# Debugging in Scratchbox

**Lauri Arimo**

**Debugging in Scratchbox**

by Lauri Arimo



This is an document about debugging in Scratchbox environment

Revision history

| Version: | Author: | Description: |
|---|---|---|
| 2005-02-15 | Savola | Removed gdbserver workaround |
| 2005-02-08 | Savola | Scratchbox 1.0 updates |
| 2004-11-08 | Arimo | Added some details |
| 2004-09-30 | Arimo | Initial revision |

# Table of Contents

# Chapter 1. Introduction

## 1.1. Purpose of this document

This document introduces some tools that can be used for debugging and are provided with Scratchbox. Focus is on describing differences between usage on native systems and Scratchbox. This is not complete user manual for debugging tools, although some good reference pointers are given.

Reader is assumed to have good Linux knowledge and some experience in debugging. Good place to start gathering common knowledge about debugging in Linux environment is paper written by Movial [1].

## 1.2. Scratchbox environment

Scratchbox [2] is a cross-compilation toolkit designed to make embedded Linux application development easier.

Due to the nature of Scratchbox there can be binaries for different architectures. This mean that specially configured tools are needed for efficient debugging.

# Chapter 2. Debugging

## 2.1. Debugging tools provided by scratchbox

Scratchbox provides following debugging tools by default:

- strace - A system call tracer [3]
- gdb - The GNU Debugger [4]
- gdbserver - Remote Server for the GNU Debugger [5]

## 2.2. strace

Strace is a system call tracer, i.e. a debugging tool which prints out a trace of all system calls made by another process/program. The program to be traced need not be recompiled for this, so you can use it on binaries for which you don't have source.

In Scratchbox strace is used through a wrapper. The wrapper uses a host version of strace when the compilation target does not use CPU-transparency (for example when using an x86 target), or when the CPU-transparency method is QEMU. Otherwise the script tries to execute the target binary `/usr/bin/strace`. If you want to use strace while using sbrsh for CPU-transparency, you can install strace on your target with the following command:

```
[sbox-ARM: ~] > sb-conf install --strace
```

> **Note:** Host strace can be used with QEMU because QEMU directs the target command's syscalls to the host kernel.

Strace can be used in two ways. Either by running a new binary (e.g. **strace /bin/ls**) or by tracing existing process by specifying its process id (e.g. **strace -p 4352**). Other really useful flags are **-f** to follow forks, **-o** to write output to file and **-e** which can be used to filter output. Detailed information can be found on strace's man page.

## 2.3. gdb

GDB is a source-level debugger, capable of breaking programs at any specific line, displaying variable values, and determining where errors occurred.

gdb needs to be configured separately for each toolchain. This is why it's built with the toolchains (see *Scratchbox toolchains* [7]). The user should not have to worry about this; Scratchbox will put right version of gdb to the PATH. One can find correct version of gdb from the `/scratchbox/compilers/<compilername>/arch_tools/bin` directory.

If you plan to debug a binary for some other architecture than x86, you need to use gdbserver to run the binary. Gdbserver usage is described in more depth in the next chapter. To use gdbserver you need to use the **target remote ip:port** command in gdb.

To make debugging with gdb possible, one should compile binaries with **-g** flag so that debugging symbols are included in the binaries. It is possible to use stripped binary with gdbserver and tell to gdb where to look for debug symbols with the **symbol-file /path/to/file** command. This is explained in the Chapter 3 chapter.

Scratchbox's gdb has also been patched so that one should be able to read x86 and ARM core files on host machine. This is a good way to find out what went wrong in the first place.

## 2.4. gdbserver

Gdbserver is a program that allows you to run gdb on a different machine than the one which is running the program being debugged. This means that one can run binaries for different architecture on target device via Scratchbox's CPU-trancparency feature. Actual debugging can then be handled on host with the gdb provided by the toolchain.

To use gdbserver, one has to first install it on the target. It can be done using the following command:

```
[sbox-ARM: ~] > sb-conf install --gdb
```

Launching gdbserver is easy:

```
[sbox-ARM: ~] > gdbserver <ip>:<port> <program> <arguments>
```

This will make gdbserver listen for TCP connections from host <ip> at port <port>. It is also possible to attach gdbserver to an already running process with the **--attach <pid>** option.

One has to use sbrsh as the CPU-transparency method because QEMU does not support gdb.

Currently gdbserver supports the following platforms:

- arm-*-linux-gnu
- i386-*-linux-gnu
- ia64-*-linux-gnu
- m68k-*-linux-gnu
- mips-*-linux-gnu
- powerpc-*-linux-gnu
- sh-*-linux-gnu

# Chapter 3. Example session

## 3.1. general

In this chapter we have an example session of gdb and gdbserver usage in Scratchbox. Chapter is divided to multiple sections so that it is easier to see what is going on. It is strongly advised to go trough this example to understand how gdb and gdbserver act together in Scratchbox.

This chapter assumes that one has correctly setup scratchbox installation with working CPU-tranparency [6].

## 3.2. setup

First one needs a program to debug.

```
[sbox-ARM: ~/debug-test] > cp /scratchbox/packages/hello.c .
[sbox-ARM: ~/debug-test] > cat hello.c
#include <stdio.h>

int main(void)
{
        printf("hello world\n");
        return 0;
}
[sbox-ARM: ~/debug-test] > gcc -g hello.c
[sbox-ARM: ~/debug-test] > ./a.out
hello world
[sbox-ARM: ~/debug-test] >
```

## 3.3. gdbserver

First, copy gdbserver to target filesystem...

```
[sbox-ARM: ~/debug-test] > sb-conf install --gdb
[sbox-ARM: ~/debug-test] > file /usr/bin/gdbserver
/usr/bin/gdbserver: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0, d
[sbox-ARM: ~/debug-test] >
```

Then just launch it with correct communication info and correct program with arguments:

```
[sbox-ARM: ~/debug-test] > gdbserver 172.16.6.66:4444 ./a.out
```

```
Process ./a.out created; pid = 642
Listening on port 4444
```

Gdbserver is now waiting connections from 172.16.6.66 to port 4444. It will wait until interrupted or successful connection is made. After connection program execution goes as the gdb commands. Below is one possible output:

```
[sbox-ARM: ~/debug-test] > ./gdbserver 172.16.6.66:4444 ./a.out
Process ./a.out created; pid = 642
Listening on port 4444
Remote debugging from host 172.16.6.66
hello world

Child exited with retcode = 0

Child exited with status 0
GDBserver exiting
[sbox-ARM: ~/debug-test] >
```

Quite simple, eh?

## 3.4. gdb

Check that correct gdb is in PATH:

```
[sbox-ARM: ~/debug-test] > gdb -v
GNU gdb 6.1
Copyright 2004 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=i686-pc-linux-gnu --target=arm-linux".
[sbox-ARM: ~/debug-test] >
```

Launch gdb, load correct symbols from a file and connect to the gdbserver. Remember to use **cont** instead of **run** when starting the program. This is because program is already being ran by gdbserver.

```
[sbox-ARM: ~/debug-test] > gdb
GNU gdb 6.1
Copyright 2004 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=i686-pc-linux-gnu --target=arm-linux".
(gdb) file ./a.out
Reading symbols from ./a.out...done.
(gdb) target remote 172.16.6.54:4444
Remote debugging using 172.16.6.54:4444
```

```
0x40000d00 in ?? ()
(gdb) cont
Continuing.

Program exited normally.
(gdb)
```

One can set breakpoints etc. before running program with **cont**. After the program exits, one has to restart gdbserver before new session.

## 3.5. gdb and corefiles

Scratchbox's gdb can handle corefiles for x86 and ARM targets.

Program must be compiled with debugging symbols (ie. **-g** switch) if some meaningful output is wanted.

```
[sbox-ARM: ~/debug-test] > cat segfault.c
int a (int *p);

int
main (void)
{
          int *p = 0;   /* null pointer */
            return a (p);
}

int
a (int *p)
{
          int y = *p;
            return y;
}
[sbox-ARM: ~/debug-test] > gcc -g segfault.c
[sbox-ARM: ~/debug-test] > ./a.out
[sbox-ARM: ~/debug-test] > gdb ./a.out core
GNU gdb 6.1
Copyright 2004 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=i686-pc-linux-gnu --target=arm-linux"...

warning: core file may not match specified executable file.
Core was generated by ‘’.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
```

```
Loaded symbols for /lib/ld-linux.so.2
#0  0x000083b0 in a (p=0x0) at segfault.c:13
13              int y = *p;
(gdb) bt
#0  0x000083b0 in a (p=0x0) at segfault.c:13
#1  0x0000838c in main () at segfault.c:7
(gdb)
```

# Chapter 4. Graphical frontends

## 4.1. General

It’s possible to use some graphical frontend for scratchbox’s gdb. Perhaps the most popular of these is DDD - The Data Display Debugger, which can be used as frontend for many different debuggers.

The trick is to tell the frontend to use Scratchbox’s gdb. In this document DDD is used as an example. Other frontends should work with same principles.

## 4.2. DDD

The Data Display Debugger (DDD) is a popular graphical user interface to UNIX debuggers such as GDB, DBX, XDB, JDB and others. Besides “usual” front-end features such as viewing source texts and breakpoints, DDD provides an interactive graphical data display, where data structures are displayed as graphs. Using DDD, you can reason about your application by watching its data, not just by viewing it execute lines of source code.

To launch DDD with Scratchbox’s gdb use following command:

```
$ ddd --debugger '/scratchbox/login gdb'
```

More information about debugging with DDD can be found from [8].

# References


[1] *Debugging linux applications (http://www.movial.fi/client-data/file/movial_debugging_linux_applications.pdf)* .

[2] *Scratchbox website (http://scratchbox.org/)* .

[3] *strace - A system call tracer (http://www.devchannel.org/article.pl?sid=03/10/24/2057246)* .

[4] *gdb - The GNU Debugger (http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html)* .

[5] *gdbserver - Remote Server for the GNU Debugger (http://sources.redhat.com/gdb/current/onlinedocs/gdb_18.html#SEC146)* .

[6] *Scratchbox toolchains (http://scratchbox.org/documentation/docbook/installdoc.html)* , Ricardo Kekki.

[7] *Installing Scratchbox (http://scratchbox.org/documentation/docbook/toolchain.html)* , Rahkonen Valtteri.

[8] *DDD - Data Display Debugger (http://www.gnu.org/software/ddd/manual/)* .