simavr Manual

Jakob Gruber
jakob.gruber@gmail.com

October 5, 2020

Contents

1. Introduction

This manual is an excerpt of the bachelor's thesis "*qsimavr: Graphical Simulation of an AVR Processor and Periphery*" by Jakob Gruber. The full thesis is available at `https://github.com/schuay/bachelors_thesis`.

Chapter 2 provides a brief overview of *simavr* internals, followed by a setup guide in appendix A.

2. simavr Internals

simavr is a small cross-platform Alf and Vegard's Risc processor (AVR) simulator written with simplicity, efficiency and hackability in mind[1]. 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[2]. We will begin by examining short (but complete) demonstration application.

2.1. 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 Electrically Erasable Programmable Read-Only Memory (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;

i2c_eeprom_t ee;
```

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

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

`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 Central Processing Unit (CPU) state, the current cycle count, callbacks for various tasks, pending interrupts, and more.

`vcd_file` represents the file target for the Value Change Dump (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. Microcontroller (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 Two-Wire Interface (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 Analog-Digital Converter (ADC), Input/Output (IO) ports, timers, etc.

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

This section sets up *simavr*'s GNU Debugger (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.

2.2. 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`[3]). 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;
```

[3]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 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 no 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.

2.3. Initialization

2.3.1. `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 2.2.

`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).

2.3.2. 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 Executable and Linkable Format (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.

2.4. 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.

2.5. 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.

2.5.1. 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.

2.5.2. 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 2.2, 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.

2.6. 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`[4]).

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)[5].

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*.

2.7. 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.

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

[5] 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 2.3.1, and when GDB handler functions are called during the main loop in section 2.2. In the following, we will explain further the methods *simavr* employs to communicate with GDB and how breakpoints and data watchpoints are implemented.

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 2.2.

Watchpoints[6] 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!

2.8. Interrupt Requests (IRQs)

The Interrupt Request (IRQ)[7] 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

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

[7] 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[8]. `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 2.1, 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
```

[8] `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);
```

2.9. 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.

2.9.1. 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 2.4), 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`[9].

```
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.

2.9.2. 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:

[9] `279 = 0x136 − 0x20 + 0x01`

```c
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.

```c
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 2.9.1.

Implementation Overview

IOCTLs provide a way to trigger arbitrary functionality[10] in modules. Whenever a IOCTL is triggered by calling `avr_ioctl`, the IOCTL handler of all

[10] 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);
```

2.10. 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 2.1).

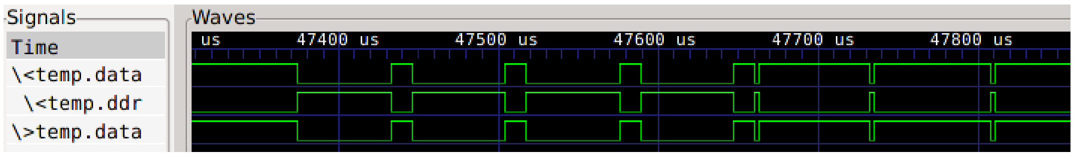

Figure 2.1.: GTKWave

The *simavr* VCD implementation uses a combination of cycle timers and IRQs to track signal[11] 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[12] 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 2.3.2, this can even be triggered from the firmware itself.

2.11. 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, Universal Asynchronous Receiver/Transmitters (UARTs), IO ports, ADCs, Serial Peripheral Interfaces (SPIs), etc ...) are defined in an internal struct using struct

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

set initialization for a terse representation. The `avr_t` initialization of the `atmega1280` therefore clocks in at only a couple of lines:

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

    .init = m1280_init,
    .reset = m1280_reset,

    .rampz = RAMPZ,
},
```

`DEFAULT_CORE`[13] 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(),
```

[13] The argument specifies the vector size.

```
        [2] = AVR_TIMER_WGM_CTC(),
        [3] = AVR_TIMER_WGM_FASTPWM8(),
        [7] = AVR_TIMER_WGM_OCPWM(),
    },
    .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.

A. Setup Guide

This section provides instructions on how to retrieve, compile and install *simavr* 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/simavr`. Stable releases are published as git repository tags (direct downloads are available at `https://github.com/buserror/simavr/tags`). To clone a local copy of the repository, run

```
git clone git://github.com/buserror/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
```