GDB

Stable: 31.1.1-2020-12.04 / Revision: 23.03.2020 - 22.04

Contents

1 Article purpose

This article provides the basic information needed to start using the **GDB**[1] application tool.

It explains how to use this GNU debugger tool connected to your ST board target via Ethernet or via ST-LINK, and how to perform cross-debugging (IDE, gdb viewer tool or command line) on Arm® Cortex®-A7 side for Linux® application, Linux® kernel (including external modules), or Arm® Cortext®-M4 firmware.

2 Introduction

The following table provides a brief description of the tool, as well as its availability depending on the software packages:

✓: this tool is either present (ready to use or to be activated), or can be integrated and activated on the software package.

✗: this tool is not present and cannot be integrated, or it is present but cannot be activated on the software package.

Tool			STM32MPU Embedded Software distribution			STM32MPU Embedded Software distribution for Android™		
Name	Category	Purpose	Starter Package	Developer Package	Distribution Package	Starter Package	Developer Package	Distribution Package
GDB	Debugging tools	The GNU Project debugger, **GDB**[1], allows monitoring program execution, or what the program was doing at the moment it crashed.	✗*	✓	✗**	✗	✗	✗
			** Cross compile gdb and openocd binaries are required and only available from Developer Package.* *** It is recommended to use the Developer Package to run the gdb debug session, which provided all dependencies*			**The STM32MPU distribution for Android™ is not yet available in the v2 ecosystem releases: please refer to the GDB page for the v1 ecosystem releases (in archived wiki).**		

The GDB can perform four main types of actions (plus other corollary actions) to help you detect bugs when running your software or application:

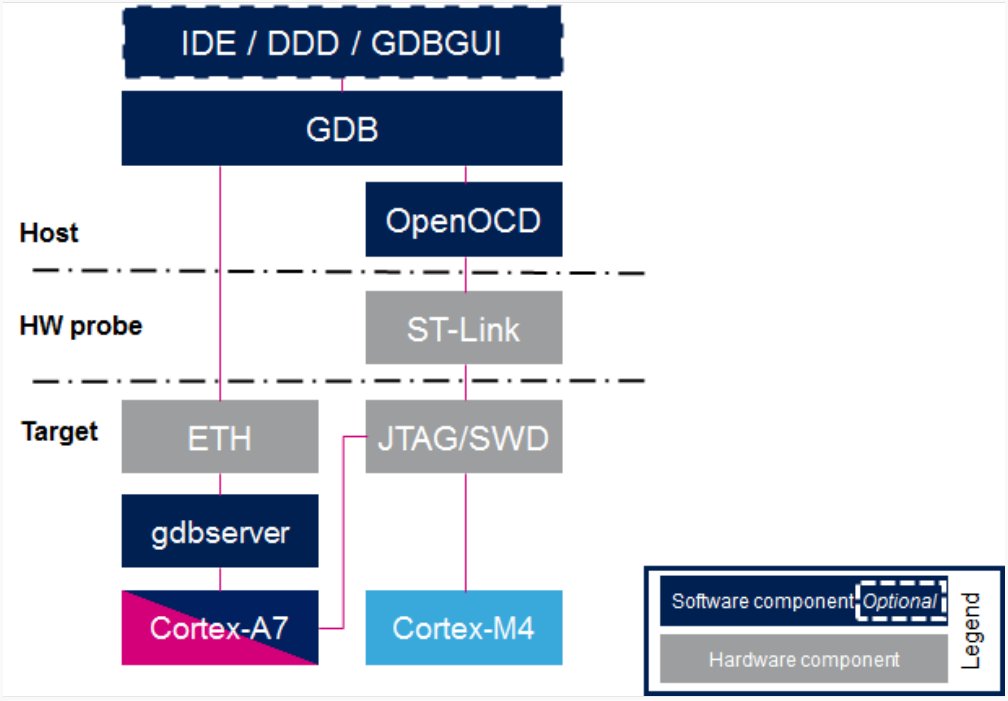
- Start the program, specifying anything that might affect its behaviour.
- Make the program stop on specific conditions.
- Examine what happened when the program stopped.
- Update the program in order to test a bug correction, and jump to the next one.

3 Overview of GDB setup for STM32MPU

3.1 GDB setup paths

Two paths can be used in the STM32MPU environment for GDB setup:

- **gdb gdbserver** path through Ethernet, **used for Cortex-A7 Linux applications**.
  In that case, two software components are required, one on the target and the other on the host PC.
- **gdb JTAG/SWD** path through OpenOCD and ST-LINK, **used both for Cortex-M4 firmware and Cortex-A7 Linux kernel**.
  In that case, only one software component is required on the host PC.



GDB paths scheme

Two components are included in OpenSTLinux Developer Package for GDB setup:

- **gdbserver**: **embedded on target rootfs** and used as remote access for a host connection
- **arm-ostl-linux-gnueabi-gdb**: **embedded on host PC side**, cross-compiled gdb binary that manages the connexion between the host computer and the target board

3.2 JTAG and SWD debug port

The STM32MPU features two debug ports through the embedded CoreSight™ component that implements an external access port for connecting debugging equipment:

- A 5-pin standard JTAG interface (JTAG-DP)
- A 2-pin (clock + data) “serial-wire debug” port (SW-DP)

These two modes are mutually exclusive since they share the same I/O pins.

Refer to STM32MP15 reference manuals for information related to JTAG and SWD.

4 Installing the GDB tool

This tool is made of two parts, one on the host PC, and a second on the target (only for debugging Linux applications).

4.1 Installing the GDB tool on your host PC

Below is the list of required components:

- The cross-compiled GDB binary
- The OpenOCD binary and configuration files
- The symbols files of all BSP components (TF-A, U-Boot and Linux kernel), corresponding to the images of the OpenSTLinux Starter Package.

4.1.1 Using the STM32MPU Embedded Software distribution

4.1.1.1 Developer Package

Only the Developer Package can be used, since it is the only one which provides all the required components.

4.2 Installing the GDB on your target board

On the target board, only the gdbserver binary is required for debugging Linux applications.

It is available by default within the Starter Package, which provides images linked to the Developer Package.

5 Getting started

This chapter describes the two ways for debugging OpenSTLinux BSP components (TF-A, U-Boot and Linux kernel including external modules), Linux applications and Cortex-M4 firmware: by using GDB commands or by using a GDB user interface such as gdbgui, DDD or IDE.

5.1 Prerequisites

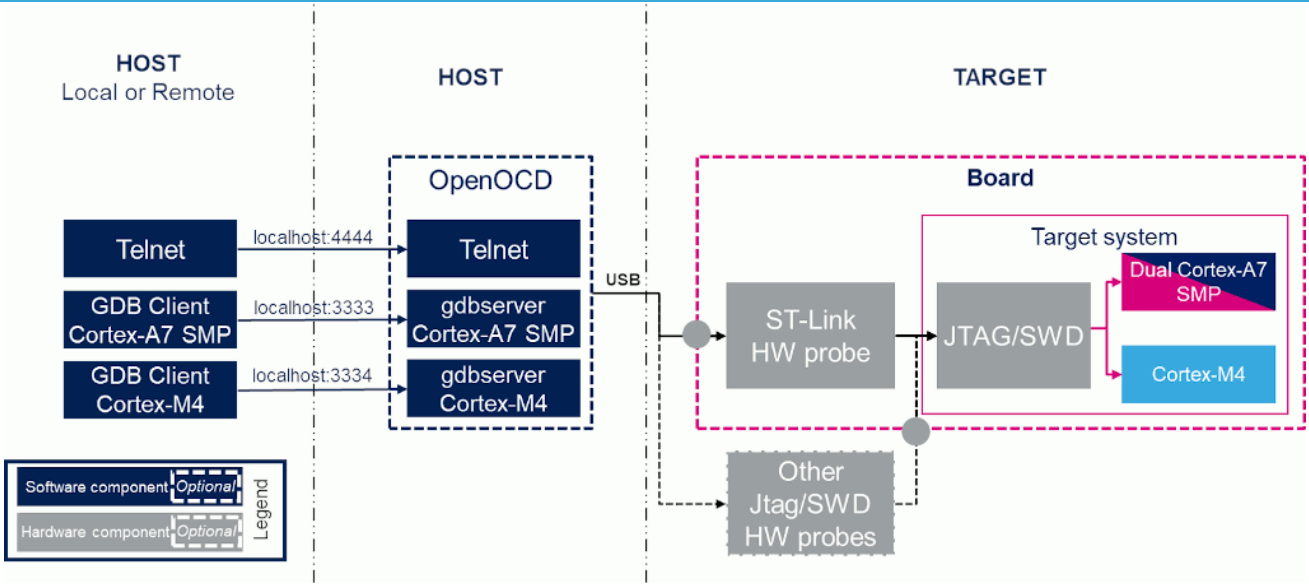
The target board is up and running.

5.2 Debug OpenSTLinux BSP components

5.2.1 Setting up GDB / OpenOCD debug path environment

- **Architecture**

The figure below shows the architecture corresponding to the GDB/OpenOCD connected to Cortex-A7 and Cortex-M4 cores.
Note: The ST-LINK probes available on the STM32MP1 board can be used through a USB interface, as well as any other external probes through the Trace or JTag connector.



- **Prerequisites**

  The Developer Package must be installed. It provides the SDK, the debug symbol files and the source files for TF-A, U-Boot and Linux kernel (refer to STM32MP1 Developer Package).

  The debug symbol files contain the symbols for the TF-A, U-Boot and Linux kernel binaries (from the Starter Package image) that have been flashed on the board.

- **Environment configuration files**

  To speed up environment setup for debugging with GDB, download two configuration files, and install them on your PC (*under the home directory: $HOME/gdbscripts/*). You can then **customize** them:

  - Setup.gdb: main configuration file in which you can define the debug context you want to use (Refer to Boot chain overview for details). Possible combinations are:

    - Trusted boot chain:

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 TF-A(BL32)	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
(0) - Boot	✓	✓	✓	✓
(1)				

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 TF-A(BL32)	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
-Running target	⊗	⊗	✓	✓

- Trusted boot chain with OP-TEE:

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 OP-TEE	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
(0)-Boot	✓	✓	✓	✓
(1)-Run				

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 OP-TEE	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
ningtarget	⊗	⊗	✓	✓

- Path_env.gdb: customization file to define all the paths to source and symbol files, which can either be directly retrieved from the **Developer Package** (refer to the Example of directory structure for Packages), or result from a new compilation. **In both cases, update the paths corresponding to your environment.**
Please read carefully the comments provided in this file to help you updating source and symbol paths.

Store these files locally on your host PC, **check for name cast (Setup.gdb and Path_env.gdb)** and update them accordingly.

5.2.2 Configuring GDB and OpenOCD for attachment on a running target

When the target board is running, you can attach the GDB only during one of following phases:

5.2.2.1 U-Boot execution phase

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 3
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 1
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When the configuration is complete, jump to Running OpenOCD and GDB.

5.2.2.2 *Linux kernel execution phase*

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 4
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 1
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When the configuration is complete, jump to Running OpenOCD and GDB.

5.2.3 Configuring GDB and OpenOCD for attachment on boot

You can attach the GDB during target boot only in the following cases:

- TF-A(BL2) boot case;
- TF-A(BL32) boot case;
- OP-TEE boot case;
- U-Boot boot case;
- Linux kernel boot case.

To handle the cases above, the FSBL image has to be wrapped through the tool stm32wrapper4dbg. This operation allows the debugger to halt the target at the very first instruction of FSBL.

In these cases, the target board will automatically reboots when the GDB starts.

5.2.3.1 *TF-A(BL2) boot case*

In that case, the GDB breaks in bl2_entrypoint function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 1
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.2 TF-A(BL32) boot case

In that case, the GDB breaks in sp_min_entrypoint function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 2
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.3 OP-TEE boot case

In that case, the GDB breaks in _start function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 2
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 1
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.4 *U-Boot boot case*

In that case, the GDB breaks in in _start function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 3
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.5 *Linux kernel boot case*

In that case, the GDB breaks in stext function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 4
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.4 Running OpenOCD and GDB

- Prerequisites

Before running OpenOCD and GDB, check that the target board is in the right state.

For all configurations except GDB attachment to a running SSBL (U-Boot), the board has to operate in OpenSTLinux running mode.

In case of attachment to a running SSBL (U-Boot) configuration, the board target must be in U-Boot console mode:

```
#Reboot the target board

#Press any key to stop at U-Boot execution when booting the board
STM32MP> ...
STM32MP> Hit any key to stop autoboot:  0
STM32MP>
```

When you are in the expected configuration, two different consoles must be started: **one for OpenOCD** and **one for GDB**.

The SDK environment must be installed on both console terminals.

- OpenOCD console

```
# First console for starting openocd with configuration file
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> openocd -f <board.cfg>
```

Possible target configuration files for *<board.cfg>*:

Target board	Adapter	SWD mode board.cfg	JTAG mode board.cfg
STM32MP157C-EV1	ST-LINK *	board /stm32mp15x_ev1_stlink_swd.cfg	board /stm32mp15x_ev1_stlink_jtag.cfg
STM32MP157C-EV1	U-LINK2	board /stm32mp15x_ev1_ulink2_swd.cfg	board /stm32mp15x_ev1_ulink2_jtag.cfg
STM32MP157C-EV1	J-LINK	board /stm32mp15x_ev1_jlink_swd.cfg	board/stm32mp15x_ev1_jlink_jtag.cfg
STM32MP157X-DK2	ST-LINK *	board/stm32mp15x_dk2.cfg	⊗**

** Both v2 and v3 are supported.*

*** JTAG wires are not connected in DK2.*

Note: It is recommended to use SWD, which is faster than JTAG.

- GDB console

The GDB must be executed from the directory where the scripts have been installed (i.e. $HOME/gdbscripts/)

```
# Second console for starting the GDB
PC $> cd $HOME/gdbscripts/
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> $GDB -x Setup.gdb
```

5.2.5 To know more about Linux kernel debug with GDB

Please refer to Debugging the Linux kernel using the GDB.

5.2.6 Access to STM32MP registers

5.2.6.1 Using gdb command line

- The following monitoring commands can be used to read a register value:

```
(gdb) monitor mdb <phys_address> [count] #Display target memory as 8-bit bytes
(gdb) monitor mdh <phys_address> [count] #Display target memory as 16-bit bytes
(gdb) monitor mdw <phys_address> [count] #Display target memory as 32-bit bytes
```

For example: Read RCC_MP_APB1ENSETR register on STM32MP1 to check RCC APB1 peripheral enable status.

```
(gdb) monitor mdw phys 0x50000a00 #full 32bits value result requested
0x50000a00: 00010000 --> UART4 is enable as explained in STM32MP15 reference manuals
```

- The following monitoring commands can be used to set a register value:

```
(gdb) monitor mwb <phys_address> <value> [count] #Write byte(s) to target memory
(gdb) monitor mwh <phys_address> <value> [count] #Write 16-bit half-word(s) to target
memory
(gdb) monitor mww <phys_address> <value> [count] #Write 32-bit word(s) to target memory
```

For example: Write RCC_MP_APB1ENCLRR register on STM32MP1 to clear the UART4 RCC of APB1 peripheral, then reenable it:

```
(gdb) monitor mww phys 0x50000a04 0x00010000 #full 32bits value given

# You can then check that UART4 is disable by reading the RCC_MP_APB1ENSETR register:
(gdb) monitor mdw phys 0x50000a00
0x50000a00: 00000000

# You can also check that the console is disabled by going on running the GDB
(gdb) c

# When the GBD has stopped running, you can re-enable UART4 RCC:
(gdb) monitor mww phys 0x50000a00 0x00010000
```

5.2.6.2 Using CMSIS-SVD environment

The CMSIS-SVD environment is useful to get detailed information on registers, such as name and bit descriptions.

It is based on python scripts and svd files which contain the description of all registers.

Refer to CMSIS-SVD environment and scripts for more details.

5.3 Debug Cortex-M4 firmware with GDB

The Arm Cortex-M4 core firmware can also be debugged using the GDB in command line (without IDE).

Either engineering boot mode and production boot mode are supported.

Please refer to the **Hardware Description** (Category:STM32 MPU boards) of your board for information on the Boot mode selection switch.

5.3.1 Debug Cortex-M4 firmware in engineering boot mode

As in previous chapter Running OpenOCD and GDB, both OpenOCD and GDB have to be started on separate consoles.

OpenOCD has to be executed in the same way as in the mentionned chapter.

GDB, instead, has to be executed with a different command line, without the script but with the path of ELF file containing the firmware to be debugged:

```
# Second console for starting the GDB
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> $GDB $HOME/path/to/file.elf
```

Then, the following commands has to be typed in the GDB console to start the execution of a firmware and halt it at the beginning of main():

```
(gdb) target extended-remote localhost:3334
(gdb) load
(gdb) thbreak main
(gdb) continue
```

Once the execution halts at main(), GDB will return the prompt allowing the debug of the firmware.

5.3.2 Debug Cortex-M4 firmware in production boot mode

In production mode the firmware is started by Linux, independently from GDB. Nevertheless, GDB can set a breakpoint before the firmware is started by Linux; thus the firmware will halt at the breakpoint, allowing GDB to debug the firmware.

Please check in Linux remoteproc framework overview how to start and stop a firmware using Linux remoteproc framework.

At first, verify that no firmware is running on Cortex-M4 or, eventually, stop it.

Then, start OpenOCD and GDB as in previous chapter Debug Cortex-M4 firmware in engineering boot mode.

Type the following commands in the GDB console:

```
(gdb) target extended-remote localhost:3334
(gdb) thbreak main
(gdb) continue
```

Finally, let Linux start the firmware. The firmware execution will halt at main() and GDB will return the prompt allowing the debug of the firmware.

Use on GDB command line the exact same ELF file that is used by Linux to read and run the firmware. If the files are not the same, the symbols on GDB will not match the firmware in execution.

5.4 Debug Linux application with gdbserver

5.4.1 Enable debug information

Once your program is built using the sdk toolchain, make sure that the **-g** option is enabled to debug your program and add the necessary debug information.

*Note: If an issue occurs during debugging, you can also force gcc to "not optimize code" using the **-O0** option.*

- Example of a simple test program build:

```
PC $> $CC -g -o myappli myappli.c
```

- Example based on Hello World: refer to "hello world" user space example

Edit and update the makefile for the user space example:

```
...
# Add / change option in CFLAGS if needed
-# CFLAGS += <new option>
+ CFLAGS += -g
...
```

5.4.2 Remote debugging using gdbserver

In this setup, an **ethernet link must be set between the host PC and the target board**.

Once your program is installed on the target (using ssh or copied from an SDcard), you can start debugging it.

- On target side: based on "Hello world" user space example

```
Board $> cd /usr/local/bin
Board $> ls
   hello_world_example
Board $> gdbserver host:1234 hello_world_example
  Process main created; pid = 11832 (this value depends on your target)
  Listening on port 1234
```

- Your target waits for remote PC connection, and then starts debugging.
- Launch the GDB command from your source file folder (easier source loading)

The SDK environment must be installed.

```
PC $> cd <source file folder path>
PC $> ls
    hello_world_example  hello_world_example.c  hello_world_example.o
kernel_install_dir  Makefile
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> $GDB
   GNU gdb (GDB) X.Y.Z
   ...
   This GDB was configured as "--host=x86_64-ostl_sdk-linux --target=arm-ostl-linux-
gnueabi".
   ...
(gdb)
```

- Connect to the target and load the source file:

```
(gdb) target remote <IP_Addr_of_Board>:1234
  Remote debugging using <IP_Addr_of_Board>:1234
  Reading /home/root/test from remote target...
  ...
(gdb) break 16 (line number in the source file)
(gdb) continue
```

- The target program breaks on the breakpoint. Proceed until the end of the program:

```
(gdb) continue
  Continuing.
  [Inferior 1 (process 16204) exited normally]
(gdb) quit
```

5.5 User interface application

5.5.1 Text user interface (TUI) mode

This user interface mode is the first step before using the graphical UI as GDBGUI or DDD.

The TUI can be very useful to map source code with instruction.

Please go through the online documentation [2][3].

5.5.2 Debugging with GDBGUI

Please refer to the dedicated gdbgui article.

5.5.3 Debugging with DDD

GNU DDD is a graphical front-end for command-line debuggers. Please refer to dedicated web page for details[4].

5.5.4 Debugging with IDE

Please refer to STM32CubeIDE.

6 To go further

6.1 Useful GDB commands

When using the GDB in command line mode, it is important to know some basic GDB commands, such as run software, set breakpoints, execute step by step, print variables and display information.

Please refer to GDB commands article.

6.2 Core dump analysis using GDB

The core dump generated for an application crash can be analysed by using the GDB.

Developer Package components, such as SDK and symbol files, must installed. Please refer to STM32MP1 Developer Package .

The symbol file containing the debug symbol of the application in which the crash occurred must also be available.

- First enable the SDK environment:

```
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- Then play the core dump with GDB:

```
PC $> $GDB <path_to_the_binary> <path_to_the_core_dump_file>
```

6.3 Tips

- Managing multiple debug setups

To manage multiple debug setups for different software versions, create different Path_env.gdb files with different names, and call the expected file in the Setup.gdb file:

```
...
#########################################################################

# Set environment configuration
#Path_env.gdb
source Path_env_dk2_18_12_14.gdb

#########################################################################
...
```

7 References

- 1.01.1 https://www.gnu.org/software/gdb
- https://sourceware.org/gdb/onlinedocs/gdb/TUI.html#TUI
- https://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_19.html
- https://www.gnu.org/software/ddd
- Useful external links

Document link	Document Type	Description
Using kgdb, kdb and the kernel debugger internals	User Guide	KGDB documentation guide
Welcome to the GDB Wiki	User guide	GDB Wiki
Building GDB and GDBserver for cross debugging	User Guide	Explain how to build gdb for target and host
A GDB Tutorial with Examples	Training	Debugging a simple application

First Stage Boot Loader