdf

[3] Coley, Gerald. *BeagleBone Black System Reference Manual* [online]. [cit. 2015-07-29]. Available on: https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true

[4] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* [online]. [cit. 2015-07-29]. Available on: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

[5] ST Microelectronics. *RM0090 Reference Manual* [online]. [cit. 2015-07-29]. Available on: http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf

[6] ST Microelectronics. *UM1472 User Manual* [online]. [cit. 2015-07-29]. Available on: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00039084.pdf

[7] Åström, Karl Johan and Murray, Richard M. *Feedback Systems: An Introduction for Scientists and Engineers*. United Kingdom: Princeton University Press, 2008. ISBN-13: 978-0-691-13576-2. ISBN-10: 0-691-13576-2.

[8] Bradley, Peter J., Puente, Juan A., Zamorano, Juan, Brosnan, Daniel. *A Platform for Real-Time Control Education with LEGO Mindstorms* [online]. [cit. 2015-07-29]. Available on: http://oa.upm.es/20410/1/INVE_MEM_2012_134452.pdf

[9] Staub, Arvid. *A Teaching Platform for Embedded Systems Engineering* [online]. [cit. 2015-07-29]. Available on: http://ti.tuwien.ac.at/cps/teaching/courses/networked-embedded-systems/materials/staub-bachelor-thesis

[10] Dolinay, J., Dostálek, P., Vašek, V., Vrba, P. *Teaching Platform for Lessons of Embedded Systems Programming.* 13th WSEAS International Conference on AUTOMATIC CONTROL, MODELLING & SIMULATION (ACMOS '11), 2011. ISBN: 978-1-61804-004-6

[11] Kaliszan, Adam and Głabowski, Maruisz. *A Didactic Platform for Practical Study of Real Time Embedded Operating Systems.* Internation Journal on Advances in Telecommunications, vol 4 nr 3&4, 2011. ISSN: 1942-2601.

List of Tables

2.1 Considered controller boards and their prices 13

2.2 Considered plant boards and their prices 14

3.1 Depiction of *encoders* command return value. 19

3.2 Depiction of *errors* command return value. 19

4.1 Depiction of the message format. 23

List of Figures

2.1 Software-in-the-loop Simulation Diagram . . . 7
2.2 Processor-in-the-loop Simulation Diagram . . . 8
2.3 Hardware-in-the-loop Simulation Diagram . . . 9
2.4 No Simulation Diagram . . . 12

3.1 Punch press working area . . . 16
3.2 Example code listing . . . 20
3.3 SPI Timing . . . 21

4.1 Format of the return value of the *status* command response. . . . 26
4.2 Format of the return value of the *session* command response. . . . 26

5.1 Connection schema of the plant and the development machine . . 29
5.2 Conncetion schema of the plant and the visualizer . . . 30

6.1 Architecture of the plant . . . 33
6.2 Simulation update computation . . . 36

7.1 Architecture of the *Visualizer* . . . 45
7.2 A screenshot of the *Visualizer* application . . . 46

8.1 *Eclipse IDE* with the base project open . . . 48

9.1 Photo of the hardware setup used . . . 50
9.2 Screenshot of the visualizer application . . . 51

List of Abbreviations

API – Application Programming Interface

ASCII – American Standard Code for Information Interchange

CLK – Clock

DHCP – Dynamic Host Configuration Protocol

FIQ – Fast Interrupt Request

GPIO – General Purpose Input Output

I2C – Inter-Integrated Circuit

I/O – Input Output

IRQ – Interrupt Request

JSON – JavaScript Object Notation

LED – Light-Emitting Diode

MISO – Master In Slave Out

MOSI – Master In Slave Out

PRU – Programmable Real-Time Unit

RTOS – Real-Time Operating System

SPI – Serial Peripheral Interface

TTL – Transistor-Transistor Logic

UART - Universal Asynchronous Receiver Transmitter

USB – Universal Serial Bus

XML – Extensible Markup Language

YAML – YAML Ain't Markup Language

A Contents of the Attached Disc

This appendix describes the content of the attached disc. Not all folders and files on the disc are described. Only the main structure is described. The content is described in the list and sub-lists below. Each item on the list corresponds to one folder on the disc, each item on a sub-list corresponds to a sub-folder. Every list item starts by a name of its corresponding folder.

- *controller* – contains programs for the controller, the programs are written in *C* and use additional open source libraries.
  - *base* – contains the base controller project, which is expected to be used by students as a base for their controller projects.
  - *evaluation* – contains the controller program used for evaluation of the platform.
- *plant* – contains the plant's programs. All the plant's programs except the *SPI Controller* are written in C, the *SPI Controller* program is written in an assembly language.
  - *hpitq_test* – contains the *Linux* kernel module used to test that the interrupt prioritization, that we implemented, works.
  - *kernel* – contains the code of the *Linux* kernel used (3.8 version with additional patches), including our alterations that allow prioritization of interrupt requests.
  - *pasm* – contains an assembler used for compilation of programs written for the *Programmable Real-Time Unit*. The assembler is an open-source program, that we used.
  - *pru_driver* – contains the *Linux* kernel driver for the *Programmable Real-Time Unit* peripheral (*PRU Driver* module), including our alterations. The driver is compiled as a kernel module. The driver is open source and we reused it.
  - *simulator* – contains the main part of the *Simulator* module and the *SPI Controller* component.
  - *spi_driver* – contains the *SPI Driver* module, which is a Linux kernel driver for the *SPI* peripheral of the plant's processor.
  - *visualizer_interface* – contains the *Visualizer Interface* application.
    - *cJSON_lib* – contains the *cJSON* library, it is a library for parsing and generation of data in *JSON* format. The library is open source and we reused it.
    - *serial_lib* – contains the library used by the *Visualizer Interface* for message communication over serial port.
- *visualizer* – contains the Visualizer program and reused open source libraries that it requires. The program is written in *Java*.

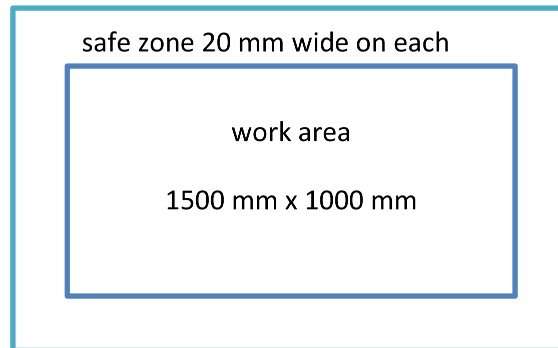
- *plant* – contains the *PlantState* module of the *Visualizer*.
- *serial* – contains the *SerialCom* module of the *Visualizer*.
- *view* – contains the *Visualizer* module of the *Visualizer*.

B Plant Interface Specification

This appendix contains the plant interface specification which we use in Chapter 3. The text is verbatim taken from [1]. The text of the specification starts on the next page.

Specification of a punch press[1]

The work area of the punch press is 1500 mm x 1000 mm. There is a 20mm wide border around the whole area called safe zone.



The punch press has a moving head which may move freely over the work area and the safe zone. Entering the safe zone with the center of the head is signaled by signals SAFE_L, SAFE_R, SAFE_T, SAFE_B. When the head is move outside the work area and the safe zone, FAIL is signaled and the whole punch press stops. To recover from this condition, reset of QEMU is required.

The head is controlled by PWR_X and PWR_Y, which may take values from -128 … 127. The values signify the power applied on the head. Positive values mean forward motion – i.e. to right or down. Negative value means backward direction – i.e. to left or up. It may take up to 1 ms until the new power is applied.

The head is subject to friction forces, thus a certain minimal power (approx. 18) is needed to get it into movement. When zero power is set, the head gradually slows down until it stops. It is however possible to apply opposite force to make the stop faster.

The movement of the head may be monitored using two quadrature encoders ENC_X and ENC_Y, which generate sequences 00, 01, 11, 10 for forward motion and sequences 10, 11, 01, 00 for backward motion. This sequence repeats once for 1 mm distance.

To punch a hole using the head, signal PUNCH is used. The signal has to remain set for at least 1ms. After that, it has to be cleared for at least 1ms. After the punch, it takes some time for the head to be lifted up its original position. This is signaled by HEAD_UP. When the signal is set, another punch may be performed or the head may be moved.

[1] Inovace tohoto kurzu byla v roce 2011/12 podpořena projektem CZ.2.17/3.1.00/33274 financovaným Evropským sociálním fondem a Magistrátem hl. m. Prahy.

OPP A PRAHA PRAGUE PRAGA PRAG EVROPSKÁ UNIE

Evropský sociální fond
Praha a EU: Investujeme do vaší budoucnosti

To perform the PUNCH, the head has to be still – that means its velocity along X as well as Y axis must be less than 0.0001 m/s (per axis). The head has to stay still until HEAD_UP is signaled. A failure to do so will result in FAIL mode. To recover from this condition, reset of QEMU is required.

The signals are mapped to ports 0x7070 – 0x7072 in the following way:

Output ports:

Port number	Bits	Name	Description
0x7070	0-7	PWR_X	Power applied along the X axis in the range of -128 .. 127.
0x7071	0-7	PWR_Y	Power applied along the Y axis in the range of -128 .. 127.
0x7072	0	PUNCH	Signal to punch a hole at the current head position.
0x7072	1	IRQ_ON	When set, IRQ (default 5) is sent with every change of encoders (ENC_X and ENC_Y).

Input ports:

Port number	Bits	Name	Description
0x7070	0-1	ENC_X	Quadrature encoder along the X axis.
	2-3	ENC_Y	Quadrature encoder along the Y axis.
	4	SAFE_L	Set when the center of the head is in the area left of the work area.
	5	SAFE_R	Set when the center of the head is in the area right of the work area.
	6	SAFE_T	Set when the center of the head is in the area top of the work area.
	7	SAFE_B	Set when the center of the head is in the area bottom of the work area.
0x7071	0	HEAD_UP	Set when the head is ready for move or punch.
	1	FAIL	Signals the error condition. Reset of QEMU is needed.

Execution of the platform

QEMU with the punch press simulator may be executed in the following way:

qemu -gdb tcp:localhost:10000 -device ers_panel8 -device **ers_punch_press,infile=punches.in,outfile=punches.out -icount 8** -m 128 -boot a -fda boot.img -hda fat:./hda --no-reboot

The simulator reads the file punches.in, which states positions of planned punches. The format of the file is the following:

<x coordinate of the 1st punch in mm from left> <y coordinate of the 1st punch in mm from top>
<x coordinate of the 2nd punch in mm from left> <y coordinate of the 2nd punch in mm from top>
<x coordinate of the 3rd punch in mm from left> <y coordinate of the 3rd punch in mm from top>
<x coordinate of the 4th punch in mm from left> <y coordinate of the 4th punch in mm from top>
…

The actual punches are produced in the same format to file punches.out.

The file names may be left empty (by omitting also the parameter infile or outfile). In that case, the input/output files are not used.

The -icount parameter ensures synchronous time so that time for redraws and computation of the head position is not accounted to the application being emulated. If RTEMS has problems with time synchronization during its start, you may increase the parameter to 9 or 10.