NXP Semiconductors
Application Note

Document Number: AN5128
Rev. 11.3.2, 08/2018

Linux Kernel and User Applications Debug Print using CodeWarrior

1 Introduction

The Linux Debug Print tool encapsulates a target server responsible for collecting Kernel Ring Buffer log user space applications messages in the unformatted way and a host which requests periodically the kernel log data from the server and displays it in a view.

The main objective of this tool is to provide a user-friendly way of monitoring the activities in a CodeWarrior console. It is composed of several modules:

- **Target side**:

  Debug Print server – reads on demand, the Kernel Ring Buffer log. It optionally clears the log and sends it to the clients using TCP/IP connection. It collects the redirected standard output from the user space applications.

  Debug Print dynamic library - is responsible for redirection of the user space application's standard output messages to the target server.
- **Host side**:

  Debug Print probe – is the actual client of the Debug Print server; it can be started from the **Debug Print** view. When started, it reads periodically the kernel log data from the server and sends it to the **Debug Print** view to display the kernel log data and other communication messages.

Contents

Debug Print view – displays the log data and other communication messages in a user-friendly manner, also allows to filter the displayed data on the basis of timestamp, module name/application path and pid, or a custom string contained in each log message.

NOTE

The Arm binaries have been compiled with tool chain *gcc-linaro-aarch64-linux-gnu-4.9.3* and LS2 SDK.

NOTE

The Debug Print is a standalone tool. It is independent from the other CodeWarrior components and does not require a debug session.

2 Debug Print tool functionality

Perform the following steps in order to see the functionality of the Debug Print tool.

1. **Configure Debug Print server**
2. Configure Debug Print library
3. **Start Debug Print probe**
4. **Open Configure Debug Print dialog**
5. **Configure Debug Print settings in Preferences dialog**

2.1 Configure Debug Print server

The debug print target server cross-compiled for Arm is located in CodeWarrior in directory: *<CWInstallDir>/ARMv8/sa_ls/linux.armv8.debugprint/bin*, which needs to be copied on the target (for example, to the home directory), using Remote System Explorer view, or an SCP connection, or manually if you have the target root file system on NFS.

The server command line is `ls.target.server [PORT] [-k]` and requires a single argument; the port number on which clients will listen. If not specified, it will start on the default port 5000. Specify `-k` to keep the kernel buffer unaltered (same as dmesg), with a server processing overhead.

Start a *ssh* console on the target and then start the server:

```
# ssh root@target_ip_address
```

```
# ./ls.target.server
```

You can access the server either as root or as a normal user. With root access, server processing overhead is less.

Accessing the server from root differs from accessing the server as a normal user in the following ways:

- Root: More efficient from both processing and communication point of view. This is because, by default, root access clears the kernel buffer after reading the messages and sends only the new messages generated by the kernel to the host, with no additional processing overhead. Another advantage of running as root is the timestamp synchronization between the kernel and the user space messages.
- User: By default, a normal user access reads all the kernel messages and sends them to the host. The detection of the new messages is done on the host, by maintaining a history of the last few messages. This has an overhead on the communication size, since buffer is always sent to host, but no other processing is done on the target.
- Both: Option `-k`, which stands for `keep` does not clear the kernel buffer, but uses an internal server logic for determining which are the newer messages, by maintaining a history in the target memory. This has the same communication efficiency as if clearing the kernel buffer, but adds a processing overhead on determining the newer messages.

2.2 Configure Debug Print library

The dynamic library cross-compiled for Arm is located in CodeWarrior directory at: `<CWInstallDir>/ARMv8/sa_ls/linux.armv8.debugprint/lib`, which needs to be copied on the target using the Remote Systems Explorer (RSE) view, or an SCP connection, or manually if you have the target root file system on NFS. This library must be loaded by the shell before the C runtime when you are running the user space applications which need to be monitored by setting the environment variable LD_PRELOAD.

NOTE

> The code for the `test-arm` application is available at the Test application section.
>
> To compile this code, create a Linux application project, replace the default code in the Linux application project with the test-arm application code, compile the application, and transfer the application to the board.

Preload the debug print library and run the test application:

```
# export LD_PRELOAD=~/libls.linux.debugprint.lib.so
```

```
# ./test-arm
```

or

```
# LD_PRELOAD=~/libls.linux.debugprint.lib.so; ./test-arm
```

You will notice next time that the test application will not display any of its standard output messages to the console, but only its standard error messages.

The standard output is sent to the target server.

2.3 Start Debug Print probe

On the host machine, open the **Debug Print** view. The Debug Print Probe can be started from the **Debug Print** view and it communicates using TCP/IP connection with the server. When started, it reads periodically the kernel log data from the server and sends it to the **Debug Print** view to display. To open the **Debug Print** view, select **Window > Show View > Other > Software Analysis > Debug Print**. The **Debug Print** view appears.

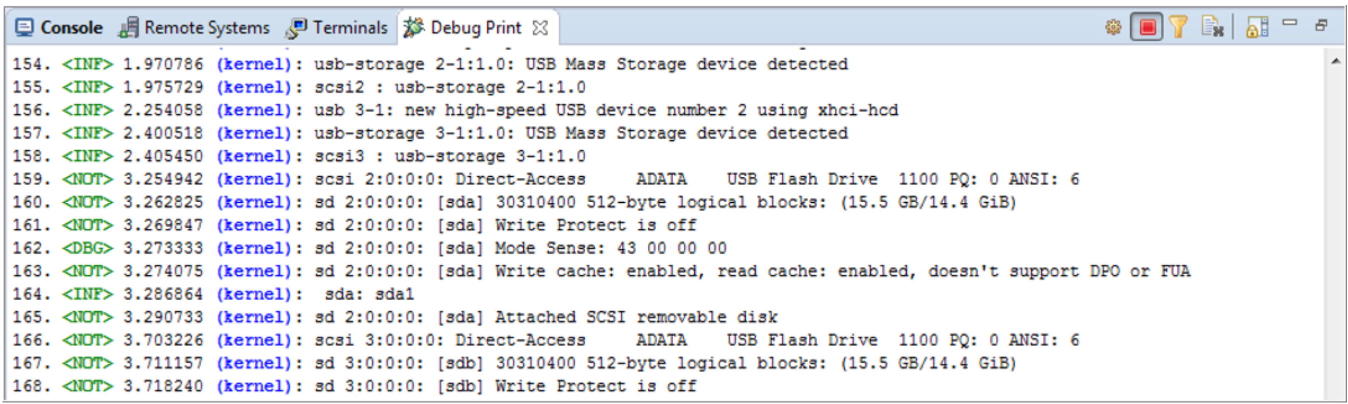


Figure 1. Debug Print view

The table below describes the icons available in the Debug Print view.

Table 1. Debug Print view icons

Icons	Description
Clear All	Removes all text from the view.
Start/Stop	Two-state button used for starting and stopping the Debug Print probe.
Scroll Lock/Unlock	Two-state button used for locking and unlocking the scrollbar. If the scrollbar is unlocked, it would always auto-scroll to the latest Debug Print message.
Configure	Opens a dialog for entering the server address and port.
Create Debug Print Filters	Opens a dialog for configuring what information is to be displayed in the **Debug Print** view (specific to timestamp, module name/application path and pid, other string patterns).

Additionally, the text manipulation Eclipse command Copy (CTRL-C) is available.

2.4 Open Configure Debug Print dialog

To configure the **Debug Print** server, click **Configure** icon on the toolbar. The **Configure Debug Print** dialog appears. You can specify the server address, port number at which the server will listen to client, and the target description (for example, address 192.168.0.2, port 5000 – must be the same as for the server at which the server will listen to client, and the target description).

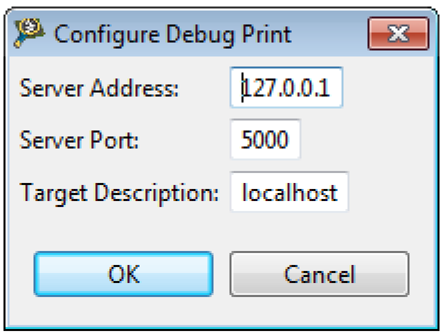


Figure 2. Configure Debug Print dialog

2.5 Configure Debug Print settings in Preferences dialog

There is also a **Preference** page associated to **Debug Print** view, which can be accessed by clicking **Window > Preferences**, expanding **Software Analysis** node, and then selecting **Debug Print**.

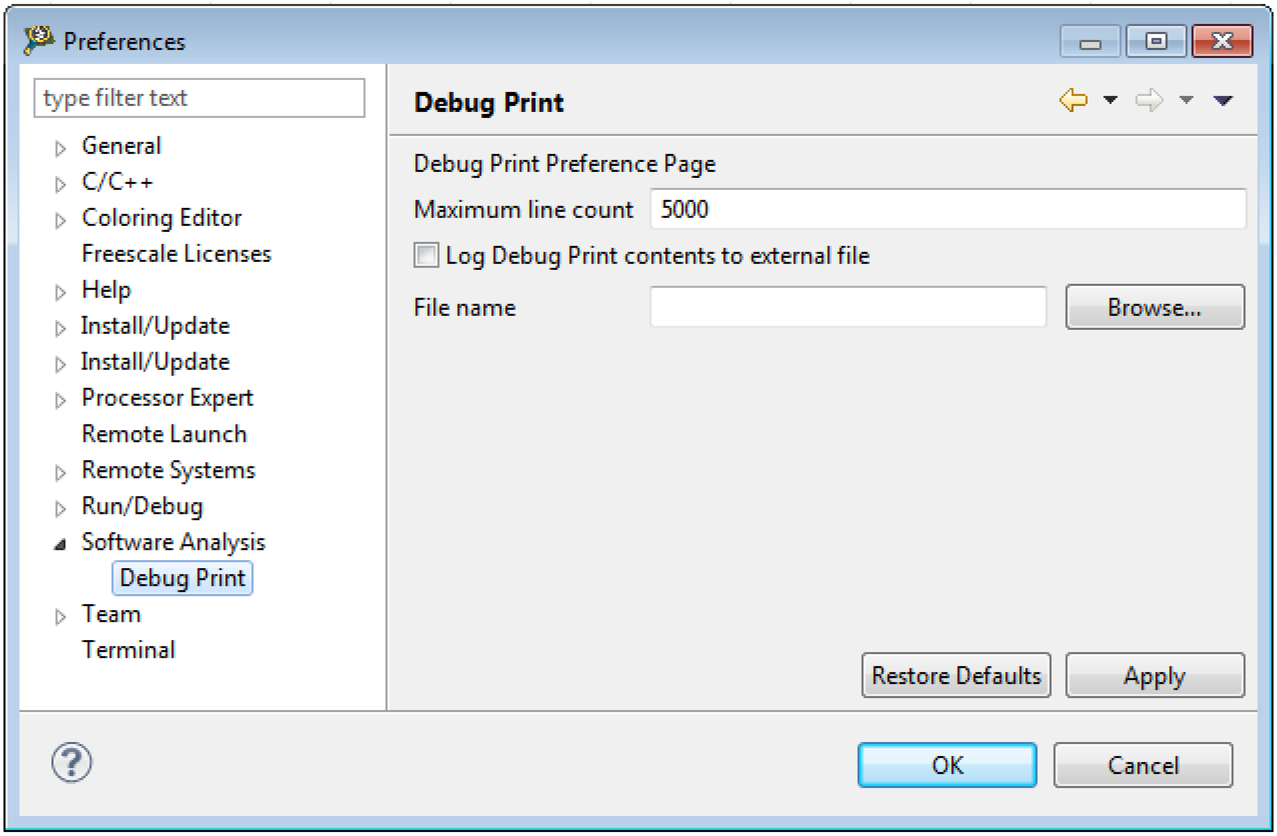


Figure 3. Preferences dialog

Table 2. Debug Print settings

Options	Description
Maximum line count	Limits the number of lines the **Debug Print** view should display. If this limit is exceeded, the old messages are deleted.
Log Debug Print contents to external file	If selected, the messages will be appended to an external file besides displaying them into the **Debug Print** view.
File name	Path for the external log file

2.6 Create Debug Print filters

The **Create Debug Print Filters** configuration dialog allows creation of multiple filters, each of them able to match the module name, application path, or PID of the messages displayed by the **Debug Print** view. These filters are OR-ed, which means that the view will display all messages which match at least one of the filters.

This dialog has three tabs:

- **Module** tab: allows creation of new filters, by selecting from the **Existing** list a module name/application path, PID, or both (if available). Click **Add Filter** to add the filter in the **Current Filters** list. These filters can be qualified with a timestamp range or a string pattern.

The **Existing** list contains all the module names/application paths/PIDs from the messages already displayed in the **Debug Print** view. When you want to filter messages from a certain module or application that is not started or did not print any messages yet, you can manually enter the module name/path or PID in the **Custom** text box.

When no module filter is selected, and no global qualification is selected, **(any)** is displayed in the **Current Filters**, which means that no filter is applied (all messages are displayed).

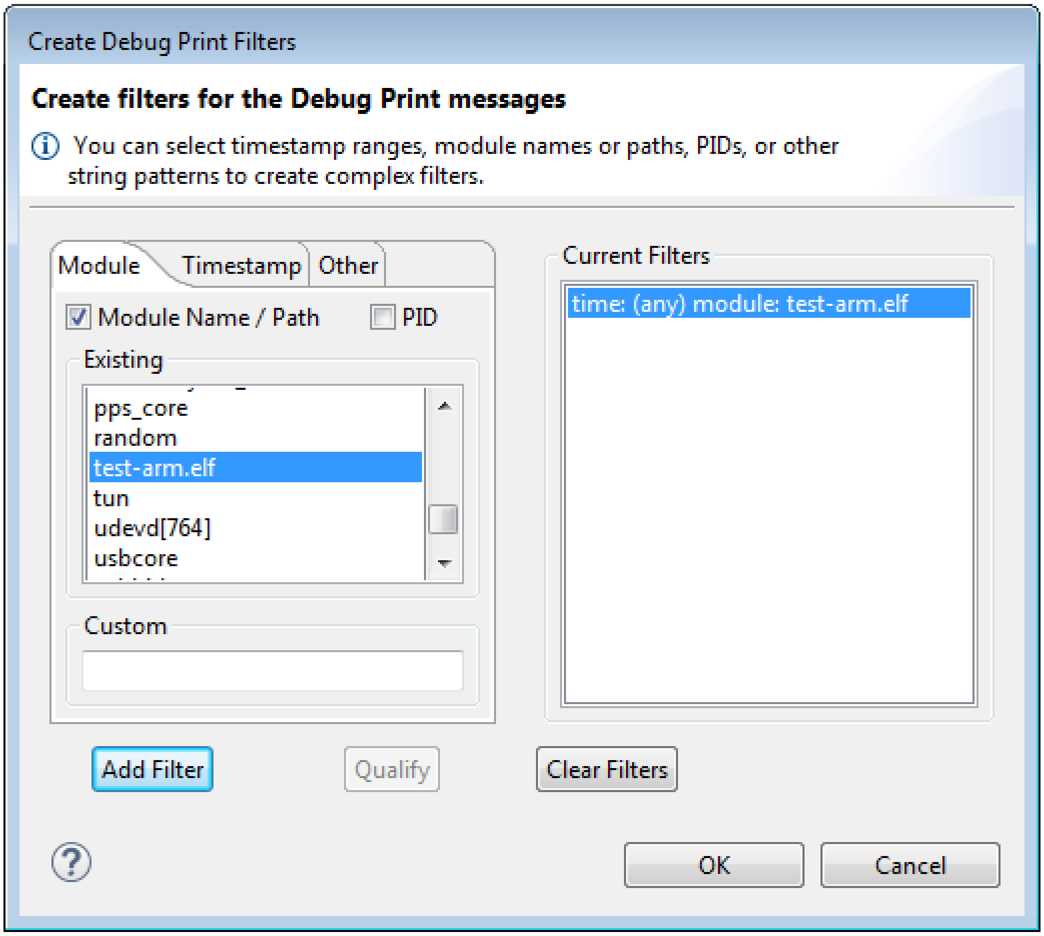


Figure 4. Create Debug Print Filters dialog - Module tab

- **Timestamp** tab: allows adding timestamp qualification to the existing filters, or a global qualification if no other filter is created (that is a generic filter which applies to all messages, with all module names, paths and PIDs).
  After the user choses the timestamp ranges in the Lower Limit/Upper Limit Spinners, you must click **Qualify** in order to add the timestamp qualification to all existing filters. If no filter exists, a global qualification is performed.

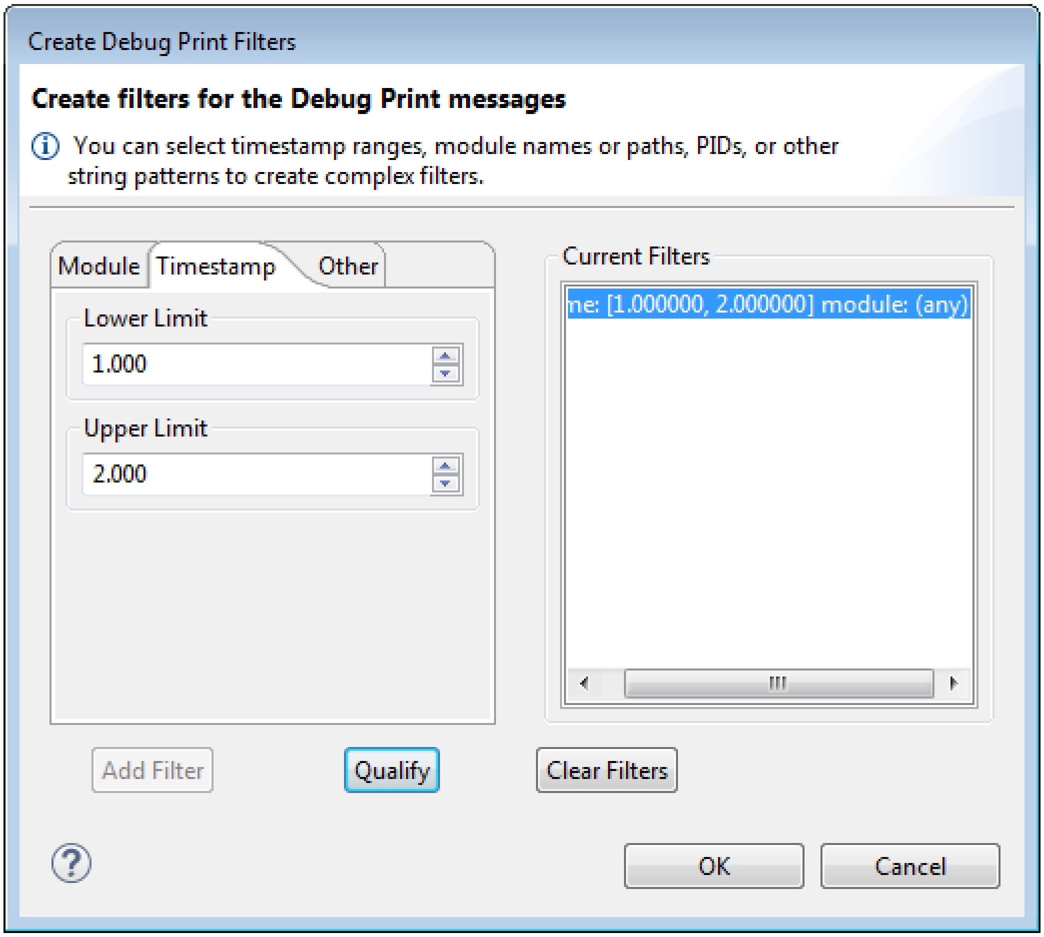


Figure 5. Create Debug Print Filters dialog - Timestamp tab

- **Other** tab: allows adding other type of qualifications to existing filters, or a global qualification if no other filter is created. Currently, the only qualification in this tab is a string pattern which is searched in all the messages (except for timestamps and module names/paths/PIDS). After you input the string pattern, you must click **Qualify** in order to add this qualification to all the existing filters. If no filter exists, a global qualification is performed.

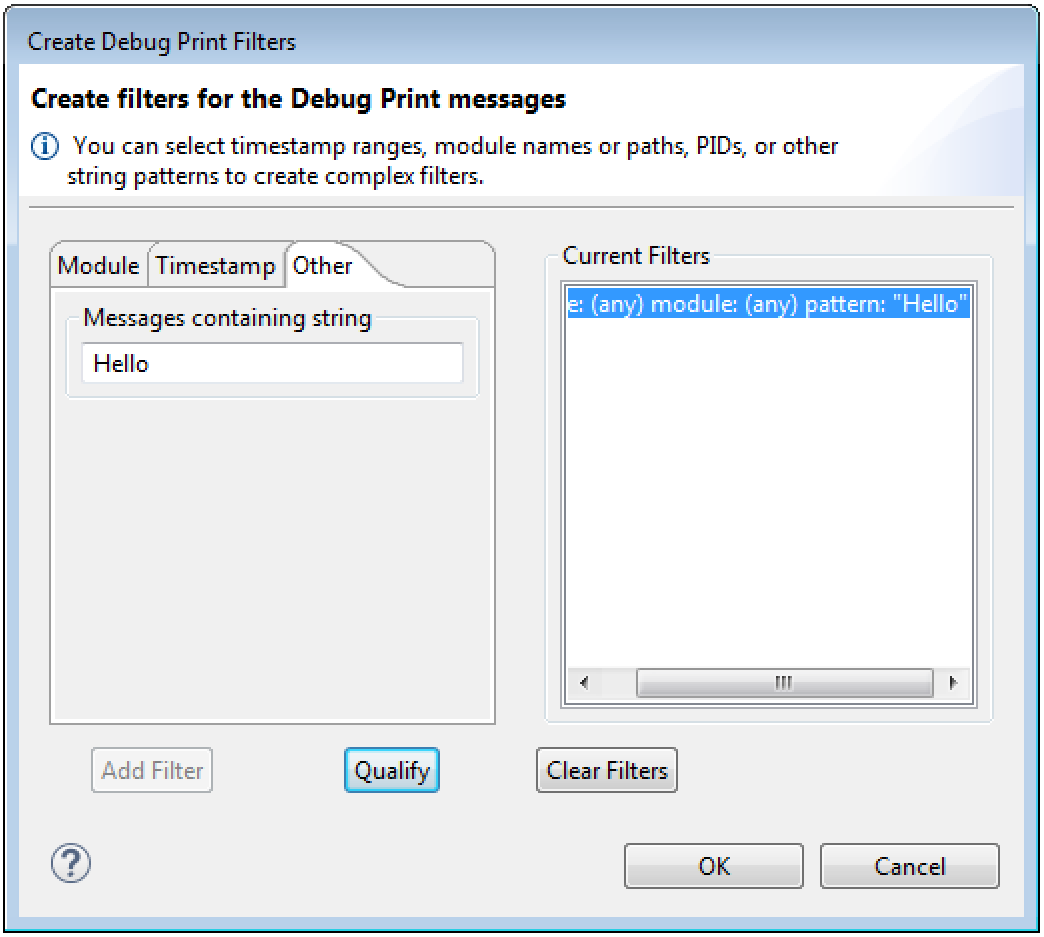


Figure 6. Create Debug Print Filters dialog - Other tab

3 Using Debug Print with Remote Systems Explorer

Remote Systems Explorer (RSE) can be used to browse the target file system, transfer files to the target directly from the CodeWarrior software, and start ssh consoles.

To enable RSE:

1. Select **Windows > Preferences**. The **Preferences** dialog appears.
2. Select **Remote Systems** in the left panel.
3. Set **Linux** and **SSH Only** system types to **True**.
4. Click **Apply > Apply and Close**.

When the target is connected to the host running the CodeWarrior software, you can create a **Linux** or **SSH Only** connection to the target.

1. To open the **Remote Systems Explorer** view, click **Window > Perspective > Open Perspective > Other > Remote System Explorer**.

   The **Remote Systems** view appears.
2. Click **Define a connection to remote system** available in the **Remote Systems** view toolbar.

   The **New Connection** wizard appears.
3. Expand **General** and select **Linux** option from the list.

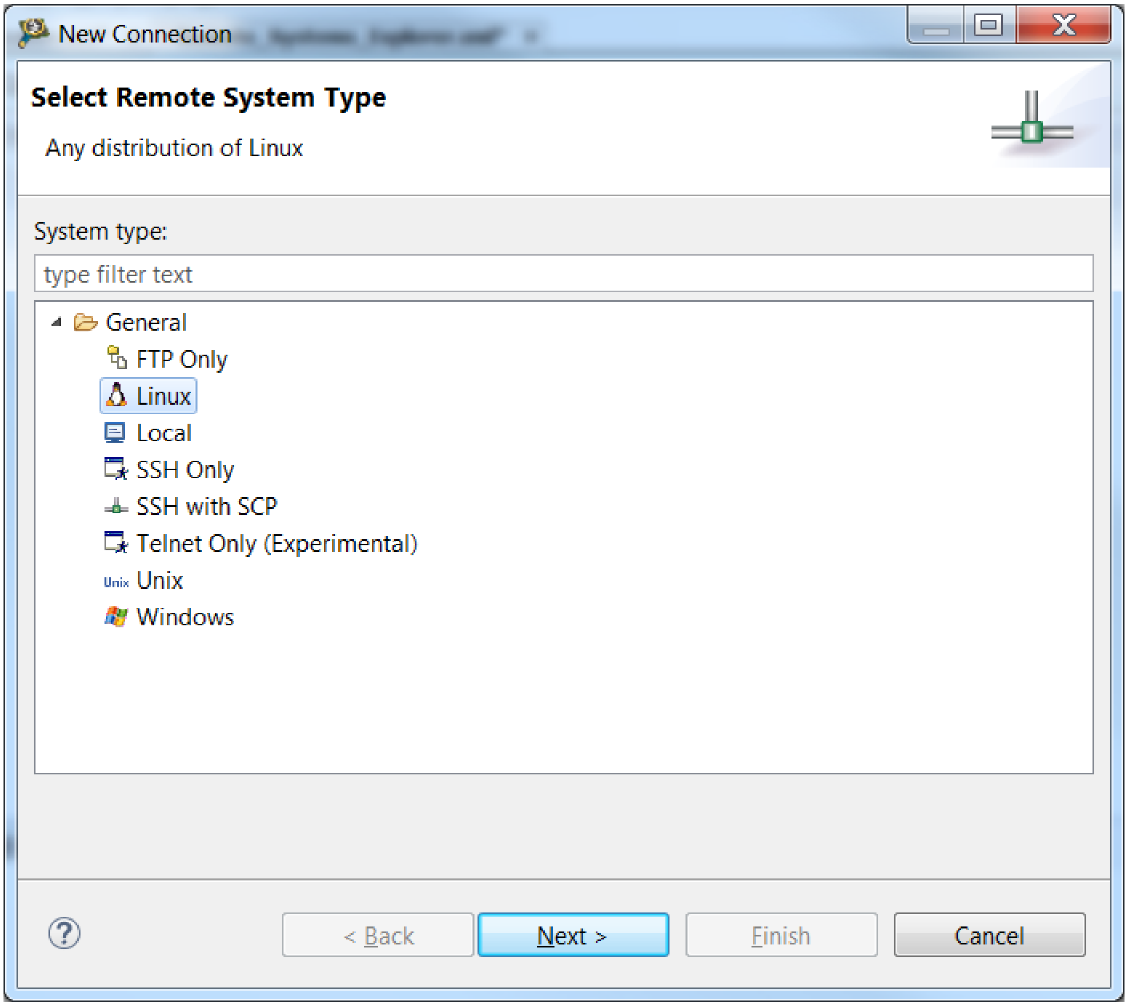


Figure 7. New Connection wizard

4. Click **Next**.

   The **Remote Linux System Connection** page appears.

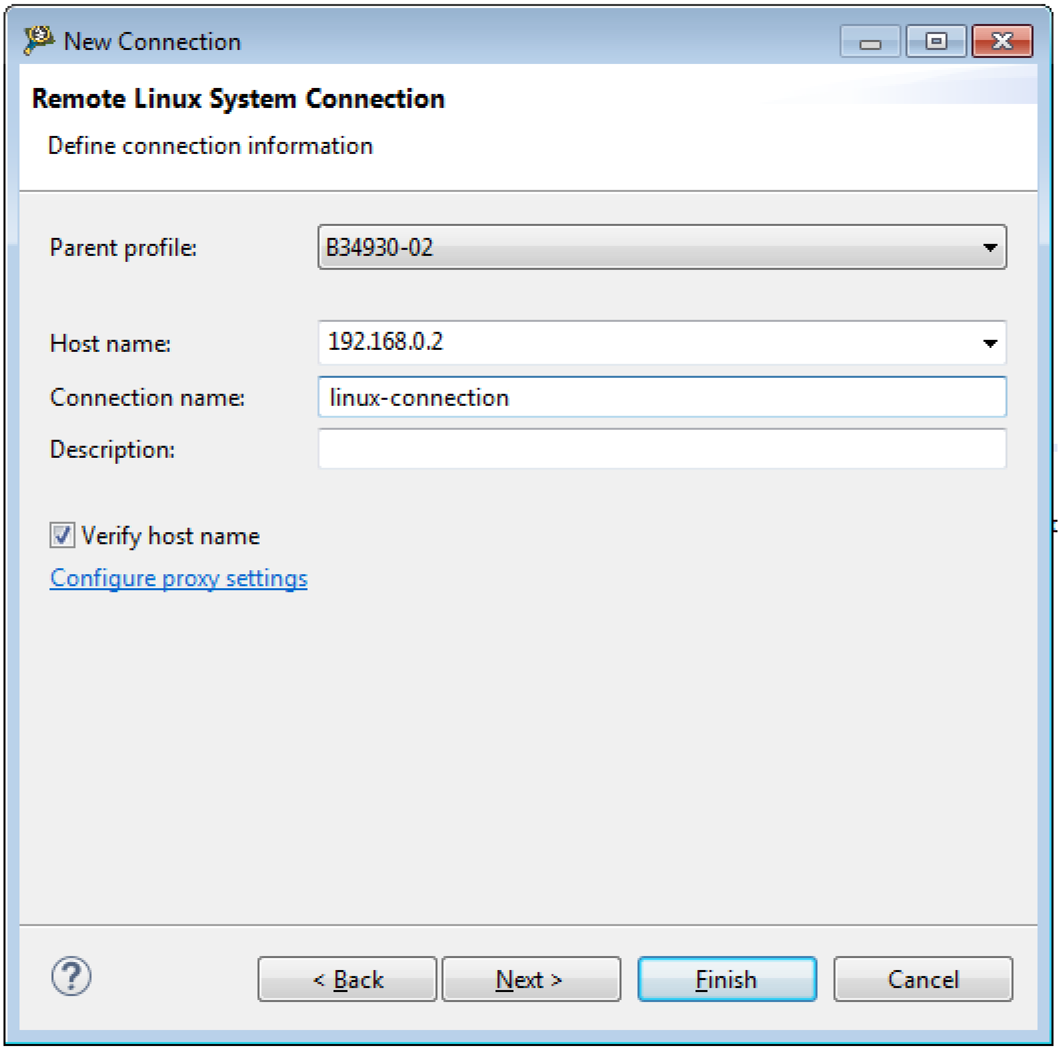


Figure 8. Remote Linux System Connection page

5. Specify the **Host name** and the **Connection name** and click **Next**.
   The **Files** page appear.

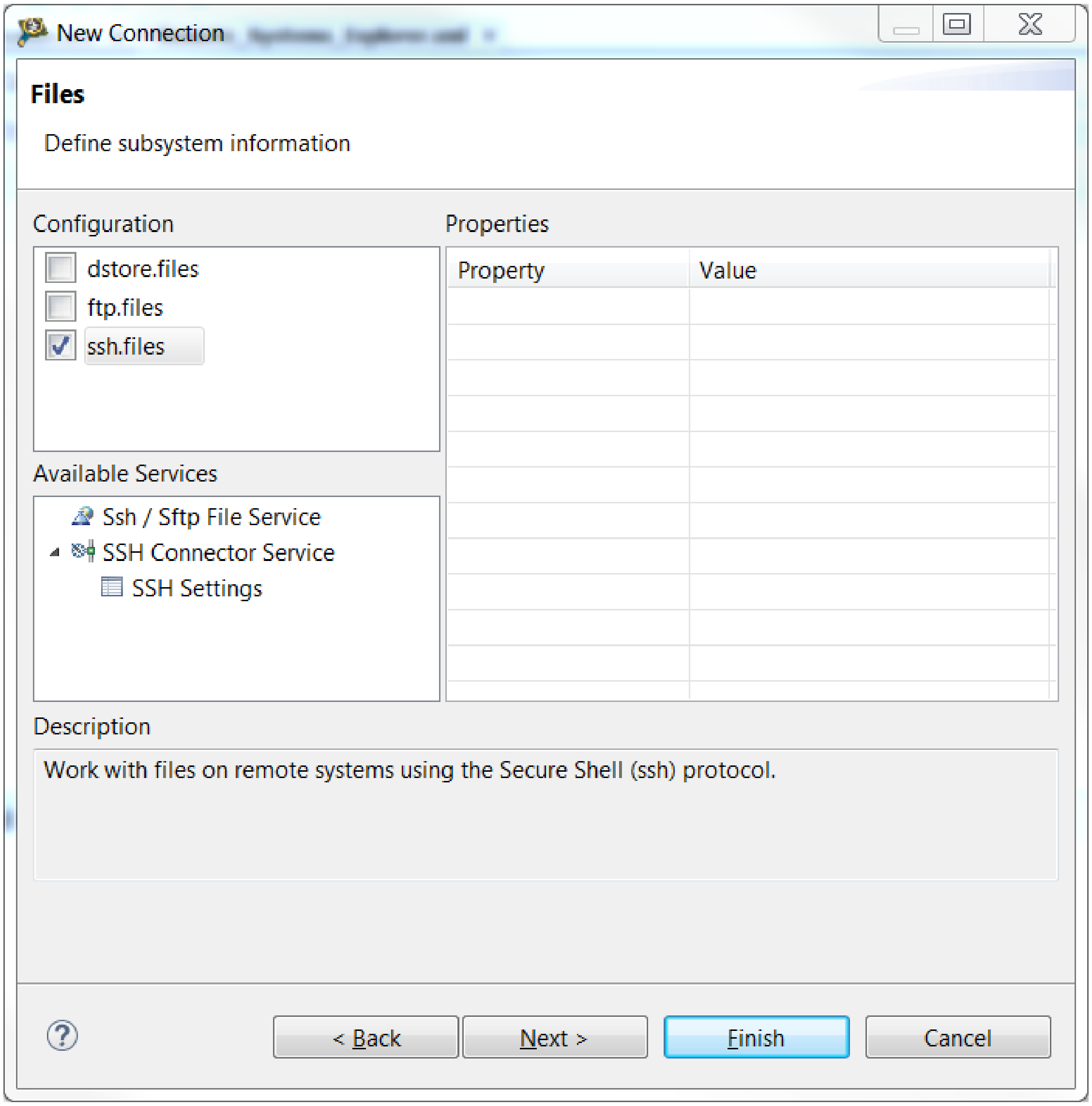


Figure 9. Files page

6. Select the **ssh.files** checkbox and click **Next**.
   The **Processes** page appears.

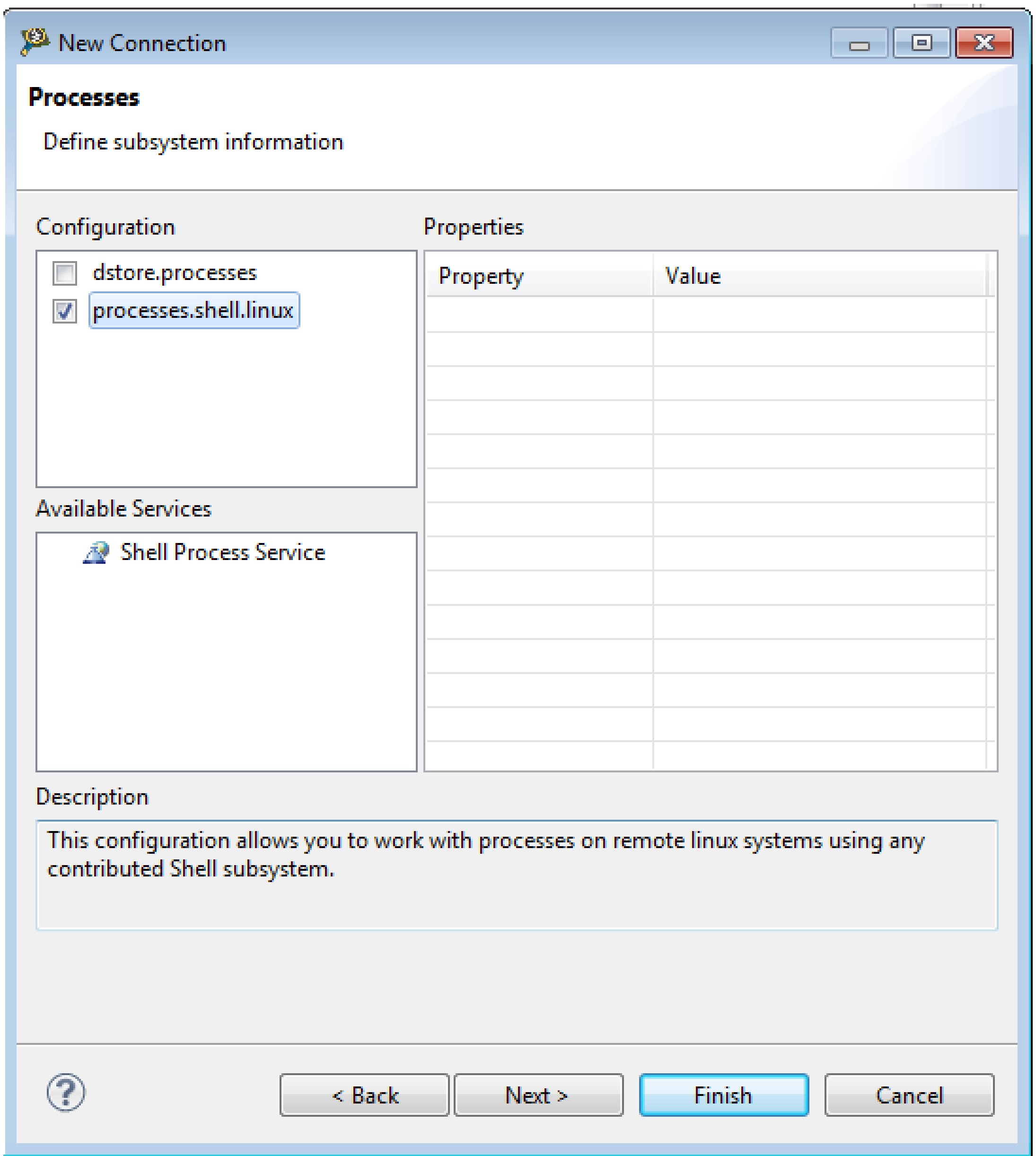


Figure 10. Processes page

7. Select the **processes.shell.linux** checkbox and click **Next**.
   The **Shells** page appears.

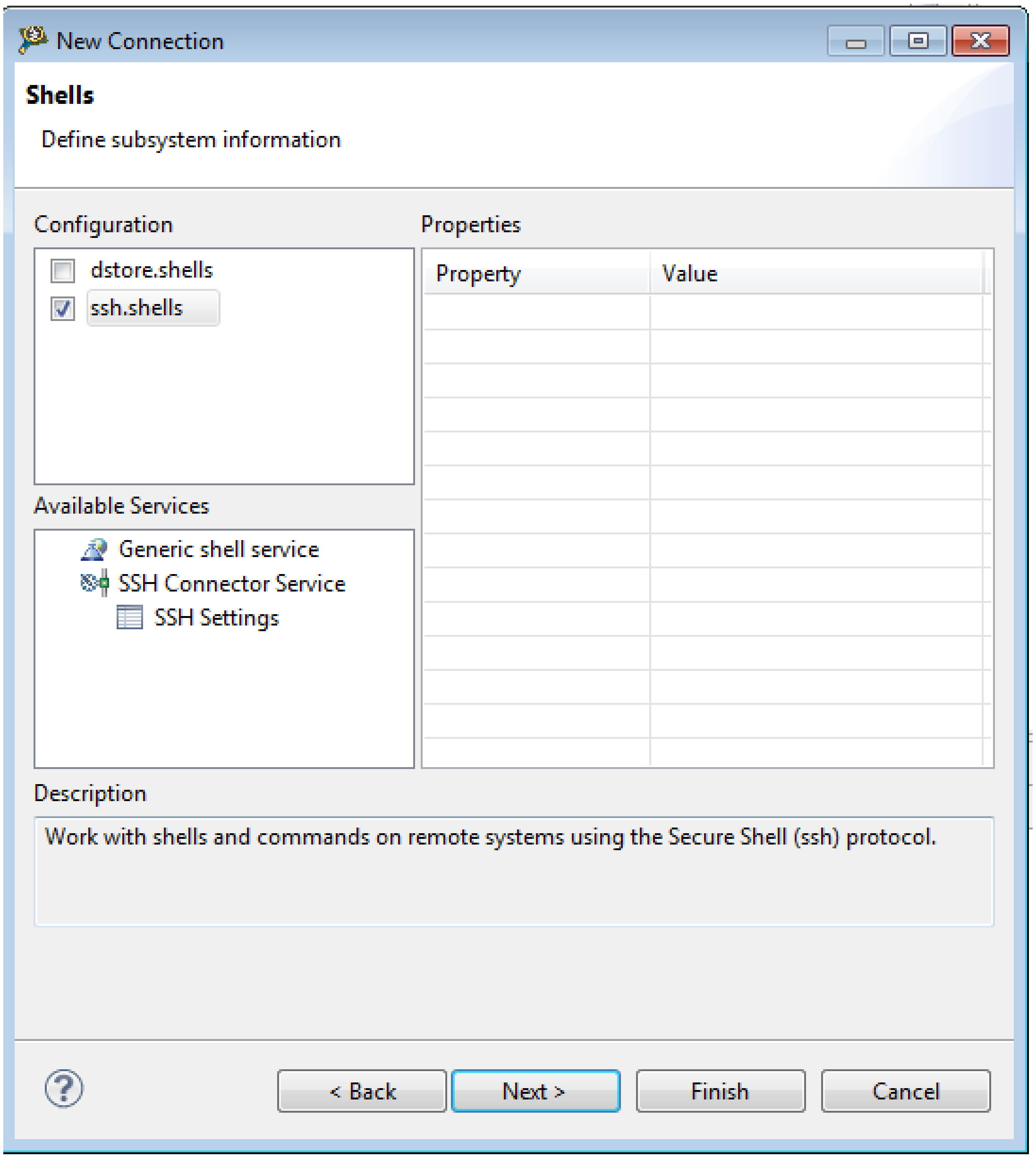


Figure 11. Shells page

8. Select the **ssh.shells** checkbox and click **Finish**.
9. In the **Remote Systems** view, you can see the new connection. The connection name is *linux-connection*.
10. Browse to the root directory to establish connection with the target board.

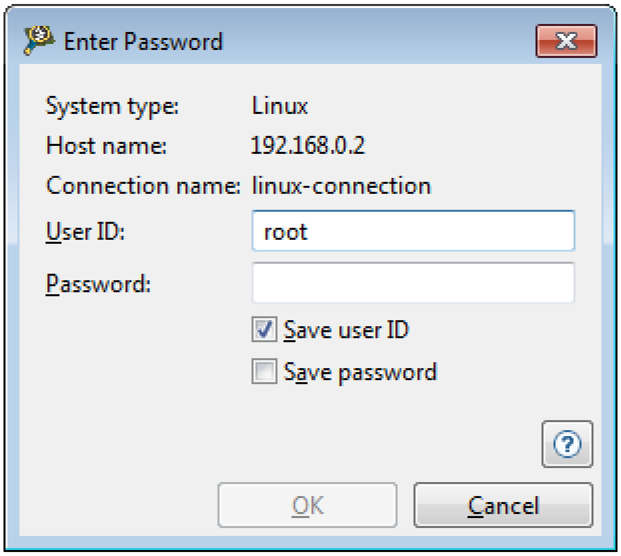
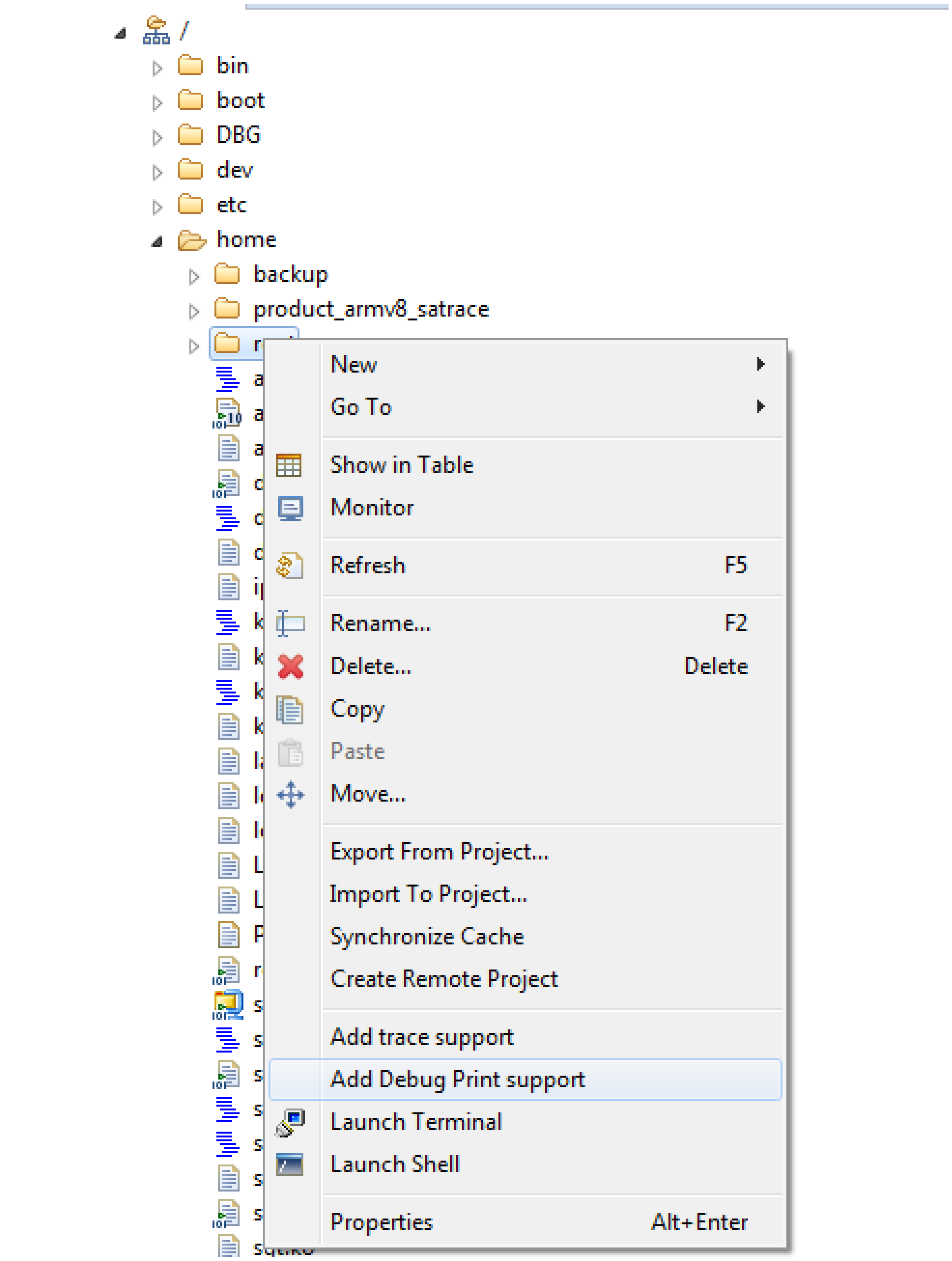


Figure 12. Establish connection to target

11. To add debug print binary, that is server or user space library, perform either of the following:
    - Right-click root home directory, select **Add Debug Print support**, and refresh the directory tree.



- Copy the debug print server and library binaries from the directory:

  ```
  <CWInstallDir>/ARMv8/sa_ls/linux.armv8.debugprint/bin
  ```

  Right-click the root home directory in the RSE view, select **Paste** to paste the binaries on the target. Then, select **Properties > Permissions** from the root home directory context menu, and set **Execute** permissions on the target server.

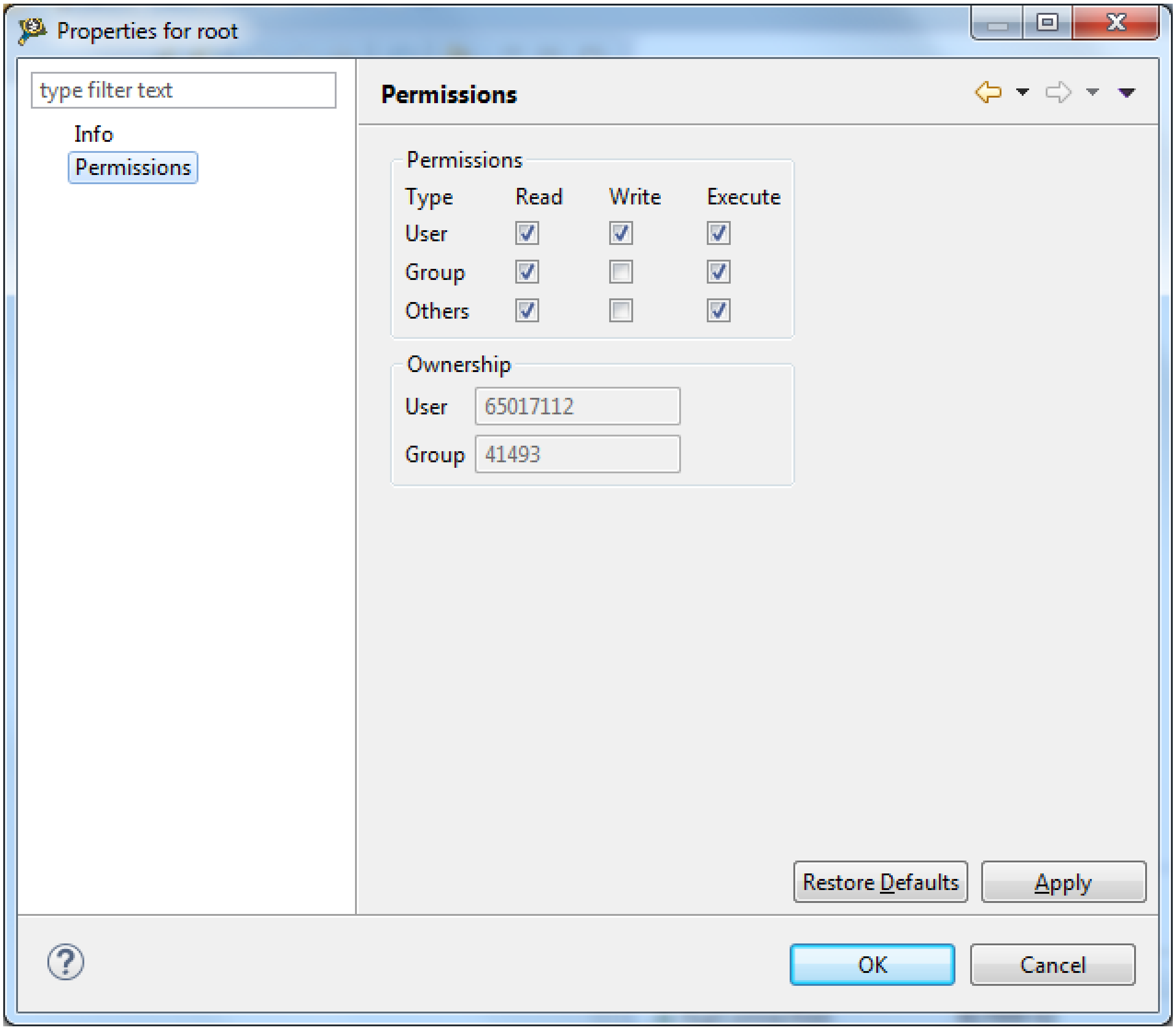


Figure 13. Set properties

12. Right-click the root home directory and select **Launch Terminal** to launch RSE ssh consoles. In this console, you can start the server or run other applications.

4 Functional examples

This section lists the following examples for ARMv8:

- **Basic ARMv8 example**
- **ARMv8 dynamic debug example**

4.1 Basic ARMv8 example

You can perform the steps in this example to see the Debug Print tool functionality. The Arm binaries are compiled with the tool chain, `gcc-linaro-aarch64-linux-gnu-4.9.3`, available in CodeWarrior for ARMv8.

Before working on the Debug Print tool, check that TCP/IP communication is established between the host and the target.

1. Deploy the Software Analysis target binaries on the target using **Remote Systems Explorer** view, or an SCP connection, or if you have the target root file system on NFS, you can copy *ls.target.server* and *libls.linux.debugprint.lib.so** to the host location *[NFS_PATH]/home/root*).
2. Start a *ssh* console on the target where the SA binaries have been deployed, and then start the server on default port 5000:

```
# ssh root@target_ip_address
```

```
# ./ls.target.server
```

3. Open the **Debug Print** view.
4. Click the (**Configure**) button, enter the server address and port. For example, 192.168.0.2, port 5000. The port number must be same as the server.
5. Click the **Start** icon; you will see the kernel log messages are being populated in the view's text area.

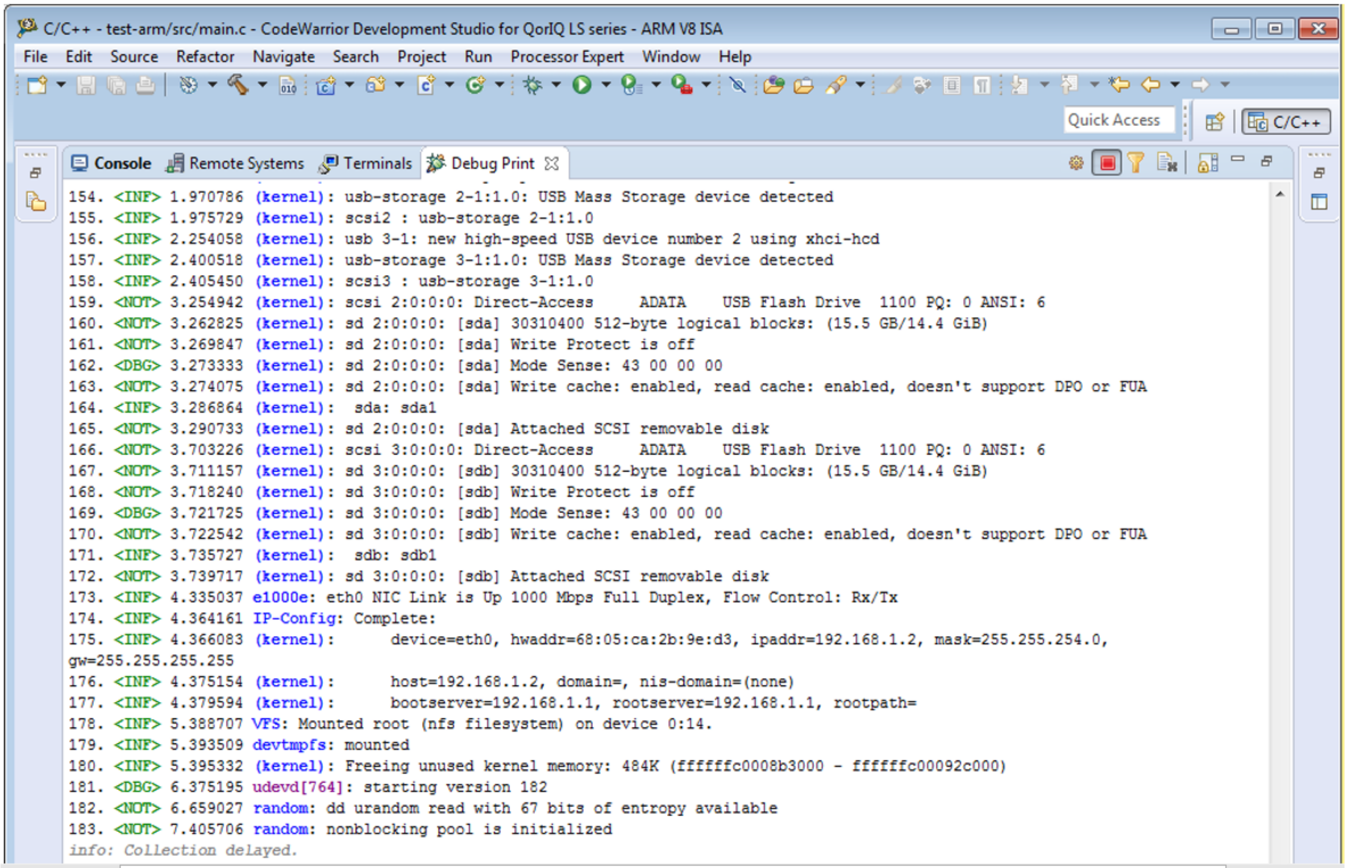


Figure 14. Debug Print view - messages from server

NOTE

The module name of the Kernel space messages is colored in Blue, the module name of the user space messages is colored in Magenta, and the message log level is colored in green. See http://linux.die.net/man/2/syslog for more information about supported log levels.

6. Open another console on the target in the same directory, preload the debug print library and run the test application:

```
# export LD_PRELOAD=~/libls.linux.debugprint.lib.so; ./test-arm
```

```
# ./test-arm
```

7. You will see the application messages getting appended in the **Debug Print** view.

```
176. <INF> 4.375154 (kernel):      host=192.168.1.2, domain=, nis-domain=(none)
177. <INF> 4.379594 (kernel):      bootserver=192.168.1.1, rootserver=192.168.1.1, rootpath=
178. <INF> 5.388707 VFS: Mounted root (nfs filesystem) on device 0:14.
179. <INF> 5.393509 devtmpfs: mounted
180. <INF> 5.395332 (kernel): Freeing unused kernel memory: 484K (ffffffc0008b3000 - ffffffc00092c000)
181. <DBG> 6.375195 udevd[764]: starting version 182
182. <NOT> 6.659027 random: dd urandom read with 67 bits of entropy available
183. <NOT> 7.405706 random: nonblocking pool is initialized
info: Collection delayed.
185. <WRN> 2831.182667 (user): Hello World
186. <DBG> 2858.116395 test-arm.elf(1224): Start of test
187. <DBG> 2858.116403 test-arm.elf(1224): New iteration
188. <DBG> 2858.116407 test-arm.elf(1224): Test message 0
189. <DBG> 2858.116409 test-arm.elf(1224): Test message
190. <DBG> 2858.116410 test-arm.elf(1224): 1st half; 2nd half 0
191. <DBG> 2858.116412 test-arm.elf(1224): New iteration
192. <DBG> 2858.116414 test-arm.elf(1224): Test message 1
193. <DBG> 2858.116415 test-arm.elf(1224): Test message
194. <DBG> 2858.116416 test-arm.elf(1224): 1st half; 2nd half 1
195. <DBG> 2858.116417 test-arm.elf(1224): New iteration
196. <DBG> 2858.116418 test-arm.elf(1224): Test message 2
197. <DBG> 2858.116419 test-arm.elf(1224): Test message
198. <DBG> 2858.116420 test-arm.elf(1224): 1st half; 2nd half 2
199. <DBG> 2858.116421 test-arm.elf(1224): New iteration
200. <DBG> 2858.116423 test-arm.elf(1224): Test message 3
201. <DBG> 2858.116423 test-arm.elf(1224): Test message
202. <DBG> 2858.116424 test-arm.elf(1224): 1st half; 2nd half 3
203. <DBG> 2858.116426 test-arm.elf(1224): New iteration
204. <DBG> 2858.116427 test-arm.elf(1224): Test message 4
205. <DBG> 2858.116428 test-arm.elf(1224): Test message
206. <DBG> 2858.116428 test-arm.elf(1224): 1st half; 2nd half 4
207. <DBG> 2858.116430 test-arm.elf(1224): End of test
info: Collection delayed.
```

Figure 15. Debug Print view - application messages

8. To see the real time functionality of the **Debug Print** view, add some more messages to the view, both from kernel and the test application from the same console where the test application was running on the target:

```
# echo Hello World > /dev/kmsg

# ./test-arm

# echo Hellooooo > /dev/kmsg
```

```
root@ls1021aqds:~# LD_PRELOAD=~/libls.linux.debugprint.lib.so.1.0 ./test-arm
start up time: 10136.052177840
current time: 10136.052418800
execution took time: 0.000240960
root@ls1021aqds:~# echo Hello World > /dev/kmsg
root@ls1021aqds:~# LD_PRELOAD=~/libls.linux.debugprint.lib.so.1.0 ./test-arm
start up time: 10632.167966720
current time: 10632.168194400
execution took time: 0.000227680
root@ls1021aqds:~# echo Hellooooo > /dev/kmsg
root@ls1021aqds:~#
```

9. See the new messages displayed in the **Debug Print** text area as you enter them in the target shell.

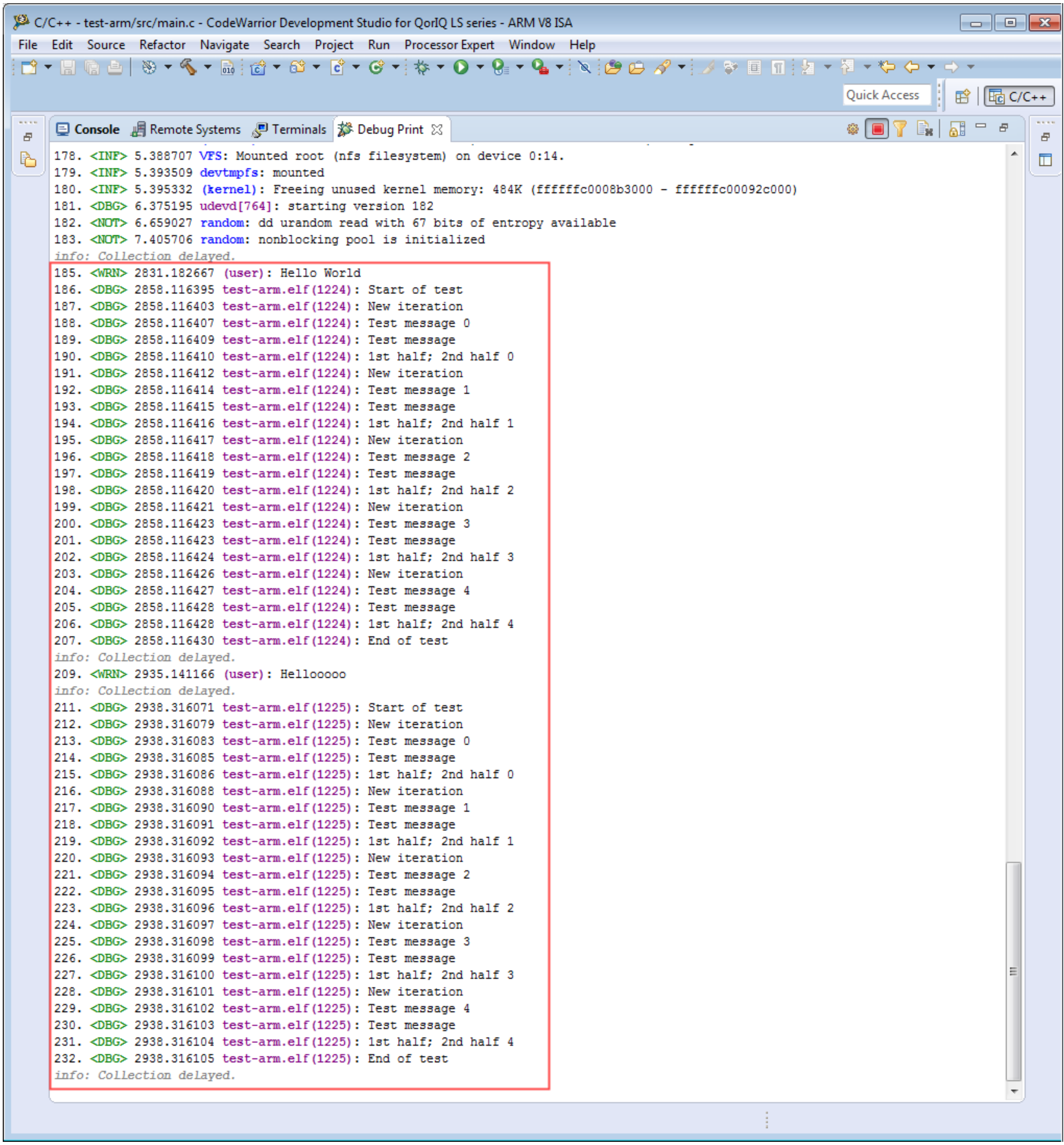


Figure 16. Debug Print view - messages from server

10. Click the **Create Debug Print Filters** button to filter the messages displayed in the **Debug Print** view. The **Create Debug Print Filters** dialog appears.
11. To filter messages from an existing module, such as `test-arm.elf`:
    a. Deselect the **PID** checkbox.
    b. Select `test.arm` in the **Existing** group.
    c. Click **Add Filter**.

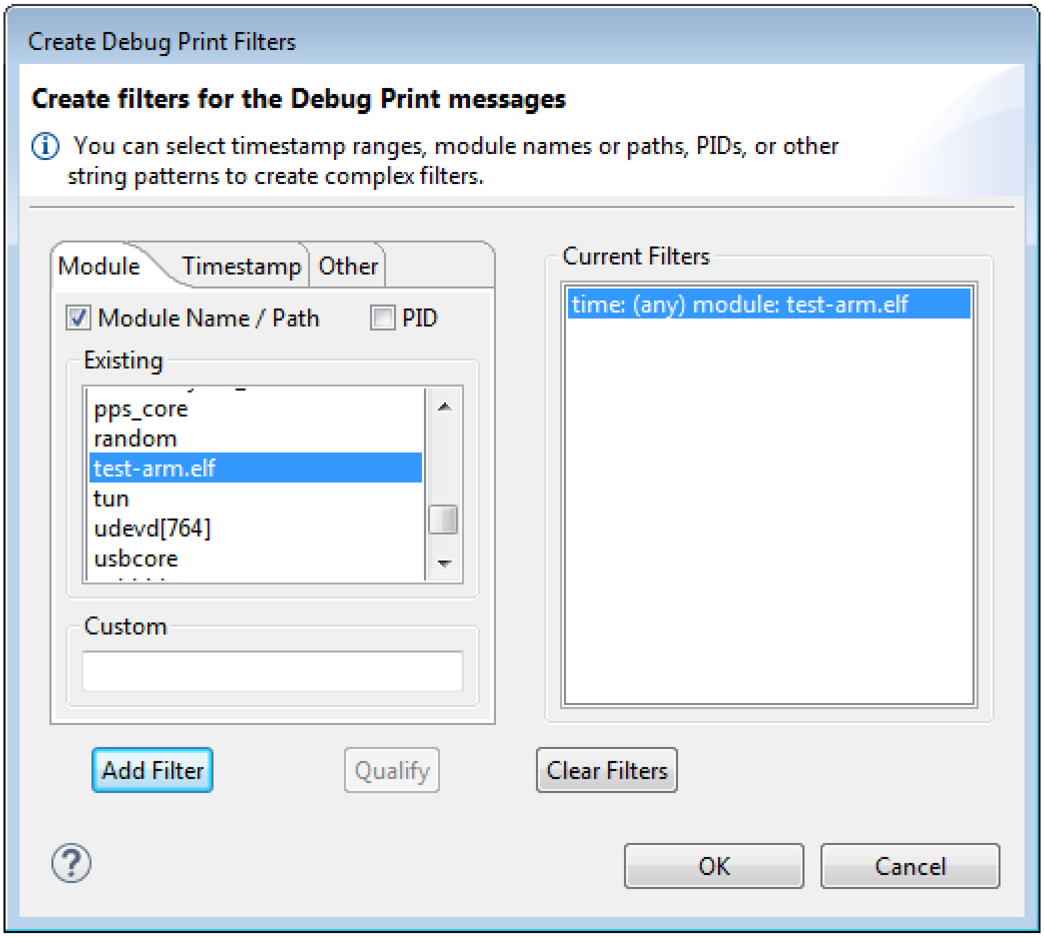


Figure 17. Create Debug Print Filters dialog

d. Click **OK** and see the new content of the view. The following figure shows the messages displayed in the **Debug Print** view using the `test-arm.elf` filter.

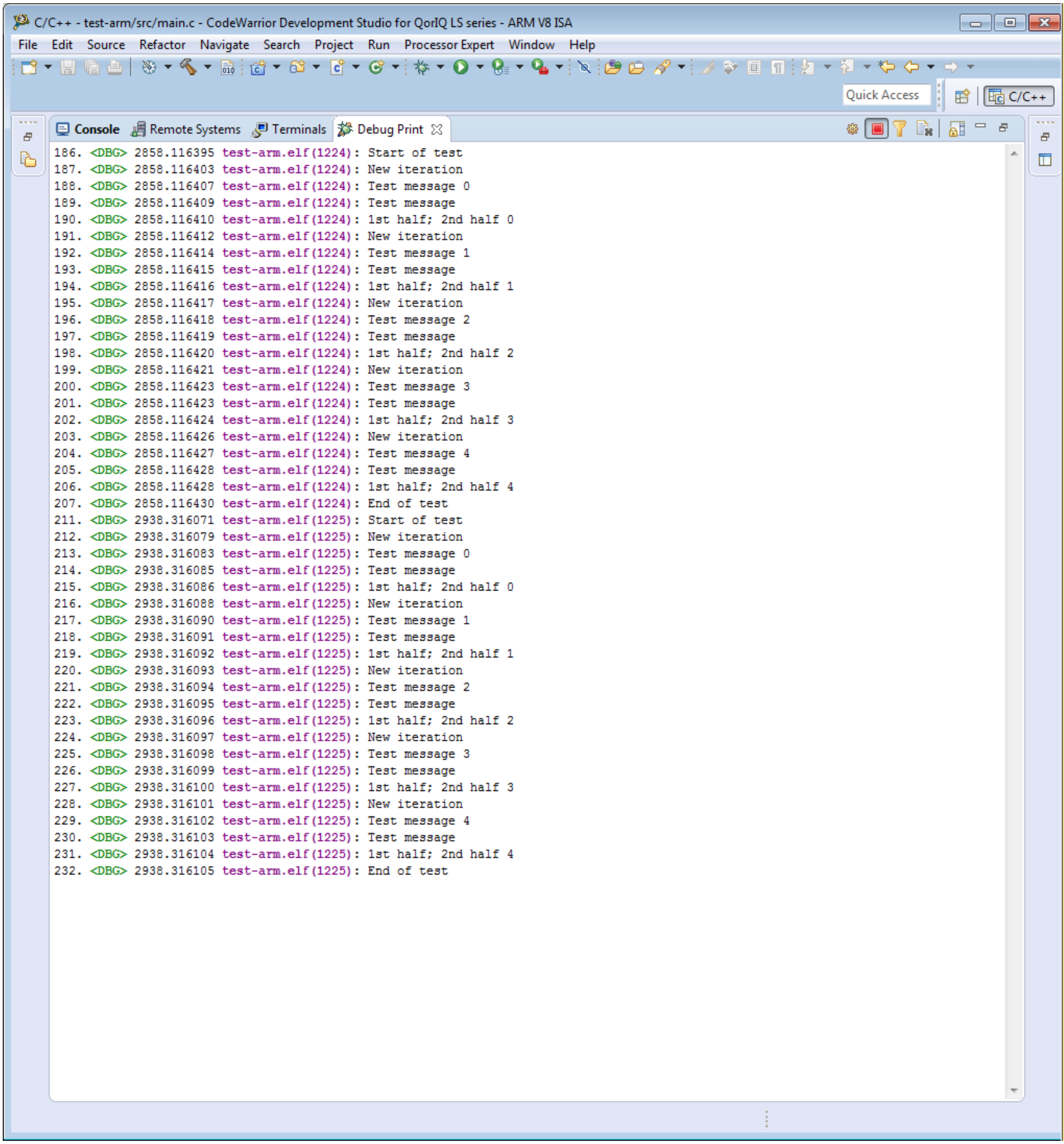


Figure 18. Debug Print view displays messages with Module filter

12. To filter all messages containing the string pattern `Hello`:
    a. Click the **Create Debug Print Filters** button in the **Debug Print** view..
    b. Click the **Clear Filters** button.
    c. Click the **Other** tab.
    d. Enter the string, `Hello`, based on which you want to filter the messages in the **Messages containing string** text box.
    e. Click the **Qualify** button.

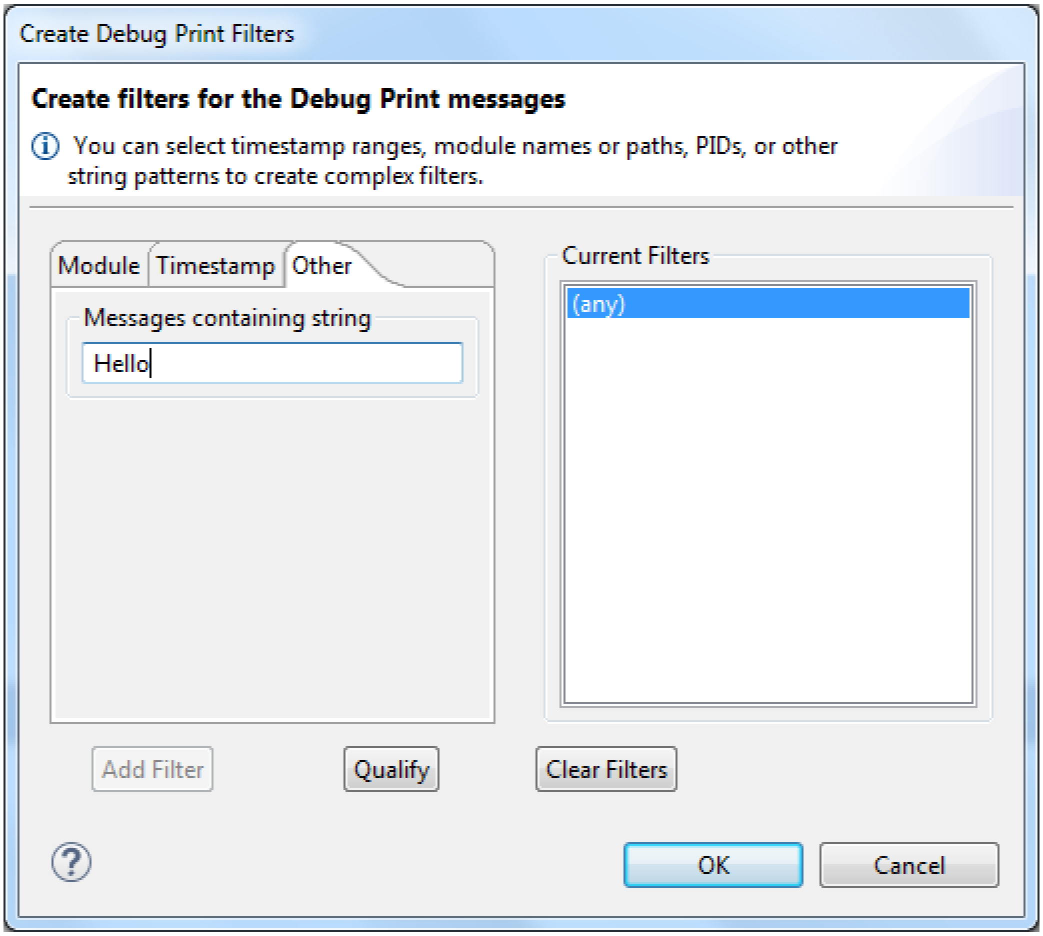


Figure 19. Create Debug Print Filters dialog - Other tab

a. Click **OK**. The following figure shows the messages displayed in the **Debug Print** view using the `Hello` string.

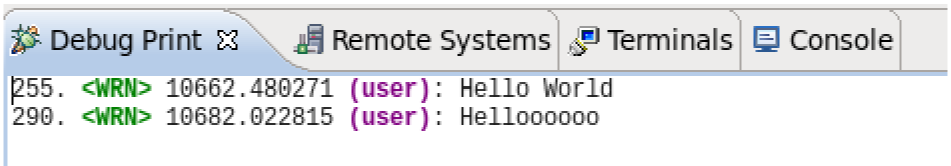


Figure 20. Debug Print view displays messages with string filter in Other tab

4.2 ARMv8 dynamic debug example

Dynamic debug lets you customize the kernel log activity when you insert/call a kernel module or anything using a `printk` call.

If the kernel is built with the dynamic debug support, you can enable various log messages for kernel modules and monitor them. See https://www.kernel.org/doc/html/v4.11/admin-guide/dynamic-debug-howto.html for information about enabling dynamic debug and **Dynamic debug demo script** for details about Debug Print with dynamic debug.

Execute the following command to display the kernel log messages.

```
# . ./generate_kmsg.sh
```

The output is displayed in the SSH console.

```
root@ls1021aqds:~# . ./generate_kmsg.sh
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.108 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.108/0.108/0.108/0.000 ms
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.107 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.107/0.107/0.107/0.000 ms
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.114 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.114/0.114/0.114/0.000 ms
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.097 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.097/0.097/0.097/0.000 ms
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.099 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.099/0.099/0.099/0.000 ms
start up time: 12138.823756640
current time: 12138.823973040
execution took time: 0.000216400
root@ls1021aqds:~#
```

Figure 21. SSH console view

The following output is displayed in the Debug Print view.

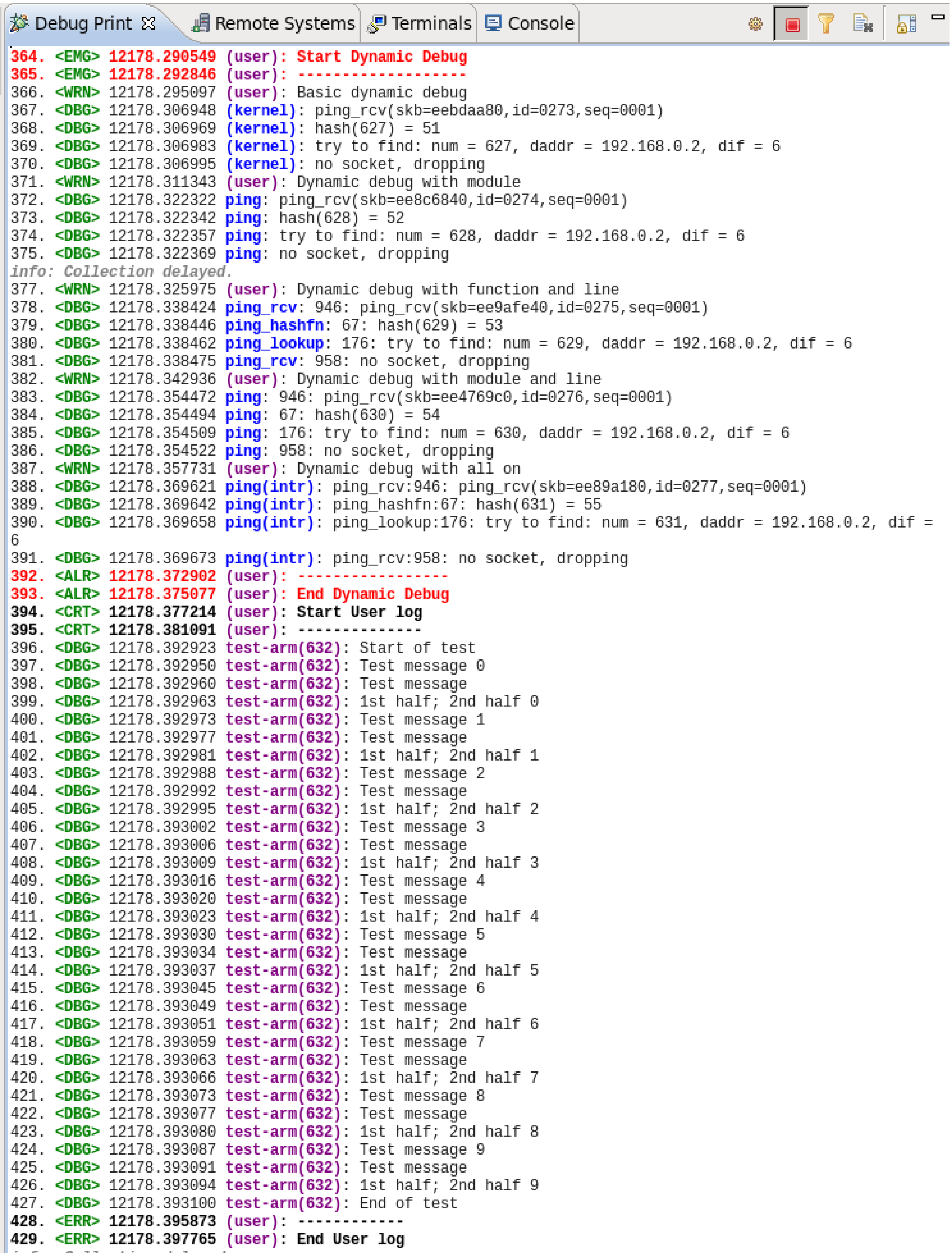


Figure 22. Debug Print view

5 Test application

Here is the test application used in this application note.

Listing 1. test-arm.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <limits.h>

/**
 * print human readable time:
 * "ddd yyyy-mm-dd hh:mm:ss:nanoseconds"
 */
void print_time(struct timespec timestamp)
{
    time_t now = timestamp.tv_sec;
    struct tm ts;
    char buf[100];
    // Format time, "ddd yyyy-mm-dd hh:mm:ss"
    ts = *localtime(&now);
    strftime(buf, sizeof(buf), "%a %Y-%m-%d %H:%M:%S", &ts);
    printf("%s:%09ld\n", buf, timestamp.tv_nsec);
}

static struct timespec MINUS = {-1, -1};

/**
 * @return t1 - t2
 */
struct timespec dif_time(struct timespec t1, struct timespec t2)
{
    if ((t1.tv_sec < t2.tv_sec) || (t1.tv_sec == t2.tv_sec && t1.tv_nsec < t2.tv_nsec))
        return MINUS;

    struct timespec res;
    res.tv_sec = t1.tv_sec - t2.tv_sec;
    if (t1.tv_nsec > t2.tv_nsec) {
        res.tv_nsec = t1.tv_nsec - t2.tv_nsec;
    } else {
        res.tv_sec--;
        res.tv_nsec = 1000000000L - t2.tv_nsec + t1.tv_nsec;
    }

    return res;
}

int main(int argc, char **argv)
{
    /* get monotonic boot time */
    struct timespec up_time, crt_time;
    int i;

    clock_gettime(CLOCK_MONOTONIC, &up_time);

    /* code goes here */

#ifdef INFINITE
    for (;;) {
#endif
        int ret = puts("Start of test");
        for (i = 0; i < 10; i++) {
```

```
            {
                char* str = "New iteration\n";
                write(STDOUT_FILENO, str, strlen(str));
            }
            fprintf(stdout, "Test message %d\n", i);
            {
                char* str = "Test message\n1st half; ";
                fwrite(str, strlen(str), 1, stdout);
            }
            printf("2nd half %d\n", i);
        }

        ret = puts("End of test\n");
#ifdef INFINITE
    }
#endif

    /* end code */

    clock_gettime(CLOCK_MONOTONIC, &crt_time);
    fprintf(stderr, "start up time: %ld.%09ld\n", up_time.tv_sec, up_time.tv_nsec);
    fprintf(stderr, "current time: %ld.%09ld\n", crt_time.tv_sec, crt_time.tv_nsec);
    crt_time = dif_time(crt_time, up_time);
    fprintf(stderr, "execution took time: %ld.%09ld\n", crt_time.tv_sec, crt_time.tv_nsec);

    exit(0);
}
```

Listing 2. Makefile

```
export PATH = [PATH_TO_BUILD_TOOLS]/gcc-linaro-arm-linux-gnueabi-4.9-2015.03_linux/bin:$$
{PATH}

CC                = arm-linux-gnueabi-gcc
CPP               = arm-linux-gnueabi-g++

CFLAGS            = -g -DDEBUG -D_DEBUG -DUNICODE -D_UNICODE
LDFLAGS          ?= -L"."

LDLIBS            = -ldl -lrt

BIN_DIR          ?= bin/

SOURCES           = arm-test.c

EXE               = $(BIN_DIR)/test-arm

EXE_INFINITE      = $(BIN_DIR)/test-arm-infinite

.PHONY: clean bindir

all:  $(EXE) $(EXE_INFINITE)

bindir:
    ((mkdir -p  $(BIN_DIR)) &> /dev/null) || true

$(EXE): bindir $(SOURCES)
    $(CC) -o "$@" $(SOURCES) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -rdynamic

$(EXE_INFINITE): bindir $(SOURCES)
    $(CC) -o "$@" $(SOURCES) -DINFINITE $(CFLAGS) $(LDFLAGS) $(LDLIBS) -rdynamic

clean:
    (rm -f *.o) || true
    (rm -f $(EXE) $(EXE_INFINITE)) || true
```

6 Dynamic debug demo script

The following script can be used to demonstrate the Debug Print feature with dynamic debug.

`192.168.0.1` is the host IP, and `/debugfs` is a link to `/sys/kernel/debug`

Listing 3. generate_kmsg.sh

```
echo "<0>Start Dynamic Debug" > /dev/kmsg
echo "<0>-------------------" > /dev/kmsg

echo "Basic dynamic debug" > /dev/kmsg
echo -n 'module ping =p' > /debugfs/dynamic_debug/control
ping -c 1 192.168.0.1

echo "Dynamic debug with module" > /dev/kmsg
echo -n 'module ping =pm' > /debugfs/dynamic_debug/control
ping -c 1 192.168.0.1

echo "Dynamic debug with function and line" > /dev/kmsg
echo -n 'module ping =pfl' > /debugfs/dynamic_debug/control
ping -c 1 192.168.0.1

echo "Dynamic debug with module and line" > /dev/kmsg
echo -n 'module ping =pml' > /debugfs/dynamic_debug/control
ping -c 1 192.168.0.1

echo "Dynamic debug with all on" > /dev/kmsg
echo -n 'module ping +pmltf' > /debugfs/dynamic_debug/control
ping -c 1 192.168.0.1

echo "<1>-----------------" > /dev/kmsg
echo "<1>End Dynamic Debug" > /dev/kmsg

echo "<2>Start User log" > /dev/kmsg
echo "<2>--------------" > /dev/kmsg

LD_PRELOAD=~/libls.linux.debugprint.libd.so.1.0 ~/test-arm

echo "<3>------------" > /dev/kmsg
echo "<3>End User log" > /dev/kmsg
```

Document Number AN5128
Revision 11.3.2, 08/2018