# lib_logging: Debug Printing

Publication Date: 2025/8/18
Document Number: XM-006383-UG v3.4.0

XMOS

IN THIS DOCUMENT

# 1 Introduction

This library provides a lightweight printf function that can be enabled or disabled via configuration defines. Code can be declared to be within a "debug unit" (usually a library or application source base) and prints can be enabled/disabled per debug unit.

`lib_logging` is intended to be used with the XCommon CMake , the *XMOS* application build and dependency management system.

# 2 API

To use this module, include `lib_logging` in the application's `APP_DEPENDENT_MODULES` list and include the `debug_print.h` header file.

void **`debug_printf`**(char fmt[], ...)

A limited functionality version of printf that is low memory.

This function works like C-standard printf except that it only accepts d, x, s, u and c format specifiers with no conversions.

The p format specifier is treated the same as a x.

The capital version of each format specifier performs the same as the lower case equivalent.

Any alignment or padding characters are simply ignored.

The function uses the functions from `print.h` to do the underlying printing.

Unlike printf this function has no return value.

Whether the function does any output can be controlled via defines such as `DEBUG_PRINT_ENABLE` or `DEBUG_PRINT_ENABLE_[debug unit name]` in the application's `debug_conf.h`

# 3 Debug units

A source file can be added to a debug unit by defining the `DEBUG_UNIT` macro before inclusion of `debug_print.h`. For example,

```
#define DEBUG_UNIT ETHERNET_MODULE
#include "debug_print.h"
```

To include all source files in a module in a particular debug unit, it is convenient to do it in the `lib_build_info.cmake` file of the module e.g.

```
set(LIB_COMPILER_FLAGS ... -DDEBUG_UNIT=ETHERNET_MODULE ...)
```

If no `DEBUG_UNIT` is defined then the default debug unit is `APPLICATION`.

# 4 Enabling printing

By default, debug printing is turned *off*. To enable printing you need to pass the correct command line option to compilation. The following defines can be set by using the `-D` option to the compiler. For example, the following in your application `CMakeLists.txt` will enable debug printing

```
set(APP_COMPILER_FLAGS ... -DDEBUG_PRINT_ENABLE=1 ...)
```

The following defines can be set:

**DEBUG_PRINT_ENABLE**

Setting this define to 1 or 0 will control whether debug prints are output.

**DEBUG_PRINT_ENABLE_[debug unit]**
Enabling this define will cause printing to be enabled for a specific debug unit. If set to 1, this will override the default set by DEBUG_PRINT_ENABLE.

**DEBUG_PRINT_DISABLE_[debug unit]**
Enabling this define will cause printing to be disabled for a specific debug unit. If set to 1, this will override the default set by DEBUG_PRINT_ENABLE.

# 5 Example

This included example shows how to use the logging library. It covers the difference between the logging library (`debug_printf()`) and the system library printing function (`printf()`).

It also covers the difference between JTAG and xSCOPE to perform the I/O to the host, including approximate values for resource usage and performance of each approach.

# 6 Example logging library usage

## 6.1 The CMakeLists.txt file

To start using the XMOS logging library, you need to add `lib_logging` to the dependent module list in the `CMakeLists.txt` file

```
set(APP_DEPENDENT_MODULES "lib_logging")
```

The dependencies for this example are specified by `deps.cmake` in the `examples` directory and are included in the application `CMakeLists.txt` file.

Also, `debug_printf()` calls are only active if you enable them in your `CMakeLists.txt` file. This is done by by setting `DEBUG_PRINT_ENABLE` to 1 in the `APP_COMPILER_FLAGS`.

```
set(APP_COMPILER_FLAGS ... -DDEBUG_PRINT_ENABLE=1 ...)
```

## 6.2 Include

The function prototypes are declared in a single header file which must be included from your source file.

```
#include "debug_print.h"
```

## 6.3 Example application output

The example application outputs “Hello world”.

```
int main() {

    debug_printf("Hello world\n");
```

## 6.4 Building and Running the application using the command line

First open a command prompt/terminal window with the tools environment setup. A setup batch/script file is provided in the XTC package to do this for you.

To configure the build run the following from an XTC command prompt.

```
cd examples
cd app_debug_unit
cmake -B build -G "Unix Makefiles"
```

Finally, the application binaries can be built using `xmake`.

```
xmake -C build
```

Running the application is then done using the command.

```
xrun --xscope bin/app_debug_unit.xe
```

## 6.5 Debug units by example

Applications can be created with different *units* whose debug output is independently controlled. The example application also calls a function in another unit:

```
void unit_function();
```

That file has put its debug messages as a separate debug unit by doing:

```
#define DEBUG_UNIT unit
#include "debug_print.h"
```

And by default these debug messages are not enabled, so running the program will only produce the following output.

```
$ xrun --xscope bin/app_debug_unit.xe
Hello world
```

In order to enable the debug_print messages in `unit.xc` it is necessary to add to the list of compiler flags in the `CMakeLists.txt` file.

```
set(APP_COMPILER_FLAGS ... -DDEBUG_PRINT_ENABLE_unit=1 ...)
```

After rebuilding the application it will now produce.

```
$ xrun --xscope bin/app_debug_unit.xe
Hello world
Unit print
```

## 6.6 xSCOPE printing

On the xCORE platform it is possible to have I/O messages sent to the console using either JTAG or xSCOPE. The default is for JTAG to be used. However, when doing I/O over JTAG all cores on the xCORE are stopped and hence real-time functionality is no longer maintained. As a result xSCOPE I/O should be preferred.

xSCOPE I/O is enabled by creating a `config.xscope` file. This file can be created in the same folder as the `CMakeLists.txt` or with the source files. When `config.xscope` exists, it controls whether I/O messages are enabled, and whether they use xSCOPE or JTAG. When xSCOPE is enabled it uses a link to communicate with the xTAG. The link is specified in the target's XN file. A basic file which enables I/O over xSCOPE contains:

```
<xSCOPEconfig ioMode="basic" enabled="true">
</xSCOPEconfig>
```

# 7 Notes

## 7.1 Resource usage

The following table shows the memory and cycle requirements for doing a simple print of "Hello world %d\n", using either `printf()` or `debug_printf()`, and using either JTAG or xSCOPE as the transport to the host.

Table 1: Resource usage

| Function | Transport | Program Memory (kB) | Time (us) | Channel Ends |
|---|---|---|---|---|
| None | N/A | 0.9 | 0.0 | 0 |
| debug_printf() | JTAG | 1.86 | 72000 | 0 |
| debug_printf() | xSCOPE | 2.88 | 8.5 | 1 per tile |
| printf() | JTAG | 9.02 | 72000 | 0 |
| printf() | xSCOPE | 9.99 | 18.6 | 1 per tile |

**Note**

The JTAG timings are approximate and will depend on a number of factors including the host machine being used.

## 7.2 Lossless vs lossy

The advantage of using xSCOPE instead of JTAG should be clear from the performance figures above. However, it is important to understand that xSCOPE is a lossy transport and as such if too much I/O is created then messages can be lost.

## 7.3 Ordering

It is also essential to understand that messages produced by different logical cores are interleaved on the host console. There is no guarantee of the order in which they will be printed.

## 7.4 print.h

The tools provide extremely lightweight printing functions in `print.h`. For example it is possible to do.

```
#include <print.h>
...
printstr("The number ");
printint(10);
printstrln(" should be 10");
```

But each of these print fragments can arrive mixed with messages from other logical cores. Whereas,

```
debug_printf("The number %d should be 10\n", 10);
```

will be printed as a single line.

## 8 Further Reading

- XMOS XTC Tools Installation Guide
- XMOS XTC Tools User Guide
- XMOS application build and dependency management system; xcommon-cmake
- XMOS Libraries