Simulavr - an AVR simulation framework

version 1.0

February 12, 2012

Contents

Copyright

Authors:

- 2001, 2002, 2003 Theodore A. Roth
- 2004 Theodore A. Roth, Klaus Rudolph
- 2005 Klaus Rudolph
- 2008 Knut Schwichtenberg
- 2009 Joel Sherrill, Onno Kortmann, Thomas Klepp
- 2010 Thomas Klepp
- 2010, 2011, 2012 Petr Hluzin and others

Introduction

The SimulAVR program is a simulator for the Atmel AVR family of microcontrollers. SimulAVR can be used either standalone or as a remote target for avr-gdb. When used in gdbserver mode, the simulator is used as a back-end so that avr-gdb can be used as a source level debugger for AVR programs.

SimulAVR started out as a C based project written by Theodore Roth. The hardware simulation part has since been completely re-written in C++. Only the instruction decoder and the avr-gdb interface are mostly copied from the original simulavr sources. This C++ based version was known as simulavrxx until it became feature compatibile with the old simulavr code, then it renamed back to simulavr.

The core of SimulAVR is functionally a library. This library is linked together with a command-line interface to create a command-line program. It is also linked together with interpreter interfaces to create libraries that can be used by a interpreter language (currently Python / TCL). In the examples directory there are examples of simulations with a graphical environment (with the Tcl/Tk interface) or writing unit tests by using Python interface. The graphic components in Tcl/Tk examples do not show any hardware / registers of the simulated CPU. It shows only external components attached to the IO-pins of the simulated CPU.

Simple example

Lets look on a simple example to demonstrate the power of simulavr.

Assume, that we have written a small program for a ATtiny2313 controller (this example code is taken from examples/simple_ex1):

```c
/* This port correponds to the "-W 0x20,-" command line option. */
#define special_output_port (*((volatile char *)0x20))

/* This port correponds to the "-R 0x22,-" command line option. */
#define special_input_port  (*((volatile char *)0x22))

/* Poll the specified string out the debug port. */
void debug_puts(const char *str) {
  const char *c;

  for(c = str; *c; c++)
    special_output_port = *c;
}

/* Main for test program.  Enter a string and echo it. */
int main() {
  volatile char in_char;

  /* Output the prompt string */
  debug_puts("\nPress any key and enter:\n> ");

  /* Input one character but since line buffered, blocks until a CR. */
  in_char = special_input_port;

  /* Print the "what you entered:" message. */
  debug_puts("\nYou entered: ");

  /* now echo the rest of the characters */
  do {
    special_output_port = in_char;
  } while((in_char = special_input_port) != '\n');

  special_output_port = '\n';
  special_output_port = '\n';
  return 0;
}
```

What does this code do:

```
#define special_output_port (*((volatile char *)0x20))
#define special_input_port  (*((volatile char *)0x22))
```

This two preprocessor lines define 2 *virtual* port register, one for reading a character, one for writing a character. Think about it as the data in/out register of a UART unit. But instead to receive/send characters by transmission line you get it from stdin/pipe or write it to stdout/pipe. This is a feature of simulavr to have a simple possibility to debug your code

```
void debug_puts(const char *str) { ... }
```

This defines a function 'debug_puts', which gets a char string and puts it out to our special "UART"

```
/* Input one character but since line buffered, blocks until a CR. */
in_char = special_input_port;
```

In this line we wait for the first character from stdin/pipe ...

```
/* now echo the rest of the characters */
do {
  special_output_port = in_char;
} while((in_char = special_input_port) != '\n');
```

and then put the received character to stdout/pipe and receive the next character until we receive a newline. After this we leave main. (not recommended for production code!)

Now we compile and link this code with avr-gcc:

```
> avr-gcc -g -O2 -mmcu=attiny2313 -o simple.elf simple.c
```

Then we start simulation:

```
> simulavr -d attiny2313 -f simple.elf -W 0x20,- -R 0x22,- -T exit

Press any key and enter:
> abcdef

You entered: abcdef

>
```

What's happen:

- we start simulation for a ATtiny2313 with our program 'simple.elf'
- we create a write pipe to stdout at register 0x20
- we create a read pipe from stdin at register 0x22
- we end simulation, if exit label is arrived (exit label will automatically inserted by avr-gcc, this is the next address after calling main function and means, that we left main function)
- our input is "abcdef" followed by enter
- we got back "abcdef"

Now lets start a debug session.

At first we have to start the simulation:

```
> simulavr -d attiny2313 -f simple.elf -g
Going to gdb...
Waiting on port 1212 for gdb client to connect...
```

It's quite similar to the call above. We tell simulavr, that we use ATtiny2313, that our program is simple.elf and - that's new - that we start a gdb session. As you can see, simulavr opens port 1212 and wait for connection from gdb.

Now we have to open a new shell and start avr-gdb:

```
> avr-gdb
GNU gdb 6.4
Copyright 2005 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i486-linux-gnu --target=avr".
(gdb)
```

(gdb) is the input prompt and avr-gdb waits now for commands:

```
(gdb) file simple.elf
Reading symbols from /home/.../simple.elf...done.
(gdb) target remote localhost:1212
Remote debugging using localhost:1212
0x00000000 in __vectors ()
(gdb) load
Loading section .text, size 0xba lma 0x0
Loading section .data, size 0x58 lma 0xba
Start address 0x0, load size 274
Transfer rate: 2192 bits in <1 sec, 137 bytes/write.
(gdb) step
Single stepping until exit from function __vectors,
which has no line number information.
0x0000001a in __trampolines_start ()
(gdb) quit
The program is running.  Exit anyway? (y or n) y
>
```

file simple.elf

  load our program into debugger

target remote localhost:1212

  now we connect us to simulavr, in shell with simulavr we can see now, that simulavr has connection to gdb: *Connection opened by host 0.0.0.0, port 33333.*

load

  now we load our program to simulavr

step

  we make here a single step, but now you're able to debug your code as you like

quit

  for now we close our debug session

After closing our debug session we have to stop simulavr by typing ^C in this shell with simulavr running. Otherwise simulavr waits for a next gdb session.

Features

What features are new:

- Run multiple AVR devices in one simulation. (only with interpreter interfaces or special application linked against simulavr library) Multiple cores can run where each has a different clock frequency.
- Connect multiple AVR core pins to other devices like LCD, LED and others. (environment)
- Connect multiple AVR cores to multiple avr-gdb instances. (each on its own socket/port number, but see first point for running multiple avr cores)
- Write simulation scripts in Tcl/Tk or Python, other languages could be added by simply adding swig scripts!
- Tracing the execution of the program, these traces support all debugging information directly from the ELF-file.
- The traces run step by step for each device so you see all actions in the multiple devices in time-correct order.
- Every interrupt call is visible.
- Interrupt statistics with latency, longest and shortest execution time and some more.
- There is a simple text based UI interface to add LCD, switches, LEDs or other components and can modify it during simulation, so there is no longer a need to enter a pin value during execution. (Tcl/Tk based)
- Execution timing should be nearly accurate, different access times for internal RAM / external RAM / EEPROM and other hardware components are simulated.
- A pseudo core hardware component is introduced to do "printf" debugging. This "device" is connected to a normal named UNIX socket so you do not have to waste a UART or other hardware in your test environment. (How?)
- ELF-file loading is supported, no objcopy needed anymore.
- Execution speed is tuned a lot, most hardware simulations are now only done if needed.
- External IO pins which are not ports are also available. (E.g. ADC7 and ADC8 on ATmega8 in TQFP package.)
- External I/O and some internal states of hardware units (link prescaler counter and interrupt states) can be dumped ot into a VCD trace to analyse I/O behaviour and timing. Or you can use it for tests.

Download

Project homepage is available at http://savannah.nongnu.org/projects/simulavr. There you'll find also a link to download area. If you want to download other versions, please use the link to download area!

Secure download

Releases are secured by gpg signatures. For every package, tarball, document, which you can download here, you'll find a signature file too. This is a cryptographic checksum over the released file and helps you to find out, if this file is unchanged by somebody unauthorized.

For this, you need a gpg installation and our gpg keyring. Download this keyring and import it to your keyring:

```
> gpg --import simulavr-keyring.gpg
```

You can list out, what's now in your keyring:

```
> gpg --list-keys
```

After you have downloaded release file (tarball, document, binary package) together with the signature file, you can verify, that your download is correct (for example, you've downloaded `simulavr-1.0.0.tar.gz` together with `simulavr-1.0.0.tar.gz.sig`):

```
> gpg --verify simulavr-1.0.0.tar.gz.sig
```

If there is no message, that file is invalid, you can use your downloaded file. (of course, you can use it also without verifying signature, but on your own risk!)

Release 1.0.0

Release date: february 12th, 2012.

Package	Description
simulavr-1.0.0.tar.gz and gpg signature	Source tarball for all systems, see build instructions for usage
manual-1.0.pdf and gpg signature	Manual in pdf format
simulavr-1.0.0-binary-linux32.tar.gz and gpg signature	Binary for linux, built on ubuntu 10.04 32bit, but should work on mostly all linux distributions. Install it somewhere you want in your home path or with root rights to sytem root to make it accessible for all users. Includes api documentation, man pages, header files and examples
simulavr-1.0.0-binary-win7-32bit.tar.gz and gpg signature	Binary for windows, built with MSys for Windows7 32bit. Install it anywhere in your system, for example in a subdirectory on programs path
simulavr-1.0-api-documentation.tar.gz and gpg signature	Doxygen API documentation in html format.
pysimulavr-1.0.0-linux32-py2.6-.egg and gpg signature	Python egg for python 2.6 on linux, install it with easy_install or pip. Should also work on python 2.4 to 2.7. Dosn't need simulavr installation. (an egg is a zip container, you can also unpack it with unzip anywhere in your python environment)

pysimulavr-1.0.0-win7-32bit-py2.6-.egg and gpg signature	Python egg for python 2.6 on win7, install it with easy_install or pip. Should also work on python 2.4 to 2.7. Dosn't need simulavr installation. (an egg is a zip container, you can also unpack it with unzip anywhere in your python environment)

Older versions

This are older versions before july 2011. The first two are from the old simulavr, written in C. The second one a earlier release of new C++ version, starting in 2005.

Version	Date	Download
0.1.2.6	05.03.2009	simulavr-0.1.2.6.tar.gz and gpg signature
0.1.2.7	03.07.2011	simulavr-0.1.2.7.tar.gz and gpg signature
0.8.006	30.07.2005	simulavrxx-0.8.006.tar.gz, gpg signature not available

More versions are available on download area on project homepage!

Usage

Invoke simulavr:

```
> simulavr {options}
```

Common options

`-v, --version`
: show the software version of simulavr

`-V, --verbose`
: output some hints to console

`-h, --help`
: show commandline help for simulavr and what devices are supported

Simulation options

`-d <device name>, --device <device name>`

tell simulavr, what type of device it has to simulate. The following devices are supported for version 1.0: (to find out, which devices are supported with your current installation, use the `--help` option)

- at90can128
- at90can32
- at90can64
- at90s4433
- at90s8515
- atmega128
- atmega1284a
- atmega16
- atmega164a
- atmega168
- atmega32
- atmega324a
- atmega328
- atmega48
- atmega644a
- atmega8
- atmega88
- attiny2313

Attention: some devices doesn't support all peripheral parts of controller. (for example CAN peripheral in at90can... devices) Ports and timer are mostly implemented.

`-f <name>, --file <name>`

load ELF-file <name> for simulation in simulated target.

`-F <value>, --cpufrequency <value>`

set the CPU frequence to <Hz>. Default is 4MHz.

`-t <file name>, --trace <file name>`

enable trace outputs into <file name>

-s, --irqstatistic
: Writes IRQ statistic to stdout at the end of simulation.

GDB options

> ***Note***
>
> Do not run simulavr with *-p*-option unattended and also not with admin rights. This could be a security hole for your system!

-g, --gdbserver
: running as avr-gdb-server

-G
: running as avr-gdb-server and write debug info for avr-gdb-connection to stdout. Use it alternative to option `-g`. **This is only useful, if you want to see, what data is sent from gdb to simulavr and back!**

-n, --nogdbwait
: do not wait for avr-gdb connection. Default is to wait for gdb connection, if option `-g` or `-G` is given.

-p <port>
: change <port> for avr-gdb server to port. Default is port 1212.

--gdb-stdin
: for use with GDB as `target remote | ./simulavr`

Control options

-m <nanoseconds>
: maximum run time of <nanoseconds>

-R <offset>,<file>, --readfrompipe <offset>,<file>
: add a special pipe register to device at IO-offset and opens <file> for reading

-T <label or address>, --terminate <label or address>
: stops simulation if PC runs on <label> or <address>. If this parameter is omitted, simulavr has to be terminated manually. For <label> you can use any label listed in the map-file of the linker - no matter if it is ever reached or not.

-W <offset>,<file>, --writetopipe <offset>,<file>
: add a special pipe register to device at IO-Offset and opens <file> for writing

-a <offset>, --writetoabort <offset>
: add a special register to device at IO-Offset which aborts simulation

-e <offset>, --writetoexit <offset>
: add a special register to device at IO-Offset which exits simulation (if you write to this IO-Offset, then the written value will be given back as exit value of the simulator!)

The commands -R / -W / -a / -e are not AVR-hardware related. Here you can link an address within the address space of the AVR to an input or output pipe. This is a simple way to create a "printf"-debugger, e.g. after leaving the debugging phase and running the AVR-Software in the simulator or to abort/exit a simulation on a specified situation inside of your program. For more details see the example in the directory examples/simple_ex1 or *here*.

VCD trace options

-o <filename|->

Writes all available VCD trace sources for a device to <filename> or to stdout, if <-> is given.

-c <trace-params>

Enable a trace dump, for valid <trace-params> see below.

Special options

-u

run with user interface for external pin handling at port 7777. This does not open any graphics but activates the interface to communicate with the TCL environment simulation.

Examples

Using the simulator with avr-gdb is very simple. Start simulavr with:

```
simulavr -g
```

Now simulavr opens a socket on port 1212. If you need another port give the port number with:

```
simulavr -p5566
```

which will start simulavr with avr-gdb socket at port 5566.

After that you can start avr-gdb or ddd with avr-gdb:

```
avr-gdb
```

or:

```
ddd --debugger avr-gdb
```

In the comandline of ddd or avr-gdb you can now enter your debug commands:

```
file a.out
target remote localhost:1212
load
step
step
....
quit
```

Attention: In the actual implementation there is a known bug: If you start in avr-gdb mode and give no file to execute `-f filename` you will run into an `"Illegal Instruction"`. The reason is that simulavr runs immediately with an empty flash. But avr-gdb is not connected and could stop the core. Solution: Please start with `simulavr -g -f <filename>`. The problem will be fixed later. It doesn't matter whether the filename of the simulavr command line is identical to the filename of avr-gdb file command. The avr-gdb downloads the file itself to the simulator. And after downloading the core of simulavr will be reset complete, so there is not a real problem.

Tracing

One of the core features is tracing one or multiple AVR cores in the simulator. To enable the trace feature you have simply to add the `-t` option to the command line. If the ELF-file you load into the simulator has debug information the trace output will also contain the label information of the ELF-file. This information is printed for all variables in flash, RAM, ext-RAM and also for all known hardware registers. Also all code labels will be written to the trace output.

What is written to trace output:

```
2000 a.out 0x0026: __do_copy_data                LDI R17, 0x00 R17=0x00
2250 a.out 0x0028: __do_copy_data+0x1            LDI R26, 0x60 R26=0x60 X=0x0060
2500 a.out 0x002a: __do_copy_data+0x2            LDI R27, 0x00 R27=0x00 X=0x0060
2750 a.out 0x002c: __do_copy_data+0x3            LDI R30, 0x22 R30=0x22 Z=0x0022
3000 a.out 0x002e: __do_copy_data+0x4            LDI R31, 0x01 R31=0x01 Z=0x0122
```

```
3250 a.out 0x0030: __do_copy_data+0x5            RJMP 38
3500 a.out 0x0038: .do_copy_data_start           CPU-waitstate
3750 a.out 0x0038: .do_copy_data_start           CPI R26, 0x60 SREG=[------Z-]
4000 a.out 0x003a: .do_copy_data_start+0x1       CPC R27, R17 SREG=[------Z-]
4250 a.out 0x003c: __SP_L__                      BRNE ->0x0032 .do_copy_data_loop
4500 a.out 0x003e: __SREG__,__SP_H__,__do_clear_bss LDI R17, 0x00 R17=0x00
4750 a.out 0x0040: __SREG__,__SP_H__,__do_clear_bss+0x1 LDI R26, 0x60 R26=0x60 X=0x0060
5000 a.out 0x0042: __SREG__,__SP_H__,__do_clear_bss+0x2 LDI R27, 0x00 R27=0x00 X=0x0060
5250 a.out 0x0044: __SREG__,__SP_H__,__do_clear_bss+0x3 RJMP 48
5500 a.out 0x0048: .do_clear_bss_start           CPU-waitstate
```

What the columns mean:

- absolute time value, it is measured in nanoseconds (ns)
- the code you simulate, normally shown as the file name of the loaded executable file. If your simulation runs multiple cores with multiple files you can see which core is stepping with which instruction.
- actual PC, meaning bytes not instructions! The original AVR documentation often writes in instructions, but here we write number of flash bytes.
- label corresponding to the address. The label is shown for all known labels from the loaded ELF-file. If multiple labels are located to one address all labels are printed. In future releases it is maybe possible to give some flags for the labels which would be printed. This is dependent on the ELF-file and BFD-library.
- after the label a potential offset to that label is printed. For example `main+0x6` which means 6 instructions after the `main` label is defined.
- The decoded AVR instruction. Keep in mind pseudo-opcodes. If you wonder why you write an assembler instruction one way and get another assembler instruction here you have to think about the Atmel AVR instruction set. Some instructions are not really available in the AVR-core. These instructions are only supported for convenience (i.e. are pseudo-ops) not actual opcodes for the hardware. For example, `CLR R16` is in the real world on the AVR-core `EOR R16,R16` which means exclusive or with itself which results also in zero.
- operands for the instruction. If the operands access memory or registers the actual values of the operands will also be shown.
    - If the operands access memory (Flash, RAM) also the labels of the accessed addresses will be written for convenience.
    - If a register is able to build a special value with 16 bits range (X,Y,Z) also the new value for this pseudo register is printed.
    - If a branch/jump instruction is decoded the branch or jump target is also decoded with the label name and absolute address also if the branch or jump is relative.
    - A special instruction @command{CPU-waitstate} will be written to the output if the core needs more then one cycle for the instruction. Sometimes a lot of wait states will be generated e.g. for eeprom access.
- if the status register is affected also the `SREG=[------Z-]` is shown.

Attention: If you want to run the simulator in connection to the avr-gdb interface and run the trace in parallel you have to keep in mind that you MUST load the file in avr-gdb and also in the simulator from command-line or script. It is not possible to transfer the symbols from the ELF-file through the avr-gdb interface. For that reason you always must give the same ELF-file for avr-gdb and for simulavr. If you load another ELF-file via the avr-gdb interface to the simulator the symbols for tracing could not be updated which means that the label information in the trace output is wrong. That is not a bug, this is related to the possibilities of the avr-gdb interface.

Graphic User Interface with TCL

To adjust reader's expectations about simulavr let's start with some design goals. The main design goals are:

- Create a framework instead of an all-purpose simulator
- Keep the simulator well structured
- Make it easy to extend this simulator
- Develop it for the needs of the developer rather than everybody future needs

To find a framework instead of an all-purpose simulator might be confusing but is the good old habit of Unix programs. Keep it simple and easy to extend. That's what can be found over here.

Next let's define what a GUI is necessary for. Showing the source code, variables and so on is done by avr-gdb and that comes with a GUI e.g. ddd. There is no need to provide an alternative. Within the examples provided together with simulavr the following graphical components are provided by the script gui.tcl:

- Digital-IO Display of the status of an port pin output as well as a mechanism to set an input value to an input pin @item Analog Input Set an analog value to a port pin
- LCD Have a 4*20 character LCD with a 4 bit data interface
- PC Keyboard Have a PC serial keyboard
- Scope This item is only mentioned here because it is available. The function is a development forecast.
- SerialRx / SerialTx Have distinct serial input and output devices

To use any of these a program providing the graphical representation of these components must run and take / provide contents via the socket 7777. Additionally each currently used instance of these components have to be registered with the simulation kernel to be updated. The current implementation adds a new graphic representation of a GUI-component whenever a new instance of the corresponding component is registered. For more details see below.

Details of the example GUI

In the following sections all currently available components defined in the script `gui.tcl` are described. The reader should be aware that `gui.tcl` is an **example**. If you don't like it feel free to change it accordingly.

UpdateControl

While processing the general registration of the GUI (-u parameter or TCL: `set UI [new_UserInterface 7777 ]`) a button is created. Pressing this button makes the button's background color change from red to green vice versa. While pressing this button values changed by the simulation are exchanged between the simulation and the GUI. Until this button pressed, any updates are ignored.

Net

Commonly spoken a Net connects a digital IO-pin of the simulated CPU with another pin like a copper wire. In the context of the GUI a Net provides the possibility to enter a value for an input pin and also shows the status of an output pin. Valid values for this GUI element are:

- H representing a "hard" high value - tied the pin directly to the supply voltage (TCL: $Pin_HIGH)
- h representing a pulled-up high - here the input is tied by a resistor to the supply (TCL: $Pin_PULLUP)
- t Tri-state this input is left open (TCL: $Pin_TRISTATE)
- l like "h" but pulled to GND (TCL: $Pin_PULLDOWN)

- L like "H" but connected to GND (TCL: $Pin_LOW)

Additionally the value "S" might appear, if there is a short circuit (TCL: $Pin_SHORTED).

For the input direction the values are selected by a radio button. The following snippet from the TCL example anacomp shows the usage of the Net component:

```
ExtPin epb $Pin_TRISTATE $ui "->B0" ".x"
Net portb
portb Add epb
portb Add [AvrDevice_GetPin $dev1 "B0"]
```

First there is an endpoint for the Net created with the instance name "epb".

- "epb" is created by calling the class ExtPin (via swig) within the simulator (see net.cpp).
- "$Pin_TRISTATE" define the level to be tri-state (no pull-up, no pull-down).
- "$ui" is the reference to the wanted GUI.
- "->B0" is the object headline / description.
- ".x" is the window reference.

Next an instance of a digital Net is created named "portb". The next two statement wire the Net, one end of the cable is connected to the graphic while the other end is connected to pin "B0" of the device "$dev1".

Each instance-name and string in the TCL script is case sensitive. CPU-Pins (e.g. "B0") always begin with a capital character. Pins names of external devices (e.g. Clock-Pin of the Keyboard) are always written in lower-case charcters ("clk"). TCL itself has some ideas of the components names. If you use lowercase characters it is mostly fine.

AnalogNet

Net and AnalogNet are at least the same. Digital Nets have potentially distinct input and output values that represent a smll number of digital states. An AnalogNet has a "continuum" of values represented by numbers in the range from 0..MAX_INT. Based on the absence of a simulated ADC this simplified analog model is sufficient but might change in the future. After entering a analog value into the AnalogNet input field a click on the update button of this graphic object forwards the analog value to the simulation:

```
ExtAnalogPin pain0 0 $ui "ain0" ".x"
Net ain0
ain0 Add pain0
ain0 Add [AvrDevice_GetPin $dev1 "D6"]
```

The parameter of ExtAnalogPin are identical to ExtPin, with the difference of the default value. Here "0" is the default value. The rest including the "Net" and "Add" commands are described above.

LCD

The LCD component simulates a simplified character LCD with a HD 44780 compatible controller. The LCD simulation is simplified for the following reasons:

- only a 4 * 20 LCD layout is available (no others like 1 * 16, ...).
- the graphic representation is character based. Display of of characters follows the rules of your display, not of the LCD character generator.
- loadable characters are not supported.
- reading of display is not supported.
- reading of busy flag does not give the current address in the lower bits.
- scrolling not supported.
- shift right / left of the display content is not supported.

- only one character set is supported - based on your diplay font.
- only the 4 bit interface is supported. At start-up the commands are interpreted as if an eight bit interface is available (one write cycle per command). After finishing the initialization switching to the four bit interface is permitted at any time.

With these limitations, one might wonder what actually is supported:

> A simple display of characters with a simplified HD 44780 interface plus some easy to implement LCD-controller commands.

The timing as described by the HD 44780 datasheet is used to set the BusyFlag. Problems detected by the LCD (such as invalid initialization, command not supported, command to early,...) are output to the standard error device. More details of the LCD specifc commands are described at the LCD example.

Keyboard

The Keyboard component simulates a simplified PC keyboard. It generates Make-Codes and Break-Codes for pressing and releasing a button of the PC's keyboard. After selecting the keyboard icon in the simulator window (gui.tcl) keys pressed and released on the PC keyboard are redirected to Keyboard simulation component. There they are transformed into a serial stream and sent synchronous with a clock signal to the AVR application. The simulation of the keyboard is simplified too. There is no communication **to** the keyboard supported. Neither reading the status nor re-/setting of the keyboard LEDs is supported. More details of the Keyboard specifc commands are described at the Keyboard example.

SerialRx / SerialTx

The SerialRx component as well as the SerialTx component simulates a serial receiver / transmitter and display. The transfer format is fixed set to 8n1 (8 Databits, No Parity, 1 Stopbit) The baud rate can be set to any "unsigned long long" value - not only to the common baud rates 9600, 19200,... By default the baud rate is set to 115.200. The graphic representation shows a display field that contains the received / entered characters. The following display translations are made for the SerialRx component: " " is displayed by "_". Characters which are not marked by the function `isprint` as printable are displayed in hex-format (e.g. 0x0d for "n").

The additional three hashed lines in the GUI shall be used for "status", "pin", "baudrate" in a future release of simulavr. The necessary data is currently not forwarded by the simulation to the GUI.

The SerialRx component provides a Pin named "rx" that has to be wired as usual. The SerialTx component provides a Pin named "tx" that has to be wired as usual. For more details of how to use the SerialRx component see the Keyboard example. A combined SerialRx / SerialTx example is added to LCD example.

Scope

The Scope does not yet have a real functioning back-end in the simulator. Before this feature was implemented completely the development was halted.

Command Line Parameter -u vs. Interpreter

Coming into touch with simulavr it might be confusing why there is a simulavr program providing a command-line switch -u and all the swig story and a interpreter program. Lets start with a closer look to the example anacomp/checkdebug.*. It's a personal preference of the reader if you look at the python or the TCL source. There is no difference in function between them. Simulavr is able to simulate the AVR silicon device as well as some external components which will be called Environment further on. Each Environment component needs a graphical representation, a registration in the simulator and a connection to one or more pins of the simulated CPU (see chapter above). To keep these tasks simple and clearly separate the graphical representation is done by the script examples/gui.tcl. This script is able only to display components and forward inputs to the simulator via socket 7777 (and currently only on the local host).

Now we should compare main.cpp of simulavr and anacomp/checkdebug.*. Both files are the "main" routines (spoken in C-language). They share major parts while other's are different. The simulator core can be understood as a library that is linked to the main to have a simulator either with the result of a command line program or with the result of an extension to an interpreter language

From the beginning of the TCL-script up to `set sc [GetSystemClock]` the script is functional identical to main.cpp with the corresponding command-line parameters set. The following line `$sc AddAsyncMember $ui` is graphic specific and registers an update button of the graphic.

The important part for understanding is, defining a NET within the simulator registers this component. Only registered components are updated by the simulator. The current implementation provides no network interface to register graphical components. Instead the swig-I/F is able to access any function of the simulator core. Here the framework character of simulavr becomes visible. Each specific simulation needs a specific main-program to display the necessary graphical components. Within a script file it is much simpler to create a case specific simulation GUI.

If there is anyone looking for a task to create an all-purpose GUI feel free to start.

Building and Installing

Note

Examples in this chapter refer to a version 1.0.0, please replace this with your current version!

Build

simulavr uses GNU auto tools. This means that, given a tarball, for version 1.0.0, for example, you should be able to use the following steps to build and install simulavr:

```
tar zxvf simulavr-1.0.0.tar.gz
cd simulavr-1.0.0
./configure <configure options>
make
make install
```

This will build `simulavr` and, if switched on by configure options (see below), some extension modules and libraries. It installs simulavr itself, libraries and some examples and the `simulavr.info` in documentation directory `$prefix/share/doc/simulavr`.

Install

If you want to install `simulavr.pdf` too, you can do that after the normal installation:

```
make install-pdf
```

To install simulavr documentation as html:

```
make install-html
```

Installing doxygen documentation is also possible, if doxygen is installed and switched on by configure option:

```
make install-doxygen
```

Same is possible for the verilog extension. avr.vpi will be installed in `$prefix/lib/ivl` if switched on by configure option:

```
make install-vpi
```

Python interface will not be installed by `make-install...`, because a right installation depends on the actual python installation. To support the installation of python module there is a `setup.py` in `src` directory:

```
cd simulavr-1.0.0/src
python setup.py install
```

If you want to create a egg-package from this python module, you have to install python's setuptools package first. Then run:

```
python setup.py build bdist_egg
```

For more possibilities on installing python interface, please see python documentation (distutils package) and documentation for setuptools python package.

Prerequsites

simulavr does rely on a few other GNU tools. In particular, it relies on libbfd from binutils, and by libbfd's dependency, it also relies on libiberty.

Here is a list of tools, which are needed by building configure script, run configure and run make:

- make (all not to old versions should work, known to work with 3.81, ubuntu 10.04)
- autoconf (version >= 2.63, known to work with 2.65, ubuntu 10.04)
- automake (version >= 1.10, known to work with 1.11.1, ubuntu 10.04)
- libtool (version >= 2.2, known to work with 2.2.6b, ubuntu 10.04)
- SWIG (version >= 1.3.18, known to work with 1.3.40, ubuntu 10.04), needed if you want to create tcl or python extensions or to run examples
- gcc (version known to work with 4.4.3, ubuntu 10.04)
- avr-gcc, avr-binutils, avr-libc (works with 4.3.4, 2.20, 1.6.7, ubuntu 10.04) for creating avr programs for examples
- tcl and tk, needed, if you want to create tcl interface
- python (version >= 2.4.0, known to work with 2.6.5, ubuntu 10.04), needed if you want to create python interface, run examples and/or `make check`

Configure options

Ideally this is all you should need to build/install simulavr. Below are some of the configure options.

`--prefix`

Use this option to specify the root directory to install simulavr to. `/usr/local` is the default.

`--with-bfd`

If configure tells you it can't find libbfd, try `--with-bfd=/your/path/`. Notice that you are expected to point to the libbfd from binutils configured for AVR.

`--with-libiberty`

In the unlikely event that your properly installed AVR binutils results in simulavr finding libbfd but not libiberty, use this option. Use of this option usually means the libiberty you are looking at did not come from the same binutils install that gave you the libbfd you have.

`--disable-tcl`

By default, the Tcl interface is enabled. However, it is possible to build a standalone simulavr executable without Tcl. When `--disable-tcl` is specified, neither the simulator shared library not the examples requiring the Tcl GUI will be built. By default, Tcl is enabled but if Tcl is not installed on your computer, Tcl will be automatically disabled.

`--with-tclconfig`

If configure tells you it can't find `tclConfig.sh`, try `--with-tclconfig=/your/path/`.

`--enable-maintainer-mode`

If specified on the configure command, the generated Makefiles will do more dependency tracking. In particular, they will check the dependencies on all automake and autoconf generated files. When not building in maintainer mode, the file `src/keytrans.h` will not be built or dependencies checked.

`--with-winsock`

Specifies, where the winsock library is located. **Only used, if you want to build simulavr for windows with MingW environment and this library cannot be found. This should not occur.**

`--with-zlib`

Specifies, where the libz library is located. Libtool want's to link against libz too, this library isn't used by simulavr. **Only used, if you want to build simulavr for windows with MingW environment and this library cannot be found. This should not occur.**

`--enable-doxygen-doc`

If Doxygen is installed, you can build too a programming documentation. If you enable this with this option, then you can build this documentation with `make doxygen-doc`. (not enabled by default)

--enable-python
: If Python is installed with a version younger than 2.1, then you can enable building the python interface. Python is also used for some tests and examples. If not enabled, (the default) then you can't run this tests and examples.

--enable-verilog
: If you have installed verilog package, then it's possible to enable building a verilog interface. (not enabled by default) See next chapter!

There are more options for running `./configure`. To find out, what's possible, see autotools documentation or try `./configure --help`.

A few words about libbfd and libiberty: simulavr dosn't use any AVR specific things from libbfd, so it should be possible to use the system libbfd (and libiberty). But I have seen cases, where building simulavr against this system libbfd was successfull and running simulavr with a AVR elf file end in a segmentation fault. Then it's necessary to use a special AVR binutils build.

Hint: where to install

I have found it useful to install my hand-configured-installed files in one area. That way I can put the AVR-tools in my path only when I'm working on AVR related work. For reference, here is how I could install AVR tools to `/home/user/install`:

```
mkdir b-binutils
tar jxvf binutils-2.19.tar.bz2
cd b-binutils
../binutils-2.19/configure --enable-install-libbfd \
   --prefix=/home/user/install --target=avr
make && make install
```

Then I configure/install simulavr as follows:

```
tar zxvf simulavr-@value{VERSION}.tar.gz
cd simulavr-@value{VERSION}
./configure --prefix=/home/user/install
make
make install
```

How to build simulavr on MingW/Windows

> ***Note***
>
> Your should have experience with shell scripts, MingW on Windows, how to configure MingW.

- Install msys and mingw on your windows box. Further you need the following packages for msys/mingw: autoconf, automake, crypt, gmp, libtool, mpfr, perl, pthreads, w32api, zlib.
- If you want to use python interface, you need to install a python package and swigwin.
- Try `autoconf --version`, if autoconf isn't found, then it could be that you can find autoconf-VVV (with VVV as autoconf version!) in your `/mingw/bin`. If so, copy autoconf-VVV to autoconf. Same procedure with automake, autoheader, autom4te, aclocal!
- Unpack simulavr package or checkout/clone a simulavr repo. If you use a simulavr distribution package (you can find configure script), then it's high recommended to remove also generated files from autoconf process, run `make clean && make distclean && ./bootstrap -c` in package root.
- Run `./bootstrap` in package root. This will (re)build configure script and also all necessary files to run configure.
- Then run configure: `./configure --with-bfd=/mingw`

- If configure was successfull, then you cann proceed with `make` and so one ...
- If you want to use python interface and you have installed Python and SWIG, then you should use the following options for configure: `./configure --with-bfd=/mingw --enable-python PYTHON_LDFLAGS="-LX:/PYPATH/libs -lpython25"` where `X:/PYPATH` is **your** path to your python installations. (e.g. where the python.exe can be found) Replace also the name of the library (here `python25`) to the right name from **your** installation, for python 2.6.x it is for example `python26` Don't use configure option `--enable-python=X:/PYPATH/python`, because there is a bug in m4 scripts.

The VPI interface to Verilog

Verilog, as a language designed for **veri**fying **log**ic allows to describe a hardware setup in a very general way. Simulators, such as Icarus Verilog can then be used to simulate this hardware setup. Tools such as `gtkwave` can be used to verify the output of a circuit by looking at the waveforms the simulation generates.

Simulavr comes with an interface to (Icarus) Verilog. If the `./configure` script finds the necessary header file for the interface, the so called VPI (Verilog Procedural Interface) to Icarus Verilog will be build. The result of this is a file called `avr.vpi`. This file, in essence a shared library, can then be used as an externally loaded module after compilation:

```
$ iverilog [...]          # compile verilog .v into .vvp

$ vvp -M<path-to-avr.vpi> -mavr [...] # run compiled verilog
                                      # with additional
                                      # avr.vpi module
```

In principle, it would also be possible to implement the AVR completely in verilog (and there are several existing models, see e.g. opencores.org), but this would result in decreased performance and duplicated effort, as not only the core needs to be implemented, but also the complex on-board periphery.

Usage

The Verilog interface comes with glue code on the verilog side, for which the main file is `avr.v` in `src/verilog`. This is a thin wrapper in Verilog around the exported methods from the core of Simulavr, consisting of the `AVRCORE` module encapsulating one AVR core and `avr_pin` for I/O through any AVR pin. On top of this, files named `avr_*.v` exist in the same directory which contain verilog modules reflecting particular AVR models from Simulavr. The modules in these files are meant to be the interface to be used to connect to simulavr by the user, they have a very simple signature:

```
module AVRxyz(CLK, port1, port2, ...);
```

where `port1`, `port2`, ... are simple arrays of `inout` wires representing the various ports of the selected AVR. Note that the width of the arrays as visible from the Verilog side is always eight; this does not mean that all bits are connected on the simulavr side!

Clock generation and distribution to the AVR cores is done from the verilog side. Simply connect a clock source with the preferred frequency to the CLK input of the AVR code.

The more complete, low level interface to simulavr in `avr.vpi` can be accessed directly. For documentation of the available functions, see either `src/vpi.cpp` or look into the implementation of the high level modules in `avr_*.v`.

Example iverilog command line

A simple run with the `avr.vpi` interface could look like this:

```
$ iverilog -s test -v -I. $(AVRS)/avr.v $(AVRS)/avr_ATtiny15.v \
    $(AVRS)/avr_ATtiny2313.v -o test.vvp
```

Here for a model having both an ATtiny15 and an ATtiny2313 in the simulation, and the top module `test` and the environment variable `$AVRS` pointing to the right directory.

A set of a few simple examples has been put into the `verilog/examples` subdirectory of the Simulavr source distribution. This directory also contains a `Makefile` which can be used as an example of command sequences for compiling verilog, running it and producing `.vcd` output files to be viewed with `gtkwave`.

Bugs and particularities

- No problems have been found when instantiating multiple AVR instances inside verilog.

- Analog pins have not been tested and will probably need some changes in the verilog-side wrapper code.

Examples

Simulavr is designed to interact in a few different ways. These examples briefly explain the examples that can be found in the source distribution's `examples` directory.

There are examples, which use Tcl/Tk. **For that you must also install Itcl package for your Tcl.** It will be used in all examples with Tcl and a Tk GUI! Over that you can find also examples for python interface and for the verilog module.

The anacomp example is all we have started with. Anacomp brings up an Itcl based GUI which shows two analog input simulations, a comparison output value, and a toggle button on bottom. After changing the inputs, hit the corresponding update to clock the simulation to respond to the changed inputs.

The avr-gdb session for me requires a "load" before hitting "continue", which actually starts the simulation.

It is strongly recommended to implement own simulation scripts very closely to the examples. Usage of a different name than `.x` for the grahic frame need changes of gui.tcl as well as some simulavr sources. So stay better close to the example.

TCL Anacomp Example

This is Klaus' very nice original example simulation.

After performing the build, go to the `examples/anacomp` directory and try `make do` (without gdb) or `make dogdb`.

Python Example

There is a file `README` in `examples/python` path, which describes examples there. You can try it with `make run_example`, this will run all available examples together. Or try `make example1` till `make example4` to run each example alone.

Simple Example

This sample uses only simulavr to execute a hacked AVR program. I say "hacked" because is shows using 3 simulator features that provide input, output and simulation termination based on "magic" port access and reaching a particular symbol. It is only really useful for getting your feet wet with simulavr, it is not a great example of how to use simulavr. It is thought to be useful enough to the absolute newbie to get you started though.

After performing the build, go to the `examples/simple_ex1` directory and try `make run_sim`. Notice the use of -W, -R and -T flags.

And again you can try `make do`, which uses Tcl interface and a Tcl script to make the simulation. Results are the same as in `make run_sim`!

LCD and SerialRx, SerialTx Example

This example is based on Klaus' Anacomp Example and uses the avr-libc example stdiodemo to display characters on the LCD.

After performing the build, go to the `examples/stdiodemo` directory and try `./checkdebug.tcl`. The following commands are taken from the LCD-specific `examples/stdiodemo/checkdebug.tcl` script:

```
Lcd mylcd $ui "lcd0" ".x"
sc AddAsyncMember  mylcd
```

The first command creates a LCD instance `mylcd` with the name `lcd0` The second command adds the LCD instance to the simulavr timer subsystem as an asynchronous member. Asynchronous Timer objects are updated every 1ns - which means every iteration in the simulavr main-loop. All timing is done internally in the `lcd.cpp`. The rest of this simulation script is the normal business create Nets for each LCD pin, wire the Nets to the CPU pins. The stdiodemo application contains a serial receiver and transmitter part to receive commands and interprete it and if possible prints it on the LCD or sends a response to the serial receiver. Transmitter and receiver application are implemented by polling opposite to the Keyboard example. The components used for the SerialRx/Tx are described below. Together with the comments in the script you should be able to understand what happens. Please mind the different names for the functions SetBaudRate and GetPin for SerialRx and SerialTx! Not optimal but that's it at the moment...

And you can try `make do` or `make dogdb`.

Keyboard and SerialRx Example

This example is based on Klaus' Anacomp Example and uses the Atmel application note AVR313 to convert the incomming data from the keyboard into a serial ASCII stream and sends this stream via the serial interface. Atmel's C-Code is ported to a current avr-gcc (4.x) and a Mega128. For this example only the serial transmitter is used. Atmel implemented the serial transmitter as interrupt controlled application, opposite to the serial transmitter / receiver of the LCD example. Here a polled solution is implemented.

After performing the build, go to the `examples/atmel-key` directory and try `./checkdebug.tcl`. This example by itself is good to show how the GUI needs to be setup to make the Keyboard component work. The output of the keyboard is displayed into SerialRx component. Let's look into the simulation script to point out some details:

Keyboard::

```
Keyboard kbd $ui "kbd1" ".x"
Keyboard_SetClockFreq kbd 40000
sc Add kbd
```

These three commands create a Keyboard instance `kbd` with the name `"kbd1"`. For this instance the clock timing is set to 40000ns. simulavr internal timing for any asynchronous activity are multiples of 1ns. The third command adds the keyboard instance to the simulavr timer. The rest of the commands in `examples/atmel-key/checkdebug.tcl` is the normal for this simmulation. Create a CPU AtMega128 with 4MHz clock. Create indicators for the digital pins (not necessary but good looking). Create a Net for each signal - here Clock(key_clk), Data(key_data), Run-LED(key_runLED), Test-Pin(key_TestPin), and Serial Output(key_txD0). Wire the pins Net specific. Run-LED and Test-Pin are specific to the Atmel AP-Note AVR313. The output of the keyboard converter is send to the serial interface. Based on an "implementation speciality" of simulavr a serial output must be either set by the AVR program to output or a Pin with a Pull-Up activated has to be wired.

SerialRx::

```
SerialRx mysrx $ui "serialRx0" ".x"
SerialRxBasic_SetBaudRate mysrx 19200
```

These two commands create a SerialRx instance `mysrx` with the name `"serialRx0"`. For this instance the baud rate is set to 19200. This SerialRx is wired to the controller pin, a display pin by the following commands:

```
ExtPin exttxD0 $Pin_PULLUP $ui "txD0" ".x"
key_txD0 Add [AvrDevice_GetPin $dev1 "E1"]
key_txD0 Add exttxD0
key_txD0 Add [SerialRxBasic_GetPin mysrx "rx"]
```

The last command ExtPin shows an alternative default value for txD0-Pin. Here it is pulled high - what is identical of adding any pull-up resistor to the device pin - no matter which resistor value is used.

While creating this example, simulavr helped to find the bugs left in the AP-Note.

Platform Related Notes

Gentoo GNU/Linux distribution

To install the AVR cross compiler toolchain, try: `crossdev -t AVR`

Of course you may need to `emerge crossdev` first.

There have been some problems reported with crossdev. Have a look to build scripts for Linux provided by the AVR-Freaks. This script has problems with the bfd-libs but the rest builds easily.

No ebuild for simulavr exists yet, but for me, the standard ./configure && make works. Let me know if this is not the case for you.

Limitations

Please be aware, that this chapter is version dependent so compare document version and software version to ensure both fit together.

Overall Limitations

This chapters describes an overview of system wide limitations for simulavr. Specific limitations see below.

- The documentation of the simulator provides a wide field of activities to be carried out.
- Currently only some AVR-CPUs are simulated. While several of the Mega-CPUs can be simulated by the available Mega128 no Tiny is around. If your Mega-CPU is not available recompile your project and use a Mega128 CPU for simulation. This works only if your destination CPU and the Mega128 share **identical** components. Comparing of the names e.g. "Timer0" is not sufficient - you need to compare each component for identical function!
- simulavr simulates an AVR-CPU and a small amount of environment, like IO-network, some analogue components as well as SPI, ... There is neither a fully description for the environment available nor comprehensive examples around.
- simulavr does not verify if the current instruction is available for the selected CPU (e.g. MUL for Tiny,...)
- The current version of simulavr is not validated against the avr-gcc regression tests.
- AVR XMEGA are completely **not yet** simulated by simulavr.

CPU Limitations

This chapters describes an overview of limitations for simulavr. Specific limitations see below. This chapter focuses only on the Mega128 CPU.

The following hardware is **not** simulated by simulavr:

- TWI/I2C Serial Interface
- Analog to Digital Converter Subsystem (Really? What about src/hwad.h file?)
- Analog comparator in some devices (ATmega16/ATmega32)
- Boot Loader Support (incl. Fuses)
- Timer 1 external crystal support (for Real Time Clock)
- Watchdog Timer
- Sleep-command
- Reset-pin is not available
- With activating the Tx-Pin of an UART the DDR-Register is not set properly to output. Workaround: Set the Pin's default value to PULLUP. While the Pin behaves as Open Colletor (pulls down only) the pull-up "resistor" lets the system run as it should.

There are 64kByte of external memory automatically attached to the Mega128.

While Atmel changed some function details of the EEPROM, Watchdog Timer, Timer Subsystem, ADC, and USART / USI these subsystems have identical names but different functions. Therefore adding a new CPU to simulavr might end in reprogramming a subsystem!

Help Wanted

Send bugs and comments on simulavr mailinglist simulavr-devel@nongnu.org.

Project homepage is available at http://savannah.nongnu.org/projects/simulavr.

License

Simulavr is released under GPLv2, this is a copy of the license text (you can find this copy too on http://www.gnu.org/licenses/old-licenses/gpl-2.0.html):