



## **NVIDIA DOCA DPA GDB Server Tool**

# Table of contents

## Introduction

Glossary

Known Limitations

## DPA-specific Notes

Token

Connection on Application Launch

Dummy Thread Concept

Watchdog Issues

## Tool TCP Port and Execution Unit (EU)

## Debugging

Preparation for Debug

Start Debugging

DPA-specific Debugging Techniques

Easy Example of Transitioning from Dummy to Real Thread

Complicated Example of Transitioning from Dummy to Real Thread

Finishing Real Thread without Finishing PUD

## Error Reporting

Tool Log Directory

Verbosity Level of gdbserver

## Useful Info Regarding Work with GDB

Command "directory"

Core Dump Usage

Debug of Optimized Code

---

Disassembly of Advanced RISC-V Commands

---

This document describes the DPA GDB Server tool.

### Info

The DPA GDB Server Tool is currently supported at beta level.

## Introduction

The DPA GDB Server tool (`dpa-gdbserver`) enables debugging FlexIO DEV programs.

DEV programs for debugging are selected using a token (8-byte value) provided by the FlexIO process owner.

### Info

Any GDB, familiar with RISC-V architecture, can be used for the debug. Refer to [this](#) page for information how to work with GDB.

## Glossary

Term	Description
PUD	Process under debug. DEV-side processes intended for debug.
EU	Execution unit (similar to hardware CPU core)
DPA	Data path accelerator
RPC	Remote process communication. Mechanism used in FlexIO to run DEV-side code instantly. Runtime is limited to 6 seconds.
HOST	x86 or aarch64 Linux OS which manages dev-side code (i.e., DEV)
DEV	RISC-V code, loaded by HOST into the DPA's device. Triggered to run by different types of interrupts. DEV side is directly connected to ConnectX adapter card.

Term	Description
GDB	GNU Project debugger. Allows users to monitor another program while it executes.
GDBSERVER	Tool for remote debug programs
RTOS	Real-time operation system running on RISC-V core. Manages handling of interrupts and calls to DEV user processes routines.
RSP	Remote serial protocol. Used for interaction between GDB and GDBSERVER.

## Known Limitations

- DPA GDB technology does not catch fatal errors. Therefore, if a fatal error occurs, core dump (created by `flexio_coredump_create()`) should be used.
- DPA GDB technology does not support Outbox access. GDB users cannot write to Doorbell or to Window configuration areas.
- DPA GDB technology does not support Window access. Read/write to Window memory does not work properly.

## DPA-specific Notes

### Token

The process under debug (PUD) can expose a debugging token. Every external process, using this token, get full access to the process with given token. To not show it constantly (e.g., for security reasons), users can modify their host application temporary. See `flexio_process_udbg_token_get()`.

## Connection on Application Launch

If the code which needs debugging begins to run immediately after launch, the user should modify the host application to stop upon start to give the user time to run

`dpa-gdbserver`. One possible way of doing this is to place function `getchar()` immediately after process creation.

## Dummy Thread Concept

Something to consider with DPA debugging is that a PUD does not have a running thread all time (e.g., the process's thread may exist but be waiting for incoming packets). In a regular Linux application, this scenario is not possible and GDB does not support such cases.

Therefore, when no thread is running, `dpa-gdbserver` reports a dummy thread:

```
(gdb) info thread
  Id  Target Id                      Frame
* 1   Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ??
()
```

In this case user can inspect memory, create breakpoints, and give the `continue` command.

Commands like `step`, `next`, and `stepi` can not be executed for the Dummy thread.

## Watchdog Issues

The RTOS has a watchdog timer that limits DEV code interrupt processes to 120 seconds. This timer is stopped when the user connects to DEV with GDB. Therefore users will have no time limitation for debugging.

## Tool TCP Port and Execution Unit (EU)

By default, `dpa-gdbserver` uses TCP port 1981 and runs on EU 29. If this conflicts with another application (or if other instances of `dpa-gdbserver` are running), users should change the defaults as follows:

```
$> dpa-gdbserver mlx5_0 -T <token> -s <port> -E <eu_id>
```

## Debugging

### Preparation for Debug

Modify your FlexIO application if needed. Make sure the HOST code prints `udbg_token` and waits for GDB connection if needed:

```
+     uint64_t udbg_token;

+     flexio_process_create(..., &flexio_process);

+     udbg_token =
flexio_process_udbg_token_get(flexio_process);
+     if (udbg_token)
+         printf("Process created. Use token >>> %#lx <<< for debug\n",
udbg_token);

+     printf("Stop point for waiting of GDB connection. Press Enter to continue..."); /* Usually
you don't need this stop point */
+     fflush(stdout);
+     getchar();
```

Extract the DPA application from the FlexIO application. For example:

```
$> dpacc-extract cc-host/app/host/flexio_app_name -o
```

```
flexio_app_name.rv5
```

## Start Debugging

1. Run your FlexIO application. It should expose the debug token:

```
$> flexio_app_name mlx5_0  
Process created. Use token >>> 0xd6278388ce4e682c <<< for  
debug
```

2. Run `dpa-gdbserver` with the debug token received:

```
$> dpa-gdbserver mlx5_0 -T 0xd6278388ce4e682c  
Registered on device mlx5_0  
Listening for GDB connection on port 1981
```

3. Run any GDB with RISC-V support. For example, `gdb-multiarch`:

```
$> gdb-multiarch -q flexio_app_name.rv5  
Reading symbols from flexio_app_name.rv5...  
(gdb)
```

4. Connect to the gdbserver using proper TCP port and hostname, if needed:

```
(gdb) target remote :1981  
Remote debugging using :1981  
0x0800000000000000 in ?? ()
```



# DPA-specific Debugging Techniques

## Easy Example of Transitioning from Dummy to Real Thread

Transitioning between the dummy thread and a real thread is not standard practice for debugging under GDB. In an ideal situation, the user would know exactly the entry points for all their routines and can set breakpoints for all of them. Then the user may run the `continue` command:

```
(gdb) target remote :1981
Remote debugging using :1981
0x0800000000000000 in ?? ()
(gdb) info threads
  Id   Target Id               Frame
* 1   Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ??
()
(gdb) b foo
Breakpoint 1 at 0x40000b2: file ../tests/path/hello.c, line 58.
(gdb) b bar
Breakpoint 2 at 0x40000518: file ../tests/path/hallo.c, line 113.
(gdb) continue
Continuing.
```

Initiate interrupts for your DEV program (depends your task), and GDB should catch a breakpoint and now the real thread of the PUD appear instead of the dummy:

```
(gdb) continue
Continuing.
(gdb) [New Thread 1.2]
[New Thread 1.130]
[New Thread 1.258]
[New Thread 1.386]
[Switching to Thread 1.2]
```

```

Thread 2 hit Breakpoint 1, foo(thread_arg=9008)
  at ../tests/path/hello.c:58
58          struct host_data *hdata = NULL;
(gdb) info threads
  Id  Target Id                                Frame
* 2   Thread 1.2 (Process 0 thread 0x1 GVM I 0)  foo (arg=9008)
at ../tests/path/hello.c:58
  3   Thread 1.130 (Process 0 thread 0x81 GVM I 0)  foo (arg=9264) at
../tests/path/hello.c:58
  4   Thread 1.258 (Process 0 thread 0x101 GVM I 0)  foo (arg=9648) at
../tests/path/hello.c:58
  5   Thread 1.386 (Process 0 thread 0x181 GVM I 0)  foo (arg=9904) at
../tests/path/hello.c:58
(gdb)

```

From this point, you may examine memory and trace your code as usual.

## Complicated Example of Transitioning from Dummy to Real Thread

In a more complicated situation, the interrupt happens after GDB connection. In this case, the real thread should start running but cannot because the PUD is in HALT state. The user can type the command `info threads`, see new thread instead of the old dummy, and then switch to the new thread manually:

```

(gdb) target remote :1981
Remote debugging using :1981
0x0800000000000000 in ?? ()
(gdb) info threads
  Id  Target Id                                Frame
* 1   Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ??
()
(gdb) info threads
[New Thread 1.32769]

```

Id	Target Id	Frame
2	Thread 1.32769 (Process 0 thread 0x8000 GVMI 0)	bar (arg=0xc0, len=0) at /path/lib/src/stub.c:167

The current thread <Thread ID 1> has terminated. See `help thread`.

```
(gdb) thread 2
```

```
[Switching to thread 2 (Thread 1.32769)]
```

```
#0 bar (arg=0xc0, len=0)
    at /path/lib/src/stub.c:167
```

```
167 {
```

```
(gdb) bt
```

```
#0 bar (arg=0xc0, len=0)
    at /path/lib/src/stub.c:167
```

```
#1 0x000000004000017a in foo (thread_arg=3221)
    at ../path/dev/hello.c:182
```

```
#2 0x0000000000000000 in ?? ()
```

```
Backtrace stopped: frame did not save the PC
(gdb)
```

### Note

The same command `info threads` in lines 4 and 7 gives different results. This happens because the interrupt occurs between the instances and the real code begins to run.

The user must switch to the new thread manually (see line 14). After this, they can trace/debug the flow as usual (i.e., using the commands `step`, `next`, `stepi`).

## Finishing Real Thread without Finishing PUD

Every interrupt handler at some point finishes its way and returns the CPU resources to RTOS. The most common way to do this is to call function `flexio_dev_thread_reschedule()`. The command `next` on this function will have the same effect as the command `continue`:

```
205         __dpa_thread_fence(__DPA_MEMORY, __DPA_W,  
__DPA_W);  
(gdb) next  
206         flexio_dev_cq_arm(dtctx,  
app_ctx.rq_cq_ctx.cq_idx, app_ctx.rq_cq_ctx.cq_number);  
(gdb) next  
208         if ((dev_errno =  
flexio_dev_get_and_rst_errno(dtctx)) {  
(gdb) next  
213         print_sim_str("Nothing to do. Wait for next duar\n", 0);  
(gdb) next  
214         flexio_dev_thread_reschedule();  
(gdb) next
```

### Info

GDB waits until the user types `^C` or a breakpoint is reached after the next interrupt occurred.

## Error Reporting

### Info

The DPA GDB server tool has been validated with `gdb-multiarch` (version 9.2) and with GDB version 12.1 from RISC-V tool chain.

### **Note**

The GDB server should support all commands described in GDB RSP (remote serial protocol) for GDB stubs. But only the most common GDB commands are supported.

Should a `dpa-gdbserver` bug occur, please provide the following data:

- Used GDB (name and version)
- Commands sequence to reproduce the issue
- DPA GDB server tool console output
- DPA GDB server tool log directory content (see next part for details)
- Optional – output data printed when `dpa-gdbserver` is run in verbose mode

## Tool Log Directory

For every run, a temporary directory is created with the template `/tmp/flexio_gdbs.XXXXXX`.

To locate the latest one, run the following command:

```
$> ls -ldtr /tmp/flexio_gdbs.* | tail
```

## Verbosity Level of `gdbserver`

By default, `dpa-gdbserver` does not print any log information to screen. Adding `-v` option to command line increases verbosity level, printing additional info to `dpa-gdbserver` terminal display. Verbosity level is incremented according to number of 'v' in command line switch (i.e. `-vv`, `-vvv` etc.).

One `-v` shows the RSP exchange. This is a textual protocol, so users can read and understand requests from GDB and answers from the GDB server:

```
<<<<< "qTStatus"
>>>>> ""
<<<<< "?"
>>>>> "S05"
<<<<< "qfThreadInfo"
>>>>> "mp01.30011981"
<<<<< "qsThreadInfo"
>>>>> "I"
<<<<< "qAttached:1"
>>>>> "1"
<<<<< "Hc-1"
>>>>> "OK"
<<<<< "qC"
>>>>> "QCp01.30011981"
```

### Info

In the examples, `<<<<<` and `>>>>>` are used to indicate data received from GDB and transmitted to GDB, respectively.

When running with a higher verbosity level (e.g., run `dpa-gdbserver` with option `-vv` or higher), the exchange with the RTOS module is shown:

```

<<<<< "qfThreadInfo"
/ 2/dgdbserver - cmd 0x5
/ 2/dgdbserver - retval 0x4
>>>>> "mp01.30011981"
<<<<< "qsThreadInfo"
/ 2/dgdbserver - cmd 0x5
/ 2/dgdbserver - retval 0x5
>>>>> "|"
<<<<< "m8000000000000000,4"
/ 2/dgdbserver - cmd 0xc
/ 2/dgdbserver - retval 0x9
>>>>> "E0a"
<<<<< "m7ffffffffffffc,4"
/ 2/dgdbserver - cmd 0xc
/ 2/dgdbserver - retval 0x9
>>>>> "E0a"
<<<<< "qSymbol:~"
>>>>> "OK"

```

### Info

Lines beginning with `/ #/` provide the number of internal RTOS threads printed from the DEV side.

## Useful Info Regarding Work with GDB

This section provides useful information about commands and methods which can help users when performing DPA debug. This is not related to the `dpa-gdbserver` itself. But this is about remote debugging and FlexIO sources.

## Command "directory"

GDB can run on a different host from the one where compilation was done. For example, users may have compiled and run their application on `host1` and run their instance of GDB on `host2`. In this case, users will see the error message

```
../xxx/yyy/zzz/your_file.c: No such file or directory
```

To solve this problem, copy sources to the host running GDB (`host2` in the example). Make sure to save the original code hierarchy. Use GDB command `directory` to inform where the sources are to GDB:

```
host2~$> gdb-multiarch -q /tmp/my_riscv.elf
Reading symbols from /tmp/my_riscv.elf...
(gdb) b foo
Breakpoint 1 at 0x4000016c: file ../xxx/yyy/zzz/my_file.c, line 182.
(gdb) target remote host1:1981
Remote debugging using host1:1981
0x0800000000000000 in ?? ()
(gdb) c
Continuing.
[New Thread 1.32769]
[Switching to Thread 1.32769]

Thread 2 hit Breakpoint 1, foo (thread_arg=5728) at
../xxx/yyy/zzz/my_file.c:182
182     ../xxx/yyy/zzz/my_file.c: No such file or directory.
(gdb) directory /tmp/apps/
Source directories searched: /tmp/apps:$cdir:$cwd
(gdb) list
179         struct flexio_dev_thread_ctx *dtctx;
180         uint64_t dev_errno;
181
182         print_sim_str("=====> NET event handler started\n", 0);
183
184         flexio_dev_print("Hello GDB user\n");
185
```



## **i** Note

Pay attention to the exact path reported by GDB. The argument for the command `directory` should point to the start point for this path. For example, if GDB looks for `../xxx/yyy/zzz` and you placed the sources in local directory `/tmp/copy_of_worktree`, then the command should be

```
(gdb) directory /tmp/copy_of_worktree/xxx/ and not
(gdb) directory /tmp/copy_of_worktree/.
```

Sometimes, the `*.elf` file provides a global path from the root. In this case, use the command `set substitute-path <from> <to>`. For example, if the file `/foo/bar/baz.c` was moved to `/mnt/cross/baz.c`, then the command `(gdb) set substitute-path /foo/bar /mnt/cross` instructs GDB to replace `/foo/bar` with `/mnt/cross`, which allows GDB to find the file `baz.c` even though it was moved.

See [this page](#) of GDB documentation for more examples of specifying source directories.

## Core Dump Usage

If the code runs into a fatal error even though the host side of your project is implemented correctly, a core dump is saved which allows analyzing the core. It should point exactly to where the fatal error occurred. The command `backtrace` can be used to examine the memory and its registers. Change the frame to see local variables of every function on the backtrace list:

```
$> gdb-multiarch -q -c crash_demo.558184.core /tmp/my_riscv.elf
Reading symbols from /tmp/my_riscv.elf...
```

```
[New LWP 1]
```

```

#0  0x000000004000126e in read_test (line=153, ptr=0x30) at
/xxx/yyy/zzz/my_file.c:109
109          val = *(volatile uint64_t *)ptr;
(gdb) bt
#0  0x000000004000126e in read_test (line=153, ptr=0x30) at
/xxx/yyy/zzz/my_file.c:109
#1  0x000000004000031a in tlb_miss_test (op_code=1) at
/xxx/yyy/zzz/my_file.c:153
#2  0x0000000040000144 in test_thread_err_events_entry_point
(h2d_daddr=3221258560) at /xxx/yyy/zzz/my_file.c:588
#3  0x00000000400013fc in
_dpacc_flexio_dev_arg_unpack_test_err_events_dev_test_thread_err_e
(argbuf=0xc0008228, func=0x400000b0
<test_thread_err_events_entry_point>)
    at /tmp/dpacc_xExkvE/test_err_events_dev.dpa.device.c:67
#4  0x0000000040001680 in flexio_hw_rpc (host_arg=3221258752) at
/local_home/www/flexio-sdk/libflexio-
dev/src/flexio_dev_entry_point.c:75
#5  0x0000000000000000 in ?? ()
Backtrace stopped: frame did not save the PC
(gdb) frame 4
#4  0x0000000040001680 in flexio_hw_rpc (host_arg=3221258752) at
/local_home/igorle/flexio-sdk/libflexio-
dev/src/flexio_dev_entry_point.c:75
75          retval = unpack_cb(&data_from_host-
>func_params.arg_buf,
(gdb) p /x *data_from_host
$2 = {poll_lkey = 0x1ff2b1, window_id = 0x3, poll_haddr =
0x55dc0f40b900, entry_point = 0x400013d8, func_params = {func_wo_pack =
0x0, dev_func_entry = 0x400000b0, arg_buf = 0xc0008140}}
(gdb)

```

## Debug of Optimized Code

Usually highly optimized code is compiled and run.

Two types of mistakes in code can be considered:

- Logical errors
- Optimization-related errors

Logical errors (e.g., using `&` instead of `&&`) are reproduced on the non-optimized version of the code. Optimization related errors (e.g., forgetting volatile classification, non-usage of memory barriers) only impact optimization. Non-optimized code is much easier for tracing with GDB, because every C instruction is translated directly to assembly code.

It is good practice to check if an issue can be reproduced on non-optimized code. That helps observing the application flow:

```
$> build.sh -O 0
```

For tracing this code, using GDB commands `next` and `step` should be sufficient.

But if an issue can only be reproduced on on optimized code, you should start debugging it. This would require reading disassembly code and using the GDB command `stepi` because it becomes a challenge to understand exactly which C-code line executed.

## Disassembly of Advanced RISC-V Commands

DPA core runs on a RISC-V CPU with an extended instruction set. The GDB may not be familiar with some of those instructions. Therefore, `asm` view mode shows numbers instead of disassembly. In this case it is recommended to disassemble your RISC-V binary code manually. Use the `dpa-objdump` utility with the additional option `--mcpu=nv-dpa-bf3`.

```
$> dpa-objdump -sSdx1 --mcpu=nv-dpa-bf3 my_riscv.elf >  
my_riscv.asm
```

The following screenshot shows the difference:

4000057a: 03 35 84 fe	ld a0, -24(s0)	4000057a: 03 35 84 fe	ld a0, -24(s0)
4000057e: 08 65	ld a0, 8(a0)	4000057e: 08 65	ld a0, 8(a0)
40000580: 13 <b>5856b</b>		40000580: 13 <b>55 85 6b</b>	<b>rev8 a0, a0</b>
40000584: e2 60	ld ra, 24(sp)	40000584: e2 60	ld ra, 24(sp)
40000586: 42 64	ld s0, 16(sp)	40000586: 42 64	ld s0, 16(sp)
40000588: 05 61	addi sp, sp, 32	40000588: 05 61	addi sp, sp, 32
4000058a: 82 80	ret	4000058a: 82 80	ret

## Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF

ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

## **Trademarks**

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

© Copyright 2024, NVIDIA. PDF Generated on 12/19/2024