Coprocessor Offload Infrastructure for Intel® Many Integrated Core Architecture 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 Intel® Xeon Phi™ (Phi*) Software installation package.

*Phi is a product family based on the Intel® Many Integrated Core (Intel® MIC) Architecture.

2 Directories

By default, COI is installed in multiple locations. These locations include:

`/usr/share/doc/intel-coi-<version>`	Documentation, including this document, the COI API Reference Manual, and the release notes.
`/usr/include`	Include files needed to build COI applications.
`/usr/share/doc/intel-coi-<version>/tutorials`	Simple code samples that can be helpful for learning how to write COI applications.
`/usr/bin`	COI tools to assist in development.
/usr/lib64	*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 Intel MIC Architecture 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 Phi Software stack automatically runs the COI daemon on the Intel MIC Architecture device. To verify that it is running, type the following commands on the host console (Note: the IP listed is the default, your administrator may have changed it):

```
[joe@joe-dev debug]$ ssh 172.31.1.1

# ps -a | grep coi_daemon
```

```
        4632 pts/0    00:00:00 coi_daemon
```

Notice that in this example, the COI daemon is running with process ID 4632.

4 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.
`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.
`streaming_buffer`	Shows how to use streaming buffers to setup a software processing pipeline.
`user_event`	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 make files needed to rebuild them. The tutorial make files are configured to use the GNU compiler (GCC) that is included in the Intel Xeon Phi Software release.

The tutorial make files 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**
`HOST_CC`	`g++`	The compiler used to build the host binaries (included with Linux installation).
`DEV_CC_DIR`	`/opt/mpss/?.?/sysroots/x86_64-mpsssdk-linux/usr/bin/?1om-mpss-linux`	The directory containing the Intel MIC Architecture compiler.
`DEV_CC`	`$(DEV_CC_DIR)/x86_64-?1om-linux-g++`	The GCC compiler used to build the Intel MIC Architecture 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 /usr/share/doc/intel-coi-
<version>/tutorials/hello_world .
[joe@joe-dev ~]$ cd hello_world
[joe@joe-dev hello_world]$ make
mkdir -p debug
g++ -lcoi_host -g -O0 -D_DEBUG -o debug/hello_world_source_host
hello_worl
d_source.cpp
mkdir -p debug
/opt/mpss/?.?/sysroots/x86_64-mpsssdk-linux/usr/bin/?1om-mpss-
linux)/x86_64-?1om-linux-g++ -lcoi_device -march=lrb -Wa,-mtune=k1om -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++ -lcoi_host -DNDEBUG -O3 -o release/hello_world_source_host hello_wor
ld_source.cpp
mkdir -p release
/opt/mpss/?.?/sysroots/x86_64-mpsssdk-linux/usr/bin/?1om-mpss-
linux)/x86_64-?1om-linux-g++ -lcoi_device -march=lrb -Wa,-mtune=k1om -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]$ ./hello_world_source_host
1 engines available
Hello from the sink!
Press enter to kill the sink process.
[joe@joe-dev hello_world]$
```

5 Using `coitrace` to assist with debugging

Included in the installation package is a tool called `coitrace`. This trace utility function similar to Unix-style tools like `strace` and shows all of the COI API invocations and input parameters. This can be helpful to trace what COI commands are being executed for tracing and debugging. To see a complete list of options run `coitrace -h`.

To use `coitrace` simply execute your program through `coitrace`. For example here is how the `hello_world` tutorial would execute through `coitrace`:

```
[joe@joe-dev hello_world]$ coitrace ./hello_world_source_host
COIEngineGetCount
       in_ISA = COI_ISA_MIC
       out_pNumEngines = 0x0x7fff28bfea38

1 engines available
COIEngineGetHandle
       in_ISA = COI_ISA_MIC
       in_EngineIndex = 0
       out_pEngineHandle = 0x0x7fff28bfea20

Got engine handle
COIProcessCreateFromFile
       in_Engine = 0x7f98f4370b40
       in_pBinaryName = hello_world_sink_mic
       in_Argc = 0
       in_ppArgv = 0
       (bool) in_DupEnv = false
       in_ppAdditionalEnv = 0
       (bool) in_ProxyActive = true
       in_ProxyfsRoot = (nil)
       in_BufferSpace = 0
       in_LibrarySearchPath = (nil)
       out_pProcess = 0x0x7fff28bfea28

COIProcessCreateFromMemory
```

```
        in_Engine = 0x7f98f4370b40
        in_pBinaryName = hello_world_sink_mic
        in_pBinaryBuffer = 0x7f98f4846000
        in_BinaryBufferLength = 10847
        in_Argc = 0
        in_ppArgv = 0
        (bool) in_DupEnv = false
        in_ppAdditionalEnv = 0
        (bool) in_ProxyActive = true
        in_ProxyfsRoot = (nil)
        in_BufferSpace = 0
        in_LibrarySearchPath = (nil)
        in_FileOfOrigin = hello_world_sink_mic
        in_FileOfOriginOffset = 0
        out_pProcess = 0x0x7fff28bfea28

Sink process created, press enter to destroy it.
Hello from the sink!

COIProcessDestroy
        in_Process = 0x24bf440
        in_WaitForMainTimeout = -1
        (bool) in_ForceDestroy = false
        out_pProcessReturn = 0x0x7fff28bfea3f
        out_pReason = 0x0x7fff28bfea34

Sink process returned 0
Sink exit reason SHUTDOWN OK
```

6 micnativeloadex for remote execution

The `micnativeloadex` utility included in the package can be used to remotely execute native code from a host console. This functions similarly to using ssh to start a remote process but does not require any logins, will automatically transfer dependent libraries and will redirect console IO back to the host console. Internally `micnativeloadex` uses COI so it follows the same library loading rules and requirements for the `SINK_LD_LIBRARY_PATH` environment variable.

7 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.

7.1 `COIProcessCreate` Hangs

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

```
[joe@joe-dev debug]$ ssh 172.31.1.1

#
```

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

```
# ps -a | grep coi_daemon
  4632 pts/0    00:00:00 coi_daemon
```

If the COI daemon is still running, try to restart it, replace micuser with the appropriate option of user name for your configuration:

```
# killall coi_daemon
[1]+  Terminated                 /bin/coi_daemon
# chmod +x /bin/coi_daemon
# /bin/coi_daemon --coiuser=micuser &
```

7.2 A COI API Returns an Error Code

Sometimes, having an accurate error code doesn't necessarily make a problem clear. For example, if `COIProcessCreateFromFile` 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.

8 Disclaimer

9 Legal Notices