Debugging Guide for GDB and Eclipse

by Brian Fraser
Last update: Sept 25, 2016

This document guides the user through:

1. Debugging an application using GDB command prompt.
2. Debugging an application using Eclipse.
3. Generating and loading core files.
4. Stripping debug symbols from a binary.

Table of Contents

Note: This guide has not yet been tested in the SFU Surrey Linux Lab (SUR4080). Some changes may be needed.

Formatting:

1. Commands starting with `$` are host Linux console commands:
   ```
   $ echo "Hello world!"
   ```
2. Commands starting with `#` are target Linux console commands:
   ```
   # echo "On the target! Hello world!"
   ```
3. Commands starting with `(gdb)` are GDB console commands.
4. Almost all commands are case sensitive in Linux and GDB.

Revision History:

- Sept 18: Initial version for fall 2016
- Sept 26: Added directions for installing `gdbserver` on target.

1. Installing gdb-multiarch

The host needs a cross-debugger to debug an application running on the target. GDB (GNU Debugger) has a version which supports multiple architectures (such as ARM, MIPS, …) named `gdb-multiarch`.

1.1 If Running Ubuntu 16.xx

1. Install GDB and GDB multi-architecture:
   ```
   $ sudo apt-get install gdb
   $ sudo apt-get install gdb-multiarch
   ```
2. Run `gdb-multiarch` and check its version.
   ```
   $ gdb-multiarch -v
   ```
   - Should display first line similar to the following:
     ```
     GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
     ```

1.2 If Running Ubuntu 14.04

Normally, you should be able to do an `apt-get` on `gdb-multiarch`; however, Ubuntu 14.04 is packaged with `gdb-multiarch` version 7.7 which is unable to correctly process core files from the target. Here is the process to get `gdb-multiarch` 7.8 which does work.

1. Remove any existing versions of GDB and GDB multi-architecture:
   ```
   $ sudo apt-get remove gdb gdb-multiarch
   ```
2. Add the Ubuntu `utopic` repository to `/etc/apt/sources.list`:
   ```
   $ sudo gedit /etc/apt/sources.list
   ```
   - At the end of the file, add the following lines:
     ```
     ## Added for GDB 7.8
     deb http://old-releases.ubuntu.com/ubuntu utopic main universe
     ```
   - Note: `/etc/apt/sources.list` is a protected file, so must be root to edit it. Use `sudo` to launch `gedit` (as shown).
3. Update the packages available through the new repository:
   ```
   $ sudo apt-get update
   ```
4. Install GDB and GDB multi-architecture:
   ```
   $ sudo apt-get install gdb
   $ sudo apt-get install gdb-multiarch
   ```
   - You may need to use the “fix” option first before the above commands will work:
     ```
     $ sudo apt-get -f install
     ```
5. Run `gdb-multiarch` and check its version.
   ```
   $ gdb-multiarch -v
   ```
   - Should display first line:
     ```
     GNU gdb (Ubuntu 7.8-1ubuntu4) 7.8.0.20141001-cvs
     ```
6. Troubleshooting:
   - If you are having problems getting the correct version to install, you can double check that apt-get is reading the correct repository to find GDB version 7.8 (or better).

View GDB:

```
$ apt-cache showpkg gdb
```

View GDB-Multiarchitecture:

```
$ apt-cache showpkg gdb-multiarch
```

If the desired version of the package is not shown, double check your `sources.list` file, re-run "`apt-get update`"

2. GDB

GDB is a text-debugger common to most Linux systems. For remote debugging, we'll run `gdbserver` on the target, and the cross-debugger (`gdb-multiarch`) on the host.

1. Build your project using the `-g` option to ensure the file gets debug symbols.
   - This likely means adding the `-g` option to your `CFLAGS` variable in your Makefile.
2. On the target, install `gdbserver` (if not already installed):
   - Ensure you have internet access. If not, see the networking guide.
     ```
     # ping google.ca
     ```
   - Install GDB server on the target:
     ```
     # apt-get update
     # apt-get install gdbserver
     ```
3. On the target, change to the directory where your application is (assumed to be named `helloWorld`), and launch `gdbserver`:
   ```
   # gdbserver localhost:2001 helloWorld
   ```
   - It should look like the following (`pid` likely to be different):
     ```
     # gdbserver localhost:2001 helloWorld
     Process helloWorld created; pid = 1068
     Listening on port 2001
     ```
4. On the host, in the directory of your `helloWorld` executable, launch the cross-debugger:
   ```
   $ gdb-multiarch -q helloWorld
   ```
5. At the GDB prompt "`(gdb)`", type in the following command to connect to the target:
   ```
   (gdb) target remote 192.168.0.102:2001
   ```
   - Change the IP address to the IP address of the target.
   - The host should look like this:
     ```
     $ gdb-multiarch -q helloWorld
     (gdb) target remote 192.168.0.171:2001
     Remote debugging using 192.168.0.171:2001
     warning: Unable to find dynamic linker breakpoint function.
     GDB will be unable to debug shared library initializers
     and track explicitly loaded dynamic code.
     0x400007b0 in ?? ()
     (gdb)
     ```
   - The target should now have displayed the additional line (your IP will be different):
     ```
     Remote debugging from host 192.168.0.188
     ```
6. You now have a GDB session. You should be familiar with the following GDB commands (parts in italics can be replaced by other values):
   - `list, frame, quit`
   - `info breakpoints, break` *`main`*`, break` *`lineNumberHere`*`, delete` *`1`*
   - `continue, print` *`myVar`*`, step, next`
   - `bt, info args, info frame, info local, up, down`
   - `Control + C` (to interrupt program when running).
7. Troubleshooting:
   - Ensure your host can communicate with the target. Try pinging the board and opening a `ssh`

prompt to the board. Refer to the quick-start guide and associated trouble shooting steps if this fails.

- If you get the wrong version of `gdbserver`, it may not run correctly on the target. When it is run without arguments, you should see the following:

```
Usage:      gdbserver [OPTIONS] COMM PROG [ARGS ...]
            gdbserver [OPTIONS] --attach COMM PID
            gdbserver [OPTIONS] --multi COMM

COMM may either be a tty device (for serial debugging), or
HOST:PORT to listen for a TCP connection.

Options:
  --debug               Enable general debugging output.
  --remote-debug        Enable remote protocol debugging output.
  --version             Display version information and exit.
  --wrapper WRAPPER --  Run WRAPPER to start new programs.
  --once                Exit after the first connection has closed.
```

- You can ignore any errors about mapping shared library sections. At the moment we do not need to worry about debugging these.
- If `bt` does not yield a meaningful stack, it may mean that you are in some library or OS code that you do not control. Try setting a break-point in a part of your code you know to be running and then let execution continue. It should hit your breakpoint and show you meaningful content.

3. Eclipse

3.1 Eclipse Installation and Project Setup

1. Install Java's run time environment:
   `$ sudo apt-get install openjdk-8-jre`
2. Install Eclipse. An easy way to do this is download the “Eclipse Installer” from https://eclipse.org/downloads/
   - When asked, install “Eclipse IDE for C/C++ Developers”.
   - Install the latest version (Neon as of Fall 2016). 64-bit version recommended (on 64-bit systems).
   - I suggest not using `apt-get`; it will likely get an older version of Eclipse.
   - When you install, Eclipse just extracts itself into a folder. If you want an icon for it, you'll have to create the icon yourself.
3. Launch Eclipse. Command is likely:
   `$ ~/eclipse/cpp-neon/eclipse/eclipse`
   - You may be asked about a workspace when starting it; it is fine to accept the default one.
4. Create a new project (File → New → Project...).
   - Under C/C++, select "Makefile Project with Existing Code"
   - Browse to the directory of your existing project with a makefile.
   - Select the “Cross GCC” tool chain.
   - Name the project and click Finish.
5. Setup the Makefile support for your project.
   - Display the Make Target view: Window → Show View → Other. Under Make, select Make Target.
   - In the Make Target view, create a new target (green bulls-eye icon) for your desired makefile target.
     - Note, by default Eclipse expects a `clean` and `all` target. You can change these by right-clicking your project, select Properties; under C/C++ Build, select the Behaviour tab.
   - Double click on the new make target. The build output should appear in the Console view. You may need to manually switch to the Console view (bottom).
6. Suggested Settings:
   - Auto-save all files when compiling:
     Window → Preferences, in left expand General → Workspace, check “Save automatically before build”.
7. Eclipse coding tips:
   - Ctrl+B to build. View the Console window to see the build messages.

- Eclipse will show you the errors found during your last compile. If you correct the error but don't rebuild yet, Eclipse will still show the error information from the last build.
- Eclipse does some code analysis of its own (in addition to the normal build process). Eclipse will show an indication of some problems even without build. However, sometimes it may find things which it thinks are errors but will actually build OK.
- What Eclipse rebuilds depends on your makefile. If you setup dependencies correctly, it will rebuild a `.c` file when it changes. However, often the .h files are missed. So, if you change a `.h` file you may need to do a make-clean and then a re-build (make targets `clean` and `all`, possibly).

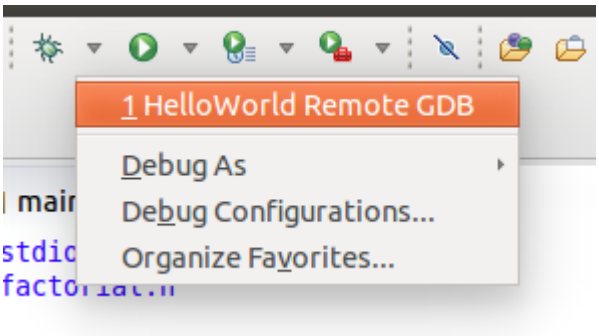
3.2 Debugging with Eclipse

1. In Eclipse, create the debug configuration for your project:
   - With your project selected, on the menu go to Run → Debug Configurations.
     - If this does not work, you should be able to right-click on your project and select Debug As..., and then debug configurations.
   - Double click on the "C/C++ Remote Application" item on the left to create a new configuration.
2. At the top, give the debug configuration a name such as “MyProjectNameHere Remote Debugging”
3. Setup the debug target (Main tab):
   - At the bottom, click on the "Select other..." hyper-link. Check the "Use configuration specific settings" box, and then select "GDB (DSF) Manual Remote Debugging Launcher". Click OK.
   - Change the Build settings to "Disabled auto build"
   - Select the application using the Browse button. Select the application you want to debug. This should be the application with debug symbols included (not stripped). This will likely be the compiled version of your current project, possibly (though not necessarily) in the `~/cmpt433/public/` folder.
4. Change to the Debugger tab:
   - Set the GDB debugger by browsing to `gdb-multiarch` debugger. The path is likely: `/usr/bin/gdb-multiarch`
     - Hint: Locate where `gdb-multiarch` is with:
       `$ whereis gdb-multiarch`
   - Still on the Debugger tab, but on the Connection sub-tab, set the connection information:
     `Type: TCP`
     `Host name or IP address: 192.168.7.2` (the IP Address of the target on your network)
     `Port number: 2001`
5. Click Apply to save the settings.
6. On the target, launch `gdbserver` using the same command as used for text-debugging.

```
# gdbserver localhost:2001 helloWorld
```

- To do this, you must already have the compiled version of your project on the target (via NFS works). This file must have been compiled with the -g option if you want symbols to be available (i.e. function/variable names etc).
- Hint: To pass arguments to your program being debugged, use the command such as the following where the arguments `10`, `42`, `end`, `of`, `world` are passed in.:
  ```
  # gdbserver localhost:2001 helloWorld 10 42 end of world
  ```

7. On the Host, click the Debug button. It should connect to the target.
   - It may ask you if you want to switch to the debug perspective; say yes.
   - Use the integrated debugger to step-through and debug your application. The application is actually run on the target, so any effects of the program will take effect on the target. For example, `printf()` statements output through the console and code to flash an LED will still flash the target's LED.
8. You can switch back to the normal perspective by clicking C/C++ button in the top-right of Eclipse.
9. Later, to re-debug your application, you will need to:
   - Restart `gdbserver` on the target:
     ```
     # gdbserver localhost:2001 helloWorld
     ```
   - Re-launch the debugger in Eclipse by clicking the drop-down arrow beside the debugger icon on the toolbar. Then select the GDB launch profile you setup.



   - Note that you cannot just click the debug button, as this will launch it locally.
   - If you right-click the project and through Debug as... select Local C/C++ Application, it will not work because you cannot run the project on the local PC (host).
10. Trouble shooting:
    - If Eclipse complains that it cannot find the application when you try to debug, you may need to relaunch the Debug Configuration window, and click "Debug" from there.
    - If you are having troubles connecting, ensure that your communication to the target is correctly configured. Try using `ping` or `ssh`.
    - If you cannot connect to the target, ensure the target is running `gdbserver`, and correctly configured with the same port number that Eclipse.
    - Eclipse may display warnings from GDB about “Unable to find dynamic linker breakpoint

function", or about unable to load symbols for shared libraries. You may disregard these for the moment.

- Eclipse displays error "Launch failed. Binary not found." You likely selected to debug the application on the local PC instead of running it through the remote GDB server.
- Eclipse displays error "Launching <project> has encountered an error. Error in final launch sequence", with details saying "connection timed out". This means there is a problem communicating with the target.
  - Ensure that `gdbserver` is correctly executing on the target. You'll have to restart it each time you restart debugging the application.
  - Ensure the IP address of the target is correct.
  - Ensure the port number used in Eclipse matches the port number used to start `gdbserver` on the target.
  - Ensure you have network connectivity between the host and target using `ping`.
- If the panels and tool bars inside Eclipse seem to be out of place or messed up, try:
  1) Go to Window → Perspective → Reset Perspective…, and then click Yes to reset all views to default locations.

  2) If missing tool bar buttons: Window → Show Toolbar

4. Core Dumps

A core file is generated when the application crashes (usually due to a segmentation fault).

1. Configure Linux on the target to generate core files:
   ```
   # ulimit -c unlimited
   ```
   - Check the change with:
     ```
     # ulimit -a
     ```
   - The output should show "`core file size          (blocks, -c) unlimited`"
2. Change to the `/tmp` directory on the target and run the program which crashes. For example:
   ```
   # cd /tmp
   # /mnt/remote/myApps/thisProgramCrashes
   ```
   - When it crashes, it should say: "`Segmentation fault (core dumped)`"
   - If you are not in the `/tmp` folder the `core` file may be created but be empty (0 bytes).
3. Check the core file with:
   ```
   # ls -l core
   ```
   - Its size should be greater than 0 bytes.
4. Change the permissions on the core file:
   ```
   # chmod a+rw core
   ```
5. Copy the core file to the shared NFS directory:
   ```
   # cp core /mnt/remote
   ```
   - You may need to ensure that your NFS directory has global write permission. This may be related to Linux permissions, or the NFS server setup.
6. On the host, under your NFS public directory (likely `/home/userName/cmpt433/public`), run the cross-debugger on the `core` file:
   ```
   $ gdb-multiarch pathToApplicationThatCrashed core
   ```
   - For example:
     ```
     $ arm-linux-gdb ./myApps/myCrashingProgram core
     GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
     …
     This GDB was configured as "x86_64-linux-gnu".
     …
     Type "apropos word" to search for commands related to "word"...
     Reading symbols from myApps/myCrashingProgram ...done.
     [New LWP 1652]

     warning: Could not load shared library symbols for 2 libraries, e.g.
     /lib/arm-linux-gnueabi/libc.so.6.
     Use the "info sharedlibrary" command to see the complete listing.
     Do you need "set solib-search-path" or "set sysroot"?
     Core was generated by `/mnt/remote/myApps/myCrashingProgram'.
     Program terminated with signal SIGSEGV, Segmentation fault.
     #0  0x00008514 in yourFunction (d1=0x11f5a) at myCrashingProgram.c:22
     (gdb)
     ```
   - Or, one can run GDB natively on the target:
     ```
     # cd /mnt/remote/myApps/
     # gdb  pathToApplicationThatCrashed core
     ```
7. Use the standard GDB commands to debug the application.

- Hint: Start with a back-trace.

8. **Optional:** Here is another way to generate a `core` dump which does not require using the `/tmp` directory (thank you to a student for sharing this approach).
   - On the target, create a new user to match your user name on the host. You do not need to set the password or create a home directory. If your host username is "`brian`" then:
     ```
     # adduser --disabled-password --no-create-home brian
     ```
     - Press ENTER to accept each of the defaults.
   - Use `su` to switch to the new user:
     ```
     $ su brian
     ```
   - Run your program which crashes
     ```
     # ./hello
     ```
   - Now, when your application crashes it should be able to write a `core` file to the NFS public directory (`/mnt/remote/`). You will still need to have setup the `ulimit` correctly.
   - From the host, check the ownership and permissions on the `core` file. It should be owned by your user, and therefore have read-access from the host. If not, use `chown` to fix.
   - Return to being the root on the target:
     ```
     # exit
     ```
9. Troubleshooting
   - If `gdb-multiarch` cannot identify the type of file for the core file, ensure that the following command can handle the file:
     ```
     $ readelf -h core
     ```
   - If `gdb-multiarch` either crashes with an assert, or is unable to identify the file type of the core file then check:
     - Ensure the `core` file is not zero bytes. Ensure you are using `/tmp` directory.
     - Ensure `gdb-multiarch` is not version 7.7. I have tested version 7.8 and it works (against target running debian package).
   - If the `core` file is 0 bytes, ensure you are in the `/tmp/` folder before running your application, otherwise the `core` file may not be successfully created.

5. Stripping a Binary

When built with debug information (`-g` GCC option), the executable can be twice the size. This can take up too much room on an embedded system, so we may want to strip the version copied to the target if there is not much room on the target (and don't need it for debugging)..

1. Copy your application (which has debug symbols) to the shared public directory.
2. Check the file size of your application:
   ```
   $ ls -l helloWorld
   ```
3. Change to the public directory, and strip the binary:
   ```
   $ arm-linux-gnueabihf-strip helloWorld
   ```
4. Check the file size of your application; it should be (much?) smaller.
   ```
   $ ls -l helloWorld
   ```
5. Be careful to use the correct version of the file as needed:
   - The target can run the stripped version (or the non-stripped version, if space is permitting).

- The host should debug using the non-stripped version. This way you get debug symbols, while still having a smaller executable on the target.