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.
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:

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

or

```bash
# 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.

![Debug Print view](image)

**Figure 1. Debug Print view**

The table below describes the icons available in the Debug Print view.
Table 1. Debug Print view icons

<table>
<thead>
<tr>
<th>Icons</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear All</td>
<td>Removes all text from the view.</td>
</tr>
<tr>
<td>Start/Stop</td>
<td>Two-state button used for starting and stopping the Debug Print probe.</td>
</tr>
<tr>
<td>Scroll Lock/Unlock</td>
<td>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.</td>
</tr>
<tr>
<td>Configure</td>
<td>Opens a dialog for entering the server address and port.</td>
</tr>
<tr>
<td>Create Debug Print Filters</td>
<td>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).</td>
</tr>
</tbody>
</table>

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).

![Configure Debug Print dialog](image)

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.
Table 2. Debug Print settings

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

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).

![Create Debug Print Filters dialog - Module tab](image)

**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 chooses 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.
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.
4. Click Next.
The Remote Linux System Connection page appears.
5. Specify the **Host name** and the **Connection name** and click **Next**. The **Files** page will appear.
6. Select the **ssh.files** checkbox and click Next.
   The **Processes** page appears.
7. Select the `processes.shell.linux` checkbox and click **Next**. The **Shells** page appears.
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.
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.
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](image)

**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](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.
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:

```bash
# echo Hello World > /dev/kmsg
# ./test-arm
# echo Helloworld > /dev/kmsg
```

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.
Functional examples

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.
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.
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.
### Functional examples

<table>
<thead>
<tr>
<th>Line</th>
<th>Timestamp</th>
<th>Process</th>
<th>Event Type</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>351.</td>
<td>12178.299549 [user]:</td>
<td>Start Dynamic Debug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>366.</td>
<td>12178.202846 [user]:</td>
<td>Dynamic debug with module</td>
<td></td>
<td></td>
</tr>
<tr>
<td>367.</td>
<td>12178.306848 [kernel]:</td>
<td>ping_rcv(skb=ee8c8640, id=0273, seq=0001)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>368.</td>
<td>12178.306969 [kernel]:</td>
<td>hash(627) = 51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>369.</td>
<td>12178.306993 [kernel]:</td>
<td>try to find: num = 627, daddr = 192.168.0.2, dif = 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>370.</td>
<td>12178.306985 [kernel]:</td>
<td>no socket, dropping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371.</td>
<td>12178.311343 [user]:</td>
<td>Dynamic debug with module and line</td>
<td></td>
<td></td>
</tr>
<tr>
<td>372.</td>
<td>12178.322322 ping:</td>
<td>ping_rcv(skb=ee8c8640, id=0273, seq=0001)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>373.</td>
<td>12178.322342 ping:</td>
<td>hash(628) = 52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>374.</td>
<td>12178.322557 ping:</td>
<td>try to find: num = 628, daddr = 192.168.0.2, dif = 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>375.</td>
<td>12178.322569 ping:</td>
<td>no socket, dropping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>376.</td>
<td>12178.325975 [user]:</td>
<td>Dynamic debug with function and line</td>
<td></td>
<td></td>
</tr>
<tr>
<td>378.</td>
<td>12178.338424 ping_rcv:</td>
<td>046: ping_rcv(skb=ee8c8640, id=0273, seq=0001)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>379.</td>
<td>12178.338446 ping_hashfn:</td>
<td>67: hash(629) = 53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>380.</td>
<td>12178.338462 ping_lookup:</td>
<td>176: try to find: num = 629, daddr = 192.168.0.2, dif = 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>381.</td>
<td>12178.338475 ping_rcv:</td>
<td>958: no socket, dropping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>382.</td>
<td>12178.342956 [user]:</td>
<td>Dynamic debug with module and line</td>
<td></td>
<td></td>
</tr>
<tr>
<td>383.</td>
<td>12178.345472 ping:</td>
<td>946: ping_rcv(skb=ee4793c6, id=0276, seq=0001)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>384.</td>
<td>12178.354454 ping:</td>
<td>67: hash(630) = 54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>385.</td>
<td>12178.355459 ping:</td>
<td>176: try to find: num = 630, daddr = 192.168.0.2, dif = 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386.</td>
<td>12178.355459 ping:</td>
<td>958: no socket, dropping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>387.</td>
<td>12178.357783 [user]:</td>
<td>Dynamic debug with all on</td>
<td></td>
<td></td>
</tr>
<tr>
<td>388.</td>
<td>12178.369621 ping(int):</td>
<td>ping_rcv:946: ping_rcv(skb=ee89a188, id=0277, seq=0001)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>391.</td>
<td>12178.369673 ping(int):</td>
<td>ping_rcv:958: no socket, dropping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>392.</td>
<td>12178.372962 [user]:</td>
<td>Start User log</td>
<td></td>
<td></td>
</tr>
<tr>
<td>393.</td>
<td>12178.375977 [user]:</td>
<td>End Dynamic Debug</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 22. Debug Print view*
5 Test application

Here is the test application used in this application note.

Listing 1. test-arm.c

```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
", 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_nsec -=
    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++) {
```
Test application

```c
{    char* str = "New iteration\n";
    write(STDOUT_FILENO, str, strlen(str));
}
if(0) /* 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 -D_UNICODE -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

```bash
# /debugfs/dynamic_debug/control to define the dynamic debug scope
# ping -c 1 192.168.0.1 to trigger the debug

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

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

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

```
echo "<1>------------------" > /dev/kmsg
```

```
echo "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
```