RZ/A1

GDB Debugging for RZ/A1 Linux with J-Link

EU_00xxx
Rev.1.20
Jun 14, 2016

Introduction

The purpose of this document is to explain how to build GDB from source and debug u-boot and the Linux kernel with a Segger J-Link as the JTAG interface.

Target Device

RZ/A1L, RZ/A1M, RZ/A1H

Contents

1. Overview

The Renesas RZ/A1 RSK comes with a Segger J-Link LITE that can be used for SPI Flash programming and debugging of bare metal and RTOS based applications using IDEs such as e2Studio, IAR Workbench and DS-5. However, Segger provides a GDB Server interface that runs in both Windows and Linux. This application note explains how to use a Segger J-Link in a Linux host environment to perform source code debugging of Linux based software for the RZ/A1.

NOTES:

- These instructions were done using Linux host running Ubuntu 12.04-LTS
- Using the J-Link LITE that comes with the RSK, your maximum JTAG clock rate will be 2 MHz. However, if you use a J-Link BASE or PLUS, your maximum clock interface speed: 15 MHz. Please see https://www.segger.com/jlink-model-overview.html for other models. The Maximum input JTAG clock rate for the RZ/A1 is 20 MHz.
- The ARM CoreSight block in the RZ/A1 has 6 hardware breaks and 4 HW Watchpoints.

2. Prebuilt GDB from Linaro Toolchain

The RZ/A1 RSK BSP by default will download the Linaro Toolchain. Included within the toolchain is a pre-built gdb that will run on a host machine (x86) and communicate with an ARM target over J-Link. This binary will work fine for the operations discussed in this document.

Within the BSP, the gdb application (arm-linux-gnueabihf-gdb) can be found here:

{rskrza1_bsp}/output/gcc-linaro-arm-linux-gnueabihf-4.8-2014.02_linux/bin/arm-linux-gnueabihf-gdb

Note that the prefix of the GDB executable (all the stuff in the name before -gdb) signifies that while the gdb application was built to execute on one type of architecture (x86 in this case), the target architecture and toolchain it was meant to debug is different (ARM in this case).

You should add this directory location to your system path to make it easier to launch gdb. For example:

```
$ echo 'PATH=$PATH:~/rskrza1_bsp/output/gcc-linaro-arm-linux-gnueabihf-4.8-
2014.02_linux/bin/' >> ~/.bashrc
```

Then, either open up a new terminal window or re-run the bashrc script in the current window

```
$ source ~/.bashrc
```

Now test it out to make sure it works.

```
$ arm-linux-gnueabihf-gdb --version
GNU gdb (crosstool-NG linaro-1.13.1-4.8-2014.02 - Linaro GCC 2014.02) 7.6.1-
2013.10
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-build_pc-linux-gnu --target=arm-linux-
gnueabihf".
For bug reporting instructions, please see:
<https://bugs.launchpad.net/gcc-linaro>.
```

3. Build GDB using Buildroot (Optional)

If you have decided to use a uClibc based toolchain from within Buildroot (instead of a prebuilt Linaro glibc based toochian), Buildroot can also build you a GDB to go with it. In Buildroot's menuconfig, simply select

```
Toolchain -> [*] Build cross gdb for the host
```

Enable the TUI interface

NOTE As you can see in the Buildroot configuration file for GDB (buildroot-2014.05/package/gdb/gdb.mk), the pseudo graphical Text-based User Interface (TUI) interface will not be built. This interface can be helpful and is recommend. To add it back in, simply open the **gdb.mk** file and under HOST_GDB_CONF_OPT, remove the following lines:

```
HOST_GDB_CONF_OPT = \
      --target=$(GNU_TARGET_NAME) \
      --enable-static --disable-shared \
      --without-uiout \        <<<<< REMOVE THIS LINE
      --disable-tui \          <<<<< REMOVE THIS LINE
      --disable-gdbtk \        <<<<< REMOVE THIS LINE
      --without-x \            <<<<< REMOVE THIS LINE
      --enable-threads \
      --disable-werror \
      --without-included-gettext \
      --disable-sim
```

Alternatively, you could follow the instructions in the next section for building GDB directly from source.

After building, your gdb executable will be located here:

```
rskrza1_bsp/
   +- output/
    +- buildroot-2014.05/
      +- output/
        +- host/
         +- usr/
           +- bin/
             +- arm-buildroot-linux-uclibcgnueabi-gdb
```

You should add this directory location to your system path to make it easier to launch gdb. For example:

```
$ echo 'PATH=$PATH:~/rskrza1_bsp/output/buildroot-2014.05/output/host/usr/bin'
>> ~/.bashrc
```

Then, either open up a new terminal window or re-run the bashrc script in the current window

```
$ source ~/.bashrc
```

Now test it out to make sure it works.

```
$ arm-buildroot-linux-uclibcgnueabi-gdb --version
GNU gdb (GDB) 7.5.1
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-buildroot-
linux-uclibcgnueabi".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

4. Build GDB from Source (Optional)

Pre-built toolchain binary packages usually come with a prebuilt GDB application. However, the GDB application might have been missing some configuration options when it was built, so it's possible to re-build it directly from source.

Below are the host command line steps you would do in order to build GDB for your RZ/A1 from source. Please perform all these command in the same terminal window.

Important: When building GDB, you build a GDB for the cross compiler toolchain you are already using. You do not have to install the specific toolchains I the examples below. You simply change what you set ARM_TARGET to.

In your Linux host, please enter the commands after the '$' prompt sign.

4.1 Download and extract GDB 7.9.1

```
$ wget ftp://sourceware.org/pub/gdb/releases/gdb-7.9.1.tar.xz
$ tar xf gdb-7.9.1.tar.xz
$ cd gdb-7.9.1
```

4.2 Install Additional Packages for building

GDB requires the package **texinfo** to be installed on your host machine in order to build the GDB software. Please enter the following in order to install that package:

```
$ sudo apt-get install texinfo
```

In order to connect to a J-Link, the build option "--with-expat" is required when building GDB. This requires you to have the **libexpat** development package installed on your host machine to allow GDB to have access to the header and library files. Please enter the following command to install that packge:

```
$ sudo apt-get install libexpat1-dev
```

4.3 Set up Build Environment

Your system path must contain the location of your ARM toolchain that you use to build your RZ/A Linux software (u-boot, kernel, apps, etc..). Note, your toolchain might be different than the example below, and this is OK.

CHANGE TO THE LOCATION OF YOUR TOOLCHAIN

```
$ export PATH=$PATH:/home/renesas/buildroot-2014.05/output/host/usr/bin
```

You need to set the prefix of your RZ/A toolchain. This is basically the string that is on the front of the GCC commands. For example, if your cross compile GCC executable is "bin/arm-buildroot-linux-uclibcgnueabi-gcc", then your prefix would be "arm-buildroot-linux-uclibcgnueabi". Note, your toolchain might be different than the example below, and this is OK, just change what you type to match your toolchain.

SET TO THE PREFIX OF YOUR TOOLCHAIN BINARIES

```
$ ARM_TARGET=arm-buildroot-linux-uclibcgnueabi
```

4.4 Build GDB Client

This will build a GDB client interface that will **run on the host PC**, but only decode instructions for a 'target' of ARM (RZ/A1). We will use the '--prefix' build option in order to specify a location to copy (install) the output to.

```
$ mkdir build-gdb
$ cd build-gdb
$ ../configure --disable-sim --with-expat --prefix=`pwd`/z_install --target=$ARM_TARGET
$ make
$ make install
$ cd ..
```

Now, your the gdb executable will be located in **build-gdb/z_install/bin/** and will be named using the prefix of the ARM toolchain you built it against. For example, here is one that was from a toolchain that was build using Buildroot and uses uClibc for the C Libraries:

build-gdb/z_install/bin/arm-buildroot-linux-uclibcgnueabi-gdb

4.5 Build GDB Server (Optional)

This will build a GDB server interface that will **run on the ARM (RZ/A)**. This step is optional and the usage is not discussed in this document.

```
$ mkdir build-gdb-server
$ cd build-gdb-server
$ ../configure --disable-sim --prefix=`pwd`/z_install --host=$ARM_TARGET
$ make
$ make install
$ cd ..
```

Now the gdbserver executable will be in build-gdb/z_install/bin.

4.6 Build GDB Native (Optional)

This will build a GDB client interface that will **run on the ARM (RZ/A)**. This is called a native build as the gdb executable is intended to be run on the target (ARM RZ/A1) and debug local running executables. Since the resulting gdb executable will be about 21MB in size, this might not be optimal in a memory constrained (Flash/RAM) environment. This step is optional and the usage is not discussed in this document.

```
$ mkdir build-gdb-native
$ cd build-gdb-native
$ ../configure --disable-gdbtk --without-tcl --without-tk --disable-tui --
disable-sim --host=$ARM_TARGET --target=$ARM_TARGET --enable-threads
$ make
$ make install
$ cd ..
```

Now the gdb native executable will be in build-gdb-native/z_install/bin.

5. Install J-Link Software for Linux

Please install the Segger Jlink drivers and utilities for Linux. This example is for a host running a 32-bit Ubuntu (Debian).

Download "Software and documentation pack for Linux V5.00i, DEB Installer 32-bit version" from

https://www.segger.com/jlink-software.html

Installing a later version than V5.00i is OK.

In a command window, install the downloaded file:

```
$ sudo dpkg -i jlink_5.0.9_i386.deb
```

After the install, on a command line you can type "JLink" then hit the 'tab' key on your keyboard to see all the utilities it installed.

```
$ JLink
JLinkDebugger      JLinkGDBServer     JLinkRTTClient     JLinkSWOViewer
JLinkExe           JLinkRemoteServer  JLinkSTM32
```

6. Start the J-Link GDB Server

Plug your JLink into both the PC and target board. If you are using a Virtual Machine like VMware or VirutalBox, make sure the USB JLink is connected to your VM environment.

In a separate terminal window, simple enter the command:

```
$ JLinkGDBServer -speed 15000 -device R7S721001
```

This will start up the J-Link GDB Server that your GDB application will talk to over **TCP port 2311.**

This terminal window will stay open during your GDB debugging session. To end the GDB server connection, simply type Ctrl+C.

Important You have to shut down the GDB SERVER (Ctrl + C) if you want to use the Jlink to re-program your board's flash. If you try to use the program scripts included in the BSP, they will fail until the GDB Server is exited.

7. Start GDB and Connect to JLinkGDBServer

For all the possible connections described in this document, you should always do this following step to connect to your board:

1. Change Directory into the exact location of your u-boot executable.
2. Launch your gdb with the '-tui' interface option enabled. For example:

```
$ cd ~/rskrza1_bsp/output/linux-3.14
$ arm-linux-gnueabihf-gdb -tui
```

Of course, you can choose not to use the TUI interface if you want. You may also choose to add the location the GDB that you built to your system path instead of having to specifying each time.

When GDB first starts, you will have to hit 'Enter' to close its opening message.

To connect to JLinkGDBServer, enter the following commands into GDB:

```
(gdb) target remote localhost:2331
(gdb) monitor go
```

The 'target' commands will create a connection to JLinkGDBServer software communicates to your board. The 'monitor go' command is a custom command that GDB sends directly to JLink that simply allows the CPU to resume its normal operation after the JLink connection. You must use that 'monitor go' command at least once after connecting with JLinkGDBServer.

As a suggestion, you could create a file named ".gdbinit" and place it in the (local) directory where you will be debugging. Then inside that file you could put these commands (one command per line). That would allow GDB to automatically run these commands for you every time. Just note that GDB disables that ability by default, so to enable the local .gdbinit feature, you will need to enable it by writing the line "set auto-load safe-path /" in your global GDB settings which basically says it trusts any .gdbinit file in the system (or you could instead specify the directory where all your projects are under)

For Example:

```
$ cd ~/rskrza1_bsp/output/linux-3.14
$ echo "target remote localhost:2331" > .gdbinit
$ echo "monitor go" >> .gdbinit

$ echo "set auto-load safe-path ~/rskrza1_bsp/" >> ~/.gdbinit

       {or just do it for any path}
$ echo "set auto-load safe-path /" >> ~/.gdbinit
```

8. Debugging u-boot

After reset, u-boot starts its execution from Flash. Even if you are booting the RZ/A1 in serial boot mode, each instruction is individually fetched from SPI flash before execution, so technically it is still running from Flash.

However, the first thing u-boot does is logically partition the end of system RAM memory into stack, heap, data, etc.., then copy the entire u-boot binary image (code and all) from flash to RAM and then jump to it. From there, it never executes out of Flash again (until the next reset of course). This is why u-boot can easily reprogram Flash: Because it is no longer executing out of it.

This is why all the u-boot code has to be position independent when it is built because of this relocation. However, this also means that you (and GDB) won't know the addresses of any functions until run-time. So when you are debugging u-boot, you need to find out where the code is being relocated to so you can adjust all the symbol addresses that GDB is using to allow for source level debugging.

8.1 Early boot – Running from Flash

If you want to debug u-boot while it is running from Flash and before it relocates to RAM, then the symbol addresses in the debug file will match and there is not much extra that you have to do. Here is an example of connecting to the board, setting a breakpoint, resetting the board and letting it run to the breakpoint.

NOTE: The text after "<<<" are just some comments, not command to be entered

1. Program the SPI Flash with your u-boot.bin file.
2. Start Jlink GDBServer (described in Section 6)
3. Start GDB and connect to Jlink (described in Section 7)

```
(gdb) target remote localhost:2331
(gdb) monitor go
```

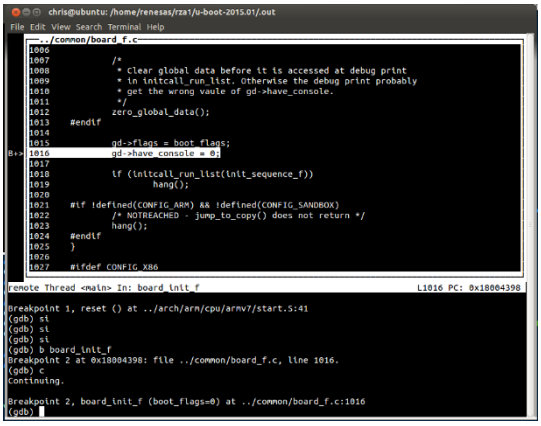
4. Load your u-boot symbol file in GDB (note that this is the debug file that contains all the symbol data, not the .bin file that you programmed). Say yes (type 'y') when prompted.

```
(gdb) file u-boot
```

5. Set breakpoint, reset board, run till breakpoint, do some stepping

```
(gdb) monitor reset          <<< custom JLINK command to reset board
(gdb) b reset                <<< Set a breakpoint at 'reset'
(gdb) c                      <<< continue command (ie, RUN)
              { BREAK POINT SHOULD GET HIT }
(gdb) si                     <<< step (just for example)
(gdb) si                     <<< step
(gdb) si                     <<< step
(gdb) b board_init_f         <<< This function runs from flash (before
                                 RAM relocation, hence the name XXX_f)
(gdb) c
              { BREAK POINT SHOULD GET HIT }
```

Your screen should look something like this at this point:



8.2 Run-time – After Relocated to RAM

Since the location that the code will get relocated to in RAM is not known until run-time, we will have to set a breakpoint and look at the value of a variable. Then we can use that value to reload our symbols with an appropriate offset.

1. Program the SPI Flash with your u-boot.bin file.
2. Start Jlink GDBServer (described in Section 6)
3. Start GDB and connect to Jlink (described in Section 7)

```
(gdb) target remote localhost:2331
(gdb) monitor go
```

4. Load your u-boot symbol file in GDB (note that this is the debug file that contains all the symbol data, not the .bin file that you programmed). Say yes (type 'y') when prompted.

```
(gdb) file u-boot
```

5. Set breakpoint, reset board, run till breakpoint, read out the address the code will get relocated to

```
(gdb) monitor reset
(gdb) b relocate_code          <<< This function that does the relocation
(gdb) c
            { BREAK POINT SHOULD GET HIT }
(gdb) print/x $r0              <<< Register R0 has the destination RAM address
                               <<< Remember the value it displays

(gdb) b relocate_done
(gdb) c
            { BREAK POINT SHOULD GET HIT }
(gdb) si                      <<< step – notice how you lost source code view
(gdb) symbol-file             <<< say yes ('y'). This deletes current symbol info
(gdb) add-symbol-file u-boot 0x2096b000   <<< say yes ('y'). Use your relocation
                                              address value you discovered
(gdb) b board_init_r       <<< This function runs from RAM (after
                                RAM relocation, hence the name XXX_r)
(gdb) c
            { BREAK POINT SHOULD GET HIT }
```

Generally, the RAM address that the code will relocate to will not vary unless you do something like specify my heap or stack space in your configuration header file. So after you do all these steps once and find that address value, you could skip those steps and just relocate the symbol file from the beginning like this:

```
(gdb) target remote localhost:2331
(gdb) monitor go
(gdb) file u-boot
(gdb) monitor reset
(gdb) symbol-file
(gdb) add-symbol-file u-boot 0x2096b000
(gdb) b board_init_r
(gdb) c
(gdb) delete 1
```

HINT You could add these commands to your local .gdbinit file so you don't have to type them in every time you start up GDB.

HINT Instead of the "break" command 'b', you could use the "temporary break" command 'tb'. What this will do is automatically delete the breakpoint after it has been reached so you do not have to use the 'delete' command to remove it.

9. Debugging the Linux Kernel

Since the Linux kernel uses the MMU, things get a little tricky when trying to set breakpoint on virtual address before the MMU has been enable. Even though the MMU is enable early in the booting process, you cannot set break points for virtual addressed GDB before the MMU is enable. The reason is because when you try to set a break point, GDB will try to go out and read that address, and if the MMU is not setup yet, GDB will not be able to read that address and will error out.

With that being said, the Coresight ARM debug hardware has no issue with setting breakpoints for address that are not valid yet, and the J-Link will do that for you, but you have to do it without the GDB client knowing about. This is why if you need to set a breakpoint early in the booting process, you have to use the 'monitor' commands within GDB which basically sends custom commands directly to J-Link.

Another thing to point out is that if you want to do source level debugging before the MMU is enabled, the PC values won't match what the symbols addresses are in the vmlinux ELF file. Therefore, like with u-boot, you will need to manually set an address adjustment.

9.1 Enabling Debug Symbols the Kernel vmlinux output file

But default, the output vmlinux ELF file produced by a kernel build will not have debug symbols. Therefore, you will need to configure the Linux kernel to include them during the build process. To do that, you need to set the build configure CONFIG_DEBUG_INFO. You can find that in the menuconfig under "Kernel hacking" >> "Compile-time checks and compiler options" >> "Compile the kernel with debug info".

Alternatively, you could just add the line CONFIG_DEBUG_INFO=y to the very end of your .config file, then simply open menuconfig and then close with save.

Make sure after you add this new option, you do a make 'clean' and then rebuild your kernel.

9.2 Debugging XIP Kernel before MMU is enabled

The follow procedure is for setting a breakpoint at the very beginning of an XIP kernel that was started from u-boot.

1. Program the SPI Flash with your xipImage binary file.
2. Start Jlink GDBServer (described in Section 6)
3. Start GDB and connect to Jlink (described in Section 7)

```
(gdb) target remote localhost:2331
(gdb) monitor go
```

4. Load your vmlinux symbol file in GDB (note that this is the debug file that contains all the symbol data, not the xipImage file that you programmed). Say yes (type 'y') when prompted.

```
(gdb) file vmlinux
```

5. Set a breakpoint at the first address of the XIP kernel. This is where u-boot will jump to.

```
(gdb) break *0x18200000          <<< Set a breakpoint based on address only
(gdb) c                          <<< continue (run) command
```

6. Now in the u-boot console window, start your XIP kernel as you normally do. However instead of booting, it should hit the breakpoint and your GDB client will say "Breakpoint 1, 0x18200000 in ?? ()".
7. Since the MMU is not on yet, we need to relocate our vmlinux symbol table to the physical address we are actually executing from. To do this, we first need to find out what that address should be (it's not going to be 0x18200000). What you need to do is look at the address for the symbol "_stext" int eh file System.map that should be in the same directory as your vmlinux. You can actually use the 'grep' command directly from GDB, simply type:

```
(gdb) shell grep _stext System.map
bf000200 T _stext                        <<< This is the output
                (gdb)                    <<< Notice now your prompt is screwed up
```

NOTE After doing this command, your "(gdb)" prompt will not be offset (has to do with the way gdb gets the output of a shell command). To reset it, simply enter **Ctrl + L** which basically is like "screen refresh" in the TUI interface (but only after you record what the output of the shell command was because that will get erased with the refresh).

8. Reload our symbols with new address offsets based on the offset of _stext. For example, if _stext was found to be "bf000200", then we'll want to specify a relacted address as 0x18200200

```
(gdb) symbol-file                              <<< say yes 'y'
(gdb) add-symbol-file vmlinux 0x18200200       <<< say yes 'y'
```

9. You should now see the source code. To show that the PC is really at address 0x18200000, you can change the view to show both source code and raw instructions using he command "layout split". To go back to just a source view, you can type "layout src". To cycle through all the different view, type 'layout next' and just keep hitting Enter until you find the view you like.

```
(gdb) layout split
(gdb) layout next
(gdb) layout next          <<< You could also just keep hitting Enter
(gdb) layout next              which repeats the last entered command
(gdb) layout next
(gdb) layout src           <<< This will put you back to the normal Source view
```

10. You can now step and set breakpoints in functions that will execute before the MMU is enabled. Just for example:

```
(gdb) n                      <<< next command (step over)
(gdb) n                      <<< (you can just hit Enter instead of typing 'n'
(gdb) n                           each time.)
(gdb) break __v7_setup
(gdb) c
            { BREAK POINT SHOULD GET HIT }
```

The address "__mmap_switched" is the location of the next instruction directly after the MMU has been turned on. At that point, you will need to switch back to the original vmlinux file symbol addresses.

9.3 Debugging kernel boot after MMU is enabled

These instructions apply to both an XIP kernel and RAM uImage kernel.

If you have a static device driver and you want to set a breakpoint during the init or probe routine, then it is best to do that after the MMU has been enabled. This will then allow you to set breakpoint more easily because the virtual address of the symbols in the debug file will match.

Unfortunately, GDB will not let us set a breakpoint using a function name for an address that it can't read (as in, it has a virtual address but the MMU is not on yet). However, it will let us set a breakpoint if we give it the address directly (with a * in front of it). So what we will do is figure out the virtual address of the function we want using the "info address" command and then set it that way.

1. Program the SPI Flash with your xipImage binary file.
2. Start Jlink GDBServer (described in Section 6)
3. Start GDB and connect to Jlink (described in Section 7)

```
(gdb) target remote localhost:2331
(gdb) monitor go
```

4. Load your vmlinux symbol file in GDB (note that this is the debug file that contains all the symbol data, not the xipImage file that you programmed). Say yes (type 'y') when prompted.

```
(gdb) file vmlinux
```

5. Here we'll set a breakpoint at the function **start_kernel( )** which is early in the boot process before drivers are loaded, but after the MMU is enabled.

```
(gdb) info address start_kernel
Symbol "start_kernel" is a function at address 0xbf453438.
(gdb) break *0xbf453438
Breakpoint 3 at 0xbf453438: file /home/renesas/rza1/linux-3.14/init/main.c, line
481.
(gdb) c
Continuing.

Breakpoint 3, start_kernel () at /home/renesas/rza1/linux-3.14/init/main.c:481
(gdb)
```

6. Once we are at kernel_start( ), we can set breakpoints using the actual function names. For example:

```
(gdb) break rskrza1_add_standard_devices
Breakpoint 4 at 0xbf45930c: file /home/renesas/rza1/linux-3.14/arch/arm/mach-
shmobile/board-rskrza1.c, line 1
632.
(gdb) c
Continuing.

Breakpoint 4, rskrza1_add_standard_devices ()
    at /home/renesas/rza1/linux-3.14/arch/arm/mach-shmobile/board-rskrza1.c:1632
(gdb)
```

HINT You could put these commands in your local .gdbinit file if you are trouble shooting a specific driver init routine and need to do this multiple times. Then after you start GDB, you just need to boot your kernel in u-boot and the breakpoint will automatically be caught. Since the address of start_kernel (or your driver's function) might change address after each build, you might have to make a script that after each kernel build, the function's address is looked up from the System.map file and then entered into the .gdbinit file before you launch GDB.

9.4 Setting Breakpoints in Drivers on Boot

If you want to set a breakpoint in a driver that you want to stop in when it boots (the probe function for example), you might set a breakpoint in kernel_start( ), then after it hits that breakpoint, set another breakpoint inside your driver. While that is the correct way of doing it…it might not actually hit your second breakpoint.

The reason is that when the driver **arch/arm/kernel/hw_breakpoint.c** is run, it will clear out all HW breakpoints set in the core. This is done in function arch_hw_breakpoint_init( ) on the line:

```
/*
 * Reset the breakpoint resources. We assume that a halting
 * debugger will leave the world in a nice state for us.
 */
on_each_cpu(reset_ctrl_regs, NULL, 1);
```

You will even notice that you can set break point at the beginning of function arch_hw_breakpoint_init( ), but when you try to step through that function, as soon as you try to step over that line, the kernel will no longer break and just keep running because all breakpoints were just cleared and that was what GDB was using to do source level stepping.

That hw_breakpoint driver is included automatically for ARM devices by HAVE_HW_BREAKPOINT in file arch/arm/Kconfig.

Therefore, to set a breakpoint in your driver from kernel_start( ), you need to temporary disable that function. You can either do that by commenting out the HAVE_HW_BREAKPOINT line in Kconfig, or by put a 'return 0' at the beginning of the arch_hw_breakpoint_init ( ) function.

For example:

```
 static int __init arch_hw_breakpoint_init(void)
 {
+      pr_info("WARNING - HW BREAKPOINT SUPPORT DISBLED IN KERNEL.\n");
+      return 0;
+
       debug_arch = get_debug_arch();
```

9.5 Attach to an already running Linux Kernel

You can create a GDB connection while your Linux kernel is already up and running (there is no need to reboot). You simply start up the JlinkGDBServer and connect to it. Since the MMU is enabled, you can set breakpoint inside the kernel by simply specifying the function name.

For example, this connects to a running kernel and sets a breakpoint in the function that relocates a module into kernel space after insmod is called.

1. Start Jlink GDBServer (described in Section 6)
2. Start GDB and connect to Jlink (described in Section 7)

```
(gdb) target remote localhost:2331
(gdb) monitor go
(gdb) file vmlinux
(gdb) break move_module
(gdb) c
```

10. Extra

10.1 Print Memory as ASCII and save to a file

Viewing large amount of memory inside GDB is a bit of a pain. As a suggestion, here is a macro that you can use to print out large amounts of data and to a file to examine. By adding this macro to your ~/.gdbinit file and have access to it every time you launch GDB.

```
define mydump
dump binary memory dump.bin $arg0 $arg0+$arg1
shell echo -e $arg0 $arg0+$arg1\\n\\n > dump.bin.txt
shell hexdump -C dump.bin >> dump.bin.txt
end
```

To use it within GDB, simple just call the macro with the first argument being the name of the buffer/variable (or it's address) and the second argument being the amount of data to save.

```
(gdb) mydump my_buffer 100
(gdb) 0x20020000 100
```

10.2 Using DDD (Graphical Front End)

GNU DDD is a graphical front-end for command-line debuggers such as GDB. It is free and open source.

https://www.gnu.org/software/ddd/

Even though the program hasn't been updated since 2009, it still works fine since all it is doing is passing text back and forth to the command line GDB.

To use it simple do the following:

1. Install DDD. If you are using Ubuntu, enter the following command

```
$ sudo apt-get install ddd
```

2. Start the JLink GDB server as specified in section 6.

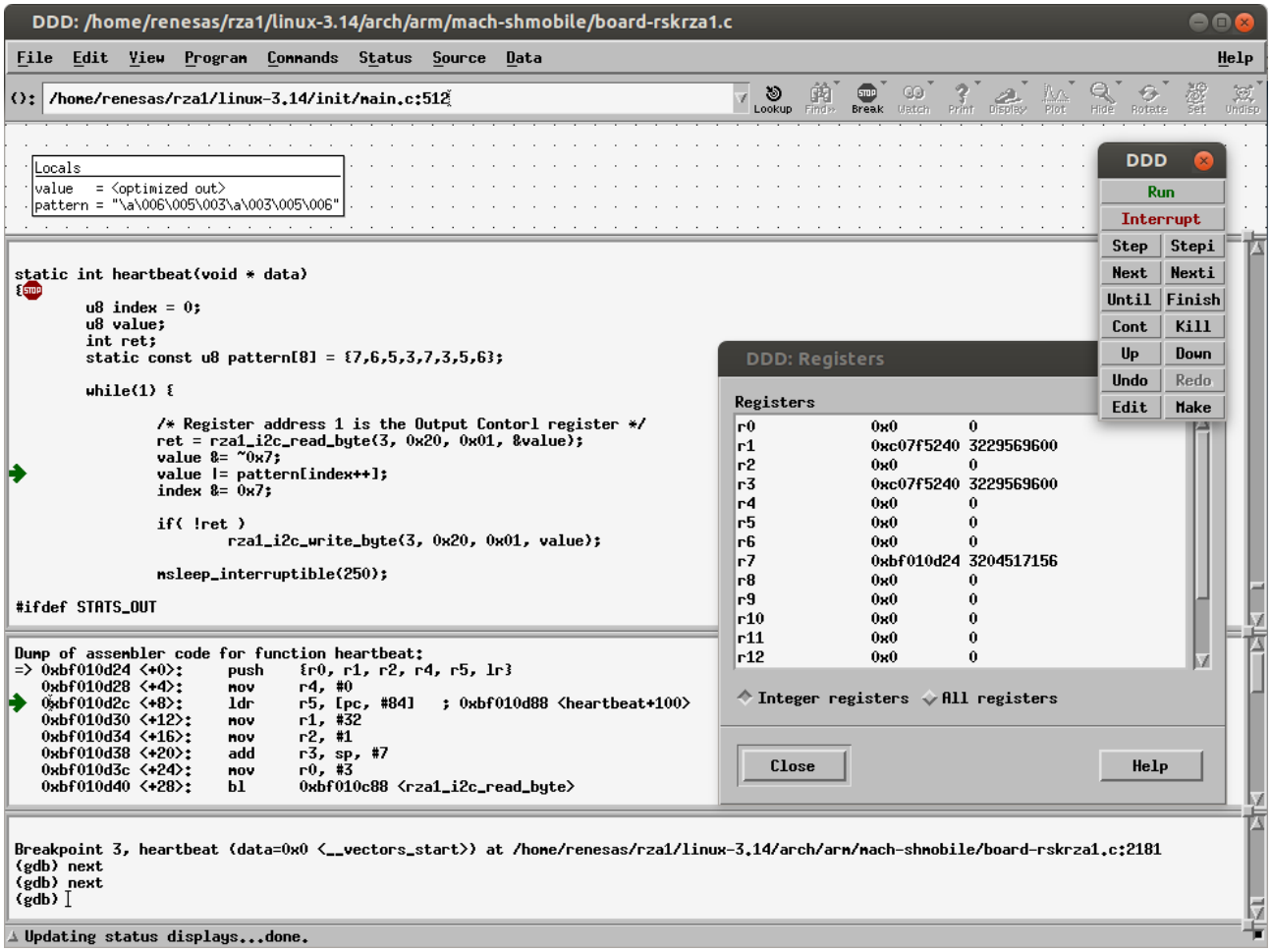
3. Start DDD, but include the full path to your GDB command line. If your GDB executable is already part of your system PATH, then you can just specify the file name

```
$ ddd --debugger arm-buildroot-linux-uclibcgnueabi-gdb
```

From there, you can enter in all the commands in the command window at the bottom. You connect to the J-Link GDB server the same was you do with the command line version.

Some note:

- Press F9 (or Program >> Continue) for continue (the run button doesn't work)
- You can hover over variables to see their values
- Right click (and hold down) on a line to set/clear breakpoints
- Show disassembly: (menu) Source >> Display Machine Code
- Show registers: (menu) Status >> Register
- The Interrupt button (or ESC key) is how you dynamically stop execution



Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/contact/

Revision Record

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Nov 19, 2015	—	First edition issued
1.10	Jun 10, 2016		Added “9.4 Setting Breakpoints in Drivers on Boot”
			Updated command line in “6. Start the J-Link GDB Server”
1.20	Jun 14, 2016		Added section 10.2 Using DDD