NXP Semiconductors
Application Note

Document Number: AN5269
Rev. 06/2018

ODP Reflector Application Debug

Contents

1 Introduction

The most popular user space application for SDK/ODP (OpenDataPlane) is the packet reflector reference application.

This application note is focused on the `odp_reflector` application. It's the extension of `odp_pktio` application with the addition of the schedule PUSH mode, where scheduled packets are received in the PUSH mode. Like `odp_pktio`, the `odp_reflector` application swaps the Ethernet and IP header addresses of the received frames and transmits them on the same interface.

This application note explains a use case, where `odp_reflector` is running on a single board using 2 ports connected back-to-back. If you need another hardware setup or full details about the ODP applications, see *LS2085 SDK Quick Start Guide*.

This application note explains:

- How you can build a real hardware setup for running `odp_reflector`
- How you can import, download, run, and debug the `odp_reflector` application from CodeWarrior
- How you can attach to a running `odp_reflector` application and debug it using CodeWarrior

2 Get the ODP reflector source files

To get the latest AIOP APP source files, follow the steps from SDK documentation or from Layerscape-SDK documentation.

3 Hardware setup

The reflector application reflects back the packet received on the same interface where the packets are originally received, and the source and destination MAC and IP addresses of the received packet are swapped.

3.1 Hardware setup using only one board

In order to demonstrate the traffic “reflected”, you can use a single board with 2 ports connected back-to-back. In the following figure, the copper red ports 5 and 6 are connected.

- Port 5 - Linux container
- Port 6 - ODP container

Figure 1. Single board with 2 ports connected back-to-back

```
LINUX                                                          ODP
dpni.0  <-> dpmac.5 <------------------------------> dpmac.6 <-> dpni.1
(ni0)
```

Run these commands when the U-Boot prompt is available.

At the U-Boot console, bring up the board via TFTP. Or, you can write the images to the flash memory using the CodeWarrior for ARMv8 flash programmer tool.

```
setenv filesize; setenv myaddr 0x580100000; tftp 0x80000000 u-boot-nor.bin; protect off
$myaddr +$filesize; erase $myaddr +$filesize; cp.b 0x80000000 $myaddr $filesize; protect on
$myaddr +$filesize

setenv filesize; setenv myaddr 0x580000000; tftp 0x80000000 PBL.bin; protect off $myaddr +
$filesize; erase $myaddr +$filesize; cp.b 0x80000000 $myaddr $filesize; protect on $myaddr +
$filesize

setenv filesize; setenv myaddr 0x580300000; tftp 0x80000000 mc.itb; protect off $myaddr +
$filesize; erase $myaddr +$filesize; cp.b 0x80000000 $myaddr $filesize; protect on $myaddr +
$filesize

setenv filesize; setenv myaddr 0x580700000; tftp 0x80000000 dpl-eth.0x2A_0x41.dtb; protect
off $myaddr +$filesize; erase $myaddr +$filesize; cp.b 0x80000000 $myaddr $filesize; protect
on $myaddr +$filesize

setenv filesize; setenv myaddr 0x580800000; tftp 0x80000000 dpc-0x2a41.dtb; protect off
```

```
$myaddr +$filesize; erase $myaddr +$filesize; cp.b 0x80000000 $myaddr $filesize; protect on
$myaddr +$filesize

<restart the board for loading new images>
```

Prepare target for running ODP application

```
fsl_mc start mc 580300000 580800000 && fsl_mc apply dpl 580700000
tftp a0000000 kernel-ls2085ardb.itb
bootm a0000000
```

NOTE

bootargs needs to contains minimal parameters in order to have a correct setup for AIOP application. Make sure `bootargs=console=ttyS1,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0600 ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256`

Configure the ni0 interface and create static ARP entry. Set the destination MAC as the ARP hardware address for all the IP flows on which the packet needs to be sent. `000000000006` is the MAC of the dpmac.6 (this information can be verified using restool).

```
root@ls2085ardb:~# ifconfig ni0 6.6.6.1 up
root@ls2085ardb:~# arp -s 6.6.6.10 000000000006
```

Configure the eth0 interface.

```
root@ls2085ardb:~# ifconfig eth0 192.168.1.2
```

Connect a new DPNI for DPMAC.6 and create the rest of the objects for the ODP container using the `dynamic_dpl.sh` script. This script uses a Linux user space tool called **restool** that can be used to create/destroy containers and objects dynamically.

```
root@ls2085ardb:~# cd /usr/odp/scripts
root@ls2085ardb:/usr/odp/scripts# ./dynamic_dpl.sh dpmac.6
Available DPRCs
dprc.1

dprc.2 Created

Validating the arguments.....

DPNI parameters :-->
        MAX_SENDERS = 8
        MAX_TCS = 1
        MAX_DIST_PER_TC = 8
        MAX_DIST_KEY_SIZE = 32
        DPNI_OPTIONS =
DPNI_OPT_MULTICAST_FILTER,DPNI_OPT_UNICAST_FILTER,DPNI_OPT_DIST_HASH,DPNI_OPT_DIST_FS,DPNI_OP
T_FS_MASK_SUPPORT


DPCON parameters :-->
        DPCON_PRIORITIES = 8


DPSECI parameters :-->
        DPSECI_QUEUES = 8
        DPSECI_PRIORITIES = 2,2,2,2,2,2,2,2


DPIO parameters :-->
        DPIO_PRIORITIES = 8


####### Parsing argument number 1 (dpmac.6) #######

        dpni.1 created with MAC addr = 00:00:00:00:0:6
```

```
        Disconnecting the dpmac.6, if already connected
        dpmac.6 Linked with dpni.1
        dpni.1 assigned to  dprc.2

 dpmac.6 <--------connected------> dpni.1 (00:00:00:00:0:6)
................
USE  dprc.2  FOR YOUR APPLICATIONS
```

Verify if the DPNI interfaces are correctly allocated using restool.

```
root@ls2085ardb:/usr/odp/scripts# restool dpni info dpni.0
dpni version: 6.0
dpni id: 0
plugged state: plugged
endpoint state: 1
endpoint: dpmac.5, link is up
link status: 1 - up
mac address: 4a:3e:27:d2:df:f6
dpni_attr.options value is: 0x180
        DPNI_OPT_UNICAST_FILTER
        DPNI_OPT_MULTICAST_FILTER
max senders: 1
max traffic classes: 1
max distribution's size per RX traffic class:
        class 0's size: 1
max unicast filters: 16
max multicast filters: 64
max vlan filters: 0
max QoS entries: 0
max QoS key size: 0
max distribution key size: 0
root@ls2085ardb:/usr/odp/scripts# restool dpni info dpni.1
dpni version: 6.0
dpni id: 1
plugged state: plugged
endpoint state: 0
endpoint: dpmac.6, link is down
link status: 0 - down
mac address: 00:00:00:00:00:06
dpni_attr.options value is: 0x401b0
        DPNI_OPT_DIST_HASH
        DPNI_OPT_DIST_FS
        DPNI_OPT_UNICAST_FILTER
        DPNI_OPT_MULTICAST_FILTER
        DPNI_OPT_FS_MASK_SUPPORT
max senders: 8
max traffic classes: 1
max distribution's size per RX traffic class:
        class 0's size: 8
max unicast filters: 16
max multicast filters: 64
max vlan filters: 0
max QoS entries: 0
max QoS key size: 0
max distribution key size: 32
```

Start the odp_reflector and check if this works using ping.

```
root@ls2085ardb:~# cd /usr/odp/bin/
root@ls2085ardb:/usr/odp/bin# export DPRC=dprc.2
root@ls2085ardb:/usr/odp/bin# ./odp_reflector -i dpni.1 -m 0 -c 8 &
```

```
<enter>
root@ls2085ardb:~# ping 6.6.6.10 -c 2
PING 6.6.6.10 (6.6.6.10) 56(84) bytes of data.
64 bytes from 6.6.6.10: icmp_seq=1 ttl=64 time=0.093 ms
64 bytes from 6.6.6.10: icmp_seq=2 ttl=64 time=0.055 ms

--- 6.6.6.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.055/0.074/0.093/0.019 ms
```

4 CodeWarrior Setup

This topic explains:

- Import and start the odp_reflector application from CodeWarrior
- Attach to a running odp_reflector application and debug using CodeWarrior
- Debug capabilities

4.1 Import and start the odp_reflector application from CodeWarrior

After compiling the `odp_reflector` application, with debug information (-ggdb) you'll need to use in CodeWarrior the elf file containing the debug symbols. This will generate the correct elf/DWARF symbolic needed for the CodeWarrior parser to make data to symbols and symbols to data (this is very important).

1. Select **File > Import > C/C++ > CodeWarrior Debug Projects**.
2. Click **Next**.
3. Select the odp_reflector elf available at:

   ```
   <Yocto>/build_ls2085ardb_release/tmp/work/aarch64-fsl-linux/odp/1.4-r0/git/example/
   reflector/odp_reflector
   ```

   CodeWarrior automatically detects the elf type and makes the settings for the Linux Application debug flow.

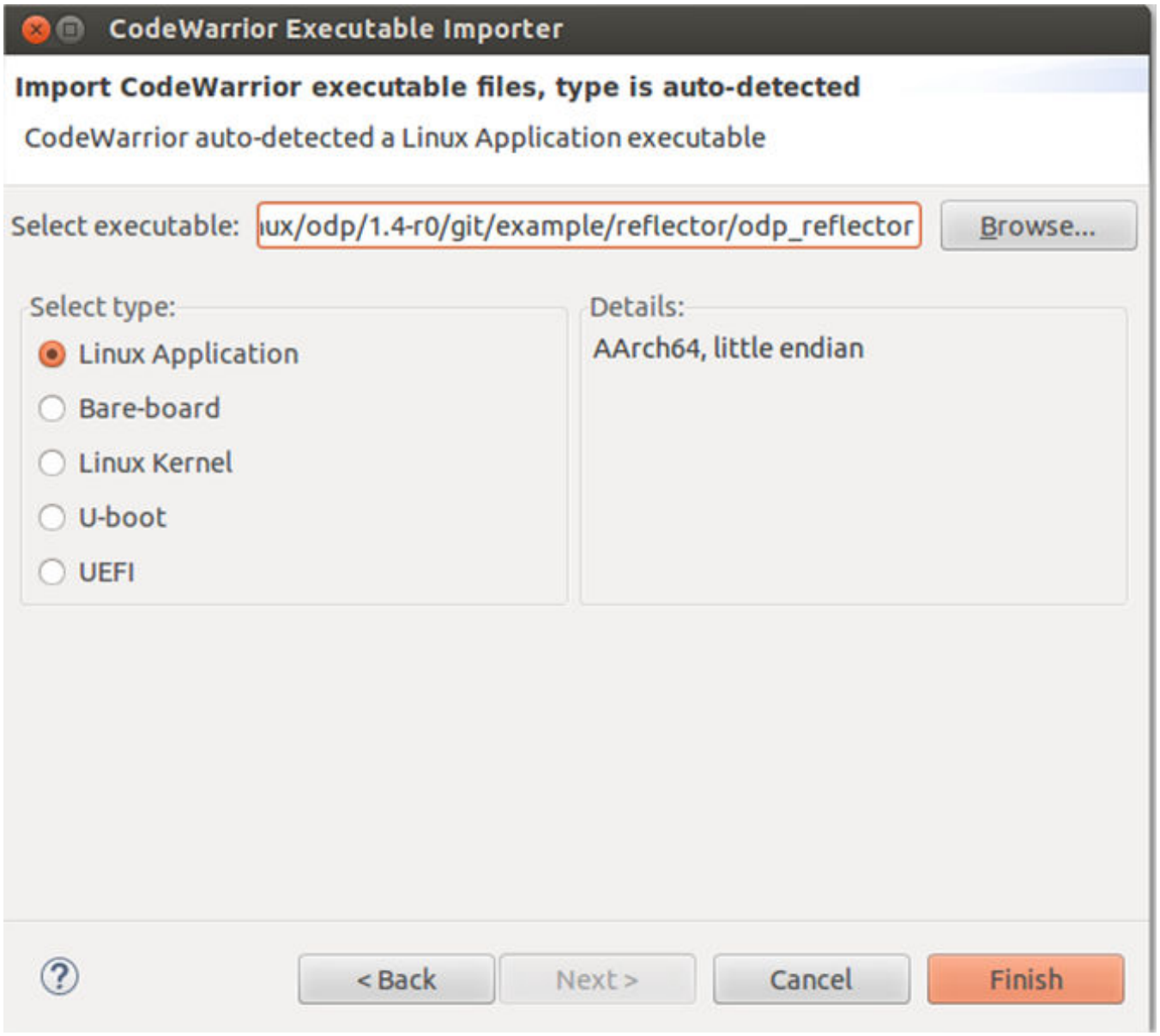


Figure 2. CodeWarrior Executable Importer

4. Set the remote absolute path to `odp_reflector` and the commands to execute before the application.

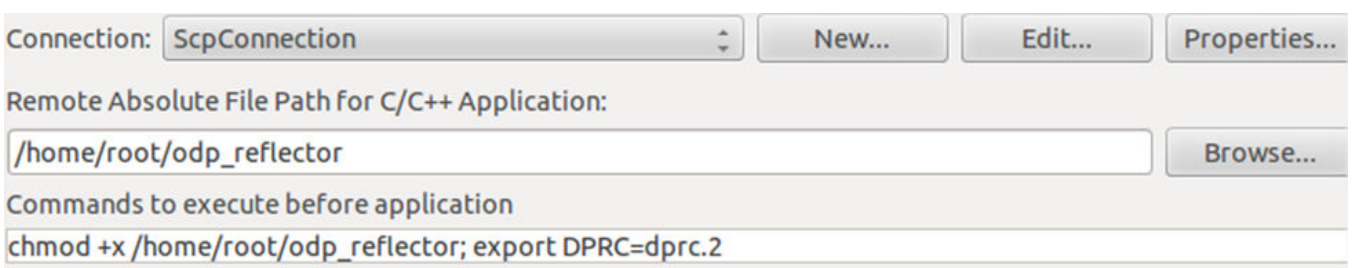


Figure 3. Absolute path and commands to execute

5. If you want, you can set up directly the remote path of the odp_reflector without downloading it (use **Skip download to target path**) In this case, **check** to have the reflector from target obtained from the same yocto/rootfs build with the one imported in CodeWarrior .

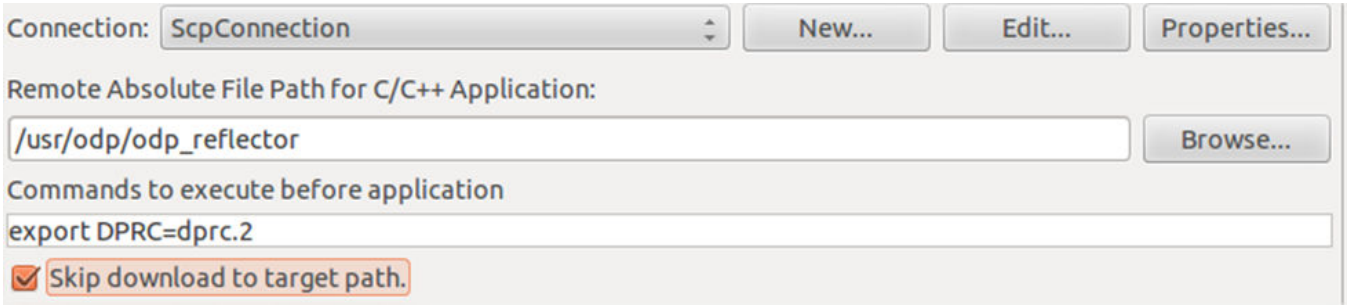


Figure 4. Skip download to target path

6. Set the host/IP name of the ODP Linux target using the **Edit** button in the **Connection** area.

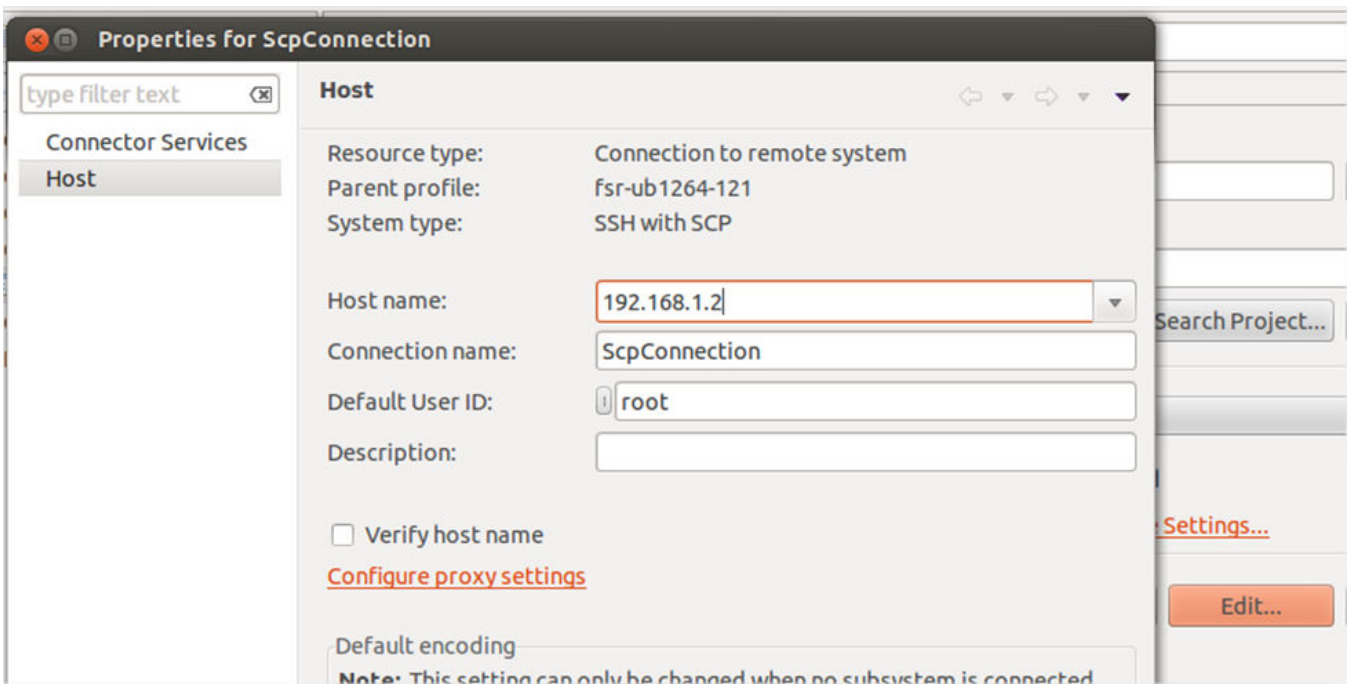


Figure 5. Scp Connection properties

7. Set sysroot in `.gdbinit` from the **Debugger > Main** tab appending this command: `set sysroot <path_to_sdk>/build_ls2085ardb_release/tmp/sysroots/ls2085ardb`

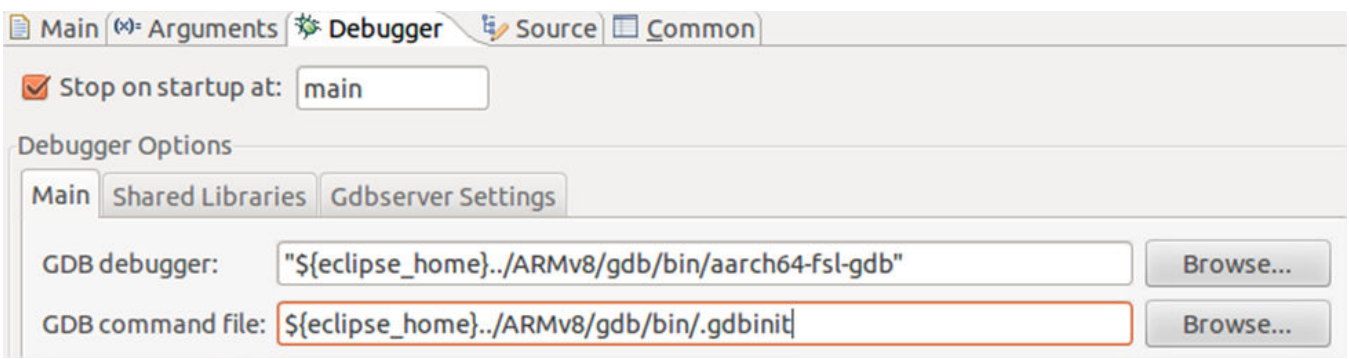


Figure 6. Set sysroot

8. Set the gdbserver port used by ssh tunnel by selecting the **Debugger > Gdbserver Settings** tab.

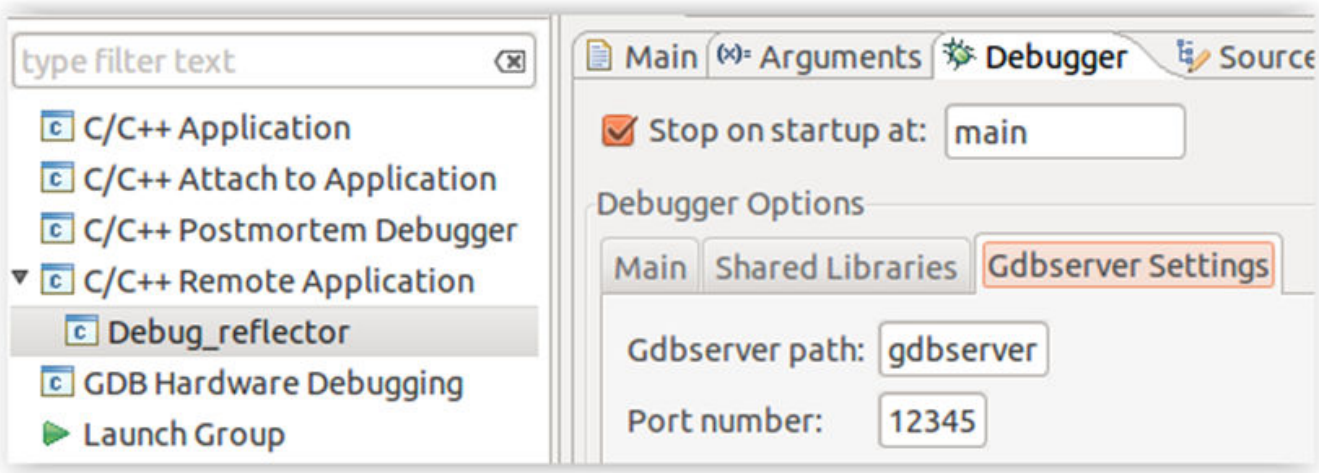


Figure 7. Gdbserver port

9. The `odp_reflector` also needs some specific arguments in order to start correctly. These can be set up using the **Arguments** tab in the **Debug Configurations** dialog. You can refer the *LS2085 SDK_Quick_Start_Guide* for details about the legal arguments for the reflector application.
10. Select **Run > Debug Configurations > C/C++ Remote Application**.
11. Click the **Debug** button to run the reflector application.

    A login window appears.

12. Enter the user ID for linux target as root and the password is blank. Click **OK**.

    Please note that now `odp_reflector` can only run as root and also the CodeWarrior will warn you, if necessary, about the changed RSA key mapping for root and the remote target Linux. For example, this can happen if you are restarting the Linux target and the remote Linux generates a different RSA key.

13. The connection between gdb and gdbserver is established, the gdbserver will start the odp_reflector and the bind script will be run. You can observe this in the remote shell console.

Figure 8. Remote shell console

14. The reflector debug sessions starts and all the debug capabilities are available. By default, the reflector will be stopped at main() function as per the **Debugger** tab settings.

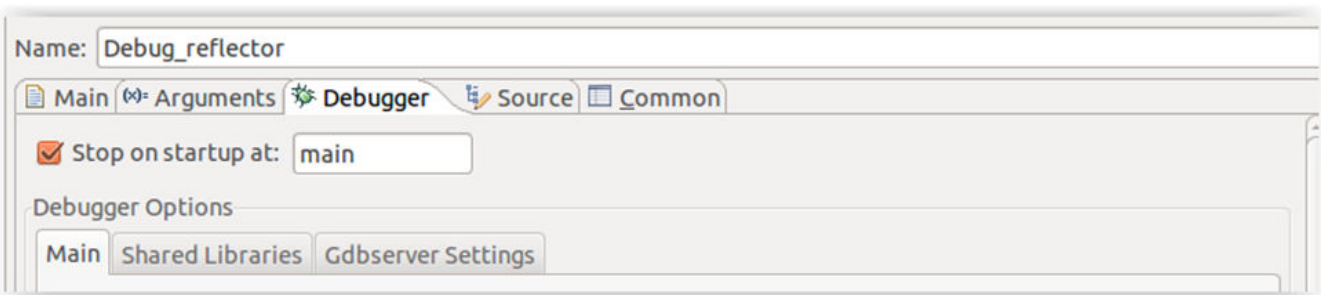


Figure 9. Debugger tab

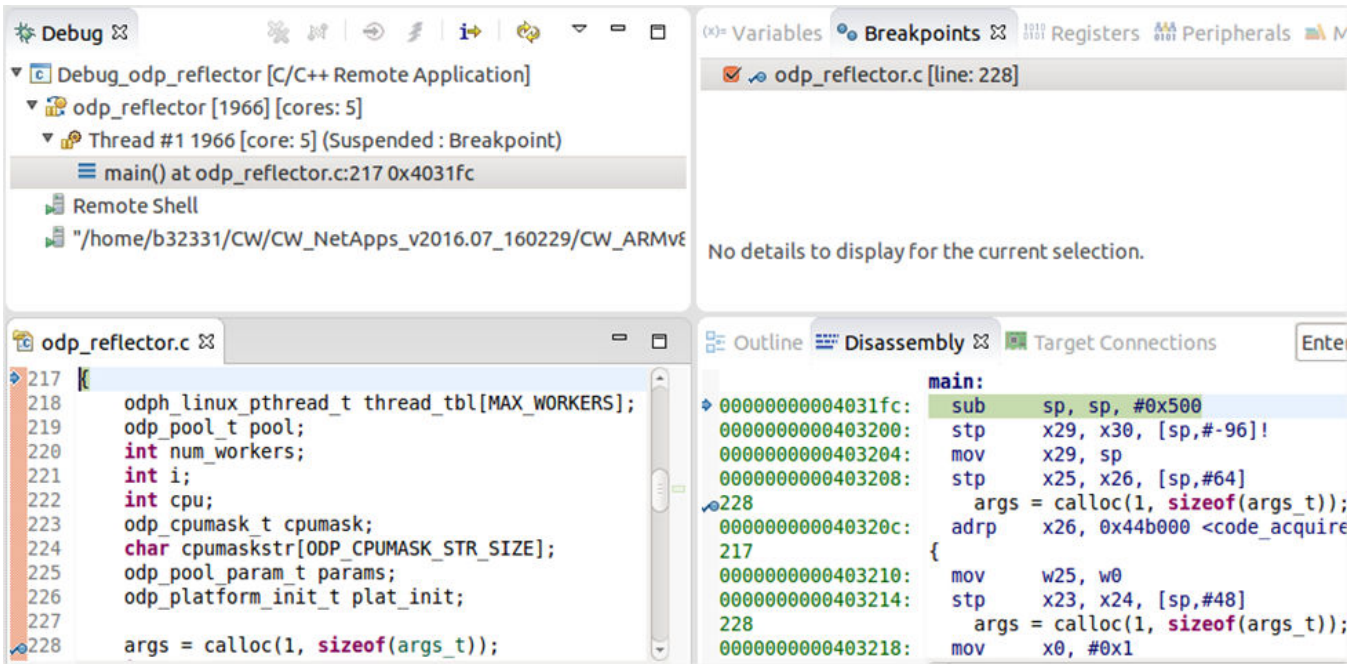


Figure 10. Debugging starts

15. Following are the key functions to debug:
    a. `pktio_thread`
    b. `odp_schedule`
    c. `odp_packet_from_event`
    d. `swap_pkt_addrs`
    e. `odp_pktio_send`

    Set up breakpoints at all these functions from the gdb command line to see how the new threads are spawned and executed.

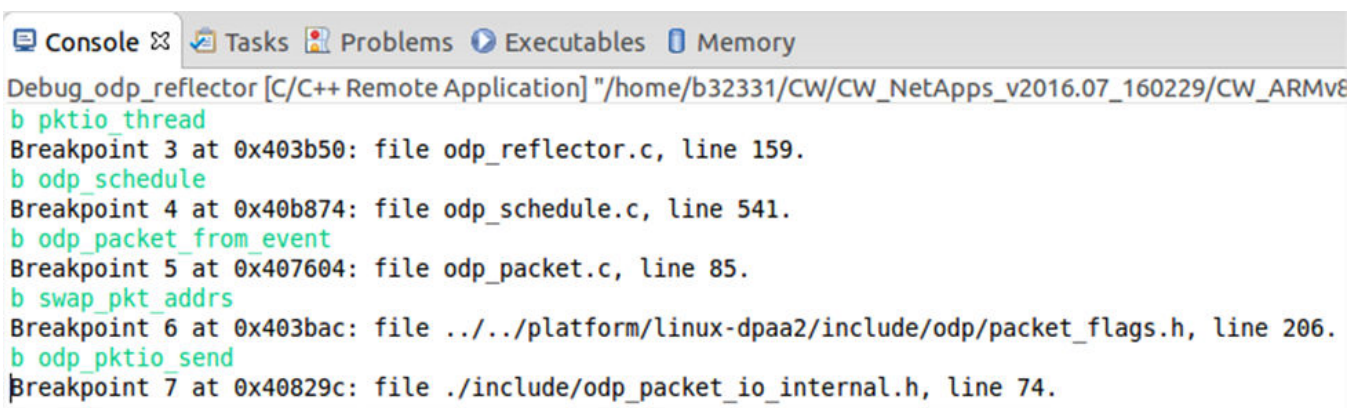


Figure 11. Set breakpoints at different functions

16. Disable all these breakpoints from GUI, resume the application.
17. At the Linux container, issue a ping (that is `ping 6.6.6.10 -c 10`) and during this run enable all breakpoints again. The breakpoints will be hit in different conditions.

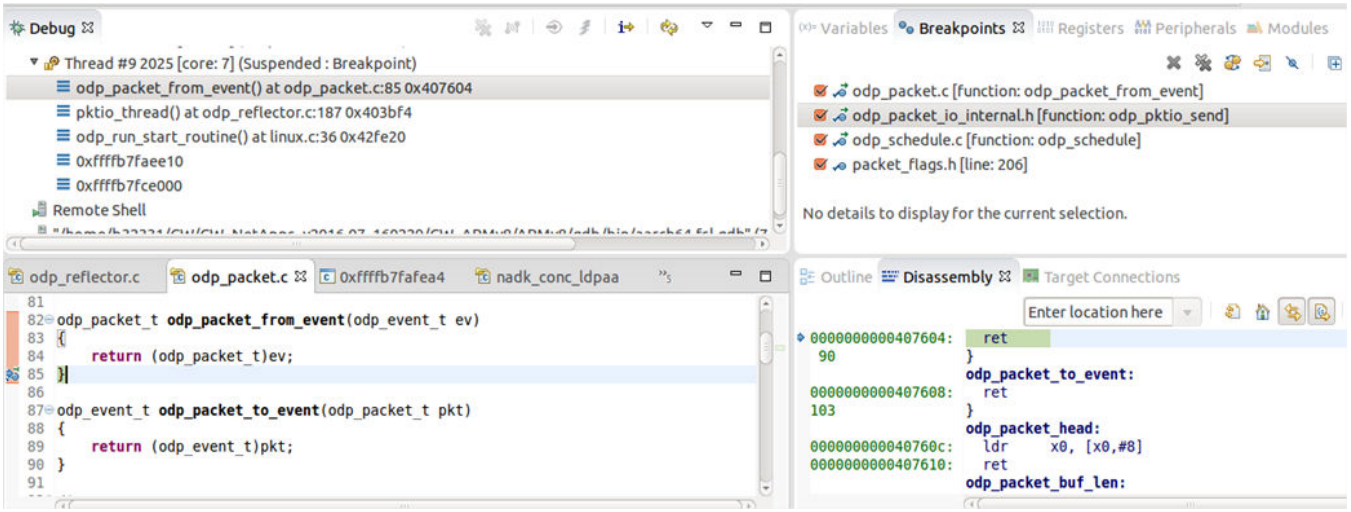


Figure 12. Re-enable breakpoints

4.2 Attach to a running odp_reflector application and debug using CodeWarrior

1. Import the `odp_reflector` elf file as explained in Import and start the odp_reflector application from CodeWarrior.
2. In the **Debug Configurations** dialog, select **C/C++ attach to Application** and click **New launch configuration**.
3. Click the **Debugger** tab and select **gdbserver** from the **Debugger** drop-down list.
4. Click the **Connection** subtab, set **Type** as **TCP**. Also, set the host IP of the Linux target and the port number for the gdbserver.

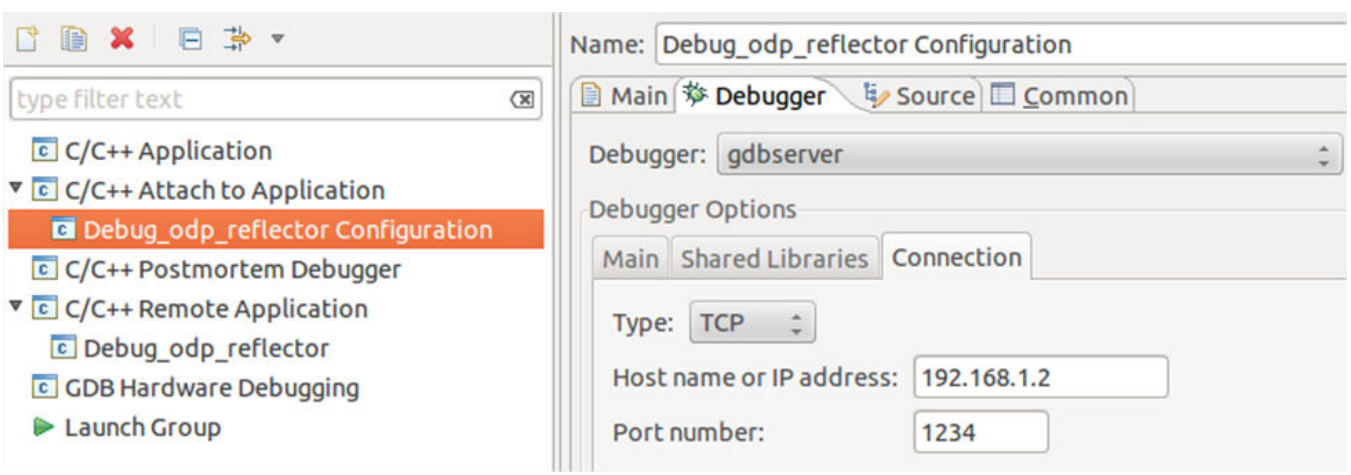


Figure 13. Set connection type

5. Start the gdbserver and the odp application standalone on the Linux target.

```
export DPRC=dprc.2
/usr/odp/bin/odp_reflector -i dpni-1 -m 0 -c 8 &
gdbserver --multi :1234
```

6. Click **Debug**. The gdb client will connect to gdbserver. Now, you can attach to any application from the Linux target, including the reflector, using the green button shown below.

Figure 14. Connect to a process

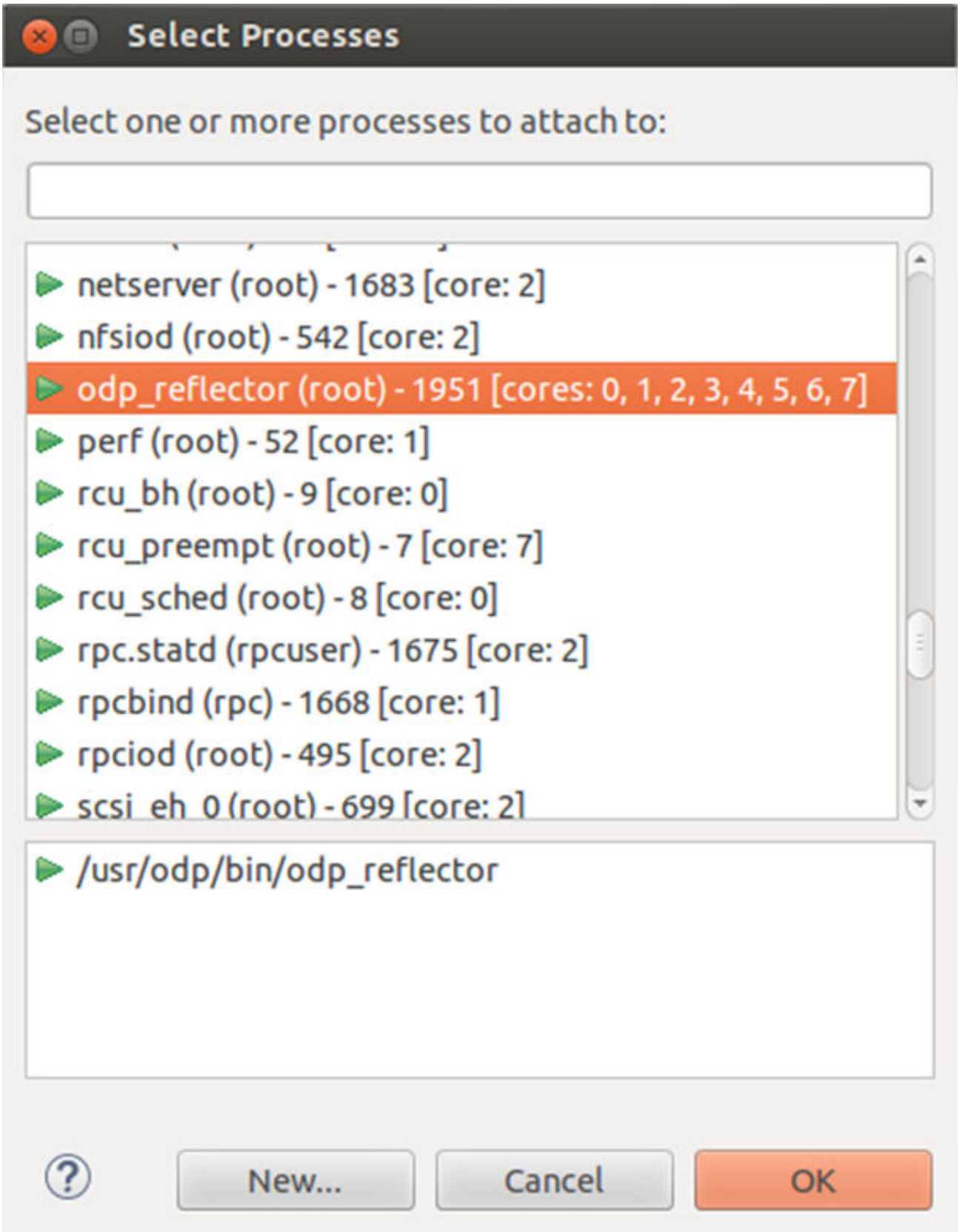


Figure 15. Select process

7. Click **OK** to attach to the running reflector. This enables all the debug capabilities. The stack after attach is shown below.

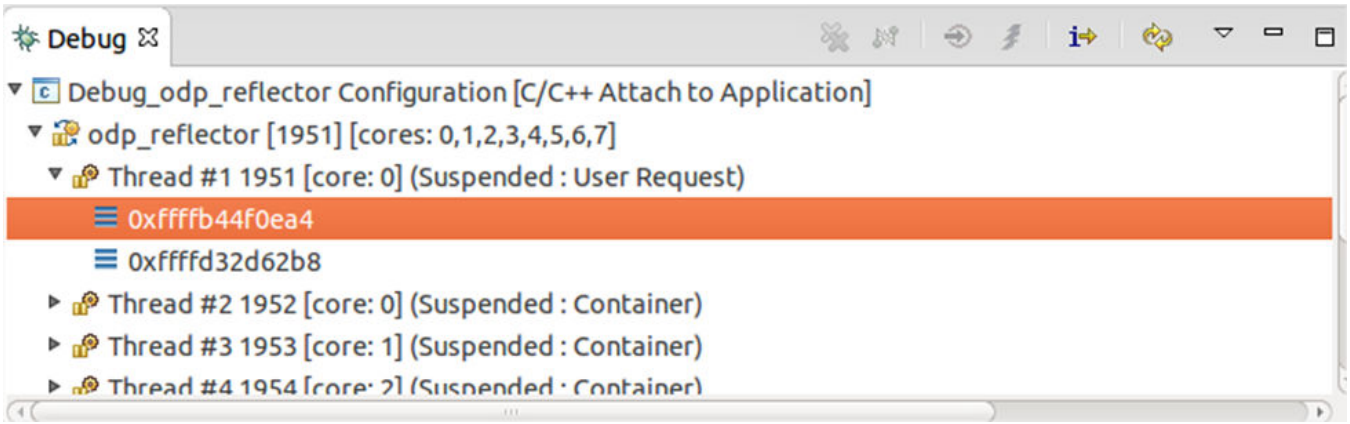


Figure 16. Debug session

8. For setting breakpoints to the global symbols, such as `main`, load the `odp_reflector` debug symbols using the *file <path_to_elf>* command.

```
file /home/b32331/LS2/sdk/EAR6.0/prerelease_iso/LS2085A-SDK-20160304-yocto/
build_ls2085ardb_release/tmp/work/aarch64-fsl-linux/odp/1.4-r0/git/example/reflector/
odp_reflector
```

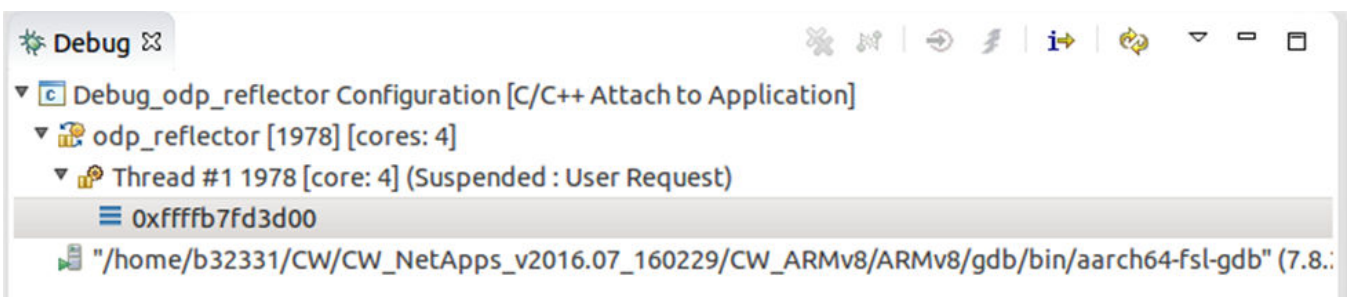
4.2.1 odp_reflector debug from entry_point

To debug odp_reflector from entry_point:

1. You can manually start the reflector via gdbserver on the Linux target as below and then just attach with CodeWarrior to the gdbserver/odp_reflector. Note that you must run the bind script manually. This is basically the same solution presented in Import and start the odp_reflector application from CodeWarrior where all steps are made by CodeWarrior.

```
export DPRC=dprc.2
gdbserver --multi :1234 /usr/odp/bin/odp_reflector -i dpni-1 -m 0 -c 8
```

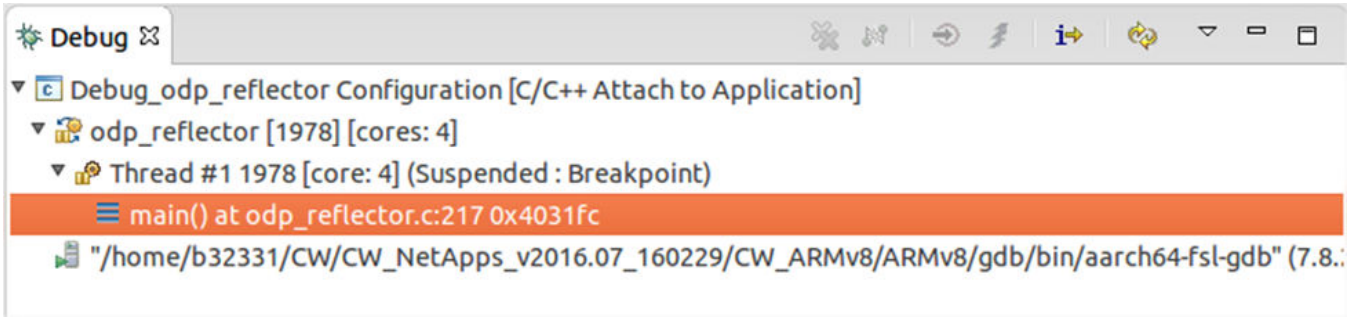
2. Click **Debug**. CodeWarrior will attach to reflector and stop at the entry_point.



3. For setting breakpoints to global symbols, such as `main`, you should load the odp_reflector debug symbols using *file <path_to_elf>* command.

```
file /home/b32331/LS2/sdk/EAR6.0/prerelease_iso/LS2085A-SDK-20160304-yocto/
build_ls2085ardb_release/tmp/work/aarch64-fsl-linux/odp/1.4-r0/git/example/reflector/
odp_reflector
```

4. Now, you can set breakpoints from the gdb console, such as `b main`. After the hitting the breakpoint from the `main` function the stack will look as below.



4.3 Debug capabilities

In the debug session, various debug capabilities are available:

1. **GDB console** (selected from right side). In this console, you can run gdb commands.

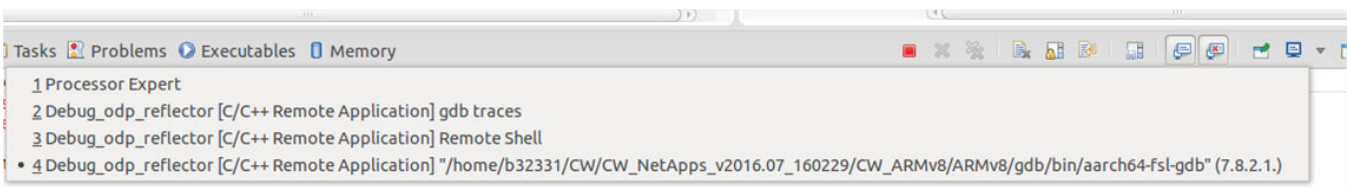


Figure 17. Select console

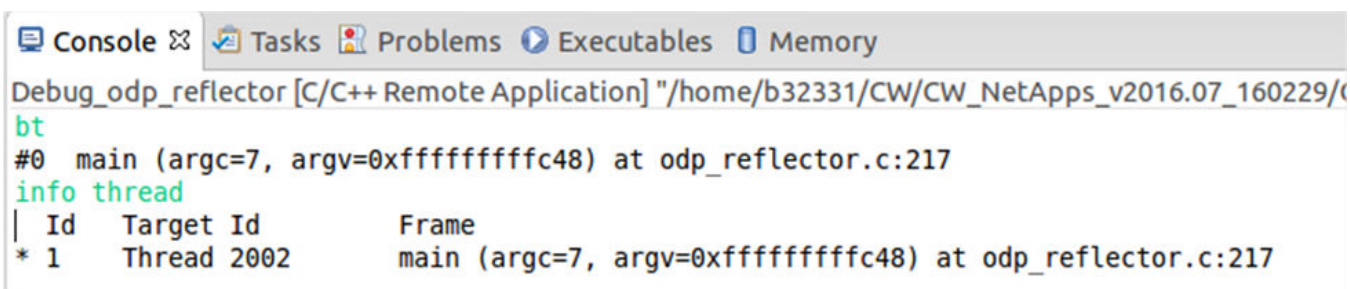


Figure 18. Console view

In the **gdb traces** console, you can see full trace details about the gdb client commands running on the host Linux.

In the **Remote Shell** console, you can see the remote commands (and outputs) which are executed by CodeWarrior on the remote Linux target.

In the **gdb** console, you can run gdb commands.

2. **Source path mapping resolved automatically** by the CodeWarrior software if you are running the CodeWarrior software on the same machine where the reflector is built.

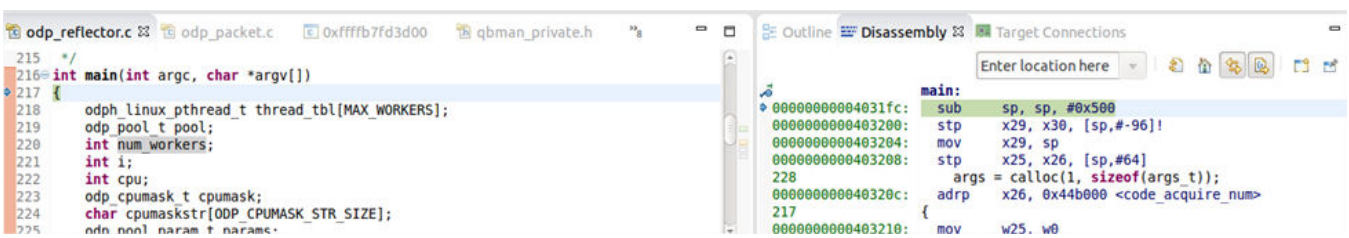


Figure 19. Source path mapping resolved automatically

If not, the CodeWarrior software will ask you where the reflector source files are located as shown below.

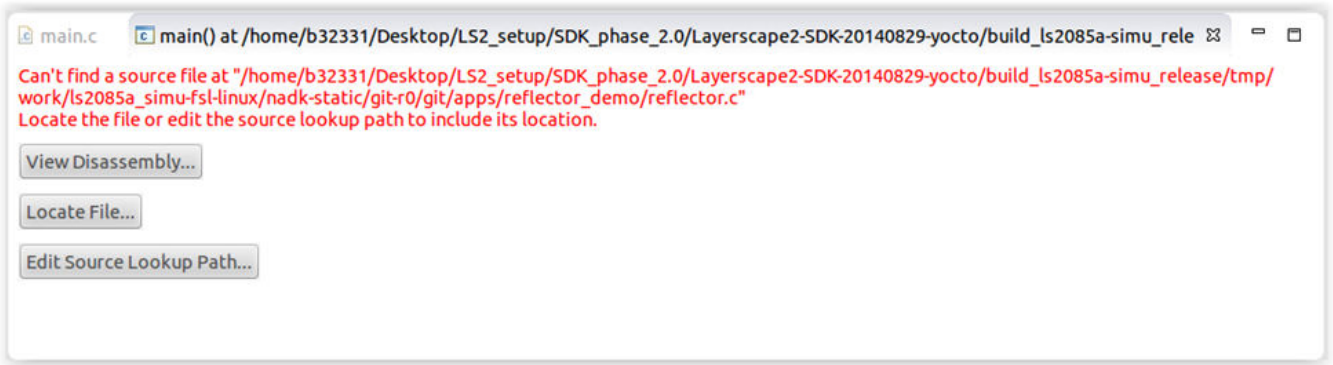


Figure 20. Not able to locate reflector source files

3. **Access to registers and memory, both from the GDB and CodeWarrior views**.

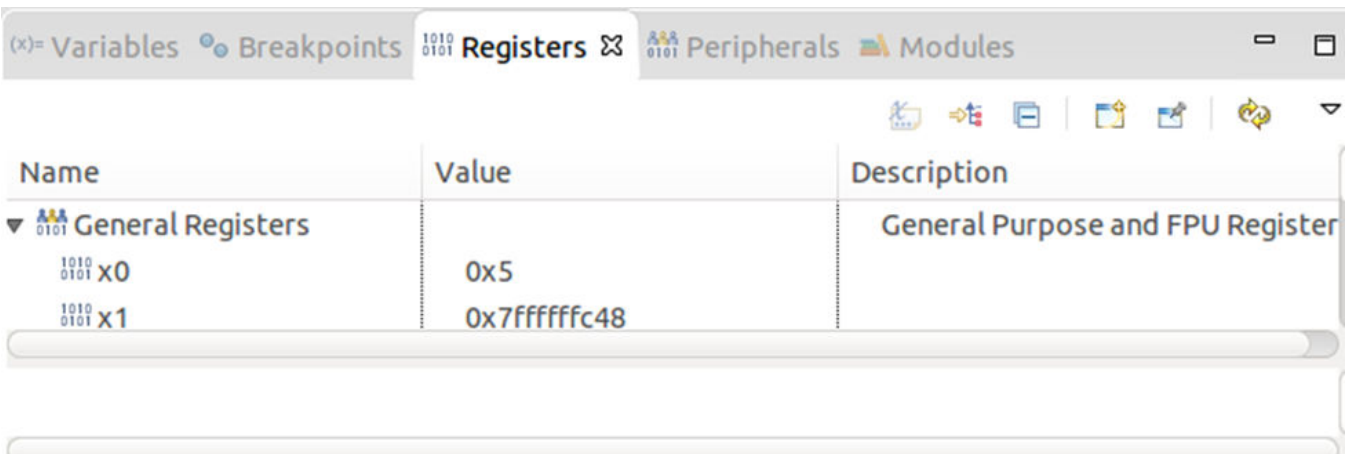


Figure 21. Registers in CodeWarrior view

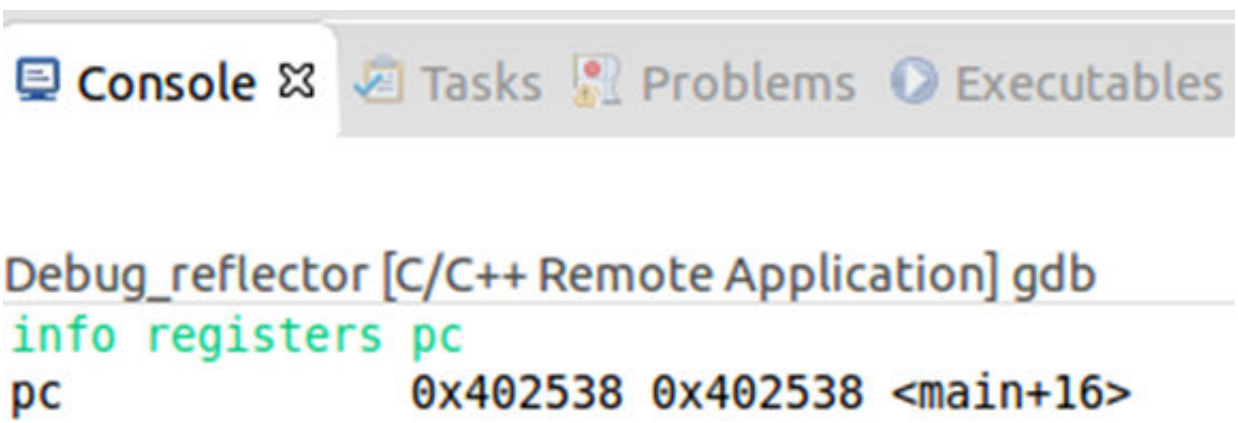


Figure 22. Registers in GDB console view

4. **Run control per process, per thread**
   By default, the run control is enabled per process, but you can enable it per thread, using the scheduler-locking option in the gdb console. For example, you can set a breakpoint at the line 193 in `odp_reflector.c`, run until the breakpoint, and then you can try the scheduler-locking to be set on. Using `<info threads>` and `<thread id>`, you can switch between different threads and perform run control operations (stepi, step, next) per thread.

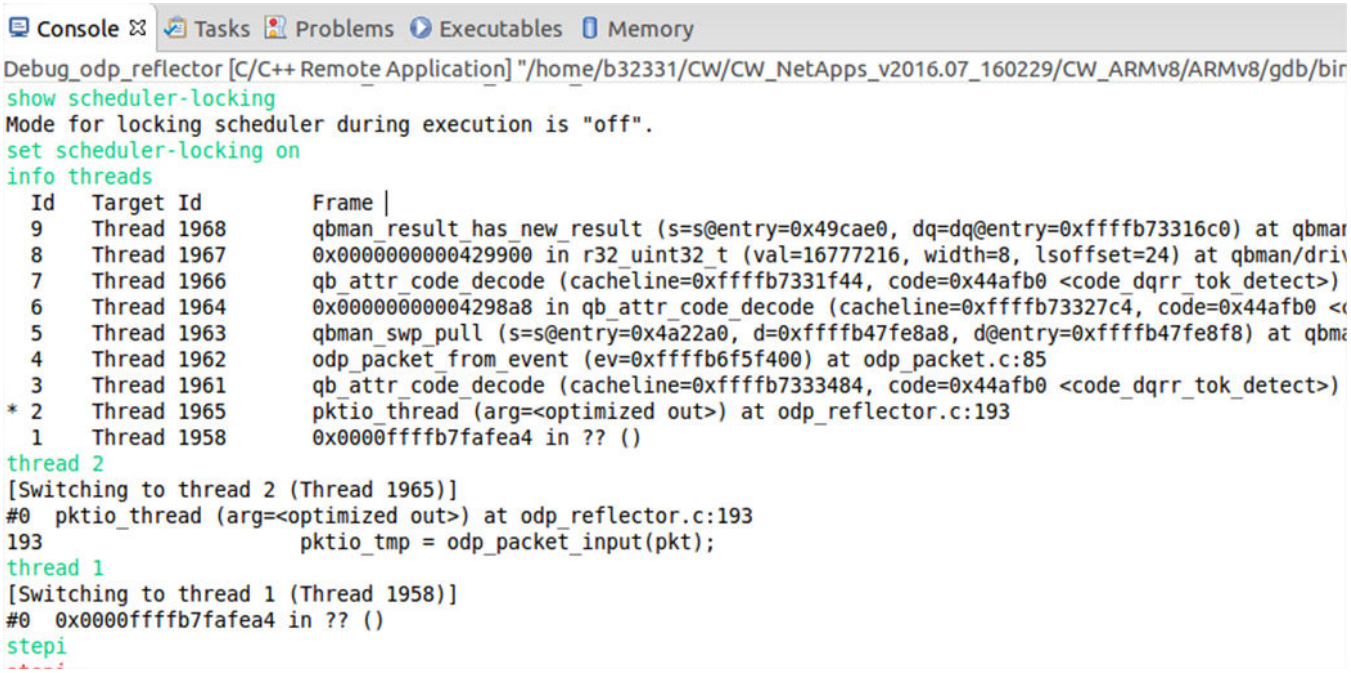
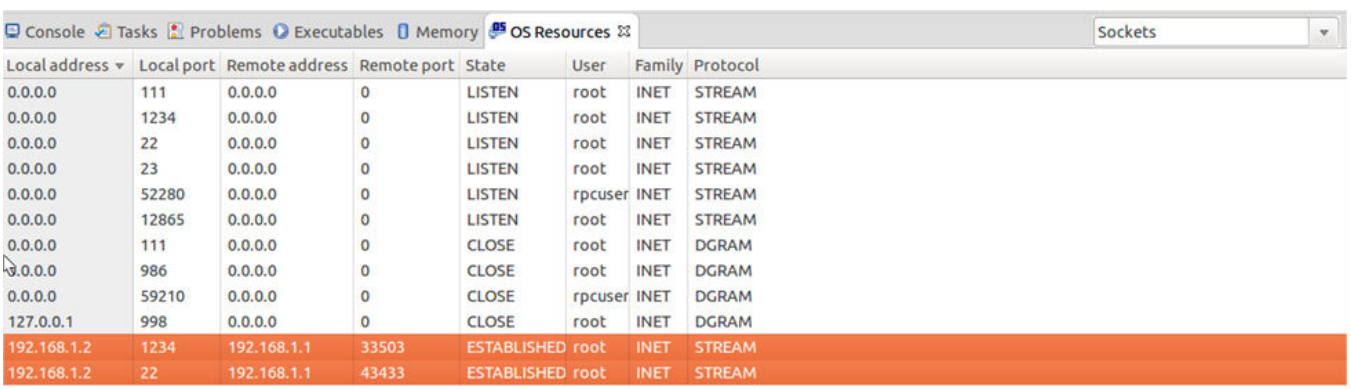


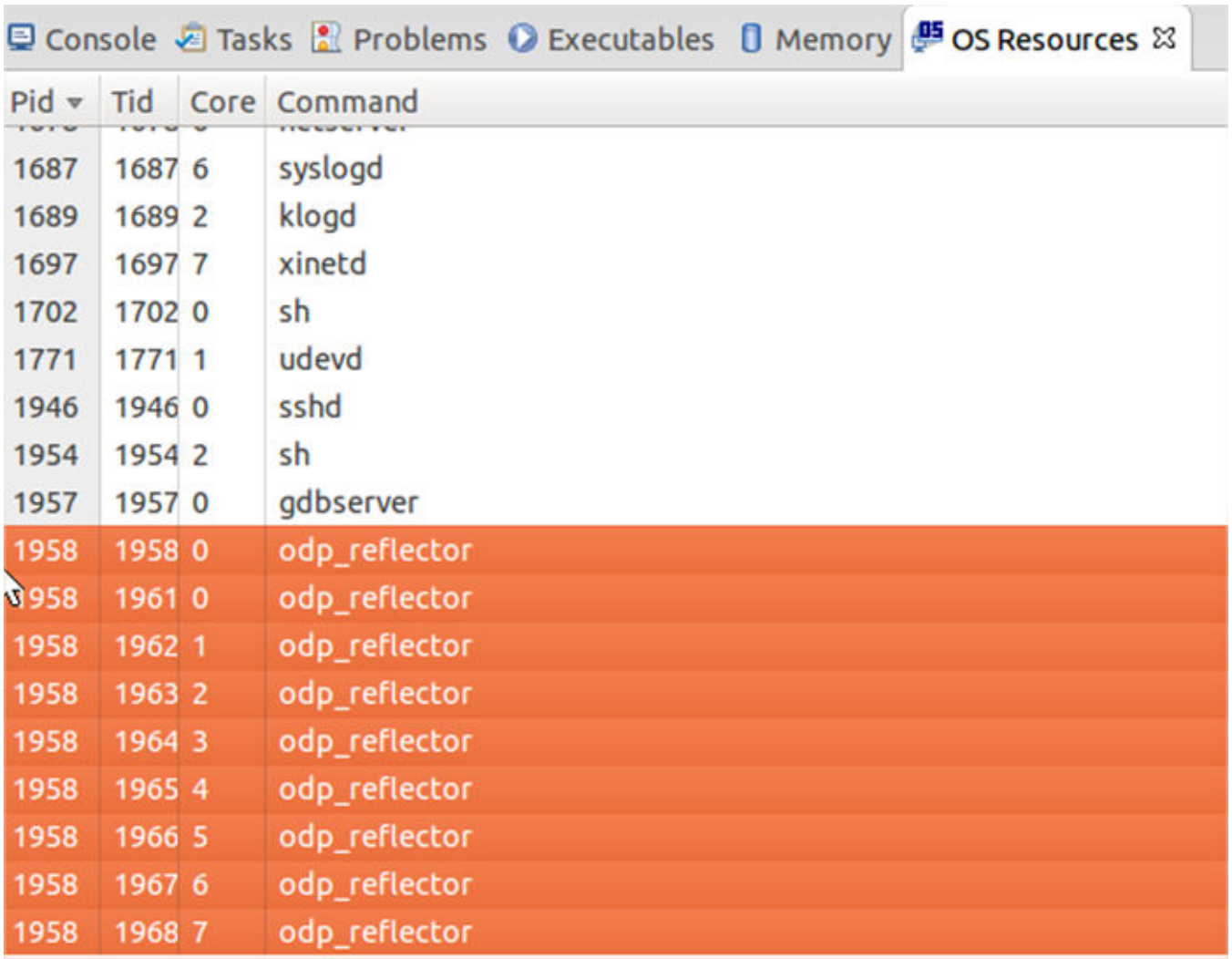
Figure 23. Run control per process, per thread

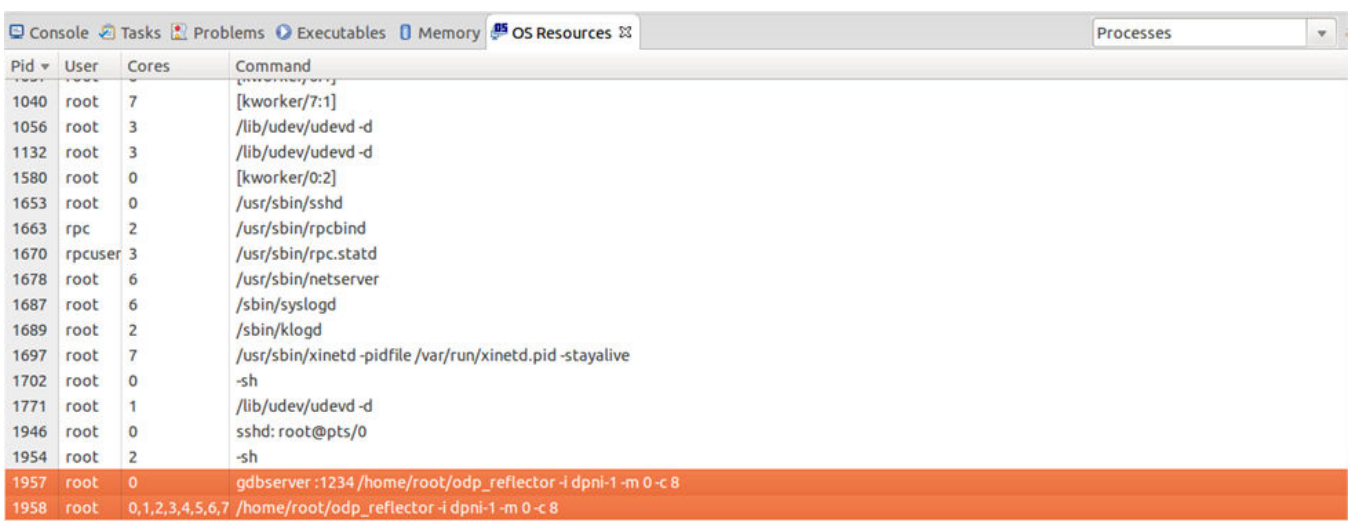
5. **OS Resources**: You can view information about processes (PID), Threads (TID), Sockets, shared-memory regions. To open this view, select **Window > Show view > Other > OS Resources**.



Local address	Local port	Remote address	Remote port	State	User	Family	Protocol
0.0.0.0	111	0.0.0.0	0	LISTEN	root	INET	STREAM
0.0.0.0	1234	0.0.0.0	0	LISTEN	root	INET	STREAM
0.0.0.0	22	0.0.0.0	0	LISTEN	root	INET	STREAM
0.0.0.0	23	0.0.0.0	0	LISTEN	root	INET	STREAM
0.0.0.0	52280	0.0.0.0	0	LISTEN	rpcuser	INET	STREAM
0.0.0.0	12865	0.0.0.0	0	LISTEN	root	INET	STREAM
0.0.0.0	111	0.0.0.0	0	CLOSE	root	INET	DGRAM
0.0.0.0	986	0.0.0.0	0	CLOSE	root	INET	DGRAM
0.0.0.0	59210	0.0.0.0	0	CLOSE	rpcuser	INET	DGRAM
127.0.0.1	998	0.0.0.0	0	CLOSE	root	INET	DGRAM
192.168.1.2	1234	192.168.1.1	33503	ESTABLISHED	root	INET	STREAM
192.168.1.2	22	192.168.1.1	43433	ESTABLISHED	root	INET	STREAM

Figure 24. OS Resources view





6. **Breakpoints** can be read/written from the IDE or gdb console.

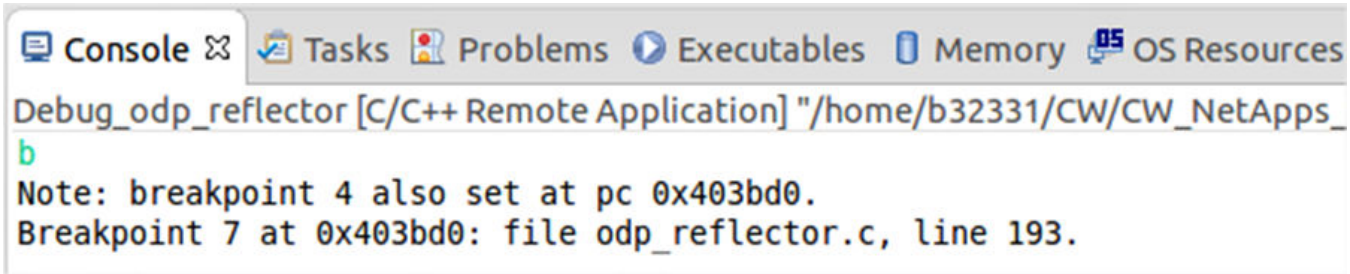
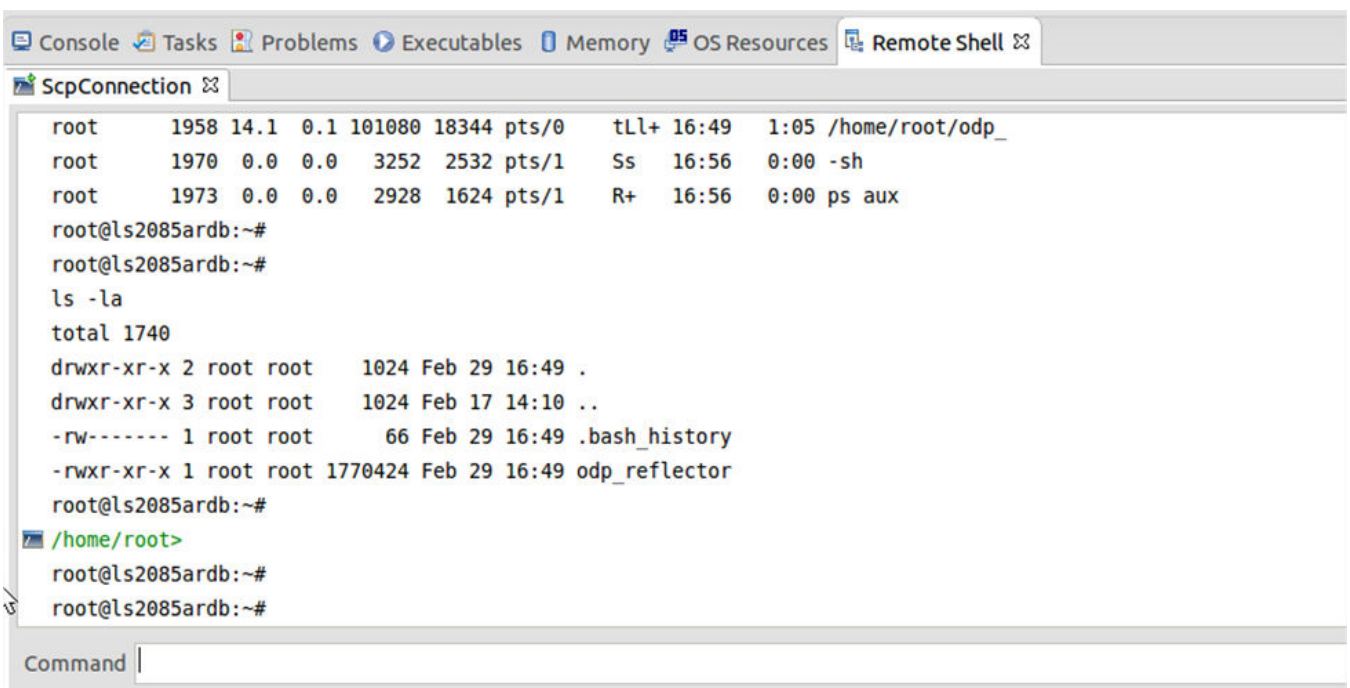


Figure 25. Breakpoints view

7. **Full Remote Shell console**. You can enable it from **Window > Show view > Other Remote Shell**.

Figure 26. Remote Shell view



8. **Dynamic printf** – Right-click the left side panel of the source code to see the menu below. A dynamic printf works like a breakpoint and when this is hit, a custom message is printed. This feature is very useful for debug.

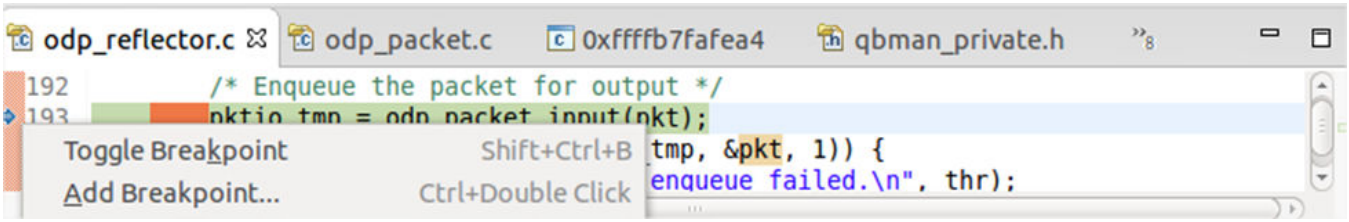
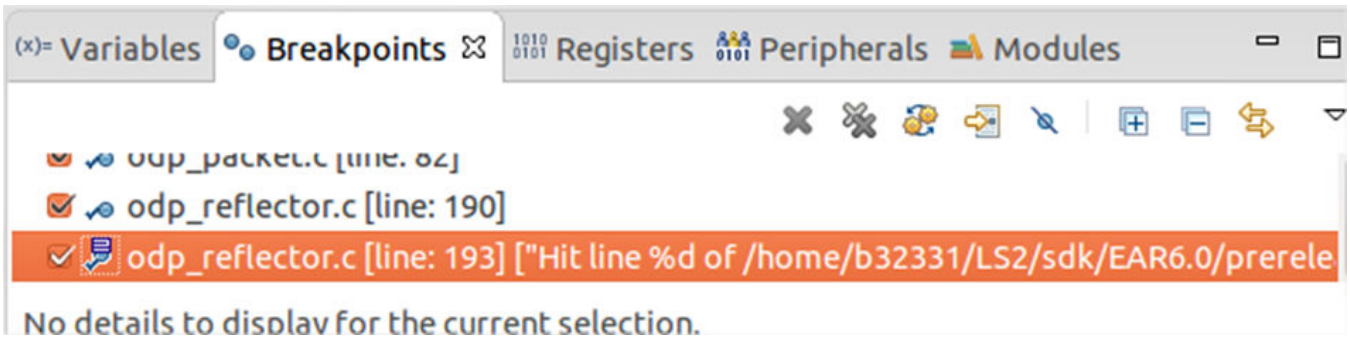


Figure 27. Dynamic printf



How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number AN5269
Revision 06/2018