Simulavr - an AVR simulation framework

version 1.1

January 05, 2020

Contents

Introduction

The SimulAVR program is a simulator for the Atmel AVR family of microcontrollers. Atmel was taken over by Microchip in the year 2016.

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 how to write for example 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.

Copyright

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 out into a VCD trace to analyze I/O behavior and timing. Or you can use it for tests.

Simple example

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

Example code

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

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

/* This port corresponds 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
```

Run the example

We start simulation with:

```
> 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”

Run it with gdb

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 1212.

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.

Usage

Invoke simulavr:

```
> simulavr {options}
```

Common options

-V, --version
print out version and exit immediately

-h, --help
print this help

-v, --verbose
output some hints to console

Simulation options

-d <name>, **--device** <name>
simulate device <name>, see below for simulated devices

-f <name>, **--file** <name>
load elf-file <name> for simulation in simulated target

-F <Hz>, **--cpufrequency** <Hz>
set the cpu frequency to <Hz>

-t <file>, **--trace** <file>
enable trace outputs to <file>

-s, --irqstatistic
prints statistic informations about irq usage after simulation is stopped

-C <name>, **--core-dump** <name>
dump a core memory image <name> to file on exit

GDB options

-g, --gdbserver
listen for GDB connection on TCP port defined by -p

-G, --gdb-debug
listen for GDB connection and write debug info

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

-p <port>
use <port> for gdb server (default is port 1212)

-n, --nogdbwait

do not wait for gdb connection

Control options

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

-W <offset_file>, **--writetopipe** <offset_file>
add a special pipe register to device at IO-Offset and opens file for writing, write argument as 'offset,file', file can be '-' to write to standard output

-R <offset_file>, **--readfrompipe** <offset_file>
add a special pipe register to device at IO-Offset and opens file for reading, write argument as 'offset,file', file can be '-' to write to standard input

-a <offset>, **--writetoabort** <offset>
add a special register at IO-offset which aborts simulator run

-e <offset>, **--writetoexit** <offset>
add a special register at IO-offset which exits simulator run

-T <label>, **--terminate** <label>
stops simulation if PC runs on <label>, <label> is a text label or a address

-B <label>, **--breakpoint** <label>
same as -T for backward compatibility

-M
disable messages for bad I/O and memory references

-l <number>, **--linestotrace** <number>
maximum number of lines in each trace file. 0 means endless. Attention: if you use gdb & trace, please use always 0!

VCD trace options

-c <tracing-option>
Enables a tracer with a set of options. The format for <tracing-option> is: <tracer>[:further-options ...]

-o <trace-value-file>
Specifies a file into which all available trace value names will be written. Use '-' for standard output

TCL ui option

-u
run with user interface for external pin handling at port 7777

Supported devices

- at90can128
- at90can32
- at90can64
- at90s4433
- at90s8515
- atmega128
- atmega1280
- atmega1284
- atmega1284a
- atmega1284p
- atmega16
- atmega164
- atmega164a
- atmega164p
- atmega168
- atmega2560
- atmega32
- atmega324
- atmega324a
- atmega324p
- atmega328
- atmega48
- atmega64
- atmega640
- atmega644
- atmega644a
- atmega644p
- atmega8
- atmega88
- attiny2313
- attiny25
- attiny45
- attiny85

Hints

`Option -d`

The option -d is mandatory, if an elf file isn't given or elf file dosn't contain device signature. To put device signature to elf file you can insert the following line to your source code (but only once!):

```
#include <avr/signature.h>
```

If this option is given and device signature will be found in elf file, then the given signature by device name is compared to signature in elf file. If this isn't equal, then simulavr stops with an error message.

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

`GDB option -g`

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

`GDB option -G`

Use it as a alternative to option `-g`. This is only useful, if you want to see, what data is sent from gdb to simulavr and back!

`Options -R / -W / -a / -e`

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.

Example usage

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 `E0R 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.

Download

Project homepage is available at https://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

> **Note**
>
> Replace X.Y.Z with the real release, this is for example 1.1.0.

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-X.Y.Z.tar.gz` together with `simulavr-X.Y.Z.tar.gz.sig`):

```
> gpg --verify simulavr-X.Y.Z.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 files

Debian packages

Package	Release date
simulavr-vpi_1.1.0_amd64.deb (42K) and gpg signature	Jan 05 2020, 17:07
simulavr-tcl_1.1.0_amd64.deb (238K) and gpg signature	Jan 05 2020, 17:07
simulavr-dev_1.1.0_amd64.deb (103K) and gpg signature	Jan 05 2020, 17:07
simulavr_1.1.0_amd64.deb (95K) and gpg signature	Jan 05 2020, 17:04
python3-simulavr_1.1.0_amd64.deb (611K) and gpg signature	Jan 05 2020, 17:04
libsim_1.1.0_amd64.deb (364K) and gpg signature	Jan 05 2020, 17:03

Documentation

Package	Release date
manual-1.1.0-rc1.pdf (1M) and gpg signature	Dec 04 2019, 16:59
manual-1.1.0.pdf (1M) and gpg signature	Jan 05 2020, 17:03
simulavr-1.1.0-api-documentation.tar.gz (4M) and gpg signature	Jan 05 2020, 17:05
simulavr-1.1.0-html-manual.tar.gz (457K) and gpg signature	Jan 05 2020, 17:06
manual-1.0.pdf (1M) and gpg signature	Feb 12 2012, 16:29
simulavr-1.0-api-documentation.tar.gz (194M) and gpg signature	Feb 12 2012, 19:37

Tarball's

Package	Release date
simulavr-1.0.0.tar.gz (989K) and gpg signature	Feb 12 2012, 19:41
simulavrxx-0.8.006.tar.gz (541K) and gpg signature	Jul 30 2005, 03:46
simulavrxx-0.8.005.tar.gz (511K) and gpg signature	Feb 09 2005, 03:29
simulavrxx-0.8.004.tar.gz (415K) and gpg signature	Nov 11 2004, 06:41
simulavrxx-0.8.003.tar.gz (320K) and gpg signature	Aug 25 2004, 22:47
simulavr-0.1.2.1.tar.bz2 (351K) and gpg signature	Jan 19 2004, 00:08
simulavr-0.1.2.2.tar.gz (446K) and gpg signature	Feb 16 2005, 23:42
simulavr-0.1.2.3.tar.gz (468K) and gpg signature	Jan 07 2008, 21:50
simulavr-0.1.2.4.tar.gz (468K) and gpg signature	Mar 09 2008, 20:39
simulavr-0.1.2.5.tar.gz (469K) and gpg signature	Mar 16 2008, 21:31
simulavr-0.1.2.6.tar.gz (469K) and gpg signature	Mar 05 2009, 09:30
simulavr-0.1.2.7.tar.gz (484K) and gpg signature	Jul 03 2011, 09:22
simulavr-0.1.2.tar.bz2 (338K) and gpg signature	Jan 18 2004, 05:24

Old binary packages

Package	Release date
simulavr-1.0.0-binary-linux32.tar.gz (16M) and gpg signature	Feb 12 2012, 19:40
simulavr-1.0.0-binary-win7-32bit.tar.gz (6M) and gpg signature	Feb 13 2012, 19:38

More other files and versions are available on download area on project homepage!

Building and Installing

> **Note**
>
> Examples in this chapter refer to a version X.Y.Z, please replace this with your current version, for example 1.1.0!

This chapter describes, how you can build and install simulavr.

> **Attention!**
>
> The build scripts with cmake aren't prepared to work with Windows or Mac OS, even if cmake itself is able to create build configuration for this platforms!

Prerequisites

To build simulavr only you need:

- cmake (at least version 3.5 or newer)
- make
- gcc (at least a version, which supports c++11)
- git
- python (at least version 3.5 or newer)

If you want to build the python extension, you need also python development files (e.g. in a debian or ubuntu platform you have to install python3-dev package) and swig. (at least version 3.x or newer)

If you want to build tcl extension, you need:

- tclsh
- tcl development files
- swig (at least version 3.x or newer)

To build the verilog extension you have to install iverilog tool. (included vvp)

For running the complete regression test you need also:

- avr-gcc (on debian it's package gcc-avr)
- avr-libc
- valgrind (optional, check for overwriting memory and memory leaks)

Building the documentation suite needs also:

- gzip
- help2man
- makeinfo
- doxygen (optional, for api documentation)
- dot (optional, for api documentation)
- python module sphinx (at least version 2.2 for web site and manual)
- python module rst2pdf (at least version 0.94 for web site and manual)

- python module beautifulsoup4 (short bs4, for web site and manual)
- python module requests (for web site and manual)

Building debian packages need also:

- dpkg
- strip
- fakeroot

Build

simulavr uses cmake and git. This means that you should be able to use the following steps to build simulavr:

```
> git clone https://git.savannah.nongnu.org/git/simulavr.git
> cd simulavr
> make build
> make check
```

This will build `simulavr` and check, if the main function is ok.

In the root directory of repository you can find a `Makefile` as wrapper for calling cmake and control configuration. If you call:

```
> make
```

you'll get a help, what targets are available and what the targets will do.

Targets

`make cfgclean`
: Removes the build directory. The make wrapper configures cmake for "out of source" build. E.g. nearly all build artifacts are below build directory. This removes also the complete cmake configuration.

`make clean`
: As usual, it removes all build artifacts but let cmake configuration untouched.

`make build`
: Build simulavr and all extensions as configured.

`make check`
: Run regression test.

`make doc`
: Build documentation, e.g. man page, info page, sphinx documentation (manual and web site).

`make doxygen`
: Build api documentation.

`make debian`
: Build debian packages for simulavr tool and lib and all configured extensions.

To switch on/off configuration options, you have to call make with the configuration target:

```
> make debug    # build program, libs and extensions with debug information
> make no-debug # build without debug information
```

Other config targets are:

- python (build/do not build python module)

- tcl (build/do not build tcl module)
- verilog (build/do not build verilog extension)
- valgrind (run the gtest regression test additional with valgrind for memory check)

With target all and simple you can switch on and off all configuration options together. There is one special target, taken over from old automake build system:

```
> make keytrans
```

This create keytrans.h for tcl extension.

Debian packages

To support install on debian systems (e.g. debian and ubuntu and derivates) it's possible to build debian packages. Go to root directory and call:

```
> make debian
```

This will produce the following packages (PLAT represents the build platform):

`libsim_X.Y.Z_PLAT.deb`
: The simulavr lib itself.

`simulavr_X.Y.Z_PLAT.deb`
: The simulavr tool, depends on libsim package.

`simulavr-dev_X.Y.Z_PLAT.deb`
: Contains header and other files, to build applications against libsim, depends on libsim package.

`simulavr-vpi_X.Y.Z_PLAT.deb`
: Verilog extension, depends on libsim package.

`python3-simulavr_X.Y.Z_PLAT.deb`
: Python3 module, static linked, so no other dependencies needed.

`simulavr-tcl_X.Y.Z_PLAT.deb`
: Tcl extension, depends on libsim package.

On a debian system you could install the simulavr tool itself then (you need root permission):

```
cd <root_of_repo>/build/debian
apt install libsim_X.Y.Z_PLAT.deb # first install lib to fullfil dependencies
apt install simulavr_X.Y.Z_PLAT.deb
```

After that you can check, if simulavr is ready:

```
simulavr -h
```

Install

If you want to make a installation on a system and not use debian packages, (maybe you system doesn't support debian packages) then you can call the install target from cmake itself after the normal build is finished:

```
cd <root_of_repo>
cmake --build build --target progdoc
cmake --build build --target install
```

After that you'll find the install tree in the build/install directory. Copy this to the destination, as you want. As example (you need root permission):

```
cd <root_of_repo>/build/install
cp -r usr /
```

Build using docker

If docker is installed, then you can create docker images to build simulavr in a stable and defined environment and independent from what's installed on your computer.

Step 1: create a docker image

There are docker scripts and a small script to create a image:

```
cd <root_of_repo>/docker
./mkimage buildscripts/bionic.build.Dockerfile
```

This will create a docker image with name "simulavrbuild" and version "bionic". (e.g. Ubuntu 18:04)You can check it with:

```
docker images
```

There you should find the new created image in the list. Attention: because of the installed packages in this image, the resulting image size is about 1G! There is also a docker script for ubuntu 16:04, called "xenial.build.Dockerfile".

Now, after the image is ready, you can create the build container:

```
docker run -it -u buildbudy --name <container_name> simulavrbuild:bionic
buildbudy@e1694c8b9f26:/
```

You stay now in your new created container, the second line is the bash prompt. You have created a container with name "<container_name>" (replace this to a name, which is useful for you!) from the new built image and running with user "buildbudy". If you omit the "-u" option, then you'll be root inside your container. Because this could be dangerous in some cases and normally not needed, it's better to run with a normal user with normal privileges also inside the container!

You can now leave the container on every time with "exit" command and come back with:

```
docker start -i <container_name>
```

In this container you can now start the build:

```
cd # to come to buildbudy's home directory
git clone -b master https://git.savannah.nongnu.org/git/simulavr.git simulavr
cd simulavr
make all # to switch on all config options exept debug option
make build
```

After that is finished, you've build sucessful simulavr and all extensions.

If you don't want to clone from official repository, as described before, you could also clone from a local repository (maybe where you've written some new code). In this case the container have to created with a extra option:

```
docker run -it -u buildbudy --name <container_name> -v /local/path:/repo simulavrbuild:bioni
```

Replace (as before) “<container_name> with a useful name and /local/path with a path, where your local repository is hold. For example /home/user/simulavr is the repository, then ypu could give “-v /home/user:/repo”. Inside the container you will find then a new directory /repo, where you see your repository. Then the clone command could be:

```
cd
git clone -b your_branch /repo/simulavr simulavr_local
```

To get out your build artefacts, you can user “docker cp” command (and after you leaved the container):

```
docker cp <container_name>:/home/buildbudy/simulavr/build/app/simulavr .
docker cp <container_name>:/home/buildbudy/simulavr/build/libsim/libsim.so .
```

This copies the simulavr tool itself and the simulavr library, which is needed, simulavr to run.

More 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.

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. See here for a more detailed information.

Go to `examples/simple_ex1` and build there the avr program:

```
> avr-gcc -g -O2 -mmcu=at90s8515 -o fred.elf fred.c
```

After performing the build, run it. Notice the use of -W, -R and -T flags:

```
> simulavr -d at90s8515 -f fred.elf -W 0x20,- -R 0x22,- -T exit
```

Then type something followed by `ENTER`. Your typed string will be printed out. For explanation of simulavr options see usage.

TCL Examples

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.

To use the TCL examples you have to prepare 2 files in :file:`examples` directory!

In `examples` directory you can find 2 files: `simulavr.tcl.sample` and `gui.tcl.sample` Copy this files to `simulavr.tcl` and `gui.tcl` and edit the copied files.

Edit simulavr.tcl:

You need before 3 file paths, the path from tclsh tool, the path from wish tool (both from the TCL/TK suite) and the path, where your TCL extention for simulavr is installed. You can find out the paths for tclsh and wish with the following commands:

```
> which tclsh # prints out the path for tclsh, if found
> which wish # prints out the path for wish, if found
```

Later you need also the ddd debugger graphical interface to avr-gdb:

```
> which ddd # prints out the path for ddd, if found
```

For the path, where libsimulavr.so (the TCL extension) is installed, you have to seek, where it's installed. This is system dependend! On debian systems and if you have made the install with debian packages, it would be: `/usr/lib/simulavr/libsimulavr.so`, that means, that you have to choose `/usr/lib/simulavr`.

Then edit the beginning of file `simulavr.tcl`:

```
#! @TCL_SHELL@
# configuration area
set WishCMD "@TCL_WISH@"
set buildPrefix "@prefix@"
# end configuration area

#
#  This demonstrates how to implement a custom Tcl "main" for the
...
```

Replace "@TCL_SHELL@" with the path for `tclsh`, "@TCL_WISH@" with the path for `wish` and "@prefix@" with the path for `libsimulavr.so`.

And the same for the beginnig of `gui.tcl`:

```
#! @TCL_WISH@
# configuration area
set installPrefix "@prefix@"
# end configuration area

package require Itcl
namespace import itcl::*
...
```

Now you're prepared to run the TCL examples.

TCL Anacomp Example

> **Note**
>
> You must have installed the ITCL extension for TCL/TK to run this example!

This is Klaus' very nice original example simulation.

To build the avr program go to `examples/anacomp` directory:

```
> avr-gcc -g -O2 -mmcu=at90s4433 -o main.elf main.c
```

After performing the build you can start the simulation:

```
> ../simulavr.tcl -d at90s4433 -f main.elf -u -s anacomp.tcl
```

This starts a simple gui and enables the user to enter analog values (0.0 .. 5.0) in the input fields. After entering a new analog value in ain0 or ain1, you must press the update button!

At this point, the output of the analog comparator will be used to determine the output state of the "->B0" field. "->B0" displays the state of the Port B 0 pin. Its value is determined by the following logic:

- if ain0 > ain1 B0 = H(igh)

- if ain0 == ain1 B0 = L(ow)
- if ain0 < ain1 B0 = L(ow)

And not to forget, you can run this simulation together with gdb debugger or also ddd:

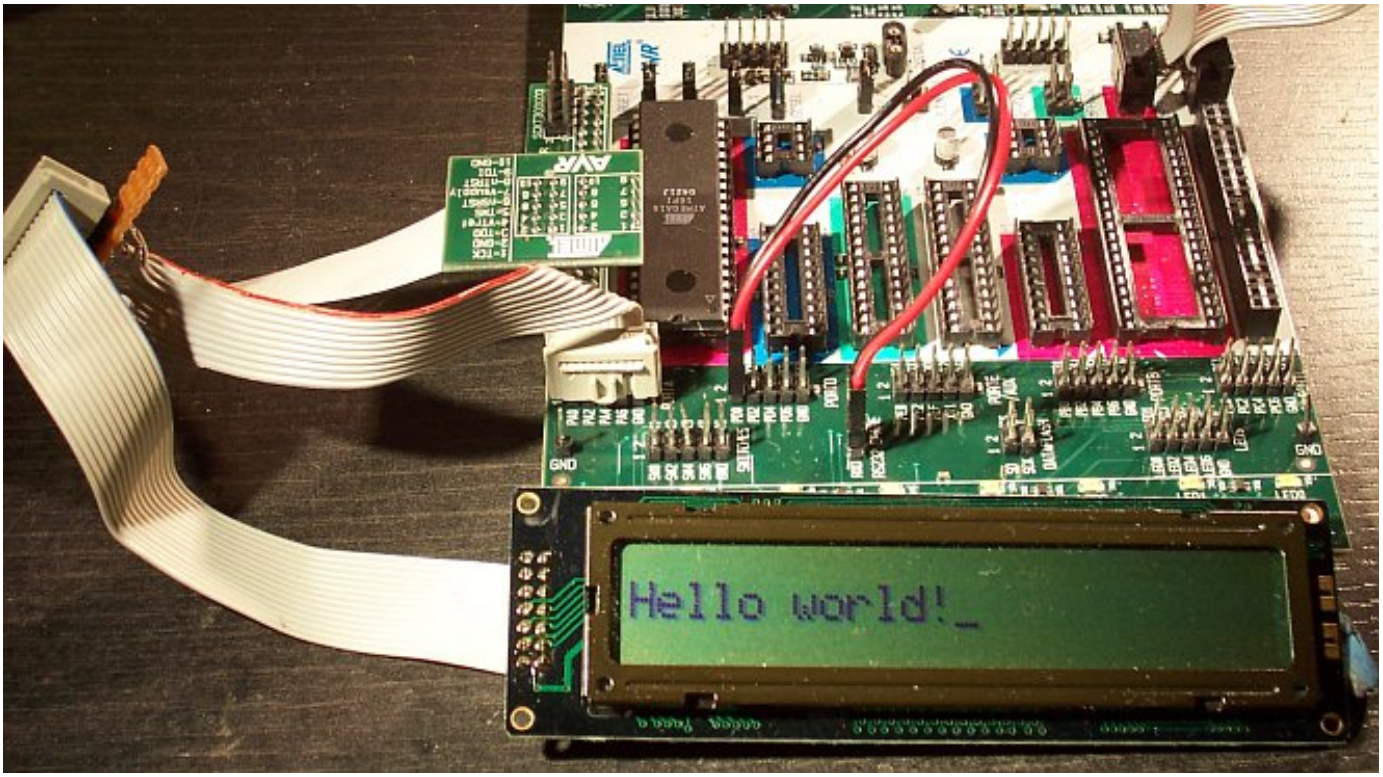
```
> tclsh ../simulavr.tcl -d at90s4433 -f main.elf -u -s anacomp.tcl -g
```

The one and only difference to the simulation command before is the “-g” option!

LCD and SerialRx, SerialTx Example

> **Note**
>
> You must have installed the ITCL extension for TCL/TK to and ddd to run this example!

This example is written by Knut Schwichtenberg and based on Klaus’ Anacomp Example and uses the avr-libc example stdiodemo to display characters on the LCD.



First we build the avr program. Go to `examples/stdiodemo` directory:

```
> avr-gcc -g -mmcu=atmega128 -Os -Wall -DF_CPU=3686400UL \
    -o stdiodemo.elf hd44780.c lcd.c stdiodemo.c uart.c -lm
```

Then we have to prepare a tcl file too (in the same way as described before), copy and edit checkdebug.tcl:

```
#! @TCL_WISH@
# configuration area
set WishCMD "@TCL_WISH@"
set DDDCMD "@DDD@"
set installPrefix "@prefix@"
```

```
# end configuration area

#
#############################################################################
#
# LCD and Serial IO example for simulavrxx
...
```

And now you can start:

```
> wish ./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.c`. 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 interpret 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 to simulate it with gdb or ddd:

```
> tclsh ../simulavr.tcl -d atmega128 -f stdiodemo.elf -u -F 271 \
          -s stdiodemo.tcl -g
```

Keyboard and SerialRx Example

> **Note**
>
> You must have installed the ITCL extension for TCL/TK to run this example!

This example is written by Knut Schwichtenberg and based on Klaus' Anacomp Example and uses the Atmel application note AVR313 to convert the incoming 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.

To build the avr program go to `examples/atmel_key` directory:

```
> avr-gcc -g -mmcu=atmega128 -I. -DF_CPU=4000000UL -Os \
    -funsigned-char -funsigned-bitfields -fpack-struct \
    -fshort-enums -Wall -Wstrict-prototypes -o atmel_key.elf \
    kb.c main.c serial.c StdDefs.c -lm
```

After performing the build you can start the simulation:

```
> ../simulavr.tcl -d atmega128 -f atmel_key.elf -u -F 250 \
    -s atmel_key.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.

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.

atmega128_timer example

This example uses Timer 2 on the ATMega128 to generate a periodic interrupt. It prints 1 to 500 as the number of ticks increases. It's not a dedicated tcl example, but shows, that you can use :file:`simulavr.tcl` in the same way as the original simulavr program.

To build the avr program go to `examples/atmega128_timer` directory:

```
> avr-gcc -g -mmcu=atmega128 -DF_CPU=4000000UL -Os \
    -o timer.elf main.c debugio.c
```

After performing the build you can start the simulation with simulavr:

```
> simulavr -d atmega128 -f timer.elf -W 0x20,- -R 0x22,- -T exit
```

or with `simulavr.tcl`:

```
> tclsh ../simulavr.tcl -d atmega128 -f timer.elf -W 0x20,- -R 0x22,- -T exit
```

atmega48 example

Demonstrates the ATMega48 and following Stimulation classes:

- HWAdmux - with additional pin inputs for not GPIO port support.
- SpiSink - monitors the /SS, SCLK and MISO pins and prints each byte to stdout.
- SpiSource - drives the /SS, SCLK and MOSI pins with data from the spidata file.
- PinMonitor - monitors Port A Bit 0 and prints changes in its binary status to stdout.
- AdcPin - Stimulates Port F Bit 0 with the values contained in the anadata{1,2,3} files.

The AVR program alternately (every other byte) echoes the byte received on the SPI or the ADCH value read from a recenet A/D converter, to the SPI. Also, the value from the A/D converter rotates through the values at the pins PC5, ADC6, and ADC7.

The spidata file contains an HDLC encoded stream of mostly flags that was used in my project at work. (We're running a form of PPP/HDLC over SPI.)

The format of the spidata file consists of comments (lines that start with a '#') and data lines. Each data line consists of 3 values.

- First Value - the value (0 or non-zero)of /SS
- Second Value - the value (0 or non-zero) of SCLK
- Third Value - the value (0 or non-zero) of MOSI

When the SpiSource program stimulator reaches the end-of-file, it rewinds and repeats ad-nauseum.

The anadata{1,2,3} files contains analog data that that is read by the AdcPin class and written to the Port C Bit 5, ADC6 and ADC7 analog inputs of the ATMega48.

The format of the anadata{1,2,3} files consists of comments (lines that start with a '#' character) and analog input lines. Each input line consists of 2 values separated by whitespace.

- First Value - number of nano-seconds before the next value is read and applied to the analog input.
- Second Value - signed integer "analogValue" to be applied to the analog input.

To try it:

Step 1:

Build the AVR test program in `examples/atmega48` directory:

```
> avr-gcc -g -mmcu=atmega48 -Os -o atmega48.elf main.cpp
```

Step 2:

Prepare `check.tcl` in the same way as in other TCL examples before:

```
#! @TCL_WISH@
# configuration area
set installPrefix "@prefix@"
# end configuration area

#load the avr-simulator package
load ${installPrefix}/libsimulavr.so
```

Replace the pathes for `wish` and the install path for libsimulavr.so

Step 3:

Run the test TCL script from this directory:

```
> wish check.tcl
```

Step 4:

Marvel at the stdio activity.

Step 5:

Try modifying the spidata file and see the results.

What you'll see on stdout:

```
Note: Comments added on the right.

spisink: /SS negated  ; SPI /SS goes HIGH (printed by SpiSink)
spisink: /SS asserted ; SPI /SS goes LOW (printed by SpiSink)
spisink: 0x7E         ; echoed HDLC Data from AVR on SPI MISO
spisink: 0x66         ; Analog Data from AVR PC5 as decoded on SPI MISO
spisink: 0x02         ; echoed HDLC Data from AVR on SPI MISO
spisink: 0x33         ; Analog Data from AVR ADC6 as decoded on SPI MISO
spisink: 0xD3         ; echoed HDLC Data from AVR on SPI MISO
spisink: 0x28         ; Analog Data from AVR ADC7 as decoded on SPI MISO
spisink: 0x7E         ; echoed HDLC Data from AVR on SPI MISO
spisink: 0x23         ; Analog Data from AVR PC5 as decoded on SPI MISO
...
...
spisink: 0x7E         ; echoed HDLC Data from AVR on SPI MISO
spisink: 0x04         ; Analog Data from AVR {PC5,ADC6, ADC7} as decoded on SPI MISO
PORTB0: NEGATE        ; Port B Bit 0 (interrupt output) set high by AVR (printed by PinMonit
...
...
```

feedback example

This example illustrates how one can provide a program external to the simulated AVR which provides "feedback" to the simulated program. A feedback program can interact with the AVR hosted program just like devices would in the "real world."

This example is certainly a primitive example of this but it illustrates the principle. The application writes the following lines to UART0:

```
hello world #1
hello world #2
hello world #3
hello world #1
```

The initial input value of ADC0 is 0. When the feedback modules sees 1, 2 or 3, it changes the "voltage" on ADC0. The debug output expected is:

```
ADC0=10 expect 10
ADC0=20 expect 20
ADC0=30 expect 30
ADC0=10 expect 10
```

To build the avr program go to `examples/feedback` directory:

```
> avr-gcc -g -mmcu=atmega128 -DF_CPU=4000000UL -Os \
    -o feedback.elf main.c debugio.c uart.c adc.c
```

Prepare `simfeedback.tcl` in the same way as in other TCL examples before:

```
#!  @TCL_SHELL@

package require Itcl
namespace import itcl::*
```

Replace the path for `tclsh`. If this is done, you can start the simulation with `simulavr.tcl`:

```
> tclsh ../simulavr.tcl -d atmega128 -f feedback.elf -s feedback.tcl \
   -W 0x20,/dev/stderr -R 0x22,- -F 4000000 -T exit -S simfeedback.tcl
```

Python examples

This are some examples to demonstrate usage of pysimulavr. You need to build python simulavr module named `pysimulavr`. Maybe you have installed the pysimulavr debian package, then you can test it:

```
> python3
>>> import pysimulavr
>>>
```

If this works without a error message, then the python module is ready.

If not, e.g. you want to run the tests against the module, you have to build just before, then you can give the environment variable PYTHONPATH with the path to `_pysimulavr.so` and `pysimulavr.py` in the same line just before the python command:

```
> PYTHONPATH=<path-to-_pysimulavr.so> python3 <other-options>
```

All python examples are to find on `examples/python` directory. Go there and try it:

Simple timer unittest

We have to build the avr program for the simulation:

```
> avr-gcc -g -mmcu=atmega128 -O2 -o example.elf example.c
```

The program is a modified variant from tcl example atmega128_timer before. The test is written as a unittest. You can start it by:

```
> python3 example.py atmega128:example.elf
```

As result you should see something like this:

```
test_01 (__main__.TestBaseClass)
just run 3000 ns + 250 ns ... ok
test_02 (__main__.TestBaseClass)
just run 2 steps ... ok
test_03 (__main__.TestBaseClass)
check PC and PC size ... ok
test_04 (__main__.TestBaseClass)
check address of data symbols ... ok
test_05 (__main__.TestBaseClass)
access to data by symbol ... ok
test_06 (__main__.TestBaseClass)
write access to data by symbol ... ok
test_07 (__main__.TestBaseClass)
```

```
test toggle output pin ... ok
test_08 (__main__.TestBaseClass)
work with breakpoints ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.842s

OK
```

So you can see, how easy it's to write unittests for simulavr or also for your avr code. But you can use pysimulavr also for other things, look at example.py how to use pysimulavr.

Connect pins and change state

Shows the usage of Pin and Net. A net connect pins together. Change the output state of one pin will result in changing the input state of the other pins. This can be used as starting point to understand usage of SetOutState/SetInState methods of Pin class and how it works. This is not a real simulation. It demonstrates to use Pin and Net class without a simulation target. You can start it by:

```
> python3 example_pin.py
```

You see the following:

```
set vcc=5.00V ...

create 2 pins ...
  pin1: (char)pin='L', (bool)pin=0, pin.GetAnalogValue(vcc)=2.75V
  pin2: (char)pin='t', (bool)pin=1, pin.GetAnalogValue(vcc)=2.75V

create net ...
  add pin1 to net:
<pin1 change: in=L/0.00V, out=L/0.00V>
  add pin2 to net:
<pin1 change: in=L/0.00V, out=L/0.00V> <pin2 change: in=L/0.00V, out=t/0.00V>
  pin1: (char)pin='L', (bool)pin=0, pin.GetAnalogValue(vcc)=0.00V
  pin2: (char)pin='t', (bool)pin=0, pin.GetAnalogValue(vcc)=0.00V

set pin2 output to PULLUP:
<pin1 change: in=L/0.00V, out=L/0.00V> <pin2 change: in=L/0.00V, out=h/0.00V>
  pin1: (char)pin='L', (bool)pin=0, pin.GetAnalogValue(vcc)=0.00V
  pin2: (char)pin='h', (bool)pin=0, pin.GetAnalogValue(vcc)=0.00V

set pin1 output to HIGH:
<pin1 change: in=H/5.00V, out=H/5.00V> <pin2 change: in=H/5.00V, out=h/5.00V>
  pin1: (char)pin='H', (bool)pin=1, pin.GetAnalogValue(vcc)=5.00V
  pin2: (char)pin='h', (bool)pin=1, pin.GetAnalogValue(vcc)=5.00V

set pin2 output to TRISTATE:
<pin1 change: in=H/5.00V, out=H/5.00V> <pin2 change: in=H/5.00V, out=t/5.00V>
  pin1: (char)pin='H', (bool)pin=1, pin.GetAnalogValue(vcc)=5.00V
  pin2: (char)pin='t', (bool)pin=1, pin.GetAnalogValue(vcc)=5.00V

set pin1 output to TRISTATE:
<pin1 change: in=t/2.75V, out=t/2.75V> <pin2 change: in=t/2.75V, out=t/2.75V>
  pin1: (char)pin='t', (bool)pin=1, pin.GetAnalogValue(vcc)=2.75V
  pin2: (char)pin='t', (bool)pin=1, pin.GetAnalogValue(vcc)=2.75V

set pin2 output to LOW:
```

```
<pin1 change: in=L/0.00V, out=t/0.00V> <pin2 change: in=L/0.00V, out=L/0.00V>
  pin1: (char)pin='t', (bool)pin=0, pin.GetAnalogValue(vcc)=0.00V
  pin2: (char)pin='L', (bool)pin=0, pin.GetAnalogValue(vcc)=0.00V
```

How to control pins

This is a more complex example. It demonstrates, how you can simply watch for pin output changes and how you could inject external pin changes to the simulator to stimulate your program functionality. We build at first the avr program for the simulation:

```
> avr-gcc -g -mmcu=atmega128 -O2 -o example_io.elf example_io.c
```

The program is a modified variant from tcl example atmega128_timer before.

In this simulation we have a external connection to pin A0, A1 and A7 from port A and set the state of pin A1 and A7 to low or high at a defined simulation time. And we can see, when and how the state of this pin is changed.

You can start the simulation by:

```
> python3 example_io.py atmega128:example_io.elf
```

As result you should see something like this:

```
simulation start: (t=0µs)
simulation end: (t=15000µs)
pin A0
  change to 't' at 0µs (dt=0µs)
  change to 'L' at 17µs (dt=17µs)
  change to 'H' at 2032µs (dt=2015µs)
  change to 'L' at 4032µs (dt=2000µs)
  change to 'H' at 6036µs (dt=2005µs)
  change to 'L' at 8035µs (dt=1999µs)
  change to 'H' at 10034µs (dt=1999µs)
  change to 'L' at 12033µs (dt=1999µs)
  change to 'H' at 14037µs (dt=2004µs)
pin A1
  change to 'H' at 0µs (dt=0µs)
  change to 'L' at 7000µs (dt=7000µs)
  change to 'H' at 14000µs (dt=7000µs)
pin A7
  change to 'H' at 0µs (dt=0µs)
  change to 'L' at 12000µs (dt=12000µs)
value 'timer2_ticks'=7
value 'port_val'=0x7e
value 'port_cnt'=3
```

How to get a more detailed view

This example is closed to the example before. ex_pinout.c initialise timer2 in CTC mode for a period of 2ms on 4MHz clock frequency. Example output shows the toggle of pin A0. But we will also write a VCD dump. If you have installed `gtkwave` you can open this VCD dump file `ex_pinout.vcd` with `gtkwave`. So you can compare time written out by this example with the results shown in `gtkwave`. The signal IRQ.VECTOR9 in VCD dump shows when and how long the ISR was running! First build the avr program:

```
> avr-gcc -g -mmcu=atmega128 -O2 -o ex_pinout.elf ex_pinout.c
```

Then start it by:

```
> python3 ex_pinout.py atmega128:ex_pinout.elf
```

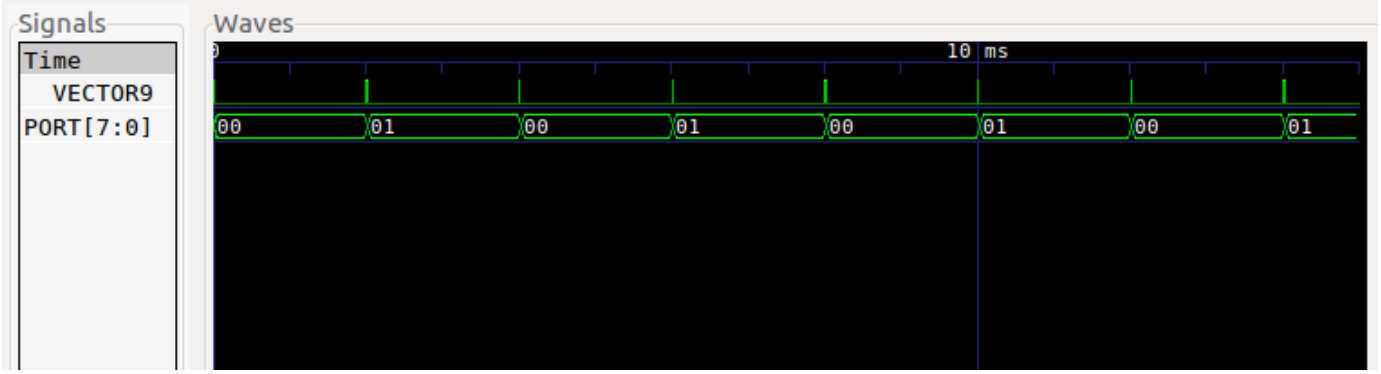
Output is:

```
port A.0 set to 't' (t=0ns)
port A.0 set to 't' (t=0ns)
simulation start: (t=0ns)
port A.0 set to 'L' (t=10750ns)
port A.0 set to 'L' (t=11000ns)
port A.0 set to 'H' (t=2017750ns)
port A.0 set to 'L' (t=4018000ns)
port A.0 set to 'H' (t=6018500ns)
port A.0 set to 'L' (t=8015750ns)
port A.0 set to 'H' (t=10016250ns)
port A.0 set to 'L' (t=12016500ns)
port A.0 set to 'H' (t=14017000ns)
simulation end: (t=15000000ns)
value 'timer2_ticks'=7
```

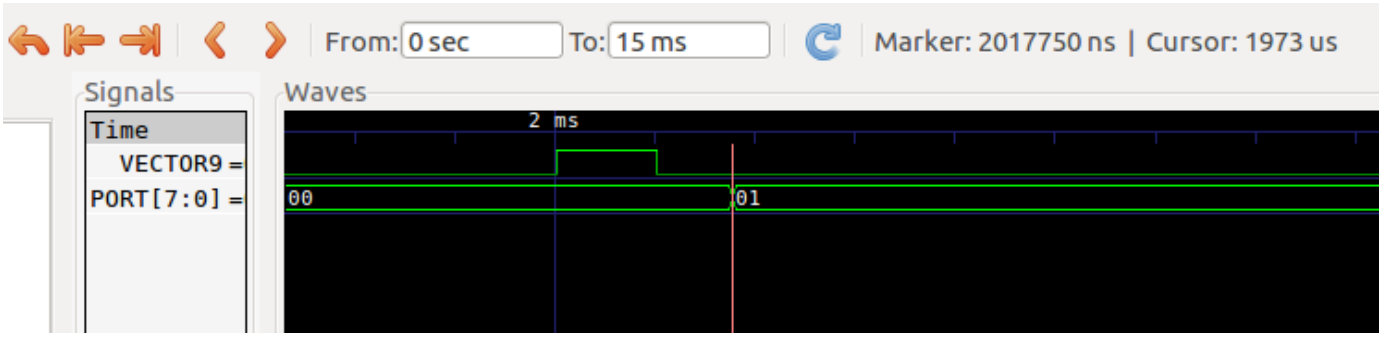
And now (if you have installed `gtkwave`) you can view the traced waveforms:

```
> gtkwave -a ex_pinout.sav ex_pinout.vcd
```

The full view:



Let's look on a detail. You can see, how long the interrupt procedure was running and when the port value was changed. Compare it with the print out on standard out before! (in picture the value for "Marker:")



Multicore example

This example demonstrates using python interface for multicore simulation. We simulate 2 ATmega16 cores:

```
+----------+            +----------+
| Core A   | PB3     PD2 | Core B   |
| 4MHz     |------>------| 10MHz    |
|          |             |          |
+----------+            +----------+
```

Core A runs as a 250Hz clock generator on pin B3. B3 from core A is conected with pin D2 on core B. Core B counts now all rising edges on pin D2 and measures the time distance between 2 events with timer T0.

This example shows:

- how to use python interface
- how it is possible to run a multicore simulation, in this example also with different clock sources for the cores
- how to connect pins between cores
- how to access global variables, how to get address for a global variable and how to read RAM values from a address

Build the 2 avr programs:

```
> avr-gcc -g -mmcu=atmega16 -O2 -DDUAL_A=1 -o multicore_a.elf multicore.c
> avr-gcc -g -mmcu=atmega16 -O2 -DDUAL_B=1 -o multicore_b.elf multicore.c
```

And run the simulation:

```
> python3 multicore.py
```

Resulting output should then look like:

```
multicore example:
  create core A ...
  create core B ...
  connect core A with core B ...
  core B: address(cnt_irq)=0x61
  core B: address(cnt_res)=0x61
  run simulation ...
  t= 4ms, cnt_irq=1, cnt_res= 78
  t= 8ms, cnt_irq=2, cnt_res=156
  t=20ms, cnt_irq=5, cnt_res=157
  t=32ms, cnt_irq=8, cnt_res=156
```

ADC example

A example to simulate analog input and how to simulate adc conversion. Build avr program and run the simulation:

```
> avr-gcc -g -mmcu=atmega16 -O2 -o adc.elf adc.c
> python3 adc.py atmega16:adc.elf
```

The output shows:

```
before simulation start:
  value 'adc_value'=43690 (before init)
  aref set to 2.5V
  a0 set to 0.3V, this will expect an converted adc int value=122
simulation start: (t=0ns)
```

```
run till main function ...
simulation main entrance: (t=24250ns)
  value 'adc_value'=5555 (after init)
simulation break: (t=144250ns)
  value 'conversions'=1
  value 'adc_value'=122 (simulation break)
simulation end: (t=474250ns)
  value 'conversions'=6
  value 'adc_value'=122 (simulation end)
```

Verilog examples

To use this examples you have to build simulavr together with the verilog extension. See here how to make it. You can find the example files in `examples/verilog` directory. Further, if you want to see the waveform you need the `gtkwave` program. It's a program to display digital waveforms.

baretest example

First compile and link avr program:

```
> avr-gcc -mmcu=at90s4433 -Os -o toggle.elf toggle.c
```

Then compile and run the verilog source file:

```
> iverilog baretest.v -s test -v avr.v -o baretest.vvp
> vvp -M<path-to-avr.vpi-directory> -mavr baretest.vvp
```

Replace <path-to-avr.vpi-directory> to the directory, where your `avr.vpi` is situated. (could be, for example, in `<root-of-repository>/build/libsim`) This will create a file `baretest.vcd`. And if you now start `gtkwave`, you can see the result:

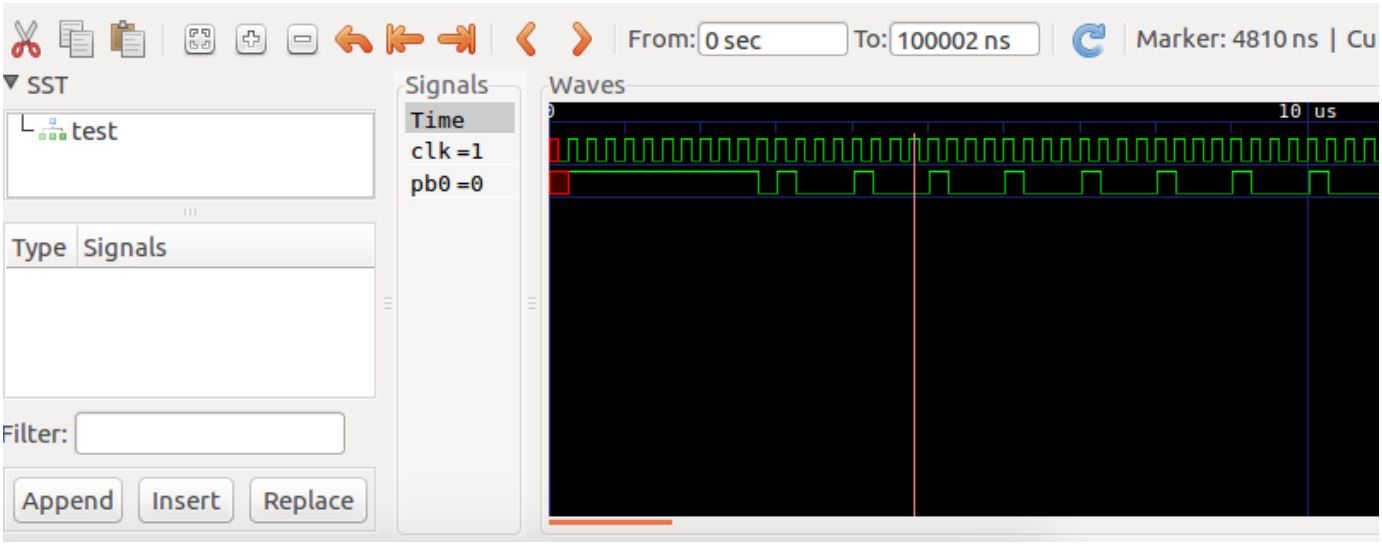
```
> gtkwave -a baretest.sav baretest.vcd
```

What this example do?

This is the code:

```
int main() {
  DDRB = 1;
  while(1) {
        PORTB = 1;
        PORTB = 0;
  }
}
```

It sets port B pin 0 to output and start a endless loop toggeling pin 0 at port B. And the result is:



loop example

Steps are the same as before for baretest example:

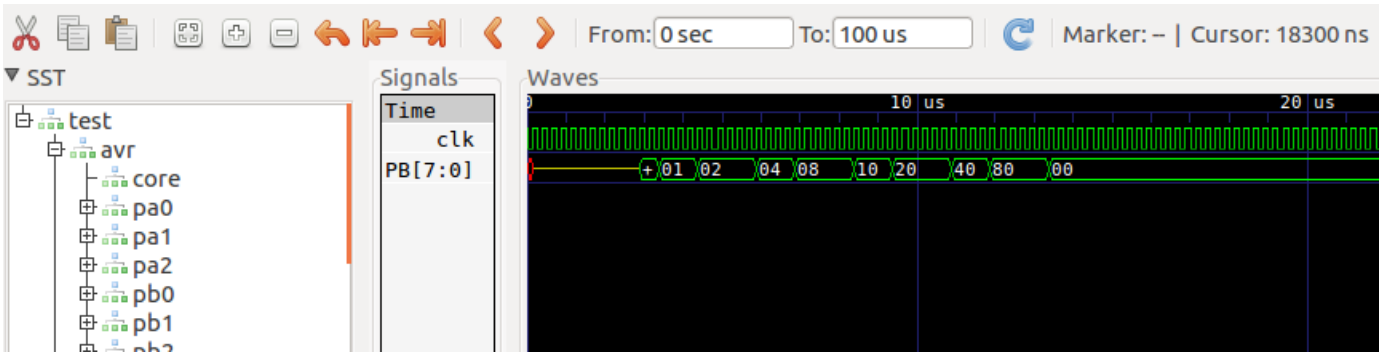
```
> avr-gcc -mmcu=attiny2313 -Os -o loop.elf loop.c
> iverilog loop.v -s test -v avr.v avr_ATtiny2313.v -o loop.vvp
> vvp -M<path-to-avr.vpi-directory> -mavr loop.vvp
```

The code is similar to toggle.c but with a twist:

```
int main() {
  DDRB = 0xff;
  PORTB = 1;
  while(1) {
        PORTB = PINB << 1;
  }
}
```

Lets see the result:

```
> gtkwave -a loop.sav loop.vcd
```

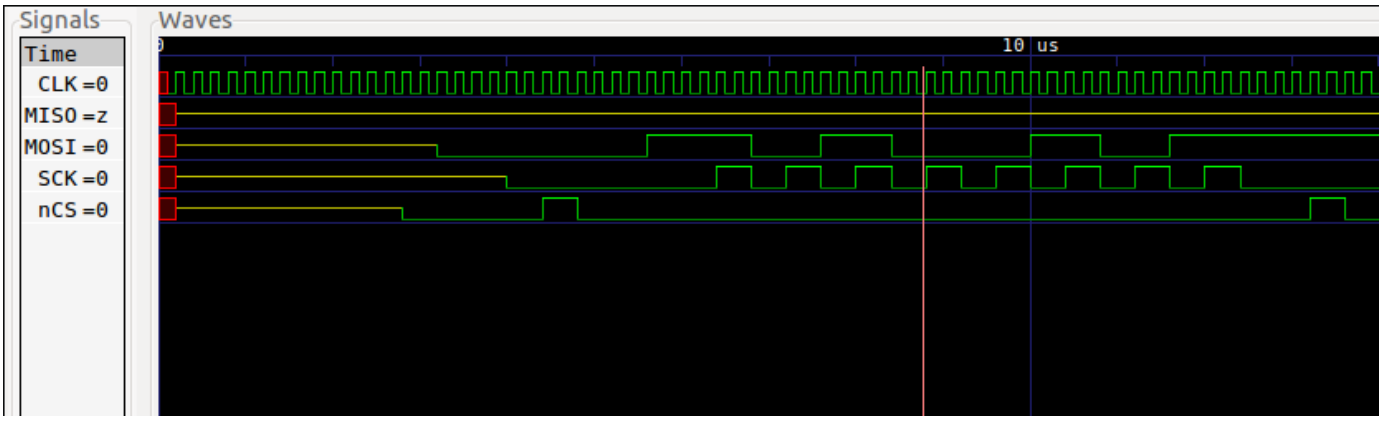


spi waveform examples

A more complicated example: send data via spi:

```
> avr-gcc -mmcu=atmega8 -Os -o spi-waveforms.elf spi-waveforms.c
> iverilog spi-waveforms.v -s test -v avr.v avr_ATmega8.v -o spi-waveforms.vvp
> vvp -M<path-to-avr.vpi-directory> -mavr spi-waveforms.vvp
> gtkwave -a spi-waveforms.sav spi-waveforms.vcd
```

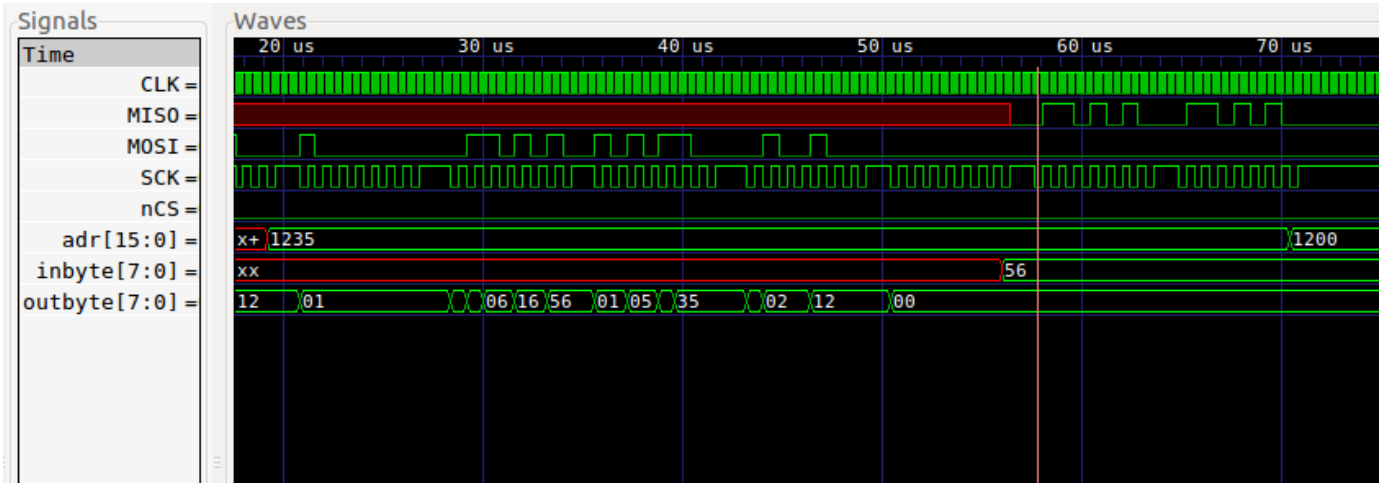
And we can see the spi signals (just the first byte sequence) in gtkwave:



And a second example where data will be send out from controller and received by controller:

```
> avr-gcc -mmcu=atmega8 -Os -o spi.elf spi.c
> iverilog spi.v -s test -v avr.v avr_ATmega8.v -o spi.vvp
> vvp -M<path-to-avr.vpi-directory> -mavr spi.vvp
> gtkwave -a spi.sav spi.vcd
```

And the simulation result:



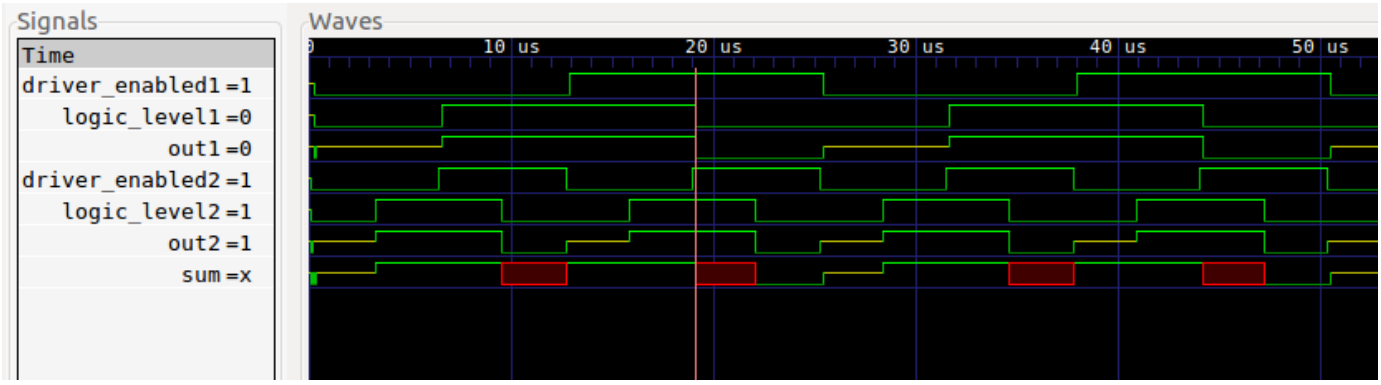
vst example

The example shows how two cores can be instantiated. Both cores are driven with different clocks:

```
> avr-gcc -mmcu=atmega32 -Os -o vst.elf vst.cpp
> iverilog vst.v -s test -v avr.v avr_ATmega32.v -o vst.vvp
> vvp -M<path-to-avr.vpi-directory> -mavr vst.vvp
> gtkwave -a vst.sav vst.vcd
```

There is a wire “out” which is connected to pins of both devices.

What the graph shows:



`driver_enabled`X (X is 1 or 2 for core 1 or core 2) shows the pin from the port driven by coreX - this represents DDR bit. `logic_level`X represents the setting of PORT bit. `out`X represents the resulting signal from above values.

Because there are 2 devices driving the `sum` wire, we get the following results:

- if both core have the driver disabled and none has pull up enabled, the result is “x”
- if only one core has the driver enabled, the wire `sum` is the value of “out” of the driver.
- if two cores have the driver enabled and both “out” signals are the same, `sum` shows the same level as “out”.
- if both cores drive the signal but with different level, the result on `sum` is “x” (short circuit!)

If the avr reads from the pins (mirrored in the signals mirrorX) the read value is “1” if the wire is in “1”, “z” or “x”! There is no definition of “z” or “x” so we simply use “1”. A logic “0” is read as “0”.

spc example

A last verilog example show also the use of 2 cores with 2 different device types and also different clocks:

```
> # create right-unit.elf
> avr-gcc -c -Wa,-gstabs -x assembler-with-cpp -o right-unit.o right-unit.s
> avr-gcc -c -Wa,-gstabs -x assembler-with-cpp -o singlepincomm.o singmepincomm.s
> avr-ld -e _start -o right-unit.elf right-unit.o singlepincomm.o
> # create left-unit.elf
> avr-gcc -mmcu=attiny2313 -Os -o left-unit.elf left-unit.c csinglepincomm.c
> # compile verilog source and run it
> iverilog spc.v -s test -v avr.v avr_ATtiny2313.v avr_ATtiny25.v -o spc.vvp
> vvp -M<path-to-avr.vpi-directory> -mavr spc.vvp
> # show result
> #gtkwave -a spc.sav spc.vcd
```

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.

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.

The VPI interface to Verilog

Verilog, as a language designed for verifying logic 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.

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 not all AVR-CPUs are simulated. There are many ATMega and some ATTiny CPU's implemented. If your CPU is not available recompile your project and use (for example) 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 https://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 https://www.gnu.org/licenses/old-licenses/gpl-2.0.html):

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.
```

```
  The precise terms and conditions for copying, distribution and
modification follow.

                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)
```

```
These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
```

```
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
```

may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission. For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this. Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

                            NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program. It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least

```
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.
```