Simulavr

A simulator for the Atmel AVR family of microcontrollers.
For simulavr version 0.1.2.5, 18 January 2004.

by Theodore A. Roth

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

Table of Contents

1 Introduction: What is simulavr? 1

2 Invoking Simulavr 2
2.1 Aliasing 2
2.2 Options 2

3 Using with GDB 4
3.1 GDB Hints 5
3.2 Building GDB for AVR 6

4 Display Coprocesses 8

5 Simulavr Internals 9

6 Reporting Bugs 10

Concept Index 11

1 Introduction: What is simulavr?

> It's just a model.
>
> — Monty Python

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

The official website for Simulavr is http://savannah.nongnu.org/projects/simulavr/.

Because it is protected by the GNU General Public License, users are free to share and change it.

Simulavr was written by Theodore A. Roth

2 Invoking Simulavr

The format for running the simulavr program is:

```
simulavr options ... [flash_image]
```

If the optional '`flash_image`' file is supplied, it will be loaded into the flash program memory space of the virtual device.

2.1 Aliasing

On most systems, if the simulavr executable is renamed to the name of an available device, it can be started without specifying the device type. The easiest way to achieve this is to create symbolic links for all the supported devices which point to the simulavr executable. For instance, this command will create a sym link for the at90s8515 device on a Unix system:

```
ln -s simulavr at90s8515
```

Once the links have been created, the following two commands are equivalent:

```
simulavr -d at90s8515 myprog.bin
at90s8515 myprog.bin
```

2.2 Options

simulavr supports the following options:

`--help`
`-h` Print an informative help message describing the options and available device types, then exit.

`--debug`
`-D` Print assembly instruction mnemonics and program counter ('`PC`') to output as device program is running.

`--version`
`-v` Print out the version number and exit.

`--gdbserver`
`-g` Run as a gdbserver process.

`--gdb-debug`
`-G` Print out messages for debugging the gdb remote serial protocol interface.

`--port <port>`
`-p` Listen for gdb connection on TCP port. If not specified, a default will be used. Run '`simulavr --help`' to see what the default is. This option is ignored if the '`--gdbserver`' is not specified.

`--device <dev>`
`-d` Specify device type. The device types available for use with a specific version of simulavr can be obtained using the '`--list-devices`' option.

`--eeprom-image <img>`
`-e` Specify an optional eeprom image file to be loaded into the device's eeprom memory space.

`--eeprom-type <type>`
`-E` Specify the type of the eeprom image file. If not specified, the default is binary.

`--flash-type <type>`
`-F` Specify the type of the flash image file. If not specified, the default is binary.

`--list-devices`
`-L` Prints a list of supported devices to stdout and exits.

`--disp-prog <prog>`
`-P` Specify a program to be used to display register and memory information in real time as a child process. The display program can also be specified by setting the `SIM_DISP_PROG` environment variable.

`--without-xterm`
`-X` Don't start display coprocess program in an xterm. This is useful if the display coprocess supplies it's own window for input and output, such as a process which uses a GUI.

`--core-dump`
`-C` Dump a core memory image to file on exit. This isn't as useful as it sounds. The display coprocess mechanism is much more informative.

`--clock-freq <freq>`
`-c` Set the simulated mcu clock freqency in Hz.

`--breakpoint <addr>`
`-B` Set a breakpoint at <*addr*>. Note that the break address is interpreted as a byte address instead of a word address. This makes it easier on the user since binutils, gcc and gdb all work in terms of byte addresses. The address can be specified in any base (decimal, hexidecimal, octal, etc).

3 Using with GDB

If you want to use gdb as a source-level debugger with simulavr running as a remote target, start simulavr with the '`--gdbserver`' or '`-g`' option. This will put simulavr into gdbserver mode. simulavr will then act as a TCP server program on the localhost listening for a connection from gdb.

Once simulavr has accepted a connection from gdb, the two programs communicate via gdb's remote serial protocol (see section "Protocol" in *Debugging with GDB*).

Here's how you would start up simulavr in gdbserver mode:

```
$ simulavr -d at90s8515 -g
```

Here's a sample gdb session showing what to do on the gdb side to get gdb to talk to simulavr:

```
This GDB was configured as "--host=i686-pc-linux-gnu --target=avr".
(gdb) file demo_kr.elf
Reading symbols from demo_kr.elf...done.
(gdb) target remote localhost:1212
Remote debugging using localhost:1212
0x0 in .__start_of_init__ ()
(gdb) load
Loading section .text, size 0x76 lma 0x0
Start address 0x0 , load size 118
Transfer rate: 944 bits in <1 sec, 29 bytes/write.
(gdb) break main
Breakpoint 1 at 0x6e: file demo_kr.c, line 17.
(gdb) continue
Continuing.

Breakpoint 1, main () at demo_kr.c:17
17          sbi(DDRC, (
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Notice that simulavr knew nothing about the program to debug when it was started. Gdb was told which file to debug with the '`file`' command. After gdb has read in the program and connected to simulavr, the program's instructions are downloaded into the simulator via the '`load`' command. The '`load`' command is not necessary if simulavr already has the program loaded into it's flash memory area. It is ok to issue multiple '`load`' commands.

Also, notice that no '`run`' command was given to gdb. Gdb assumes that the simulator has started and is ready to continue. Giving gdb the '`run`' command, will cause it to stop the current debug session and start a new one, which is not likely to be what you want to do.

When specifying the remote target to connect to, it is sufficient to write "target remote :1212" instead of "target remote localhost:1212".

Hitting *CTRL-c* in gdb can be used to interrupt the simulator while it is processing instructions and return control back to gdb. This is most useful when gdb is waiting for a response from the simulator and the program running in the simulator is in an infinite loop.

Issuing a '`signal SIGxxx`' command from gdb will send the signal to the simulator via a *continue with signal* packet. The simulator will process and interpret the signal, but will not pass it on to the AVR program running in the simulator since it really makes no sense to do so. In some circumstances, it may make sense to use the gdb signal mechanism as a way to initiate some sort of external stimulus to be passed on to the virtual hardware system of the simulator. Signals from gdb which are processed have the following meanings:

`SIGHUP` Initiate a reset of the simulator. (Simulates a hardware reset).

3.1 GDB Hints

Since debugging an AVR program with gdb requires gdb to connect to a remote target (either simulavr or some other debugging tool, such as avarice), a series of commands must be issued every time gdb is started. The easiest way around this is to put the commands into a '`.gdbinit`' file in the project directory. The following example is from a '`.gdbinit`' which I use for many projects.

```
## Print out structures in a sane way

echo (gdb) set print pretty
set print pretty

## Use this for debugging the remote protocol. (Don't use unless
## debugging simulavr or avr-gdb)

#echo (gdb) set debug remote 1\n
#set debug remote 1

## If you don't want specify the program to debug when invoking gdb,
## you can tell gdb to read it in here. The file should be an elf file
## compiled with debugging information (-g for C files and -gstabs for
## asm files).

#echo (gdb) file myprog.elf\n
#file myprog.elf

## Connect to the remote target via a TCP socket on host:port.

echo (gdb) target remote localhost:1212\n
target remote localhost:1212

## If you are using simulavr as the remote target, this will upload
## the program into flash memory for you.

echo (gdb) load\n
load

## Set a break point at the beginning of main().

echo (gdb) break main\n
break main

## Run the program up to the first break point.  Gdb's `run` command
## does not work when using a remote target, must use continue.

echo (gdb) continue\n
continue
```

As you can see, I `echo` every command so I can see what gdb has done when it runs the commands in the '`.gdbinit`' file.

3.2 Building GDB for AVR

In order to use simulavr as a backend to gdb, you must build a special AVR version of gdb. All gdb versions starting with gdb-5.2.1 officially support the AVR target. You can just configure gdb with the `--target=avr` option. For example, you can use this procedure to install avr-gdb in /usr/local/bin:

```
$ ./configure --target=avr
$ make
$ su
# make install
# exit
```

4 Display Coprocesses

This chapter documents the protocol that simulavr uses to pass register and memory information to a display coprocess.

A display coprocess is a separate program started by simulavr for the sole purpose of displaying register and memory information while an AVR program is running in the simulator. Using a separate program and a standardized communication protocol, keeps the simulavr code simpler and allows for a variety of display programs to be used.

When the user asks simulavr to display register and memory information during execution, simulavr will start a coprocess to perform the display work. A pipe will be opened in simulavr into which the data will be written using the following commands:

'q'	Quit.
'r<reg>:<val>'	Set register to val.
'p<val>'	Set program counter to val.
'i<reg>:<val>'	Set io register to val.
'I<reg>:<name>'	Set io register name.
's<addr>,<len>:XX'	Set sram addrs to values (one XX pair per addr).
'e<addr>,<len>:XX'	Set eeprom addrs to values (one XX pair per addr).
'f<addr>,<len>:XXXX'	Set flash addrs to values (one XXXX quad per addr).
'n<clock_ticks>'	Update the number of clock ticks.

All values are hexidecimal numbers, except for <name> which is a string.

In order for the display process to know which pipe to read the information, it must handle either the '`--pfd <fd>`' option or check the `SIM_PIPE_FD` enviroment variable. The value passed using either method will be the file descriptor number of the pipe from which the display prgram will read the informtion.

Simulavr will start all display programs like so (sizes are decimal numbers of bytes and sram_start is just the decimal address of the first byte of sram, usually 0x60 [96] or 0x100 [256]):

'`<prog> --pfd <fd> <flash_size> <sram_size> <sram_start> <eeprom_size>`'

The user can specify the display program to use via the '`--disp-prog`' option to simulavr or using the `SIM_DISP_PROG` environment variable. If both are not specified, then no display will be used.

5 Simulavr Internals

Simulavr internals are documented using the doxygen system to automate generation of the documentation from the source code comments. The documentation for the latest release is always available at:

http://savannah.nongnu.org/download/simulavr/doc/internals_html/

The most up-to-date documents will most likely be those in the source code itself. If you wish to help develop simulavr, it is highly recommended that you get the latest source from cvs and consult the internals documents there.

6 Reporting Bugs

If you find a bug in simulavr, please send electronic mail to simulavr-devel@nongnu.org. Include the version number, which you can find by running '`simulavr --version`'. Also include in your message the output that simulavr produced, a simple AVR program which reproduces the bug, and the output you expected. If you are using avr-gdb also include the version number reported by '`avr-gdb --version`'.

If you have other questions, comments or suggestions about simulavr, contact me via electronic mail at the above address.

Concept Index

-

--breakpoint ... 3
--clock-freq ... 3
--core-dump ... 3
--debug ... 2
--device ... 2
--disp-prog ... 3
--eeprom-image ... 2
--eeprom-type ... 2
--flash-type ... 3
--gdb-debug ... 2
--gdbserver ... 2
--help ... 2
--list-devices ... 3
--port ... 2
--version ... 2
--without-xterm ... 3

A

aliasing ... 2
avr-gdb ... 7

B

bugs ... 10

D

developing ... 9
display ... 8
display protocol ... 8

G

gdb ... 4
gdb, building ... 7
gdb, hints ... 5
gdbserver ... 4

I

internals ... 9
introduction ... 1
invoking ... 2

O

options ... 2

P

problems ... 10

R

running ... 2

S

SIGHUP, from gdb ... 5
SIM_DISP_PROG ... 8
SIM_PIPE_FD ... 8
symbolic linking ... 2