Debugging Guide for GDB and VS Code

by Brian Fraser

Last update: Oct 4, 2022

Guide has been tested on

BeagleBone (Target):	**Debian 11.4**
PC OS (host):	**Debian 11.5**

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

Formatting

1. Commands for the host Linux's console are show as:
   `(host)$ echo "Hello PC world!"`
2. Commands for the target (BeagleBone) Linux's console are shown as:
   `(bbg)$ echo "Hello embedded world!"`
3. Commands starting with `(gdb)` are GDB console commands.
4. Almost all commands are case sensitive in Linux and GDB.

Revision History:

- Oct 2, 2019: Add directions to working with Ubuntu 18.04
- Jan 31, 2021: Add VS Code graphical debugging, update to Ubuntu 20.04
- Feb 16, 2021: Added directions for building via a Makefile in VS Code
- Feb 4, 2022: Added more troubleshooting to valgrind section.
- Oct 4, 2022: Updated for Debian 11.x

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. Install GDB and GDB multi-architecture:
   ```
   (host)$ sudo apt-get install gdb gdb-multiarch
   ```
2. Run `gdb-multiarch` and check its version.
   ```
   (host)$ gdb-multiarch -v
   ```
   - Should display first line similar to the following:
     ```
     GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
     ```
3. 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 8.2 (or better?).

     View GDB-Multiarchitecture:
     ```
     (host)$ 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`"
   - Changing GDB multiarch version **(only do if needed; older versions of Ubuntu/BBG!)**
     - `apt-get` for `gdb-multiarch` should normally work; however, some version mismatches between `gdbserver` on the target and `gdb-multiarch` on the host are possible. For example, target `gdbserver` 7.12.0.20016007 is incompatible with host `gdb-multiarch` 8.1 (fails to execute correctly, load libraries, ...). Steps to resolve:
     - Remove any existing versions of GDB and GDB multi-architecture from the host:
       ```
       (host)$ sudo apt-get purge gdb gdb-multiarch
       ```
     - Add the Ubuntu repository you need to `/etc/apt/sources.list`:[1]
       ```
       (host)$ sudo nano /etc/apt/sources.list
       ```
       - At the end of the file, add the following lines (change '`cosmic`' to distro you need):
         ```
         ## Added for GDB 8.2 (replacing 8.1)
         deb http://ca.archive.ubuntu.com/ubuntu cosmic main universe
         ```
       - Note: `/etc/apt/sources.list` is a protected file, so must use `sudo` to edit.
     - Update the packages available through the new repository:
       ```
       (host)$ sudo apt-get update
       ```
     - Install GDB and GDB multi-architecture:
       ```
       (host)$ sudo apt-get install gdb gdb-multiarch
       ```
       - You may need to use the "fix" option first before the above commands will work:
         ```
         (host)$ sudo apt-get -f install
         ```

1 For Ubuntu 14.xx, add the utopic repository to get gdb-multiarch 7.8.1ubuntu4:
deb http://old-releases.ubuntu.com/ubuntu utopic main universe

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.
     ```
     (bbg)$ ping google.ca
     ```
   - Install GDB server on the target:
     ```
     (bbg)$ sudo apt-get update
     (bbg)$ sudo apt-get install gdbserver
     ```
3. On the target, change to the directory where your application is (assumed to be named `helloWorld`), and launch `gdbserver`:
   ```
   (bbg)$ gdbserver localhost:2001 helloWorld
   ```
   - It should look like the following (`pid` likely to be different):
     ```
     (bbg)$ 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:
   ```
   (host)$ gdb-multiarch helloWorld
   ```
5. At the GDB prompt "`(gdb)`", type in the following command to connect to the target:
   ```
   (gdb) target remote 192.168.7.2:2001
   ```
   - Change the IP address to the IP address of the target.
   - The host should look like this (some lines omitted):
     ```
     (host)$ gdb-multiarch helloWorld
     Reading symbols from primeThread...
     (gdb) target remote 192.168.7.2:2001
     Remote debugging using 192.168.7.2:2001
     Reading …        (lines omitted)
     warning: …       (lines omitted)
     (No debugging symbols found in target:/lib/ld-linux-armhf.so.3)
     0xb6fd5a80 in ?? () from target:/lib/ld-linux-armhf.so.3
     (gdb)
     ```
   - The target's terminal should now display an additional line similar to:
     ```
     Remote debugging from host ::ffff:192.168.7.1, port 57462
     ```
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 file.c:`*`lineNumberHere`*`, delete` *`1`*
   - `continue, print` *`myVar`*`, step, next`
   - `bt, info args, info frame, info local, up, down`
   - `Control + C` (to interrupt program when running to get gdb prompt).
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. VS Code for Graphical Debugging

Section optional: You may use VS Code or Eclipse; you need *not* use both.

1. Install VS Code on the host (if not already installed):
```
(host)$ sudo apt-get install snap
(host)$ sudo snap install --classic code
```
2. From the folder of your code, launch VS Code:
```
(host)$ code .
```
3. Install the "**GDB Debug**" extension in VS Code via the Extensions view on the left.
4. Create a `launch.json` file by Run –> Add Configurations. You may select anything when prompted, and then overwrite `launch.json` with the following:[2]:
```
{
  // SOURCE: https://medium.com/@karel.l.vermeiren/ \
  //   cross-architecture-remote-debugging-using-gdb-with-visual-studio-code-vscode-on-linux-c0572794b4ef
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // More information at: https://go.microsoft.com/fwlink/linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "GDB debug - custom",
      "type": "cppdbg",
      "request": "launch",
      "program": "~/cmpt433/public/myApps/my_awesome_app_here",
      "args": [],
      "stopAtEntry": true,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ],
      "miDebuggerPath": "/usr/bin/gdb-multiarch",
      "miDebuggerServerAddress": "192.168.7.2:2001"
    }
  ]
}
```
   - Change "`program`" to be the path, on the *host*, to the *cross-compiled* executable.
   - If needed, update `miDebuggerServerAddress` to the IP of the target.
5. On the host, cross-compile your program with the `-g` flag to GCC, which adds debug information.
   - Hint: Just have your makefile include the `-g` flag all the time.

2 File launch.json described by Karel Vermeiren via medium.com (retrieved Jan 30, 2021)

6. On the target, from the NFS folder containing the cross-compiled executable of the program to debug, launch GDB server:

```
(bbg)$ cd /mnt/remote/myApps
(bbg)$ gdbserver localhost:2001 ./my_awesome_app_here
```

7. In VS Code, select **Run** > **Start Debugging.**

- On the target, you should now see it display a new line of:
  ```
  Remote debugging from host 192.168.7.1
  ```

8. Debug your application using the VS Code graphical debugger.
   - As you step into code in other source files, VS Code should show you the code.
   - If your program crashes, VS Code should detect an exception and stop at the line of code which generate the error.
9. Steps to rebuild and re-debug your program:
   - Stop any debug sessions.
   - Rebuild application on host, and copy to NFS mount
   - Re-run `gdbserver` on target
   - Reconnect to `gdbserver` from the host.
10. Optional: It is possible to setup VS Code to SCP your compiled executable to the target, and then run `gdbserver` directly. See Karel Vermeiren's guide for more if interested.
11. Troubleshooting
    - If VS Code is unable to connect to the target with the error:
      ```
      "Unable to start debugging. Unexpected GDB output from command "-target-select remote … Connection timed out."
      ```
      It likely means:
      - `gdbserver` is not running on the target
      - The IP address or port in the launch.json file for VS Code does not match the target. Find the IP address of the target using `ifconfig`.
    - If unable to see `printf()` output on VS Code while debugging: this is expected. `printf()` outputs from the program being debugged will be displayed on the terminal for the target device (SSH or serial) because the program is actually running on the target.

3.1 Makefile in VS Code

Optional: Configure VS Code to run the `all` target in your `makefile` with the Terminal → Run Build Task (Ctrl + Shift + B)

1. Open VS code in a folder containing your `makefile`
2. Go to the menu Terminal –> Configure Default Build Task
   - Select any type (we'll be replacing it)
   - This creates a `.vscode/tasks.json` file
3. Replace the contents of the `.vscode/tasks.json` file with the following:

```
{
  "version": "2.0.0",
  "tasks": [
   {
     "label": "Make: run project's Makefile",
     "command": "make",
     "args": [
      "all"
     ],
     "options": {
      "cwd": "${workspaceFolder}"
     },
     "problemMatcher": [
      "$gcc"
     ],
     "group": {
      "kind": "build",
      "isDefault": true
     },
   }
  ]
}
```

4. Build your project via the menu Terminal → Run Build Task (Ctrl + Shift + B)
   - The built-in terminal should now show the results of your build.
5. If your build has any errors you can Ctrl + click on the filename/line-number in the build output and then press ENTER to have VS Code jump to that location in your code.

4. Eclipse for Graphical Debugging

Section optional: You may use VS Code or Eclipse; you need *not* use both.

4.1 Eclipse Installation and Project Setup

1. Use snap to install Eclipse:
   ```
   (host)$ sudo apt-get install snap
   (host)$ sudo snap install --classic eclipse
   ```
2. Launch Eclipse (should have an icon, or run `eclipse`)
   - You may be asked about a workspace when starting it; it is fine to accept the default one.
   - If Eclipse fails to load with an error directing you to a log file saying
     ```
     java.lang.ClassNotFoundException:
     org.eclipse.core.runtime.adaptor.EclipseStarter
     ```
     then ensure no other versions of Eclipse are installed and repeat the install:
     ```
     (host)$ sudo apt-get purge eclipse
     (host)$ sudo apt-get autoremove
     (host)$ sudo snap remove eclipse
     (host)$ sudo snap install eclipse
     ```
3. Install C/C++ support for Eclipse:
   - Launch Eclipse (should have desktop icon, or run `eclipse` from terminal)
   - Go to Help → Install New Software
   - In the drop-down at the top, select "All Available Sites"
   - Expand Programming Languages, and check "C/C++ Development Tools"
   - Click Finish, accept licenses, etc.
4. Create a new project for a folder with a `makefile`
   - Go to 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 (if available).
   - Name the project and click Finish.
5. Setup the Makefile support for your project.
   - Display the Make Target view:
     Go to 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).

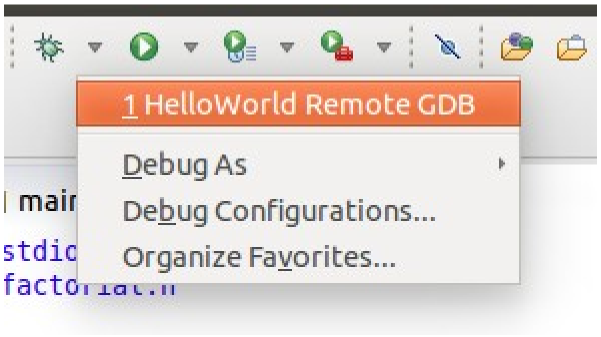
6. Suggested Settings:
   - Auto-save all files when compiling:
     Window → Preferences, in left expand General → Workspace → Build
     Check "Save automatically before build".
7. Eclipse coding tips:
   - Ctrl+B to build. View the Console window to see the build messages.
   - Eclipse will show you 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).

4.2 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):
   - Change the Build settings to "Disabled auto build"
   - Select the "C/C++ 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 `/home/<yourid>/cmpt433/public/` folder. Note: Does not work with ~ for your home directory.
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:
       ```
       (host)$ 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.
   ```
   (bbg)$ 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.:
     ```
     (bbg)$ 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:
   - Re-run `gdbserver` on the target:
     ```
     (bbg)$ 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

5. 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:
   ```
   (bbg)$ ulimit -c unlimited
   ```
   - Check the change with:
     ```
     (bbg)$ 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:
   ```
   (bbg)$ cd /tmp
   (bbg)$ /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:
   ```
   (bbg)$ ls -l core
   ```
   - Its size should be greater than 0 bytes.
4. Change the permissions on the core file:
   ```
   (bbg)$ chmod a+rw core
   ```
5. Copy the core file to the shared NFS directory:
   ```
   (bbg)$ 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 run the cross-debugger on the `core` file:
   ```
   (host)$ cd ~/cmpt433/public
   (host)$ gdb-multiarch pathToApplicationThatCrashed core
   ```
   - For example:

```
(host)(host)$ gdb-multiarch ./myApps/segfaulter core
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
...
Reading symbols from ./segfaulter...

warning: Can't open file /mnt/remote/myApps/segfaulter during file-backed
mapping note processing

warning: Can't open file /lib/arm-linux-gnueabihf/libc-2.31.so during
file-backed mapping note processing

warning: Can't open file /lib/arm-linux-gnueabihf/ld-2.31.so during file-
backed mapping note processing
[New LWP 1590]

warning: Could not load shared library symbols for 2 libraries, e.g.
/lib/arm-linux-gnueabihf/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 `./segfaulter'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x004d059a in dereferenceIt (ptr=0x0) at segfaulter.c:6
6           return *ptr;
(gdb)
```

- Or, instead of cross debugging from the host, one can run GDB natively on the target:

```
(bbg)$ cd /mnt/remote/myApps/
(bbg)$ gdb pathToApplicationThatCrashed core
```

7. Use the standard GDB commands to debug the application.
   - *Hint: Start with a back-trace (`bt`), and print variables (`print myVarName`)*

8. **Optional (may be out-dated):** 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). (Note: first try generating a core file in your shared folder and see if it works).
   - 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:
     ```
     (bbg)$ sudo adduser --disabled-password --no-create-home brian
     ```
     - Press ENTER to accept each of the defaults.
   - Use `su` to switch to the new user:
     ```
     (bbg)$ su brian
     ```
   - Run your program which crashes
     ```
     (bbg)$ ./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:
     ```
     (bbg)$ 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:
     ```
     (host)$ 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 8.1 which work.
   - 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.
   - Read the output of `gdb-multiarch` carefully. It may tell you it cannot find the executable file, or the core file. In which case, you should carefully check your paths.

6. Valgrind

Valgrind is a utility which allows you to do powerful memory analysis. It can find memory leaks and other pointer errors.

Install Valgrind on Target

1. Ensure you have internet access from your target. If not, follow the networking guide.
   ```
   (bbg)$ ping google.ca
   ```
2. Install Valgrind on target
   ```
   (bbg)$ sudo apt-get update
   (bbg)$ sudo apt-get install valgrind
   ```
3. Check Valgrind is working by running it on a simple "hello world" style application or the like:
   ```
   (bbg)$ valgrind /mnt/remote/myApps/hello_world
   ```
   Should exit without error. Run it on a program which seg-faults and see its output!
4. **Troubleshooting**
   - You can fully remove Valgrind from the target (either the stock version, or the updated version) using:
     ```
     (bbg)$ sudo apt-get purge valgrind valgrind-dbg
     (bbg)$ sudo apt-get autoremove
     ```
   - If you try to use apt-get after installing the valgrind packages, you may get the error:
     ```
     You might want to run 'apt --fix-broken install' to correct these.
     The following packages have unmet dependencies:
      valgrind : Depends: libc6 (>= 2.28) but 2.24-11+deb9u4 is to be installed
     E: Unmet dependencies. Try 'apt --fix-broken install' with no packages
      (or specify a solution).
     ```
     An easy way to resolve this is to remove `valgrind`/`valgrind-dbg` (see above note), then install the package you want with apt-get. If needed, reinstall `valgrind` as per this guide.
   - If you run Valgrind and see:
     ```
     valgrind: m_transtab.c:2459 (vgPlain_init_tt_tc): \
      Assertion 'sizeof(TTEntryC) <= 88' failed.
     ```
     It means you have the incorrect version of Valgrind install; follow directions below to get newer version.
   - Note that BBB image version 2018-01-28 includes a buggy version of Valgrind: Valgrind-3.12.0.SVN. If using this version, you must upgrade it to a newer version:
     - You should *still* use `apt-get` to install valgrind (above) so that most dependencies are met.
     - On the **target**, download newer version of Valgrind (prompt truncated to fit on page):

```
$ mkdir ~/valgrind_update
$ cd ~/valgrind_update
$ wget http://ftp.us.debian.org/debian/pool/main/v/valgrind/valgrind_3.14.0-3_armhf.deb
$ wget http://ftp.us.debian.org/debian/pool/main/v/valgrind/valgrind-dbg_3.14.0-3_armhf.deb
```

- Try to install the updated packages to check what dependencies are unmet:

```
(bbg)$ sudo dpkg -i valgrind*.deb
…
dpkg: dependency problems prevent configuration of valgrind:
 valgrind depends on libc6 (>= 2.28); however:
  Version of libc6:armhf on system is 2.24-11+deb9u4.

dpkg: error processing package valgrind (--install):
 dependency problems - leaving unconfigured
dpkg: dependency problems prevent configuration of valgrind-dbg:
 valgrind-dbg depends on valgrind (= 1:3.14.0-3); however:
  Package valgrind is not configured yet.

dpkg: error processing package valgrind-dbg (--install):
 dependency problems - leaving unconfigured
Errors were encountered while processing:
 valgrind
 valgrind-dbg
```

If it complains about missing dependencies *other than those shown above*, install those dependencies. OK to have incorrect version of `libc6`, `libc6-dbg`, `valgrind-dbg`

- If the above step is missing no more dependencies than suggested, then force the install of the updated Valgrind packages, in spite of missing dependencies:

```
(bbg)$ sudo dpkg --force-all -i valgrind*.deb
(Reading database ... 36025 files and directories currently installed.)
Preparing to unpack valgrind_3.14.0-3_armhf.deb ...
Unpacking valgrind (1:3.14.0-3) over (1:3.14.0-3) ...
Preparing to unpack valgrind-dbg_3.14.0-3_armhf.deb ...
Unpacking valgrind-dbg (1:3.14.0-3) over (1:3.14.0-3) ...
dpkg: valgrind: dependency problems, but configuring anyway as you requested:
 valgrind depends on libc6 (>= 2.28); however:
  Version of libc6:armhf on system is 2.24-11+deb9u4.

Setting up valgrind (1:3.14.0-3) ...
Setting up valgrind-dbg (1:3.14.0-3) ...
```

7. 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:
   `(host)$ ls -l helloWorld`
3. Strip the binary:
   `(host)$ arm-linux-gnueabihf-strip helloWorld`
4. Check the file size of your application; it should be (much?) smaller.
   `(host)$ 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.