Coprocessor Offload Infrastructure for MIC Getting Started Guide

1 About This Document

This document is designed to help developers get started using the Coprocessor Offload Infrastructure (COI). It contains detailed steps for building and running small COI applications. Detailed COI API documentation is included in the COI API Reference Manual, which is part of the MIC Platform Systems Software (MPSS) installation package.

2 Directories

By default, COI is installed in the `/opt/intel/mic/coi` directory. Under this directory, there are a number of subdirectories:

`docs`	Documentation, including this document, the COI API Reference Manual, and the release notes.
`include`	Include files needed to build COI applications.
`tutorials`	Simple code samples that can be helpful for learning how to write COI applications.
`{host|device}-linux-{debug|release}/bin`	COI executable binaries. This includes the COI daemon, and COI smoke tests. Four different versions of each of these binaries are provided, with all combinations of host or device and debug or release.
`{host|device}-linux-{debug|release}/lib`	COI shared libraries needed to build COI applications and the Google Test libraries needed to run the included smoke tests. Four different versions of each of these libraries are provided, with all combinations of host or device and debug or release.

3 COI Daemon

COI requires a daemon to run on the sink device in order to launch sink processes. This daemon is a program that runs in the background and listens for SCIF connections from other nodes. For example, in the typical offload case, the COI daemon will be running on the MIC device, and COI applications running on the host will connect to this daemon to enumerate devices and launch sink processes.

The default installation of the MIC Platform Systems Software (MPSS) stack automatically runs the COI daemon on the MIC device. To verify that it is running, type the following commands on the host console:

```
[joe@joe-dev debug]$ telnet 192.168.1.100
Trying 192.168.1.100...
```

```
Connected to 192.168.1.100.
Escape character is '^]'.

# ps |grep coi_daemon
16802 root     32336 S    /tmp/coi_daemon
16819 root      1416 S    grep coi_daemon
```

Notice that in this example, the COI daemon is running with process ID 16802.

4 Running COI Smoke Tests

The COI smoke tests are designed to provide simple verification that COI is functioning properly and that the COI installation is correct. For example, to run the debug version of COI Pipeline smoke test, follow these steps:

```
[joe@joe-dev bin]$ cd /opt/intel/mic/coi/host-linux-debug/bin
[joe@joe-dev bin]$ SINK_LD_LIBRARY_PATH=../../device-linux-debug/lib
./smoke_pipeline_source
[==========] Running 17 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 16 tests from COIPipeline
[ RUN      ] COIPipeline.CreateDestroy
[       OK ] COIPipeline.CreateDestroy (993 ms)
[ RUN      ] COIPipeline.Create_NegativeParams
[       OK ] COIPipeline.Create_NegativeParams (0 ms)
[ RUN      ] COIPipeline.Destroy_NegativeParams
[       OK ] COIPipeline.Destroy_NegativeParams (0 ms)
[ RUN      ] COIPipeline.RunFunction_NegativeParams
[       OK ] COIPipeline.RunFunction_NegativeParams (1 ms)
[ RUN      ] COIPipeline.RunFunction
[       OK ] COIPipeline.RunFunction (957 ms)
[ RUN      ] COIPipeline.VerifySinkStart
[       OK ] COIPipeline.VerifySinkStart (5736 ms)
[ RUN      ] COIPipeline.RunFunction_MiscData
[       OK ] COIPipeline.RunFunction_MiscData (963 ms)
[ RUN      ] COIPipeline.MultiplePipeline_Test_2
[       OK ] COIPipeline.MultiplePipeline_Test_2 (1008 ms)
[ RUN      ] COIPipeline.Flush_NegativeParams
[       OK ] COIPipeline.Flush_NegativeParams (0 ms)
[ RUN      ] COIPipeline.GetEngine_NegativeParams
[       OK ] COIPipeline.GetEngine_NegativeParams (1 ms)
[ RUN      ] COIPipeline.SetCPUMask_NegativeParams
[       OK ] COIPipeline.SetCPUMask_NegativeParams (965 ms)
[ RUN      ] COIPipeline.ClearCPUMask_NegativeParams
[       OK ] COIPipeline.ClearCPUMask_NegativeParams (0 ms)
[ RUN      ] COIPipeline.ClearCPUMask
[       OK ] COIPipeline.ClearCPUMask (0 ms)
[ RUN      ] COIPipeline.PipelineCreate_SetPipelineStack
[       OK ] COIPipeline.PipelineCreate_SetPipelineStack (1072 ms)
[ RUN      ] COIPipeline.PipelineCreate_SingleThreadAffinity
[       OK ] COIPipeline.PipelineCreate_SingleThreadAffinity (997 ms)
[ RUN      ] COIPipeline.PipelineCreate_MultipleThreadAffinity
[       OK ] COIPipeline.PipelineCreate_MultipleThreadAffinity (1009 ms)
[----------] 16 tests from COIPipeline (13703 ms total)

[----------] 1 test from COIPipelines
[ RUN      ] COIPipelines.MultiplePipeline_Test_1
[       OK ] COIPipelines.MultiplePipeline_Test_1 (1113 ms)
[----------] 1 test from COIPipelines (1113 ms total)

[----------] Global test environment tear-down
```

```
[==========] 17 tests from 2 test cases ran. (14816 ms total)
[  PASSED  ] 17 tests.

  YOU HAVE 5 DISABLED TESTS
```

Notice that the command line sets the `SINK_LD_LIBRARY_PATH` environment variable. This variable tells COI where to find the sink-side dynamic libraries needed to run the sink-side executable. Of course, the contents of the tests may vary from release to release, so your results may not match the above results perfectly. And it is normal to have disabled tests for things that are known not to function yet. But all of the tests should pass.

5 Building and Running Tutorials

The COI release comes with a number of simple code tutorials, including the following:

`hello_world`	Shows an application with no pipelines that uses the COI I/O proxy to print output on the source. This type of usage can be useful if you just want to run a remote application.
`coi_simple`	Shows the use of a single pipeline and run function.
`buffers_with_pipeline_function`	Shows simple buffer operations.
`xn_to_coi`	Illustrates how code written using XN (the offload API offered on older versions of the MIC systems software stack) can be ported to use COI.
`multiple_pipeline_implicit`	Shows how to use implicit buffer dependencies to coordinate between multiple pipelines.
`multiple_pipeline_explicit`	Shows how to use explicit dependencies to coordinate between multiple pipelines.
`user_barrier`	Shows how to use user-created barriers for synchronization between sink and source.
`buffer_references`	Illustrates the use of buffer reference counting to implement out-of-order asynchronous operations.

Each tutorial directory contains pre-built binaries as well as the source and makefiles needed to rebuild them. The tutorial makefiles are configured to use the GNU compiler (GCC) that is included in the MPSS release. This compiler is not ABI-compatible with the native ICC for MIC, so binaries built with GCC will not run properly when linked with binaries built with ICC.

The tutorial makefiles expose a few variables that can be used to configure how they are built. These are described in the table below.

Variable Name	Default	Description
`MIC_DIR`	`/opt/intel/mic`	The base directory of the MPSS installation.
`COI_DIR`	`$(MIC_DIR)/coi`	The base directory where COI files are installed (included in MPSS installation).
`HOST_CC`	`g++`	The compiler used to build the host binaries (included with Linux installation).
`MIC_CC_DIR`	`/usr/linux-l1om/bin`	The directory containing the MIC compiler.

MIC_CC	$(MIC_CC_DIR)/g++	The GCC compiler used to build the MIC binaries.

The tutorials can be built by copying a tutorial's entire directory into a user directory, and typing `make` like this:

```
[joe@joe-dev ~]$ cp -r /opt/intel/mic/coi/tutorial/hello_world .
[joe@joe-dev ~]$ cd hello_world
[joe@joe-dev hello_world]$ make
mkdir -p debug
g++ -L/opt/intel/mic/coi/host-linux-debug/lib -I /opt/intel/mic/coi/inclu
de -lcoi_host -g -O0 -D_DEBUG -o debug/hello_world_source_host hello_worl
d_source.cpp
mkdir -p debug
/usr/linux-l1om-4.7/bin/g++ -L/opt/intel/mic/coi/device-linux-debug/lib -
I /opt/intel/mic/coi/include -lcoi_device -march=lrb -Wa,-mtune=l1om -rdy
namic -Wl,--enable-new-dtags -Wl,-rpath=/lib64:/lib -g -O0 -D_DEBUG -o de
bug/hello_world_sink_mic hello_world_sink.cpp
mkdir -p release
g++ -L/opt/intel/mic/coi/host-linux-release/lib -I /opt/intel/mic/coi/inc
lude -lcoi_host -DNDEBUG -O3 -o release/hello_world_source_host hello_wor
ld_source.cpp
mkdir -p release
/usr/linux-l1om-4.7/bin/g++ -L/opt/intel/mic/coi/device-linux-release/lib
-I /opt/intel/mic/coi/include -lcoi_device -march=lrb -Wa,-mtune=l1om -r
dynamic -Wl,--enable-new-dtags -Wl,-rpath=/lib64:/lib -DNDEBUG -O3 -o re
lease/hello_world_sink_mic hello_world_sink.cpp
[joe@joe-dev hello_world]$
```

After building the tutorial, it can be executed by running the host executable, like this:

```
[joe@joe-dev hello_world]$ cd debug
[joe@joe-dev debug]$ LD_LIBRARY_PATH=/opt/intel/mic/coi/host-linux-deb
ug/lib SINK_LD_LIBRARY_PATH=/opt/intel/mic/coi/device-linux-deb
ug/lib ./hello_world_source_host
1 engines available
Hello from the sink!
Press enter to kill the sink process.
[joe@joe-dev hello_world]$
```

Notice that running the tutorial requires an environment variable to be set. The `LD_LIBRARY_PATH` variable points to the directory where the COI host library is stored. The `SINK_LD_LIBRARY_PATH` variable is set to where the COI device library is stored.

6 Troubleshooting

As with any development system, sometimes things don't work as designed. Here are some techniques that can be done to fix or mitigate problems that may arise. If for some reason following these steps doesn't resolve the problem, or if some of these steps need to be done consistently, please file a defect in HSD. The URL for the database is:

```
https://vthsd.intel.com/hsd/vpg_mic_sw/default.aspx
```

6.1 `COIProcessCreate` Hangs

If a COIProcessCreate call hangs, there could be a number of culprits. First, check to see if the Linux uOS on the MIC device is still active. You can do this by attempting to telnet to the device:

```
[joe@joe-dev debug]$ telnet 192.168.1.100
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.

#
```

If this works properly, use the telnet session to validate that the COI daemon is still running:

```
# ps |grep coi_daemon
16802 root     32336 S    /tmp/coi_daemon
16819 root      1416 S    grep coi_daemon
```

If the COI daemon is still running, try to restart it:

```
# killall coi_daemon
[1]+  Terminated                 /tmp/coi_daemon
# chmod +x /tmp/coi_daemon
# /tmp/coi_daemon &
```

6.2 A COI API Returns an Error Code

Sometimes, having an accurate error code doesn't necessarily make a problem clear. For example, if `COIProcessCreate` returns `COI_MISSING_DEPENDENCY`, this indicates that a dynamic library needed by the executable could not be found in the source or sink file systems. If the debug version of the COI library is used, however, there is a possibility that more information can be learned by looking at the automatically-produced log file. This file is named `<executable>.coilog`, where `<executable>` is the name of the source executable. It is located in the current directory in effect when the application was launched.

7 Disclaimer

8 Legal Notices