A Desktop 3D Printer in Safety-Critical Java

Tórur Biskopstø Strøm

Kongens Lyngby 2012
IMM-MSc-2012-141

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-141

Summary

It is desirable to bring Java technology to safety-critical systems. To this end The Open Group has created the safety-critical Java specification, which will allow Java applications, written according to the specification, to be certifiable in accordance with safety-critical standards. Although safety-critical Java framework implementations are well under way, there is a lack of safety-critical use cases implemented according to the specification.

This thesis presents a RepRap 3D desktop printer as a use case. As a part of the thesis it is implemented as a safety-critical Java level 1 application. Based on the implementation the specification and its usability is evaluated. It is shown that whilst there are several problematic areas in safety-critical Java, such as WCET analysability and lack of garbage collection, it is still possible to create a functioning RepRap in safety-critical Java.

Preface

This thesis was prepared between July the 1st and December the 2nd 2012 at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Engineering. The thesis was done under supervision of Prof. Dr. Martin Schoeberl and credited at 40 ECTS points.

Lyngby, 02-December-2012

Tórur Biskopstø Strøm

Acknowledgements

I would like to thank my supervisor Prof. Dr. Martin Schoeberl for the opportunity of working with this project. I also appreciate that as a part of the project I was able to publish a paper with him and attend the JTRES 2012 conference.

I would also like to thank Christian Riis Sørensen for the initial interface board layout and thank both him and Lasse Møller for general help with the electronics.

Contents

Summary **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**
1.1 Project Goal 2

2 Background **3**
2.1 Safety Critical Java 3
2.2 Java Optimized Processor 5
2.3 RepRap 6
2.4 Related Work 9

3 Implementation **11**
3.1 Overview 11
3.2 RepRap 12
3.3 Interface Board 13
3.4 FPGA 14
3.5 Firmware 17

4 Evaluation **23**
4.1 SCJ Programming Experience 23
4.2 Schedulability 26
4.3 Safety-Critical Java vs. C 29
4.4 SCJ RepRap Test 31

5 Conclusion **33**

A Interface Board Schematic **35**

B Interface Board Layout **37**

Bibliography **39**

Chapter 1

Introduction

The popularity of Java has spawned projects that have brought Java into several areas, including real-time systems. A continuity of this process is done by The Open Group with the safety-critical Java (SCJ) specification [LAB+12], such that Java can be used in certifiable safety-critical (SC) applications. This specification is a work in progress. To properly evaluate the expressiveness of the specification, the simplicity of the API, and the ease in which safety-critical applications can be written in Java, it is necessary to have SCJ and use case implementations.

SCJ implementations are already on the way [SKR12, SR12], however use cases are still very rare. As a part of this thesis a RepRap 3D desktop printer is implemented as a use case, and based on the use case the specification is evaluated. A RepRap is a desktop printer capable of creating 3-dimensional (3D) objects in plastic [Rep12e]. Some of the components of a RepRap are printable by the RepRap itself, making the RepRap partially self-replicable. The 3D drawings are interpreted by a host computer (a normal PC) and printing instructions are sent to the RepRap controller (firmware). The RepRap controller interprets the instructions, moves the printing head, heats the plastic and extrudes it. This controlling has real-time constraints. In this project the microcontroller is substituted with an FPGA board and the firmware is written from scratch as a SCJ application.

As SCJ platform the Java processor JOP is used on an Altera DE2-70 FPGA platform. The FPGA platform allows application specific I/O devices to be built for accessing the sensors and actuators of the RepRap. Besides implementing the firmware in SCJ, and some hardware components on the FPGA, this project also includes the RepRap assembly and the development of the electrical circuit interface between the RepRap and FPGA platform.

The main contribution of the project is the first real SCJ-based application controlling a robot and providing it as open-source (see Section 3.5 for source). Feedback on the SCJ specification and API is also provided from the point of view of a Java programmer. The following published paper on the subject was written and presented at the JTRES 2012 conference [JTR12] as a part of the project as well:

- Tórur Biskopstø Strøm and Martin Schoeberl. A desktop 3d printer in safety-critical java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 72–79, New York, NY, USA, 2012. ACM

The thesis is organized as follows: Chapter 2 gives background information on the project and technologies used. The various implementations done in the project, such as the RepRap, firmware, etc., are described in Chapter 3. Chapter 4 evaluates the SCJ specification and Chapter 5 concludes the thesis.

1.1 Project Goal

There is a lack of SC use cases implemented in SCJ and without use cases it is not evident whether SCJ is complete or if there is functionality missing. A lack of use cases also means that its usability is not properly evaluated, i.e., does it provide improvements compared to other SC frameworks such as ease of use, performance improvements, etc. The goal of this project is to use the RepRap as a real-world use case and implement it as a SCJ level 1 application. Based on the implementation and its development the SCJ specification and its expressive power is evaluated.

Chapter 2

Background

In this chapter the project and background knowledge is specified. First the project's main technologies are explained in Section 2.1, 2.2 and 2.3, after which related work is discussed in Section 2.4.

2.1 Safety Critical Java

The Safety-critical Java (SCJ) specification intends to bring Java technology to safety-critical (SC) systems [LAB+12]. A system can be thought of as SC when its failure may cause extensive damage to equipment and environment, often resulting in human injury or death [LAB+12, p. 2-3]. SC systems therefore require high level of assurance that they will not fail. This assurance is achieved by extensive validation and certification, which is both time-consuming and expensive. The SCJ specification is therefore designed to enable applications, written according to the specification, to be certifiable according to SC standards, such as DO-178B, Level A.

Each SCJ application consists of a single `Safelet`, which acts as an entry class similarly to `Servlet`. The `Safelet.getSequencer` returns a `MissionSequencer`, which defines a sequence of `Mission` to be executed. A mission can be thought

of as a mode of operation and all SCJ applications have at least one mission, i.e. at least one mode of operation. Each mission has three phases: initialization, execution and cleanup. At startup the first mission defined by the mission sequencer goes through its initialization phase and into its execution phase. If there is more than one mission defined in the mission sequencer, the next mission will initialize and execute after the previous mission has finished its cleanup phase.

A mission has a number of schedulable objects (SOs) which correspond to tasks in schedulability analysis. These are created during mission initialization. During the execution phase a mission's SOs are scheduled by a fixed-priority scheduler shipped with the SCJ implementation.

In SCJ the garbage collector (GC) has been replaced with a scoped memory model. Memory is split into scopes, with scopes being nested and sometimes parallel. A program will enter and exit a number of scopes during execution. When entering a scope it is active until another scope is entered or the current scope is exited, which makes the outer scope active. It is only possible to enter inner or outer scopes, not parallel scopes. All objects created when a scope is active are allocated in the scope. When an innermost scope is exited, the scope and all objects created within are deallocated, thereby freeing the scoped memory area. This allows the creation and cleanup of garbage without using a GC. It should be noted that entering either an inner or outer scope does not deallocate objects in the current scope. The process is similar to method invocation where stack allocated variables are cleaned up when the invocation completes.

As seen in Figure 2.1 every application has three types of memory: `ImmortalMemory` (IM), `MissionMemory` (MM) and `PrivateMemory` (PM). There is only one IM and the whole application can access it. MM is created during a mission's initialization and is only accessible by the mission and its SOs. Every SO has a PM which is entered when the SO executes, and exited when the SO is released. Note that a SO's PM is not exited when a SO is preempted. Instead the newly scheduled SO's last active scope is activated.

SOs can create inner PMs, as seen in Figure 2.1 where one of the SO's PM has a nested PM. This allows the creation and cleanup of garbage without having to finish a SO's execution.

There are three compliance levels defined in SCJ, with 0 being the least complex and 2 being the most complex. The compliance levels are used both for the platform and for the applications. An application should be able to execute on a platform that has the same or higher compliance level, e.g. a level 1 application can execute on level 1 or 2 platform, but not on a level 0 platform. Applications

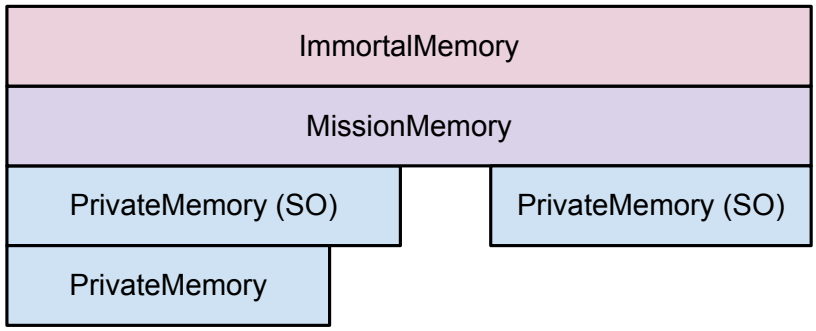


Figure 2.1: Scoped memory example

of any compliance level can consist of multiple sequential Missions.

In a level 0 application the only SOs supported are `PeriodicEventHandlers` (PEHs). Level 0 uses a cyclic execute programming model, where essentially the schedule is predefined, so on a level 0 platform none of the PEHs will interrupt each other.

Level 1 introduces concurrency, with computations being done with either PEHs, that execute with a fixed period, or `AperiodicEventHandlers` (APEHs). All SOs have a fixed-priority and are interruptible by a higher priority SO. Synchronized blocks are not allowed in SCJ, so synchronizations are done using synchronized methods. Level 1 and 2 are designed to be executable on multicore systems.

Level 2 expands on level 1 with nested missions. Although a level 2 application starts with an initial Mission sequence, it can create further sequences that are able to run concurrently.

2.2 Java Optimized Processor

The Java Optimized Processor (JOP) is a hardware implementation of the Java Virtual Machine (JVM) [Sch08]. It is intended to be time predictable and thereby worst-case execution time (WCET) analysable. JOP uses Java bytecodes as instruction set, although internally the processor translates bytecodes

into one or more microcodes which are then executed.

To increase execution speed JOP uses an instruction cache. Caching individual instructions can result in an overly complex WCET analysis, so instead whole methods are cached [Sch04]. Cache misses are therefore only possible on method invocations and returns. Most bytecodes have constant execution time on JOP, with variability mainly stemming from cache misses.

JOP is a soft-core processor implemented in VHDL, a hardware description language. This allows JOP to be realized in different ways on different platforms, though most commonly through logical synthesis on an FPGA. Using an FPGA also allows extending JOP with additional hardware components and interfaces, making it a modular microcontroller.

Additional hardware modules are connected to JOP using SIMPCON [Sch07]. The modules can then be accessed from the Java application using hardware objects [SKKR11]. From a programmer's perspective they are similar to other Java objects, however their attributes are linked to a hardware address through SIMPCON. Reading or writing to the attributes therefore corresponds to reading or writing to the hardware module.

2.3 RepRap

The Replicating Rapid-prototyper (RepRap) is an open source desktop printer capable of creating 3-dimensional (3D) objects in plastic [JHS+11]. Several of its components are made of plastic and are printable by a RepRap, making the machine partially self-replicable. There are several variants of the RepRap with different focus areas [Rep12e]. For this project the Prusa Mendel is used, which focuses on low cost and ease of sourcing.

The Prusa Mendel has 5 stepper-motors driving 4 axis (see Figure 2.2): X,Y,Z and E, with E being the extrusion dimension. Filament (plastic) is fed into the extruder (printing head) which has a heater that melts the filament and pushes it out through a thin nozzle. The length of filament extruded corresponds to E.

In a standard setup the host, running RepRap host software, reads a 3D drawing, such as a Standard Tessellation Language (STL) file from a computer-aided design (CAD) application, and "slices" it into printing instructions called G-codes [Rep12b]. The G-codes are sent to the RepRap firmware which executes them, moving the extruder to the instructed coordinates whilst heating and extruding filament.

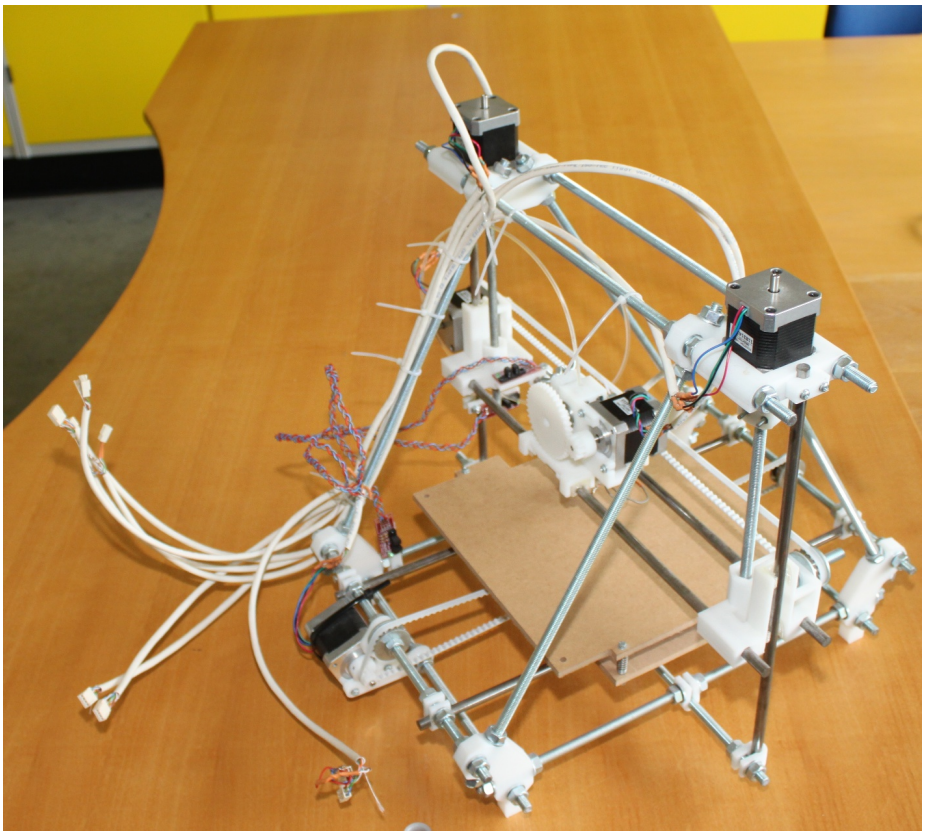


Figure 2.2: RepRap - Prusa Mendel variant

```
1 G1 X90.6 Y13.8 E22.4 F800
2 N7 G1 X2.0 Y2.0 F3000.0*85
3 G92 E0
```

Figure 2.3: G-code examples

The host is any computer capable of running RepRap host software. There are many variants of host software, with some only slicing, in which case the slicer creates a G-code file which is then put on an SD-card for the printer. For this project the Printrun toolchain [Kli12] is used which combines a slicer, a transceiver and a GUI, and requires a serial line between the RepRap controller and the host.

A G-code consist of 1 or more fields, with each field having a letter followed by a number. A single field denotes the command type, whilst other fields can be thought of as command parameters. There might also be two fields, N and *, that are used as checksum. G-code examples are shown in Figure 2.3. The first command is a buffered movement command that tells the extruder to move to the X and Y coordinates (90.6 mm, 13.8 mm) whilst extruding filament from the currently extruded length up to 22.4 mm. This move has to be done with a speed of $800\frac{mm}{s}$. The next command is similarly a move command but excludes extrusion and includes a checksum. `N7` indicates that this should be command number 7 in the series. `*85` specifies that the entire line up to, but not including, `*` has the checksum value 85. The final command resets the measured extruded length to 0, i.e., the current E position is set to 0. This command is typically used after several extrusion commands, allowing extrusion to be measured in absolute values without overflowing, instead of using relative positioning.

The RepRap does not fall under the definition of a safety-critical system where human lives might be in danger [LAB+12, p. 2-3], however it is still a useful use case. There are a number of real-time requirements such as ensuring that stepper motors move at a specified speed, measuring temperature and maintaining the temperature. The temperature has to be high before the filament is optimally extruded, e.g. as high as 230°C for PLA [Rep12d], but it should not overshoot, as too high temperatures might destroy the filament and RepRap components. It is therefore reasonable to implement the firmware as a SCJ application, where all measurements and controls are done within well defined timing boundaries.

2.4 Related Work

While there is a lack of use cases for SCJ, some related work has been done. In [PZS+10] the CDx benchmark is ported to the SCJ level 0 compliant oSCJ framework. The benchmark is used to evaluate performance of the framework compared to the equivalent C code. Their conclusion is that SCJ implementation delivers predictability and throughput rivalling the C implementation. In this project a level 1 compliant framework is used instead and there is more focus on the problems a developer may encounter when using SCJ for a real world use case.

In [SJL+09] a framework called PERC Pico is described, which slightly diverges from SCJ. The paper describes the porting of it to two ARINC 653 compliant operating systems. A simplified flight warning system developed by THALES is used to validate the porting. It is desirable to implement genuine SC use cases in SCJ, such as a flight warning system, however they are not readily accessible. The RepRap is freely available and, depending on the implementation, provides a real-time use case capable of testing a large part of the SCJ specification.

A recent project presented at JTRES 2012 uses a cardiac pacemaker as a use case [SWC12]. The pacemaker is implemented both in Ravenscar Ada [BDR98] and SCJ, and the two implementations are compared to each other. The conclusion is that SCJ is missing a watchdog timer and it is expected that future versions of SCJ will support one-shot timers. In the RepRap project all tasks are time triggered, so one-shot timers have not been missed. Furthermore the pacemaker project focuses mostly on the concurrency and timing models supported by SCJ, whereas the RepRap project also considers dynamic memory management. On the other hand the pacemaker project makes use of multiple missions corresponding to the different operating modes, as suggested my the SCJ specification, whereas the RepRap project only has a single mission. The single mission model was chosen because the RepRap does not need multiple modes of operation, but instead a single mode that supports all the necessary commands.

While this project uses SCJ on JOP [SR12] another option is to use the Hardware near Virtual Machine (HVM) and its SCJ implementation [SKR12]. A combination of Java interpretation and Java-to-C compilation is used for the HVM, allowing easy porting to platforms that have a C compiler as a part of the development environment. It is designed for low-end embedded systems, which might be interesting for the SCJ RepRap firmware, since most RepRap controllers run on low-end systems. At JTRES 2012 a future project was suggested to port JOP's RepRap firmware to the HVM platform. The HVM platform uses hardware objects [SKKR11] for device access similarly to JOP, which

should make porting easier.

Chapter 3

Implementation

In this chapter the various implementation details and design decisions are described. Section 3.1 provides an implementation overview followed by the RepRap implementation (mechanical parts) in Section 3.2. The interface board is described in Section 3.3 with Section 3.4 providing the implementation details of the FPGA components. Finally the SCJ RepRap firmware is presented in Section 3.5.

3.1 Overview

The implementation of a RepRap as a SCJ application has required knowledge from several fields. Although the RepRap project is open source, with source files freely available, the use of JOP for the firmware has required customizing the software and hardware, including changes to JOP. This is mentioned in the respective sections when applicable.

An overview of the implementation is shown in Figure 3.1, with arrows showing the communication flow. The host computer, executing the Printrun software, slices an STL file into G-codes and transmits them to the FPGA over a serial line. The FPGA, configured with JOP and executing the RepRap firmware,

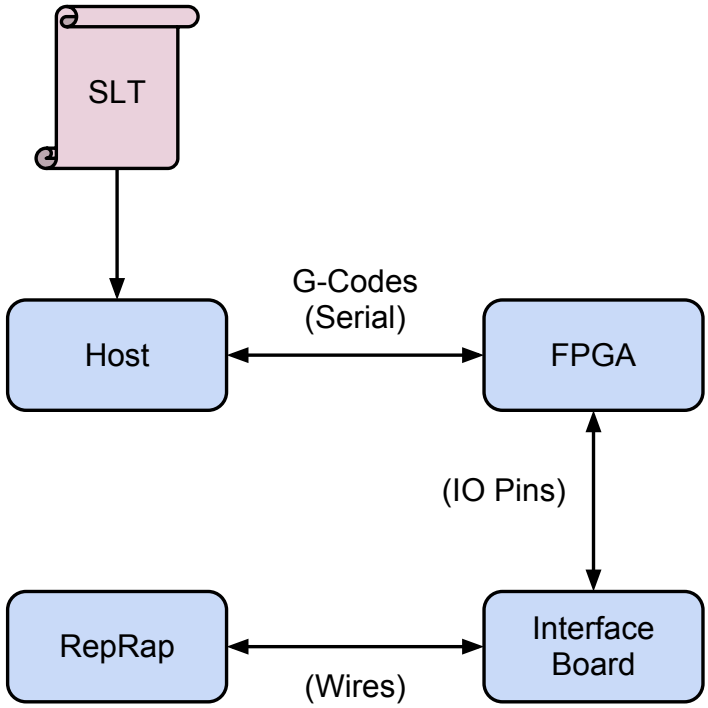


Figure 3.1: Hardware Layout

analyses the G-codes and executes them, controlling the RepRap through an interface board.

3.2 RepRap

There are different ways to acquire a RepRap all the way from building one from scratch to buying a fully functioning one. Purchasing a finished RepRap can be quite expensive, so this has not been an option. Instead, metal parts (nuts, bolts, threaded rods, etc.) and printed plastic parts have been bought as partial kits from two providers. The rest, such as stepper-motors, cables and electronics, have been acquired from various sources. The optical end-stops, used to sense when the X, Y or Z axis have reached their 0 point, have been

bought as kits, with only assembly necessary.

The instructions at [Rep12a] have been used to assemble the Prusa Mendel RepRap. They are quite thorough, but since this project uses custom electronics for the firmware, the instructions have only been usable for the mechanical parts. The result is the RepRap shown in Figure 2.2

The extruder uses a brass nozzle with a resistor as heating element, simply by running current through it. The resistor can handle temperatures of more than 250°C, which is more than enough to melt PLA filament. A thermistor embedded in the heating block allows the temperature to be measured by measuring its resistance.

3.3 Interface Board

The interface board, shown in Figure 3.3, has been designed with the help of two fellow engineering students. The schematic and layout is shown in Appendix A and B respectively. The purpose of the board is to convert the logical, low voltage IO pins of the FPGA to higher voltage pulses used for the heater and stepper-motors on the RepRap. The board is designed to be used with the 12V 4-pin connector of a standard PC power supply unit. The 12V line is shifted to a 5V line, which is further shifted down to a 3.3V line.

The board contains 5 slots for Pololu [Rep12c]/Stepstick [Rep12f] stepper-motor drivers. These are used to easily control the stepper-motors. When controlling stepper-motors it is desirable to run with a high voltage to increase reaction time, and thereby allow higher speeds. However, this requires active current limiting to prevent damaging the motors. The drivers enforce this and also act as a simple interface for controlling the motors, with one logical pin controlling the direction and another one for pulses. Every pulse to the driver results in a motor step in the set direction. The drivers use the 3.3V line as power supply and the 12V line for the motors.

For the heater the 12V line is connected to the heater's resistor through a MOSFET. A logical pin from the FPGA is connected to the MOSFET's gate, allowing the FPGA to control the current flow (on/off).

Initially an analog-to-digital converter (ADC) chip was used to measure the thermistor's resistance, however the I2C communication with it never worked. Instead the method from [Tel12], shown in Figure 3.2, is used. One logical pin (pin 1) is connected through the thermistor to another logical pin (pin 2), with

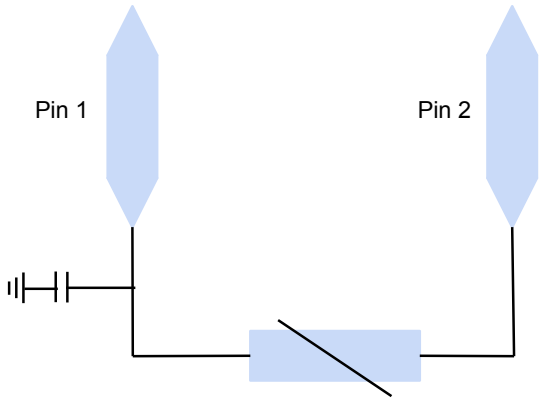


Figure 3.2: Temperature sensor consisting of thermistor, two FPGA IO pins and capacitor

a decoupling capacitor between pin 1 and the thermistor. First the capacitor is discharged by setting both pins low. Pin 1 is then set as input and pin 2 is set high. The time it takes pin 1 to become a logical 1 depends on the thermistor's resistance, which means the temperature can be derived by timing this process.

3.4 FPGA

For the project an Altera DE2-70 board with a Cyclone II FPGA is used. The board comes equipped with several interfaces with mainly the Serial and PATA interfaces being of interest to the project. As shown in Figure 3.1, the serial interface is connected to the host and The PATA interface is used to connect the FPGA with the interface board, forwarding the FPGA logical IO pins.

The FPGA is configured with the JOP implementation, as shown in Figure 3.4. On top of JOP a SCJ level 1 compliant framework [SR12] is running, with the SCJ RepRap firmware executing on top of the framework.

The two interfaces are available to the firmware as hardware objects. For the serial interface the existing JOP implementation is used, however there are a couple of considerations for the application. The default baud rate used in JOP is $115200\frac{bit}{s}$. JOP's serial implementation is such that 8 bits are processed si-

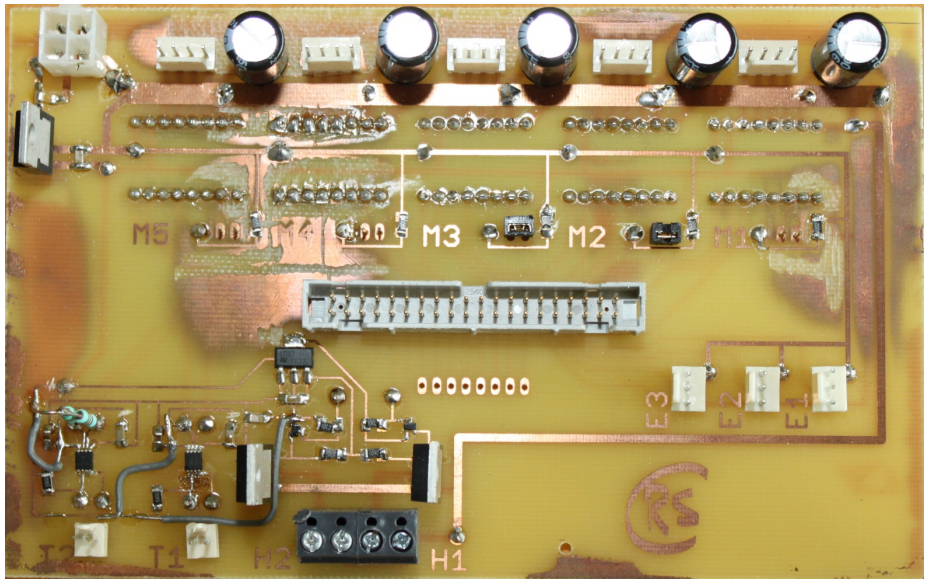


Figure 3.3: Interface Board

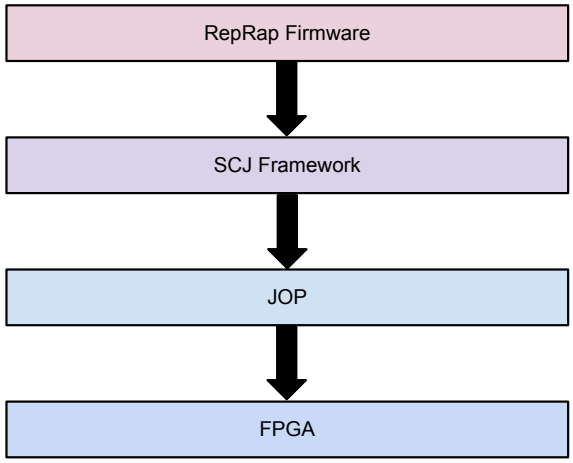


Figure 3.4: Firmware layers

```
1  process(clk, reset)
2  begin
3          if(reset='1') then
4                  readtemp <= '0';
5                  cntvalue <= (others => '0');
6                  cnt <= (others => '0');
7          elsif(rising_edge(clk)) then
8                  if(readtemp = '1') then
9                          if(GPIO_0(30) = '1' OR cnt = 20000000) then
10                                 cntvalue <= cnt;
11                                 cnt <= (others => '0');
12                                 readtemp <= '0';
13                         else
14                                 cnt <= cnt + 1;
15                         end if;
16                 else
17                         if(cnt = 20000000) then
18                                 readtemp <= '1';
19                                 cnt <= (others => '0');
20                         else
21                                 cnt <= cnt + 1;
22                         end if;
23                 end if;
24         end if;
25 end process;
```

Figure 3.5: Custom AD converter in VHDL

multaneously. The shortest period available for a PEH in JOP's SCJ framework is 1 ms. This means that to avoid loosing any characters, before a PEH can process them, JOP's serial receive buffer must be more than $\frac{115200}{8*1000} = 14.4$ bytes. This is under the assumption that the PEH will empty the receive buffer when it is running. To allow some room to receive characters while the PEH is reading from the buffer, it is set to 16 bytes.

Most of the PATA pins are either readable or writable in the firmware as a hardware object. This means that pins connected to the end-stops and motor drivers can be read/written directly from the firmware. Two of the pins are not directly accessible but instead used for the custom ADC mentioned in section 3.3. The module shown in Figure 3.5 is implemented as a part of the ADC. The code is implemented by assuming a clock rate of 60 MHz. The capacitor is discharged for $\frac{1}{3}$ of a second and is then charged until a logical 1 is read on the input or $\frac{1}{3}$ of a second is timed. The charge time in microseconds is then readable from the same hardware object used for the other IO pins. All IO is therefore handled by only 2 hardware objects.

3.5 Firmware

The SCJ RepRap firmware is implemented from scratch. Available RepRap firmwares are implemented in C/C++ and do not make use of existing real-time or safety-critical libraries, which makes porting troublesome, especially considering SCJ's memory scopes.

The firmware is designed as a SCJ level 1 single mission application and consists of 4 PEHs, as shown in Figure 3.6. The idea is to have a pipeline where a G-code is received (`HostController`), parsed (`CommandParser`) and executed (`CommandController` and `RepRapController`). Although all stages are created as PEHs, the possibility of using APEHs for some processing was suggested at JTRES 2012. It would be possible to implement the CommandParser and the CommandController as APEHs, since they do not need to periodically communicate with hardware.

The communication between the firmware and the host software uses a simple protocol: At startup the firmware is waiting for G-codes. Each code received from the host must be confirmed by sending an acknowledgement back to the host. Since the host does not send any further codes until it receives an acknowledgement, the firmware can control the communication flow and avoid overflowing the command queue. If checksums are used in the communication a corrupted code triggers a resend request by the firmware to the host, which resends the requested code line.

For each type of G-code used there is a respective command class who's instances represent received valid G-codes. Each command class has an object pool with at least one instance of itself that resides in MM. Instead of creating a new command object when the G-code has been parsed, the respective object is referenced from MM and its parameters are set. This allows the object reference to be passed between PEHs. Command classes that have more than one object in the pool represent G-codes that are buffered, i.e., the firmware should send an acknowledge to the host as soon as the code is verified, and not wait until it has executed like the other codes.

The HostController represents the transceiver and handles the serial communication with the host computer. It executes every millisecond to match the serial line's baud rate. It has a send buffer and a receive buffer. When executing, 14 characters are copied from the send buffer to the serial output buffer. The character limit ensures that the serial line buffer does not overflow, i.e., $14 < \frac{115200}{8*1000} <=> 14 < 14.4$. Afterwards up to 16 characters are copied from the serial input buffer to the receive buffer. As PEHs have guaranteed response times, given a valid schedule, no characters are left unprocessed as long as the

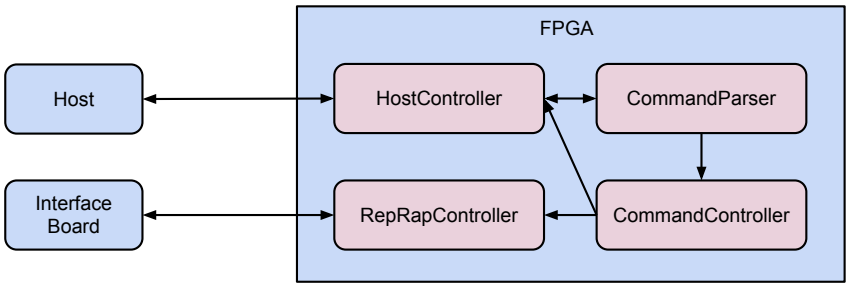


Figure 3.6: PeriodicEventHandler communication

host does not send characters faster then the agreed upon baud rate. Characters belonging to G-code comments are not stored to decrease the necessary size of the receive buffer, although the host might remove comments before sending the G-code. The receive buffer is marked ready when a command delimiter is registered. Until the CommandParser has copied the characters from the buffer the HostController does not store further received characters. Although this might seem to indicate that the two PEHs should be merged, this is not feasible when considering the baud rate requirement and the schedulability analysis shown in Section 4.2. Furthermore, according to the protocol the host should not send further characters until the firmware has sent its acknowledgement.

At JTRES 2012 it was suggested to use two ring buffers so that the CommandParser could read from the buffer while the HostController could add to it. This solution would avoid larger synchronization blocks and saves memory, however WCET analysis has shown that copying individual characters from the buffers and processing them takes too many cycles compared to copying the entire character set at once and letting the CommandParser process the set in its own time.

The CommandParser represents the parser. It executes every 60 ms, as this is the fastest rate based on the WCET analysis. The CommandParser polls the HostController for a ready character set and, if ready, copies it into its own character buffer. The buffer is then parsed. If the set is invalid, either because of an invalid checksum or because of an incomplete command, the CommandParser enqueues a resend request in the HostController. If the set represents a valid command the respective command object is pulled out of its pool and enqueued in the CommandController. If the command is buffered an acknowledgement is enqueued in the HostController as well.

The command pools are limited in size but the number of buffered commands that the host may send is not. If a command pool is empty no command is enqueued. Instead the CommandParser marks its buffer as unprocessed and no reply is sent to the host, to prevent the host from sending further commands. Whenever the CommandParser subsequently executes it will not copy another character set from the HostController, but instead parse the buffer again until the command is successfully enqueued, after which the CommandParser will execute normally.

The CommandController represents the command executer. Similarly to the CommandParser it is also scheduled every 60 ms. It has a command object queue which is traversed in FIFO order. When an object is pulled out of the queue the CommandController calls the command's `execute` method, which performs the command. If the command needs to interact with the RepRap or host as part of the execution, such as setting the next movement position, it calls the necessary methods in the RepRapController or HostController. After a command has finished execution it is returned to its respective command pool.

The RepRapController handles all communication with the RepRap hardware and manages positioning, heating and sensors. It is set to execute every millisecond so that the stepper-motor drivers can be pulsed as fast as possible.

When controlling the hardware first the values of each individual output pin is calculated, after which all the output pins are set at the same time. This ensures that the motor drivers are pulsed as synchronously as possible, ensuring better diagonal moves. If there is delay between pulsing the drivers a diagonal line can instead become a "staircase step" as shown in Figure 3.7. Here the driver for the Y axis is pulsed after the X axis, skewing the line.

Since the firmware entirely controls the motor driver pulses it is worth mentioning that this is a limiting factor in the movement speed. Each driver responds to the rising edge of a pulse, which must come down at some point before the next pulse can be issued. The RepRapController has a period of 1 ms, currently the lowest possible on JOP's SCJ framework, with one period being used to set the pulse high and the next period to set it low. This limits the maximum speed to $500\frac{steps}{s}$. There is however a solution that might solve this problem. The reason for using whole periods to shift the pulse is that the motor drivers require the pulse to be held high and low for at least 1 μs each [All12]. The default clock rate for JOP on the Cyclone II FPGA is 60 MHz. It was therefore initially unknown if the timing requirements would be met if the pulse was completed in a single period. Utilizing JOP's WCET analysis tool it has become evident that even in the fastest case the RepRapController uses enough cycles to meet the driver's minimum timing requirements, so this will be implemented in a future version.

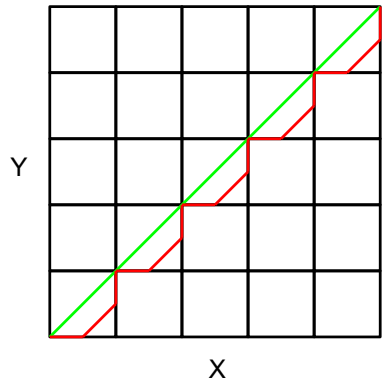


Figure 3.7: Example of unsynchronized stepping of the X and Y actuators resulting in the red line instead of the intended green line

One of the main responsibilities of the RepRapController is calculating when to move the motors. All movements are done in straight lines, but as the motors can only move in steps it is necessary to calculate how to best print a straight line in a grid system. This problem is not new as it also exists in other areas, such as drawing lines on monitors. An often used solution is the Bresenham line algorithm [Bre65], which is used for this project as well. The interesting aspect is that the algorithm is expanded to the RepRap's 4 dimensions (X,Y,Z,E) by using time as the baseline. When a movement command is executed by the CommandController it calls the `setTarget` method in the RepRapController. This method calculates the difference between the current position and the target position for all axes. The 4 dimensional straight line distance is then calculated as $L = \sqrt{\Delta X^2 * \Delta Y^2 * \Delta Z^2 * \Delta E^2}$. Using a given move speed H and the maximum amount of steps (pulses) per minute, the total time (in steps) for the move is $\Delta t = \frac{L * 30000 \frac{steps}{minute}}{H}$. This gives a common baseline for the axes. When the RepRapController executes it uses the standard Bresenham algorithm on each individual axis coupled with the baseline to calculate when a motor should be pulsed, as shown in Figure 3.8. In the example shown the RepRapController executes 10 times (ignoring putting the pulse low) which corresponds to the width of the graph. The X motor is pulsed once almost every other period for a total of 6 times, while the Y motor is only pulsed twice over the same time period.

The heater is controlled by turning it on and off. Since the RepRapController has a fixed period it is possible to utilize pulse-density modulation to set the power output at a certain percentage. This can be used to implement an ad-

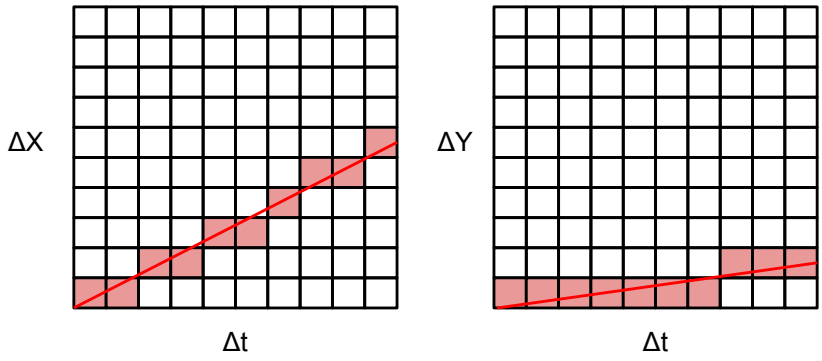


Figure 3.8: Example of Bresenham calculation using the common baseline Δt

vanced temperature regulator, although this idea is abandoned. Instead a bang-bang controller is used for the temperature: If the temperature is lower or higher than the target temperature the heater is turned either on or off respectively, ignoring hysteresis. To measures the temperature the ADC hardware object, described in section 3.4, is used. The data supplied by the ADC is the number of microseconds it last took to charge the capacitor. This is converted into temperature by using a lookup table. The table is constructed by setting the heater at a fixed output and letting the heat level off. The temperature is then measured using a thermometer and put into the table together with the ADC data. Using linear interpolation the values supplied by the ADC can then be converted into temperature. It should be noted that the relationship between the ADC output and the temperature is exponential. Initially the corresponding exponential function was constructed from the table measurements, however there were too many operations necessary to evaluate the function. This led to the current solution with several linear interpolants.

Requests for a RepRap simulator were given both as feedback to the article and at JTRES 2012, so that the SCJ firmware could be executed on platforms other than JOP and independently of the RepRap. This is because hardware objects are not part of SCJ. New SCJ compliant hardware access methods were presented at JTRES 2012 [Hun12], but these are currently not implemented in JOP and therefore not viable for the firmware. A simple simulator has therefore been constructed that does not use hardware objects, allowing the firmware and simulator to act as a general SCJ test case. The use of hardware objects for the JOP specific solution has actually made the simulator implementation simpler.

Instead of creating hardware objects, two simulator objects are created that simulate the serial line and RepRap. They simulate and maintain the hardware state and have the same interface as the corresponding hardware objects. Activating simulation is simply a matter of removing the hardware objects and adding the simulator objects with the same variable name. Currently the simulator only executes a few extrusion movements and repeats, however it will be expanded with other G-codes so that the capabilities of the firmware and SCJ platform are more thoroughly tested.

The SCJ RepRap implementation is open source and hosted at `https://github.com/torurstrom/jop` as a fork of JOP. The basic application, as well as the simulator components, are according to the current SCJ specification and not dependent on the JOP platform. However, access to real I/O devices is always platform dependent, so using the firmware with the RepRap requires JOP and hardware objects. The firmware and simulator code is located in `https://github.com/torurstrom/jop/tree/master/java/target/src/rtapi/org/reprap`.

Chapter 4

Evaluation

In this chapter the various evaluations are presented. Section 4.1 describes the programming experience and Section 4.2 considers schedulability and WCET analysis. A performance comparison of the safety-critical Java firmware and an optimized C firmware is done in Section 4.3. Section 4.4 argues the functionality of the implementation.

4.1 SCJ Programming Experience

From a programmer's perspective there can be several reasons to want to use Java technology for SC applications, e.g. avoid/reduce memory and timing management, comfortable with Java, etc. SCJ aims to bring Java to safety-critical systems. However, SCJ restricts Java in several areas, so the question is whether the difference between SCJ and Java alienates Java, and other, developers.

Being used to Java threads, programming with PEHs is similar, since the application can be functionally distributed across PEHs in the same manner as threads. One difference is that PEHs are created at mission start-up and are periodic, which might be a problem where one would create threads on the fly in Java, such as when processing large data sets. However, this is not necessarily a problem for safety-critical applications. In Java, to create a periodic task,

```
1  @Override
2  public void run()
3  {
4          boolean loop = true;
5          while(loop)
6          {
7                  long time = System.currentTimeMillis()+10;
8                  /*
9                   * Do work
10                  */
11                 try
12                 {
13                         wait(time-System.currentTimeMillis());
14                 }
15                 catch (InterruptedException e)
16                 {
17                         e.printStackTrace();
18                 }
19         }
20 }
```

Figure 4.1: Periodic Java thread

one would typically create a thread and override the `run` method, as shown in Figure 4.1. In SCJ this done by overriding the `handleAsyncEvent` method, as shown in Figure 4.2.

The storage parameters are troublesome when creating a PEH. It is possible to count the number of objects and primitives used in a memory scope, however the size of the objects is platform dependent, so unless the programmer has a thorough knowledge of the platform, the correct values for the parameters are not clear. This also applies when creating safelets, mission sequencers and missions. During development the storage parameters were set to some initial value and in the case of memory shortage simply increased. Although this works for non-critical applications it is not viable for true SC systems where the maximum memory usage should be guaranteed and bounded. Development would be made easier if vendors supplied a tool to statically analyse the maximum memory usage of the application and its parts.

Java programmers are used to the JVM handling memory management, with a GC taking care of object deallocation. In SCJ there is no GC. Instead objects are created in memory scopes and deallocated when the scope is exited. Since each PEH has a private scope, this means that all objects created in one execution of the PEH are deallocated when the PEH is released. It is therefore not a problem to create temporary objects during execution, similarly to programming with a GC. However, the objects cannot be referenced in outer scopes, since their

```
1  new PeriodicEventHandler(new PriorityParameters(1),
2          new PeriodicParameters(null, new RelativeTime(10,0)),
3          new StorageParameters(50, null, 0, 0), 40)
4  {
5
6          @Override
7          public void handleAsyncEvent()
8          {
9                  /*
10                  * Do work
11                  */
12         }
13 };
```

Figure 4.2: PeriodicEventHandler

existence is not guaranteed. This is a major difference to Java, where the programmer can freely pass references between threads, and essentially modifies the Java semantics, which might not be well received by Java programmers. If one PEH generates a result object that is needed in another PEH, the primitive values of the object have to be copied to a shared object in either MM or IM. This is why the SCJ RepRap firmware uses object pools in MM, allowing different PEHs to reference the same command objects.

Initially the command object pools were created in IM, however the PEHs have to be created in MM and some command objects need references to some of the PEHs residing in IM, which results in illegal referencing. Using IM can therefore be problematic. It is also not evident when an object is initialized in IM, e.g. is it only before `Mission.initialize`? Objects that should reside in MM are created in `Mission.initialize`. This solution could also be done for Safelet with a `Safelet.initialize` method, ensuring that objects created there reside in IM.

The referencing problem also affects the use of library code which creates new objects. For example `StringBuilder` was initially used for the HostController. A `StringBuilder` automatically creates a new array in its `append` method if the buffer is full. If the `StringBuilder` is created in MM and `append` called from the HostController's `handleAsyncEvent` method, the creation of a new buffer results in an illegal reference. If the `StringBuilder` is created in `handleAsyncEvent`, it does not live after the HostController's release. It is still possible to use the `StringBuilder` by not overflowing the buffer, however any `String` created from it would live in the HostController's PM and therefore not be referenceable by the CommandParser. This is in contrast with the normal programming flow of the `StringBuilder` where characters are appended as necessary, and a `String` is derived from it and freely referenced for further processing. The absence of a

PEH	Priority	Period (ms)	WCET (ms)	Max. blocked (ms)
RepRapController	4	1	0,0718667	0,0016
HostController	3	1	0,42593	0,1529833
CommandController	2	20	0,9138333	0,1529833
CommandParser	1	20	3,5771167	0,1529833

Table 4.1: WCET for the PeriodicEventHandlers

GC therefore requires more effort from the programmer and changes the normal usage of library code.

During development faulty references were captured using an on-line scope checker on JOP. This has helped identify the problems with references in SCJ, however it can be cumbersome to execute the application and reach a point where an illegal reference is made. Optimally any wrong references should be caught with static analysis, such as the tool described in [DHS12], especially if it can be integrated into the IDE.

4.2 Schedulability

In SCJ it is the programmer's responsibility to ensure schedulability. Similar to the problem with the storage parameters, the programmer cannot be sure if a PEH's priority and periodic parameters present a feasible schedule until the WCET of each PEH is found. Since the execution time of each line of code is platform dependent, the analysis must cover the application, the framework and the platform. JOP's WCET analysis tool is used for this task [SPPH10]. As it is the `handleAsyncEvent` method that is called at each PEH's execution, the WCET tool is used on this method for each PEH. The results are shown in Table 4.1. The priorities are set inversely proportional to the period and WCET, i.e. the RepRapController has the lowest period and WCET, and therefore the highest priority. Since SCJ uses fixed priority scheduling, the priority allocation essentially turns it into rate-monotonic scheduling.

The tool presents the results in cycles, which are converted to execution time (in ms) when the clock frequency (60 MHz in our case) is known. The tool is not able to include the maximum time a PEH can be blocked due to a lower priority thread taking the same resource. This is found by manually tracing a PEH's execution and finding each synchronization lock. Each other block of code that uses the same lock and is called by another PEH is analysed with the

WCET tool. The largest one is added to the table in the last column.

To check if the PEHs are schedulable, the test from [SRL90] is used in the following calculations, which takes into account potential blocking times:

$$\forall i, 1 \leq i \leq n, \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1)$$

Here n is the number of tasks. For task i the C_i, T_i and B_i are the WCET, period and maximum blocking time respectively.

RepRapController

$$\frac{0{,}0718667}{1} + \frac{0{,}0016}{1} \leq 1 * (2^{\frac{1}{1}} - 1) \Leftrightarrow 0,0734667 \leq 1$$

HostController

$$\frac{0{,}0718667}{1} + \frac{0{,}42593}{1} + \frac{0{,}1529833}{1} \leq 2 * (2^{\frac{1}{2}} - 1) \Leftrightarrow 0,65078 \leq 0,8284271$$

CommandController

$$\frac{0{,}0718667}{1} + \frac{0{,}42593}{1} + \frac{0{,}9138333}{20} + \frac{0{,}1529833}{20} \leq 3 * (2^{\frac{1}{3}} - 1) \Leftrightarrow 0,55113753 \leq 0,7797631$$

CommandParser

$$\frac{0{,}0718667}{1} + \frac{0{,}42593}{1} + \frac{0{,}9138333}{20} + \frac{3{,}5771167}{20} + \frac{0{,}1529833}{20} \leq 4 * (2^{\frac{1}{4}} - 1) \Leftrightarrow 0,729993365 \leq 0,7568285$$

All inequalities are satisfied so the PEHs are schedulable. Note that in this analysis PEH switching times are not included, as context switching code is not reachable from the `handleAsyncEvent` methods. Analysing the system's entry method should result in a full system analysis, however this is not currently possible on JOP and requires a lot of changes to framework code. To achieve the current analysis it was necessary to avoid most framework libraries, such as `String`, since these were not analysable or resulted in WCETs that were far too high.

To produce an analysable application it is necessary to program while keeping in mind schedulability, e.g. the time PEH1 blocks PEH2 is relevant to PEH2's schedulability test, which is why blocking times must be diminished. Figure 4.3 contains an excerpt of the HostController which shows a design with this intent. Instead of locking the entire `handleAsyncEvent` method, only a small part is synchronized with the `getInputStatus` and `setInputStatus` methods. Although the locking could be done with synchronization blocks, thereby avoiding the overhead of method calls, they are not available according to the SCJ

```
1  synchronized private void setInputStatus(boolean status)
2  {
3          inputStatus = status;
4  }
5
6  synchronized private boolean getInputStatus()
7  {
8          return inputStatus;
9  }
10
11 @Override
12 public void handleAsyncEvent()
13 {
14         char[] output = outputBuffer.getChars(16);
15         for (int i = 0; i < output.length; i++) //@WCA loop = 16
16         {
17                 SP.write(output[i]);
18         }
19         //Input buffer is still full so do nothing
20         if(getInputStatus())
21         {
22                 return;
23         }
24         for (int i = 0; i < 16; i++) //@WCA loop = 16
25         {
26                 char character;
27                 if(!SP.rxFull())
28                 {
29                         //No input
30                         return;
31                 }
32                 character = (char)SP.read();
33                 if(character == ';')
34                 {
35                         comment = true;
36                 }
37                 else if(character == '\n')
38                 {
39                         comment = false;
40                         if(inputCount > 0)
41                         {
42                                 setInputStatus(true);
43                                 return;
44                         }
45                 }
46                 else if(!comment) //Ignore comments
47                 {
48                         if(inputBuffer.add(character))
49                         {
50                                 inputCount++;
51                         }
52                 }
53         }
54 }
```

Figure 4.3: Excerpt of `HostController.java`

	SCJ firmware	Teacup
Firmware size (KB)	84	∼32
Maximum steps per second	500 @ 60 MHz	15570 @ 20 MHz

Table 4.2: Firmware performance costs

specification. It is not evident why this is so. It is therefore recommended that the SCJ specification re-introduces synchronization blocks.

The `//@WCA loop=16` line acts as an annotation for the WCET tool and indicates that the loop will run a maximum of 16 times. This annotation is necessary for most non-trivial loops, as the tool is otherwise unable to determine the execution time. This is relevant to system classes such as `String` that are designed for strings of almost arbitrary size. As the tool is not able to see if an application only uses strings with a fixed length, the maximum length must be manually added to the library loops, such as in `String.substring`. Another problem is `while` loops which, from the tool's perspective, are unbounded. It is therefore necessary to annotate or avoid them. This becomes especially troublesome when the framework itself uses them for blocking reads/writes, e.g. `System.in.read()`. Reading and writing to streams needs to be organized such that a PEH can check availability before reading/writing. If characters on the stream are not available, the PEH can be released allowing other PEHs to execute.

4.3 Safety-Critical Java vs. C

There are several RepRap firmwares written in either C or C++ for microcontrollers. One of them is the Teacup firmware, a rewrite of the original FiveD firmware, written entirely in C [Rep12g]. It does not use any real-time framework, so there are no guarantees for execution times. The application handles the timing itself, such as setting up and deactivating timer interrupts. The SCJ firmware lets the framework handle timing, which seems simpler for the programmer. The two firmwares use two different programming languages, but basic advantages/disadvantages in using object oriented languages will not be discussed here. Instead the performance costs of using SCJ based firmware, as opposed to using a minimalistic firmware such as Teacup, are highlighted in Table 4.2.

The firmware size includes all framework and application code. The size difference is not as bad as expected, especially considering that the SCJ firmware

size can be further reduced by using an optimization tool that removes unused methods from classes. The steps indicate how many steps the firmware can move the motors per second. The frequency shows the frequency of the CPU. Teacup has a clear advantage even at a lower frequency. The SCJ firmware steps are based on the period of the RepRapController, which is limited by the framework. However, as the previous schedulability test shows, the period cannot be decreased without severely affecting schedulability. In the current state the best Teacup stepping performance is roughly 30 times better than the SCJ firmware performance.

One way to improve the SCJ performance is to increase the number of cores on JOP, thereby negating the affect the RepRapController's WCET has on the other PEHs. Ignoring the current framework limit, this will allow the RepRapController to execute with a period of 0.1 ms, reducing the performance gap by a factor of 10. This is not as interesting from a schedulability perspective, since the PEHs would simply run in parallel.

Another solution is to delegate some of the RepRapController's responsibilities to hardware components, such as writing the stepping controls in VHDL. The simple, but numerous, Bresenham error calculations and incrementations could be handled by hardware, thereby significantly increasing the stepping performance and even greatly outperforming the Teacup firmware. The more advanced and slow operations, such as calculating the straight line distance, could still be done by the SCJ firmware. This solution seems to fit better with SCJ, as it has taken quite a lot of effort to reduce the execution time of the HostController and RepRapController to do the lower level work required by the motor drivers and serial line. SCJ might simply be better suited for work at a higher abstraction level.

At JTRES 2012 it was mentioned that the stepping performances are incomparable, since the SCJ RepRap firmware is limited by the WCET and therefore can guarantee the stepping time. The Teacup performance is the best case but without any guarantees. According to some of the JTRES 2012 attendees the performance difference can be as high as a factor 100 between the best case execution time and the WCET for C applications. Whilst it is true that a big performance gap is to be expected between a guaranteed performance and the best case, the default stepping speed for other RepRap machines is still 10 times higher than for the SCJ firmware. The observation regarding the use of SCJ for primitive tasks therefore still stands.

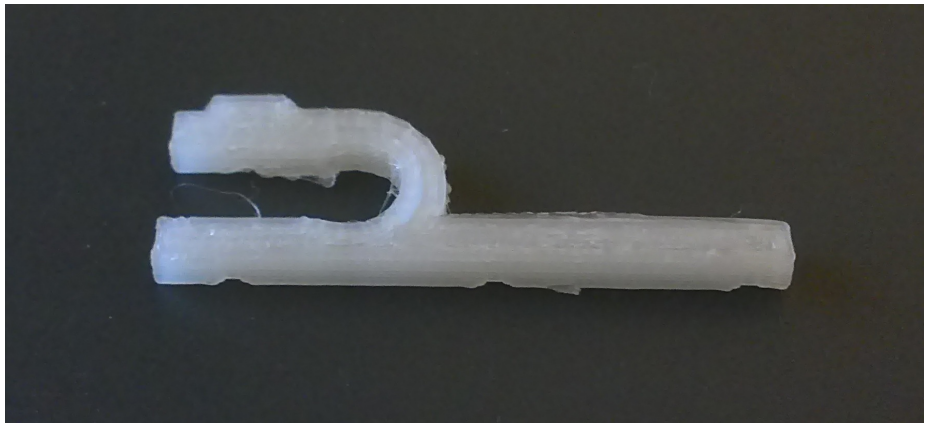


Figure 4.4: A RepRap end-stop produced at the JTRES 2012 presentation

4.4 SCJ RepRap Test

Several points have been presented that criticize SCJ and the implementation difficulties that arise when using it, however the result of the project is still a functioning RepRap implemented as SCJ level 1 application. Figure 4.5 shows the full RepRap setup excluding the host. This setup (including a host) was used at the JTRES 2012 presentation for a live demonstration. The printer produced the item shown in Figure 4.4 during the demonstration, although it did not finish it because of time limitations. It is a mount for an optical end-stop used in the printer. The RepRap has thereby started the process of replicating itself, confirming the usability of SCJ.

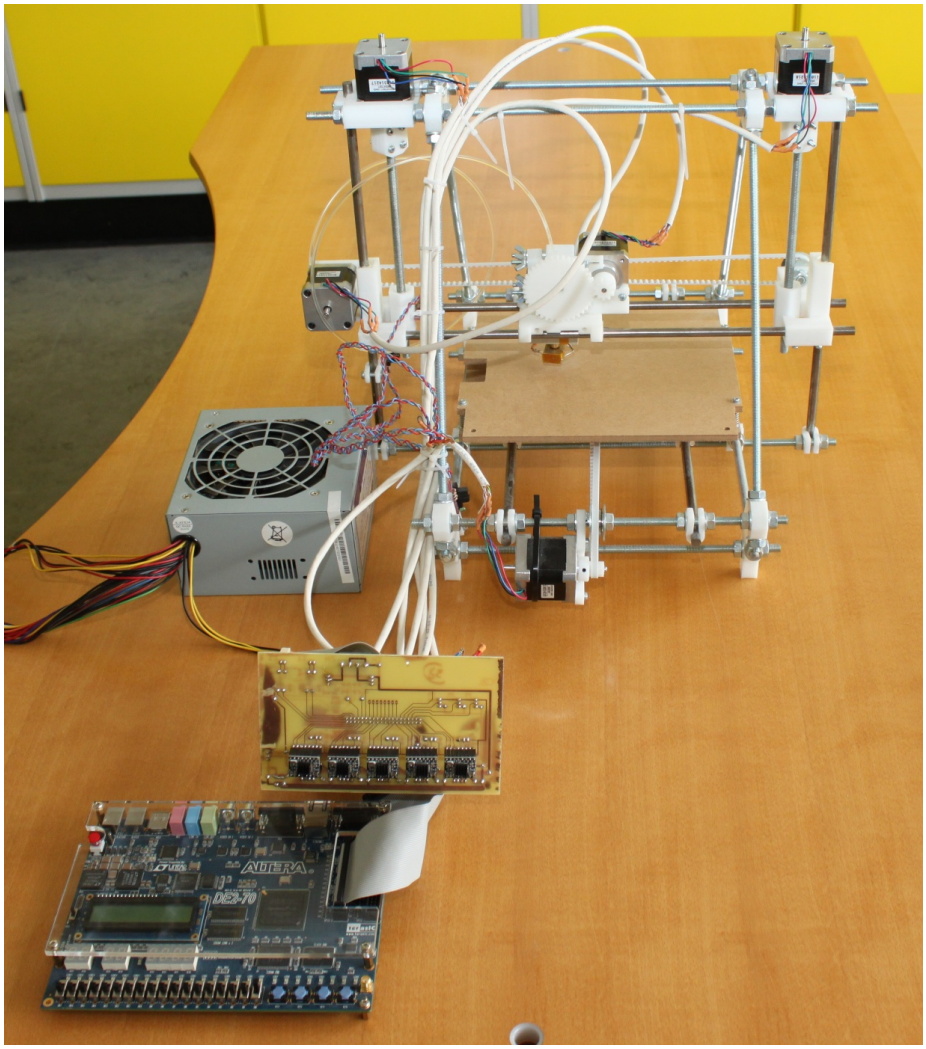


Figure 4.5: SCJ RepRap setup without host

Chapter 5

Conclusion

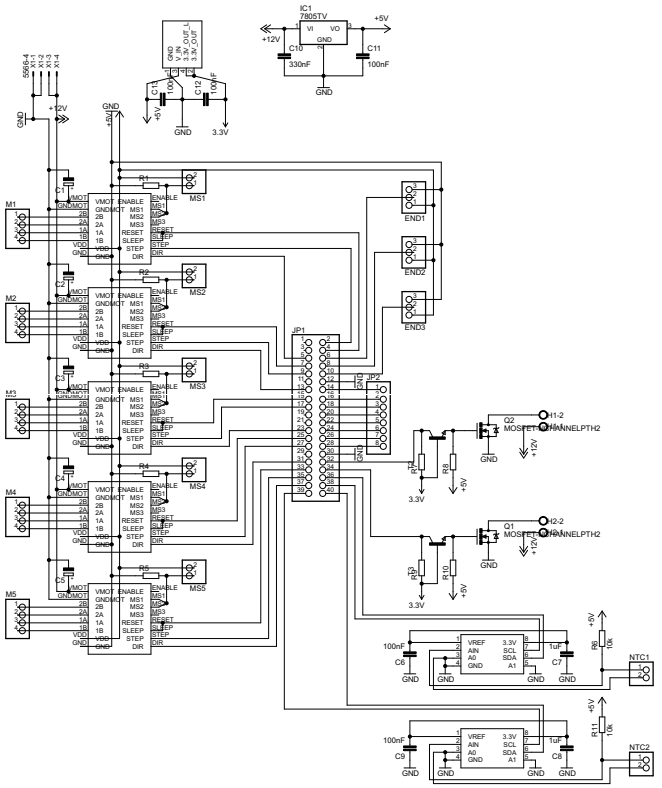
The process of creating a RepRap 3D desktop printer in SCJ has required knowledge from several fields. The RepRap firmware has been created from scratch as a SCJ level 1 application and runs on top of JOP's SCJ implementation. As a part of the project the RepRap's mechanical parts have been assembled. Additionally an interface has been designed and constructed as a printed circuit board to facilitate the communication between the firmware and the RepRap's sensors and actuators. The SCJ RepRap is functioning and able to print 3D objects in plastic, including parts of itself. To allow the firmware to function as a general SCJ test case an initial version of a RepRap simulator has been implemented. The simulator allows the firmware to execute on SCJ platforms other than JOP's and independently of the RepRap hardware. Another part of the project as been the creation of a published paper on the subject that was presented at JTRES 2012

The RepRap implementation reveals several points of interest in SCJ that affect both vendors and developers. Tools should be available to analyse WCET and maximum memory usage of the applications. Platforms, frameworks and libraries must be modified so that the tools are able to perform the analysis. This means that code should not use unbounded loops or otherwise block PeriodicEventHandlers indefinitely. Java programmers will have familiarity with SCJ but must learn to code with more responsibility. The lack of garbage-collection changes the usual Java semantics and requires that programmers have

to be careful where created objects are referenced. Programmers must also have a deeper knowledge of the library code to ensure that objects aren't created and wrongly referenced during execution. This problem can be lessened if vendors supply static analysis tools that verify references, avoiding run-time verification. The SCJ implemented firmware has a larger execution overhead than a C based firmware, which could indicate that SCJ applications are better suited for computing at higher abstraction levels. Despite the problems encountered the result of the project is a functioning SCJ Java level 1 RepRap, showing that SCJ is indeed usable.

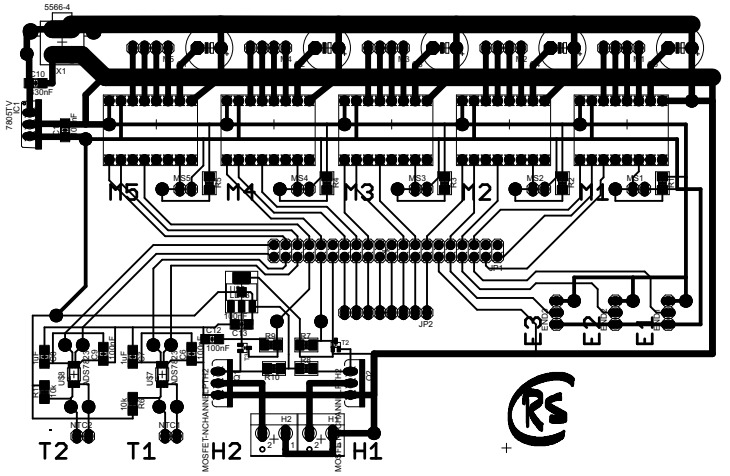
Appendix A

Interface Board Schematic


IC1
7805TV
VI
VO
GND
+12V
+5V
C10
330nF
C11
100nF
GND
5566-4
X1-1
X1-2
X1-3
X1-4
GND
+12V
GND
+5V
C1
10nF
C2
10nF
+5V
GND
3.3V
M1
M2
M3
M4
M5
MS1
MS2
MS3
MS4
MS5
R1
R2
R3
R4
R5
C1
C2
C3
C4
C5
VMOT
GND
2B
2A
1A
1B
VDD
ENABLE
MS1
MS2
MS3
RESET
SLEEP
STEP
DIR
END1
END2
END3
JP1
JP2
Q2
MOSFET-CHANNELPTH2
Q1
MOSFET-CHANNELPTH2
H1-2
H2-2
R7
R8
R9
R10
3.3V
+5V
+12V
VREF
AIN
A0
GND
3.3V
SCL
SDA
A1
100nF
C6
C9
1uF
C7
C8
R6
R11
10k
NTC1
NTC2

Appendix B

Interface Board Layout



Bibliography

[All12] Allegro MicroSystems. A4982 DMOS microstepping driver with translator and overcurrent protection. `http://www.allegromicro.com/Products/Motor-Driver-And-Interface-ICs/Bipolar-Stepper-Motor-Drivers/A4982.aspx`, November 2012.

[BDR98] Alan Burns, Brian Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–275. Springer-Verlag, 1998.

[Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25 –30, 1965.

[DHS12] Andreas E. Dalsgaard, René Rydhof Hansen, and Martin Schoeberl. Private memory allocation analysis for safety-critical java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 9–17, New York, NY, USA, 2012. ACM.

[Hun12] James J. Hunt. A new i/o model for the real-time specification for java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 26–33, New York, NY, USA, 2012. ACM.

[JHS+11] Rhys Jones, Patrick Haufe, Edward Sells, Pejman Iravani, Vik Olliver, Chris Palmer, and Adrian Bowyer. Reprap - the replicating rapid prototyper. *Robotica*, 29(Special Issue 01):177–191, 2011.

[JTR12] JTRES2012. The jtres 2012 website. `http://jtres2012.imm.dtu.dk/index.html`, October 2012.

[Kli12] Kliment. Printrun. `http://reprap.org/wiki/Printrun`, November 2012.

[LAB+12] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. Safety-critical Java technology specification, public draft version 0.90, 2012.

[PZS+10] Ales Plsek, Lei Zhao, Veysel H. Sahin, Daniel Tang, Tomas Kalibera, and Jan Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 95–101, New York, NY, USA, 2010. ACM.

[Rep12a] RepRap Project. Build instructions. `http://www.reprap.org/wiki/Prusa_Mendel`, July 2012.

[Rep12b] RepRap Project. G-codes. `http://www.reprap.org/wiki/G-code`, November 2012.

[Rep12c] RepRap Project. Polulu motor driver. `http://www.reprap.org/wiki/Pololu_stepper_driver_board`, November 2012.

[Rep12d] RepRap Project. Polylactic acid. `http://reprap.org/wiki/PLA`, July 2012.

[Rep12e] RepRap Project. The reprap project website. `http://reprap.org/wiki/Main_Page`, July 2012.

[Rep12f] RepRap Project. Stepstick motor driver. `http://www.reprap.org/wiki/StepStick`, November 2012.

[Rep12g] RepRap Project. Teacup firmware. `http://reprap.org/wiki/Teacup_Firmware`, July 2012.

[Sch04] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[Sch07] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[Sch08] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[SJL+09] Tobias Schoofs, Eric Jenn, Stéphane Leriche, Kelvin Nilsen, Ludovic Gauthier, and Marc Richard-Foy. Use of perc pico in the aida avionics platform. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 169–178, New York, NY, USA, 2009. ACM.

[SKKR11] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, November 2011.

[SKR12] Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 44–53, New York, NY, USA, 2012. ACM.

[SPPH10] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[SR12] Martin Schoeberl and Juan Ricardo Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 54–61, New York, NY, USA, 2012. ACM.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175 –1185, sep 1990.

[SS12] Tórur Biskopstø Strøm and Martin Schoeberl. A desktop 3d printer in safety-critical java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 72–79, New York, NY, USA, 2012. ACM.

[SWC12] Neeraj Kumar Singh, Andy Wellings, and Ana Cavalcanti. The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[Tel12] TeleFox. Low-cost ADC using only Digital I/O. `http://letsmakerobots.com/node/13843`, November 2012.