BAKKALAUREATSARBEIT

qsimavr Graphical Simulation of an AVR Processor and Periphery

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Technischen Informatik

unter der Leitung von

Alexander Kössler
Institut für Technische Informatik 182

durchgeführt von

Jakob Gruber
Matr.-Nr. 0203440
Kirschenallee 6/1, A-2120 Obersdorf

Wien, im September 2012 ....

qsimavr
Graphical Simulation of an AVR Processor and Periphery

Simulation of AVR programs offers substantial advantages, most notably zero costs, no hardware requirements, and easy debugging with familiar tools. For this thesis, a new graphical, extensible AVR simulation frontend based on *simavr* and Qt has been developed in C++. This frontend, called *qsimavr*, provides simulation of commonly used IO components such as the GLCD, LCD, Temperature Sensor, EEPROM, RTC, LEDs and buttons. *qsimavr* is now able to execute a large majority of the tasks and exercises of the Technical University (TU) Vienna's Microcontroller lab course. New components can be easily added through a plugin interface as needed.

Contents

1. Introduction

Developing complex applications can be an extremely difficult task, even more so when the targeted platform is as close to the hardware as microcontrollers. Unlike typical computer programs, programming for a microcontroller is done at a very low level, requiring knowledge of the exact hardware specifications and using strict protocols to correctly communicate with external periphery. These protocols often require sending signals with specific timing bounds (these are often in the micro- and nanosecond range), and upon failure to do so proceed with undefined behavior instead of printing a (hopefully) informative error message or throwing a well-defined exception.

Debugging on the other hand is complicated by the fact that programs are executed not on the development host, but on a separate piece of hardware. There are several options available to developers: for example, one could use `printf` to output text over a Universal Asynchronous Receiver/Transmitter (UART) interface, or toggle Light-Emitting Diodes (LEDs) at specific points of interest within the program. While debuggers may be used by connecting a Joint Test Action Group (JTAG) device such as the Atmel JTAGICE mkII, these are often expensive[1] and can introduce side effects of their own.

The answer to these problems lies in simulation. Binary Executable and Linkable Format (ELF) files can be interpreted by a simulator to mimic the state and actions of an Alf and Vegard's Risc processor (AVR) controller executing an ELF firmware file. This program can not only run directly on the development host, but also implement sophisticated development aids such as debugging support using standard tools, producing execution and signal traces, and replaying recorded signal sequences. Simulation simplifies development of AVR programmers and improves the quality of produced artifacts while simultaneously lowering the entry barrier for new programmers by allowing them to debug without acquiring expensive hardware (or even doing initial development without possessing a hardware microcontroller at all). Unit testing and regression checks can be created that run automatically on the development machine, and environments may be simulated by recording and then replaying signal sequences at will.

[1] And specifically, not available to students in the Technical University (TU) Vienna Microcontroller course.

The aim of this thesis is to produce such a tool for AVR microntrollers, specifically the `atmega1280` processor running on a frequency of 16 Megahertz (MHz) on a MikroElektronika BIGAVR6 Development System[2], based on the AVR core simulation provided by *simavr*[3]. External periphery of the BIGAVR6 board will also be simulated, including the Liquid Crystal Display (LCD) and Graphical Liquid Crystal Display (GLCD) displays, the Two-Wire Interface (TWI) Electrically Erasable Programmable Read-Only Memory (EEPROM) and Real-Time Clock (RTC), on-board LEDs and push buttons connected to the Input/Output (IO) ports, a touchscreen attached to the GLCD, and a temperature sensor using the 1-wire protocol.

This document is structured as follows: Chapter 2 includes a short summary of the concept of simulations, and computer simulations in particular. Chapter 3 summarizes the current state of related AVR simulation software. The internals of *simavr* are examined in-depth in chapter 4. Chapter 5 discusses the project requirements, and how the chosen design approach will achieve these as well as important decisions made during the development process. Chapter 6 walks through the implementation of the *qsimavr* core as well as the provided components. Results and potential for further work are presented in chapter 7. Chapter 8 contains the conclusion of this thesis.

Appendix A provides a setup guide for getting up and running with both *simavr* and *qsimavr*. Finally, appendix B contains a *qsimavr* user guide and brief tutorial for debugging AVR programs with *qsimavr* and GNU Debugger (GDB).

[2] `http://www.mikroe.com/bigavr/`, accessed 2012-09-01.
[3] `https://github.com/buserror-uk/simavr`, accessed 2012-09-01.

2. Concepts

Simulation is the imitation of a real-world process or system over time[?]. There are numerous variations of this concept, including simulation for training purposes (such as aircraft simulators), simulation of natural phenomenons for scientific study or for creating forecast (like weather systems), and simulation simply for entertainment purposes as can be found most prominently in computer games.

This thesis concerns itself with yet another kind of simulation, which imitates the operation of a microcontroller and attached peripherals on common desktop computer architectures. More specifically, *simavr* simulates the family of 8-bit AVR microcontroller systems on any modern system supported by *avr-libc*.

In this case, simulation simplifies both the development and debugging of AVR programs by making software engineering staples such as automatic unit tests possible and vastly easing access to debugging methods, and tools such as GDB without further required hardware.

Emulation is another related concept which is only indistinctly different from simulation. Originally, emulation was first used in 1963 to refer to a type of simulation which consisted of mixing hardware microcode and software components to achieve faster simulation speed[?]. However, this definition is now largely obsolete. The current concensus seems to be that emulation focuses on imitating the tangible effects of a system, while simulation concerns itself especially with a system's inner state.

In this context, *simavr* can be considered to be a mixture between a simulator and emulator; it tries to both maintain approximately realistic execution times while producing identical execution results as when running on an AVR Microcontroller (MCU) as well as maintaining an accurate internal state at all times.

Emulators usually consist of a Central Processing Unit (CPU) and memory subsystems which interact with IO device simulations[?].

The CPU subsystem is responsible for imitating the state and operation of a processor. Specifically, essential components of an AVR processor are simulation of the Status Register (SREG) and instruction decoding/processing. During each simulation cycle, the subsystem decodes an instruction from program memory and subsequently "executes" it by performing semantically equivalent operations on the host CPU. Finally, the SREG is updated accordingly.

The memory subsystem not only keeps track of memory contents, but also limits access to the available memory space, and handles IO mapped registers which trigger interactions with some IO device when read or written.

Finally, IO devices may be connected to the core subsystems and be programmed to respond to (and emit) signals.

simavr already contains the CPU and memory subsystems, as well as some IO devices closely associated with AVR MCUs such as an internal EEPROM, Analog-Digital Converters (ADCs), IO ports, timers, and more. *qsimavr* will extend upon this existing functionality to implement several further IO modules which interact seamlessly with *simavr*, while also offering a graphical user interface to control the simulation.

3. Related Work

Besides *simavr*, there are currently several other AVR simulator applications available.

3.1. Atmel Studio

Atmel's AVR Integrated Development Environment (IDE), *Atmel Studio*[4], which is based on Microsoft Visual studio, includes tools for AVR simulation and debugging. According to Atmel, "Simulation supports debug commands such as Run, Break, Reset, Single Step, Set Breakpoints, and Watch Variables. The I/O, memory and register views are fully functional using the simulator. [...] With the debuggers connected, Atmel Studio 6 can present the status of the processor, memories and all communication and analog interfaces in views that are easy to understand, giving you fast access to critical system parameters." [?] However, *Atmel Studio* is only available on the Windows operating system.

3.2. simulavr

simulavr[5] is an open source simulator for AVR microcontrollers. It is written in C++ and provides a feature set very similar to *simavr* itself, including ELF file loading, Value Change Dump (VCD) traces, GDB support, and execution traces. Additionally, it includes bindings for Tcl/Tk and Python which can be used for simple frontend additions and writing automated unit tests. *simulavr* also offers execution traces including all debugging information present in the ELF file:

```
fred.elf 0x0038: __do_copy_data+0x9 CPI R26, 0xb8 SREG=[------Z-]
fred.elf 0x003a: __do_copy_data+0xa CPC R27, R17 SREG=[------Z-]
```

[4] `http://www.atmel.com/microsite/atmel_studio6/default.aspx`, accessed 2012-08-31.

[5] `http://www.nongnu.org/simulavr/`, accessed 2012-08-31.

```
fred.elf 0x003c: __SP_L__           BRNE ->0xfff4 main+0xf
fred.elf 0x003e: __SP_H__,__SREG__  RCALL 5a SP=0x25e 0x20
                                    SP=0x25d 0x0
fred.elf 0x003e: __SP_H__,__SREG__  CPU-waitstate
fred.elf 0x003e: __SP_H__,__SREG__  CPU-waitstate
fred.elf 0x005a: main               PUSH R28 SP=0x25c 0x5f
fred.elf 0x005a: main               CPU-waitstate
fred.elf 0x005c: main+0x1           PUSH R29 SP=0x25b 0x2
fred.elf 0x005c: main+0x1           CPU-waitstate
```

Unfortunately, there are still only few AVR cores supported by *simulavr*[6]; the `atmega1280` is not one of them, making *simulavr* initially useless for our purposes. This may be because writing core definitions seems to be a more difficult task in *simulavr* than *simavr*, requiring knowledge of C++ and *simulavr* internals. Although development appeared to be stagnating for a while, version 1.0 has been released only a few months ago in February 2012.

3.3. Other AVR Simulators

Several other projects simulating external devices on top of an AVR core simulation also exist. While researching for this chapter, we stumbled upon an application called *HAPSIM*[7], which not only includes much of the same simulated components but also seems to have made the same design decisions as *qsimavr* regarding its interface. *HAPSIM* can simulate a HD44780U LCD display, push buttons and LEDs connected to IO ports, a UART terminal, and a 4x4 matrix keypad. *HAPSIM* requires Atmel AVR Studio 4, and as such is only available for Microsoft Windows.

The author of *simavr* has also written a full simulation of a RepRap[8] 3D printer called *simreprap*[9]. It is based on *simavr* (in fact it started out as a *simavr* example), and renders output to Open Graphics Library (OpenGL).

[6] 18 devices are supported in version 1.0.
[7] Or, Helmi's AVR Periphery Simulator. `http://www.helmix.at/hapsim/`, accessed 2012-09-01.
[8] "Humanity's first general-purpose self-replication manufacturing machine.“ `http://www.reprap.org/wiki/RepRap`, accessed 2012-09-01.
[9] `https://github.com/buserror-uk/simreprap`, accessed 2012-09-01.

4. simavr Internals

It is necessary to understand *simavr*'s internals before going on to discuss *qsimavr*'s design.

simavr is a small cross-platform AVR simulator written with simplicity, efficiency and hackability in mind[10]. It is supported on Linux and OS X, but should run on any platform with avr-libc support.

In the following sections, we will take a tour through *simavr* internals[11]. We will begin by examining short (but complete) demonstration application.

4.1. simavr Example Walkthrough

The following program is taken from the board_i2ctest *simavr* example. Minor modifications have been made to focus on the essential section. Error handling is mostly omitted in favor of readability.

```
#include <stdlib.h>
#include <stdio.h>
#include <libgen.h>
#include <pthread.h>

#include "sim_avr.h"
#include "avr_twi.h"
#include "sim_elf.h"
#include "sim_gdb.h"
#include "sim_vcd_file.h"
#include "i2c_eeprom.h"
```

The actual simulation of the external EEPROM component is located in i2c_eeprom.h. We will take a look at the implementation later on.

```
avr_t * avr = NULL;
avr_vcd_t vcd_file;
```

[10] For some more technical principles, *simavr* also tries to avoid heap allocation at runtime and often relies on C99's struct set initialization.

[11] Most, if not all of the code examined in this chapter is taken directly from *simavr*.

```
i2c_eeprom_t ee;
```

`avr` is the main data structure. It encapsulates the entire state of the core simulation, including register, Static Random-Access Memory (SRAM) and flash contents, the CPU state, the current cycle count, callbacks for various tasks, pending interrupts, and more.

`vcd_file` represents the file target for the VCD module. It is used to dump the level changes of desired pins (or Interrupt Requests (IRQs) in general) into a file which can be subsequently viewed using utilities such as *gtkwave*.

`ee` contains the internal state of the simulated external EEPROM.

```
int main(int argc, char *argv[])
{
    elf_firmware_t f;
    elf_read_firmware("atmega1280_i2ctest.axf", &f);
```

The firmware is loaded from the specified file. Note that exactly the same file can be executed on the AVR hardware without changes. MCU and frequency information have been embedded into the binary and are therefore available in `elf_firmware_t`.

```
    avr = avr_make_mcu_by_name(f.mmcu);
    avr_init(avr);
    avr_load_firmware(avr, &f);
```

The `avr_t` instance is then constructed from the core file of the specified MCU and initialized. `avr_load_firmware` copies the firmware into program memory.

```
    i2c_eeprom_init(avr, &ee, 0xa0, 0xfe, NULL, 1024);
    i2c_eeprom_attach(avr, &ee, AVR_IOCTL_TWI_GETIRQ
       (0));
```

`AVR_IOCTL_TWI_GETIRQ` is a macro to retrieve the internal IRQ of the TWI simulation. IRQs are the main method of communication between *simavr* and external components and are also used liberally throughout *simavr* internals. Similar macros exist for other important AVR parts such as the ADC, IO ports, timers, etc.

```
    avr->gdb_port = 1234;
    avr->state = cpu_Stopped;
    avr_gdb_init(avr);
```

This section sets up *simavr*'s GDB infrastructure to listen on port 1234. The CPU is stopped to allow GDB to attach before execution begins.

```
    avr_vcd_init(avr, "gtkwave_output.vcd", &vcd_file,
        100000 /* usec */);
    avr_vcd_add_signal(
        &vcd_file,
        avr_io_getirq(avr, AVR_IOCTL_TWI_GETIRQ(0),
           TWI_IRQ_STATUS),
        8 /* bits */,
        "TWSR");
```

Next, a value change dump output is configured to track changes to the `TWI_IRQ_STATUS` IRQ. The file may then be viewed using the *gtkwave* application.

```
    int state = cpu_Running;
    while ((state != cpu_Done) && (state !=
       cpu_Crashed))
        state = avr_run(avr);

    return 0;
}
```

Finally, we have reached the simple main loop. Each iteration executes one instruction, handles any pending interrupts and cycle timers, and sleeps if possible. As soon as execution completes or crashes, simulation stops and we exit the program.

We will now examine the relevant parts of the `i2c_eeprom` implementation. Details have been omitted and only communication with the `avr_t` instance are shown.

```
static const char * _ee_irq_names[2] = {
                [TWI_IRQ_MISO] = "8>eeprom.out",
                [TWI_IRQ_MOSI] = "32<eeprom.in",
};

void
i2c_eeprom_init(
                struct avr_t * avr,
                i2c_eeprom_t * p,
                uint8_t addr,
                uint8_t mask,
                uint8_t * data,
                size_t size)
{
    /* [...] */

        p->irq = avr_alloc_irq(&avr->irq_pool, 0, 2,
           _ee_irq_names);
        avr_irq_register_notify(p->irq + TWI_IRQ_MOSI,
            i2c_eeprom_in_hook, p);

    /* [...] */
}
```

First, the EEPROM allocates its own private IRQs. The EEPROM implementation does not know or care to which *simavr* IRQs they will be attached. It then attaches a callback function (`i2c_eeprom_in_hook`) to the Master Out, Slave In (MOSI) IRQ. This function will be called whenever a value is written to the IRQ. The pointer to the EEPROM state p is passed to each of these callback function calls.

```
void
i2c_eeprom_attach(
                struct avr_t * avr,
                i2c_eeprom_t * p,
                uint32_t i2c_irq_base )
{
        avr_connect_irq(
                p->irq + TWI_IRQ_MISO,
                avr_io_getirq(avr, i2c_irq_base,
                   TWI_IRQ_MISO));
        avr_connect_irq(
                avr_io_getirq(avr, i2c_irq_base,
                   TWI_IRQ_MOSI),
                p->irq + TWI_IRQ_MOSI );
}
```

The private IRQs are then attached to *simavr*'s internal IRQs. This is called chaining - all messages raised are forwarded to all chained IRQs.

```
static void
i2c_eeprom_in_hook(
                struct avr_irq_t * irq,
                uint32_t value,
                void * param)
{
        i2c_eeprom_t * p = (i2c_eeprom_t*)param;
```

```
    /* [...] */

    avr_raise_irq(p->irq + TWI_IRQ_MISO,
            avr_twi_irq_msg(TWI_COND_ACK, p->selected,
                1));

    /* [...] */
}
```

Finally, we've reached the IRQ callback function. It is responsible for simulating communications between *simavr* (acting as the TWI master) and the EEPROM (as the TWI slave). The EEPROM state which was previously passed to `avr_irq_register_notify` is contained in the `param` variable and cast back to an `i2c_eeprom_t` pointer for further use.

Outgoing messages are sent by raising the internal IRQ. This message is then forwarded to all chained IRQs.

4.2. The Main Loop

We will now take a closer look at the main loop implementation. Each call to `avr_run` triggers the function stored in the run member of the `avr_t` structure (`avr->run`[12]). The two standard implementations are `avr_callback_run_raw` and `avr_callback_run_gdb`, located in sim_avr.c. The essence of both function is identical; since `avr_callback_run_gdb` contains additional logic for GDB handling (network protocol, stepping), we will examine it further and point out any differences to the the raw version. Several comments and irrelevant code sections have been removed.

```
void avr_callback_run_gdb(avr_t * avr)
{
    avr_gdb_processor(avr, avr->state == cpu_Stopped);

    if (avr->state == cpu_Stopped)
        return ;

    int step = avr->state == cpu_Step;
    if (step)
        avr->state = cpu_Running;
```

[12]Whenever `avr` is mentioned in a code section, it is assumed to be the main `avr_t` struct.

This initial section is GDB specific. `avr_gdb_processor` is responsible for handling GDB network communication. It also checks if execution has reached a breakpoint or the end of a step and stops the CPU if it did.

If GDB has transmitted a step command, we need to save the state during the main section of the loop (the CPU "runs" for one instruction) and restore to the `StepDone` state at on completion.

In total, there are eight different states the CPU can enter:

```
enum {
    cpu_Limbo = 0,
    cpu_Stopped,
    cpu_Running,
    cpu_Sleeping,
    cpu_Step,
    cpu_StepDone,
    cpu_Done,
    cpu_Crashed,
};
```

A CPU is `Running` during normal execution. `Stopped` occurs for example when hitting a GDB breakpoint. `Sleeping` is entered whenever the `SLEEP` instruction is processed. As mentioned, `Step` and `StepDone` are related to the GDB stepping process. Execution can terminate either with `Done` or `Crashed` on error. Upon initialization, the CPU is in the `Limbo` state.

```
    avr_flashaddr_t new_pc = avr->pc;

    if (avr->state == cpu_Running) {
        new_pc = avr_run_one(avr);
    }
```

We have now reached the actual execution of the current instruction. If the CPU is currently running, `avr_run_one` decodes the instruction located in flash memory (`avr->flash`) and triggers all necessary actions. This can include setting the CPU state (SLEEP), updating the status register SREG, writing or reading from memory locations, altering the Program Counter (PC), etc ...

Finally, the cycle counter (`avr->cycle`) is updated and the new program counter is returned.

```
    if (avr->sreg[S_I] && !avr->i_shadow)
        avr->interrupts.pending_wait++;
    avr->i_shadow = avr->sreg[S_I];
```

This section ensures that interrupts are not triggered immediately when enabling the interrupt flag in the status register, but with an (additional) delay of one instruction.

```
avr_cycle_count_t sleep = avr_cycle_timer_process(
   avr);
avr->pc = new_pc;
```

Next, all due cycle timers are processed. Cycle timers are one of the most important and heavily used mechanisms in *simavr*. A timer allows scheduling execution of a callback function once a specific count of execution cycles have passed, thus simulating events which occur after a specific amount of time has passed. For example, the `avr_timer` module uses cycle timers to schedule timer interrupts.

The returned estimated sleep time is set to the next pending event cycle (or a hardcoded limit of 1000 cycles if none exist).

```
if (avr->state == cpu_Sleeping) {
    if (!avr->sreg[S_I]) {
        avr->state = cpu_Done;
        return;
    }
    avr->sleep(avr, sleep);
    avr->cycle += 1 + sleep;
}
```

If the CPU is currently sleeping, the time spent is simulated using the callback stored in `avr->sleep`. In GDB mode, the time is used to listen for GDB commands, while the raw version simply calls usleep.

It is worth noting that we have improved the timing behavior by accumulating requested sleep cycles until a minimum of 200 usec has been reached. usleep cannot handle lower sleep times accurately, which caused an unrealistic execution slowdown.

A special case occurs when the CPU is sleeping while interrupts are turned off. In this scenario, there is way of ever waking up. Therefore, execution is halted gracefully.

```
if (avr->state == cpu_Running || avr->state ==
   cpu_Sleeping)
    avr_service_interrupts(avr);
```

Finally, any immediately pending interrupts are handled. The highest priority interrupt (this depends solely on the interrupt vector address) is removed from the pending queue, interrupts are disabled in the status register, and the program counter is set to the interrupt vector.

If the CPU is sleeping, interrupts can be raised by cycle timers.

```
    if (step)
        avr->state = cpu_StepDone;
}
```

Wrapping up, if the current loop iteration was a GDB step, the state is set such that the next iteration will inform GDB and halt the CPU.

4.3. Initialization

4.3.1. `avr_t` Initialization

The `avr_t` struct requires some initialization before it is ready to be used by the main loop as discussed in section 4.2.

`avr_make_mcu_by_name` fills in all details specific to an MCU. This includes settings such as memory sizes, register locations, available components, the default CPU frequency, etc ...

The MCU definitions are located in the `simavr/cores` subdirectory of the *simavr* source tree and are compiled conditionally depending on the the local *avr-libc* support. A complete list of locally supported cores is printed by running *simavr* without any arguments.

On successful completion, it returns a pointer to the `avr_t` struct.

If GDB support is desired, `avr->gdb_port` must be set, and `avr_gdb_init` must be called to create the required data structures, set the `avr->run` and `avr->sleep` callbacks, and listen on the specified port. It is also recommended to initially stop the cpu (`avr->state = cpu_Stopped`) to delay program execution until it is started manually by GDB.

Further settings can now be applied manually (typical candidates are logging and tracing levels).

4.3.2. Firmware

We now have a fully initialized `avr_t` struct and are ready to load code. This is accomplished using `avr_read_firmware`, which uses elfutils to decode the ELF file and read it into an `elf_firmware_t` struct and `avr_load_firmware` to load its contents into the `avr_t` struct.

Besides loading the program code into `avr->flash` (and EEPROM contents into `avr->eeprom`, if available), there are several useful extended features which can be embedded directly into the ELF file.

The target MCU, frequency and voltages can be specified in the ELF file by using the `AVR_MCU` and `AVR_MCU_VOLTAGES` macros provided by `avr_mcu_section.h`:

```
#include "avr_mcu_section.h"
AVR_MCU(16000000 /* Hz */, "atmega1280");
AVR_MCU_VOLTAGES(3300 /* milliVolt */, 3300 /*
   milliVolt */, 3300 /* milliVolt */);
```

VCD traces can be set up automatically. The following code will create an 8-bit trace on the UDR0 register, and a trace masked to display only the UDRE0 bit of the UCSR0A register.

```
const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ =
    {
    { AVR_MCU_VCD_SYMBOL("UDR0"), .what = (void*)&UDR0
       , },
    { AVR_MCU_VCD_SYMBOL("UDRE0"), .mask = (1 << UDRE0
       ), .what = (void*)&UCSR0A, },
};
```

Several predefined commands can be sent from the firmware to *simavr* during program execution. At the time of writing, these include starting and stopping VCD traces, and putting UART0 into loopback mode. An otherwise unused register must be specified to listen for command requests. During execution, writing a command to this register will trigger the associated action within *simavr*.

```
AVR_MCU_SIMAVR_COMMAND(&GPIOR0);

int main() {
    /* [...] */
    GPIOR0 = SIMAVR_CMD_VCD_START_TRACE;
    /* [...] */
}
```

Likewise, a register can be specified for use as a debugging output. All bytes written to this register will be output to the console.

```
AVR_MCU_SIMAVR_CONSOLE(&GPIOR0);

int main() {
    /* [...] */
    const char *s = "Hello␣World\r";
    for (const char *t = s; *t; t++)
        GPIOR0 = *t;
    /* [...] */
}
```

Usually, UART0 is used for this purpose. The simplest debug output can be achieved by binding `stdout` to `UART0` as described by the avr-libc documentation [?], and then using `printf` and similar functions. This alternate console output is provided in case using UART0 is not possible or desired.

4.4. Instruction Processing

We have now covered `avr_t` initialization, the main loop, and loading firmware files. But how are instructions actually decoded and executed? Let's take a look at `avr_run_one`, located in sim_core.

The opcode is reconstructed by retrieving the two bytes located at `avr->flash[avr->pc]`. `avr->pc` points to the Least Significant Byte (LSB), and `avr->pc + 1` to the Most Significant Byte (MSB). Thus, the full opcode is reconstructed with:

```
uint32_t opcode = (avr->flash[avr->pc + 1] << 8) | avr
   ->flash[avr->pc];
```

As we have seen, `avr->pc` represents the byte address in flash memory. Therefore, the next instruction is located at `avr->pc + 2`. This default new program counter may still be altered in the course of processing in case of jumps, branches, calls and larger opcodes such as STS[?].

Note also that the AVR flash addresses are usually represented as word addresses (`avr->pc >> 1`).

Similar to the program counter, the spent cycles are set to a default value of 1.

The instruction and its operands are then extracted from the opcode and processed in a large switch statement. The instructions themselves can be roughly categorized into arithmetic and logic instructions, branch instructions, data transfer instructions, bit and bit-test instructions, and MCU control instructions.

Processing these will involve a number of typical tasks:

- Status register modifications

  The status register is stored in `avr->sreg` as a byte array. Most instructions alter the SREG in some way, and convenience functions such as `get_compare_carry` are used to ease this task. Note that whenever the firmware reads from SREG, it must be reconstructed from `avr->sreg`.

- Reading or writing memory

  `_avr_set_ram` is used to write bytes to a specific address. Accessing an SREG will trigger a reconstruction similar to what has been discussed above. IO register accesses trigger any connected IO callbacks and raise all associated IRQs. If a GDB watchpoint has been hit, the CPU is stopped and a status report is sent to GDB. Data watchpoint support has been added by the author.

- Modifying the program counter

  Jumps, skips, calls, returns and similar instructions alter the program counter. This is achieved by simply setting `new_pc` to an appropriate value. Care must be taken to skip 32 bit instructions correctly.

- Altering MCU state

  Instructions such as SLEEP and BREAK directly alter the state of the simulation.

- Stack operations

  Pushing and popping the stack involve altering the stack pointer in addition to the actual memory access.

Upon conclusion, `avr->cycle` is updated with the actual instruction duration, and the new program counter is returned.

4.5. Interrupts

An interrupt is an asynchronous signal which causes the the CPU to jump to the associated Interrupt Service Routine (ISR) and continue execution there. In the AVR architecture, the interrupt priority is ordered according to its place in the interrupt vector table. When an interrupt is serviced, interrupts are disabled globally.

4.5.1. Data Structures

Let's take a look at how interrupts are represented in *simavr*:

```
typedef struct avr_int_vector_t {
    uint8_t         vector;
    avr_regbit_t    enable;
    avr_regbit_t    raised;
    avr_irq_t       irq;
    uint8_t         pending : 1,
                    trace : 1,
                    raise_sticky : 1;
} avr_int_vector_t;
```

Each interrupt vector has an `avr_int_vector_t`. `vector` is actual vector address, for example `INT0_vect`. `enable` and `raised` specify the IO register index for, respectively, the interrupt enable flag and the interrupt raised bit (again taking `INT0` as an example, enable would point to the `INT0` bit in `EIMSK` , and raised to `INTF0` in `EIFR`. `irq` is raised to 1 when the interrupt is triggered, and to 0 when it is serviced. `pending` equals 1 whenever the interrupt is queued for servicing, and `trace` is used for debugging purposes.

Usually, raised flags are cleared automatically upon interrupt servicing. However, this does not count for all interrupts(notably, `TWINT`). `raise_sticky` was introduced by the author to handle this special case.

Interrupt vector definitions are stored in an `avr_int_table_t`, `avr->interrupts`.

```
typedef struct  avr_int_table_t {
    avr_int_vector_t * vector[64];
    uint8_t         vector_count;
    uint8_t         pending_wait;
    avr_int_vector_t * pending[64];
    uint8_t         pending_w,
                    pending_r;
} avr_int_table_t, *avr_int_table_p;
```

`pending_wait` stores the number of cycles to wait before servicing pending interrupts. This simulates the real interrupt delay that occurs between raising and servicing, and whenever interrupts are enabled (and previously disabled).

`pending` along with `pending_w` and `pending_r` represents a ringbuffer of pending interrupts. Note that servicing an interrupt removes the one with the highest priority.

4.5.2. Raising and Servicing Interrupts

When an interrupt `vector` is raised, `vector->pending` is set, `vector` is added to the `pending` First In, First Out (FIFO) of `avr->interrupts`, and a non-zero pending_wait time is ensured. If the CPU is currently sleeping, it is woken up.

As we've already covered in section 4.2, servicing interrupts is only attempted if the CPU is either running or sleeping. Additionally, interrupts must be enabled globally in SREG, and pending_wait (which is decremented on each avr_service_interrupts call) must have reached zero. The next pending vector with highest priority is then removed from the pending ringbuffer and serviced as follows:

```
if (!avr_regbit_get(avr, vector->enable) || !vector->
   pending) {
     vector->pending = 0;
```

If the specific interrupt is masked or has been cleared, no action occurs.

```
} else {
     _avr_push16(avr, avr->pc >> 1);
     avr->sreg[S_I] = 0;
     avr->pc = vector->vector * avr->vector_size;
     avr_clear_interrupt(avr, vector);
}
```

Otherwise, the current program counter is pushed onto the stack. This illustrates the difference between byte addresses (as used in avr->pc) and word addresses (as expected by the AVR processor). Interrupts are then disabled by clearing the I bit of the status register, and the program counter is set to the ISR vector. Finally, if raise_sticky is 0, the interrupt flag is cleared.

4.6. Cycle Timers

Cycle timers allow scheduling an event after a certain amount of cycles have passed.

```
typedef avr_cycle_count_t (*avr_cycle_timer_t)(
        struct avr_t * avr,
        avr_cycle_count_t when,
        void * param);

void
avr_cycle_timer_register(
        struct avr_t * avr,
        avr_cycle_count_t when,
        avr_cycle_timer_t timer,
        void * param);
```

In `avr_cycle_timer_register`, `when` is the minimum count of cycles that must pass until the `timer` callback is executed (`param` and `when` are passed back to `timer`[13])

Once dispatched, the cycle timer is removed from the list of pending timers. If it returns a nonzero value, it is readded to occur *at or after that cycle has been reached*. It is important to realize that it therefore differs from the `when` argument of `avr_cycle_timer_register`, which expects a relative cycle count (in contrast to the absolute cycle count returned by the callback itself)[14].

The cycle timer system is used during the main loop to determine sleep durations; if there are any pending timers, the sleep callback may sleep until the next timer is scheduled. Otherwise, a default value of 1000 cycles is returned. Besides achieving a runtime behavior similar to execution on a real AVR processor, sleep is important for lowering *simavr* CPU usage whenever possible.

IRQs and interrupts caused by external events (for example, a "touch" event transmitted from the simulated touchscreen component) are and can *not* be taken into account. This means that scheduled sleep times will always be simulated to completion by `avr->sleep`, even if an external event causing CPU wakeup is triggered immediately after going to sleep. Given a situation in which the next scheduled timer is many cycles in the future and the CPU is currently sleeping, the simulation will become extremely unresponsive to external events.

However, in real applications this situation is very unlikely, since manual events (which cannot be scheduled through cycle timers) occur very rarely, and most applications will have at least some cycle timers with a short period.

It is worth remembering though, that cycle timers are the preferred and most accurate method of scheduling interrupts in *simavr*.

4.7. GNU Debugger (GDB) Support

A debugger is incredibly useful during program development. Simple programming mistakes which can be discovered in minutes using GDB can sometimes consume hours to find without it.

[13] *qsimavr* exploits `param` to implement callbacks to class instances by passing the `this` pointer as `param`.

[14] Treating the return value of `avr_cycle_timer_t` as an absolute value and passing the actually scheduled cycle allows for precise handling of recurring timers without drift. A system based on relative cycle counts could not guarantee accuracy, because *simavr* does not guarantee cycle timer execution exactly at the scheduled point in time.

We have covered how to enable GDB support in section 4.3.1, and when GDB handler functions are called during the main loop in section 4.2. In the following, we will explain further the methods *simavr* employs to communicate with GDB and how breakpoints and data watchpoints are implemented. For a short guide to debugging AVR programs with GDB, see section B.2

simavr has a fully featured implementation of the GDB Remote Serial Protocol, which allows it to communicate with *avr-gdb*. A complete reference of the protocol can be obtained from the GDB manual [?]. Essentially, communication boils down to packets of the format `$packet-data#checksum`. The packet data itself consists of a command and its arguments. The syntax of all commands supported by *simavr* is as follows:

```
'?' Indicate the reason the target halted.
'G XX...' Write general registers.
'g' Read general registers.
'p n' Read the value of register n.
'P n...=r...' Write register n with value r.
'm addr,length' Read length bytes of memory starting at address
                addr.
'M addr,length:XX...' Write length bytes of memory starting
                      address addr. XX... is the data.
'c' Continue.
's' Step.
'r' Reset the entire system.
'z type,addr,kind' Delete break and watchpoints.
'Z type,addr,kind' Insert break and watchpoints.
```

Many of these commands expect a reply value. This could be a simple as sending `"OK"` to confirm successful execution, or it could contain the requested data, such as the reply to the `'m'` command. A single reply can chain several data fields. For example, whenever a watchpoint is hit, the reply contains the signal the program received (`0x05` represents the “trap” signal), the SREG, Stack Pointer (SP), and PC values, the type of watchpoint which was hit (either `"awatch"`, `"watch"`, or `"rwatch"`), and the watchpoint address.

The packets themselves are received and sent over an `AF_INET` socket listening on the `avr->gdb_port`.

Both watchpoints and breakpoints are stored within an `avr_gdb_watchpoints_t` struct in `avr->gdb` and are limited to 32 active instances of each. Breakpoints are set at a particular location in flash memory. Whenever the PC reaches that that point, execution is halted, a status report containing a summary of current register values is sent, and control is passed to GDB. This range check takes place in `avr_gdb_processor`, which is called first during each iteration of the `avr_callback_run_gdb` function as we have already discussed in section 4.2.

Watchpoints[15] on the other hand are used to notify the user of accesses to SRAM. GDB uses a fixed offset of `0x800000` to reference locations in SRAM; this offset must be masked out when receiving GDB commands, and added when sending watchpoint status reports. Three types of watchpoints exist: Read watchpoints are triggered by data reads, write watchpoints by writes, and access watchpoints by both. Handling of these is integrated into the `avr_core_watch_write` and `avr_core_watch_read` functions. Whenever applicable watchpoints exist for a data access, execution is halted, and a status report is sent to GDB.

Finally, since program crashes often occur unexpectedly, *simavr* helpfully provides GDB passive mode, which opens a GDB listening socket whenever an exception occurs if the GDB port is specified. It is therefore always a good idea to initialize `avr->gdb_port`, even if you have no intention of using *simavr*'s GDB features!

4.8. Interrupt Requests (IRQs)

The Interrupt Request (IRQ)[16] subsystem provides the message passing mechanism in *simavr*. Let's begin by examining the main IRQ data structures:

```
typedef struct avr_irq_t {
    struct avr_irq_pool_t * pool;
    const char * name;
    uint32_t            irq;
    uint32_t            value;
    uint8_t             flags;
    struct avr_irq_hook_t * hook;
} avr_irq_t;
```

An IRQ consists of an associated IRQ pool, a name (for debugging purposes), an Identifier (ID), its current value, flags, and a list of callback functions. The ID (`irq`) is when a callback function connected to several IRQs needs to determine which specific IRQ has been raised.

The semantics of `value` are not fixed and are specific to each IRQ; for example, `ADC_IRQ_ADC0` treats `value` as milliVolts, while `IOPORT_IRQ_PIN0` expects

[15]Watchpoint support has been added by the author.

[16] Despite the name, IRQs have nothing in particular to do with interrupts; the interrupt system uses IRQs, and IRQs may trigger interrupts, but they are not strictly linked to each other. Many IRQ usages will not involve interrupts at all.

it to equal either 1 (high) or 0 (low). `flags` is a bitmask of several options[17]. `IRQ_FLAG_NOT` flips the polarity of the signal (raising an IRQ with `value` 1 results in a `value` of 0 and vice versa). Setting `IRQ_FLAG_FILTERED` instructs *simavr* to ignore IRQ raises with unchanged values.

`hook` contains a linked list of chained IRQs and `avr_irq_notify_t` callbacks.

```
typedef void (*avr_irq_notify_t)(
        struct avr_irq_t * irq,
        uint32_t value,
        void * param);

void
avr_irq_register_notify(
        avr_irq_t * irq,
        avr_irq_notify_t notify,
        void * param);
```

Callbacks are executed whenever an IRQ is raised (and is not filtered). Chained IRQs are raised whenever the IRQ they are connected to is raised.

As briefly mentioned in section 4.1, module implementations usually structure communication with the *simavr* core by allocating their own private IRQs, which are then connected to the target *simavr* IRQs. Callbacks are registered on private IRQs; likewise, only private IRQs are raised. This ensures maximum flexibility since IRQ connections are defined in one single location. Relevant functions are:

```
avr_irq_t *
avr_alloc_irq(
        avr_irq_pool_t * pool,
        uint32_t base,
        uint32_t count,
        const char ** names /* optional */);

void
avr_irq_register_notify(
        avr_irq_t * irq,
        avr_irq_notify_t notify,
        void * param);

void
```

[17] `IRQ_FLAG_ALLOC` and `IRQ_FLAG_INIT` are of internal interest only and not mentioned further.

```
avr_connect_irq(
        avr_irq_t * src,
        avr_irq_t * dst);

void
avr_raise_irq(
        avr_irq_t * irq,
        uint32_t value);
```

4.9. Input/Output (IO)

The `IO` module consists of two separate, yet complementary parts: on the one hand, a systematic way of defining actions that take place when `IO` registers are accessed, and on the other the `avr_io_t` infrastructure, which provides unified access to module IRQs, reset and deallocation callbacks, and a Input/Output Control (IOCTL) system.

4.9.1. Input/Output (IO) Register Callbacks

We will examine the IO register callback system first. Whenever the *simavr* core reads or writes an IO register during instruction processing (see section 4.4), it first checks if a callback exists for that address. Assuming it does, a write access will result in a call to the write callback instead of setting `avr->data` directly:

```
static inline void _avr_set_r(avr_t * avr, uint8_t r,
   uint8_t v)
{
    /* [...] */
    uint8_t io = AVR_DATA_TO_IO(r);
    if (avr->io[io].w.c)
        avr->io[io].w.c(avr, r, v, avr->io[io].w.param
           );
    else
        avr->data[r] = v;
    if (avr->io[io].irq) {
        avr_raise_irq(avr->io[io].irq +
           AVR_IOMEM_IRQ_ALL, v);
        for (int i = 0; i < 8; i++)
            avr_raise_irq(avr->io[io].irq + i, (v >> i
               ) & 1);
    }
    /* [...] */
}
```

This snippet contains several interesting bits; first of all, we are reminded that IO addresses are offset by `0x20` (these are added by `AVR_DATA_TO_IO`). Next up, we see that write callbacks need to set the `avr->data` value themselves if necessary. Notice also that a custom parameter is passed into the callback, like most other callback systems in *simavr*. Finally, the associated `IOMEM` IRQs are raised; both bitwise and the byte IRQ `AVR_IOMEM_IRQ_ALL`.

Read accesses are very similar, except that (somewhat counter-intuitively), the value returned by the callback is automatically written to `avr->data`.

Access callbacks plus associated `IOMEM` IRQs are stored in the `avr->io` array. `MAX_IOs` is currently set to `279`, enough to handle all used IO registers on AVRs like the `atmega1280`, which go up to an address of `0x136`[18].

```
struct {
    struct avr_irq_t * irq;
    struct {
        void * param;
        avr_io_read_t c;
    } r;
    struct {
        void * param;
        avr_io_write_t c;
    } w;
} io[MAX_IOs];
```

Callbacks are registered using the function duo of `avr_register_io_write` and `avr_register_io_read`. IRQs are created on-demand whenever the `avr_iomem_getirq` function is called.

The included *simavr* modules (implemented in files beginning with the `avr_` prefix) provide many practical examples of IO callback usage; for example, the `avr_timer` module uses IO callbacks to start the timer when a clock source is enabled through the timer registers.

4.9.2. The `avr_io_t` Module

The `avr_io_t` infrastructure provides additional functionality to modules, including reset and deallocation callbacks, central IRQ handling, and a IOCTL function. The full struct reference is provided here for reference:

[18] 279 = 0x136 − 0x20 + 0x01

```
typedef struct avr_io_t {
    struct avr_io_t *   next;
    avr_t *             avr;
    const char *        kind;

    const char ** irq_names;

    uint32_t            irq_ioctl_get;
    int                 irq_count;
    struct avr_irq_t *  irq;

    void (*reset)(struct avr_io_t *io);
    int (*ioctl)(struct avr_io_t *io, uint32_t ctl,
       void *io_param);
    void (*dealloc)(struct avr_io_t *io);
} avr_io_t;
```

Initialization in the `avr_ioport` Module

For a typical way of initializing an `avr_io_t` struct, let's look at the `avr_ioport` module.

```
static const char * irq_names[IOPORT_IRQ_COUNT] = {
    [IOPORT_IRQ_PIN0] = "=pin0",
    [IOPORT_IRQ_PIN1] = "=pin1",
    /* [...] */
    [IOPORT_IRQ_PIN7] = "=pin7",
    [IOPORT_IRQ_PIN_ALL] = "=all",
    [IOPORT_IRQ_DIRECTION_ALL] = ">ddr",
};

static  avr_io_t    _io = {
    .kind = "port",
    .reset = avr_ioport_reset,
    .ioctl = avr_ioport_ioctl,
    .irq_names = irq_names,
};
```

Once again, struct set initialization is used to partially configure a module. Passed in are the reset and IOCTL handlers, a module name (for debugging purposes), and a list of IRQ names. The deallocation handler is not used by the `avr_ioport` module.

```
void avr_ioport_init(avr_t * avr, avr_ioport_t * p)
{
    p->io = _io;

    avr_register_io(avr, &p->io);
    avr_register_vector(avr, &p->pcint);
    avr_io_setirqs(&p->io, AVR_IOCTL_IOPORT_GETIRQ(p->
       name), IOPORT_IRQ_COUNT, NULL);

    avr_register_io_write(avr, p->r_port,
       avr_ioport_write, p);
    avr_register_io_read(avr, p->r_pin,
       avr_ioport_read, p);
    avr_register_io_write(avr, p->r_pin,
       avr_ioport_pin_write, p);
    avr_register_io_write(avr, p->r_ddr,
       avr_ioport_ddr_write, p);
}
```

Moving on to `avr_ioport_init`; the private, partially initialized `avr_io_t` is copied to the `avr_ioport_t`. `io` is the first member of the module struct to facilitate easy simple conversion between `avr_io_t` and `avr_ioport_t` pointers (this is used in the IOCTL function).

`avr_register_io` adds the IO module to the linked list stored in the main `avr_t` instance, which is iterated at AVR reset and deallocation events; it is also used by the IOCTL and to retrieve IRQs.

`avr_io_setirqs` is then called to create the `IOPORT` IRQs. The ID generated by `AVR_IOCTL_IOPORT_GETIRQ` is stored for subsequent use during IRQ retrieval.

The remaining functions called by `avr_ioport_init` have been left in to convey a complete picture of `avr_ioport` initialization. `avr_register_vector` registers the external interrupt vector, and the `avr_register_io_*` functions create access handlers on IO registers as discussed in section 4.9.1.

Implementation Overview

IOCTLs provide a way to trigger arbitrary functionality[19] in modules. Whenever a IOCTL is triggered by calling `avr_ioctl`, the IOCTL handler of all

[19] For example, the `avr_ioport` module uses the IOCTL system to allow extracting the state of a particular port's `PORT`, `PIN`, and `DDR` registers; `avr_eeprom` allows getting and setting memory locations.

modules registered in the `avr->io_port` linked list is called in sequence until one responds to that particular command by returning a value other than `-1`. This is then returned to the caller.

The reset handler is called whenever `avr_reset` is called, allowing the module to do react appropriately. In *qsimavr*, a major reason for registering as a `avr_io_t` module was to recreate cycle timers and restart VCD traces.

If a module allocates resources, these can be freed during the deallocation handler.

Finally, `avr_io_getirq` lets a module "publish" its IRQs for use by other modules or applications built on top of *simavr*. This function is used whenever a *qsimavr* component is connected to *simavr* modules:

```
avr_connect_irq(avr_io_getirq(avr,
   AVR_IOCTL_IOPORT_GETIRQ(PORT), PIN), irq +
   IRQ_TEMP_DQ);
```

4.10. Value Change Dump (VCD) Files

VCD is a simple file format for dumps of signal changes over time. Each file consists of a header containing general information (most importantly, the used timescale which is always 1ns in *simavr* dumps), variable definitions (containing the name and size of each tracked signal), and finally the value changes themselves. The following example contains the header section, variable definitions, and initial value changes of a three signal VCD file generated by *simavr*:

```
$timescale 1ns $end
$scope module logic $end
$var wire 1 ! <temp.data $end
$var wire 1 " >temp.data $end
$var wire 1 # <temp.ddr $end
$upscope $end
$enddefinitions $end
$dumpvars
0!
0"
0#
$end
#36072750
0!
#36072875
```

```
1"
1!
1#
[...]
```

VCD files can be displayed and analyzed graphically by wave viewers. On Linux, *gtkwave* is well suited for this task (see Figure 4.1).

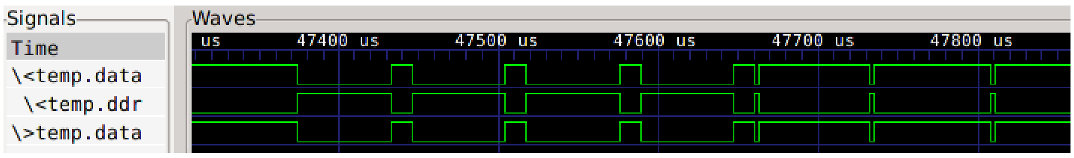

Figure 4.1.: GTKWave

The *simavr* VCD implementation uses a combination of cycle timers and IRQs to track signal[20] changes. After initializing an `avr_vcd_t` with `avr_vcd_init`, tracked signals are configured by calling `avr_vcd_add_signal` . This connects an internal IRQ[21] to the tracked signal, which has `_avr_vcd_notify` registered as a callback function. The latter is called whenever a tracked signal changes, and registers the updated value, the current cycle, and the source IRQ in its log.

Accumulated log data is flushed periodically by a cycle timer, the period of which is specified on `avr_vcd_t initialization`. When a large amount is produced on the tracked signals, it may be necessary to decrease the used period to avoid log overflows.

Tracking can be started and stopped at any time during program execution by calling `avr_vcd_start` and `avr_vcd_stop`. As explained in section 4.3.2, this can even be triggered from the firmware itself.

4.11. Core Definitions

The actual core definitions used by *simavr* are located in the `simavr/cores` subdirectory of the source tree. These definitions rely on *avr-libc* headers to specify the internal structure of an MCU needed for simulation.

The core and all internal components (such as timers, UARTs, IO ports, ADCs, Serial Peripheral Interfaces (SPIs), etc ...) are defined in an internal struct using struct set initialization for a terse representation. The `avr_t` initialization of the `atmega1280` therefore clocks in at only a couple of lines:

[20] In *simavr*, each tracked signal is actually an IRQ.
[21] Limited to 32 connections.

```
.core = {
    .mmcu = "atmega1280",
    DEFAULT_CORE(4),

    .init = m1280_init,
    .reset = m1280_reset,

    .rampz = RAMPZ,
},
```

DEFAULT_CORE[22] initializes basic parameters included in *avr-libc* headers for every MCU such as RAMEND, FLASHEND, etc . . .). The init and reset members point to callbacks which are used to (obviously) initialize and reset the MCU.

Internal components are connected to the avr_t core in the init function:

```
void m1280_init(struct avr_t * avr)
{
    struct mcu_t * mcu = (struct mcu_t*)avr;

    avr_eeprom_init(avr, &mcu->eeprom);
    avr_flash_init(avr, &mcu->selfprog);
    avr_extint_init(avr, &mcu->extint);
    avr_watchdog_init(avr, &mcu->watchdog);
    avr_ioport_init(avr, &mcu->porta);

    /* [...] */
}
```

A short excerpt of atmega1280's TIMER0 initialization should throw some light on how components are configured. Notice how all register and bit locations rely on *avr-libc* definitions:

```
.timer0 = {
    .name = '0',
    .wgm = { AVR_IO_REGBIT(TCCR0A, WGM00),
       AVR_IO_REGBIT(TCCR0A, WGM01), AVR_IO_REGBIT(
       TCCR0B, WGM02) },
    .wgm_op = {
        [0] = AVR_TIMER_WGM_NORMAL8(),
        [2] = AVR_TIMER_WGM_CTC(),
        [3] = AVR_TIMER_WGM_FASTPWM8(),
        [7] = AVR_TIMER_WGM_OCPWM(),
```

[22] The argument specifies the vector size.

```
    },
    .cs = { AVR_IO_REGBIT(TCCR0B, CS00), AVR_IO_REGBIT
       (TCCR0B, CS01), AVR_IO_REGBIT(TCCR0B, CS02) },
    .cs_div = { 0, 0, 3 /* 8 */, 6 /* 64 */, 8 /* 256
       */, 10 /* 1024 */ },

    .r_tcnt = TCNT0,

    .overflow = {
        .enable = AVR_IO_REGBIT(TIMSK0, TOIE0),
        .raised = AVR_IO_REGBIT(TIFR0, TOV0),
        .vector = TIMER0_OVF_vect,
    },
    /* ... */
}
```

Adding a new MCU definition is a simple matter of creating a new `sim_*.c` file in `simavr/cores` and defining all included components with the help of *avr-libc* and a datasheet.

This concludes our tour of the *simavr* core modules. You should now have a good idea of how *simavr* internals work together and complement each other to create an AVR simulation which is accurate, reliable, yet simple, efficient, and easy to extend. For an example of all of these concepts in practice, take a look at the modules included with *simavr*. A good example is the `avr_eeprom` module, which uses a combination of interrupts, an `avr_io_t` module, and IO access callbacks to achieve the desired functionality.

5. Design Approach

Having covered *simavr* in-depth, we now turn our attention to the program developed during the course of this thesis, namely *qsimavr*.

5.1. Project Outline

The aim was to provide an application that could facilitate easy, reliable and useful simulation not only of the AVR core, but also of various external components (such as LCD displays) and their interactions. Specifically, the following set of components should be simulated for the `atmega1280` MCU, roughly based on the BIGAVR6 board by MikroElektronika[23]:

- A standard 2x16 character LCD display based on the HD44780 chip
- An external 1 Kbit EEPROM connected to the MCU via TWI
- A DS1820 Temperature Sensor using 1-wire communication
- The DS1307 RTC, also connected via TWI
- All LEDs connected to `PORT` pins
- Likewise, all push buttons connected to `PORT` pins
- A 128x64 Graphic LCD display containing two chips based on the NT7108 controller
- A touch-sensitive panel (usually attached to the GLCD)

The MMC/SDcard reader which is present on the BIGAVR6 board as well as other external components[24] used by the Microcontroller course at the TU Vienna are currently *not* included (these are all good candidates for further work though).

These components should be simple to connect and disconnect from a running AVR simulation. They should also offer development assistance in the form of VCD optional traces of relevant signals. Graphical output of component state

[23] http://www.mikroe.com/products/view/322/bigavr6-development-system/

[24] Among others, the SmartMP3 board, a Serial Ethernet board, a UART based Bluetooth board, and an external MMC/SDcard reader.

should be presented to the user, and user input should be forwarded to the simulation.

The frontend should offer at least rudimentary control of firmware loading, GDB connection and simulation execution.

Additionally, bonus points are rewarded for cross-platform compatibility and easy extensibility with new components.

5.2. Design Decisions

To satisfy the requirements stated in section 5.1, we settled on a plugin based design with a small core handling the simulation main loop and components loaded from external shared libraries.

The implementation language of choice was C++, both for the sake of familiarity and because interfacing C code with C++ applications is extremely simple and robust.

Qt[25] was used as the Graphical User Interface (GUI) toolkit. It is widely used, distributed and installed by default on most Linux systems. Qt also has great cross-platform compatibility, and as such *qsimavr* should be very simple to port to any other platform that is supported by both *simavr* and Qt. The signal and slot paradigm employed by Qt is well suited to GUI programming, and the QGraphicsScene/QGraphicsView classes are very pleasant to use for more advanced drawing and IO tasks. Finally, QtCreator[26] is a personal favorite IDE of the author, which also works well with cmake.

The build system of our choice is cmake[27]. Again, it provides great cross-platform compatibility and ease of use. Standard Qt specific macros (such as .ui file and `QObject` preprocessing handling) are available. Similar to most build tools, cmake saves compilation time by only compiling needed (changed) files. It also handles installation of the application after compilation. For further details, see section A.2 and the official cmake documentation at `http://www.cmake.org/cmake/help/documentation.html`.

QsLog[28] was chosen as the logging framework for its simplicity. Currently, logging is done to the console only, but a log file is very simple to implement.

The alternative to the chosen software ecosystem was the obvious choice of staying close to the example component implementations included with *simavr*;

[25] `http://qt.nokia.com/products` (Qt is currently in the process of being sold to Digia, don't expect this link to stay valid for long.)
[26] `http://qt.nokia.com/products/developer-tools/`
[27] `http://www.cmake.org/`
[28] `https://bitbucket.org/razvanpetru/qt-components/wiki/QsLog`

namely, C code with graphical components relying on Freeglut[29], a thin layer on top of OpenGl. However, it was clear from the start that standard (and not-so-standard) elements such as menus, toolbars, and Multi Document Interface (MDI) windows would be needed. These widgets are provided out of the box by Qt, and would have taken an infeasible amount of time with a low level library such as Freeglut. While the original plan was to write the *qsimavr* core in C++ but keep components in C, it turned out to feel more natural to implement entirely in C++.

Every Qt application has a main thread where the GUI lives. The *simavr* simulation main loop runs in a separate `QThread` and communicates with the Qt main thread by using queued signal delivery to avoid threading problems. The introduced latency is not detrimental to the application's purpose.

Components are implemented as plugins. They adhere to a standard interface and can be loaded at runtime. This ensures easy extensibility; the component interface is fixed and publicly available. New plugins can be created and loaded easily by third parties without requiring bundled distribution with *qsimavr*. Additionally, the use of a plugin architecture forces strict separation between both the components themselves, and between components and the core.

Each component consists of a mandatory logic and an optional GUI part. The logic part executes in the *simavr* thread and again uses queued signals to communicate with the GUI part living in the main Qt thread. User input is passed from the GUI to the logic handler, and display information in the other direction.

The GUI itself needs to be able to display all component frontends, which again can be enabled and disabled at any time. This immediately brings a multi-window interface to mind. Since we were not completely happy with the idea of many small windows floating individually around the desktop, we looked for further options and soon hit on Gt's MDI interface. The `QMDIArea` (plus its child components `CMDISubWindow`) provide a multi-window interface but encapsulate within a single "main" window.

It would be ideal to have flexible wiring between the components and the core. Some of the standard BIGAVR6 components can be configured to connect to different pins using dip-switches; the connection of external components is completely undefined. MCUs also differ in the pins they have available. However, component wiring in *qsimavr* is currently hardcoded. This allows plugin management to boil down to a simple on/off switch. Wirings are configured for the BIGAVR6 board used in the TU Vienna Microcontroller course.

[29] `http://freeglut.sourceforge.net/`

6. Implementation

We will now examine the actual implementation of *qsimavr* and how its individual parts work together, beginning with the core, the component interface, a short walkthrough of all components, and finally brief descriptions of the modifications and additions made to *simavr* during the course of this thesis.

6.1. qsimavr Core

The core of *qsimavr* is located in the `QSimAVR/` subfolder of the source tree. Its responsibilities include handling the routine tasks of the main window (including menu and action handling, status bar updates, displaying "File Open" dialogs when loading firmware, etc ...), loading and subsequently managing plugins, and actually running the *simavr* main loop.

In the following sections, we will discuss the most interesting parts of the `SimAVR` and `PluginManager` classes. Interested readers should consult the source code for complete implementation details.

6.1.1. `SimAVR`

The `SimAVR` class encapsulates the `avr_t` instance. It runs in its own thread by subclassing `QThread` and being started with the `start()` function.

```
class SimAVR : public QThread
{
    Q_OBJECT

public:
    SimAVR();
    virtual ~SimAVR();

    void load(const QString &filename);
    void run();

public slots:
    void pauseSimulation();
    void stopSimulation();
    void attachGdb();

private:
    avr_t *avr;

    /* [...] */
};
```

`SimAVR` can load firmware, initialize the `avr_t` instance, and execute its main loop:

```
int state;
uint8_t i = 0;
do {
    state = avr_run(avr);
    if (i++ == 0) {
        QCoreApplication::processEvents();
    }
} while (state != cpu_Done && state != cpu_Crashed);
emit simulationStateChanged(Done);
```

The loop itself is very similar to ones we have already seen in section 4.1; additionally, since we heavily use queued signals, we need to call `processEvents` to allow the internal Qt event handler to forward signals. However, profiling sessions have shown `processEvents` to require an unacceptable amount of processing time when called each loop iteration - it appears that every call will require at least one mutex lock and unlock cycle. For performance reasons, we therefore limited event processing to occur only once every 2^8 iterations.[30] Changes in simulation state are in turn emitted as signals (which are then processed by `MainWindow` and displayed in the status bar).

`SimAVR` also provides functions for pausing, unpausing and stopping the simulation, and for preparing *simavr* to listen to incoming GDB connections. Each of these operations also results in changes to the simulation state.

Whenever a firmware is loaded or unloaded, signals are emitted and processed by `PluginManager` to respectively connect and disconnect components from the `avr_t` instance.

[30] This is also somewhat less than ideal when using GDB for short steps, since results will only be visible with an average delay of 2^7 iterations.

6.1.2. `PluginManager`

The plugin manager

- Can load components from shared libraries,
- Creates MDI sub-windows within the main `QMdiArea` as needed,
- Manages their connections to `avr_t` instances as plugins are enabled or disabled and simulation is restarted or loaded with a new firmware file,
- Turns on VCD tracing depending on the selections made in component settings, and
- Saves and restores component settings.

Loading plugins is implemented such that files in a specific directory (which can be set during installation, see section A.2) are iterated one by one. If loading a file fails, an error message is printed to the log targets and loading continues with the next file. Each component must adhere to the component interface (see section 6.1.3) and contain the plugin registration function. It is also possible to keep components in different directories, only placing symlinks (symbolic links) into `PLUGINDIR`.

```
QLibrary lib(filename);
RegisterFunction registerPlugin = (RegisterFunction)
    lib.resolve(PUBLISH_FNAME);

QSharedPointer<ComponentFactory> factory(
    registerPlugin());
```

Component registration uses the `QLibrary` class to load the shared library, and then creates a factory instance by calling the `registerPlugin` function common to each component.

The factory then creates the actual logic and GUI parts. If a GUI part exists, a `QMdiSubWindow` is created containing the component's widget. The logic part is moved to the `SimAVR` thread.

Whenever `firmwareLoaded` or `firmwareUnloaded` signal is sent by `SimAVR`, the logic part of components is, respectively, connected or disconnected from the `avr_t` instance.

Component settings are saved and loaded using the Qt's `QSettings` class, which provides a simple and cross-platform method of handling settings files.

6.1.3. Component Interface

Each component must implement the component interface defined in `QSimAVR/component.h`. It consists of factory, GUI and logic classes, and a `registerPlugin` function which returns a factory instance. Let's take a look at each of these:

```
template<typename Interface>
class Factory {
public:
    Factory() { }
    virtual Interface create() = 0;
    virtual ~Factory() { }

private:
    Q_DISABLE_COPY(Factory)
};

typedef Factory<Component> ComponentFactory;
```

The factory is a generic class which simply provides a pure abstract `create` function to create, initialize and return an object. In *qsimavr*, we only use the instantiated version `ComponentFactory`, which returns a `Component`.

`Q_DISABLE_COPY` prevents copying a class by declaring the copy and assignment constructors in the private section:

```
#define Q_DISABLE_COPY(Class) \
    Class(const Class &); \
    Class &operator=(const Class &);
```

In practice, `ComponentFactory::create` constructs all parts of a component, connects their signals and slots[31] and packs them into a `Component`, which is simply a `struct` containing smart pointers to the component parts:

```
struct Component
{
    QSharedPointer<ComponentGui> gui;
    QSharedPointer<ComponentLogic> logic;
};
```

The GUI part of a component provides a method for the user to interact with the plugin by both displaying component state and allowing user input.

```
class ComponentGui
{
public:
```

[31] It is particularly important to realize that these must use queued signal delivery, since the logic and GUI parts do not reside in the same thread. Specify `Qt::QueuedConnection` in `QObject::connect()`.

```
    ComponentGui() { }

    virtual QWidget *widget() = 0;
    virtual ~ComponentGui() { }

private:
    Q_DISABLE_COPY(ComponentGui)
};
```

Its interface is very simple; communication with the logic part is handled internally by signals, and user interaction is handled by the widget itself. The `QWidget` is displayed as the central widget in a `QMdiSubWindow`. Most of the time, the functionality provided by a GUI component is simple enough such that it can inherit both from `ComponentGui` and `QWidget` itself[32].

The logic part encapsulates a component's internal state; it also reacts to signals, communicates with it's GUI, and is responsible for accurately simulating the behavior of the actual component.

```
class ComponentLogic : public QObject
{
public:
    ComponentLogic(QObject *parent = NULL);
    virtual ~ComponentLogic() { }

    virtual void wire(avr_t *avr);
    virtual void unwire();
    virtual void enableVcd(bool vcdEnabled);

protected:
    virtual void wireHook(avr_t *avr) = 0;
    virtual void unwireHook() = 0;
    virtual void resetHook() { }

protected:
    bool connected;
    bool vcdEnabled;

    avr_vcd_t vcdFile;

    struct component_io_t {
        avr_io_t io;
```

[32] Therefore, `ComponentGui::widget` can simply return the `this` pointer.

```
        ComponentLogic *instance;
    } io;

private:
    static void resetHookPrivate(avr_io_t *io);

private:
    Q_DISABLE_COPY(ComponentLogic)
};
```

Common functionality is handled in `ComponentLogic` itself by implementing it in the public function, which then calls a private hook at an appropriate time (for example, see `wire` and `wireHook`).

`wire` connects a component to the `avr_t` instance. Most of the time, this involves creating internal IRQs, connecting them to the desired target IRQs, setting up notification callbacks, and defining the signals traced by a VCD.

Callback functions require a little extra work because *simavr* is written in plain C and has no notion of classes or member functions. A simple solution to this problem is to register a static class function as the callback, pass `this` as the callback `param`, and let the static function execute the actual member function. A typical example (taken from the temperature component):

```
void TemperatureLogic::wireHook(avr_t *avr)
{
    /* [...] */
    avr_irq_register_notify(irq + IRQ_TEMP_DQ,
       TemperatureLogic::pinChangedHook, this);
    /* [...] */
}

void TemperatureLogic::pinChangedHook(avr_irq_t *,
   uint32_t value, void *param)
{
    TemperatureLogic *p = (TemperatureLogic *)param;
    p->pinChanged(value);
}
```

`enableVcd` simply toggles VCD traces on and off. A default implementation is provided in `ComponentLogic` itself.

Again, supporting *simavr* VCD traces has an interesting quirk; as explained in section 4.10, VCD files depend upon *simavr* cycle timers[33], which are tran-

[33] See section 4.6.

sient and deleted entirely when `avr_reset` is called. We therefore require a way to recreate cycle timers on each reset. This functionality is provided the `avr_io_t` module (section 4.9.2), or more specifically, its `reset` callback. Each component is registered with the `avr_t` core as such a module. On each reset event, VCD connections are automatically restored, and the (optional) `ComponentLogic::resetHook` of every component is called for any custom setup steps.

The final piece of the `Component` interface is the registration function. Plugins must declare themselves using the `PUBLISH_PLUGIN` macro, which defines a C function returning a factory instance:

```
#define PUBLISH_PLUGIN(factory) \
    extern "C" { \
        ComponentFactory *registerPlugin() { return
          new factory; } \
    }
```

6.2. TWI Component

We now turn our attention to the actual component implementations. The TWI component is actually not a component as defined by the component interface in section 6.1.3; rather, it is a library providing helper functions for other components acting as a slave using the TWI protocol[34].

To prevent duplicate work and the Not Invented Here (NIH) syndrome[35], the TWI implementation is an adaption of the work done by the *simavr* author in the `i2c_eeprom` component. The TWI slave logic was extracted into a separate library because it is needed by both the EEPROM (section 6.3) and RTC (section 6.4) components.

We simply connect ourselves to the `TWI_IRQ_MISO` and `TWI_IRQ_MOSI` IRQs, process incoming and outgoing messages according to the TWI protocol, and notify interested parties when we've either received a complete message, or we need to send data to the TWI master.

For that purpose, a `TwiSlave` interface is defined which components using `TwiComponent` must implement.

```
class TwiSlave
{
```

[34] The TWI component compiles into a static library which can be included by any other component requiring it.
[35] For programmers, laziness is a virtue.

```
public:
    virtual uint8_t transmitByte() = 0;
    virtual bool matchesAddress(uint8_t address) = 0;
    virtual void received(const QByteArray &data) = 0;
};
```

`transmitByte` is called by the TWI component whenever the master begins a read transaction and returns the next byte to send. `matchesAddress` returns true, if, and only if (iff) the slave address matches the address sent by the master or if the general call address (`0x0`) has been sent. `received` is called with the received data once a write transaction completes, and allows the slave to react to a transmission.

6.3. EEPROM Component

The EEPROM uses the TWI component briefly described in section 6.2. It has been implemented using the datasheet [?] as a reference. The simulated EEPROM consists of 1 kbit of storage and is controlled over a TWI bus.

The implementation itself is very simple since the TWI logic is encapsulated by the TWI component. We can therefore look at the EEPROM from a very high level, and are only interested in keeping track of the Address Pointer (AP) and the state of the internal memory. Reading from the EEPROM returns the byte pointed to by the AP and then increments the AP. Similarly, writing data section writes it to the AP and subsequently increments it.

Pin/IRQ	Function
TWI IRQs	TWI connections

Table 6.1.: EEPROM Wiring

This functionality is achieved by implementing the `TwiSlave` methods as follows (edited for clarity):

```
uint8_t EepromLogic::transmitByte()
{
    return eeprom[incrementAddress()];
}

void EepromLogic::received(const QByteArray &data)
{
    addressPointer = data[0];
    for (int i = start; i < data.size(); i++) {
        eeprom[incrementAddress()] = data[i];
    }
}

bool EepromLogic::matchesAddress(uint8_t address)
{
    return ((address & EEPROM_MASK) == (EEPROM_ADDR &
       EEPROM_MASK) ||
            (address & 0xfe) == 0x00);
}
```

The `QByteArray` is used to represent the byte storage internally as it provides both the convenience of many helper functions, and the advantages of raw memory arrays through `QByteArray::data`.

6.4. RTC Component

The RTC component consists of a 56 byte memory area[36], communications over a TWI bus, and timekeeping logic which updates its clock registers once per second [**?**].

Pin/IRQ	Function
TWI IRQs	TWI connections

Table 6.2.: RTC Wiring

Internally, the implementation is very similar to the EEPROM examined in section 6.3; so much in fact, that only differences to the EEPROM will be discussed here. Writing and reading from the memory storage is identical.

The first 7 bytes represent the clock's date and time in Binary Coded Decimal (BCD) format and need to be updated periodically to reflect the passing of time. Once again, *simavr*'s cycle timers (discussed in section 4.6) came in handy to achieve the desired functionality. A callback is registered to occur once every second (of simulated time), which updates the the date and time bytes. Handling of leap years is simplified by relying on Qt's `QDate` class.

The square wave pin of the actual component is *not* implemented.

[36] 8 of which are used to represent clock state, with the rest usable as generic storage.

6.5. GLCD Component

The GLCD component simulates a 128x64 Graphic LCD display, based internally on two NT7108 controller chips. Each of these has 512 bytes of display Random-Access Memory (RAM), and three address pointers (storing the horizontal pixel address, the vertical page address, and the first page displayed at the top edge of the screen) [**?**, **?**, **?**].

The touchscreen is also included within this component. The initial idea was to implement it in a separate component which would be displayed in a transparent MDI window the same size as the GLCD screen and could therefore be layered on top of it. However, limitations in MDI transparency handling (which either drew the entire window transparent including borders and title bar or everything opaque) put an end to that idea.

Pin/IRQ	Function
`PORTA0-7`	Data Pins
`PORTE2`	Chip Select 1 (CS1)
`PORTE3`	Chip Select 2 (CS2)
`PORTE4`	Register Select (RS)
`PORTE5`	Read/Write (RW)
`PORTE6`	Enable (E)
`PORTE7`	Reset (RST)
ADC0 IRQ	Touchscreen X Axis
ADC1 IRQ	Touchscreen Y Axis

Table 6.3.: GLCD Wiring

This is one of the most complex components, as it both requires nontrivial steps to decode instructions sent by the AVR core, and contains a GUI part which must handle both rudimentary pixel-wise drawing and mouse input. The GLCD reacts to edges of the E pin. On the rising edge, the pin state is saved into the command buffer. If the command triggers a read operation[37], we queue a cycle timer which will write the requested data to the data pins upon completion of 320 nanoseconds (ns) data delay time. Note that display read data first passes an intermediate read buffer before it is actually output to the data pins. This simulates latching data into the output register which actually occurs in the real hardware[38].

Write operations are handled on the falling E edge. After decoding the

[37] By "read operation", we mean that the AVR core wants to read from the GLCD component.

[38] Status reads are not affected by this.

buffered command, current state of the data pins is used to perform the operation.

Each controller chip is simulated by an instance of the `NT7108` class. These are coordinated by the `GlcdLogic` class, which forwards signals according to the current state of the CS1 and CS2 pins. Whenever an operation causes the visible state of the display to change, a signal is emitted which is in turn handled by the GUI part of the component.

The frontend to the GLCD is implemented using a `QGraphicsScene`, which simplifies routine tasks such as drawing and receiving user input. Each pixel is represented by a `QGraphicsRectItem`; if it is turned off, we use a `Qt::green` brush, otherwise it is displayed as `Qt::black`. Now, when a page of the GLCD display RAM is changed, all eight affected pixels are updated.

The scene also receives mouse events and emits these as signals. These are received by the logic part and forwarded to an instance of the `Touchscreen` class, where the affected coordinates are buffered. These can be requested by the AVR core by querying ADC0 and ADC1, which are respectively responsible for the X and Y axis. An ADC value of 2700 millivolts corresponds approximately to the upper and right edges of the screen. The voltages scale linearly until reaching 0 volts at the lower and left edges. If no "touch" is currently active, the coordinates are reset to the neutral lower-left corner. Touchscreen pins only return meaningful values if its drive pins are set correctly, otherwise a default of zero millivolts is returned and a warning is printed to `stderr`.

6.6. LCD Component

This component simulates a 2x16 LCD display based on the HD44780 chip [?, ?, ?]. A substantial part of the implementation was taken from the `hd44780` example part included with *simavr*.

Pin/IRQ	Function
`PORTC4-7`	Data Pins
`PORTC2`	Register Select (RS)
`PORTC3`	Enable (E)

Table 6.4.: LCD Wiring

Communication is very similar to the GLCD examined in section 6.5 and will not be discussed further in this section. Font selection, custom fonts and cursor display are not implemented, allowing us to use a simple `QTextBrowser` to display LCD output.

6.7. LEDs & Buttons Component

The 86 buttons and LEDs of the BIGAVR6 board are bundled into this simple component.

Pin/IRQ	Function
`PORTA0-7`	LEDs & Buttons
`PORTB0-7`	LEDs & Buttons
`PORTC0-7`	LEDs & Buttons
`PORTD0-7`	LEDs & Buttons
`PORTE0-7`	LEDs & Buttons
`PORTF0-7`	LEDs & Buttons
`PORTG0-5`	LEDs & Buttons
`PORTH0-7`	LEDs & Buttons
`PORTJ0-7`	LEDs & Buttons
`PORTK0-7`	LEDs & Buttons
`PORTL0-7`	LEDs & Buttons

Table 6.5.: LEDs & Buttons Wiring

Each `QPushButton` also doubles as a LED by changing its background color to represent LED state. When a button is pressed, the pin is pulled low. We assumee that pull-ups are enabled on all ports, and therefore pull pins to high when a button is released. The implementation is otherwise fairly trivial and there is not much of interest to be mentioned.

6.8. Temperature Sensor Component

The DS1820 temperature sensor contains a 9 byte scratchpad memory, a 2 byte EEPROM, and uses a 1-wire protocol to communicate with the AVR core [?].

Pin/IRQ	Function
`PORTG0`	1-Wire Data
`DDRG`	1-Wire Data Direction

Table 6.6.: Temperature Sensor Wiring

This component surprised us with the unexpected complexity of the 1-wire protocol. Implementation was further complicated by the difficulties we've encountered caused by the implementation of IO pin levels in *simavr*, which does not know how to to behave correctly with a connected component; for example, *simavr* blindly sets pin levels, even if the IO port is set to input mode. Unfortunately, we had to use fairly unpleasant methods to counteract that.

Let's recall how *simavr* handles a write to an IO port. *simavr* has no notion of any connected components; there is no way to tell it whether the component has a pull-up resistor nor whether the component is currently transmitting high or low. When *simavr* receives a IO port write, it also does not know if it is coming from the AVR core, or from an external component.

```
static void avr_ioport_write(struct avr_t * avr,
   avr_io_addr_t addr, uint8_t v, void * param)
{
    avr_ioport_t * p = (avr_ioport_t *)param;
    avr_core_watch_write(avr, addr, v);

    for (int i = 0; i < 8; i++)
        avr_raise_irq(p->io.irq + i, (v >> i) & 1);
    avr_raise_irq(p->io.irq + IOPORT_IRQ_PIN_ALL, v);
}
```

As you can see, the `IOPORT` IRQs are triggered on every port write. This of course presents us with a problem; even if the port is set to input[39], every port write will be propagated to the IRQ representing pin state. The component not only has to ignore this invalid level change, it also has to revert it. Since setting a new IRQ level is not possible within the IRQ callback function, we've fallen back on scheduling a cycle timer with a duration of zero to correct the IRQ value. Of course, all of this only serves to complicate component code and to hide its actual purpose.

Even after ignoring the issue of erroneous pin values (which is solved by ignoring all pin changes not set by the component itself), the temperature component still has to simulate the behavior of its internal pull-up. We have achieved this by listening for changes of the DDR pin. When the port is switched to input mode, we pull the 1-wire pin high manually. Upon switching to output mode, *simavr* automatically restores the pin level appropriately.

Having discussed the encountered difficulties and the employed solutions, we can now turn our attention to the actual protocol implementation located in the `DS1820` class. 1-wire is a time-based protocol, again pointing to the use of cycle timers.

By saving the cycle of the most recent pin edge and comparing it with the current cycle, we can easily calculate the low- and high periods. Receiving a

[39] A port is set to input by clearing the Data Direction Register (DDR) pin.

low pulse of a minimum of 480 microseconds (μs) resets the state machine, regardless of the state we are currently in. After a reset, the DS1820 expects a ROM command, which is often followed by a function command. Either of these can involve both reads and writes. Details of the protocol are excellently demonstrated in the datasheet [**?**].

The DS1820 has two memory areas - a transient scratchpad, and a 2 byte EEPROM for permanent storage of alarm levels. Both can be accessed by the AVR core using specific command sequences. On each scratchpad write access, the Cyclic Redundancy Check (CRC) sum stored in the 9th byte is updated using the `crc8` function[40]. Changes to either area are emitted as signals to the GUI part, which displays the memory in an editable hex editor widget[41]. If the user changes memory areas, these are in turn signaled to the `DS1820` class, which updates the corresponding memory areas.

Triggering a temperature conversion in this implementation does nothing, since the current temperature can always be written directly into the scratchpad memory by the user.

6.9. Changes and additions to simavr

While working on *qsimavr*, we have also spent a very substantial amount of time with *simavr* itself; we have not only worked on adding new features, but also encountered several misbehaviors which we have often been able to correct after extensive debugging. In this section, we will discuss the more interesting changing to *simavr* which we have contributed during the course of this thesis[42].

The `atmega1280` core definition has been amended to include correct definitions of the upper ADC differential channels, timers 4 and 5, UART 2 and 3, and EINT4 to EINT7, and raised `MAX_IOs` to include all required IO registers.

We've added support for GDB data watchpoints (which has already been partly described in section 4.7).

An error preventing correctly reading and writing SREG from GDB was fixed by de- and reconstructing SREG on each access.

Inaccurate sleep timing caused by short cycle timer periods were solved by accumulating requested sleep times until a certain threshold has been reached (see also section 4.2).

[40] Based on code by Colin O'Flynn.

[41] The used widget is QHexEdit2 by Winfried Simon. The source code is available at `http://code.google.com/p/qhexedit2/`.

[42] A complete list of contributions is available in the simavr git commit log. For instructions on how to retrieve the source code see section A.1.

We prevented spurious CPU wakeups while stopped by explicitly checking for `avr->state == cpu_Sleeping` (instead of `avr->state != cpu_Running`).

Although many parts of *simavr* could already be included into C++ projects without issues, some were still missing the required preprocessor instructions. We've amended these such that every header can now be used by C++ applications.

Inconsistent handling of `avr_terminate` has been improved by never calling it within *simavr*. `avr_t` destruction must *always* be done by the user. This interfered with cleanup routines which expected `avr_t` to stay valid while in fact they could be destructed when the AVR core sleeps while interrupts have been turned off.

Support has been added for interrupts which (contrary to the norm) do not clear their "raised" flag bit when entering the ISR. In particular, this concerns the TWI interrupt flag (TWINT) and caused problems during the TWI simulation.

We've corrected an issue which would incorrectly identify an "sleeping with interrupts off" state and terminate simulation. This occurred when an interrupt was raised and a SLEEP instruction processed before the interrupt could be serviced due to `pending_wait`.

Logging output, which was previously printed to `stderr` and `stdout` without distinction, is now categorized into log levels with increasing verbosity `LOG_ERROR`, `LOG_WARNING`, and `LOG_TRACE`. The log level can be selected by passing `-v` when running the *simavr* binary, or setting `avr->log`.

We've fixed a bug in timer handling related to TCNT IO register writes that could result in an infinite cycle timer loop.

A situation in which the IO port IRQ would miss level changes was fixed by removing manual filtering and relying instead on the `IRQ_FLAG_FILTERED` behavior.

Larger AVRs now correctly use the `EICRB` register for external interrupts 4 to 7.

GDB deinitializes itself during `avr_terminate`. This is relevant when running multiple simavr sessions in a single program to avoid memory leaks and correct socket operation.

Finally, `IO` port IRQs now restore themselves to PORT the register value when DDR is switched to output mode.

7. Results and Discussion

During the course if this thesis, we have developed a new open source frontend to *simavr* targeted towards simulating AVR hardware components using a simple, extensible, and powerful plugin architecture. Six components (simulating eight actual hardware components) with very diverse tasks have been implemented, serving as both initial functionality and examples for further component implementations. Example code is available for the TWI and 1-wire protocols.

Benefits have also reached upstream[43] in the form of numerous bug fixes (especially regarding the `atmega1280` MCU) and new features such as GDB watchpoints. The existing documentation has already been improved, and hopefully the *simavr* chapters of this thesis will also be of great value (sections 4 and A.1).

We would like to point out that none of this would have been possible without the open source ecosystem. We have used other developers' code, learned through many freely available examples, and contributed back to upstream projects. Having the source code available is also extremely important while debugging. Of course, *qsimavr* is also released as an open source project, in the hope that it will be useful to others.

We also welcome all contributions, further developments and bug fixes. In particular there are several topics which could use further attention and would be good candidates for follow-up projects, both within *qsimavr* and *simavr*:

- IO pin level handling

  simavr's current `avr_ioport` implementation complicates matters when intricate interactions with connected components are required. This issue has already been discussed in section 6.8. We envision a solution in which *simavr* is aware of the state of the communication partner (for example, it could be pulling the pin high with a weak pull-up resistor), and automatically sets the pin level correctly.

[43] In free and open source projects, the upstream of a program or set of programs is the project that develops those programs. [...] This term comes from the idea that water and the goods it carries float downstream and benefit those who are there to receive it. [?] In this case, *simavr* is the upstream project.

- Sleep modes

  simavr has no notion of different sleep modes. So far, we have not run into a situation in which this turned out to be a problem, but improving simulation accuracy is always a valid goal.

- Component chaining

  qsimavr currently forces all components to connect directly to the core. While we use a chaining-like mechanism with the EEPROM, RTC, and TWI components, it is currently not possible to properly connect one component to another component.

- Flexible wiring

  At the time of writing, all wiring between components and the AVR core is hardcoded. Flexible wiring (allowing the user to configure which pins components are connected to) would provide substantial advantages, both for using components with different AVR cores with different configurations, and for altering the configuration of a component on the board itself (the BIGAVR6 board even allows configuring the wirings of several internal components with dip-switches).

- Per-component logging

  Components currently print all messages to the console. It would be useful to be able to separate logging messages by component, maybe even displaying these within *qsimavr* itself.

- Custom component configurations

  While we do store component settings, these are common to all components. Ideally, each component should be able to manage its own custom configurations. This could also be used for the purpose of storing permanent state such as EEPROM contents.

- Cross-platform compatibility

  The current version of *qsimavr* has only been developed and tested on Linux. All used technologies support several platforms, and porting *qsimavr* should take only a short time. Compilation and setup instructions would also have to be created.

8. Conclusion

qsimavr fills a gap in the open source ecosystem of AVR utilities by providing a simple way of simulating AVR microcontrollers including external periphery. The application is simple to use, and full documentation is provided for *qsimavr* and *simavr* internals.

Components are implemented using a plugin architecture with a well documented interface and numerous examples, allowing for easy extensibility by interested parties. All components can include graphical output and accept input from the user.

Furthermore, VCD trace files can be produced during execution which substantially aids development and debugging of hardware drivers.

GDB debugging is also supported through the use of the GDB Remote Serial Protocol, allowing developers to use established debugging tools they are already familiar with.

qsimavr provides simulation for eight components that should cover all base use cases of the BIGAVR6 board. If required, further components such as external Bluetooth and Ethernet modules could be implemented as follow-up projects.

9. Acknowledgements

This thesis has been made possible by the kind help of several people.

First and foremost, I'd like to thank the author of *simavr*, Michel Pollet, for the help and many discussions during the past couple of months. (And of course for writing *simavr*, the basis for this thesis!)

The entire open source ecosystem, for making a project like this possible in the first place. In particular, Martin Thomas (the author of the DS18X20 demo application), Colin O' Flynn (the author of the CRC routine used by the temperature sensor), and Winfried Simon (for the QHexEdit widget).

My colleages Mino Sharkhawy and Ondrej Hosek for kindly providing their AVR applications as testbeds.

Alexander Kössler, who has lent valuable assistance whenever I needed it; together with the entire Microcontroller team at the University of Vienna, for their support, code, motivation and company last semester.

And finally, Moni Linke, who has always supported me in everything I do.

Bibliography

[1] avr-libc authors. *avr-libc*. http://savannah.nongnu.org/download/avr-libc/avr-libc-user-manual-1.8.0.pdf.bz2, accessed 2012-08-27.

[2] Wikipedia contributors. Emulator. http://en.wikipedia.org/w/index.php?title=Emulator&oldid=511120965, accessed 2012-09-12.

[3] Wikipedia contributors. Simulation. http://en.wikipedia.org/w/index.php?title=Simulation&oldid=510520615, accessed 2012-09-12.

[4] Atmel Corporation. *8-bit AVR Instruction Set*. www.atmel.com/images/doc0856.pdf, accessed 2012-08-27.

[5] Atmel Corporation. Atmel studio6 - two architectures, one studio - debugging. http://www.atmel.com/microsite/atmel_studio6/debugging_simulation.aspx, accessed 2012-08-31.

[6] Samsung Electronics. *KS0066U Datasheet*. www.britestone.com/shop/upload/DN/LCD/LC1621-SMLYH6.pdf, accessed 2012-08-27.

[7] Samsung Electronics. *KS0108B Datasheet*. http://www.alldatasheet.com/datasheet-pdf/pdf/37323/SAMSUNG/KS0108B.html, accessed 2012-08-27.

[8] Ltd. Hitachi. *HD44780U Datasheet*. www.sparkfun.com/datasheets/LCD/HD44780.pdf, accessed 2012-08-27.

[9] Microchip Technology Inc. *24AA01/24LC01B Datasheet*. ww1.microchip.com/downloads/en/devicedoc/21711c.pdf, accessed 2012-08-27.

[10] Microchip Technology Inc. *ENC28J60 Datasheet*. ww1.microchip.com/downloads/en/devicedoc/39662b.pdf, accessed 2012-08-27.

[11] Neotec Semiconductor Ltd. *NT7108 Datasheet*. http://www.datasheetarchive.com/NT7108-datasheet.html, accessed 2012-08-27.

[12] Maxim Integrated Products. *DS1307 Datasheet*. datasheets.maxim-ic.com/en/ds/DS1307.pdf, accessed 2012-08-27.

[13] Maxim Integrated Products. *DS18S20 Datasheet*. datasheets.maxim-ic.com/en/ds/DS18S20.pdf, accessed 2012-08-27.

[14] Emerson W. Pugh. Building ibm: Shaping an industry and its technology. p. 274.

[15] Richard Stallman, Roland Pesch, Stan Shebs, and et al. *Debugging with GDB*. `http://sourceware.org/gdb/current/onlinedocs/gdb.pdf.gz`, accessed 2012-08-28.

[16] Rahul Sundaram. Staying close to upstream projects. `https://fedoraproject.org/wiki/Staying_close_to_upstream_projects`, accessed 2012-08-31.

[17] LTD Winstar Display Co. *WDG0151-TMI-V#N00 Datasheet*. `http://www.mikroe.com/esupport/index.php?_m=downloads&_a=viewdownload&downloaditemid=156`, accessed 2012-08-27.

[18] LTD Winstar Display Co. *WH1602B-TMI-ET# Datasheet*. `http://www.datasheetarchive.com/WH1602B-datasheet.html`, accessed 2012-08-27.

A. Setup Guide

This section provides instructions on how to retrieve, compile and install *simavr* and *qsimavr* on the GNU/Linux operating system.

A.1. simavr

A.1.1. Getting the source code

The official home of *simavr* is `https://github.com/buserror-uk/simavr`. Stable releases are published as git repository tags (direct downloads are available at `https://github.com/buserror-uk/simavr/tags`). To clone a local copy of the repository, run

```
git clone git://github.com/buserror-uk/simavr.git
```

A.1.2. Software Dependencies

elfutils is the only hard dependency at run-time. The name of this package may differ from distro to distro. For example, in Ubuntu the required package is called *libelf-dev*.

At compile-time, *simavr* additionally requires *avr-libc* to complete its built-in AVR core definitions. It is assumed that further standard utilities (*git*, *gcc* or *clang*, *make*, etc ...) are already present.

simavr has been tested with the following software versions:

- Arch Linux x86_64 and i686
- elfutils 0.154
- avr-libc 1.8.0
- gcc 4.7.1
- make 3.82

Furthermore, the board_usb example depends on libusb_vhci and vhci_hcd. For further details, see *examples/board_usb/README*. Note however that these are not required for a fully working *simavr* build.

A.1.3. Compilation and Installation

simavr's build system relies on standard makefiles. The simplest compilation boils down to the usual

```
make
make install
```

As usual, there are several variables to allow configuration of the build procedure. The most important ones are described in the following section:

- AVR_ROOT

  The path to the system's *avr-libc* installation.

  While the default value should be correct for many systems, it may need to be set manually if the message 'WARNING . . . did not compile, check your avr-gcc toolchain' appears during the build. For example, if iomxx0_1.h is located at /usr/avr/include/avr/iomxx0_1.h, AVR_ROOT must be set to /usr/avr.

- CFLAGS

  The standard compiler flags variable.

  It may be useful to modify CFLAGS for easier debugging (in which case optimizations should be disabled and debugging information enabled: -O0 -g). Additionally adding -DCONFIG_SIMAVR_TRACE=1 enables extra verbose output and extended execution tracing.

These variables may be set either directly in Makefile.common, or alternatively can be passed to the make invocation (make AVR_ROOT=/usr/avr DESTDIR=/usr install).

Installation is managed through the usual

```
make install
```

The DESTDIR variable can be used in association with the PREFIX variable to create a *simavr* package. DESTDIR=/dest/dir PREFIX=/usr installs to /dest/dir but keeps the package configured to the standard prefix (/usr).

For development, we built and installed *simavr* with the following procedure:

```
make clean
make AVR_ROOT=/usr/avr CFLAGS="-O0 -Wall -Wextra -g -fPIC \
  -std=gnu99 -Wno-sign-compare -Wno-unused-parameter"
make DESTDIR="/usr" install
```

A.2. qsimavr

A.2.1. Getting the source code

The *qsimavr* source code is available from `git://github.com/schuay/qsimavr.git`, with the homepage located at `https://www.github.com/schuay/qsimavr`. Clone it using

```
git clone git://github.com/schuay/qsimavr.git
```

A.2.2. Software Dependencies

Since *qsimavr* depends on *simavr*, it shares all of its dependencies. The only additional requirements are *qt*, *cmake* and *avr-gcc*.

qsimavr has been tested with

- Arch Linux x86_64 and i686
- qt 4.8.2
- cmake 2.8.9
- avr-gcc 4.7.1

A.2.3. Compilation and Installation

qsimavr follows standard cmake procedure and supports out-of-tree builds. To generate makefiles, run

```
cmake /path/to/qsimavr/source
```

To switch to a debug build, add -DCMAKE_BUILD_TYPE="Debug" to the cmake invocation. Installation paths can be customized using -DCMAKE_INSTALL_PREFIX. See the cmake documentation for further information.

The rest of the compilation and installation process consists of running

```
make
make install
```

As usual, DESTDIR may be used for packaging. Other standard make variables also apply.

B. User Guide

B.1. qsimavr User Guide

Within this guide, we will assume the use of firmware compiled for the `atmega1280` with a frequency of 16 MHz[44]. After starting *qsimavr*, you are presented with a screen which should look somewhat similar to figure B.1.

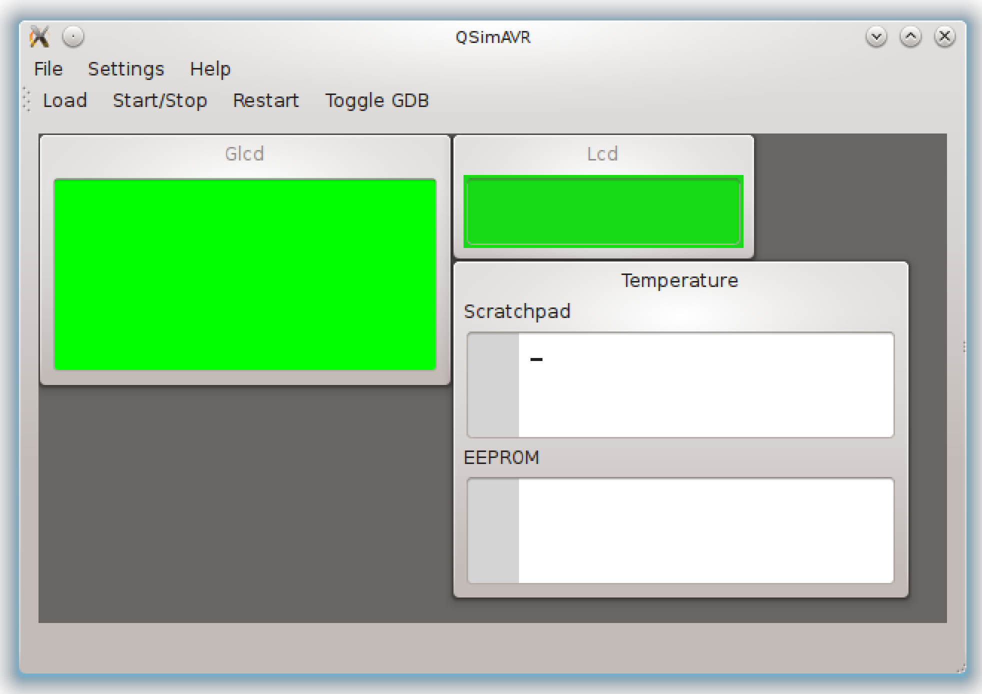

Figure B.1.: *qsimavr*

Trace and error output, as well as *simavr* messages are printed directly to the console. VCD files are saved into the current working directory.

Loaded components can be configured in the *Settings* menu. Currently, possible options include enabling or disabling a component and toggling VCD

[44] These are the default settings.

traces.

A firmware file may be loaded by either pressing *Load* in the toolbar, selecting *Load Firmware...* in the *File* menu, or clicking on one of the recently loaded files listed in the *File* menu. Simulation starts automatically once the firmware file has been loaded successfully.

The current simulation state is displayed in the lower left of the status bar once a firmware file has been loaded.

For debugging (also see B.2), GDB support must be enabled by clicking the *Toggle GDB* button in the toolbar. If a simulation is currently in progress, execution is halted (note the change in simulation state); otherwise the CPU will be paused once the next firmware file has been loaded or the simulation is restarted.

Once running, it is possible to pause and unpause the simulation by pressing the *Start/Stop* button in the toolbar, or restarting it by pressing *Restart*.

The components are all fairly self-explanatory to use. The EEPROM, RTC and temperature sensor display their memory contents in a hex editor which can also be used to alter their state; the GLCD acts as a touchscreen on mouse presses. The LED buttons can of course be clicked to interact with the AVR core[45].

B.2. Debugging with simavr and GDB

As discussed in section 4.7, *simavr* supports GDB through the Remote Serial Protocol, which allows us to debug AVR programs locally or even from a remote location. Debugging is enabled in *qsimavr* by clicking on the *Toggle GDB* button in the toolbar. Further steps are exactly the same as when starting *simavr* with the `-g` flag.

Let's walk through a debugging session of a short example program[46], reproduced in full here:

```
/**
    This is the C interrupt callback demo.

    Enable the LEDs for port A.
```

[45] Because of the large number of connected IRQs, it is sometimes a good idea to disable the LED buttons component when running a firmware with a high frequency of port traffic.

[46] This is the interrupt & callback demo of the Microcontroller course at the TU Vienna. The source code is also available at `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/c-interrupt-demo/at_download/file` (accessed 2012-08-31).

```
    Whenever INT7 or INT6 is pressed the counting
    mode changes from increasing to decreasing and
       vice versa.
*/

#define F_CPU       (16000000UL)

#include <avr/io.h>
#include <avr/sleep.h>
#include <avr/interrupt.h>

#ifndef NULL
#define NULL    ((void *) 0)
#endif
#ifndef TRUE
#define TRUE    (0 == 0)
#endif
#ifndef FALSE
#define FALSE   (!TRUE)
#endif

/* prototypes */
static uint8_t increment(const uint8_t counter);
static uint8_t decrement(const uint8_t counter);
static void setCallback(uint8_t (*_interrupt)(const
   uint8_t counter));
static void callCallback(volatile uint8_t *counter);

/* variables */
static uint8_t (*interrupt)(const uint8_t counter);
static volatile uint8_t isFallingEdge, click;

int main(void)
{
    /* initialize variables */
    setCallback(&increment);
    isFallingEdge = TRUE;
    click = FALSE;

    /* Initialize LEDs. */
    DDRA = 0xFF;
    PORTA = 0x00;
    /* Set up interrupts */
    DDRE  &= ~((1<<PE6) | (1<<PE7));
    PORTE |= (1<<PE6)   | (1<<PE7);
    EICRB = (EICRB & ~((1<<ISC60) | (1<<ISC70))) |
       (1<<ISC61) | (1<<ISC71);
    EIMSK |= (1<<INT6)  | (1<<INT7);

    /* Set up timer interrupt */
    TCCR1A &= ~(_BV(COM1A1) | _BV(COM1A0) | _BV(COM1B1
       ) | _BV(COM1B0) | _BV(COM1C1) | _BV(COM1C0) |
       _BV(WGM11) | _BV(WGM10));
    TCCR1B &= ~(_BV(ICNC1) | _BV(WGM13) | _BV(ICES1) |
        _BV(CS12) | _BV(CS11) | _BV(CS10));
    OCR1A = ((250 * F_CPU) / (64 * 1000U)) - 1;
    TIMSK1 |= _BV(OCIE1A);
    TCCR1B |= _BV(CS11) | _BV(CS10) | _BV(WGM12);

    sei();

    set_sleep_mode(SLEEP_MODE_IDLE);
    sleep_enable();

    while(TRUE)
    {
        /* Set CPU to sleep mode. */
        sleep_cpu();
    }
    return 0;
}

/* helper functions */
static uint8_t decrement(const uint8_t counter)
{
    return counter - 1;
}

static uint8_t increment(const uint8_t counter)
{
    return counter + 1;
}
```

```
static void setCallback(uint8_t (*_interrupt)(const
   uint8_t counter))
{
    interrupt = _interrupt;
}

static void callCallback(volatile uint8_t *counter)
{
    if(interrupt != NULL)
    {
        *counter = (*interrupt)(*counter);
    }
}

/* interrupts */
ISR(TIMER1_COMPA_vect, ISR_BLOCK)              /*
   Blocking timer interrupt */
{
    static volatile uint8_t counter = 0;

    click = FALSE;
    callCallback(&counter);
    PORTA = counter;
}

ISR(INT7_vect, ISR_NOBLOCK)                    /* Non
   blocking interrupt on PE7 */
{
    if(!click)
    {
        if(isFallingEdge)
            setCallback(&decrement);
        else
            setCallback(&increment);

        isFallingEdge = !isFallingEdge;
    }
    click = TRUE;
}

ISR(INT6_vect, ISR_ALIASOF(INT7_vect));          /* Alias
    for PE6 to do the same as on PE7 */
```

This program executes a callback function once per timer interrupt, which either increases or decreases the value displayed on the `PORTA` LEDs. The selected callback can be switched by triggering `INT7` or `INT6`, which correspond to the buttons on `PORTE6` and `PORTE7`.

We will begin by testing *simavr*'s GDB passive mode, which is triggered on invalid memory accesses, such as trying to read from `RAMEND + 1`:

```
diff --git a/main.c b/main.c
index 5cc85f9..2d04541 100644
--- a/main.c
+++ b/main.c
@@ -41,7 +41,7 @@ int main(void)

     /* Initialize LEDs. */
     DDRA = 0xFF;
-    PORTA = 0x00;
+    PORTA = *((uint8_t *)RAMEND + 1);

     /* Set up interrupts */
     DDRE  &= ~((1<<PE6) | (1<<PE7));
```

Compile this program, making sure to include debug symbols by passing the `-g` flag:

```
avr-gcc -mmcu=atmega1280 -Wall -Wstrict-prototypes -Os
    -frename-registers -fshort-enums -fpack-struct -g
    -c -o main.o main.c
avr-gcc -mmcu=atmega1280 -Wl,-u,vfprintf -lprintf_min
    -o main.elf main.o
```

Load the firmware into *qsimavr*. As soon as execution hits the invalid memory access, an error message is printed, and GDB is automatically set up to listen on port 1234:

```
CORE: *** Invalid read address PC=0498 SP=21fd O=4091
    Address 2200 out of ram (21ff)
avr_sadly_crashed
```

We can now connect GDB[47] to the simulation by executing `target remote localhost:1234`. Debugging symbols are then loaded

[47] Make sure you are using the AVR version of GDB, usually called *avr-gdb*.

by pointing GDB towards the ELF file with `file main.elf`. We can now start digging around for the error; for example, to see a disassembly of area around the current PC, type in `disassemble`. To retrieve the values of all registers, type `info registers`. A backtrace can be created by typing `backtrace`.

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000049c in ?? ()
(gdb) file main.elf
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from main.elf...done.
(gdb) disassemble
Dump of assembler code for function main:

   [...]

   0x00000498 <+26>:    lds     r20, 0x2200
=> 0x0000049c <+30>:    out     0x02, r20       ; 2
   0x0000049e <+32>:    in      r21, 0x0d       ; 13

   [...]

End of assembler dump.
(gdb) info registers

    [...]

r29            0x21     33
r30            0x0      0
r31            0x0      0
SREG           0x2      2
SP             0x8021fd 0x8021fd
PC2            0x49e    1182
pc             0x49e    0x49e <main+32>
(gdb) bt
#0  main () at main.c:47
```

You can either experiment further with GDBs commands, read through the inline help (type `help`), or quit GDB by typing `quit`. *simavr*'s passive mode is very useful in situations where a program crashes unexpectedly.

Let'd do some more deliberate debugging. Fix your program by reverting the invalid memory reference we inserted above, make sure GDB support is switched on in *qsimavr*, and load the newly compiled firmware. You already know how to attach GDB and load debugging symbols:

```
(gdb) target remote localhost:1234
(gdb) file main.elf
```

Say we're interested in the `increment` function, and we'd like to examine every execution of it. Settings a breakpoint will halt simulation every time that function is called. We can either set breakpoints by function names, or by file name and line numbers:

```
(gdb) break increment
Breakpoint 1 at 0x128: file main.c, line 81.
(gdb) break main.c:81
Note: breakpoint 1 also set at pc 0x128.
Breakpoint 2 at 0x128: file main.c, line 81.
```

Breakpoints can be deleted by using the `delete` command:

```
(gdb) delete 2
```

Now, continue running the simulation with the `continue` command. As soon as `increment` is called for the first time, simulation pauses and control is returned to GDB:

```
(gdb) continue
Continuing.

Breakpoint 1, increment (counter=0 '\000') at main.c:81
81      }
```

The current location within the source code can be printed by using the `list` command. We can also examine (and even change) variable values with the `print` and `list` commands. It is even possible to call functions on demand with `call`:

```
(gdb) print counter
$1 = 0 '\000'
(gdb) set counter = 42
(gdb) print counter
$2 = 42 '*'
(gdb) call decrement(counter)
$3 = 41 ')'
```

What if you were actually interested in all places in which the counter value is accessed instead of specific locations in your program? This kind of monitoring is also possible using GDB watchpoints, which alert GDB whenever the value of an SRAM location changes:

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) watch counter
Watchpoint 3: counter
(gdb) continue
Continuing.
Watchpoint 3: counter

Old value = 42 '*'
New value = 43 '+'
0x0000012a in increment (counter=43 '+') at main.c:81
81      }
```

This concludes our short tour of debugging AVR programs with *qsimavr* and GDB. For further details, consult the GDB documentation [?] or one of the innumerable GDB tutorials which can be found on the net[48].

[48] A good one is RMS' gdb Debugger Tutorial at `http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html`.