

DXNN Runtime (DX-RT) User Manual

2026/07/28 (v3.4.1)

DEEPX.ai

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Table of contents

1. DXNN Runtime Overview 4
1.1 DEEPX SDK Architecture 4
1.2 Inference Flow of DX-RT 5
2. Installation on Linux 8
2.1 System Requirements 8
2.2 Build Environment Setup 9
2.3 Source File Structure 11
2.4 Framework Build on Linux 12
2.5 Linux Device Driver Installation 20
2.6 Python Package Installation 29
2.7 Service Registration 31
2.8 Sanity Check 32
3. Installation on Windows 37
3.1 Overview & System Requirements 37
3.2 Pre-installation Setup 39
3.3 DX-RT Build Methods 41
3.4 Runtime Setup (Service Configuration) 46
3.5 Installation Verification (Sanity Check) 48
3.6 Tool Descriptions & Integration 50
3.7 Troubleshooting 51
3.8 Related Links 52
4. Model Inference 53
4.1 Model File Format 53
4.2 Inference Workflow 53
4.3 Multiple Device Inference 58
4.4 Data Format of Device Tensor 58
4.5 Profile Application 60
4.6 RuntimeEventDispatcher 65
4.7 How To Create an Application Using DX-RT 69
4.8 (Optional) Improving Inference Throughput 70
5. Command Line Interface 76
5.1 Legacy Command Aliases 76
5.2 Parse Model 76
5.3 Run Model 77
5.4 DX-RT CLI Tool (Firmware Interface) 79

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5.5 NPU Monitoring Utility 80
5.6 Performance Benchmarking Utility 84
6. Inference API Guide 87
6.1 C++ Inference API 87
6.2 Python Inference API 92
6.3 Input Format Parsing 97
6.4 Output Format Specification 98
6.5 Edge Cases and Special Handling 99
6.6 Performance Optimization Guidelines 100
7. Multi-input Inference Guide 102
7.1 Identifying a Multi-Input Model 102
7.2 Multi-Input Inference 103
7.3 Multi-Input Batch Inference 106
7.4 Multi-Input Asynchronous Inference 108
8. Configuration and DeviceStatus Guide 110
8.1 Configuration Management 110
8.2 Device Status Monitoring 117
9. Tutorials 125
9.1 C++ Tutorials 125
9.2 Python Tutorials 158
10. API Reference 177
10.1 C++ API Reference 177
10.2 Python API Reference 200
11. Change Log 219
11.1 DX-RT v3.4.1 (July 2026) 219
11.2 DX-RT v3.4.0 (July 2026) 219
11.3 DX-RT v3.3.2 (May 2026) 221
11.4 DX-RT v3.3.1 (April 2026) 221
11.5 DX-RT v3.3.0 (April 2026) 222
11.6 DX-RT v3.2.0 (December 2025) 223
11.7 DX-RT v3.1.0 (November 2025) 224
11.8 DX-RT v3.0.0 (September 2025) 226
11.9 DX-RT v2.9.5 (May 2025) 228
11.10 DX-RT v2.8.2 (April 2025) 229

ⓒ Copyright 2026 DEEPX All Rights Reserved.

1. DXNN Runtime Overview

This chapter provides an overview of the DEEPX SDK architecture and explains each core component and its role in the AI development workflow.

1.1 DEEPX SDK Architecture

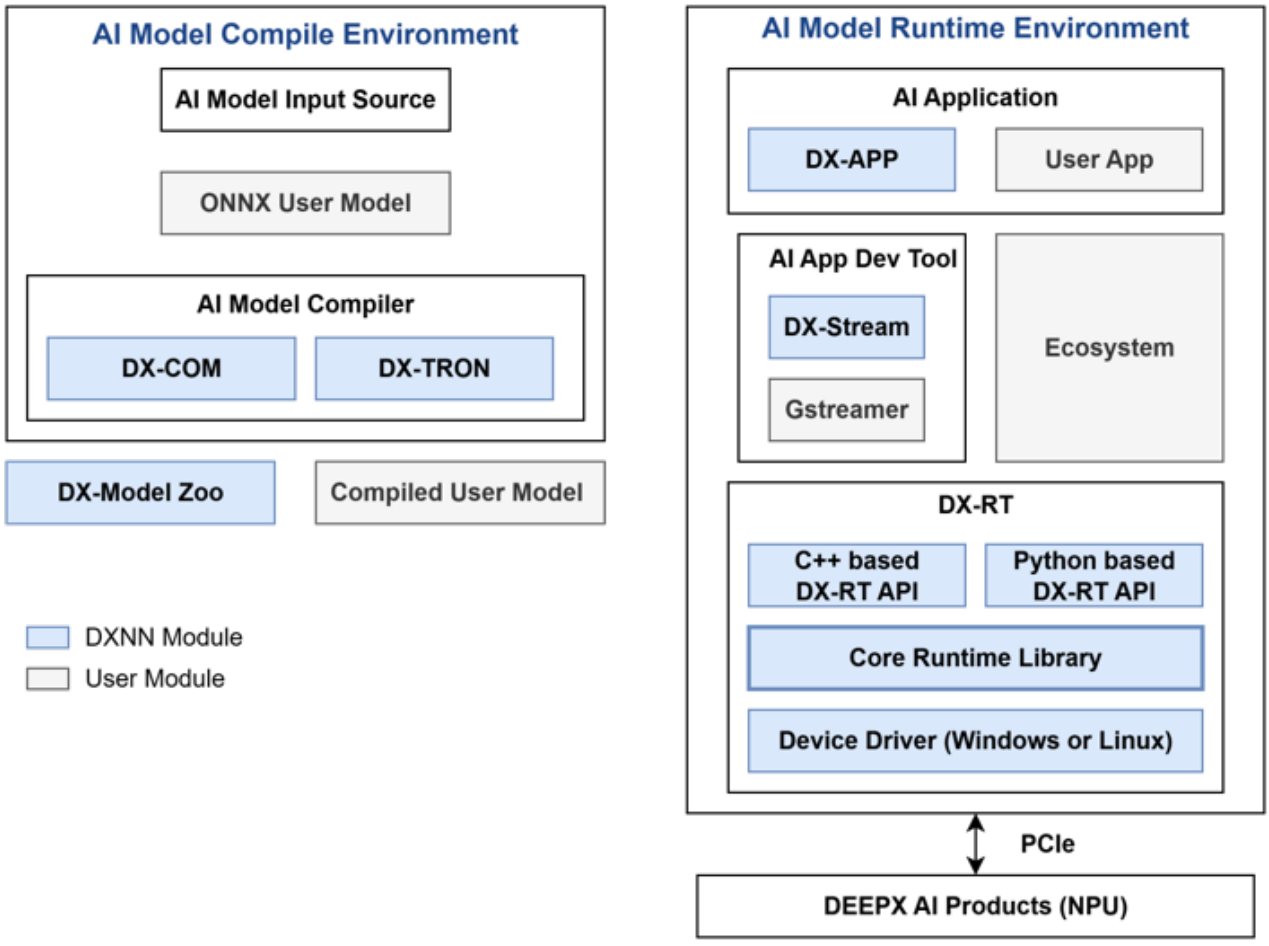

Figure. DEEPX SDK Architecture

DEEPX SDK is an all-in-one software development platform that streamlines the process of compiling, optimizing, simulating, and deploying AI inference applications on DEEPX NPUs (Neural Processing Units). It provides a complete toolchain, from AI model creation to runtime deployment, optimized for edge and embedded systems, enabling developers to build high-performance AI applications with minimal effort.

© Copyright 2026 DEEPX All Rights Reserved.

DX-COM is the compiler in the DEEPX SDK that converts a pre-trained ONNX model and its associated configuration JSON file into a hardware-optimized .dxnn binary for DEEPX NPUs. The ONNX file contains the model structure and weights, while the JSON file defines pre/post-processing settings and compilation parameters. DX-COM provides a fully compiled .dxnn file, optimized for low-latency, high-efficiency inference on DEEPX NPU.

DX-RT is the runtime software responsible for executing .dxnn models on DEEPX NPU hardware. DX-RT directly interacts with the DEEPX NPU through firmware and device drivers, using PCIe interface for high-speed data transfer between the host and the NPU, and provides C/C++ and Python APIs for application-level inference control. DX-RT offers a complete runtime environment, including model loading, I/O buffer management, inference execution, and real-time hardware monitoring.

DX ModelZoo is a curated collection of pre-trained neural network models optimized for DEEPX NPU, designed to simplify AI development for DEEPX users. It includes pre-trained ONNX models, configuration JSON files, and pre-compiled DXNN binaries, allowing developers to rapidly test and deploy applications. DX ModelZoo also provides benchmark tools for comparing the performance of quantized INT8 models on DEEPX NPU with full-precision FP32 models on CPU or GPU.

DX-STREAM is a custom GStreamer plugin that enables real-time streaming data integration into AI inference applications on DEEPX NPU. It provides a modular pipeline framework with configurable elements for preprocessing, inference, and postprocessing, tailored to vision AI work. DX-STREAM allows developers to build flexible, high-performance applications for use cases such as video analytics, smart cameras, and edge AI systems.

DX-APP is a sample application that demonstrates how to run compiled models on actual DEEPX NPU using DX-RT. It includes ready-to-use code for common vision tasks such as object detection, face recognition, and image classification. DX-APP helps developers quickly set up the runtime environment and serves as a template for building and customizing their own AI applications.

1.2 Inference Flow of DX-RT

Here is the inference flow of **DX-RT**.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

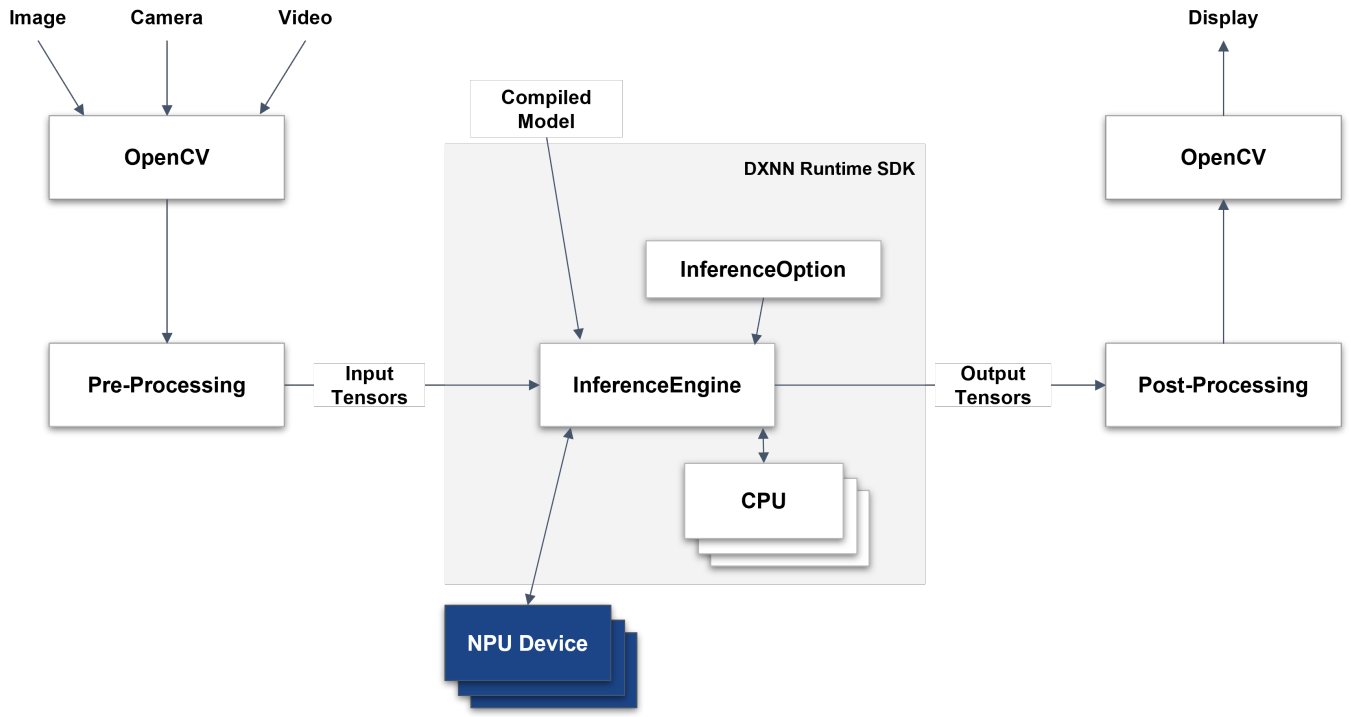

Figure. Inference Flow of DXNN Runtime

This figure illustrates the inference workflow of the DXNN Runtime SDK, which integrates OpenCV-based input/output handling with efficient NPU-accelerated model execution.

Input & Pre-Processing

Input data—such as images, camera feeds, or video—is captured using OpenCV. The data is then passed through a Pre-Processing module, which transforms it into input tensors suitable for the model.

Feeding Input to the Inference Engine

The pre-processed input tensors are fed into the InferenceEngine along with the compiled model (.dxnn). Before execution, you **must** configure the InferenceOption, which specifies the target device and available resources.

Model Execution

The InferenceEngine is the core component of the DXNN Runtime SDK. It:

- Initializes and controls the NPU device
- Manages memory for input/output tensors
- Schedules inference tasks across NPU and CPU, optimizing their interaction for real-time performance

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Post-Processing & Display

The output tensors are passed to a Post-Processing stage, typically involving OpenCV for decoding, formatting, or visualization. Finally, the results are displayed or forwarded to the next processing step.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

2. Installation on Linux

This chapter describes the system requirements, source file structure, and the installation instructions for setting up **DX-RT** on a Linux-based host system.

> **Prebuilt Package (Recommended for end-users)**
>
> If you received a prebuilt `libdxrt.so` package from DEEPX, install it with the package-provided instructions. No source build is required.
> The build/install steps in this chapter are for developers building DX-RT from source.

After you check the system requirements, follow these instructions.

- System Requirement Check
- Build Environment Setup (source build — for developers)
- Source File Structure Check
- Framework Build on Linux
- Linux Device Driver Installation
- Python Package Installation
- Service Registration
- Sanity Check

2.1 System Requirements

This section describes the hardware and software requirements for running **DX-RT** on Linux.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Hardware and Software Requirements

- **CPU:** amd64(x86_64), aarch64(arm64)
- **RAM:** 8GB RAM (16GB RAM or higher is recommended)
- **Storage:** 4GB or higher available disk space
- **OS**: Ubuntu 20.04 / 22.04 / 24.04 / 26.04 (x64) Debian 12 / Debian 13 (x64)
- **Driver:** v2.5.0 or later
- **PCIe Driver:** v2.4.0 or later
- **Firmware:** v2.7.0 or later
- **Hardware:** The system **must** support connection to an **M1 M.2** module with the **M.2 interface** on the host PC.

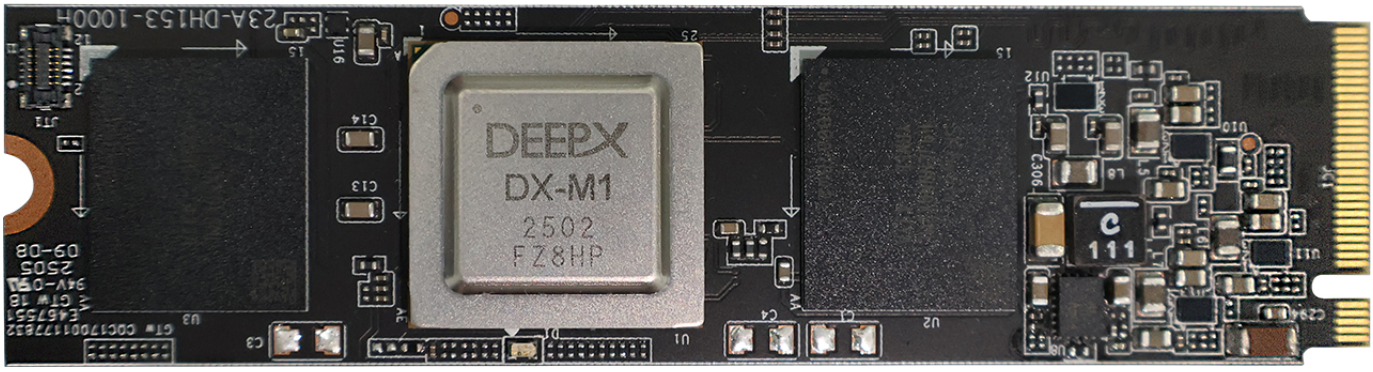

Figure. DX-M1 M.2 Module

2.2 Build Environment Setup

DEEPX provides an installation shell script to set up the **DX-RT** build environment. You can install the entire toolchain installation or perform a partial installation as necessary.

DX-RT supports the Target OS of **Ubuntu 20.04, Ubuntu 22.04, Ubuntu 24.04,** and **Ubuntu 26.04**.

Installation of DX-RT
To install the full **DX-RT** toolchain, use the following commands.

```
cd dx_rt
./install.sh --all
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Here are the available `install.sh` options.

```
Usage: ./install.sh [OPTIONS]
Install necessary components and libraries for the project.

Options:
  --help                        Display this help message and exit.
  --arch <ARCH>                 Specify the target CPU architecture. Valid options:
[x86_64, aarch64].
  --dep                         Install core dependencies such as cmake, gcc, ninja,
etc and python3.
  --onnxruntime                 (Optional) Install the ONNX Runtime library.
  --all                         Install all dependencies and the ONNX Runtime library.

  --python_version <VERSION>    Specify the Python version to install (e.g., 3.10.4).
                                  * Minimum supported version: 3.8.10.
                                  * If not specified:
                                      - For Ubuntu 20.04+, the OS default Python 3
will be used.
                                      - For Ubuntu 18.04, Python 3.8.10 will be
source-built.
  --venv_path <PATH>           Specify the path for the virtual environment.
                                  * If this option is omitted, no virtual environment
will be created.
```

Installation with ONNX Runtime

Use the ONNX Runtime option if you need to offload certain neural network (NN) operations to the CPU that are **not** supported by the NPU.

We recommend using ONNX Runtime linux x64 version v1.20.1 or higher.

```
https://github.com/microsoft/onnxruntime/releases/download/v1.20.1/onnxruntime-linux-
x64-1.20.1.tgz

sudo tar -xvzf onnxruntime-linux-x64-1.20.1.tgz -C /usr/local --strip-components=1
sudo ldconfig
```

To install the ONNX Runtime library, run the following command.

```
./install.sh --onnxruntime
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Installation for a Specific CPU Architecture

The **DX-RT** targets the **x86_64** architecture. If you're compiling for another architecture (e.g., **aarch64**), specify it using the `--arch` option.

```
./install.sh --arch aarch64 --onnxruntime
```

2.3 Source File Structure

The **DX-RT** source directory is organized as follows. You can install the full toolchain using the `install.sh`, and the build and library using `build.sh`.

```
.
├── assets
├── bin
├── cli
├── build.sh
├── build_x86_64
├── build_aarch64
├── cmake
├── docs
├── examples
├── extern
├── install.sh
├── lib
├── python_package
├── release
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
├── service
└── tool
```

- `assets` Reference images.
- `bin` Compiled binary executables.
- `cli` Command-line application source code.
- `build.sh` Shell script for building the framework.
- `build_x86_64` / `build_aarch64` These directories are where the build artifacts are stored, specifically for the **x86_64** and **aarch64** architectures, respectively.
- `cmake` CMake scripts for build configuration.
- `docs` Markdown documents.
- `examples` Provides a set of practical inference examples that demonstrate how to use the DX-RT framework.
- `extern` Third-party libraries.
- `install.sh` Shell script for toolchain installation.
- `lib` The core source code for the DX-RT libraries.
- `python_package` Python modules for DX-RT.
- `release` Build output directory for release artifacts, including generated prebuilt binary Debian packages (for example, `libdxrt-bin_<version>_amd64.deb` and `libdxrt-bin_<version>_arm64.deb`).
- `service` Service unit files for runtime management.
- `tool` Profiler result visualization tools.

2.4 Framework Build on Linux

After the **DX-RT** environment setup, you can build the framework using the provided build.sh shell script.

2.4.1 Build Overview

DX-RT supports the following target CPU architectures:

- **x86_64** for desktop/server-class hosts
- **aarch64** for embedded systems and ARM-based platforms

You can select the target architecture using the `--arch` option in the build script.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

2.4.2 Recommended Build Process (Clean option)

This chapter explains the importance of performing a clean build to maintain a stable and conflict-free build environment.
To do this, you **must** run a `clean` operation before starting any new build process.

```
# First, clean the previous build artifacts
./build.sh --clean

# Then, run the build with your desired options
./build.sh
```

You can also combine these commands. The `&&` operator ensures that the build command will only run if the clean command succeeds.

```
./build.sh --clean && ./build.sh
```

2.4.3 Build Script Options

This chapter explains the available options for the `build.sh` script, which is used to compile the **DX-RT** framework.

```
Usage: ./build.sh [OPTIONS]
Build the dx-rt project with various options.

Options:
  --help            Display this help message and exit.
  --clean           Perform a clean build, removing previous build artifacts.
  --verbose         Show detailed build commands during the process.
  --type <TYPE>     Specify the CMake build type. Valid options: [Release, Debug,
RelWithDebInfo].
  --arch <ARCH>     Specify the target CPU architecture. Valid options: [x86_64,
aarch64].
  --install <PATH>  Specify the installation path for dx-rt files.
  --uninstall       Remove previously installed dx-rt files.
  --docker          Build the project within a Docker environment.
  --clang             Use Clang as the compiler for the build.

  --python-exec <PATH> Specify the Python executable to use for the build.
                       If omitted, the default system 'python3' will be used.
  --venv_path <PATH>  Specify the path to a virtual environment to activate for the
build.
                    If omitted, no virtual environment will be activated.
```


© Copyright 2026 DEEPX All Rights Reserved.

2.4.4 Examples for Build Script Option

Example 1: Clean Build and Install (Recommended Workflow)

This example shows the safest way to build and install the library.
First, you clean the workspace, then you build for a specific architecture and install the libraries to a system path.

```
# Step 1: Clean the workspace to ensure no old files interfere
./build.sh --clean

# Step 2: Build for the aarch64 architecture and install to /usr/local
./build.sh --arch aarch64 --install /usr/local
```

Example 2: Build a Debug Version

To build the framework with debugging symbols, which is useful for development and troubleshooting.

```
# Clean is recommended here as well, especially if your last build was a Release
version
./build.sh --clean

# Build with the 'Debug' type
./build.sh --type Debug
```

Example 3: Uninstall the Library

To remove all library files installed by a previous `--install` command.

```
./build.sh --uninstall
```

2.4.5 Debian Package Installation

DX-RT is distributed as a **prebuilt binary Debian package** (`libdxrt-bin`), provided per CPU architecture:

File name example	Target architecture
`libdxrt-bin_3.4.0_amd64.deb`	amd64 (x86_64)
`libdxrt-bin_3.4.0_arm64.deb`	aarch64 (arm64)

The package contains precompiled shared libraries, headers, CLI tools, and the Python module — no on-target compilation is required.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Supported OS versions

The prebuilt binaries are compiled against **glibc ≥ 2.27** and **libstdc++ from GCC 10+**, so the package supports the following distributions and newer:

Distribution	Minimum supported version
Ubuntu	**20.04 LTS** (glibc 2.31)
Debian	**11 (bullseye)** (glibc 2.31)

Ubuntu 18.04 and older releases are not supported — they ship glibc 2.27 with an older libstdc++ ABI and lack the `libgcc-s1` package (renamed from `libgcc1` starting Ubuntu 20.04). Attempting to install on these systems will fail with unmet dependency errors. Use the source-based build (`./build.sh`) if you must target older distributions.

NOTE

The source-based Debian package (`libdxrt_<version>_all.deb`) is **no longer distributed**. If you need to build from source, use `./build.sh` as described in the **Framework Build on Linux** section above.

Installing the Prebuilt Binary Package (`libdxrt-bin`)

Use the file that matches your target CPU architecture (check with `dpkg --print-architecture`).

For amd64 (x86_64) hosts:

```
sudo dpkg -i libdxrt-bin_3.4.0_amd64.deb
```

For aarch64 (arm64) hosts:

```
sudo dpkg -i libdxrt-bin_3.4.0_arm64.deb
```

If `dpkg` reports unmet dependencies, resolve them with:

```
sudo apt --fix-broken install
```


© Copyright 2026 DEEPX All Rights Reserved.

NOTE

Replace `3.4.0` in the file name with the version you actually received.

NOTE

When installing via the Debian package, the Python module is installed to the system Python. For virtual environments, please refer to the section "Using a Virtual Environment" below.

Bundled Python Wheels

The Debian package includes pre-built wheel files for Python 3.8 – 3.14 in `/usr/share/libdxrt-bin/python/`. These wheels can be used to install `dx_engine` in virtual environments or custom Python installations.

Post-Installation Verification

```
# Verify dx_engine Python module
python3 -c "import dx_engine; print(dx_engine.__version__)"

# Verify shared libraries are registered
ldconfig -p | grep libdxrt
```

If shared library errors occur, run:

```
sudo ldconfig
```

Using a Virtual Environment

OPTION A: ACCESS SYSTEM INSTALLATION

While the Debian package installs the Python module to the system Python, you can make it visible inside a virtual environment as follows:

```
# Install python3-venv if not already installed
sudo apt install python3-venv

# Create a virtual environment with access to system site-packages
python3 -m venv --system-site-packages my_venv
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# Activate the virtual environment
source my_venv/bin/activate
```

OPTION B: INSTALL FROM BUNDLED WHEEL (RECOMMENDED)

For more flexibility, install directly from the bundled wheel file:

```
# Create a clean virtual environment (without system packages)
python3 -m venv my_venv
source my_venv/bin/activate

# Install from the bundled wheel (adjust Python version as needed)
pip install /usr/share/libdxrt-bin/python/dx_engine-3.4.0-cp311-cp311-
linux_x86_64.whl

# Verify
python -c "import dx_engine; print(dx_engine.__version__)"
```

To find the correct wheel for your Python version:

```
ls /usr/share/libdxrt-bin/python/
# Example output:
# dx_engine-3.4.0-cp38-cp38-linux_x86_64.whl
# dx_engine-3.4.0-cp39-cp39-linux_x86_64.whl
# dx_engine-3.4.0-cp310-□cp310-linux_x86_64.whl
# dx_engine-3.4.0-cp311-cp311-linux_x86_64.whl
# ...
# dx_engine-3.4.0-cp314-cp314-linux_x86_64.whl
```

Uninstalling the package

```
sudo dpkg --purge libdxrt-bin
```

2.4.6 Build Target Options

This chapter explains how to configure optional build targets in **DX-RT**, such as enabling Python API support or shared library builds.

These options can be set by editing the following configuration file:

```
cmake/dxrt.cfg.cmake
```


© Copyright 2026 DEEPX All Rights Reserved.

Here are the available options for building targets.

```
option(USE_ORT "Use ONNX Runtime" ON)
option(USE_PYTHON "Use Python" ON)
option(USE_SERVICE "Use DXRT Service" ON)
```

- `USE_ORT` : Enables ONNX Runtime for NN (neural network) operations that NPU does **not** support
- `USE_PYTHON` : Enables Python API support
- `USE_SERVICE` : Enables service for multi-process support

> **NOTE**
>
> After modifying `dxrt.cfg.cmake`, re-run the build with **./build.sh --clean** to apply changes.

2.4.7 Build Guide for Cross-compile

This section explains how to build the **DX-RT** framework for different CPU architectures, including native and cross-compile environments.
Cross-compilation is typically used to build **DX-RT** on an **x86_64** host system for deployment on an ARM-based (**aarch64**) target.

Setup Files for Cross-compile
DX-RT supports the following CPU architectures:

- **x86_64** — Native build on host system
- **aarch64** — Cross-compile for ARM-based targets

The default architecture is **x86_64**.

Toolchain Configuration
To cross-compile, configure a CMake toolchain file located at:

```
cmake/toolchain.<CMAKE_SYSTEM_PROCESSOR>.cmake
```

Example: Toolchain for **aarch64**

```
SET(CMAKE_C_COMPILER /usr/bin/aarch64-linux-gnu-gcc )
SET(CMAKE_CXX_COMPILER /usr/bin/aarch64-linux-gnu-g++ )
SET(CMAKE_LINKER /usr/bin/aarch64-linux-gnu-ld )
SET(CMAKE_NM /usr/bin/aarch64-linux-gnu-nm )
SET(CMAKE_OBJCOPY /usr/bin/aarch64-linux-gnu-objcopy )
```


© Copyright 2026 DEEPX All Rights Reserved.

```
SET(CMAKE_OBJDUMP /usr/bin/aarch64-linux-gnu-objdump )
SET(CMAKE_RANLIB /usr/bin/aarch64-linux-gnu-ranlib )
```

Customize the paths based on your toolchain installation.

Non Cross-compile Case (Build on Host)
To build and install **DX-RT** on the host system, run the following command.

```
./build.sh --install /usr/local
```

Recommended install path: `/usr/local` (commonly included in system library search paths)

Cross-compile Case (Build for Target Architecture)
Cross-compile for a specific architecture, run the following command.

```
./build.sh --arch <target_cpu>
```

Here are the examples of cross-compile cases.

```
./build.sh --arch aarch64
./build.sh --arch x86_64
```

Output Directory
After a successful build, output binaries are in the following directory.

```
<build directory> /bin/
```

Typical contents:

```
<build directory>/bin/
 ├── dxrtd
 ├── dxcli
 ├── dxparse
 ├── dxrun
 └── examples
```

NOTE

Legacy names `dxrt-cli`, `parse_model`, and `run_model` are preserved as backward-compatible aliases and continue to work.

© Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> Python for cross-compile is **not** supported in the current version, but will be supported in the next release.

2.5 Linux Device Driver Installation

After building the **DX-RT** framework, you can install the Linux device driver for **M1 AI Accelerator (NPU).**

> **NOTE**
>
> To install the Linux device driver, you can choose either **Section. DKMS and Debian Package** or **Section. Driver Source**.

2.5.1 DKMS and Debian Package

Before you begin, ensure you have `sudo` privileges and an active internet connection.

Installing DKMS Debian Package

Here's how to install a DKMS (Dynamic Kernel Module Support) Debian package, along with explanations of messages you might see during the process.

Pre-installation Step (Recommended)
Before installing any Debian package, it's highly recommended to update your package lists. This ensures that your system has the most current information about available packages and their dependencies from the configured repositories. This step is particularly important when using `apt` for dependency resolution.

Update Package Lists:

```
sudo apt update
```

This command refreshes your system's package cache.

Installation Steps
There are two primary ways to install a local DKMS Debian package:

© Copyright 2026 DEEPX All Rights Reserved.

Using `sudo dpkg -i` (For direct package installation)

```
sudo dpkg -i [package_name.deb]
```

This command directly installs the `.deb` file. While it's quicker for simple installations, it does not automatically resolve dependencies. If your package has unmet dependencies, `dpkg` will warn you, and you might need to run `sudo apt --fix-broken install` afterward to resolve them.

Just replace `[package_name.deb]` with the actual name of your Debian package file.

Using `sudo apt install` (Recommended for dependency handling)

```
sudo apt install ./[package_name.deb]
```

This method is generally preferred because `apt` automatically handles any package dependencies for you.

Understanding Installation Messages
You might see various messages during installation, especially when using `sudo apt install`. Here's what they mean:

`initramfs` Messages

```
I: The initramfs will attempt to resume from /dev/dm-1
I: (/dev/mapper/vgubuntu-swap_1)
I: Set the RESUME variable to override this.
```

These are simply informational messages about your system's boot process, specifically regarding how `initramfs` handles resuming from a swap partition. They are not errors related to your DKMS package installation and you can safely ignore them. They indicate that your system is configured to resume from a swap partition during boot, which is a normal setup.

"Permission denied" Message (when using `apt`)

```
N: Download is performed unsandboxed as root as file '/.../release/dkms_package/dxrt-driver-dkms_[version]_all.deb' couldn't be accessed by user '_apt'. - pkgAcquire::Run (13: Permission denied)
```


© Copyright 2026 DEEPX All Rights Reserved.

This message can be a bit confusing, but it generally means your package installed successfully. It appears because:

- You're installing a local `.deb` file directly using `sudo apt install ./...`.
- Normally, `apt` downloads packages from remote repositories using a lower-privileged user (like `_apt`) for security.
- However, since you're using `sudo` to install a local file, you're already operating with root privileges. The message simply tells you that the `_apt` user, which has lower permissions, couldn't access the local file.

In short, the installation went through fine because you used `sudo`. This message is just `apt` letting you know about a security check it couldn't perform on a local file, but it doesn't affect the successful installation.

Checking DKMS Driver Status

You can check the status of your DKMS drivers (which driver versions have been added, built, or installed for specific kernel versions) using the `dkms status` command:

```
dkms status
```

For instance:

```
dxrt-driver-dkms/1.7.1-1, 6.5.0-27-generic, x86_64: installed
```

In summary, these lines mean:

- The `dxrt-driver-dkms` module, version `1.7.1-1`, is currently installed and fully operational for your system running the `6.5.0-27-generic` kernel on an `x86_64` architecture.
- The same `dxrt-driver-dkms` module, version `1.7.1-1`, is also installed and fully operational for your system running the `6.8.0-60-generic` kernel on an `x86_64` architecture.

Viewing Package information

To quickly view detailed information about a Debian/Ubuntu package, including those that provide DKMS drivers, you can use the `apt show` command.

Command:

```
apt show [package-name]
```


© Copyright 2026 DEEPX All Rights Reserved.

Replace `[package-name]` with the actual name of the package you want to inspect (e.g., `dxrt-driver-dkms`).

Example Usage and Output Interpretation:
If you run `apt show dxrt-driver-dkms`, you'd see output similar to this:

```
Package: dxrt-driver-dkms
Version: 1.7.1-1
Status: install ok installed
Priority: optional
Section: kernel
Maintainer: ...
Installed-Size: 698 kB
Depends: dkms
Homepage: https://deepx.ai
Download-Size: unknown
APT-Manual-Installed: yes
APT-Sources: /var/lib/dpkg/status
Description: A DEEPX NPU Driver
```

This tells you that `dxrt-driver-dkms` is version `1.7.1-1`, is a DKMS module, requires the `dkms` package, comes from a specific APT source, and its purpose is to provide the **DX-RT** driver source for automatic rebuilding with new kernels.

Finding Locally Installed Source Code

The source code for DKMS drivers installed via packages is reliably located under the `/usr/src/` directory. This is the designated place where DKMS expects to find the source files needed for compilation.

Path:

```
/usr/src/[module-name]-[module-version]/
```

Example:
For the `dxrt-driver-dkms` package (version `1.7.1-1`), the source code would likely be in `/usr/src/dxrt-driver-1.7.1-1/`. You can then `cd` into this directory to explore the source files.

© Copyright 2026 DEEPX All Rights Reserved.

2.5.2 Driver Source

Prerequisites

Before installing the Linux device driver, you should check that the accelerator device is properly recognized by the system.

To check PCIe device recognition, run the following command.

```
lspci -vn | grep 1ff4

0b:00.0 1200: 1ff4:0000
```

> **NOTE**
>
> **If there is no output,** the PCIe link is **not** properly connected. Please check the physical connection and system BIOS settings.

Optional. Display the DEEPX name in `lspci`.
If you want to display the DEEPX name in `lspci`, you can modify the PCI DB. (Only for Ubuntu)
To display the DeepX device name, run the following command.

```
sudo update-pciids
lspci
...
0b:00.0 Processing accelerators: DEEPX Co., Ltd. DX_M1
```

Linux Device Driver Structure

The **DX-RT** Linux device driver source is structured to support flexible builds across devices, architectures, and modules. The directory layout is as follows.

```
- .gitmodules

- [modules]
    |
    - device.mk
    - kbuild
    - Makefile
    - build.sh
    - [rt]
        - Kbuild
    - [pci_deepx] : submodule
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        - Kbuild

- [utils] : submodule
```

- `device.mk` : Device configuration file
- `kbuild` : Top-level build rules
- `Makefile` : Build entry point
- `build.sh` : Build automation script
- `rt` : Runtime driver source (`dxrt_driver.ko`)
- `pci_deepx` : PCIe DMA driver (`submodule, dx_dma.ko`)
- `utils` : Supporting utilities (`submodule`)

Here are the descriptions of the key components.

`device.mk`
Defines supported device configuration.

To build for a specific device, run the following command.

```
make DEVICE=[device]
```

For example, in the case of a device like **M1**, you should select a submodule, such as PCIe, that has a dependency on **M1**.

```
make DEVICE=m1 PCIE=[deepx]
```

`kbuild`
Linux kernel build configuration file for each module directory. It instructs the kernel build system on how to compile driver modules.

`build.sh`
Shell script to streamline the build process. It runs the Makefile with common options.

Here are the options for `build.sh` .

```
Usage:
   build.sh <options>

options:
   -d, --device   [device]      select target device: m1
   -m, --module   [module]      select PCIe module: deepx
   -k, --kernel   [kernel dir]  'KERNEL_DIR=[kernel dir]', The directory where the
                                kernel source is located
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
                                default: /lib/modules/6.5.0-18-generic/build)
    -a, --arch     [arch]       set 'ARCH=[arch]' Target CPU architecture for
                                cross-compilation, default: x86_64
    -t, --compiler [cross tool] 'CROSS_COMPILE=[cross tool]' cross compiler binary,
                                e.g aarch64-linux-gnu-
    -i, --install  [install dir] 'INSTALL_MOD_PATH=[install dir]', module install
                                directory install to:
                                [install dir]/lib/modules/[KERNELRELEASE]/extra/
    -c, --command  [command]    clean | install | uninstall
                                - uninstall: Remove the module files installed
                                on the host PC.
    -j, --jobs     [jobs]       set build jobs
    -f, --debug    [debug]      set debug feature [debugfs | log | all]
    -v, --verbose               build verbose (V=1)
    -h, --help                  show this help
```

The build process generates the following kernel modules.

- `modules/rt` -> `dxrt_driver.ko`
  : a core runtime driver for **M1 NPU** devices. This is responsible for system-level communication, memory control, and device command execution.
- `modules/pci_deepx` -> `dx_dma.ko`
  : PCIe DMA (Direct Memory Access) kernel module for high-speed data transfer between host and the **M1** device. This enables efficient data movement with minimal CPU overhead, ideal for real-time and data intensive AI workloads.

Linux Device Driver Build

After completing the environment setup of the DXNN Linux Device Driver, you can build the kernel modules using either the make(Makefile) or `build.sh` script. Both methods are supported by DEEPX.

> **NOTE**
>
> If the DKMS Debian package is already installed, it needs to be removed in order to build and install the driver from the source.

Option 1. Build Using `Makefile`

`build`

```
e.g $ cd modules
e.g $ make DEVICE=m1 PCIE=deepx
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

clean

```
e.g $ cd modules
e.g $ make DEVICE=m1 PCIE=deepx clean
```

install

Installs the driver to: `/lib/modules/$(KERNELRELEASE)/extra/`

```
e.g $ cd modules
e.g $ make DEVICE=m1 PCIE=deepx install
```

Option 2. Build Using `build.sh`

Use this method if your system supports self-compiling kernel modules (`.ko` files).

build

```
e.g $ ./build.sh -d m1 -m deepx
(Default device: m1, PCIe module: deepx)
```

clean

```
e.g $ ./build.sh -c clean
```

install

Installs the driver to: `/lib/modules/$(KERNELRELEASE)/extra/`

```
e.g $ sudo ./build.sh -c install
```

Auto-Loading Modules at Boot Time

DEEPX allows kernel modules to be automatically loaded at system boot, either through manual setup or using the `build.sh` script.

Manual Installation Method

Step 1. Install Kernel Modules
Installs modules to: `/lib/modules/$(KERNELRELEASE)/extra/`

```
make DEVICE=m1 PCIE=deepx install
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Step 2. Update Module Dependencies
Updates: `/lib/modules/$(KERNELRELEASE)/modules.dep`

```
sudo depmod -A
```

Step 3. Add Module Configuration
Copy the preconfigured module config file.

```
sudo cp modules/dx_dma.conf /etc/modprobe.d/
```

This ensures the modules (`dxrt_driver` , `dx_dma`) are auto-loaded on boot.

Step 4. Test with modprobe
To verify the correct installation.

```
sudo modprobe dx_dma
lsmod

  dxrt_driver            40960  0
  dx_dma                176128  1 dxrt_driver
```

Automated Installation Using `build.sh`
The `build.sh` script automates installation and setup, including dependency updates and modprobe configuration.

Run the following command

```
sudo ./build.sh -d m1 -m deepx -c install

- DEVICE        : m1
- PCIE          : deepx
- MODULE CONF   : /.../rt_npu_linux_driver/modules/dx_dma.conf
- ARCH (HOST)   : x86_64
- KERNEL        : /lib/modules/5.15.0-102-generic/build
- INSTALL       : /lib/modules/5.15.0-102-generic/extra/


*** Build : install ***
make DEVICE=m1 PCIE=deepx install

make -C /lib/modules/5.15.0-102-generic/build M=/home/jhk/deepx/dxrt/module/
rt_npu_linux_driver/modules  modules_install

 ....
 - SUCCESS
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
*** Update : /lib/modules/5.15.0-102-generic/modules.dep ***
depmod -A
cp /home/jhk/deepx/dxrt/module/rt_npu_linux_driver/modules/dx_dma.conf /etc/
modprobe.d/
```

Uninstalling Modules

To completely remove the installed modules and configs.

```
./build.sh -d m1 -m deepx -c uninstall

- DEVICE        : m1
- PCIE          : deepx
- MODULE CONF   : /.../rt_npu_linux_driver/modules/dx_dma.conf
- ARCH (HOST)   : x86_64
- KERNEL        : /lib/modules/5.15.0-102-generic/build
- INSTALL       : /lib/modules/5.15.0-102-generic/extra/


*** Remove : /lib/modules/5.15.0-102-generic/extra ***
rm -rf /lib/modules/5.15.0-102-generic/extra/pci_deepx
rm -rf /lib/modules/5.15.0-102-generic/extra/rt


*** Remove : /etc/modprobe.d ***
rm /etc/modprobe.d/dx_dma.conf


*** Update : /lib/modules/5.15.0-102-generic/modules.dep ***
depmod
```

2.6 Python Package Installation

This chapter explains how to install the **DX-RT** Python package, provided under the module name dx_engine. The package supports Python 3.8.10 or later and enables Python-based applications to interface with the **DX-RT** runtime.

Installation Steps

Step 1. Navigate to the python_package directory.

```
cd python_package
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Step 2. Install the package

```
pip install .
```

Step 3. Verify the installation

```
pip list | grep dx

dx-engine          1.1.4
```

Alternatively, you can check the version programmatically:

Using Python Interactive Shell:

```
import dx_engine
print(dx_engine.__version__)
# Output: 1.1.4
```

Using Command Line:

```
python3 -c "import dx_engine; print(dx_engine.__version__)"
# Output: 1.1.4
```

Verify Available Modules and Classes:

```
import dx_engine
print(dir(dx_engine))
# Output: ['Configuration', 'DeviceStatus', 'InferenceEngine', 'InferenceOption',
#          'RuntimeEventDispatcher', '__version__', ...]
```

These methods are useful for: - Automated build scripts and CI/CD pipelines - Version compatibility checks before running applications - Debugging installation issues

For details on using **DX-RT** with Python, refer to **Section. Python Tutorials**.

© Copyright 2026 DEEPX All Rights Reserved.

NOTE

Python C++ Module Version Mismatch: If you encounter errors related to Python C++ module version mismatch, it typically means the compiled Python extension was built with a different Python version than the one currently being used. Common error messages include:

```
malloc(): corrupted top size
```

or

```
free(): invalid pointer
```

or

```
double free or corruption (out)
```

Solutions
1. **Rebuild with correct Python version**: Ensure you're using the same Python version for both building and running:

```
python3 --version  # Check current Python version
pip --version      # Check current pip version
cd python_package
pip install .
```

2. **Use virtual environment**: Create a virtual environment with the correct Python version:

```
python3.10 -m venv dxrt_env
source dxrt_env/bin/activate
cd python_package
pip install .
```

2.7 Service Registration

This chapter explains how to register and run the **DX-RT** background service (`dxrtd`) to enable multi-process inference. **DX-RT** uses a systemd-managed daemon to support concurrent access to the NPU by multiple applications.

© Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> - **DX-RT must** be built with `USE_SERVICE=ON`. (default setting)
> - **DX-RT must** be registered and managed as a system service using `systemd`.

Registering and Running the DX-RT Service

Step 1. Modify the service unit file.
Ensure the ExecStart path is correctly configured.

```
vi ./service/dxrt.service
```

Step 2. Copy the service file to the system folder.

```
sudo cp ./service/dxrt.service /etc/systemd/system
```

Step 3. Start the service.

```
sudo systemctl start dxrt.service
```

Service Management Commands

```
sudo systemctl stop dxrt.service           # Stop the service
sudo systemctl status dxrt.service      # Check service status
sudo systemctl restart dxrt.service     # Restart the service
sudo systemctl enable dxrt.service      # Enable on boot
sudo systemctl disable dxrt.service     # Disable on boot
sudo journalctl -u dxrt.service            # View service logs
```

2.8 Sanity Check

This chapter explains how to use the Sanity Check script to verify that the **DX-RT** driver is correctly installed and that the NPU device is recognized by the system.

2.8.1 Script Usage

The general syntax for running the Sanity Check script is:

```
sudo ./SanityCheck.sh [option]
```


© Copyright 2026 DEEPX All Rights Reserved.

If no option is provided, the script defaults to all mode.

2.8.2 Available Options

The following options are available:

- `all` (Default): Performs a comprehensive check of all components, including both the runtime and the driver. If no option is specified, this is the action performed.
- `dx_rt` : Runs a check specifically on the DEEPX Runtime (dx_rt) components.
- `dx_driver` : Runs a check specifically on the DEEPX driver (dx_driver).
- `help` : Displays a help message outlining the available commands and their functions.

2.8.3 Usage Examples

Step 1. Run a full system check (Recommended)

This performs a comprehensive check of both the runtime and the driver.
Use this after installation or when verifying overall system status.

```
sudo ./SanityCheck.sh
```

or

```
sudo ./SanityCheck.sh all
```

```
================================================================================
==== Sanity Check Date : ... ====
Log file location : /.../sanity_check_result_..._....log

==== PCI Link-up Check ====
[OK] Vendor ID 1ff4 is present in the PCI devices.(num=1)
==== Device File Check ====
[OK] /dev/dxrt0 exists.
[OK] /dev/dxrt0 is a character device.
[OK] /dev/dxrt0 has correct permissions (0666).
==== Kernel Module Check ====
[OK] dxrt_driver module is loaded.
[OK] dx_dma module is loaded.
[OK] PCIe 02:00.0 driver probe is success.
==== Legacy Driver Installed Check ====
[INFO] /lib/modules/6.8.0-87-generic/kernel/drivers/dxrt_driver.ko ... NONE
[INFO] /lib/modules/6.8.0-87-generic/kernel/drivers/dx_dma.ko ... NONE
```


© Copyright 2026 DEEPX All Rights Reserved.

```
==== DKMS Driver Installed Check ====
[INFO] dxrt-driver-dkms/1.8.0-2, 6.8.0-87-generic, x86_64: installed
[INFO] /lib/modules/6.8.0-87-generic/updates/dkms/dxrt_driver.ko ... OK
[INFO] /lib/modules/6.8.0-87-generic/updates/dkms/dx_dma.ko ... OK
==== Runtime Version Dependency Check ====
[OK] Version Dependency Check
   Runtime Framework Version: 3.4.0
   Device Driver Version: v2.5.0
   PCIe Driver Version: v2.4.0
   Firmware Version: device-id=0 v2.7.0
   ONNX Runtime Version: v1.20.1
==== Runtime Executable File Check ====
[OK] Executable File Check
   Runtime Framework Version: 3.4.0
   Device Driver Version: v2.5.0
   PCIe Driver Version: v2.4.0
   Firmware Version: device-id=0 v2.7.0
   ONNX Runtime Version: v1.20.1
   dxcli ...OK
   dxrun ...OK
   dxparse ...OK
   dxtop ...OK
   dxrtd ...OK
   Header: /usr/local/include/dxrt ...OK
==== Runtime dxrtd Service Check ====
[OK] The dxrtd service is running correctly. This is the expected behavior for builds
with USE_SERVICE=ON on cmake/dxrt.cfg.cmake

================================================================================
** Sanity check PASSED!
================================================================================
```

Step 2. Check only **DX-RT**

Use this option to validate that the runtime components (`dx_rt`) are properly installed and functional.

```
sudo ./SanityCheck.sh dx_rt
```

```
================================================================================
==== Sanity Check Date : ... ====
Log file location : /.../sanity_check_result_..._....log

==== Runtime Version Dependency Check ====
[OK] Version Dependency Check
   Runtime Framework Version: 3.4.0
   Device Driver Version: v2.5.0
   PCIe Driver Version: v2.4.0
   Firmware Version: device-id=0 v2.7.0
   ONNX Runtime Version: v1.20.1
==== Runtime Executable File Check ====
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
[OK] Executable File Check
   Runtime Framework Version: 3.4.0
   Device Driver Version: v2.5.0
   PCIe Driver Version: v2.4.0
   Firmware Version: device-id=0 v2.7.0
   ONNX Runtime Version: v1.20.1
   dxcli ...OK
   dxrun ...OK
   dxparse ...OK
   dxtop ...OK
   dxrtd ...OK
   Header: /usr/local/include/dxrt ...OK
==== Runtime dxrtd Service Check ====
[OK] The dxrtd service is running correctly. This is the expected behavior for builds
with USE_SERVICE=ON on cmake/dxrt.cfg.cmake

================================================================================
** Sanity check PASSED!
================================================================================
```

Step 3. Check **only** the **DX-RT** Driver

Use this option to test the driver (`dx_driver`) independently, especially after driver installation or kernel updates.

```
sudo ./SanityCheck.sh dx_driver
```

```
================================================================================
==== Sanity Check Date : ... ====
Log file location : /.../sanity_check_result_..._....log

==== PCI Link-up Check ====
[OK] Vendor ID 1ff4 is present in the PCI devices.(num=1)
==== Device File Check ====
[OK] /dev/dxrt0 exists.
[OK] /dev/dxrt0 is a character device.
[OK] /dev/dxrt0 has correct permissions (0666).
==== Kernel Module Check ====
[OK] dxrt_driver module is loaded.
[OK] dx_dma module is loaded.
[OK] PCIe 02:00.0 driver probe is success.
==== Legacy Driver Installed Check ====
[INFO] /lib/modules/6.8.0-87-generic/kernel/drivers/dxrt_driver.ko ... NONE
[INFO] /lib/modules/6.8.0-87-generic/kernel/drivers/dx_dma.ko ... NONE
==== DKMS Driver Installed Check ====
[INFO] dxrt-driver-dkms/1.8.0-2, 6.8.0-87-generic, x86_64: installed
[INFO] /lib/modules/6.8.0-87-generic/updates/dkms/dxrt_driver.ko ... OK
[INFO] /lib/modules/6.8.0-87-generic/updates/dkms/dx_dma.ko ... OK

================================================================================
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
** Sanity check PASSED!
================================================================================
```

2.8.4 Help Command

To display usage instructions and available options for the Sanity Check script, run the following command:

```
sudo ./SanityCheck.sh help
```

This will print a summary of all supported options, including descriptions for `all, dx_rt, and dx_driver`.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3. Installation on Windows

This chapter describes the system requirements, source file structure, and the installation instructions for setting up **DX-RT** on a Windows-based host system.

After you check the system requirements, follow these instructions.

- System Requirement Check
- DX-RT Windows Driver Installation
- Visual Studio 2022 Setup
- DX-RT Framework Build
- Runtime Service Setup
- Installation Verification

3.1 Overview & System Requirements

DX-RT is a runtime software package for operating DEEPX NPU. This software is compatible with multiple DEEPX NPU products, including **DX-M1**, **DX-M1M**, and **DX-H1 Quattro**. The main build outputs from this repository are:

Output	Description
`dxrt.dll`	Core runtime library for NPU inference
`dxrt.lib`	Link library for application builds
`dxrtd.exe`	Runtime daemon service for managing apps using `dxrt.dll`
`dxcli.exe`	Device management and firmware interface tool
`dxrun.exe`	Model inference and performance verification tool
`dxparse.exe`	Model information viewer
`dxbenchmark.exe`	Benchmarking utility
`dxtop.exe`	Real-time NPU monitoring tool

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The built libraries are used by dx_app, and users can also develop their own applications by referencing the dx_app code.

> **NOTE. Legacy Names**
>
> Legacy executable names `dxrt-cli.exe`, `run_model.exe`, and `parse_model.exe` are preserved as backward-compatible copies and continue to work.

> **NOTE. Pre-built Binaries**
>
> To use pre-built binaries without building, refer to the dx_rt_windows repository.

3.1.1 Hardware Requirements

Item	Requirement	Notes
CPU	amd64 (x86_64)	
RAM	8GB or more	16GB or more recommended
Storage	4GB or more	Free disk space
NPU Connection	DEEPX NPU device	Choose one of the connection methods below

Supported NPU Connection Methods

Connection Method	Description
PCIe Slot	Direct connection to PCIe slot
M.2 Slot	Connection via M.2 interface
USB 4.0	PCIe tunneling support required (Thunderbolt™ or USB4)

3.1.2 Software Requirements

Component	Requirement
OS	Windows 10 or later (x64)

© Copyright 2026 DEEPX All Rights Reserved.

Component	Requirement
Visual Studio	Visual Studio Community 2022 (Desktop development with C++ workload)
CMake	Included with Visual Studio 2022
VCPKG	Included with Visual Studio 2022
Driver	v1.8.0 or later
PCIe Driver	v1.6.1 or later
Firmware	v2.7.0 or later

> ⚠ **IMPORTANT**
>
> **The NPU device driver must be installed first.** For driver installation instructions, see Section. **DX-RT Windows Driver Installation** below.

3.2 Pre-installation Setup

3.2.1 DX-RT Windows Driver Installation

Before building and running DX-RT, you need to install the DX-RT Windows driver.

Driver Download

Drivers are provided in the dx_rt_windows repository.

```
dx_rt_windows/
└── m1/v.3.4.0/
  ├── dxm1drv/          # PCIe driver package (dxm1drv.zip)
  ├── dx_rt/            # Runtime (bin/, include/, lib/)
   └── dx_app/           # Demo applications and examples
```

- `dxm1drv/` : DX-RT Windows driver (dxm1drv.zip archive)
- `dx_rt/` : DLLs and executables built from DX-RT code
- `dx_app/` : Executables built by linking dx_app repository with dxrt.lib

Driver Installation Steps

© Copyright 2026 DEEPX All Rights Reserved.

Step 1. Connect the Device

Connect the DEEPX NPU device to your system.

- **PCIe or M.2 connection:** Install device in an available PCIe or M.2 slot
- **USB connection:** Connect to a USB 4.0 port with PCIe tunneling support (regular USB 3.x ports are not supported)

Step 2. Install the Driver

2-1. Navigate to the `m1/v.3.4.0/dxm1drv/` directory
2-2. Extract `dxm1drv.zip`
2-3. Open the extracted `dxm1drv/` folder
2-4. Right-click on the `dxm1drv.inf` file
2-5. Select **Install** from the context menu
2-6. If a security warning about the driver publisher appears, click **Install** or **Yes**
2-7. Wait for installation to complete

Advanced. Installation via Command Line

Open Command Prompt or PowerShell as Administrator and run.

```
pnputil /add-driver "path\to\extracted\dxm1drv\dxm1drv.inf" /install
```

Step 3. Verify Installation

3-1. Open Device Manager (right-click Start menu → Device Manager)
3-2. Look for **M1 PCI CONTROLLER** device
3-3. Verify that the device appears in the list without warning icons

3.2.2 Visual Studio 2022 Setup

Visual Studio Community 2022 is required to build DX-RT.

Installation Steps

Step 1. Download Visual Studio Community 2022
Step 2. Run the installer and select the following workload: **Desktop development with C++**
Step 3. (Optional) Select any additional workloads or individual components as needed
Step 4. Click **Install** to proceed with installation

© Copyright 2026 DEEPX All Rights Reserved.

> **IMPORTANT. Visual Studio 2022 Required**
>
> **You must use Visual Studio 2022.** Visual Studio 2019 or earlier, or 2026 or later versions cannot build DX-RT due to compiler compatibility issues. Even if you have other versions installed, install Visual Studio 2022 separately.

3.3 DX-RT Build Methods

DX-RT can be built using two methods.

- Build using Visual Studio 2022 IDE
- Build using Command Line Interface (CLI)

3.3.1 CMake Presets Configuration

DX-RT provides the following build presets via `CMakePresets.json`

Preset	Description	Generator
`x64-Debug`	x64 Debug build	Ninja
`x64-Release`	x64 Release build	Ninja
`x64-VS`	Generate Visual Studio solution/ project files	Visual Studio 17 2022

3.3.2 Method 1: Build Using Visual Studio 2022 IDE

You can build the CMake project directly using Visual Studio 2022's Open Folder feature.

Step 1. Open Project Folder

1-1. Launch Visual Studio Community 2022
1-2. Select **Open a local folder** from the start screen
1-3. Select the DX-RT project folder (`dx_rt`)

© Copyright 2026 DEEPX All Rights Reserved.

> **IMPORTANT**
>
> **You must use Visual Studio 2022.** Compatibility with other versions (VS 2019, VS Code, etc.) is not guaranteed.

Step 2. CMake Configuration

When you open the project.

- Dependencies specified in `vcpkg.json` are automatically downloaded and installed in the `vcpkg_installed` directory.
- CMake automatically generates the build cache and configuration.

Step 3. Select Build Configuration

3-1. Click the build configuration dropdown in the top toolbar
3-2. Select the desired preset
- **x64-Debug**: For debugging builds
- **x64-Release**: For release builds

Step 4. Run Build

4-1. Go to the **Build** menu
4-2. Click **Build All** (or **Rebuild All**)

When the build succeeds, outputs are generated in

- Debug: `out/build/x64-Debug/`
- Release: `out/build/x64-Release/`

Step 5. Install

To copy built outputs to the install directory.

5-1. **Build** menu → Select **Install**

Install paths

- Debug: `out/install/x64-Debug/`
- Release: `out/install/x64-Release/`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.3.3 Method 2: Build Using build.bat

Using the `build.bat` script, you can automatically handle Visual Studio environment setup, CMake configuration, build, and installation all at once.

build.bat Overview

`build.bat` automatically handles the following:

- Auto-detection and setup of Visual Studio 2022 environment
- CMake configuration and build execution
- Installation and cleanup tasks

> **NOTE. Installing Visual Studio Build Tools (For CLI-only builds without IDE)**
>
> To build using only the command line without Visual Studio IDE, you can install just the Build Tools.
>
> ```
> build.bat install-tools
> ```
>
> This command automatically installs Visual Studio Build Tools 2022 via winget.

Step 1. Open Command Prompt

Open a regular Command Prompt (`cmd.exe`). **Developer Command Prompt is not required** - `build.bat` automatically sets up the Visual Studio environment.

Step 2. Navigate to Project Directory

```
cd C:\path\to\dx_rt
```

Step 3. Run Build

Default build (Release configuration + build)

```
build.bat
```

Debug build

```
build.bat debug
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Build and install

```
build.bat install
build.bat debug install
```

Full build (clean + configure + build + install)

```
build.bat all
build.bat debug all
```

build.bat Command Options

Command	Description
`build.bat`	Release configure and build (default)
`build.bat debug`	Debug configure and build
`build.bat configure`	Run CMake configure only (skip build)
`build.bat build`	Run build only (skip configure)
`build.bat install`	Build and install (installs to `out\install\`)
`build.bat clean`	Clean build directories
`build.bat rebuild`	Clean and rebuild (configure + build)
`build.bat all`	Full build: clean + configure + build + install
`build.bat distclean`	Full reset (delete `out/`, `build_vs/`, `vcpkg_installed/`)
`build.bat vs`	Generate Visual Studio solution file (`build_vs\dx_rt.sln`)
`build.bat install-tools`	Install VS Build Tools 2022 (using winget)
`build.bat help`	Show help

Usage Examples

```
:: Release build (configure + build)
build.bat

:: Debug build (configure + build)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
build.bat debug

:: Release build only (skip configure)
build.bat build

:: Debug build only (skip configure)
build.bat debug build

:: Release configure only
build.bat configure

:: Debug configure only
build.bat debug configure

:: Release clean and rebuild
build.bat rebuild

:: Debug clean and rebuild
build.bat debug rebuild

:: Release build + install
build.bat install

:: Debug build + install
build.bat debug install

:: Release full build (clean + configure + build + install)
build.bat all

:: Debug full build (clean + configure + build + install)
build.bat debug all

:: Generate Visual Studio project files
build.bat vs

:: Delete all build artifacts (keep source only)
build.bat distclean
```

NOTE. Generate Visual Studio Solution

Running `build.bat vs` generates the `build_vs\dx_rt.sln` file. You can open this file in Visual Studio to build in the IDE environment.

```
start build_vs\dx_rt.sln
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.3.4 Build Output & Directory Layout

Directory structure generated after build and installation.

```
dx_rt/
├── out/
│   ├── build/
│   │   ├── x64-Debug/        # Debug build outputs
│   │   └── x64-Release/      # Release build outputs
│   └── install/
│       ├── x64-Debug/        # Debug install outputs
│       │   ├── bin/          # Executables and DLLs
│       │   ├── include/      # Header files
│       │   └── lib/          # Library files
│       └── x64-Release/      # Release install outputs
│           ├── bin/
│           ├── include/
│           └── lib/
└── build_vs/                   # Visual Studio solution (when using x64-VS preset)
    └── dx_rt.sln
```

> ⚠ **WARNING. DLL Placement for Execution**
>
> The executables (`.exe`) and the DLLs (`dxrt.dll, onnxruntime.dll`) **must reside in the same folder.**
> - **Issue:** Windows may prioritize an older version of `onnxruntime.dll` found in `C:\Windows\System32`.
> - **Solution:** Always keep the DLLs in the local bin folder to ensure the correct version is loaded first and avoid runtime crashes.

3.4 Runtime Setup (Service Configuration)

After the build is complete, you must configure the runtime environment. The most critical component in this stage is the DEEPX Runtime Daemon (`dxrtd.exe`).

Understanding the DX-RT Architecture

© Copyright 2026 DEEPX All Rights Reserved.

Before proceeding with the setup, it is important to understand how the application interacts with the NPU hardware. The `dxrtd.exe` acts as a middleman (daemon) that manages the communication between your software and the physical device.

- **User Application:** Links with `dxrt.lib` and calls functions from `dxrt.dll`.
- **dxrtd.exe (Daemon):** A background service that manages NPU resource scheduling, memory allocation, and concurrency.
- **Multi-Process Support:** Because `dxrtd` manages the hardware access, multiple applications can perform inference on the NPU simultaneously without resource conflicts.

> ⚠ **IMPORTANT. Why is the Daemon Required?**
>
> Without dxrtd.exe running in the background, dxrt.dll cannot establish a connection to the DX-RT driver. Therefore, **the service must be active** before you run any inference tasks or CLI tools.

3.4.1 Option A: Register as Windows Service (Recommended)

This method ensures the daemon starts automatically with Windows and runs efficiently in the background.

A-1. Open Command Prompt or PowerShell as Administrator

A-2. Navigate to the built `bin/` directory

```
cd out\install\x64-Release\bin
```

A-3. Install Windows service

```
dxrtd.exe --install
```

A-4. Start the service

```
dxrtd.exe --start
```

A-5. Verify service is running
: Check DeepX runtime service in Windows Services list (`services.msc`)
: Or verify `dxrtd.exe` process in Task Manager

ⓒ Copyright 2026 DEEPX All Rights Reserved.

To remove the service

```
dxrtd.exe --stop
dxrtd.exe --uninstall
```

3.4.2 Option B: Manual Execution (Debug Mode)

For development and testing purposes, you can run manually. In this case, you need to keep the terminal window open.

```
cd out\install\x64-Release\bin
dxrtd.exe
```

> **IMPORTANT**
>
> When running manually, closing the terminal window will terminate the service. For production environments, **Method A (Windows Service registration)** is recommended.

> **NOTE**
>
> `dxrtd.exe --run` (or `-r`) is for Windows SCM (Service Control Manager) and should not be used for regular interactive execution.

3.5 Installation Verification (Sanity Check)

After completing the installation and service setup, follow these steps to verify that the DEEPX NPU and runtime environment are functioning correctly.

3.5.1 Device Status & Daemon Verification

Step 1. Check Device Status

```
cd out\install\x64-Release\bin
dxcli.exe -s
```

If device information is displayed without errors, the driver is installed correctly.

Step 2. Verify Runtime Daemon

© Copyright 2026 DEEPX All Rights Reserved.

Verify that `dxrtd.exe` is running.

Using Task Manager

2-1. Open Task Manager with `Ctrl + Shift + Esc`
2-2. Go to the **Details** tab
2-3. Verify that `dxrtd.exe` is in the process list

Using command line

```
tasklist | findstr dxrtd.exe
```

Step 3. Check Detailed Device Information

Check detailed information including hardware info, temperature, and utilization.

```
dxcli.exe -i
```

3.5.2 Performance & Model Inference Test

Step 4. Real-time NPU Monitoring (Optional)

Monitor NPU core utilization, memory usage, and device temperature in real-time.

```
dxtop.exe
```

Press `q` to exit the monitoring tool.

Step 5. Model Inference Test

If you have a `.dxnn` model file.

Parse model

```
dxparse.exe -m your_model.dxnn
```

Run inference

```
dxrun.exe -m your_model.dxnn -l 10
```

This command runs 10 inference loops and displays performance metrics.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.6 Tool Descriptions & Integration

3.6.1 Core Libraries & DLL Placement Warning

File	Description
`dxrt.dll`	Core runtime library for NPU inference
`dxrt.lib`	Library for application linking
`onnxruntime.dll`	ONNX Runtime library for CPU operations

WARNING. DLL Placement and Version Conflict

To ensure correct operation, `dxrt.dll` and `onnxruntime.dll` **must be located in the same directory as your executable** (`.exe`).
- **The Reason:** Windows may prioritize an older version of `onnxruntime.dll` located in `C:\Windows\System32` .
- **The Consequence:** DX-RT requires a specific higher version; using the system's default version will lead to runtime crashes or entry-point errors.
- **Status:** A permanent fix for this search-order dependency is scheduled for the next release.

3.6.2 Command Line Tool Reference

System Services

File	Description
`dxrtd.exe`	DEEPX Runtime Daemon - manages multi-process/device support

Command Line Tools

Tool	Description	Key Options
`dxcli.exe`	Device management and firmware interface	`-s` status, `-i` info, `-r` reset, `-u` firmware update
`dxrun.exe`	Model inference and performance verification tool	`-m` model, `-l` loop count, `-v` verbose
`dxparse.exe`	Model information viewer	

© Copyright 2026 DEEPX All Rights Reserved.

Tool	Description	Key Options
		`-m` model, `-v` verbose, `-j` JSON extract
`dxbenchmark.exe`	Batch performance benchmarking	`--dir` model directory, `-l` loop count
`dxtop.exe`	Real-time NPU monitoring	-

For detailed usage of each tool, check the `-h` option

```
dxrun.exe -h
dxbenchmark.exe -h
```

3.6.3 Integration with dx_app

The built **DX-RT** library can be used with dx_app.

- Role: `dx_app` provides practical reference code and demo applications using the dxrt framework.
- For integration instructions with dx_app, refer to the DX-APP Installation and Build documentation.

3.7 Troubleshooting

3.7.1 Device Not Detected

- Verify the device is properly seated in the PCIe/M.2 slot
- Check for warning icons in Device Manager
- Try reinstalling the driver using the right-click INF file installation method

3.7.2 dxrtd.exe Won't Start

- Manually run `dxrtd.exe` from the `bin/` directory
- Check if Windows Firewall or antivirus software is blocking it
- Run as Administrator if permission issues occur

3.7.3 Build Failure

- Verify Visual Studio 2022 is correctly installed

© Copyright 2026 DEEPX All Rights Reserved.

- Verify "Desktop development with C++" workload is installed
- Delete CMake cache and retry: delete the `out/build` folder

3.7.4 Performance Issues

- Verify `dxrtd.exe` is running
- Change Windows power management setting to **High Performance**
- Ensure proper cooling for the NPU device
- Monitor thermal throttling with `dxtop.exe`

3.8 Related Links

- dx_rt_windows - Pre-built Windows binaries and drivers
- dx_app - Demo applications using DX-RT

ⓒ Copyright 2026 DEEPX All Rights Reserved.

4. Model Inference

This chapter explains how to run AI model inference using the **DX-RT SDK**. It covers the model file format, step-by-step inference workflow, support for multi-device execution, tensor data handling, and profiling tools. You can also find guidance on building complete applications, enabling hardware acceleration, and optimizing CPU task throughput when using **DX-RT** in performance-critical environments.

4.1 Model File Format

The original ONNX model is converted by **DX-COM** into the following structure.

```
Model dir.
   └── graph.dxnn
```

- `graph.dxnn`

  A unified DEEPX artifact that contains NPU command data, model metadata, model parameters.

This file is used directly for inference on DEEPX hardware.

4.2 Inference Workflow

Here is the inference workflow of the DXNN Runtime.

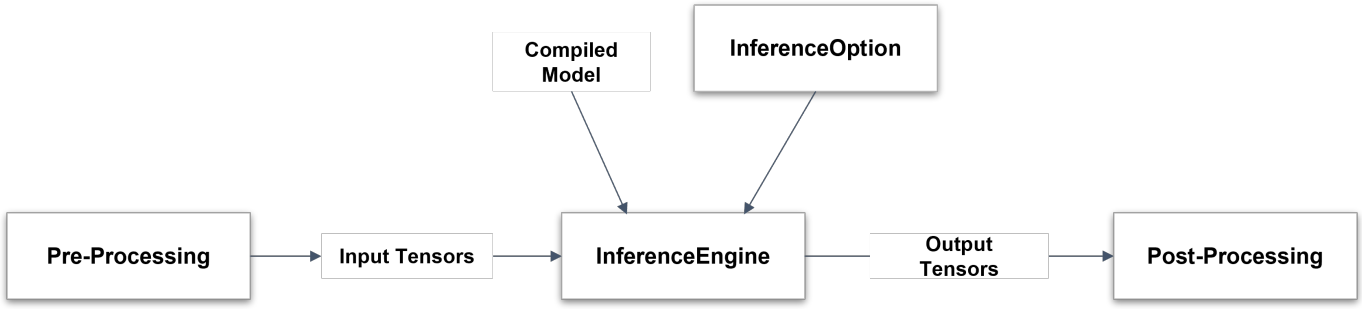

Figure. Inference Workflow

© Copyright 2026 DEEPX All Rights Reserved.

- Compiled Model and optional InferenceOption are provided to initialize the InferenceEngine.
- Pre-processed Input Tensors are passed to the InferenceEngine for inference.
- The InferenceEngine produces Output Tensors as a result of the inference.
- These outputs are then passed to the Post-Processing stage for interpretation or further action.

4.2.1 Prepare the Model

Choose one of the following options.

- Use a pre-built model from **DX ModelZoo**
- Compile an ONNX model into the **DX-RT** format using **DX-COM** (Refer to the **DEEPX DX-COM User Manual** for details.)

4.2.2 Configure Inference Options

Create a `dxrt::InferenceOption` object to configure runtime settings for the inference engine. This allows you to control device selection, NPU core binding, CPU task execution, and buffer allocation.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Available Configuration Options

- **`devices`** (default: empty = all devices)
  Specifies which device IDs to use for inference. If empty, all available devices are used.

```
option.devices = {0, 1};  // Use device 0 and device 1 only
```

- **`boundOption`** (default: `NPU_ALL`)
  Selects which NPU core(s) inside the device to use for inference.
- `NPU_ALL`: Uses all NPU cores simultaneously (recommended for maximum throughput)
- `NPU_0`, `NPU_1`, `NPU_2`: Uses only a single specific NPU core
- `NPU_01`, `NPU_12`, `NPU_02`: Uses a pair of NPU cores

```
option.boundOption = dxrt::InferenceOption::NPU_ALL;
```

- **`useORT`** (default: depends on build configuration)
  Enables ONNX Runtime for executing CPU-side tasks that are not supported by the NPU.
- `true`: Both NPU and CPU (via ONNX Runtime) tasks are executed
- `false`: Only NPU tasks are executed

```
option.useORT = true;  // Enable CPU fallback for unsupported operations
```

- **`bufferCount`** (default: `DXRT_TASK_MAX_LOAD_VALUE`)
  Specifies the number of internal buffers allocated for inference. Higher values may improve throughput in pipelined scenarios.

```
option.bufferCount = 4;  // Allocate 4 inference buffers
```

Example Usage

```
dxrt::InferenceOption option;
option.devices = {0};                              // Use only device 0
option.boundOption = dxrt::InferenceOption::NPU_ALL;  // Use all cores
option.useORT = true;                              // Enable CPU tasks
option.bufferCount = 4;                            // Use 4 buffers

auto ie = dxrt::InferenceEngine("model.dxnn", &option);
```


© Copyright 2026 DEEPX All Rights Reserved.

4.2.3 Load the Model into the Inference Engine

Create a `dxrt::InferenceEngine` instance using the path to the compiled model directory. Hardware resources are automatically initialized during this step.

If `dxrt::InferenceOption` is **not** provided, a default option is applied.

```
// Option 1: Load from file path
auto ie = dxrt::InferenceEngine("yolov5s.dxnn");

// Option 2: Load from file path with options
auto ie = dxrt::InferenceEngine("yolov5s.dxnn", &option);

// Option 3: Load from memory buffer
std::ifstream file("yolov5s.dxnn", std::ios::binary | std::ios::ate);
std::streamsize yolov5s_buffer_size = file.tellg();
file.seekg(0, std::ios::beg);
std::vector<uint8_t> yolov5s_buffer(yolov5s_buffer_size);
if (file.read(reinterpret_cast<char*>(yolov5s_buffer.data()), yolov5s_buffer_size)) {
    auto ie = dxrt::InferenceEngine(yolov5s_buffer.data(), yolov5s_buffer_size);
}

// Option 4: Load from memory buffer with options
auto ie = dxrt::InferenceEngine(yolov5s_buffer.data(), yolov5s_buffer_size, &option);
```

4.2.4 Connect Input Tensors

Prepare input buffers for inference.

The following example shows how to initialize the buffer with the appropriate size.

```
std::vector<uint8_t> inputBuf(ie.GetInputSize(), 0);
```

Refer to **DX-APP User Guide** for practical examples on connecting inference engines to image sources such as cameras or video, along with the preprocessing routines.

4.2.5 Inference

DX-RT provides both synchronous and asynchronous execution modes for flexible inference handling.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Run - Synchronous Execution

Use the `dxrt::InferenceEngine::Run()` method for blocking, single-core inference.

```
auto outputs = ie.Run(inputBuf.data());
```

- This method processes input and output on the same thread.
- This method is suitable for simple and sequential workloads.

Run - Asynchronous Execution

With `Wait()`

Use `RunAsync()` to perform the inference in non-blocking mode, and retrieve results later with `Wait()`.

```
auto jobId = ie.RunAsync(inputBuf.data());
auto outputs = ie.Wait(jobId);
```

- This method is ideal for parallel workloads where inference can run in the background.
- This method is continuously executed while waiting for the result.

With `Callback function`

Use a callback function to handle output as soon as inference completes.

```
std::function<int(vector<shared_ptr<dxrt::Tensor>>, void*)> postProcCallBack = \
    [&](vector<shared_ptr<dxrt::Tensor>> outputs, void *arg)
    {
        /* Process output tensors here */
        ... ...
        return 0;
    };
ie.RegisterCallback(postProcCallBack)
```

- The callback is triggered by a background thread after inference.
- You can pass a custom argument to track input/output pairs.

> **NOTE**
>
> Output data is **only** valid within the callback scope.

© Copyright 2026 DEEPX All Rights Reserved.

4.2.6 Process Output Tensors

Once inference is complete, the output tensors are processed using Tensor APIs and custom post-processing logic. You can find the templates and example code in **DX-APP** to help you implement post-process smoothly.
As noted earlier, using callbacks allows for more efficient and real-time post-processing.

4.3 Multiple Device Inference

This feature is **not** applicable to single-NPU devices. Basically, the inference engine schedules and manages multiple devices in real time.
If the inference option is explicitly set, the inference engine may **only** use specific devices during real-time inference for the model.

4.4 Data Format of Device Tensor

Compiled models use the **NHWC** format by default.

However, the input tensor formats on the device side may vary depending on the hardware's processing type.

Input Tensor Formats

Type	Compiled Model Format	Device Format	Data Size
`Formatter`	`[N, H, W, C]`	`[N, H, W, C]`	8-bit
`IM2COL`	`[N, H, W, C]`	`[N, H, align64(W*C)]`	8-bit

- Formatter Type Example: `[1, 3, 224, 224] (NCHW) -> [1, 224, 224, 3] (NHWC)`
- IM2COL Type Example: `[1, 3, 224, 224] (NCHW) -> [1, 224, 224*3+32] (NH, aligned width x channel)`

Output Tensor Formats

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The output tensor format is also aligned with the NHWC format, but with padding applied for alignment.

Type	Compiled Model Format	Device Format
`Aligned NHWC`	`[N, H, W, C]`	`[N, H, W, align64(C)]`

- Output Example: `[1, 40, 52, 36] (NCHW) -> [1, 52, 36, 40+24]` (Channel size is aligned for optimal memory access.)

Post-processing can be performed directly without converting formats.
The API to convert from device format to **NCHW/NHWC** format will be supported in the next release.

4.4.1 Automatic Dummy Padding/Slicing (USE_ORT=OFF)

Starting with v3.0.0, when ONNX Runtime is disabled (built with `USE_ORT=OFF` or runtime option `InferenceOption.use_ort = False`), the DX-RT runtime automatically:

- Pads input tensors to the NPU-aligned format required by each task (e.g., IM2COL, align64 width/channel), and
- Slices any alignment padding from output tensors before returning them to the application.

As a result, applications no longer need to manually attach input dummy bytes or remove output dummy bytes in non-ORT inference paths. This behavior applies to both C++ and Python APIs, including PPU models. If you provide user output buffers, ensure the buffer size is at least `ie.GetOutputSize()` (C++) or `ie.get_output_size()` (Python).

NOTE

- This automatic handling is internal to the runtime's NPU format processing and does not change model-visible tensor shapes reported by the APIs.
- When `use_ort = True`, CPU-side execution for unsupported subgraphs is enabled via ONNX Runtime; NPU tasks still follow the same alignment policy internally.

© Copyright 2026 DEEPX All Rights Reserved.

4.5 Profile Application

4.5.1 Extract Profiling Data Using `dxrun`

When you run a model with the `--profiler` option using `dxrun`, a `profiler.json` file is automatically generated in the working directory.

```
dxrun -m model.dxnn --profiler
# Check the generated profiler.json file
```

This provides a quick way to collect profiling data without modifying your application code.

4.5.2 Gather Timing Data per Event

You can profile events within your application using the Profiler APIs. Please refer to **Section. API reference**.

Enable the Profiler

Before collecting profiling data, enable the profiler and configure its output options.

```
// Enable the profiler
dxrt::Configuration::GetInstance().SetEnable(dxrt::Configuration::ITEM::PROFILER, tru
e);

// Set attributes to show data in console and save to a file
dxrt::Configuration::GetInstance().SetAttribute(dxrt::Configuration::ITEM::PROFILER,
    dxrt::Configuration::ATTRIBUTE::PROFILER_SHOW_DATA, "ON");

dxrt::Configuration::GetInstance().SetAttribute(dxrt::Configuration::ITEM::PROFILER,
    dxrt::Configuration::ATTRIBUTE::PROFILER_SAVE_DATA, "ON");
```

After the application finishes, `profiler.json` is created in the working directory.

Get Per-Job Metrics with `GetJobMetrics()`

Use `dxrt::Profiler::GetJobMetrics()` to retrieve detailed timing breakdowns for a specific inference job. This is the **recommended** way to access per-job inference timing data programmatically.

Call it immediately after `Wait()` returns with the corresponding job ID.

```
auto& profiler = dxrt::Profiler::GetInstance();
```


© Copyright 2026 DEEPX All Rights Reserved.

```
int jobId = ie.RunAsync(inputBuf.data());
auto outputs = ie.Wait(jobId);

auto jm = profiler.GetJobMetrics(jobId);
if (jm.valid) {
    // NPU task metrics (per device)
    auto npu = jm.GetTask("npu_0");
    if (npu.valid) {
        auto& dev = npu.devices[0];         // device 0 metrics
        double h2d      = dev.h2d_us;       // Host-to-Device transfer (µs)
        double inference = dev.inference_core_all_us; // NPU compute (µs)
        double d2h      = dev.d2h_us;       // Device-to-Host transfer (µs)
        double total    = dev.total_us;     // End-to-end NPU task time (µs)
    }

    // CPU task metrics (when USE_ORT is enabled)
    auto cpu = jm.GetTask("cpu_0");
    if (cpu.valid) {
        double cpuExec = cpu.cpu_task_us;   // CPU op execution time (µs)
    }
}
```

`JobMetrics` Structure

Field	Type	Description
`tasks`	`vector<TaskMetrics>`	One entry per task that participated in the job
`valid`	`bool`	`false` if the job ID was not found
`GetTask(name)`	method	Safe lookup by task name; returns `valid=false` if not found

`TaskMetrics` Structure

Field	Type	Description
`task_name`	`string`	Task name (e.g., `"npu_0"`, `"cpu_0"`)
`devices`	`map<int, NpuDeviceMetrics>`	Per-device NPU metrics (NPU tasks only)
`cpu_task_us`	`double`	CPU task execution time in µs (CPU tasks only; 0 for NPU tasks)

© Copyright 2026 DEEPX All Rights Reserved.

Field	Type	Description
`valid`	`bool`	`true` if this task was measured

`NpuDeviceMetrics` Structure

Field	Type	Description
`input_format_us`	`double`	NPU input format handler time (µs)
`h2d_us`	`double`	Host-to-Device PCIe transfer time (µs)
`inference_core_all_us`	`double`	NPU compute time across all cores (µs)
`inference_core_0_us`	`double`	NPU compute time on core 0 (µs)
`inference_core_1_us`	`double`	NPU compute time on core 1 (µs)
`inference_core_2_us`	`double`	NPU compute time on core 2 (µs)
`d2h_us`	`double`	Device-to-Host PCIe transfer time (µs)
`output_format_us`	`double`	NPU output format handler time (µs)
`total_us`	`double`	End-to-end NPU task time (µs)
`valid`	`bool`	`true` if at least one of H2D/NPU/D2H was measured

NOTE

`GetJobMetrics()` is the recommended API. The older `GetPerformanceData()` and `GetPerformanceDataByDevice()` methods are **deprecated** and will be removed in a future release.

© Copyright 2026 DEEPX All Rights Reserved.

Record Custom User Events

You can also record arbitrary named events in your application to include them in the profiler output.

```
auto& profiler = dxrt::Profiler::GetInstance();
profiler.Start("my_preprocess");
// ... preprocessing code ...
profiler.End("my_preprocess");
```

User events are included in `profiler.json` when saved via `Save(file, userEvents)`.

4.5.3 Visualize Profiler Data

You can visualize the profiling results using the following Python script.

```
python3 tool/profiler/plot.py -i profiler.json
```

This generates timeline chart image files from the profiling data. When multi-device profiling data is present, separate images are generated per device (e.g., `profiler_Device_0.png`, `profiler_Device_1.png`). If the number of jobs exceeds the limit (default: 200), the output is automatically split into multiple images per chunk (e.g., `profiler_Device_0_part1.png`).

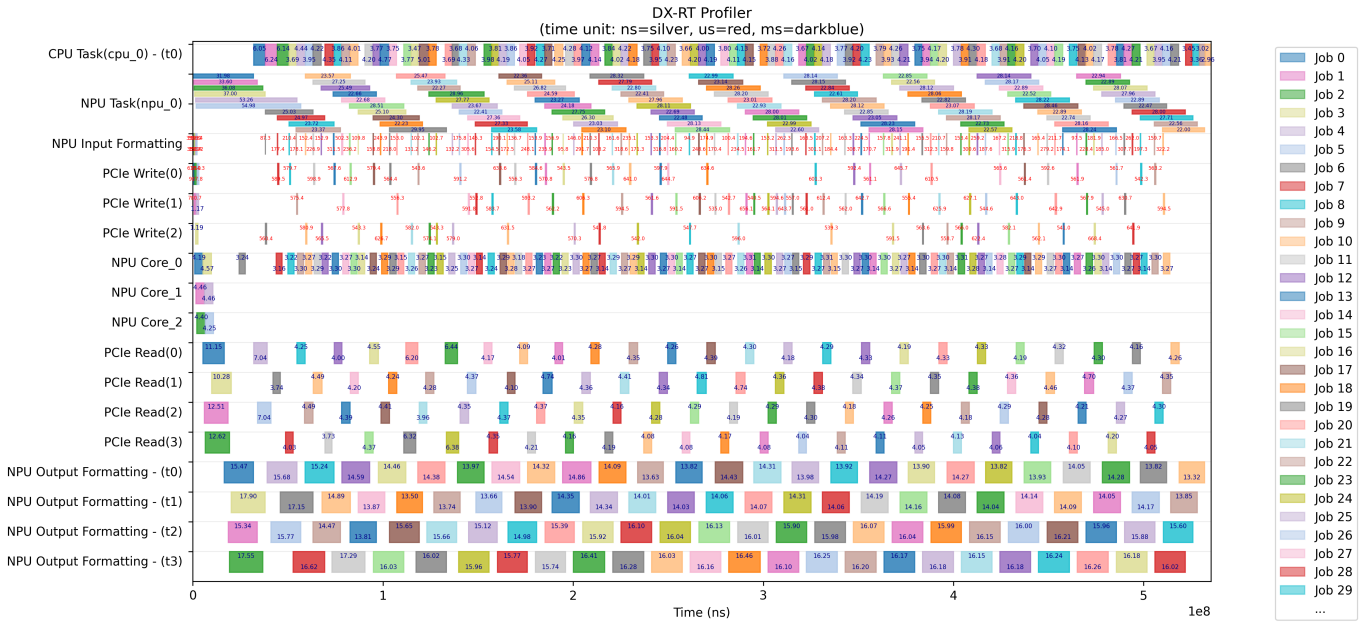

Figure. DX-RT Profiling Report

Script Usage: `tool/profiler/plot.py`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Use this script to draw a timeline chart from profiling data generated by **DX-RT**.

```
usage: plot.py [-h] [-i INPUT] [-o OUTPUT] [-s START] [-e END] [-j JOBS_PER_IMAGE] [-
t] [-a]
```

Optional Arguments

- `-h, --help` : Show help message and exit
- `-i INPUT, --input INPUT` : Input `.json` file to visualize
- `-o OUTPUT, --output OUTPUT` : Output base filename; device/job suffixes are added automatically (default: `profiler.png`)
- `-s START, --start START` : Start ratio (0.0–1.0) of the time window
- `-e END, --end END` : End ratio (0.0–1.0) of the time window
- `-j JOBS_PER_IMAGE, --jobs-per-image JOBS_PER_IMAGE` : Max jobs per image before splitting (default: 200)
- `-t, --show_text` : Show duration text labels on bars

> **NOTE**
>
> The `-a, --auto-select` option automatically selects 200 jobs from the stable centre region, which can be useful for filtering out warm-up and tail effects.

4.5.4 Profiler Events

The profiler records the following events during inference. Use this table to identify performance bottlenecks and apply the suggested remedies.

Event	Description	Bottleneck Remedy
Buffer Wait	Waiting for an available inference buffer	`export DXRT_DYNAMIC_CPU_THREAD=ON`
NPU Input Format Handler	Padding insertion and NHWC conversion for NPU-optimized data alignment	Enable data format conversion acceleration (see *Hardware-Accelerated Data Processing*)
PCIe Write	Transferring input data to NPU over PCIe	—

© Copyright 2026 DEEPX All Rights Reserved.

Event	Description	Bottleneck Remedy
NPU Core	NPU hardware computation	—
PCIe Read	Reading output data from NPU over PCIe	—
NPU Output Format Handler	Padding removal and NHWC→NCHW conversion of NPU output data	Enable data format conversion acceleration (see *Hardware-Accelerated Data Processing*)
CPU Task Queue Wait	Waiting in the CPU task queue before execution	Enable CPU op acceleration (see *Hardware-Accelerated Data Processing*)
cpu_N	CPU operator execution on thread N (e.g., `cpu_0`, `cpu_1`)	Enable CPU op acceleration (see *Hardware-Accelerated Data Processing*)

NOTE

NPU Task represents the total NPU processing time from input formatting through NPU computation to output formatting. It is the aggregate of NPU Input Format Handler, PCIe Write, NPU Core, PCIe Read, and NPU Output Format Handler.

4.6 RuntimeEventDispatcher

The `RuntimeEventDispatcher` is a singleton class that provides a centralized event dispatching mechanism for monitoring and handling runtime events from the DX-RT system. It enables applications to receive notifications about device errors, warnings, and operational events through custom event handlers.

4.6.1 Overview

Runtime events include device status changes, memory operations, core errors, and other operational notifications. The dispatcher supports different severity levels and event categories, allowing applications to filter and respond to events based on their importance.

© Copyright 2026 DEEPX All Rights Reserved.

4.6.2 Event Severity Levels

Events are categorized by severity using `RuntimeEventDispatcher::LEVEL`:

- **`LEVEL_INFO`** - Informational messages for normal operation events
- **`LEVEL_WARNING`** - Warning messages for potential issues that don't stop execution
- **`LEVEL_ERROR`** - Error messages for recoverable failures
- **`LEVEL_CRITICAL`** - Critical errors that may cause system instability

4.6.3 Event Types

Events are classified by their source using `RuntimeEventDispatcher::TYPE`:

- **`DEVICE_CORE`** - Events related to NPU core operations
- **`DEVICE_STATUS`** - Device status change events
- **`DEVICE_IO`** - Input/Output operation events
- **`DEVICE_MEMORY`** - Memory management events
- **`UNKNOWN`** - Unknown or unclassified event types

4.6.4 Event Codes

Specific event codes identify the exact nature of events using `RuntimeEventDispatcher::CODE`:

- **`WRITE_INPUT`** - Input data write operation event
- **`READ_OUTPUT`** - Output data read operation event
- **`MEMORY_OVERFLOW`** - Memory overflow or capacity exceeded
- **`MEMORY_ALLOCATION`** - Memory allocation failure or issue
- **`DEVICE_EVENT`** - General device event notification
- **`RECOVERY_OCCURRED`** - Device recovery action taken
- **`TIMEOUT_OCCURRED`** - Operation timeout event
- **`THROTTLING_NOTICE`** - Device throttling notification
- **`THROTTLING_EMERGENCY`** - Device throttling emergency notification
- **`UNKNOWN`** - Unknown or unclassified event code

© Copyright 2026 DEEPX All Rights Reserved.

4.6.5 Usage Example

Basic Event Handler Registration

```
#include <dxrt/dxrt_cxx_api.h>
#include <iostream>

int main()
{
    // Get the singleton instance
    auto& dispatcher = dxrt::RuntimeEventDispatcher::GetInstance();

    // Set the minimum event level threshold
    dispatcher.SetCurrentLevel(dxrt::RuntimeEventDispatcher::LEVEL::LEVEL_WARNING);

    // Register a custom event handler
    dispatcher.RegisterEventHandler(
        [](dxrt::RuntimeEventDispatcher::LEVEL level,
           dxrt::RuntimeEventDispatcher::TYPE type,
           dxrt::RuntimeEventDispatcher::CODE code,
           const std::string& message,
           const std::string& timestamp)
        {
            std::cout << "[" << timestamp << "] ";

            // Handle different severity levels
            switch (level) {
                case dxrt::RuntimeEventDispatcher::LEVEL::LEVEL_INFO:
                    std::cout << "INFO: ";
                    break;
                case dxrt::RuntimeEventDispatcher::LEVEL::LEVEL_WARNING:
                    std::cout << "WARNING: ";
                    break;
                case dxrt::RuntimeEventDispatcher::LEVEL::LEVEL_ERROR:
                    std::cout << "ERROR: ";
                    break;
                case dxrt::RuntimeEventDispatcher::LEVEL::LEVEL_CRITICAL:
                    std::cout << "CRITICAL: ";
                    break;
            }

            std::cout << message << std::endl;

            // Take action based on event type or code
            if (code == dxrt::RuntimeEventDispatcher::CODE::RECOVERY_OCCURRED) {
                std::cout << "Device recovery detected - reinitializing..." << std::e
ndl;

                // Implement recovery logic here
            }
        }
    );
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    // Your inference code here
    dxrt::InferenceEngine ie("model.dxnn");
    // ... perform inference ...

    return 0;
}
```

Advanced Event Filtering

```
// Example: Filter events by type and severity
dispatcher.RegisterEventHandler(
    [](dxrt::RuntimeEventDispatcher::LEVEL level,
       dxrt::RuntimeEventDispatcher::TYPE type,
       dxrt::RuntimeEventDispatcher::CODE code,
       const std::string& message,
       const std::string& timestamp)
    {
        // Only handle critical device core events
        if (level == dxrt::RuntimeEventDispatcher::LEVEL::LEVEL_CRITICAL &&
            type == dxrt::RuntimeEventDispatcher::TYPE::DEVICE_CORE)
        {
            std::cerr << "CRITICAL DEVICE ERROR: " << message << std::endl;
            // Log to file, send alert, etc.
        }

        // Monitor memory events
        if (type == dxrt::RuntimeEventDispatcher::TYPE::DEVICE_MEMORY)
        {
            std::cout << "Memory event: " << message << std::endl;
            // Track memory usage statistics
        }
    }
);
```

4.6.6 Key Features

- **Singleton Pattern** - Single instance accessible throughout the application
- **Thread-Safe** - Uses mutexes and atomic operations for concurrent access
- **Custom Handlers** - Register callback functions to process events
- **Event Filtering** - Set minimum severity level threshold
- **Automatic Logging** - Events are automatically logged with formatted output

4.6.7 Notes

- Only one event handler can be registered at a time; subsequent registrations replace the previous handler

© Copyright 2026 DEEPX All Rights Reserved.

- The event handler is invoked synchronously with minimal lock duration
- Events below the current level threshold may be filtered by custom handlers
- The dispatcher uses a copy-and-execute pattern to minimize mutex contention

4.7 How To Create an Application Using DX-RT

This guide provides step-by-step instructions for creating a new CMake project using the **DX-RT** library.

Step 1. Build the **DX-RT** Library
Before starting, make sure the **DX-RT** library is already built.

Refer to **Section. Installation on Linux** and **Section. Installation on Windows** for detailed build instructions.

Step 2. Create a New CMake Project
Create a project directory and an initial `CMakeLists.txt` file.

```
mkdir MyProject
cd MyProject
touch CMakeLists.txt
```

Step 3. "Hello World" with DX-RT API
Create a simple source file (`main.cpp`) that uses a **DX-RT** API.

```
#include <dxrt/dxrt_cxx_api.h>
using namespace std;

int main(int argc, char *argv[])
{
 auto& devices = dxrt::CheckDevices();
 cout << "hello, world" << endl;
 return 0;
}
```

Step 4. Modify `CMakeLists.txt`
Edit the `CMakeLists.txt` file as follows.

```
cmake_minimum_required(VERSION 3.14)
project(app_template)

set(CMAKE_CXX_STANDARD_REQUIRED "ON")
set(CMAKE_CXX_STANDARD "14")
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
# Set the DX-RT library installation path (adjust as needed)
set(DXRT_LIB_PATH "/usr/local/lib")

# Locate the DX-RT library
find_library(DXRT_LIBRARY REQUIRED NAMES dxrt_${CMAKE_SYSTEM_PROCESSOR} PATHS $
{DXRT_LIB_PATH})

# Add executable and link libraries
add_executable(HelloWorld main.cpp)
target_link_libraries(HelloWorld PRIVATE ${DXRT_LIBRARY} protobuf)
```

Replace `/usr/local/lib` with the actual path where the **DX-RT** library is installed.

Step 5. Build the Project
Compile your project using the following commands.

```
mkdir build
cd build
cmake ..
make
```

Step 6. Run the Executable
After a successful build, run the generated executable.

```
./HelloWorld
```

You now successfully create and build a CMake project using the **DX-RT** library.

4.8 (Optional) Improving Inference Throughput

The `USE_ORT` enables the use of ONNX Runtime to handle operations that are **not** supported by the NPU. When this option is active, CPU-based execution is applied for the unsupported subgraphs of the model via ONNX Runtime.

Beyond this CPU fallback, **DX-RT** provides two optional features to further improve inference throughput when CPU-side tasks become a bottleneck: **hardware-accelerated data processing** and **dynamic CPU threading**.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

4.8.1 Hardware-Accelerated Data Processing

DX-RT can leverage platform-specific acceleration libraries to speed up two key CPU-side tasks in the inference pipeline:

- **Data Format Conversion Acceleration** — Depending on the compiled model, input data may need to be transposed and padded into the NPU-specific layout, and output data may need to be converted back to the host format. When this conversion occurs, it runs on every affected input and output tensor and can become a significant per-frame overhead. DX-RT can accelerate these operations using platform-optimized libraries. Internally, this conversion stage is called the **NPU Format Handler (NFH)**.
- **CPU Op Acceleration** — Uses optimized execution providers for ONNX Runtime CPU tasks (operations not supported by the NPU), replacing the default CPU provider with a hardware-tuned alternative.

Platform Support

Acceleration Target	x86_64	aarch64
Data Format Conversion (NFH)	Intel IPP	ARM NEON
CPU Op (ORT EP)	OpenVINO EP	XNNPACK EP

Supported Environments

Architecture	OS	Status
x86_64	Ubuntu 22.04 – 24.04	Supported
aarch64	Ubuntu 20.04 – 24.04	Supported
x86_64 / aarch64	Windows	Planned

When to Use

- **Data Format Conversion (NFH)**: Effective when the profiler shows that NPU Input/Output Format Handler events occupy a significant portion of the inference time.
- **CPU Op**: Effective only when the CPU-fallback subgraphs contain **computation-heavy operations** such as Conv or MatMul. The acceleration libraries (OpenVINO, XNNPACK) primarily optimize arithmetic-intensive operations; memory-bound operations like Reshape, Transpose, or Concat will see little to no improvement. Simply having long CPU processing time does not guarantee acceleration — the benefit depends on the **type of operations** in the CPU subgraphs.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

You can use the profiler (see *Profile Application* section) to identify whether data format conversion or CPU tasks are bottlenecks. If NPU computation dominates the pipeline, enabling acceleration will have minimal effect.

Benchmark Reference

Test Environment

Item	Detail
Host	Orange Pi 5 Plus (aarch64)
NPU	DEEPX DX-M1 with PCIe Gen3 x4
OS	Ubuntu 22.04
DX-RT Version	v3.3.0
Model	Q-PRO variants from DEEPX ModelZoo
Measurement Tool	`dxrun` (no pre/post-processing)
Iterations	2000
Build Option	`USE_ORT=ON`

Results (FPS)

Model	Baseline	NFH Only	CPU Op Only	Both
YoloV8S	84	86	128	**131**
YoloV10S	68	68	96	**98**
YoloXS	250	288	258	**288**
YoloV11S	82	82	127	**131**

NOTE

These numbers are for reference only. Actual performance depends on device conditions, input resolution, and system load. When pre/post-processing is added to the application pipeline, end-to-end FPS may differ from the values shown here.

© Copyright 2026 DEEPX All Rights Reserved.

Step 1. Build with Acceleration Support

Acceleration support is **not included in the default build**. To enable it, edit `cmake/dxrt.cfg.cmake` and set the following options to `ON`:

```
# cmake/dxrt.cfg.cmake
option(USE_NPU_FORMAT_CONVERSION_ACCELERATION "..." ON)   # default: OFF
option(USE_CPU_OP_ACCELERATION "..." ON)                   # default: OFF
```

Build Option	Default	Description
`USE_NPU_FORMAT_CONVERSION_ACCELERATION`	`OFF`	Include data format conversion (NFH) acceleration code. Automatically disabled if the platform lacks the required library (e.g., Intel IPP on x86_64).
`USE_CPU_OP_ACCELERATION`	`OFF`	Include CPU op acceleration code. Automatically disabled if the required EP library is unavailable (e.g., OpenVINO on x86_64).

After changing the options, perform a **clean build**:

```
./build.sh --clean
```

> **IMPORTANT**
>
> A clean build is required after changing these options. Incremental builds may not pick up the change correctly.

> **NOTE**
>
> The build system automatically downloads and installs the required acceleration libraries (Intel IPP, OpenVINO, XNNPACK, etc.) during the clean build. **An internet connection is required** at build time. You do not need to install these libraries manually.
> If a library cannot be obtained (e.g., due to network restrictions or platform incompatibility), the corresponding acceleration option is disabled even when set to `ON`.

Step 2. Enable at Runtime

© Copyright 2026 DEEPX All Rights Reserved.

Even after building with acceleration support, the feature is **disabled at runtime by default**. To activate it, use the Configuration API in your application:

C++ Example

```cpp
#include <dxrt/dxrt_cxx_api.h>

auto& config = dxrt::Configuration::GetInstance();
config.SetEnable(dxrt::Configuration::ITEM::NFH_ACCELERATION, true);
config.SetEnable(dxrt::Configuration::ITEM::CPU_OP_ACCELERATION, true);
```

Python Example

```python
from dx_engine import Configuration

config = Configuration()
config.set_enable(Configuration.ITEM.NFH_ACCELERATION, True)
config.set_enable(Configuration.ITEM.CPU_OP_ACCELERATION, True)
```

TIP

To quickly test the effect of acceleration without modifying application code, you can use the `dxrun` CLI tool with the `--accel-nfh` and `--accel-cpu` options:

```
dxrun -m model.dxnn --accel-nfh --accel-cpu
```

Build Dependency

The acceleration APIs (`ITEM::NFH_ACCELERATION` , `ITEM::CPU_OP_ACCELERATION`) and CLI options (`--accel-nfh` , `--accel-cpu`) are only available when the corresponding CMake option is set to `ON` at build time.
If the feature was **not** compiled in (default `OFF` build), these enum values and CLI options **do not exist**: using them will result in a compilation error (C++) or an "unrecognized arguments" error (CLI).

WARNING

Enabling acceleration does **not** guarantee a performance improvement for every model. The effect depends on the model structure, operation types, and host platform. In particular, CPU op acceleration primarily benefits arithmetic-heavy operations (Conv, MatMul); memory-bound operations (Reshape, Concat) may see little improvement.

© Copyright 2026 DEEPX All Rights Reserved.

4.8.2 Improving CPU Capacity with Dynamic Threading

When executing CPU task via ONNX Runtime, performance bottlenecks may arise depending on the Host CPU performance and symbol load. To address this, **DX-RT** provides an optional dynamic multi-threading feature that can improve throughput in high-load scenarios.

Feature Overview

- Dynamically increases the number of threads allocated to ONNX Runtime tasks
- Monitors the input queue load to determine CPU congestion
- Designed to boost FPS when CPU-bound tasks become a bottleneck

Enabling Dynamic CPU Threading

To enable this feature, set the following environment variable:

```
export DXRT_DYNAMIC_CPU_THREAD=ON
```

This activates internal logic to automatically adjust the ONNX Runtime thread pool size based on queue pressure.

> **NOTE**
>
> When high CPU task load is detected at runtime, the system may print the following message:
>
> ```
> To improve FPS, set: 'export DXRT_DYNAMIC_CPU_THREAD=ON'
> ```
>
> This serves as a recommendation to enable the feature for improved inference performance.

> **WARNING**
>
> Enabling the `DXRT_DYNAMIC_CPU_THREAD=ON` option does **not** guarantee an FPS improvement in all cases. The effectiveness of this feature depends on the specific workload, input size, and CPU capacity of the system.

© Copyright 2026 DEEPX All Rights Reserved.

5. Command Line Interface

This chapter introduces the **DX-RT** command-line tools for model inspection, inference testing, and device management. These tools allow developers to validate models, measure performance, and control hardware directly from the terminal.

These utilities are essential for validating model compilation, tuning inference performance, and maintaining the health and configuration of the underlying NPU hardware.

5.1 Legacy Command Aliases

The legacy command names from earlier releases are preserved as aliases and continue to work.

New	Legacy alias
`dxparse`	`parse_model`
`dxrun`	`run_model`
`dxcli`	`dxrt-cli`

5.2 Parse Model

This tool parses a compiled `.dxnn` model file and displays detailed information such as model structure, input/output tensors, and metadata.

Source

```
cli/dxparse.cpp
```

Usage

```
dxparse -m <model_dir>
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Option

```
  -m, --model FILE    model path (required)
  -v, --verbose       show detailed task dependencies and memory usage
  -o, --output FILE   save the raw console output to a file (without color codes)
  -j, --json          extract JSON binary data (graph_info, rmap_info) to files
  -h, --help          show this help message
```

Example

```
$./dxparse -m model.dxnn
```

5.3 Run Model

This tool executes a compiled model to verify basic functionality, measure inference time, and optionally perform repeated runs for stress testing.

Source

```
cli/dxrun.cpp
```

Usage

```
dxrun -m <model_dir> [-l <number of loops>] [-t <seconds>] [-d <devices>] [--use-ort]
[--profiler] [--buffer-count <N>]
```

Option

```
  -m, --model arg         Model file (.dxnn)
  -b, --benchmark         Perform a benchmark test (Maximum throughput)
                          (This is the default mode,
                           if --single or --fps > 0 are not specified)
  -s, --single            Perform a single run test
                          (Sequential single-input inference on a
                          single-core)
  -v, --verbose           Shows NPU Processing Time and Latency
  -n, --npu arg           NPU bounding (default:0)
                            0: NPU_ALL
                            1: NPU_0
                            2: NPU_1
                            3: NPU_2
                            4: NPU_0/1
                            5: NPU_1/2
                            6: NPU_0/2
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  -l, --loops arg         Number of inference loops to perform (default:
                          30)
  -t, --time arg          Duration of inference in seconds (benchmark and
                          target-fps mode, overrides --loops) (default: 0)
  -w, --warmup-runs arg   Number of warmup runs before actual measurement
                          (default: 0)
  -d, --devices arg       Specify target NPU devices.
                          Examples:
                            'all' (default): Use all available/bound NPUs
                            '0': Use NPU0 only
                            '0,1,2': Use NPU0, NPU1, and NPU2
                            'count:N': Use the first N NPUs
                            (e.g., 'count:2' for NPU0, NPU1) (default: all)
  -f, --fps arg           Target FPS for TARGET_FPS_MODE (default: 0)
                          (enables this mode if > 0 and --single is not
                          set)
      --use-ort           Enable ONNX Runtime for CPU tasks in the model
                          graph
                          If disabled, only NPU tasks operate
      --profiler          Enable profiler
      --accel-nfh         Enable NFH (transpose) acceleration
                          (if available)
      --accel-cpu         Enable CPU op acceleration (OpenVINO/XNNPACK)
                          (if available)
      --buffer-count arg  Number of input/output buffers, count's range is
                          1~100 (default: 6)
  -h, --help              Print usage
```

Examples

Basic inference with 100 loops:

```
dxrun -m /.../model.dxnn -i /.../input.bin -l 100
```

Enable profiler to collect detailed timing data:

```
dxrun -m model.dxnn --profiler -l 50
```

This generates a `profiler.json` file in the working directory containing detailed performance metrics.

Configure buffer count for optimized throughput:

```
dxrun -m model.dxnn --buffer-count 8 -l 100
```

Adjust the number of input/output buffers (range: 1-100) to balance memory usage and inference throughput. Higher buffer counts can improve performance in pipelined scenarios.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Combined options with profiling and custom buffer count:

```
dxrun -m model.dxnn --profiler --buffer-count 10 -l 200 -v
```

Benchmark mode with profiling enabled:

```
dxrun -m model.dxnn --benchmark --profiler -t 60
```

Run benchmark for 60 seconds with profiling enabled to analyze performance bottlenecks.

5.4 DX-RT CLI Tool (Firmware Interface)

This tool provides a CLI for interacting with **DX-RT** accelerator devices, supporting device status queries, hardware resets, and firmware updates.

> **NOTE**
>
> This tool is applicable **only** for accelerator devices.

Usage

```
dxcli <option> <argument>
```

Option

```
  -s, --status            Get device status
  -i, --info              Get device info
  -m, --monitor arg       Monitoring device status every [arg] seconds
                          (arg > 0)
  -r, --reset [=arg(=0)]  Reset device(0: reset only NPU) (default: 0)
  -d, --device arg        Device ID (if not specified, CLI commands will
                          be sent to all devices.) (default: -1)
  -u, --fwupdate arg      Update firmware with deepx firmware file.
                          sub-option : [force:force update, unreset:device
                          unreset(default:reset)]
  -g, --fwversion arg     Get firmware version with deepx firmware file
  -C, --fwconfig_json arg Update firmware settings from [JSON]
  -v, --version           Print minimum versions
  -h, --help              Print usage
```


© Copyright 2026 DEEPX All Rights Reserved.

Example

```
dxcli -s
dxcli --status

dxcli -r
dxcli -r=1
dxcli --reset
dxcli --reset=1

dxcli --fwupdate fw.bin
dxcli -u fw.bin

dxcli -m 1
```

NOTE

Firmware update issue
If an issue occurs during firmware update with `dxcli -u` or `dxcli --fwupdate`, run the following sequence:

```
# service off
sudo systemctl disable dxrt.service
sudo systemctl stop dxrt.service

dxcli -u fw.bin

# service on
sudo systemctl enable dxrt.service
sudo systemctl start dxrt.service
```

5.5 NPU Monitoring Utility

DX-TOP is a real-time command-line utility designed to monitor the status, utilization, and health metrics of Neural Processing Unit (NPU) devices managed by the DX-Runtime. It provides critical information necessary for performance tuning, debugging, and system health checks.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5.5.1 Utility Versions (Header Information)

The top section of the screen displays the version of the core software components.

Item	Description
DX-RT	**DX-Runtime Version.** The version of the core software stack that manages NPU operations.
NPU Device Driver	**NPU Driver Version.** The version of the operating system driver responsible for hardware communication.
DX-TOP	**Monitoring Tool Version.** The version of the DX-TOP utility itself.

5.5.2 Device Status Monitoring

DX-TOP organizes information by individual NPU devices installed in the system.

Device Hardware Information
This section identifies the physical hardware and firmware configuration of the NPU card.

Item	Description
Device ID	The unique identifier assigned to the NPU card (e.g., 'Device :0').
Variant	The model or variant of the NPU hardware (e.g., `M1`).
PCIE Bus Number	The PCI Express bus location of the device (e.g., `02:00:00`). This is crucial for hardware-level identification.
Firmware	The version of the embedded firmware running on the NPU hardware.
NPU Memory	The memory utilization of the on-board dedicated NPU memory. Displays Used MiB / Total GiB (Usage %).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Core Performance Metrics

Each device contains multiple processing Cores. This section displays real-time operational metrics for each core.

Item	Description
Core ID	The identifier for the individual processing core within the NPU (e.g., `Core :0`).
Util	**Utilization (Percentage).** The percentage of time the core is actively processing NPU tasks. High utilization (near 100%) indicates high workload.
Temp	**Temperature (Degrees Celsius).** The current operating temperature of the core. Critical for checking for overheating and thermal throttling.
Voltage	**Voltage (Millivolts).** The electrical voltage currently being supplied to the core. Reflects power management status.
Clock	**Clock Rate (Megahertz).** The current operating frequency of the core. Directly impacts processing speed and performance.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

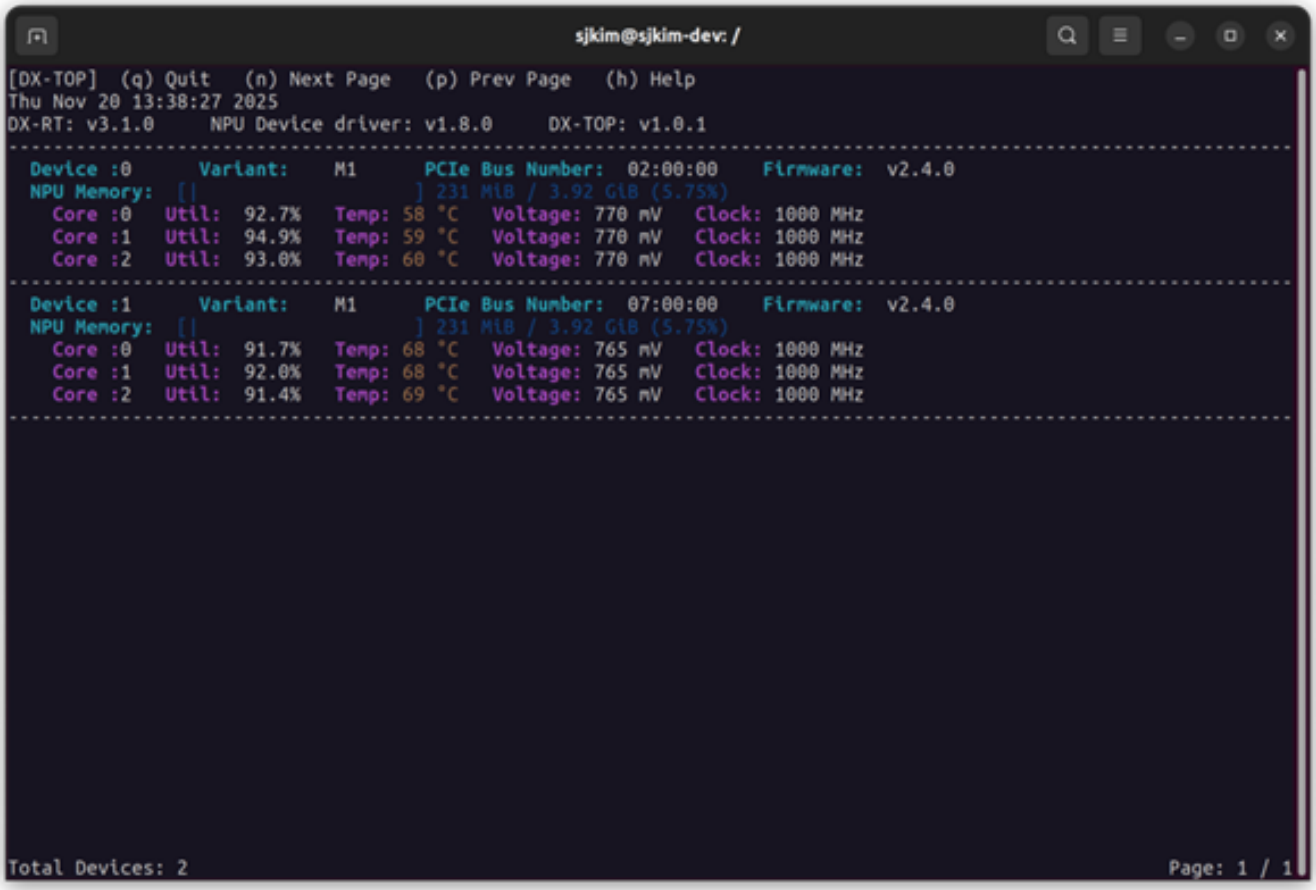

Figure. Example of DX-TOP Monitoring Utility Screen

5.5.3 Command Controls (Help Menu)

The following commands are available at the top of the DX-TOP interface for navigation and control.

Command	Action
(q) Quit	Exits the DX-TOP application.
(n) Next Page	Navigates to the next page of device information (if more devices are present than can fit on one screen).
(p) Prev Page	Navigates to the previous page of device information.
(h) Help	Displays the help screen or a summary of available commands.

© Copyright 2026 DEEPX All Rights Reserved.

5.6 Performance Benchmarking Utility

The dxbenchmark tool is a CLI utility designed to automate the performance benchmarking of deep learning models and generate detailed, visualized reports.

- **Comprehensive Execution:** Runs all detectable models in the target directory, including sub-directories (`via --recursive`).
- **Detailed Reporting:** Generates a single, integrated HTML document with visualizations of key performance indicators (KPIs) and stability metrics.
- **Stability Analysis:** Includes time-series data visualization (Performance Metrics Over Loops) and the Coefficient of Variation (CV) for jitter analysis.
- **Raw Data for Comparison:** Outputs version-tagged JSON/CSV files for easy comparison across different Runtime and Driver versions.

5.6.1 CLI Guide

The benchmark tool is executed using the `dxbenchmark` command. Running the build script `build.sh` creates the binary named `dxbenchmark` in `dx_rt/bin` .

```
DXRT v3.1.0 dxbenchmark
Usage:
  dxbenchmark [OPTION...]

      --dir arg          Model directory
      --result-path arg  Destination of result file (default: .)
      --sort arg         Sorting criteria
                           name: Model Name
                           fps: FPS
                           time: NPU Inference Time
                           latency: Latency (default: name)
      --order arg        Sorting order
                           asc: Ascending order
                           desc: Descending order (default: asc)
  -l, --loops arg        Number of inference loops to perform (default: 0)
  -t, --time arg         Time duration to perform (default: 0)
      --warmup arg       Warmup time (default: 10)
  -n, --npu arg          NPU bounding (default:0)
                           0: NPU_ALL
                           1: NPU_0
                           2: NPU_1
                           3: NPU_2
                           4: NPU_0/1
                           5: NPU_1/2
                           6: NPU_0/2
  -d, --devices arg      Specify target NPU devices.
```


© Copyright 2026 DEEPX All Rights Reserved.

```
                        Examples:
                          'all' (default): Use all available/bound NPUs
                          '0': Use NPU0 only
                          '0,1,2': Use NPU0, NPU1, and NPU2
                          'count:N': Use the first N NPUs
                          (e.g., 'count:2' for NPU0, NPU1) (default: all)
    --only-data         Make only data result (csv, json)
    --recursive         Search models recursively in subdirectories
                        (default: base directory only)
    --use-ort           Enable ONNX Runtime for CPU tasks in the model
                        graph
                        If disabled, only NPU tasks operate
-v, --verbose           Shows NPU Processing Time and Latency
-h, --help              Print usage
```

5.6.2 Basic Usage and Syntax

The execution requires specifying the model source directory and one of the execution termination conditions (`-t` or `-l`).

Execution Commands
After building, you can execute the tool directly using the dxbenchmark command without navigating into the dx_rt/bin directory.

Execution Method	Syntax	Description
Direct Path	`./dx_rt/bin/dxbenchmark [OPTION...]`	Executes by directly specifying the path to the generated binary
Direct Execute (Recommended)	`dxbenchmark [OPTION...]`	Executes directly using the name registered in the environment

Mandatory Parameters

Execution Method	Format	Description
Model Directory	`--dir {Directory}`	The path to the directory containing the models.
Termination Condition	`-t {Time}`	Time duration in seconds. This option is currently prioritized over `-l.`
Loop	`-l {Loops}`	Number of inference loops.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> **Execution Termination Condition Priority**
> Currently, the execution duration -t {Time} is prioritized . While running based on time is generally suitable for benchmarking, large variations in inference time between models can lead to distortion in graph results.
> - `--dir {Directory}` : The path to the directory containing the models.
> - `EITHER -t {Time}` : Time duration in seconds (takes precedence).
> - `OR -l {Loops}` : Number of inference loops.

Execution Examples

```
# Example 1: Run models in the 'models/' directory for 60 seconds
./dxbenchmark --dir models/ -t 60

# Example 2: Run recursively, sort by latency descending, target NPU 0 and 1
./dxbenchmark --dir models/ --recursive --sort latency --order desc -d '0,1'
```

5.6.3 Output Files and Reporting

The tool generates a highly structured HTML report for visualization and version-controlled raw data files.

HTML Report Structure and Visualization

- **Filename Format:** `DXBENCHMARK_{YYYY_MM_DD_HHMMSS}.html`
- **Example:** `DXBENCHMARK_2025_08_13_174251.html`

Raw Data Files

- **Purpose:** To store the underlying data for version comparison and external analysis.
- **Filename Format:** `rt_{Runtime Version}_dd_{Device Driver Version}_pd_{PCIe Driver Version}.json/csv`
- **Example:** `rt_3.0.0_dd_1.7.1_pd_1.4.1.json`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

6. Inference API Guide

This chapter introduces the inference APIs provided by **DX-RT**, including both C++ and Python interfaces. It covers synchronous and asynchronous execution, support for single and multi-input models, and guidance on input/output formatting. Key topics include model execution, input parsing, special handling cases, and performance tuning to help developers integrate inference efficiently.

> **Recommended Header**
>
> Include `<dxrt/dxrt_cxx_api.h>` — a single, self-contained C++14 header that provides the full inference API. Existing code using `dxrt_api.h` continues to build without changes.
> Do not include `<dxrt/dxrt_api.h>` and `<dxrt/dxrt_cxx_api.h>` in the same translation unit.

6.1 C++ Inference API

This section describes the C++ interface for executing inference using the **DX-RT** SDK. It covers both synchronous and asynchronous execution modes, and provides guidance on initializing the engine, managing input/output buffers, and utilizing multiple NPU cores. These APIs are optimized for performance-critical environments and offer granular control over execution flow.

6.1.1 Running Synchronous Inference

This section covers the blocking inference methods in the **DX-RT** C++ API. It includes single-input, batch, and multi-input execution using dictionary or vector formats. These APIs return results after completion and are suited for real-time or latency-sensitive applications.

Run (Single Input/Output)

The `Run()` function provides a synchronous inference interface for single-frame execution. It accepts a raw input pointer and returns a vector of output tensors. The behavior of this API varies depending on the model type and how the input buffer is formatted.

© Copyright 2026 DEEPX All Rights Reserved.

```
TensorPtrs Run(void *inputPtr, void *userArg = nullptr, void *outputPtr = nullptr)
```

Input Format	Description	Model Type	Output Format	Notes
`void* inputPtr`	Single input pointer	Single-Input	`TensorPtrs` (Vector)	Traditional method
`void* inputPtr`	Concatenated buffer pointer	Multi-Input	`TensorPtrs` (Vector)	Auto-split applied

Example

```
// Single input model
auto outputs = ie.Run(inputData);

// Multi-input model (auto-split)
auto outputs = ie.Run(concatenatedInput);
```

In both cases, the output is returned as a vector of TensorPtr, representing each output tensor.

Run (Batch)

The `Run()` batch variant enables synchronous batched inference for both single-input and multi-input models. It accepts a flat list of input pointers, optional output buffers, and optional user arguments. The input vector is interpreted based on model type and size.

```
std::vector<TensorPtrs> Run(
    const std::vector<void*>& inputBuffers,
    const std::vector<void*>& outputBuffers,
    const std::vector<void*>& userArgs
)
```

Input Format	Condition	Interpretation	Output Format	Notes
`vector<void*>` (size=1)	Single-Input	Single Inference	`vector<TensorPtrs>` (size=1)	Special case
`vector<void*>` (size=N)	Single-Input	Batch Inference	`vector<TensorPtrs>` (size=N)	N samples
`vector<void*>` (size=M)	Multi-Input, M==input_count	Single Inference	`vector<TensorPtrs>` (size=1)	Multi-input single
		Batch Inference		

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Input Format	Condition	Interpretation	Output Format	Notes
`vector<void*>` (size=N)	Multi-Input, each pointer = concatenated M inputs		`vector<TensorPtrs>` (size=N)	N samples; per-sample buffers must concatenate all M inputs (total size = `GetInputSize()`)

Example

```
// Single input batch
std::vector<void*> batchInputs = {sample1, sample2, sample3};
auto batchOutputs = ie.Run(batchInputs, outputBuffers, userArgs);

// Multi-input single
std::vector<void*> multiInputs = {input1, input2}; // M=2
auto singleOutput = ie.Run(multiInputs, {outputBuffer}, {userArg});

// Multi-input batch — each pointer holds one sample with all inputs concatenated
// (concatenated buffer size per sample = ie.GetInputSize())
std::vector<void*> multiBatch = {sample1_concat, sample2_concat}; // N=2
auto batchOutputs = ie.Run(multiBatch, outputBuffers, userArgs);
// Note: Flat N*M layout (e.g., {s1_i1, s1_i2, s2_i1, s2_i2}) is supported by the
// Python binding only. Use RunMultiInput() or concatenated buffers in C++.
```

RunMultiInput (Dictionary)

The `RunMultiInput()` function provides a synchronous interface for multi-input models using a dictionary-style input. Each input tensor is mapped by name, allowing for flexible and explicit data assignment.

```
TensorPtrs RunMultiInput(
    const std::map<std::string, void*>& inputTensors,
    void *userArg = nullptr,
    void *outputPtr = nullptr
)
```

Input Format	Constraints	Output Format	Notes
`map<string, void*>`	Must include all input tensor names	`TensorPtrs`	For multi-input models only

© Copyright 2026 DEEPX All Rights Reserved.

Example

```
std::map<std::string, void*> inputs = {
    {"input1", data1},
    {"input2", data2}
};
auto outputs = ie.RunMultiInput(inputs);
```

RunMultiInput (Vector)

```
TensorPtrs RunMultiInput(
    const std::vector<void*>& inputPtrs,
    void *userArg = nullptr,
    void *outputPtr = nullptr
)
```

Input Format	Constraints	Output Format	Notes
`vector<void*>`	size == input_tensor_count	`TensorPtrs`	Order matches GetInputTensorNames()

6.1.2 Running Asynchronous Inference

This section describes the non-blocking inference functions in the C++ API. Asynchronous methods support parallel execution across NPU cores and allow synchronization via callbacks or `Wait()`. They are optimized for high-throughput and multi-threaded workloads.

RunAsync (Single)

The `RunAsync()` function initiates a non-blocking inference job using a single input pointer. It supports both single-input and multi-input models. Upon submission, the function returns a job ID, which is later used with the `Wait()` function to retrieve the result.

```
int RunAsync(void *inputPtr, void *userArg = nullptr, void *outputPtr = nullptr)
```

Input Format	Model Type	Output Format	Notes
`void* inputPtr`	Single-Input	`int` (jobId)	Result received via Wait(jobId)
`void* inputPtr`	Multi-Input	`int` (jobId)	Auto-split applied

© Copyright 2026 DEEPX All Rights Reserved.

RunAsync (Vector)

The `RunAsync()` vector-based variant provides non-blocking inference using an explicit list of input pointers. It is especially suitable for multi-input models, where each input tensor is provided as a separate pointer in the vector.

```
int RunAsync(const std::vector<void*>& inputPtrs, void *userArg = nullptr, void *outp
utPtr = nullptr)
```

Input Format	Condition	Interpretation	Output Format	Notes
`vector<void*>` (size==input_count)	Multi-Input	Multi-input single	`int` (jobId)	Recommended method
`vector<void*>` (size!=input_count)	Any	Uses only the first element	`int` (jobId)	Fallback

RunAsyncMultiInput (Dictionary)

The `RunAsyncMultiInput()` function performs non-blocking asynchronous inference for multi-input models, where each input tensor is provided in a named dictionary. This is the most explicit and type-safe method for specifying inputs, ensuring each tensor is matched to the correct model input.

```
int RunAsyncMultiInput(
    const std::map<std::string, void*>& inputTensors,
    void *userArg = nullptr,
    void *outputPtr = nullptr
)
```

Input Format	Constraints	Output Format	Notes
`map<string, void*>`	For multi-input models only	`int` (jobId)	Most explicit method

This API is intended only for multi-input models. If used with single-input models, behavior is undefined.

RunAsyncMultiInput (Vector)

© Copyright 2026 DEEPX All Rights Reserved.

The `RunAsyncMultiInput()` function performs asynchronous inference for multi-input models using a vector of input pointers. Each element in the vector corresponds to one input tensor, and the function returns a job ID to retrieve results later via `Wait()`.

```
int RunAsyncMultiInput(
    const std::vector<void*>& inputPtrs,
    void *userArg = nullptr,
    void *outputPtr = nullptr
)
```

Input Format	Constraints	Output Format	Notes
`vector<void*>`	size == input_tensor_count	`int` (jobId)	Converted to a dictionary internally

This method requires the input vector to be in the same order as the model's input tensor definitions. You can retrieve the correct order using `GetInputTensorNames()`.

Wait

The `Wait()` function blocks the current thread until the specified asynchronous inference job, identified by jobId, is completed. Once the job finishes, the function returns the corresponding inference output.

```
TensorPtrs Wait(
    int jobId
)
```

6.2 Python Inference API

This section outlines the Python API for running inference with the **DX-RT** SDK. Designed for ease of use and rapid development, the Python interface supports both synchronous and asynchronous execution, and offers flexible input handling for single, batch, and multi-input models. It is ideal for prototyping and integration into Python-based AI workflows.

6.2.1 Running Synchronous Inference

This section introduces the blocking inference methods in the Python API. These functions support single and multi-input models using a unified or dictionary-based interface. Results are returned after execution, making them suitable for sequential or real-time use.

run (Unified API)

© Copyright 2026 DEEPX All Rights Reserved.

The run() function is the primary Python API for synchronous inference. It supports both single-input and multi-input models, with automatic detection of batch size and input format. This unified interface simplifies inference for a variety of model types and use cases.

```
def run(
    input_data: Union[np.ndarray, List[np.ndarray], List[List[np.ndarray]]],
    output_buffers: Optional[Union[List[np.ndarray], List[List[np.ndarray]]]] = None,
    user_args: Optional[Union[Any, List[Any]]] = None
) -> Union[List[np.ndarray], List[List[np.ndarray]]]
```

Detailed Input/Output Matrix

Input: `np.ndarray`

- **Multi-Input Model** (`size == total_input_size`)

  Interpretation: Auto-split single

  Output: `List[np.ndarray]` (Single sample output)

- **Single-Input Model** (`size != total_input_size`):

  Interpretation: Single Inference

  Output: `List[np.ndarray]` (Single sample output)

© Copyright 2026 DEEPX All Rights Reserved.

Input: `List[np.ndarray]`

- **Single-Input Model** (`len == 1`)

  Interpretation: Single Inference

  Output: `List[np.ndarray]` (Single sample output)

- **Multi-Input Model** (`len == input_count`)

  Interpretation: Single Inference

  Output: `List[np.ndarray]` (Single sample output)

- **Multi-Input Model** (`len == N * input_count`)

  Interpretation: Batch Inference (N samples)

  Output: `List[List[np.ndarray]]` (N sample outputs)

- **Single-Input Model** (`len > 1`)

  Interpretation: Batch Inference

  Output: `List[List[np.ndarray]]` (`len` sample outputs)

Input: `List[List[np.ndarray]]`

- **Any Model** (Explicit batch)

  Interpretation: Batch Inference

  Output: `List[List[np.ndarray]]` (Matches outer list size)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Auto-split Special Cases

- **Multi-input + first element is total_size** (e.g., `[concatenated_array]`)

  Interpretation: Auto-split single

  Output: `List[np.ndarray]`

- **Multi-input + all elements are total_size** (e.g., `[concat1, concat2, concat3]`)

  Interpretation: Auto-split batch

  Output: `List[List[np.ndarray]]`

Example

```
# 1. Single array auto-split (multi-input)
concatenated = np.zeros(ie.get_input_size(), dtype=np.uint8)
outputs = ie.run(concatenated)  # List[np.ndarray]

# 2. Multi-input single
input_list = [input1_array, input2_array]  # len == 2
outputs = ie.run(input_list)  # List[np.ndarray]

# 3. Multi-input batch (flattened)
flattened = [s1_i1, s1_i2, s2_i1, s2_i2]  # 2 samples, 2 inputs each
outputs = ie.run(flattened)  # List[List[np.ndarray]], len=2

# 4. Multi-input batch (explicit)
explicit_batch = [[s1_i1, s1_i2], [s2_i1, s2_i2]]
outputs = ie.run(explicit_batch)  # List[List[np.ndarray]], len=2

# 5. Single-input batch
single_batch = [sample1, sample2, sample3]
outputs = ie.run(single_batch)  # List[List[np.ndarray]], len=3
```

run_multi_input (Dictionary)

The `run_multi_input()` function performs synchronous inference using a dictionary format for input, where each key-value pair maps an input tensor name to a corresponding NumPy array. This method is ideal for multi-input models, offering clarity and input safety through name matching.

```
def run_multi_input(
    input_tensors: Dict[str, np.ndarray],
    output_buffers: Optional[List[np.ndarray]] = None,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    user_arg: Any = None
) -> List[np.ndarray]
```

Input Type	Constraints	Output Type	Notes
`Dict[str, np.ndarray]`	Must include all input tensors	`List[np.ndarray]`	For multi-input models only

6.2.2 Running Asynchronous Inference

This section covers the non-blocking inference methods in the Python API. These functions support parallel execution and multi-input handling. Use `wait()` to retrieve results when ready, ideal for high-throughput or multi-threaded workflows.

run_async

The `run_async()` function submits a non-blocking inference request using either a single input tensor or a list of input tensors. It is suitable for single-sample asynchronous execution and returns a job ID used to retrieve the result via wait(`job_id`).

```
def run_async(
    input_data: Union[np.ndarray, List[np.ndarray]],
    user_arg: Any = None,
    output_buffer: Optional[Union[np.ndarray, List[np.ndarray]]] = None
) -> int
```

Input Type	Condition	Interpretation	Output Type	Constraints
`np.ndarray`	Any	Single Inference	`int` (jobId)	Batch not supported
`List[np.ndarray]`	len == input_count	Multi-input single	`int` (jobId)	Batch not supported
`List[np.ndarray]`	len == 1	Single-input single	`int` (jobId)	Batch not supported

run_async_multi_input

The `run_async_multi_input()` function performs asynchronous single-sample inference using a dictionary to map input tensor names to their corresponding NumPy arrays. This method is intended for multi-input models and returns a job ID to be used with wait(`job_id`).

```
def run_async_multi_input(
    input_tensors: Dict[str, np.ndarray],
    user_arg: Any = None,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    output_buffer: Optional[List[np.ndarray]] = None
) -> int
```

Input Type	Constraints	Output Type	Notes
`Dict[str, np.ndarray]`	For multi-input models only	`int` (jobId)	Single inference only

wait

The `wait()` function blocks the execution until the specified asynchronous inference job is complete and returns the corresponding inference results.

```
def wait(job_id: int) -> List[np.ndarray]
```

6.3 Input Format Parsing

This section explains how the SDK determines the appropriate inference mode based on the input data format. The engine automatically analyzes inputs such as np.ndarray, `List[np.ndarray]`, and nested lists to decide between single, batch, or auto-split inference.

6.3.1 Python Input Parsing Flow

```
def _analyze_input_format(input_data):
    # 1. Check for np.ndarray
    if isinstance(input_data, np.ndarray):
        if should_auto_split_input(input_data):
            return auto_split_single_inference()
        else:
            return single_inference()

    # 2. Check for List
    if isinstance(input_data, list):
        if isinstance(input_data[0], list):
            # List[List[np.ndarray]] - Explicit batch
            return explicit_batch_inference()
        else:
            # List[np.ndarray] - Requires complex analysis
            return analyze_list_ndarray(input_data)
```


© Copyright 2026 DEEPX All Rights Reserved.

6.3.2 `List[np.ndarray]` Parsing Rules

```
def analyze_list_ndarray(input_data):
    input_count = len(input_data)

    if is_multi_input_model():
        expected_count = get_input_tensor_count()

        if input_count == expected_count:
            return single_inference()
        elif input_count % expected_count == 0:
            batch_size = input_count // expected_count
            return batch_inference(batch_size)
        elif all(should_auto_split_input(arr) for arr in input_data):
            return auto_split_batch_inference()
        else:
            raise ValueError("Invalid input count")
    else:  # Single-input model
        if input_count == 1:
            return single_inference()
        else:
            return batch_inference(input_count)
```

6.4 Output Format Specification

This section describes the structure of inference outputs returned by the SDK, depending on the API (C++ or Python) and the inference mode (single, batch, or asynchronous). The output format varies to match the input pattern and execution method.

6.4.1 Single Inference Output

API	Output Format	Structure
C++ Run	`TensorPtrs`	`vector<shared_ptr<Tensor>>`
Python run	`List[np.ndarray]`	`[output1, output2, ...]`

6.4.2 Batch Inference Output

API	Output Format	Structure
C++ Run (batch)	`vector<TensorPtrs>`	`[sample1_outputs, sample2_outputs, ...]`

© Copyright 2026 DEEPX All Rights Reserved.

API	Output Format	Structure
Python run (batch)	`List[List[np.ndarray]]`	`[[s1_o1, s1_o2], [s2_o1, s2_o2], ...]`

6.4.3 Asynchronous Inference Output

API	Immediate Return	After `wait`
C++ RunAsync	`int` (jobId)	`TensorPtrs`
Python run_async	`int` (jobId)	`List[np.ndarray]`

6.5 Edge Cases and Special Handling

This section describes how the SDK handles non-standard input scenarios, such as auto-splitting, batch size calculation, invalid input conditions, and custom output buffers. These rules ensure robustness and flexibility across a wide range of inference use cases.

6.5.1 Auto-Split Detection Logic

C++

```cpp
bool shouldAutoSplitInput() const {
    return _isMultiInput && _inputTasks.size() == 1;
}
```

Python

```python
def _should_auto_split_input(input_data: np.ndarray) -> bool:
    if not self.is_multi_input_model():
        return False

    expected_total_size = self.get_input_size()
    actual_size = input_data.nbytes

    return actual_size == expected_total_size
```


© Copyright 2026 DEEPX All Rights Reserved.

6.5.2 Batch Size Determination

Condition	Batch Size Calculation
Single-input + List[np.ndarray]	`len(input_data)`
Multi-input + List[np.ndarray]	`len(input_data) // input_tensor_count`
List[List[np.ndarray]]	`len(input_data)`

6.5.3 Common Error Conditions

Condition	Error Type	Message
Multi-input + invalid size	`ValueError`	"Invalid input count for multi-input model"
Async + batch	`ValueError`	"Batch inference not supported in async"
Empty input	`ValueError`	"Input data cannot be empty"
Type mismatch	`TypeError`	"Expected np.ndarray or List[np.ndarray]"

6.5.4 Custom Output Buffer Handling (Python)

Input Format	Output Buffer Format	Handling
Single Inference	`None`	Auto-allocated
Single Inference	`List[np.ndarray]`	User-provided
Single Inference	`np.ndarray` (total_size)	Used after auto-split
Batch Inference	`List[List[np.ndarray]]`	Explicit batch buffer
Batch Inference	`List[np.ndarray]`	Flattened batch buffer

6.6 Performance Optimization Guidelines

This section describes key performance-related trade-offs when using the inference API, including memory allocation strategies and the impact of different inference methods on latency and throughput.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

6.6.1 Memory Allocation Strategy

Method	Pros	Cons
Auto-allocation (No Buffer)	Ease of use	Memory allocated on every call
User-provided (With Buffer)	Performance optimization	Complex memory management

6.6.2 Choosing Inference Methods (Sync vs Async vs Batch)

Method	Use Case	Characteristics
Synchronous	Simple processing	Sequential execution
Asynchronous	High throughput	Requires callback management
Batch	Bulk processing	Increased memory usage

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7. Multi-input Inference Guide

This section describes how to use **DX-RT**'s inference APIs with models that require multiple input tensors. It covers methods for executing multi-input inference in both synchronous and asynchronous modes, with or without user-managed output buffers. Support for batch processing formats is also explained to help optimize throughput and integration flexibility.

7.1 Identifying a Multi-Input Model

C++

```cpp
dxrt::InferenceEngine ie(modelPath);

// Check if the model is multi-input
bool isMultiInput = ie.IsMultiInputModel();

// Get the number of input tensors
int inputCount = ie.GetInputTensorCount();

// Get the input tensor names
std::vector<std::string> inputNames = ie.GetInputTensorNames();

// Get the input tensor to task mapping
std::map<std::string, std::string> mapping = ie.GetInputTensorToTaskMapping();
```

Python

```python
from dx_engine import InferenceEngine

ie = InferenceEngine(model_path)

# Check if the model is multi-input
is_multi_input = ie.is_multi_input_model()

# Get the number of input tensors
input_count = ie.get_input_tensor_count()

# Get the input tensor names
input_names = ie.get_input_tensor_names()

# Get the input tensor to task mapping
mapping = ie.get_input_tensor_to_task_mapping()
```


© Copyright 2026 DEEPX All Rights Reserved.

7.2 Multi-Input Inference

This section describes how to perform inference on multi-input models using either auto-allocated or user-provided output buffers for single-sample execution.

7.2.1 Without Output Buffers (Auto-Allocation)

In this method, the inference engine automatically allocates memory for outputs, allowing you to omit output buffer management. Supported input formats include dictionary-based, vector-based, and auto-split inputs.

Dictionary Format (Recommended)

This method involves providing input tensors mapped by their names. It is the most explicit and least error-prone method.

C++

```cpp
// Using Dictionary format (auto-allocation)
std::map<std::string, void*> inputTensors;
inputTensors["input1"] = input1_data;
inputTensors["input2"] = input2_data;

// Synchronous inference without output buffers (auto-allocation)
auto outputs = ie.RunMultiInput(inputTensors);
```

Python

```python
# Using Dictionary format (auto-allocation)
input_tensors = {
    "input1": input1_array,
    "input2": input2_array
}

# Synchronous inference without output buffers (auto-allocation)
outputs = ie.run_multi_input(input_tensors)
```

Vector Format

This method involves providing input tensors in a vector/list. The order **must** match the order returned by `GetInputTensorNames()`.

© Copyright 2026 DEEPX All Rights Reserved.

C++

```cpp
// Using Vector format (must match the order of GetInputTensorNames())
std::vector<void*> inputPtrs = {input1_data, input2_data};

// Synchronous inference without output buffers (auto-allocation)
auto outputs = ie.RunMultiInput(inputPtrs);
```

Python

```python
# Using Vector format (must match the order of get_input_tensor_names())
input_list = [input1_array, input2_array]

# Synchronous inference without output buffers (auto-allocation)
outputs = ie.run(input_list)
```

Auto-Split Format

This method automatically splits a single concatenated buffer into multiple inputs. It is applied automatically when the total size of the provided buffer matches the model's total input size.

C++

```cpp
// A single buffer with all inputs concatenated
std::vector<uint8_t> concatenatedInput(ie.GetInputSize());
// ... fill data ...

// Processed via auto-split (output buffers auto-allocated)
auto outputs = ie.Run(concatenatedInput.data());
```

Python

```python
# A single array with all inputs concatenated
concatenated_input = np.zeros(ie.get_input_size(), dtype=np.uint8)
# ... fill data ...

# Processed via auto-split (output buffers auto-allocated)
outputs = ie.run(concatenated_input)
```

7.2.2 With User-Provided Output Buffers

In this method, the user supplies pre-allocated output buffers, offering greater control over memory usage and improved performance. Supported input formats include dictionary, vector, and auto-split.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Dictionary Format

C++

```cpp
// Using Dictionary format
std::map<std::string, void*> inputTensors;
inputTensors["input1"] = input1_data;
inputTensors["input2"] = input2_data;

// Create output buffer
std::vector<uint8_t> outputBuffer(ie.GetOutputSize());

// Synchronous inference (with user-provided output buffer)
auto outputs = ie.RunMultiInput(inputTensors, userArg, outputBuffer.data());
```

Python

```python
# Using Dictionary format
input_tensors = {
    "input1": input1_array,
    "input2": input2_array
}

# Create output buffers
output_buffers = [np.zeros(size, dtype=np.uint8) for size in ie.get_output_tensor_siz
es()]

# Synchronous inference (with user-provided output buffers)
outputs = ie.run_multi_input(input_tensors, output_buffers=output_buffers)
```

Vector Format

C++

```cpp
// Using Vector format (must match the order of GetInputTensorNames())
std::vector<void*> inputPtrs = {input1_data, input2_data};

// Create output buffer
std::vector<uint8_t> outputBuffer(ie.GetOutputSize());

// Synchronous inference (with user-provided output buffer)
auto outputs = ie.RunMultiInput(inputPtrs, userArg, outputBuffer.data());
```

Python

```python
# Using Vector format (must match the order of get_input_tensor_names())
input_list = [input1_array, input2_array]
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
# Create output buffers
output_buffers = [np.zeros(size, dtype=np.uint8) for size in ie.get_output_tensor_siz
es()]

# Synchronous inference (with user-provided output buffers)
outputs = ie.run(input_list, output_buffers=output_buffers)
```

Auto-Split Format

C++

```
// A single buffer with all inputs concatenated
std::vector<uint8_t> concatenatedInput(ie.GetInputSize());
// ... fill data ...

// Create output buffer
std::vector<uint8_t> outputBuffer(ie.GetOutputSize());

// Processed via auto-split (with user-provided output buffer)
auto outputs = ie.Run(concatenatedInput.data(), userArg, outputBuffer.data());
```

Python

```
# A single array with all inputs concatenated
concatenated_input = np.zeros(ie.get_input_size(), dtype=np.uint8)
# ... fill data ...

# Create output buffers
output_buffers = [np.zeros(size, dtype=np.uint8) for size in ie.get_output_tensor_siz
es()]

# Processed via auto-split (with user-provided output buffers)
outputs = ie.run(concatenated_input, output_buffers=output_buffers)
```

7.3 Multi-Input Batch Inference

This section explains how to perform batch inference on multi-input models. The SDK supports two input formatting styles: explicit batch format, where each sample's inputs are grouped clearly, and flattened batch format, where all inputs are provided as a flat list.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.3.1 Explicit Batch Format

This method requires grouping input tensors per sample, allowing the inference engine to clearly distinguish each item in the batch.

C++

```cpp
// Batch input buffers (concatenated format)
std::vector<void*> batchInputs = {sample1_ptr, sample2_ptr, sample3_ptr};
std::vector<void*> batchOutputs = {output1_ptr, output2_ptr, output3_ptr};
std::vector<void*> userArgs = {userArg1, userArg2, userArg3};

// Batch inference
auto results = ie.Run(batchInputs, batchOutputs, userArgs);
```

Python

```python
# Format: List[List[np.ndarray]]
batch_inputs = [
    [sample1_input1, sample1_input2],  # First sample
    [sample2_input1, sample2_input2],  # Second sample
    [sample3_input1, sample3_input2]   # Third sample
]

batch_outputs = [
    [sample1_output1, sample1_output2],  # Output buffers for the first sample
    [sample2_output1, sample2_output2],  # Output buffers for the second sample
    [sample3_output1, sample3_output2]   # Output buffers for the third sample
]

# Batch inference
results = ie.run(batch_inputs, output_buffers=batch_outputs)
```

7.3.2 Flattened Batch Format

This method provides all input tensors as a single flattened list, with inputs from all batch samples arranged sequentially.

Python

```python
# Flattened format: [sample1_input1, sample1_input2, sample2_input1,
sample2_input2, ...]
flattened_inputs = [
    sample1_input1, sample1_input2,  # First sample
    sample2_input1, sample2_input2,  # Second sample
    sample3_input1, sample3_input2   # Third sample
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
]

# Automatically recognized as a batch (input count is a multiple of the model's input
count)
results = ie.run(flattened_inputs, output_buffers=batch_outputs)
```

7.4 Multi-Input Asynchronous Inference

This section describes how to perform asynchronous inference on multi-input models using the **DX-RT** API. Two methods are supported: callback-based execution for event-driven workflows and simplified execution using job IDs and polling via `wait()`.

7.4.1 Callback-Based Asynchronous Execution

This method registers a user-defined callback to handle inference results asynchronously, allowing the engine to automatically invoke the callback when each job completes.

C++

```
// Register callback function
ie.RegisterCallback([](dxrt::TensorPtrs& outputs, void* userArg) -> int {
    // Process outputs
    return 0;
});

// Dictionary format asynchronous inference
int jobId = ie.RunAsyncMultiInput(inputTensors, userArg);
// or
// Vector format asynchronous inference
int jobId = ie.RunAsyncMultiInput(inputPtrs, userArg);
```

Python

```
# Define callback function
def callback_handler(outputs, user_arg):
    # Process and validate outputs
    return 0

# Register callback
ie.register_callback(callback_handler)

# Dictionary format asynchronous inference
job_id = ie.run_async_multi_input(input_tensors, user_arg=user_arg)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# Vector format asynchronous inference
job_id = ie.run_async(input_list, user_arg=user_arg)
```

7.4.2 Simplified Asynchronous Execution

This method performs asynchronous inference using job IDs and retrieves results manually via `wait()`, providing a simple and non-callback-based alternative.

C++

```
// Single buffer asynchronous inference
int jobId = ie.RunAsync(inputPtr, userArg);

// Wait for the result
auto outputs = ie.Wait(jobId);
```

Python

```
# Single buffer asynchronous inference
job_id = ie.run_async(input_buffer, user_arg=user_arg)

# Wait for the result
outputs = ie.wait(job_id)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

8. Configuration and DeviceStatus Guide

This chapter introduces global utility classes provided by the **DX-RT** SDK for managing configuration settings and querying device status. These classes are implemented as singletons to ensure consistent state across C++ and Python, enabling centralized control over runtime behavior and hardware monitoring.

8.1 Configuration Management

The `Configuration` class serves as the centralized interface for managing global runtime settings in the **DX-RT** library. Designed as a thread-safe singleton, it ensures consistent configuration across both C++ and Python environments. In Python, this class wraps the underlying C++ singleton, maintaining a shared state between the two languages.

Key Features:

- **Singleton Design**: Guarantees a single, globally accessible configuration instance.
- **Runtime Configurability**: Supports dynamic enabling/disabling of features and real-time attribute updates.
- **Version Access**: Provides functions to retrieve library, driver, and device version information.
- **Cross-Language Support**: Fully accessible from both C++ and Python with identical behavior.

8.1.1 Obtaining the Configuration Instance

The method of accessing the global Configuration object differs slightly between C++ and Python, but both ensure interaction with the same underlying singleton.

C++ In C++, the global configuration instance is accessed via the static method `GetInstance()`. Always use `GetInstance()` to obtain the shared singleton — constructing a local `Configuration` object would create a separate instance that does not reflect the global runtime state.

```
#include "dxrt/common.h"

// Recommended: Get the shared, global instance
dxrt::Configuration& config = dxrt::Configuration::GetInstance();
```


© Copyright 2026 DEEPX All Rights Reserved.

Python In Python, the Configuration class can be instantiated directly. Internally, this constructor accesses the shared C++ singleton, ensuring all instances reflect the same state.

```python
from dx_engine.configuration import Configuration

# Create a Configuration object.
# This holds a reference to the global settings instance.
config = Configuration()
```

Regardless of language, all operations performed on the Configuration instance affect the global runtime state.

8.1.2 Configuration Scopes: `ITEM` and `ATTRIBUTE`

The Configuration interface organizes runtime settings using two scoped enumerations: `ITEM` and `ATTRIBUTE`. These are supported consistently in both C++ and Python.

ITEM

An `ITEM` represents a high-level feature or module within the **DX-RT** that can be enabled or disabled. Common examples include runtime profiling, logging, or device tracing.

Item	Description
`DEBUG`	Enables general debug mode.
`PROFILER`	Enables profiler functionality.
`SERVICE`	Configures service-related operations.
`DYNAMIC_CPU_THREAD`	Manages dynamic CPU thread settings.
`TASK_FLOW`	Controls task flow management features.
`SHOW_THROTTLING`	Enables the display of throttling information.
`SHOW_PROFILE`	Enables the display of profile results.
`SHOW_MODEL_INFO`	Enables the display of detailed model information.
`CUSTOM_INTRA_OP_THREADS`	Enables custom ONNX Runtime intra-operator thread count configuration.
`CUSTOM_INTER_OP_THREADS`	Enables custom ONNX Runtime inter-operator thread count configuration.

© Copyright 2026 DEEPX All Rights Reserved.

ATTRIBUTE

An `ATTRIBUTE` defines a property associated with a specific ITEM. It is typically used to set or retrieve string-based values such as file paths, flags, or operational modes.

Attribute	Associated `ITEM`	Description
`PROFILER_SHOW_DATA`	`PROFILER`	Attribute for showing profiler data.
`PROFILER_SAVE_DATA`	`PROFILER`	Attribute for saving profiler data to a file.
`CUSTOM_INTRA_OP_THREADS_NUM`	`CUSTOM_INTRA_OP_THREADS`	Number of threads for ONNX Runtime intra-operator parallelism (integer string, 1-hardware_concurrency).
`CUSTOM_INTER_OP_THREADS_NUM`	`CUSTOM_INTER_OP_THREADS`	Number of threads for ONNX Runtime inter-operator parallelism (integer string, 1-hardware_concurrency).

8.1.3 Core Operations and Examples

This section outlines the primary operations supported by the Configuration class, with usage examples for both C++ and Python.

Enabling and Disabling Features

Enable or disable specific runtime modules using the `ITEM` enumeration. This allows dynamic control over major DXRT features at runtime.

C++

```cpp
// Enable the profiler
config.SetEnable(dxrt::Configuration::ITEM::PROFILER, true);

// Check if the profiler is enabled
if (config.GetEnable(dxrt::Configuration::ITEM::PROFILER)) {
    std::cout << "Profiler is enabled." << std::endl;
}
```


© Copyright 2026 DEEPX All Rights Reserved.

Python

```python
# Enable showing model information
config.set_enable(Configuration.ITEM.SHOW_MODEL_INFO, True)

# Check if showing model info is enabled
is_enabled = config.get_enable(Configuration.ITEM.SHOW_MODEL_INFO)
print(f"SHOW_MODEL_INFO is enabled: {is_enabled}")
```

Working with Attributes

Configure detailed runtime behavior by setting or retrieving string-based values using the ATTRIBUTE enumeration. `Attributes` are typically tied to a specific `ITEM`.

C++

```cpp
// First, ensure the parent item is enabled
config.SetEnable(dxrt::Configuration::ITEM::PROFILER, true);

// Set the path where profiler data should be saved
std::string profile_path = "/var/log/my_app_profile.json";
config.SetAttribute(dxrt::Configuration::ITEM::PROFILER,
                    dxrt::Configuration::ATTRIBUTE::PROFILER_SAVE_DATA,
                    profile_path);

// Retrieve the attribute value later
std::string saved_path = config.GetAttribute(dxrt::Configuration::ITEM::PROFILER,
                                             dxrt::Configuration::ATTRIBUTE::PROFILE
R_SAVE_DATA);
```

Python

```python
# First, ensure the parent item is enabled
config.set_enable(Configuration.ITEM.PROFILER, True)

# Set the path for saving profiler data
profile_log_path = "/var/log/dx_profile.json"
config.set_attribute(Configuration.ITEM.PROFILER,
                     Configuration.ATTRIBUTE.PROFILER_SAVE_DATA,
                     profile_log_path)

# Retrieve the path later
saved_path = config.get_attribute(Configuration.ITEM.PROFILER,
                                  Configuration.ATTRIBUTE.PROFILER_SAVE_DATA)
print(f"Profiler data will be saved to: {saved_path}")
```


© Copyright 2026 DEEPX All Rights Reserved.

Retrieving Version Information

Query the current DXRT library and driver versions. These functions are essential for debugging, compatibility checks, and system diagnostics.

C++

```
#include <vector>
#include <utility>
#include <string>

try {
    std::cout << "DXRT Library Version: " << config.GetVersion() << std::endl;
    std::cout << "Driver Version: " << config.GetDriverVersion() << std::endl;

    // Get firmware versions for all detected devices
    std::vector<std::pair<int, std::string>> fw_versions =
config.GetFirmwareVersions();
    for (const auto& fw : fw_versions) {
        std::cout << "Device " << fw.first << " Firmware Version: " << fw.second << s
td::endl;
    }
} catch (const std::runtime_error& e) {
    std::cerr << "Error retrieving version information: " << e.what() << std::endl;
}
```

Python

```
print(f"Library Version: {config.get_version()}")
print(f"Driver Version: {config.get_driver_version()}")
print(f"PCIe Driver Version: {config.get_pcie_driver_version()}")
```

8.1.4 Loading Configuration from File

The Configuration class supports loading settings from external configuration files using the `LoadConfigFile()` method. This allows you to manage runtime settings through configuration files rather than hardcoding them in your application.

© Copyright 2026 DEEPX All Rights Reserved.

Configuration File Format

Configuration files use a simple key-value format with `KEY=VALUE` pairs. Here's an example configuration file (`common.cfg`):

```
# General debug and profiling settings
ENABLE_DEBUG=0
USE_PROFILER=1
ENABLE_SHOW_PROFILER_DATA=1
ENABLE_SAVE_PROFILER_DATA=1

# ONNX Runtime thread settings (Opt-in Example)
# Note: Default common.cfg sets these to 0 (disabled)
USE_CUSTOM_INTRA_OP_THREADS=1
USE_CUSTOM_INTER_OP_THREADS=1
CUSTOM_INTRA_OP_THREADS_COUNT=4
CUSTOM_INTER_OP_THREADS_COUNT=2
```

> **IMPORTANT**
>
> The ONNX Runtime thread settings shown above are an **opt-in example**. The actual default `common.cfg` file sets `USE_CUSTOM_INTRA_OP_THREADS=0` and `USE_CUSTOM_INTER_OP_THREADS=0` , which means ONNX Runtime will use its automatic thread management by default.

Configuration Parameters

General Settings:

- **`ENABLE_DEBUG`** : Enable/disable debug mode (0=disabled, 1=enabled)
- **`USE_PROFILER`** : Enable/disable profiler functionality (0=disabled, 1=enabled)
- **`ENABLE_SHOW_PROFILER_DATA`** : Enable/disable showing profiler data in console (0=disabled, 1=enabled)
- **`ENABLE_SAVE_PROFILER_DATA`** : Enable/disable saving profiler data to file (0=disabled, 1=enabled)

Thread Configuration Parameters:

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The following parameters control ONNX Runtime thread behavior:

- **`USE_CUSTOM_INTRA_OP_THREADS`**: Enable/disable custom intra-operator thread count (0=disabled, 1=enabled)
- **`USE_CUSTOM_INTER_OP_THREADS`**: Enable/disable custom inter-operator thread count (0=disabled, 1=enabled)
- **`CUSTOM_INTRA_OP_THREADS_COUNT`**: Number of threads for intra-operator parallelism (integer string, range: 1 to `hardware_concurrency()`)
- **`CUSTOM_INTER_OP_THREADS_COUNT`**: Number of threads for inter-operator parallelism (integer string, range: 1 to `hardware_concurrency()`)

Thread Count Validation:

- Values are automatically clamped to the range [1, `std::thread::hardware_concurrency()`]
- Invalid values (non-numeric strings) default to 1
- Empty values default to 1
- Out-of-range values are clamped with debug logging

ONNX Runtime Behavior:

- **Default behavior** (when `USE_CUSTOM_INTRA_OP_THREADS=0`): ONNX Runtime uses automatic thread count (typically equals hardware concurrency)
- **Default behavior** (when `USE_CUSTOM_INTER_OP_THREADS=0`): Uses sequential execution mode with 1 thread
- **Custom behavior** (when enabled=1): Uses the specified `CUSTOM_*_THREADS_COUNT` values with validation and clamping

Loading Configuration Files

C++

```cpp
#include "dxrt/common.h"

// Load configuration from file
dxrt::Configuration& config = dxrt::Configuration::GetInstance();
config.LoadConfigFile("path/to/common.cfg");

// Configuration settings are now applied globally
```


© Copyright 2026 DEEPX All Rights Reserved.

Python

```python
from dx_engine.configuration import Configuration

# Load configuration from file
config = Configuration()
config.load_config_file("path/to/common.cfg")

# Configuration settings are now applied globally
```

8.2 Device Status Monitoring

The `DeviceStatus` class provides a real-time snapshot of the NPU device's state, including static properties (e.g., memory) and dynamic metrics (e.g., temperature, clock speed). Each instance represents the status of a specific device at the time it was queried.

Workflow Overview:

- Retrieve the total number of available devices.
- Access a specific device's status using its device ID.
- Query hardware information and real-time metrics via instance methods.

8.2.1 Getting Started: Accessing Devices

The first step is to find out how many devices are available, then create a status object for the one you want to inspect.

Step 1: Get the Device Count

Use the static method to determine how many devices are currently recognized by the system.

C++

```cpp
#include <dxrt/dxrt_cxx_api.h>

int deviceCount = dxrt::DeviceStatus::GetDeviceCount();
std::cout << "Found " << deviceCount << " devices." << std::endl;
```

Python

```python
from dx_engine.device_status import DeviceStatus # Main Python class
```


© Copyright 2026 DEEPX All Rights Reserved.

```
device_count = DeviceStatus.get_device_count()
print(f"Found {device_count} devices.")
```

Step 2: Retrieve the Device Status

Once the count is known, access the status object using a valid device ID (`0` to `device_count - 1`).

C++

Use `IsValid()` to verify that the device data was successfully retrieved:

```
// Get a status snapshot for device with ID 0
dxrt::DeviceStatus status = dxrt::DeviceStatus::GetCurrentStatus(0);
if (status.IsValid())
{
    std::cout << "Successfully created status object for device " << status.GetId() <
< std::endl;
}
else
{
    std::cerr << "Warning: Device 0 data is not available." << std::endl;
}
```

Python

Use the factory method `get_current_status()` to get a `DeviceStatus` object:

```
# Get the status object for the first device (ID 0)
status_obj = DeviceStatus.get_current_status(0)
if status_obj.is_valid():
    print(f"Successfully created status object for device ID: {status_obj.get_id()}")
else:
    print("Warning: Device 0 data is not available.")
```

8.2.2 Querying Device Properties and Metrics

Once you obtain a `DeviceStatus` object, you can retrieve both static hardware properties and real-time operational metrics of the NPU device.

Accessing Specific Attributes (C++ and Python)

For programmatic use, both C++ and Python interfaces offer methods to retrieve specific values from the status object:

© Copyright 2026 DEEPX All Rights Reserved.

Hardware Status (C++ and Python)

Metric	C++ Method	Python Method	Return Value
Device ID	`GetId()`	`get_id()`	`int`
Temperature	`GetTemperature(ch)`	`get_temperature(ch)`	`int` (Celsius)
NPU Voltage	`GetNpuVoltage(ch)`	`get_npu_voltage(ch)`	`uint32_t` / `int` (mV)
NPU Clock	`GetNpuClock(ch)`	`get_npu_clock(ch)`	`uint32_t` / `int` (MHz)
Core Utilization	`GetCoreUtilization(coreId)`	`get_core_utilization(core_id)`	`double` / `float` (0.0~100.0 %)
Memory Used	`GetMemoryUsed()`	`get_memory_used()`	`uint64_t` / `int` (bytes)
Memory Free	`GetMemoryFree()`	`get_memory_free()`	`uint64_t` / `int` (bytes)
Data Validity	`IsValid()`	`is_valid()`	`bool`

NOTE

The monitoring thread updates shared memory approximately every 1 second. Polling `GetCurrentStatus()` more frequently than this interval will return the same data.

Formatted Summary Strings (C++ Only)

Method	Description	Return Value
`GetInfoString()`	Returns static hardware info (model, memory, board, firmware)	`std::string`
`GetStatusString()`	Returns dynamic real-time status (voltage, clock, temp, DVFS state)	`std::string`

For a complete list of C++ methods (memory info, board type, device variants, etc.), see the C++ API Reference.

© Copyright 2026 DEEPX All Rights Reserved.

8.2.3 Complete Usage Example

This section demonstrates how to iterate through all available NPU devices and retrieve their full status information — including hardware status, core utilization, and memory usage — using both C++ and Python. These examples are based on the `device_monitoring` sample provided with the SDK.

For method details, see the C++ API Reference and Python API Reference.

C++

```cpp
#include <iostream>
#include <dxrt/dxrt_cxx_api.h>

void printDeviceInfo(const dxrt::DeviceStatus &ds)
{
    std::cout << "=== Device Info (id=" << ds.GetId() << ") ===" << std::endl;
    std::cout << ds.GetInfoString() << std::endl;
}

void printDeviceDynamicStatus(const dxrt::DeviceStatus &ds)
{
    std::cout << "[Device " << ds.GetId() << "]"
              << " temp=["
              << ds.GetTemperature(0) << ", "
              << ds.GetTemperature(1) << ", "
              << ds.GetTemperature(2) << "]C"
              << ", voltage=["
              << ds.GetNpuVoltage(0) << ", "
              << ds.GetNpuVoltage(1) << ", "
              << ds.GetNpuVoltage(2) << "]mV"
              << ", clock=["
              << ds.GetNpuClock(0) << ", "
              << ds.GetNpuClock(1) << ", "
              << ds.GetNpuClock(2) << "]MHz"
              << " | util=["
              << ds.GetCoreUtilization(0) << "%, "
              << ds.GetCoreUtilization(1) << "%, "
              << ds.GetCoreUtilization(2) << "%]"
              << ", mem_used=" << ds.GetMemoryUsed()
              << ", mem_free=" << ds.GetMemoryFree()
              << std::endl;
}

int main()
{
    int device_count = dxrt::DeviceStatus::GetDeviceCount();
    if (device_count == 0)
    {
        std::cout << "No DEEPX devices found." << std::endl;
        return 1;
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    }

    std::cout << "Querying status for " << device_count << " device(s)...\n" <<
std::endl;

    for (int id = 0; id < device_count; id++)
    {
        auto ds = dxrt::DeviceStatus::GetCurrentStatus(id);
        if (!ds.IsValid())
        {
            std::cerr << "Warning: Device " << id << " data is not available." <<
std::endl;
            continue;
        }
        printDeviceInfo(ds);
        printDeviceDynamicStatus(ds);
        std::cout << std::endl;
    }

    return 0;
}
```

Python

```
from dx_engine.device_status import DeviceStatus

def print_device_status(ds):
    """Print all fields from a DeviceStatus object."""
    print(f"=== Device Status (id={ds.get_id()}) ===")

    print(f"[STATUS] temp=["
          f"{ds.get_temperature(0)}, "
          f"{ds.get_temperature(1)}, "
          f"{ds.get_temperature(2)}]C, "
          f"voltage=["
          f"{ds.get_npu_voltage(0)}, "
          f"{ds.get_npu_voltage(1)}, "
          f"{ds.get_npu_voltage(2)}]mV, "
          f"clock=["
          f"{ds.get_npu_clock(0)}, "
          f"{ds.get_npu_clock(1)}, "
          f"{ds.get_npu_clock(2)}]MHz")

    print(f"[USAGE] util=["
          f"{ds.get_core_utilization(0):.1f}%, "
          f"{ds.get_core_utilization(1):.1f}%, "
          f"{ds.get_core_utilization(2):.1f}%], "
          f"mem_used={ds.get_memory_used()}, "
          f"mem_free={ds.get_memory_free()}")

def main():
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    device_count = DeviceStatus.get_device_count()
    if device_count == 0:
        print("No devices found.")
        return

    print(f"Querying status for {device_count} device(s)...\n")

    for device_id in range(device_count):
        ds = DeviceStatus.get_current_status(device_id)
        if not ds.is_valid():
            print(f"Warning: Device {device_id} data is not available.")
            continue
        print_device_status(ds)
        print()

if __name__ == "__main__":
    main()
```

8.2.4 Periodic Monitoring Example

The following examples demonstrate how to continuously monitor a device in a background thread with a configurable refresh interval. This is useful for frontend or business applications that need periodic device status updates while performing other tasks.

> **NOTE**
>
> The runtime's monitoring thread updates shared memory approximately every **1 second**. Setting the refresh interval below 1 second will not yield more frequent data updates. A minimum interval of **1 second** is recommended.

C++

```
#include <iostream>
#include <thread>
#include <atomic>
#include <chrono>
#include <dxrt/dxrt_cxx_api.h>

std::atomic<bool> g_running{true};

void monitorDevice(int deviceId, int intervalSec)
{
    while (g_running.load())
    {
        auto ds = dxrt::DeviceStatus::GetCurrentStatus(deviceId);
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        if (ds.IsValid())
        {
            std::cout << "[Device " << ds.GetId() << "]"
                      << " temp=" << ds.GetTemperature(0) << "C"
                      << ", util=" << ds.GetCoreUtilization(0) << "%"
                      << ", mem_used=" << ds.GetMemoryUsed()
                      << std::endl;
        }
        std::this_thread::sleep_for(std::chrono::seconds(intervalSec));
    }
}

int main()
{
    int device_id = 0;
    int interval_sec = 2;  // Refresh interval (minimum: 1 second)

    std::thread monitor_thread(monitorDevice, device_id, interval_sec);

    // Let the monitoring run for 30 seconds
    std::this_thread::sleep_for(std::chrono::seconds(30));

    g_running.store(false);
    monitor_thread.join();

    return 0;
}
```

Python

```
import threading
import time
from dx_engine.device_status import DeviceStatus

def monitor_device(device_id: int, interval_sec: int, stop_event: threading.Event):
    while not stop_event.is_set():
        try:
            ds = DeviceStatus.get_current_status(device_id)
            if ds.is_valid():
                print(f"[Device {ds.get_id()}]"
                      f" temp={ds.get_temperature(0)}C"
                      f", util={ds.get_core_utilization(0):.1f}%"
                      f", mem_used={ds.get_memory_used()}")
        except Exception as e:
            print(f"Warning: {e}")
        stop_event.wait(interval_sec)

if __name__ == "__main__":
    device_id = 0
    interval_sec = 2  # Refresh interval (minimum: 1 second)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    stop_event = threading.Event()
    monitor_thread = threading.Thread(
        target=monitor_device,
        args=(device_id, interval_sec, stop_event)
    )
    monitor_thread.start()

    # Let the monitoring run for 30 seconds
    time.sleep(30)

    stop_event.set()
    monitor_thread.join()
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

9. Tutorials

9.1 C++ Tutorials

This section provides C++ tutorials covering core DXRT SDK features, including synchronous/ asynchronous inference, batch processing, and runtime configuration. The examples help developers quickly integrate DXRT into real-time and embedded applications.

Run (Synchronous)

The synchronous Run method uses a single NPU core to perform inference in a blocking manner. It can be configured to utilize multiple NPU cores simultaneously by employing threads to run each core independently.

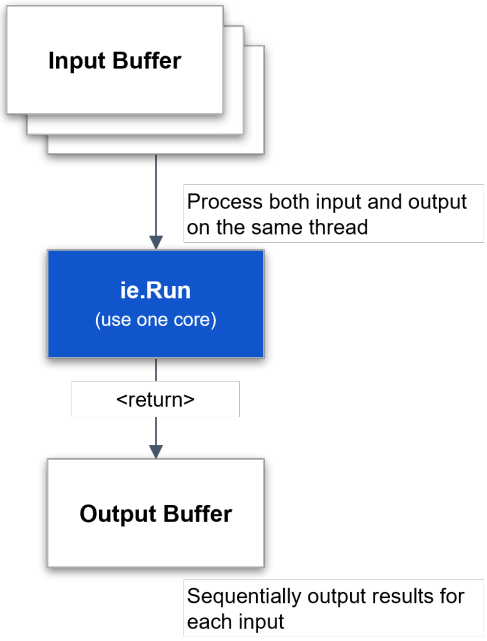

Figure. Synchronous Inference Operation

Inference Engine Run synchronous

- Inference synchronously
- Use **only** one npu core

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The following is the simplest example of synchronous inference.

run_sync_model.cpp

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

int main()
{
    std::string modelPath = "model-path";

    try
    {
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        // inference loop
        for(int i = 0; i < 100; ++i)
        {
            // inference synchronously
            // use only one npu core
            auto outputs = ie.Run(inputPtr.data());

            // post processing
            postProcessing(outputs);

        } // for i
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

RunAsync (Asynchronous)

The asynchronous Run mode is a method that performs inference asynchronously while utilizing multiple NPU cores simultaneously. It can be implemented to maximize NPU resources through a callback function or a thread wait mechanism.

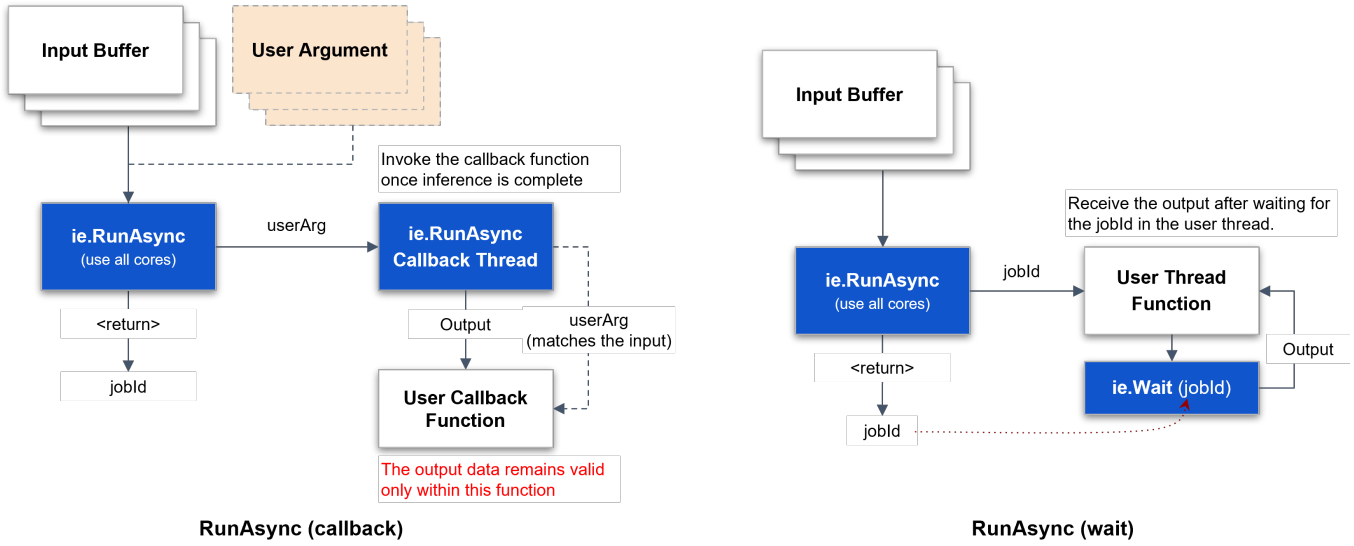

Figure. Asynchronous Inference Operation

Inference Engine RunAsync, Callback, User Argument

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

The following is an example of asynchronous inference using a callback function. A user argument can be used to synchronize the input with the output of the callback.

run_async_model.cpp

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

int main(int argc, char* argv[])
{
    ...

    int callback_count = 0;
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    try
    {

        std::mutex cv_mutex;
        std::condition_variable cv;

        // create inference engine instance with model
        dxrt::InferenceEngine ie(model_path);

        // register call back function
        ie.RegisterCallback([&callback_count, &loop_count, &cv_mutex, &cv]
            (dxrt::TensorPtrs &outputs, void *userArg) {

            std::ignore = outputs;
            std::ignore = userArg;

            std::unique_lock<std::mutex> lock(cv_mutex);
            callback_count++;
            if ( callback_count == loop_count ) cv.notify_one();

            return 0;
        });

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        auto start = std::chrono::high_resolution_clock::now();

        // inference loop
        for(int i = 0; i < loop_count; ++i)
        {
            // user argument
            std::pair<int, int> *userData = new std::pair<int, int>(i, loop_count);

            // inference asynchronously, use all npu cores
            ie.RunAsync(inputPtr.data(), userData);

            log.Debug("Inference request submitted with user_arg(" +
std::to_string(i) + ")");
        }

        // wait until all callbacks have been processed
        std::unique_lock<std::mutex> lock(cv_mutex);
        cv.wait(lock, [&callback_count, &loop_count] {
            return callback_count == loop_count;
        });

        ...

    }
    catch (const dxrt::Exception& e)
    {
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        ...
        return -1;
    }
    catch (const std::exception& e)
    {
        ...
        return -1;
    }
    catch(...)
    {
        ...
        return -1;
    }

    return (callback_count == loop_count ? 0 : -1);
}
```

The following is an example where multiple threads start input and inference, and a single callback processes the output.

Inference Engine RunAsync, Callback, User Argument, Thread

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

run_async_model_thread.cpp

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

static const int THREAD_COUNT = 3;
static std::atomic<int> gResultCount = {0};
static std::atomic<int> gTotalCount = {0};
static ConcurrentQueue<int> gResultQueue(1);
static std::mutex gCBMutex;

static int inferenceThreadFunc(dxrt::InferenceEngine& ie, std::vector<uint8_t>&
inputPtr, int threadIndex, int loopCount)
{

    // inference loop
    for(int i = 0; i < loopCount; ++i)
    {
        // user argument
        UserData *userData = new UserData();
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        // thread index
        userData->setThreadIndex(threadIndex);

        // total loop count
        userData->setLoopCount(loopCount);

        // loop index
        userData->setLoopIndex(i);

        try
        {
            // inference asynchronously, use all npu cores
            // if device-load >= max-load-value, this function will block
            ie.RunAsync(inputPtr.data(), userData);
        }
        catch(const dxrt::Exception& e)
        {
            std::cerr << e.what() << " error-code=" << e.code() << std::endl;
            std::exit(-1);
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << std::endl;
            std::exit(-1);
        }

    } // for i

    return 0;

}

// invoke this function asynchronously after the inference is completed
static int onInferenceCallbackFunc(dxrt::TensorPtrs &outputs, void *userArg)
{

    // the outputs are guaranteed to be valid only within this callback function
    // processing this callback functions as quickly as possible is beneficial
    // for improving inference performance

    // user data type casting
    UserData *user_data = reinterpret_cast<UserData*>(userArg);

    // thread index
    int thread_index = user_data->getThreadIndex();

    // loop index
    int loop_index = user_data->getLoopIndex();

    // post processing
    // transfer outputs to the target thread by thread_index
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    // postProcessing(outputs, thread_index);
    (void)outputs;


    // result count
    {
        // Mutex locks should be properly adjusted
        // to ensure that callback functions are thread-safe.
        std::lock_guard<std::mutex> lock(gCBMutex);

        gResultCount++;
        if ( gResultCount.load() == gTotalCount.load() ) gResultQueue.push(0);
    }

    // delete argument object
    delete user_data;

    return 0;
}


int main(int argc, char* argv[])
{
    ...

    bool result = false;

    try
    {
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // register call back function
        ie.RegisterCallback(onInferenceCallbackFunc);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        gTotalCount.store(loop_count * THREAD_COUNT);

        // thread vector
        std::vector<std::thread> thread_array;

        for(int i = 0; i < THREAD_COUNT; ++i)
        {
            // create thread
            thread_array.push_back(std::thread(inferenceThreadFunc, std::ref(ie),
std::ref(inputPtr), i, loop_count));
        }

        for(auto &t : thread_array)
        {
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
            t.join();
        } // for t


        // wait until all callbacks have been processed
        gResultQueue.pop();

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return result ? 0 : -1;
}
```

The following is an example of performing asynchronous inference by creating an inference wait thread. The main thread starts input and inference, and the inference wait thread retrieves the output data corresponding to the input.

Inference Engine RunAsync, Wait

- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

run_async_model_wait.cpp

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

// concurrent queue is a thread-safe queue data structure
// designed to be used in a multi-threaded environment
static ConcurrentQueue<int> gJobIdQueue;

// user thread to wait for the completion of inference
static int inferenceThreadFunc(dxrt::InferenceEngine& ie, int loopCount)
{
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    int count = 0;

    while(...)
    {
        // pop item from queue
        int jobId = gJobIdQueue.pop();

        try
        {
            // waiting for the inference to complete by jobId
            auto outputs = ie.Wait(jobId);

            // post processing
            postProcessing(outputs);

        }
        catch(const dxrt::Exception& e)  // exception for inference engine
        {
            std::cerr << e.what() << " error-code=" << e.code() << std::endl;
            std::exit(-1);
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << std::endl;
            std::exit(-1);
        }

        // something to do

        count++;
        if ( count >= loopCount ) break;

    } // while

    return 0;
}

int main()
{
    const int LOOP_COUNT = 100;
    std::string modelPath = "model-path";

    try
    {
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // do not register call back function
        // inferenceEngine.RegisterCallback(onInferenceCallbackFunc);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        // create thread
        auto t1 = std::thread(inferenceThreadFunc, std::ref(ie), LOOP_COUNT);

        // inference loop
        for(int i = 0; i < LOOP_COUNT; ++i)
        {

            // no need user argument
            // UserData *userData = getUserDataInstanceFromDataPool();

            // inference asynchronously, use all npu cores
            // if device-load >= max-load-value, this function will block
            auto jobId = ie.RunAsync(inputPtr.data());

            // push jobId in global queue variable
            gJobIdQueue.push(jobId);

        } // for i

        t1.join();
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch(std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```

Run (Batch)

The following is an example of batch inference with multiple inputs and multiple outputs.

run_batch_model.cpp

```
int main(int argc, char* argv[])
{
    ...

    try
    {
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // create temporary input buffer for example
        std::vector<uint8_t> inputBuffer(ie.GetInputSize(), 0);

        // input buffer vector
        std::vector<void*> inputBuffers;
        for(int i = 0; i < batch_count; ++i)
        {
            // assigns the same buffer pointer in this example
            inputBuffers.emplace_back(inputBuffer.data());
        }

        // output buffer vector
        std::vector<void*> output_buffers(batch_count, 0);

        // create user output buffers
        for(auto& ptr : output_buffers)
        {
            ptr = new uint8_t[ie.GetOutputSize()];
        } // for i

        // batch inference loop
        for(int i = 0; i < loop_count; ++i)
        {
            // inference asynchronously, use all npu core
            auto outputPtrs = ie.Run(inputBuffers, output_buffers);

            // postProcessing(outputs);
            (void)outputPtrs;
        }

        // Deallocated the user's output buffers
        for(auto& ptr : output_buffers)
        {
            delete[] static_cast<uint8_t*>(ptr);
        } // for i

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return 0;
}
```

Run & RunAsync

The method for converting a synchronous inference approach using one NPU core into an asynchronous inference approach using multiple NPU cores is as follows. It requires the use of callbacks or threads, as well as the implementation of multiple input buffers to support concurrent operations effectively.

Converting Run(Sync) to RunAsync

- Shift from Single NPU Core to Multiple Cores
  : Modify the existing Run(Sync) structure, which utilizes a single NPU core, to RunAsync structure capable of leveraging multiple NPU cores simultaneously.
- Create Multiple Input/Output Buffers
  : Implement multiple input/output buffers to prevent overwriting. Ensure an appropriate number of buffers are created to support concurrent operations effectively.
- Introduce Multi-Buffer Concept
  : To handle simultaneous inference processes, integrate a multi-buffer mechanism. This is essential for managing concurrent inputs and outputs without data conflicts.
- Asynchronous Inference with Threads or Callbacks
  : Adjust the code to ensure that inference inputs and outputs operate asynchronously using threads or callbacks for efficient processing.
- Thread-Safe Data Exchange
  : For data exchange between threads or callbacks, use a thread-safe queue or structured data mechanisms to avoid race conditions and ensure integrity.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

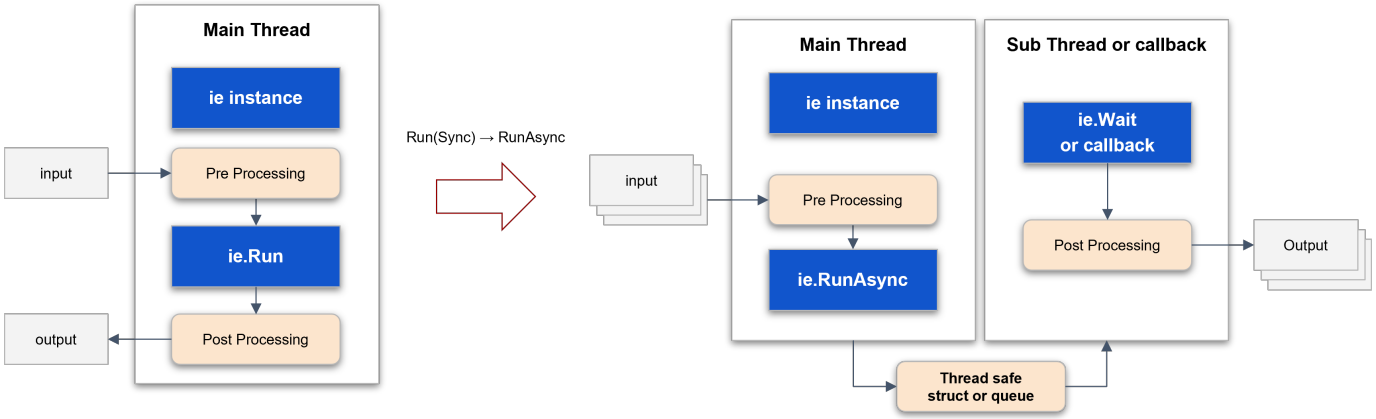

Figure. Converting Run(Sync) to RunAsync

Inference Option

The following inference options allow you to specify an NPU core for performing inference.

Inference Engine Run, Inference Option

- Select devices
  : default devices is `{}`
  : Choose devices to utilize

- Select bound option per device
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_ALL`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_0`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_1`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_2`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_01`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_12`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_02`

- Use onnx runtime library (`ORT`)
  : `useORT` on or off

`run_sync_model_bound.cpp`

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

int main()
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
{
    std::string modelPath = "model-path";

    try
    {

        // select bound option NPU_0 to NPU_2 per device
        dxrt::InferenceOption op;

        // first device only, default null
        op.devices.push_back(0); // use device 0
        op.devices.push_back(3); // use device 3

        // use BOUND_OPTION::NPU_0 only
        op.boundOption = dxrt::InferenceOption::BOUND_OPTION::NPU_0;

        // use ORT
        op.useORT = false;

        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath, op);

        // inference loop
        for(int i = 0; i < 100; ++i)
        {
            // input
            uint8_t* inputPtr = readInputData();

            // inference synchronously with boundOption
            // use only one npu core
            // ownership of the outputs is transferred to the user
            auto outputs = ie.Run(inputPtr);

            // post processing
            postProcessing(outputs);

        } // for i
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

> **WARNING**
>
> If multiple dxrt::InferenceEngine instances are initialized with specific NPU bindings (occupying NPUs 0, 1, and 2), any subsequent attempt to instantiate a new engine using BOUND_OPTION::NPU_ALL will enter a blocking state. The new instance will remain suspended until all required NPU resources become available. This synchronization behavior is consistent across both multi-threaded and multi-process environments.

Configuration and Device Status

This guide explains how to use the `Configuration` class to set up the inference engine and the `DeviceStatus` class to monitor hardware status in C++.

ENGINE CONFIGURATION

The `Configuration` class, implemented as a Singleton, allows you to set global parameters for the inference engine before it runs.

```
// Get the singleton instance and set engine parameters
dxrt::Configuration::GetInstance().SetEnable(dxrt::Configuration::ITEM::SHOW_MODEL_IN
FO, true);
dxrt::Configuration::GetInstance().SetEnable(dxrt::Configuration::ITEM::SHOW_PROFILE,
true);
```

- **`Configuration::GetInstance()`**: Accesses the single, global instance of the configuration manager.
- **`.SetEnable(...)`**: Enables engine features. Here, it's configured to print detailed model information and performance profiling data when the `InferenceEngine` is initialized.

QUERYING DEVICE STATUS

The `DeviceStatus` class provides real-time device information including hardware spec, voltage, clock, temperature, NPU core utilization, and DRAM memory usage.

```
// Get the number of available devices
auto device_count = dxrt::DeviceStatus::GetDeviceCount();

// Loop through each device
for(int i = 0; i < device_count; ++i)
{
    // Get a status object for the current device
    auto device_status = dxrt::DeviceStatus::GetCurrentStatus(i);

    // Query and print specific metrics
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    log.Info("Device: " + std::to_string(device_status.GetId()));
    log.Info("   Temperature: " + std::to_string(device_status.GetTemperature(0)));
    log.Info("   Voltage: " + std::to_string(device_status.GetNpuVoltage(0)));
    log.Info("   Clock: " + std::to_string(device_status.GetNpuClock(0)));
    log.Info("   Core Utilization: " + std::to_string(device_status.GetCoreUtilizatio
n(0)) + "%");
    log.Info("   Memory Used: " + std::to_string(device_status.GetMemoryUsed()) + "
bytes");
    log.Info("   Memory Free: " + std::to_string(device_status.GetMemoryFree()) + "
bytes");
}
```

- **`DeviceStatus::GetDeviceCount()`** : A static method that returns the number of connected DEEPX devices.
- **`DeviceStatus::GetCurrentStatus(i)`** : Returns a `DeviceStatus` object containing all device metrics for device `i`.
- **`device_status.Get...()`** : Instance methods to retrieve individual metrics such as `GetTemperature()`, `GetNpuVoltage()`, `GetNpuClock()`, `GetCoreUtilization()`, `GetMemoryUsed()`, `GetMemoryFree()`.

Profiler Configuration

This guide provides a simple, code-focused manual on how to configure the profiler using the DXRT SDK. The profiler is a powerful tool for analyzing the performance of each layer within your model.

Configuration is managed through the `dxrt::Configuration` singleton instance.

ENABLING THE PROFILER

Before you can use any profiler features, you **must** first enable it. This is the essential first step for any profiling activity.

```
// Enable the profiler feature
dxrt::Configuration::GetInstance().SetEnable(dxrt::Configuration::ITEM::PROFILER, tru
e);
```

- **`SetEnable`** : This function activates or deactivates a specific DXRT feature.
- **`dxrt::Configuration::ITEM::PROFILER`** : Specifies that the target feature is the profiler.
- **`true`** : Enables the profiler. Set to `false` to disable it.

CONFIGURATION OPTIONS

Once enabled, you can set specific attributes for the profiler's behavior.

© Copyright 2026 DEEPX All Rights Reserved.

Displaying Profiler Data in the Console

To see the profiling results printed directly to your console after the inference runs, use the `PROFILER_SHOW_DATA` attribute.

```
// Configure the profiler to print its report to the console
dxrt::Configuration::GetInstance().SetAttribute(dxrt::Configuration::ITEM::PROFILER,
                                                dxrt::Configuration::ATTRIBUTE::PROF
ILER_SHOW_DATA, "ON");
```

- **`SetAttribute`**: Sets a specific property for a DXRT feature.
- **`PROFILER_SHOW_DATA`**: The attribute to control console output.
- **`"ON"`**: A string value to enable this attribute. Use `"OFF"` to disable it.

Saving Profiler Data to a File

To save the profiling report to a file for later analysis, use the `PROFILER_SAVE_DATA` attribute. The resulting report is generated in the same folder with the name **`profiler.json`**.

```
// Configure the profiler to save its report to a file
dxrt::Configuration::GetInstance().SetAttribute(dxrt::Configuration::ITEM::PROFILER,
                                                dxrt::Configuration::ATTRIBUTE::PROF
ILER_SAVE_DATA, "ON");
```

- **`PROFILER_SAVE_DATA`**: The attribute to control file output.
- **`"ON"`**: A string value to enable file saving. Use `"OFF"` to disable it.

COMPLETE CODE EXAMPLE

Here is a complete example showing how to apply all the configurations within a `try-catch` block before creating the `InferenceEngine`.

```
try
{
    // Step 1: Enable the profiler

dxrt::Configuration::GetInstance().SetEnable(dxrt::Configuration::ITEM::PROFILER, tru
e);

    // Step 2: Set attributes to show data in console and save to a file
    dxrt::Configuration::GetInstance().SetAttribute(dxrt::Configuration::ITEM::PROFIL
ER,
    dxrt::Configuration::ATTRIBUTE::PROFILER_SHOW_DATA, "ON");

    dxrt::Configuration::GetInstance().SetAttribute(dxrt::Configuration::ITEM::PROFIL
```


© Copyright 2026 DEEPX All Rights Reserved.

```
ER,
    dxrt::Configuration::ATTRIBUTE::PROFILER_SAVE_DATA, "ON");

    // Step 3: Create the InferenceEngine instance and run inference
    // The profiler will automatically work on the models run by this engine.
    dxrt::InferenceEngine ie(model_path);

    // ... run inference loop ...
}
catch (const dxrt::Exception& e)
{
    // ... handle exceptions ...
}
```

Camera / Inference / Display

The following is an example of a pattern that performs inference using two models on a single camera input and combines the results from both models for display.

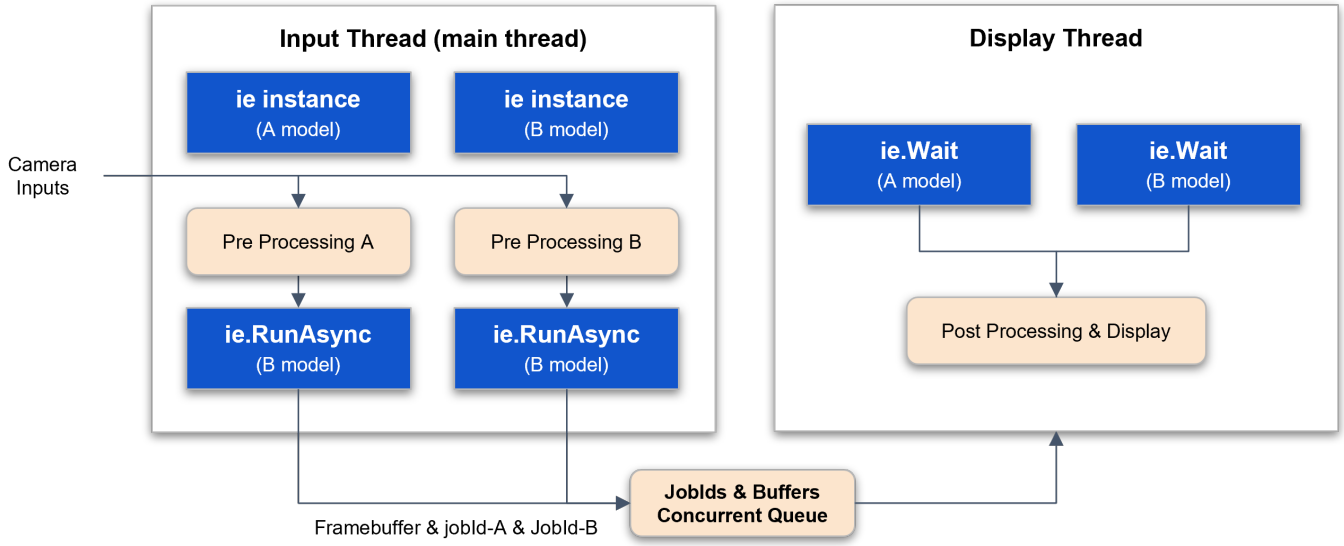

Figure. Multi-model and Multi-output

Multi-model, Async, Wait Thread `(CPU_1 → {NPU_1 + NPU_2} → CPU_2)`

display_async_wait.cpp

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

// input processing main thread with 2 InferenceEngine (asynchronous)
// display thread
```


© Copyright 2026 DEEPX All Rights Reserved.

```
struct FrameJobId {
    int jobId_A = -1;
    int jobId_B = -1;
    void* frameBuffer = nullptr;
    int loopIndex = -1;
};

static const int BUFFER_POOL_SIZE = 10;
static const int QUEUE_SIZE = 10;

static ConcurrentQueue<FrameJobId> gFrameJobIdQueue(QUEUE_SIZE);
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_A;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_B;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gFrameBufferPool;

// total display count
static std::atomic<int> gTotalDisplayCount = {0};

static int displayThreadFunc(int loopCount, dxrt::InferenceEngine& ieA,
dxrt::InferenceEngine& ieB)
{

    while(gTotalDisplayCount.load() < loopCount)
    {
        // consumer framebuffer & jobIds
        auto frameJobId = gFrameJobIdQueue.pop();

        // output data of ieA
        auto outputA = ieA.Wait(frameJobId.jobId_A);

        // output data of ieB
        auto outputB = ieB.Wait(frameJobId.jobId_B);

        // post-processing w/ outputA & outputB
        postProcessing(outputA, outputB);

        gTotalDisplayCount++;

        // display (update framebuffer)
    }

    return 0;
}


int main(int argc, char* argv[])
{
    ...

    try
    {
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        // create inference engine instance with model
        dxrt::InferenceEngine ieA(modelPath_A);

        gInputBufferPool_A =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieA.GetInputSize());

        // create inference engine instance with model
        dxrt::InferenceEngine ieB(modelPath_B);

        gInputBufferPool_B =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieB.GetInputSize());

        const int W = 512, H = 512, CH = 3;
        gFrameBufferPool =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE, W*H*CH);

        // create thread
        std::thread displayThread(displayThreadFunc, loop_count, std::ref(ieA),
std::ref(ieB));

        // input processing
        for(int i = 0; i < loop_count; ++i)
        {
            uint8_t* frameBuffer = gFrameBufferPool->pointer();
            readFrameBuffer(frameBuffer, W, H, CH);

            uint8_t* inputA = gInputBufferPool_A->pointer();
            preProcessing(inputA, frameBuffer);

            uint8_t* inputB = gInputBufferPool_B->pointer();
            preProcessing(inputB, frameBuffer);

            // struct to pass to cpu operation thread
            FrameJobId frameJobId;

            // start inference of A model
            frameJobId.jobId_A = ieA.RunAsync(inputA);

            // start inference of B model
            frameJobId.jobId_B = ieB.RunAsync(inputB);

            // framebuffer used for input data
            frameJobId.frameBuffer = frameBuffer;
            frameJobId.loopIndex = i;

            // producer frame & jobId
            gFrameJobIdQueue.push(frameJobId);

        }
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        displayThread.join();


    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return 0;
}
```

The following is an example of a pattern that sequentially performs operations using two models and CPU processing. The inference result from Model A is processed through CPU computation and then used as input data for Model B. Finally, the result from Model B is handled for display.

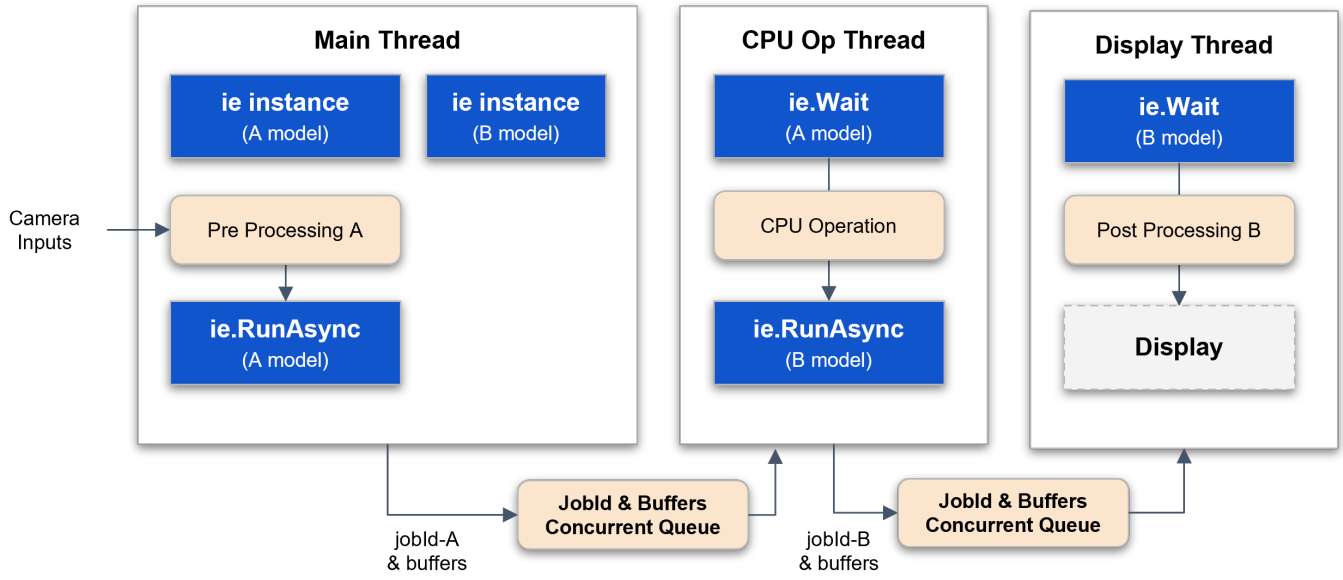

Figure. CPU and NPU Pipeline Operation

Multi-model, Async, Wait Thread `(CPU_1 → NPU_1 → CPU_2 → NPU_2 → CPU_3)`

© Copyright 2026 DEEPX All Rights Reserved.

display_async_pipe.cpp

```
// DX-RT includes
#include <dxrt/dxrt_cxx_api.h>
...

// input main thread
// 1 cpu processing thread
// 1 display thread

struct FrameJobId {
    int jobId_A = -1;
    int jobId_B = -1;
    uint8_t* inputBufferA;
    uint8_t* inputBufferB;
    void* frameBuffer = nullptr;

    int loopIndex;
};

static const int BUFFER_POOL_SIZE = 10;
static const int QUEUE_SIZE = 10;

static ConcurrentQueue<FrameJobId> gCPUOPQueue(QUEUE_SIZE);
static ConcurrentQueue<FrameJobId> gDisplayQueue(QUEUE_SIZE);
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_A;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_B;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gFrameBufferPool;

// total display count
static std::atomic<int> gTotalDisplayCount = {0};


static int displayThreadFunc(int loopCount, dxrt::InferenceEngine& ieB)
{

    while(gTotalDisplayCount.load() < loopCount)
    {
        // consumer framebuffer & jobIds
        auto frameJobId = gDisplayQueue.pop();

        // output data of ieB
        auto outputB = ieB.Wait(frameJobId.jobId_B);

        // post-processing w/ outputA & outputB
        postProcessingB(outputB);

        gTotalDisplayCount++;

        // display (update framebuffer)
        if ( frameJobId.loopIndex == (loopCount - 1)) break;
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    }

    return 0;
}

static int cpuOperationThreadFunc(int loopCount, dxrt::InferenceEngine& ieA,
dxrt::InferenceEngine& ieB)
{

    while(gTotalDisplayCount.load() < loopCount)
    {
        // consumer framebuffer & jobIds
        auto frameJobIdA = gCPUOPQueue.pop();

        // output data of ieA
        auto outputA = ieA.Wait(frameJobIdA.jobId_A);

        // post-processing w/ outputA
        postProcessingA(frameJobIdA.inputBufferB, outputA);

        FrameJobId frameJobIdB;
        frameJobIdB.loopIndex = frameJobIdA.loopIndex;
        frameJobIdB.jobId_B = ieB.RunAsync(frameJobIdA.inputBufferB);

        gDisplayQueue.push(frameJobIdB);

        // display (update framebuffer)
        if ( frameJobIdA.loopIndex == (loopCount - 1)) break;
    }

    return 0;
}


int main(int argc, char* argv[])
{

    ...

    try
    {

        // create inference engine instance with model
        dxrt::InferenceEngine ieA(modelPath);

        gInputBufferPool_A =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieA.GetInputSize());

        // create inference engine instance with model
        dxrt::InferenceEngine ieB(modelPath);
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        gInputBufferPool_B =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieB.GetInputSize());

        const int W = 512, H = 512, CH = 3;
        gFrameBufferPool =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE, W*H*CH);

        // create thread
        std::thread cpuOperationThread(cpuOperationThreadFunc, loop_count,
std::ref(ieA), std::ref(ieB));
        std::thread displayThread(displayThreadFunc, loop_count, std::ref(ieB));


        // input processing
        for(int i = 0; i < loop_count; ++i)
        {
            uint8_t* frameBuffer = gFrameBufferPool->pointer();
            readFrameBuffer(frameBuffer, W, H, CH);

            uint8_t* inputA = gInputBufferPool_A->pointer();
            preProcessing(inputA, frameBuffer);

            // struct to pass to a thread
            FrameJobId frameJobId;

            frameJobId.inputBufferA = inputA;
            frameJobId.inputBufferB = gInputBufferPool_B->pointer();

            // start inference of A model
            frameJobId.jobId_A = ieA.RunAsync(inputA);

            // framebuffer used for input data
            frameJobId.frameBuffer = frameBuffer;
            frameJobId.loopIndex = i;

            // producer frame & jobId
            gCPUOPQueue.push(frameJobId);

        }

        cpuOperationThread.join();
        displayThread.join();

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return 0;
}
```

Exception

The error codes and types of exceptions for error handling are as follows.

```
enum ERROR_CODE {
        DEFAULT = 0x0100,
        FILE_NOT_FOUND,
        NULL_POINTER,
        FILE_IO,
        INVALID_ARGUMENT,
        INVALID_OPERATION,
        INVALID_MODEL,
        MODEL_PARSING,
        SERVICE_IO,
        DEVICE_IO
    };
```

- FileNotFoundException
- NullPointerException
- FileIOException
- InvalidArgumentException
- InvalidOperationException
- InvalidModelException
- ModelParsingException
- ServiceIOException
- DeviceIOException

```
    // try/catch prototype

    try
    {
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        // DX-RT APIs ...
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1; // or std::exit(-1);
    }
    catch(std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;  // or std::exit(-1);
    }
```

Multi-Input Inference

This guide explains various methods for performing inference on multi-input models using the `dxrt::InferenceEngine`. The examples cover different input formats, synchronous and asynchronous execution, and batch processing.

MODEL INFORMATION

Before running inference, it's useful to inspect the model's properties. The `printModelInfo` function shows how to query the inference engine for details about the model's input and output tensors.

- **`ie.IsMultiInputModel()`** : Checks if the loaded model has multiple inputs.
- **`ie.GetInputTensorCount()`** : Gets the number of input tensors.
- **`ie.GetInputTensorNames()`** : Retrieves the names of all input tensors.
- **`ie.GetInputTensorSizes()`** : Gets the size (in bytes) of each input tensor.
- **`ie.GetOutputTensorNames()`** / **`ie.GetOutputTensorSizes()`** : Provide similar information for output tensors.

```
void printModelInfo(dxrt::InferenceEngine& ie) {
    if (ie.IsMultiInputModel()) {
        std::cout << "Input tensor count: " << ie.GetInputTensorCount() << std::endl;
        auto inputNames = ie.GetInputTensorNames();
        auto inputSizes = ie.GetInputTensorSizes();
        for (size_t i = 0; i < inputNames.size(); ++i) {
            std::cout << "  " << inputNames[i] << ": " << inputSizes[i] << " bytes" <
< std::endl;
        }
    }
}
```


© Copyright 2026 DEEPX All Rights Reserved.

SYNCHRONOUS SINGLE INFERENCE

These examples demonstrate different ways to run a single inference request synchronously.

Input Formats

A. Dictionary Format (`std::map<std::string, void*>`)

This is the most robust method. You provide a map where keys are the tensor names and values are pointers to the input data. This format is **not** sensitive to the order of tensors.

- **API**: `ie.RunMultiInput(inputTensors)`
- **Use Case**: Recommended for clarity and to avoid errors from tensor reordering.

```
// Create input data
std::map<std::string, void*> inputTensors;
inputTensors["input_1"] = inputData1.data();
inputTensors["input_2"] = inputData2.data();

// Run inference
auto outputs = ie.RunMultiInput(inputTensors);
```

B. Vector Format (`std::vector<void*>`)

You provide a vector of pointers to the input data. The order of pointers in the vector **must** match the order returned by `ie.GetInputTensorNames()` .

- **API**: `ie.RunMultiInput(inputPtrs)`
- **Use Case**: When tensor order is known and fixed. Can be slightly more performant than the map-based approach due to less overhead.

```
// Create input data in the correct order
std::vector<void*> inputPtrs;
inputPtrs.push_back(inputData1.data()); // Corresponds to first name in
GetInputTensorNames()
inputPtrs.push_back(inputData2.data()); // Corresponds to second name

// Run inference
auto outputs = ie.RunMultiInput(inputPtrs);
```

C. Auto-Split Concatenated Buffer

ⓒ Copyright 2026 DEEPX All Rights Reserved.

You provide a single, contiguous buffer containing all input data concatenated together. The engine automatically splits this buffer into the correct tensor inputs based on their sizes. The concatenation order **must** match the order from `ie.GetInputTensorNames()`.

- **API**: `ie.Run(concatenatedInput.data())`
- **Use Case**: Efficient when input data is already in a single block or when interfacing with systems that provide data this way.

```
// Create a single buffer with all input data concatenated
auto concatenatedInput = createDummyInput(ie.GetInputSize());

// Run inference
auto outputs = ie.Run(concatenatedInput.data());
```

Output Buffer Management

For each synchronous method, you can either let the engine allocate output memory automatically or provide a pre-allocated buffer for performance gains.

- **Auto-Allocated Output (No Buffer Provided)**: Simpler to use. The engine returns smart pointers to newly allocated memory.

```
// Engine allocates and manages output memory
auto outputs = ie.RunMultiInput(inputTensors);
```

- **User-Provided Output Buffer**: More performant as it avoids repeated memory allocations. The user is responsible for allocating a buffer of size `ie.GetOutputSize()`.

```
// User allocates the output buffer
std::vector<uint8_t> outputBuffer(ie.GetOutputSize());

// Run inference, placing results in the provided buffer
auto outputs = ie.RunMultiInput(inputTensors, nullptr, outputBuffer.data());
```

Synchronous Batch Inference

© Copyright 2026 DEEPX All Rights Reserved.

For processing multiple inputs at once to maximize throughput, you can use the batch inference API. This is more efficient than running single inferences in a loop.

- **API**: `ie.Run(batchInputPtrs, batchOutputPtrs, userArgs)`
- **Input**: A vector of pointers, where each pointer is a concatenated buffer for one sample in the batch.
- **Output**: A vector of pointers, where each pointer is a pre-allocated buffer for the corresponding sample's output.

```
int batchSize = 3;
std::vector<void*> batchInputPtrs;
std::vector<void*> batchOutputPtrs;

// Prepare input and output buffers for each sample in the batch
for (int i = 0; i < batchSize; ++i) {
    // Each input is a full concatenated buffer
    batchInputData[i] = createDummyInput(ie.GetInputSize());
    batchInputPtrs.push_back(batchInputData[i].data());

    // Pre-allocate output buffer for each sample
    batchOutputData[i].resize(ie.GetOutputSize());
    batchOutputPtrs.push_back(batchOutputData[i].data());
}

// Run batch inference
auto batchOutputs = ie.Run(batchInputPtrs, batchOutputPtrs);
```

Asynchronous Inference

Asynchronous APIs allow you to submit inference requests without blocking the calling thread. The results are returned later via a callback function. This is ideal for applications that need to remain responsive, such as those with a user interface.

- **APIs**:
  - `ie.RunAsyncMultiInput(inputTensors, userArg)`
  - `ie.RunAsync(concatenatedInput.data(), userArg)`
- **Callback Registration**: `ie.RegisterCallback(callback_function)`

© Copyright 2026 DEEPX All Rights Reserved.

The `AsyncInferenceHandler` class demonstrates how to manage state across multiple asynchronous calls.

- **Register a Callback**: Provide a function that the engine will call upon completion of each async request. The callback receives the output tensors and a `userArg` pointer for context.
- **Submit Requests**: Call an `RunAsync` variant. This call returns immediately with a job ID.
- **Process in Callback**: The callback function is executed in a separate worker thread. Here, you can process the results. It's crucial to ensure thread safety if you modify shared data.

```
// 1. Create a handler and register its callback method
AsyncInferenceHandler handler(asyncCount);
ie.RegisterCallback([&handler](dxrt::TensorPtrs& outputs, void* userArg) -> int {
    return handler.callback(outputs, userArg);
});

// 2. Submit multiple async requests in a loop
for (int i = 0; i < asyncCount; ++i) {
    void* userArg = reinterpret_cast<void*>(static_cast<uintptr_t>(i));
    // Each call is non-blocking
    ie.RunAsyncMultiInput(asyncInputTensors[i], userArg);
}

// 3. Wait for all callbacks to complete
handler.waitForCompletion();

// 4. Clear the callback when done
ie.RegisterCallback(nullptr);
```

Examples

The DX-RT SDK provides comprehensive C++ examples in the `examples/cpp/` directory. These examples demonstrate various inference patterns, configuration options, and performance optimization techniques. All examples are ready to build and run.

Location: `examples/cpp/`

BASIC INFERENCE EXAMPLES

- **`run_sync_model`**
  Demonstrates synchronous inference using a single thread. This is the simplest way to perform inference, suitable for sequential processing.

© Copyright 2026 DEEPX All Rights Reserved.

- **`run_async_model`**
  Performance-optimized asynchronous inference using callback functions. Demonstrates how to maximize NPU utilization with non-blocking execution.
- **`run_async_model_wait`**
  Asynchronous inference using the `Wait()` method instead of callbacks, providing more control over output retrieval timing.
- **`run_batch_model`**
  Shows how to perform batch inference to process multiple inputs simultaneously, improving throughput for batch-oriented workloads.

CONFIGURATION EXAMPLES

- **`run_sync_model_bound`**
  Demonstrates how to bind inference to specific NPU cores using `InferenceOption.boundOption` for fine-grained resource control.
- **`run_async_model_bound`**
  Asynchronous version of NPU core binding, showing how to control which NPU cores execute async tasks.
- **`run_sync_model_bufcount`**
  Illustrates buffer count configuration with `InferenceOption.bufferCount` to optimize memory usage in synchronous mode.
- **`run_async_model_bufcount`**
  Shows buffer count tuning for asynchronous inference to balance throughput and memory consumption.
- **`run_async_model_conf`**
  Comprehensive example of using `dxrt::Configuration` to set runtime parameters and monitor device status.

MEMORY MANAGEMENT EXAMPLES

- **`run_sync_model_memory`**
  Demonstrates loading models from memory buffers instead of files, useful for embedded systems or when models are loaded from custom sources.
- **`run_sync_model_output`**
  Shows how to provide pre-allocated output buffers to the inference engine, giving you control over output memory management.
- **`run_async_model_output`**
  Asynchronous inference with user-managed output buffers, demonstrating zero-copy output handling.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

PROFILING EXAMPLES

- **`run_async_model_profiler`**
  Demonstrates how to enable and use the DX-RT profiler to collect detailed timing information for performance analysis.

MULTI-INPUT MODEL EXAMPLES

- **`multi_input_model_inference`**
  Shows how to handle models with multiple input tensors, including proper buffer preparation and data layout.

ADVANCED PIPELINE EXAMPLES

- **`display_async_pipe`**
  Pipeline pattern: `[CPU_1 → {NPU_1 + NPU_2} → CPU_2]`
  Demonstrates preprocessing on CPU, parallel NPU execution, and postprocessing.
- **`display_async_wait`**
  Sequential pattern: `[CPU_1 → NPU_1 → CPU_2 → NPU_2 → CPU_3]`
  Shows how to chain multiple processing stages with intermediate CPU tasks.
- **`display_async_thread`**
  Single model with multiple threads, demonstrating concurrent inference requests to the same model.
- **`display_async_models_1`**
  Multi-model multi-thread example where each thread creates its own `InferenceEngine` instance.
- **`display_async_models_2`**
  Multi-model multi-thread example where `InferenceEngine` instances are created in the main thread and shared.

BUILDING AND RUNNING EXAMPLES

All examples can be built using CMake:

```
cd examples/cpp
mkdir build && cd build
cmake ..
make

# Run an example (replace with actual model path)
./run_sync_model -m /path/to/model.dxnn -l 100
./run_async_model -m /path/to/model.dxnn -l 1000 -v
./run_batch_model -m /path/to/model.dxnn -b 4 -l 100
```


© Copyright 2026 DEEPX All Rights Reserved.

Most examples support the following common options: - `-m, --model` : Path to the model file (.dxnn) - `-l, --loops` : Number of inference iterations - `-v, --verbose` : Enable verbose/debug logging - `-h, --help` : Display usage information

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.2 Python Tutorials

This section offers Python tutorials demonstrating inference, configuration, and multi-input handling using the DXRT SDK. These examples are ideal for quick prototyping and integration into Python-based AI workflows.

---

run (Synchronous)

The synchronous Run method uses a single NPU core to perform inference in a blocking manner. It can be configured to utilize multiple NPU cores simultaneously by employing threads to run each core independently. (Refer to **Figure** in **Section. Inference Workflow**)

Inference Engine Run (Python)

run_sync_model.py

```
# DX-RT imports
from dx_engine import InferenceEngine
...


if __name__ == "__main__":
    ...

    # create inference engine instance with model
    ie = InferenceEngine(modelPath)

    input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

    # inference loop
    for i in range(loop_count):

        # inference synchronously
        # use only one npu core
        outputs = ie.run(input)

        # post processing
        postProcessing(outputs)

    exit(0)
```

ⓒ Copyright 2026 DEEPX All Rights Reserved.

run_async (Asynchronous)

The asynchronous Run mode is a method that performs inference asynchronously while utilizing multiple NPU cores simultaneously. It can be implemented to maximize NPU resources through a callback function or a thread wait mechanism.

Inference Engine RunAsync, Callback, User Argument

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

The following is an example of asynchronous inference using a callback function. A user argument can be used to synchronize the input with the output of the callback.

Inference Engine RunAsync, Callback, User Argument (Python)

`run_async_model.py`

```
from dx_engine import InferenceEngine
...

q = queue.Queue()
gLoopCount = 0

lock = threading.Lock()

def onInferenceCallbackFunc(outputs, user_arg):

    # the outputs are guaranteed to be valid only within this callback function
    # processing this callback functions as quickly as possible is beneficial
    # for improving inference performance

    global gLoopCount

    # Mutex locks should be properly adjusted
    # to ensure that callback functions are thread-safe.
    with lock:
        # user data type casting
        index, loop_count = user_arg


        # post processing
        #postProcessing(outputs);

        # something to do
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        print("Inference output (callback) index=", index)

        gLoopCount += 1
        if ( gLoopCount == loop_count ) :
            print("Complete Callback")
            q.put(0)

    return 0


if __name__ == "__main__":

    ...

    # create inference engine instance with model
    ie = InferenceEngine(modelPath)

    # register call back function
    ie.register_callback(onInferenceCallbackFunc)


    input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

    # inference loop
    for i in range(loop_count):

        # inference asynchronously, use all npu cores
        # if device-load >= max-load-value, this function will block
        ie.run_async(input, user_arg=[i, loop_count])

        print("Inference start (async)", i)

    exit(q.get())
```

The following is an example where multiple threads start input and inference, and a single callback processes the output.

Inference Engine RunAsync, Callback, User Argument, Thread

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

Inference Engine run_async, Callback, User Argument, Thread (Python)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

run_async_model_thread.py

```
from dx_engine import InferenceEngine
...

THREAD_COUNT = 3
total_count = 0
q = queue.Queue()

lock = threading.Lock()


def inferenceThreadFunc(ie, threadIndex, loopCount):

    # input
    input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

    # inference loop
    for i in range(loopCount):

        # inference asynchronously, use all npu cores
        # if device-load >= max-load-value, this function will block
        ie.run_async(input,user_arg = [i, loopCount, threadIndex])

    return 0

def onInferenceCallbackFunc(outputs, user_arg):
    # the outputs are guaranteed to be valid only within this callback function
    # processing this callback functions as quickly as possible is beneficial
    # for improving inference performance

    global total_count

    # Mutex locks should be properly adjusted
    # to ensure that callback functions are thread-safe.
    with lock:
        # user data type casting
        index = user_arg[0]
        loop_count = user_arg[1]
        thread_index = user_arg[2]

        # post processing
        #postProcessing(outputs);

        # something to do

        total_count += 1

        if ( total_count == loop_count * THREAD_COUNT) :
            q.put(0)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    return 0


if __name__ == "__main__":
    ...


    # create inference engine instance with model
    ie = InferenceEngine(modelPath)

    # register call back function
    ie.register_callback(onInferenceCallbackFunc)


    t1 = threading.Thread(target=inferenceThreadFunc, args=(ie, 0, loop_count))
    t2 = threading.Thread(target=inferenceThreadFunc, args=(ie, 1, loop_count))
    t3 = threading.Thread(target=inferenceThreadFunc, args=(ie, 2, loop_count))

    # Start and join
    t1.start()
    t2.start()
    t3.start()


    # join
    t1.join()
    t2.join()
    t3.join()


    exit(q.get())
```

The following is an example of performing asynchronous inference by creating an inference wait thread. The main thread starts input and inference, and the inference wait thread retrieves the output data corresponding to the input.

Inference Engine run_async, wait

- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

Inference Engine run_async, wait (Python)

`run_async_model_wait.py`

```
# DX-RT imports
from dx_engine import InferenceEngine
...
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
q = queue.Queue()


def inferenceThreadFunc(ie, loopCount):

    count = 0

    while(True):

        # pop item from queue
        jobId = q.get()

        # waiting for the inference to complete by jobId
        # ownership of the outputs is transferred to the user
        outputs = ie.wait(jobId)

        # post processing
        # postProcessing(outputs);

        # something to do

        count += 1
        if ( count >= loopCount ):
            break

    return 0


if __name__ == "__main__":
    ...

    # create inference engine instance with model
    with InferenceEngine(modelPath) as ie:

        # do not register call back function
        # ie.register_callback(onInferenceCallbackFunc)

        t1 = threading.Thread(target=inferenceThreadFunc, args=(ie, loop_count))

        t1.start()

        input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

        # inference loop
        for i in range(loop_count):

            # inference asynchronously, use all npu cores
            # if device-load >= max-load-value, this function will block
            jobId = ie.run_async(input, user_arg=0)

            q.put(jobId)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
        t1.join()

    exit(0)
```

run (Batch)

The following is an example of batch inference with multiple inputs and multiple outputs.

run_batch_model.py

```
import numpy as np
import sys
from dx_engine import InferenceEngine
from dx_engine import InferenceOption


if __name__ == "__main__":
    ...

    # create inference engine instance with model
    with InferenceEngine(modelPath) as ie:

        input_buffers = []
        output_buffers = []
        index = 0
        for b in range(batch_count):
            input_buffers.append([np.array([np.random.randint(0, 255)],
dtype=np.uint8)])
            output_buffers.append([np.zeros(ie.get_output_size(), dtype=np.uint8)])
            index = index + 1

        # inference loop
        for i in range(loop_count):

            # batch inference
            # It operates asynchronously internally
            # for the specified number of batches and returns the results
            results = ie.run(input_buffers, output_buffers)

            # post processing

    exit(0)
```

Inference Option

The following inference options allow you to specify an NPU core for performing inference.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Inference Engine Run, Inference Option

- select devices

  default device is `[]`

  Choose the device to utilize (ex. `[0, 2]`)

- select bound option per device

  `InferenceOption.BOUND_OPTION.NPU_ALL`

  `InferenceOption.BOUND_OPTION.NPU_0`

  `InferenceOption.BOUND_OPTION.NPU_1`

  `InferenceOption.BOUND_OPTION.NPU_2`

  `InferenceOption.BOUND_OPTION.NPU_01`

  `InferenceOption.BOUND_OPTION.NPU_12`

  `InferenceOption.BOUND_OPTION.NPU_02`

- use onnx runtime library (`ORT`)

  `set_use_ort / get_use_ort`

NPU_ALL / NPU_0 / NPU_1 / NPU_2

```
# DX-RT imports
from dx_engine import InferenceEngine, InferenceOption
...

if __name__ == "__main__":
    ...

    # inference option
    option = InferenceOption()

    print("Inference Options:")

    # select devices
    option.devices = [0]
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    # NPU bound option (NPU_ALL or NPU_0 or NPU_1 or NPU_2)
    option.bound_option = InferenceOption.BOUND_OPTION.NPU_ALL

    # use ONNX Runtime (True or False)
    option.use_ort = False

    # create inference engine instance with model
    with InferenceEngine(modelPath, option) as ie:

        input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

        # inference loop
        for i in range(loop_count):

            # inference synchronously
            # use only one npu core
            # ownership of the outputs is transferred to the user
            outputs = ie.run(input)

            # post processing
            #postProcessing(outputs)
            print("Inference outputs ", i)

    exit(0)
```

> **WARNING**
>
> If multiple InferenceEngine instances are initialized with specific NPU bindings (occupying NPUs 0, 1, and 2), any subsequent attempt to instantiate a new engine using BOUND_OPTION.NPU_ALL will enter a blocking state. The new instance will remain suspended until all required NPU resources become available. This synchronization behavior is consistent across both multi-threaded and multi-process environments.

Configuration and DeviceStatus

This guide explains how to use the `Configuration` class to set up the inference engine and the `DeviceStatus` class to monitor hardware status.

ENGINE CONFIGURATION

The `Configuration` class allows you to set engine parameters and retrieve version information before running inference.

```
# Create a configuration object
config = Configuration()
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# Enable options like showing model details or profiling information
config.set_enable(Configuration.ITEM.SHOW_MODEL_INFO, True)
config.set_enable(Configuration.ITEM.SHOW_PROFILE, True)

# Retrieve version information
logger.info('Runtime framework version: ' + config.get_version())
logger.info('Device driver version: ' + config.get_driver_version())
```

- **`Configuration()`**: Creates an object to manage engine settings.
- **`config.set_enable(...)`**: Turns specific engine features on or off. In this case, it enables printing model information and performance profiles upon loading.
- **`config.get_version()`**: Fetches read-only information, such as software and driver versions.

QUERYING DEVICE STATUS

The `DeviceStatus` class provides real-time device information including voltage, clock, temperature, NPU core utilization, and DRAM memory usage.

```
# Get the number of available devices
device_count = DeviceStatus.get_device_count()

# Loop through each device
for i in range(device_count):
    # Get a status object for the current device
    device_status = DeviceStatus.get_current_status(i)
    logger.info(f'Device {device_status.get_id()}')

    # Loop through each NPU core to get its metrics
    for c in range(3):
        logger.info(
            f'   NPU Core {c} '
            f'Temperature: {device_status.get_temperature(c)} '
            f'Voltage: {device_status.get_npu_voltage(c)} '
            f'Clock: {device_status.get_npu_clock(c)}'
        )

    # Runtime usage (requires dxrtd service)
    logger.info(f'   Core Utilization: {device_status.get_core_utilization(0):.1f}%')
    logger.info(f'   Memory Used: {device_status.get_memory_used()} bytes')
    logger.info(f'   Memory Free: {device_status.get_memory_free()} bytes')
```

- **`DeviceStatus.get_device_count()`**: Returns the number of connected DEEPX devices.
- **`DeviceStatus.get_current_status(i)`**: Returns a `DeviceStatus` object with all device metrics.
- **`device_status.get_temperature(c)`**, **`get_npu_voltage(c)`**, **`get_npu_clock(c)`**: Hardware status.
- **`device_status.get_core_utilization(c)`**, **`get_memory_used()`**, **`get_memory_free()`**: Runtime usage data.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Profiler Configuration

This guide provides a simple, code-focused manual on how to configure the profiler using the DXRT Python wrapper. The profiler is a powerful tool for analyzing the performance of each layer within your model.

Configuration is managed through an instance of the `Configuration` class.

ENABLING THE PROFILER

Before you can use any profiler features, you **must** first create a `Configuration` object and enable the profiler. This is the essential first step.

```
# Create a Configuration instance
config = Configuration()

# Enable the profiler feature
config.set_enable(Configuration.ITEM.PROFILER, True)
```

- **`config = Configuration()`** : Creates the object that controls system-wide settings for the runtime.
- **`set_enable()`** : This method activates or deactivates a specific DXRT feature.
- **`Configuration.ITEM.PROFILER`** : Specifies that the target feature is the profiler.
- **`True`** : Enables the profiler. Set to `False` to disable it.

CONFIGURATION OPTIONS

Once enabled, you can set specific attributes for the profiler's behavior using the same `config` object.

Displaying Profiler Data in the Console

To see the profiling results printed directly to your console after the inference runs, use the `PROFILER_SHOW_DATA` attribute.

```
# Configure the profiler to print its report to the console
config.set_attribute(Configuration.ITEM.PROFILER,
                     Configuration.ATTRIBUTE.PROFILER_SHOW_DATA, "ON")
```

- **`set_attribute()`** : Sets a specific property for a DXRT feature.
- **`PROFILER_SHOW_DATA`** : The attribute to control console output.
- **`"ON"`** : A string value to enable this attribute. Use `"OFF"` to disable it.

© Copyright 2026 DEEPX All Rights Reserved.

Saving Profiler Data to a File

To save the profiling report to a file for later analysis, use the `PROFILER_SAVE_DATA` attribute. The resulting report is generated in the same folder with the name **`profiler.json`**.

```
# Configure the profiler to save its report to a file
config.set_attribute(Configuration.ITEM.PROFILER,
                     Configuration.ATTRIBUTE.PROFILER_SAVE_DATA, "ON")
```

- **`PROFILER_SAVE_DATA`**: The attribute to control file output.
- **`"ON"`**: A string value to enable file saving. Use `"OFF"` to disable it.

COMPLETE CODE EXAMPLE

Here is a complete example showing how to apply all the configurations at the start of your script. These settings are applied globally, and any `InferenceEngine` instance created afterward will automatically use them.

```
if __name__ == "__main__":

    # Step 1: Create a Configuration instance and enable the profiler
    config = Configuration()
    config.set_enable(Configuration.ITEM.PROFILER, True)

    # Step 2: Set attributes to show data in console and save to a file
    config.set_attribute(Configuration.ITEM.PROFILER,
                         Configuration.ATTRIBUTE.PROFILER_SHOW_DATA, "ON")
    config.set_attribute(Configuration.ITEM.PROFILER,
                         Configuration.ATTRIBUTE.PROFILER_SAVE_DATA, "ON")

    # The configuration is now active.
    # ...

    # Create an inference engine instance that will now be profiled
    with InferenceEngine(modelPath) as ie:
        # ... register callback and run inference loop ...
```

Multi-input Inference

This guide explains various methods for performing inference on multi-input models using the `InferenceEngine`. The examples cover different input formats, synchronous and asynchronous execution, and batch processing.

© Copyright 2026 DEEPX All Rights Reserved.

MODEL INFORMATION

Before running inference, it's useful to inspect the model's properties. The `print_model_info` function in the example script shows how to query the inference engine for details about the model's input and output tensors.

- **`ie.is_multi_input_model()`** : Checks if the loaded model has multiple inputs.
- **`ie.get_input_tensor_count()`** : Gets the number of input tensors.
- **`ie.get_input_tensor_names()`** : Retrieves the names of all input tensors.
- **`ie.get_input_tensor_sizes()`** : Gets the size (in bytes) of each input tensor.
- **`ie.get_output_tensor_names()` / `ie.get_output_tensor_sizes()`** : Provide similar information for output tensors.

```
def print_model_info(ie: InferenceEngine) -> None:
    if ie.is_multi_input_model():
        print(f"Input tensor count: {ie.get_input_tensor_count()}")
        input_names = ie.get_input_tensor_names()
        input_sizes = ie.get_input_tensor_sizes()
        for i, name in enumerate(input_names):
            print(f"  {name}: {input_sizes[i]} bytes")
```

SYNCHRONOUS SINGLE INFERENCE

These examples demonstrate different ways to run a single inference request synchronously.

Input Formats

A. Dictionary Format (`Dict[str, np.ndarray]`)

This is the most robust method. You provide a dictionary where keys are the tensor names and values are the `numpy` arrays. This format is **not** sensitive to the order of tensors.

- **API**: `ie.run_multi_input(input_tensors)`
- **Use Case**: Recommended for clarity and to avoid errors from tensor reordering.

```
# Create input data
input_names = ie.get_input_tensor_names()
input_sizes = ie.get_input_tensor_sizes()
input_tensors = {name: create_dummy_input(size) for name, size in zip(input_names, in
put_sizes)}

# Run inference
outputs = ie.run_multi_input(input_tensors)
```


© Copyright 2026 DEEPX All Rights Reserved.

B. List Format (`List[np.ndarray]`)

You provide a list of `numpy` arrays. The order of arrays in the list **must** match the order returned by `ie.get_input_tensor_names()` .

- **API**: `ie.run(input_list)`
- **Use Case**: When tensor order is known and fixed. Can be slightly more performant than the dictionary-based approach due to less overhead.

```
# Create input data in the correct order
input_sizes = ie.get_input_tensor_sizes()
input_list = [create_dummy_input(size) for size in input_sizes]

# Run inference
outputs = ie.run(input_list)
```

C. Auto-Split Concatenated Buffer

You provide a single, contiguous `numpy` array containing all input data concatenated together. The engine automatically splits this buffer into the correct tensor inputs based on their sizes. The concatenation order **must** match the order from `ie.get_input_tensor_names()` .

- **API**: `ie.run(concatenated_input)`
- **Use Case**: Efficient when input data is already in a single block or when interfacing with systems that provide data this way.

```
# Create a single buffer with all input data concatenated
total_input_size = ie.get_input_size()
concatenated_input = create_dummy_input(total_input_size)

# Run inference
outputs = ie.run(concatenated_input)
```

Output Buffer Management

© Copyright 2026 DEEPX All Rights Reserved.

For each synchronous method, you can either let the engine allocate output memory automatically or provide pre-allocated buffers for performance gains.

- **Auto-Allocated Output (No Buffer Provided)**: Simpler to use. The engine returns a new list of `numpy` arrays.

```
# Engine allocates and manages output memory
outputs = ie.run_multi_input(input_tensors)
```

- **User-Provided Output Buffers**: More performant as it avoids repeated memory allocations. The user is responsible for creating a list of `numpy` arrays with the correct sizes.

```
# User creates the output buffers
output_sizes = ie.get_output_tensor_sizes()
output_buffers = [np.zeros(size, dtype=np.uint8) for size in output_sizes]

# Run inference, placing results in the provided buffers
outputs = ie.run_multi_input(input_tensors, output_buffers=output_buffers)
```

SYNCHRONOUS BATCH INFERENCE

For processing multiple inputs at once to maximize throughput, you can use the batch inference capabilities of the `run` method. This is more efficient than running single inferences in a loop.

A. Explicit Batch Format (`List[List[np.ndarray]]`)

This is the clearest way to represent a batch. The input is a list of lists, where the outer list represents the batch and each inner list contains all input tensors for a single sample.

- **API**: `ie.run(batch_inputs, output_buffers=...)`
- **Input**: A `List[List[np.ndarray]]` .
- **Output**: A `List[List[np.ndarray]]` .

```
batch_size = 3
input_sizes = ie.get_input_tensor_sizes()
batch_inputs = []
for i in range(batch_size):
    sample_inputs = [create_dummy_input(size) for size in input_sizes]
    batch_inputs.append(sample_inputs)

# Output buffers must also match the batch structure
# ... create batch_outputs ...
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# Run batch inference
results = ie.run(batch_inputs, output_buffers=batch_outputs)
```

B. Flattened Batch Format (`List[np.ndarray]`)

As a convenience, the API can also accept a single "flattened" list of `numpy` arrays. The total number of arrays **must** be a multiple of the model's input tensor count. The engine will automatically group them into batches.

- **API**: `ie.run(flattened_inputs, output_buffers=...)`
- **Input**: A `List[np.ndarray]` containing `batch_size * num_input_tensors` arrays.
- **Output**: The result is still returned in the explicit batch format (`List[List[np.ndarray]]`).

```
batch_size = 3
input_sizes = ie.get_input_tensor_sizes()
flattened_inputs = []
for i in range(batch_size):
    for size in input_sizes:
        flattened_inputs.append(create_dummy_input(size))

# ... create flattened_output_buffers ...

# Run batch inference
results = ie.run(flattened_inputs, output_buffers=flattened_output_buffers)
```

ASYNCHRONOUS INFERENCE

Asynchronous APIs allow you to submit inference requests without blocking the calling thread. The results are returned later via a callback function. This is ideal for applications that need to remain responsive.

- **APIs**:
  - `ie.run_async_multi_input(input_tensors, user_arg=...)`
  - `ie.run_async(input_data, user_arg=...)`
- **Callback Registration**: `ie.register_callback(callback_function)`

© Copyright 2026 DEEPX All Rights Reserved.

The `AsyncInferenceHandler` class in the example demonstrates how to manage state across multiple asynchronous calls.

- **Register a Callback**: Provide a function that the engine will call upon completion of each async request. The callback receives the output arrays and a `user_arg` for context.
- **Submit Requests**: Call an `run_async` variant. This call returns immediately with a job ID.
- **Process in Callback**: The callback function is executed in a separate worker thread. Here, you can process the results. It's crucial to ensure thread safety (e.g., using a `threading.Lock`) if you modify shared data.

```
# 1. Create a handler and register its callback method
handler = AsyncInferenceHandler(async_count)
ie.register_callback(handler.callback)

# 2. Submit multiple async requests in a loop
for i in range(async_count):
    user_arg = f"async_sample_{i}"
    # Each call is non-blocking
    job_id = ie.run_async_multi_input(input_tensors, user_arg=user_arg)

# 3. Wait for all callbacks to complete
handler.wait_for_completion()

# 4. Clear the callback when done
ie.register_callback(None)
```

Examples

The DX-RT SDK provides comprehensive Python examples in the `examples/python/` directory. These examples demonstrate various inference patterns, configuration options, and performance optimization techniques using the Python API. All examples are ready to run.

Location: `examples/python/`

BASIC INFERENCE EXAMPLES

- **`run_sync_model.py`**
  Demonstrates synchronous inference using a single thread. This is the simplest way to perform inference in Python, suitable for sequential processing and prototyping.
- **`run_async_model.py`**
  Performance-optimized asynchronous inference using callback functions. Shows how to maximize NPU utilization with non-blocking execution and thread-safe callback handling.

© Copyright 2026 DEEPX All Rights Reserved.

- **`run_async_model_callback.py`**
  Additional asynchronous inference example focusing on callback pattern variations and user argument passing for input-output synchronization.
- **`run_async_model_wait.py`**
  Asynchronous inference using the `wait()` method instead of callbacks, providing more control over output retrieval timing and job management.
- **`run_async_model_thread.py`**
  Single inference engine with multiple threads using callbacks. Demonstrates how to handle multiple concurrent inputs to a single model efficiently.
- **`run_batch_model.py`**
  Shows how to perform batch inference to process multiple inputs simultaneously, improving throughput for batch-oriented workloads.

CONFIGURATION EXAMPLES

- **`run_sync_model_bound.py`**
  Demonstrates how to bind inference to specific NPU cores using `InferenceOption.bound_option` for fine-grained resource control.
- **`run_sync_model_bufcount.py`**
  Illustrates buffer count configuration with `InferenceOption.buffer_count` to optimize memory usage and inference pipeline depth.
- **`run_async_model_bufcount.py`**
  Shows buffer count tuning for asynchronous inference to balance throughput and memory consumption in high-performance scenarios.
- **`run_async_model_conf.py`**
  Comprehensive example of using `dx_engine.Configuration` to set runtime parameters, monitor device status, and control logging behavior.

MEMORY MANAGEMENT EXAMPLES

- **`run_sync_model_memory.py`**
  Demonstrates loading models from memory buffers (bytes) instead of files, useful for embedded systems or when models are loaded from custom sources.

PROFILING EXAMPLES

- **`run_async_model_profiler.py`**
  Demonstrates how to enable and use the DX-RT profiler to collect detailed timing information for performance analysis and bottleneck identification.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

MULTI-INPUT MODEL EXAMPLES

- `multi_input_model_inference.py`
  Shows how to handle models with multiple input tensors, including proper buffer preparation, numpy array handling, and correct data layout.

RUNNING EXAMPLES

All examples support command-line arguments for flexibility:

```
# Basic synchronous inference
python examples/python/run_sync_model.py -m /path/to/model.dxnn -l 100

# Asynchronous inference with verbose output
python examples/python/run_async_model.py -m /path/to/model.dxnn -l 1000 -v

# Batch inference
python examples/python/run_batch_model.py -m /path/to/model.dxnn -b 4 -l 100

# Bound to specific NPU core
python examples/python/run_sync_model_bound.py -m /path/to/model.dxnn

# With profiling enabled
python examples/python/run_async_model_profiler.py -m /path/to/model.dxnn -l 500
```

Common Command-Line Options: - `-m, --model` : Path to the model file (.dxnn) - **required** - `-l, --loops` : Number of inference iterations (default: 1) - `-b, --batch` : Batch count for batch inference (default: 1) - `-v, --verbose` : Enable verbose/debug logging - `-h, --help` : Display usage information

Prerequisites:

Ensure the DX-RT Python package is installed:

```
pip install dx_engine
```


© Copyright 2026 DEEPX All Rights Reserved.

10. API Reference

10.1 C++ API Reference

This section documents the core C++ classes provided by the DX-RT SDK. It includes detailed descriptions of the inference engine, configuration options, device status monitoring, and tensor structures. These APIs are designed for high-performance, real-time applications and offer fine-grained control over NPU execution.

Header

Header	Description
`<dxrt/dxrt_cxx_api.h>`	**Recommended.** Single self-contained C++14 header for all applications.
`<dxrt/dxrt_api.h>`	Legacy header — still works, no source changes needed. Emits a compile-time maintenance note by default (`DXRT_LEGACY_HEADER_OK` to suppress).

> **NOTE**
>
> Internal headers (e.g., `dxrt/internal/*.h`) are no longer publicly accessible. Code that directly included internal headers will get compile errors — switch to `dxrt_cxx_api.h` .
> Also, do not include `<dxrt/dxrt_api.h>` and `<dxrt/dxrt_cxx_api.h>` in the same translation unit.

---

class dxrt::InferenceEngine

This class abstracts the runtime inference executor for a user's compiled model. After a model is loaded, real-time device tasks are scheduled by internal runtime libraries. It supports both synchronous and asynchronous inference modes.

© Copyright 2026 DEEPX All Rights Reserved.

CONSTRUCTOR

`explicit InferenceEngine(const std::string &modelPath, InferenceOption &option = DefaultInferenceOption)`

- **Description**: Loads a model from the specified file path and configures the NPU to run it.
- **Parameters**:
  - `modelPath`: The file path to the compiled model (e.g., `model.dxnn`).
  - `option`: A reference to an `InferenceOption` object to configure devices and NPU cores. If not provided, uses `DefaultInferenceOption`.
- **Example**:

```
// Load model from file path
dxrt::InferenceEngine ie("model.dxnn");

// Load model with custom options
dxrt::InferenceOption option;
option.devices = {0, 1};
option.boundOption = dxrt::InferenceOption::NPU_ALL;
dxrt::InferenceEngine ie2("model.dxnn", option);
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

`explicit InferenceEngine(const uint8_t* modelBuffer, size_t modelSize, InferenceOption &option = DefaultInferenceOption)`

- **Description**: Loads a model from a memory buffer and configures the NPU to run it. This constructor is useful for embedded systems or when models are loaded from custom sources (e.g., encrypted storage, network).
- **Parameters**:
  - `modelBuffer` : A pointer to the compiled model data in memory.
  - `modelSize` : The size of the model data in bytes.
  - `option` : A reference to an `InferenceOption` object to configure devices and NPU cores. If not provided, uses `DefaultInferenceOption` .
- **Example**:

```
// Load model file into memory buffer
std::ifstream file("model.dxnn", std::ios::binary | std::ios::ate);
std::streamsize size = file.tellg();
file.seekg(0, std::ios::beg);
std::vector<uint8_t> buffer(size);
if (file.read(reinterpret_cast<char*>(buffer.data()), size)) {
    // Create inference engine from memory buffer
    dxrt::InferenceEngine ie(buffer.data(), buffer.size());

    // With custom options
    dxrt::InferenceOption option;
    option.useORT = true;
    dxrt::InferenceEngine ie2(buffer.data(), buffer.size(), option);
}
```

- **Use Cases**:
  - Loading models from encrypted or compressed storage
  - Network-based model distribution
  - Embedded systems with models stored in ROM
  - Dynamic model loading without filesystem access

MEMBER FUNCTIONS

`Dispose()`

- **Signature**: `void Dispose()`

- **Description**: Deallocates resources and performs cleanup. This should be called to release memory and handles held by the engine.

© Copyright 2026 DEEPX All Rights Reserved.

`GetAllTaskOutputs()`
- **Signature**: `std::vector<TensorPtrs> GetAllTaskOutputs()`
- **Description**: Retrieves the output tensors of all internal tasks in the model.
- **Returns**: A vector of `TensorPtrs`, where each element represents the outputs of a single task.
- **Note**: The legacy function `get_outputs()` is deprecated.

`GetBitmatchMask(int index)`
- **Signature**: `std::vector<uint8_t> GetBitmatchMask(int index)`
- **Description**: An internal function to get the bitmatch mask for a given NPU task index.
- **Parameters**:
- `index`: The index of the NPU task.
- **Returns**: A vector of `uint8_t` representing the mask.
- **Note**: The legacy function `bitmatch_mask(int index)` is deprecated.

`GetCompileType()`
- **Signature**: `std::string GetCompileType()`
- **Description**: Returns the compile type of the loaded model.
- **Returns**: The compile type as a `std::string`.
- **Note**: The legacy function `get_compile_type()` is deprecated.

`GetInputSize()`
- **Signature**: `uint64_t GetInputSize()`
- **Description**: Gets the total size of all input tensors combined in bytes.
- **Returns**: The total input size as a `uint64_t`.
- **Note**: The legacy function `input_size()` is deprecated.

`GetInputs(void *ptr = nullptr, uint64_t phyAddr = 0)`
- **Signature**: `Tensors GetInputs(void *ptr = nullptr, uint64_t phyAddr = 0)`
- **Description**: Retrieves the input tensors for the model. If `ptr` is null, it returns information about the input memory area within the engine. If `ptr` and `phyAddr` are provided, it returns tensor objects pointing to those addresses.
- **Parameters**:
- `ptr`: An optional pointer to a virtual address for the input data.
- `phyAddr`: An optional pointer to a physical address for the input data.
- **Returns**: A `Tensors` (vector of `Tensor`) object.
- **Note**: The legacy function `inputs(...)` is deprecated.

`GetInputs(int devId)`
- **Signature**: `std::vector<Tensors> GetInputs(int devId)`
- **Description**: Retrieves the input tensors for a specific device ID.
- **Parameters**:

© Copyright 2026 DEEPX All Rights Reserved.

- `devId` : The ID of the device.
- **Returns**: A vector of `Tensors` objects.
- **Note**: The legacy function `inputs(int devId)` is deprecated.

`GetInputTensorCount()`
- **Signature**: `int GetInputTensorCount() const`
- **Description**: Returns the number of input tensors required by the model.
- **Returns**: The count of input tensors.

`GetInputTensorNames()`
- **Signature**: `std::vector<std::string> GetInputTensorNames() const`
- **Description**: Returns the names of all input tensors in the order they should be provided.
- **Returns**: A vector of input tensor names.

`GetInputTensorSizes()`
- **Signature**: `std::vector<uint64_t> GetInputTensorSizes()`
- **Description**: Gets the individual sizes (in bytes) of each input tensor for multi-input models.
- **Returns**: A vector of input tensor sizes, in the order specified by `GetInputTensorNames()`.

`GetInputTensorToTaskMapping()`
- **Signature**: `std::map<std::string, std::string> GetInputTensorToTaskMapping() const`
- **Description**: Returns the mapping from input tensor names to their target tasks within the model graph.
- **Returns**: A map where the key is the tensor name and the value is the task name.

`GetLatency()`
- **Signature**: `int GetLatency()`
- **Description**: Gets the latency of the most recent inference in microseconds.
- **Returns**: The latency value.
- **Note**: The legacy function `latency()` is deprecated.

`GetLatencyCnt()`
- **Signature**: `int GetLatencyCnt()`
- **Description**: Gets the total count of latency measurements recorded.
- **Returns**: The number of latency measurements.

`GetLatencyMean()`
- **Signature**: `double GetLatencyMean()`
- **Description**: Gets the mean (average) of all collected latency values.
- **Returns**: The mean latency in microseconds.

© Copyright 2026 DEEPX All Rights Reserved.

`GetLatencyStdDev()`
- **Signature**: `double GetLatencyStdDev()`
- **Description**: Gets the standard deviation of all collected latency values.
- **Returns**: The standard deviation of latency.

`GetLatencyVector()`
- **Signature**: `std::vector<int> GetLatencyVector()`
- **Description**: Gets a vector of recent latency measurements.
- **Returns**: A vector of latencies in microseconds.

`GetModelName()`
- **Signature**: `std::string GetModelName()`
- **Description**: Gets the name of the model.
- **Returns**: The model name as a `std::string`.
- **Note**: The legacy function `name()` is deprecated.

`GetModelVersion()`
- **Signature**: `std::string GetModelVersion()`
- **Description**: Returns the DXNN file format version of the loaded model.
- **Returns**: The model version string.

`GetNpuInferenceTime()`
- **Signature**: `uint32_t GetNpuInferenceTime()`
- **Description**: Gets the pure NPU processing time for the most recent inference in microseconds.
- **Returns**: The NPU inference time.
- **Note**: The legacy function `inference_time()` is deprecated.

`GetNpuInferenceTimeCnt()`
- **Signature**: `int GetNpuInferenceTimeCnt()`
- **Description**: Gets the total count of NPU inference time measurements recorded.
- **Returns**: The number of measurements.

`GetNpuInferenceTimeMean()`
- **Signature**: `double GetNpuInferenceTimeMean()`
- **Description**: Gets the mean (average) of all collected NPU inference times.
- **Returns**: The mean NPU inference time in microseconds.

`GetNpuInferenceTimeStdDev()`
- **Signature**: `double GetNpuInferenceTimeStdDev()`
- **Description**: Gets the standard deviation of all collected NPU inference times.
- **Returns**: The standard deviation of NPU inference time.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

GetNpuInferenceTimeVector()
- **Signature**: `std::vector<uint32_t> GetNpuInferenceTimeVector()`
- **Description**: Gets a vector of recent NPU inference time measurements.
- **Returns**: A vector of NPU inference times in microseconds.

GetNumTailTasks()
- **Signature**: `int GetNumTailTasks()`
- **Description**: Returns the number of "tail" tasks in the model, which are tasks that have no subsequent tasks.
- **Returns**: The number of tail tasks.
- **Note**: The legacy function `get_num_tails()` is deprecated.

GetOutputs(void *ptr = nullptr, uint64_t phyAddr = 0)
- **Signature**: `Tensors GetOutputs(void *ptr = nullptr, uint64_t phyAddr = 0)`
- **Description**: Retrieves the output tensors. If `ptr` is null, it returns information about the output memory area within the engine. If `ptr` and `phyAddr` are provided, it returns tensor objects pointing to those addresses.
- **Parameters**:
- `ptr`: An optional pointer to a virtual address for the output data.
- `phyAddr`: An optional pointer to a physical address for the output data.
- **Returns**: A `Tensors` (vector of `Tensor`) object.
- **Note**: The legacy function `outputs(...)` is deprecated.

GetOutputSize()
- **Signature**: `uint64_t GetOutputSize()`
- **Description**: Gets the total size of all output tensors combined in bytes.
- **Returns**: The total output size as a `uint64_t`.
- **Note**: The legacy function `output_size()` is deprecated.

GetOutputTensorNames()
- **Signature**: `std::vector<std::string> GetOutputTensorNames() const`
- **Description**: Returns the names of all output tensors in the order they are produced.
- **Returns**: A vector of output tensor names.

GetOutputTensorOffset(const std::string& tensorName) const
- **Signature**: `size_t GetOutputTensorOffset(const std::string& tensorName) const`
- **Description**: Gets the byte offset for a specific output tensor within the final concatenated output buffer.
- **Parameters**:
- `tensorName`: The name of the output tensor.
- **Returns**: The offset in bytes.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

`GetOutputTensorSizes()`
- **Signature**: `std::vector<uint64_t> GetOutputTensorSizes()`
- **Description**: Gets the individual sizes (in bytes) of each output tensor.
- **Returns**: A vector of output tensor sizes, in the order specified by `GetOutputTensorNames()`.

`GetTaskOrder()`
- **Signature**: `std::vector<std::string> GetTaskOrder()`
- **Description**: Gets the model's task execution order.
- **Returns**: A vector of strings representing the task order.
- **Note**: The legacy function `task_order()` is deprecated.

`IsMultiInputModel()`
- **Signature**: `bool IsMultiInputModel() const`
- **Description**: Checks if the loaded model requires multiple input tensors.
- **Returns**: `true` if the model has multiple inputs, `false` otherwise.

`IsOrtConfigured()`
- **Signature**: `bool IsOrtConfigured()`
- **Description**: Checks whether ONNX Runtime (ORT) is configured and available for use.
- **Returns**: `true` if ORT is configured, `false` otherwise.

`IsPPU()`
- **Signature**: `bool IsPPU()`
- **Description**: Checks if the loaded model utilizes a Post-Processing Unit (PPU).
- **Returns**: `true` if the model uses a PPU, `false` otherwise.
- **Note**: The legacy function `is_PPU()` is deprecated.

`RegisterCallback(std::function<int(TensorPtrs& outputs, void* userArg)> callbackFunc)`
- **Signature**: `void RegisterCallback(std::function<int(TensorPtrs& outputs, void* userArg)> callbackFunc)`
- **Description**: Registers a user-defined callback function that will be executed upon completion of an asynchronous inference request.
- **Parameters**:
- `callbackFunc`: The function to be called. It receives the output tensors and the user-provided argument.
- **Note**: The legacy function `RegisterCallBack(...)` is deprecated.

`Run(void *inputPtr, void *userArg = nullptr, void *outputPtr = nullptr)`
- **Signature**: `TensorPtrs Run(void *inputPtr, void *userArg = nullptr, void *outputPtr = nullptr)`
- **Description**: Performs a synchronous inference for a single input, blocking until the operation is complete.

© Copyright 2026 DEEPX All Rights Reserved.

- **Parameters**:
- `inputPtr`: A pointer to the input data.
- `userArg`: An optional user-defined argument.
- `outputPtr`: An optional pointer to a pre-allocated output buffer.
- **Returns**: A `TensorPtrs` object containing the output data.

`Run(const std::vector<void*>& inputBuffers, const std::vector<void*>& outputBuffers, const std::vector<void*>& userArgs = {})`
- **Signature**: `std::vector<TensorPtrs> Run(const std::vector<void*>& inputBuffers, const std::vector<void*>& outputBuffers, const std::vector<void*>& userArgs = {})`
- **Description**: Performs a synchronous batch inference.
- **Parameters**:
- `inputBuffers`: A vector of pointers to input data for each sample in the batch.
- `outputBuffers`: A vector of pointers to pre-allocated output buffers.
- `userArgs`: An optional vector of user-defined arguments.
- **Returns**: A vector of `TensorPtrs`, where each element corresponds to the output of one sample.

`RunAsync(void *inputPtr, void *userArg=nullptr, void *outputPtr = nullptr)`
- **Signature**:
`int RunAsync(void *inputPtr, void *userArg=nullptr, void *outputPtr = nullptr)`
- **Description**: Submits a non-blocking, asynchronous inference request for a single input.
- **Parameters**:
- `inputPtr`: A pointer to the input data.
- `userArg`: An optional user-defined argument to be passed to the callback.
- `outputPtr`: An optional pointer to a pre-allocated output buffer.
- **Returns**: An integer `jobId` for this asynchronous operation.

`RunAsync(const std::vector<void*>& inputPtrs, void *userArg=nullptr, void *outputPtr = nullptr)`
- **Signature**:
`int RunAsync(const std::vector<void*>& inputPtrs, void *userArg=nullptr, void *outputPtr = nullptr)`
- **Description**: Submits an asynchronous inference request, automatically detecting if the input is for a multi-input model.
- **Parameters**:
- `inputPtrs`: A vector of pointers to input data.
- `userArg`: An optional user-defined argument.
- `outputPtr`: An optional pointer to a pre-allocated output buffer.
- **Returns**: An integer `jobId`.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

`RunAsyncMultiInput(const std::map<std::string, void*>& inputTensors, void *userArg=nullptr, void *outputPtr = nullptr)`

- **Signature**: `int RunAsyncMultiInput(const std::map<std::string, void*>& inputTensors, void *userArg=nullptr, void *outputPtr = nullptr)`
- **Description**: Submits an asynchronous inference request for a multi-input model using a map of named tensors.
- **Parameters**:
- `inputTensors`: A map of tensor names to input data pointers.
- `userArg`: An optional user-defined argument.
- `outputPtr`: An optional pointer to a pre-allocated output buffer.
- **Returns**: An integer `jobId`.

`RunAsyncMultiInput(const std::vector<void*>& inputPtrs, void *userArg=nullptr, void *outputPtr = nullptr)`

- **Signature**: `int RunAsyncMultiInput(const std::vector<void*>& inputPtrs, void *userArg=nullptr, void *outputPtr = nullptr)`
- **Description**: Submits an asynchronous inference request for a multi-input model using a vector of input pointers.
- **Parameters**:
- `inputPtrs`: A vector of input pointers in the order specified by `GetInputTensorNames()`.
- `userArg`: An optional user-defined argument.
- `outputPtr`: An optional pointer to a pre-allocated output buffer.
- **Returns**: An integer `jobId`.

`RunBenchmark(int num, void* inputPtr = nullptr)`

- **Signature**: `float RunBenchmark(int num, void* inputPtr = nullptr)`
- **Description**: Runs a performance benchmark for a specified number of loops.
- **Parameters**:
- `num`: The number of inference iterations to run.
- `inputPtr`: An optional pointer to the input data to use for the benchmark.
- **Returns**: The average frames per second (FPS) as a float.
- **Note**: The legacy function `RunBenchMark(...)` is deprecated.

`RunMultiInput(const std::map<std::string, void*>& inputTensors, void *userArg=nullptr, void *outputPtr=nullptr)`

- **Signature**: `TensorPtrs RunMultiInput(const std::map<std::string, void*>& inputTensors, void *userArg=nullptr, void *outputPtr=nullptr)`
- **Description**: Runs synchronous inference for a multi-input model using a map of named tensors.
- **Parameters**:

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- `inputTensors` : A map of tensor names to input data pointers.
- `userArg` : An optional user-defined argument.
- `outputPtr` : An optional pointer to a pre-allocated output buffer.
- **Returns**: A `TensorPtrs` object containing the output.

`RunMultiInput(const std::vector<void*>& inputPtrs, void *userArg=nullptr, void *outputPtr=nullptr)`
- **Signature**: `TensorPtrs RunMultiInput(const std::vector<void*>& inputPtrs, void *userArg=nullptr, void *outputPtr=nullptr)`
- **Description**: Runs synchronous inference for a multi-input model using a vector of input pointers.
- **Parameters**:
- `inputPtrs` : A vector of input pointers in the order specified by `GetInputTensorNames()` .
- `userArg` : An optional user-defined argument.
- `outputPtr` : An optional pointer to a pre-allocated output buffer.
- **Returns**: A `TensorPtrs` object containing the output.

`Wait(int jobId)`
- **Signature**: `TensorPtrs Wait(int jobId)`
- **Description**: Blocks execution and waits until the asynchronous request identified by `jobId` is complete.
- **Parameters**:
- `jobId` : The job ID returned from a `RunAsync` call.
- **Returns**: A `TensorPtrs` object containing the output from the completed job.

---

class dxrt::InferenceOption

This class specifies inference options applied to an `InferenceEngine` , allowing users to configure which devices and NPU cores are used.

NESTED ENUMS

`enum BOUND_OPTION`
- **Description**: Defines how NPU cores are bound or utilized for inference.
- **Members**: `NPU_ALL` , `NPU_0` , `NPU_1` , `NPU_2` , `NPU_01` , `NPU_12` , `NPU_02` .

PUBLIC MEMBERS

- **`boundOption`** : `uint32_t` . Selects the NPU core(s) to use within a device, using a value from the `BOUND_OPTION` enum. Default is `NPU_ALL` .

© Copyright 2026 DEEPX All Rights Reserved.

- **`devices`**: `std::vector<int>`. A list of device IDs to be used for inference. If the list is empty (default), all available devices are used.
- **`useORT`**: `bool`. If `true`, both NPU and CPU (via ONNX Runtime) tasks will be executed. If `false`, only NPU tasks will run.
- **`bufferCount`**: `int`. Specifies the number of internal buffers allocated for inference. Higher values can improve throughput in pipelined inference scenarios by allowing more concurrent operations, but consume more memory. Default is `DXRT_TASK_MAX_LOAD_VALUE` (typically 6). Valid range is 1-100.

```
dxrt::InferenceOption option;
option.bufferCount = 8;  // Allocate 8 buffers for higher throughput
dxrt::InferenceEngine ie("model.dxnn", option);
```

---

class dxrt::Configuration

A singleton class for managing global application configurations. Access is thread-safe and should be done via the `GetInstance()` method.

NESTED ENUMS

`enum class ITEM`
- **Description**: Defines configuration categories.
- **Members**: `DEBUG`, `PROFILER`, `SERVICE`, `DYNAMIC_CPU_THREAD`, `TASK_FLOW`, `SHOW_THROTTLING`, `SHOW_PROFILE`, `SHOW_MODEL_INFO`, `CUSTOM_INTRA_OP_THREADS`, `CUSTOM_INTER_OP_THREADS`.

`enum class ATTRIBUTE`
- **Description**: Defines attributes for configuration items.
- **Members**: `PROFILER_SHOW_DATA`, `PROFILER_SAVE_DATA`, `CUSTOM_INTRA_OP_THREADS_NUM`, `CUSTOM_INTER_OP_THREADS_NUM`.

STATIC MEMBER FUNCTIONS

`GetInstance()`
- **Signature**: `static Configuration& GetInstance()`
- **Description**: Returns the unique static instance of the `Configuration` class. This is the only way to access the configuration object.
- **Returns**: A reference to the `Configuration` instance.

© Copyright 2026 DEEPX All Rights Reserved.

MEMBER FUNCTIONS

`GetAttribute(const ITEM item, const ATTRIBUTE attrib)`
- **Signature**: `std::string GetAttribute(const ITEM item, const ATTRIBUTE attrib)`
- **Description**: Retrieves the value of a specific attribute for a given configuration item.
- **Parameters**:
- `item` : The configuration item.
- `attrib` : The attribute to retrieve.
- **Returns**: The attribute value as a `std::string` .

`GetIntAttribute(const ITEM item, const ATTRIBUTE attrib)`
- **Signature**: `int GetIntAttribute(const ITEM item, const ATTRIBUTE attrib)`
- **Description**: Retrieves the value of a specific attribute as an integer for a given configuration item. Automatically converts string attributes to integers.
- **Parameters**:
- `item` : The configuration item.
- `attrib` : The attribute to retrieve.
- **Returns**: The attribute value as an `int` . Returns 0 if the attribute is not found or cannot be parsed as an integer.

`GetDriverVersion()`
- **Signature**: `std::string GetDriverVersion() const`
- **Description**: Retrieves the version of the associated device driver.
- **Returns**: The driver version string.

`GetEnable(const ITEM item)`
- **Signature**: `bool GetEnable(const ITEM item)`
- **Description**: Retrieves the enabled status of a specific configuration item.
- **Parameters**:
- `item` : The configuration item to check.
- **Returns**: `true` if the item is enabled, `false` otherwise.

`GetFirmwareVersions()`
- **Signature**: `std::vector<std::pair<int, std::string>> GetFirmwareVersions() const`
- **Description**: Retrieves the firmware versions of all detected devices.
- **Returns**: A vector of pairs, where each pair contains a device ID and its firmware version string.

`GetONNXRuntimeVersion()`
- **Signature**: `std::string GetONNXRuntimeVersion() const`
- **Description**: Retrieves the version of the ONNX Runtime library being used.
- **Returns**: The ONNX Runtime version string.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

`GetPCIeDriverVersion()`
- **Signature**: `std::string GetPCIeDriverVersion() const`
- **Description**: Retrieves the version of the PCIe driver.
- **Returns**: The PCIe driver version string.

`GetVersion()`
- **Signature**: `std::string GetVersion() const`
- **Description**: Retrieves the version of the DXRT library.
- **Returns**: The library version string.

`LoadConfigFile(const std::string& fileName)`
- **Signature**: `void LoadConfigFile(const std::string& fileName)`
- **Description**: Loads configuration settings from the specified file.
- **Parameters**:
- `fileName`: The path and name of the configuration file.

`LockEnable(const ITEM item)`
- **Signature**: `void LockEnable(const ITEM item)`
- **Description**: Locks a specific configuration item, making it read-only.
- **Parameters**:
- `item`: The configuration item to lock.

`SetAttribute(const ITEM item, const ATTRIBUTE attrib, const std::string& value)`
- **Signature**:
`void SetAttribute(const ITEM item, const ATTRIBUTE attrib, const std::string& value)`
- **Description**: Sets a specific attribute value for a given configuration item (e.g., setting `PROFILER_SAVE_DATA` to `"ON"`).
- **Parameters**:
- `item`: The configuration item.
- `attrib`: The attribute to set.
- `value`: The attribute value as a string.

`SetEnable(const ITEM item, bool enabled)`
- **Signature**: `void SetEnable(const ITEM item, bool enabled)`
- **Description**: Sets the enabled status for a specific configuration item (e.g., enables the profiler).
- **Parameters**:
- `item`: The configuration item.
- `enabled`: `true` to enable, `false` to disable.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

class dxrt::DeviceStatus

Provides an abstraction for retrieving device information and real-time status.

STATIC MEMBER FUNCTIONS

`GetCurrentStatus(int id)`
- **Signature**: `static DeviceStatus GetCurrentStatus(int id)`
- **Description**: Retrieves the status for the device with the specified ID. The status includes key metrics such as core utilization and temperature. Monitoring data is refreshed every 1 second.
- **Parameters**:
- `id` : The unique identifier of the device.
- **Returns**: A `DeviceStatus` object containing the device's current status. If the specified device ID is invalid, the device does not exist, or shared memory cannot be opened, the returned object will have `IsValid() == false`.
- **Example**:

```
DeviceStatus status = DeviceStatus::GetCurrentStatus(0);
if (status.IsValid()) {
    std::cout << "Device 0 Status: " << status.GetStatusString() << std::endl;
} else {
    LOG_DXRT_ERR("Device 0 status is not available.");
}
```

`GetDeviceCount()`
- **Signature**: `static int GetDeviceCount()`
- **Description**: Retrieves the total number of hardware devices currently recognized by the system. This includes devices that are initialized and ready for operation.
- **Returns**: The total number of available devices.
- **Example**:

```
int deviceCount = dxrt::DeviceStatus::GetDeviceCount();
std::cout << "Number of available devices: " << deviceCount << std::endl;
```

MEMBER FUNCTIONS

`AllMemoryInfoStr()`
- **Signature**: `std::string AllMemoryInfoStr() const`
- **Description**: Retrieves a summary of the device's memory specifications (type, frequency, size) in a single line. Example: `"Memory: LPDDR4 4200 MHz, 1.98 GiB"`.
- **Returns**: A formatted string.

© Copyright 2026 DEEPX All Rights Reserved.

`BoardTypeStr()`
- **Signature**: `std::string BoardTypeStr() const`
- **Description**: Returns the device board type.
- **Returns**: A string such as "SOM" or "M.2".

`DdrBitErrStr()`
- **Signature**: `std::string DdrBitErrStr() const`
- **Description**: Retrieves the count of LPDDR Double-bit & Single-bit Errors.
- **Returns**: A formatted string.

`DdrStatusStr(int ch)`
- **Signature**: `std::string DdrStatusStr(int ch) const`
- **Description**: Retrieves the status of a specified LPDDR memory channel, including configuration and real-time temperature information based on MR4 register values. Example: `"LPDDR Channel 0: MR4=0x1A, Temperature Normal"`.
- **Parameters**:
- `ch`: The LPDDR memory channel index (0 to 3).
- **Returns**: A formatted string containing the channel status.

`DeviceTypeStr()`
- **Signature**: `std::string DeviceTypeStr() const`
- **Description**: Retrieves the device type as a three-letter abbreviation.
- **Returns**: A string ("ACC" for Accelerator or "STD" for Standalone).

`DeviceTypeWord()`
- **Signature**: `std::string DeviceTypeWord() const`
- **Description**: Retrieves the full name of the device type.
- **Returns**: A string ("Accelerator" or "Standalone").

`DeviceVariantStr()`
- **Signature**: `std::string DeviceVariantStr() const`
- **Description**: Returns the device chip variant type.
- **Returns**: A string such as "L1" or "M1".

`DmaChannel()`
- **Signature**: `uint64_t DmaChannel() const`
- **Description**: Retrieves the number of DMA (Direct Memory Access) channels available for the NPU. A higher number of DMA channels allows better parallel data movement, improving performance.
- **Returns**: The number of DMA channels.

`FirmwareVersionStr()`
- **Signature**: `std::string FirmwareVersionStr() const`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Description**: Retrieves the firmware version of the NPU, following the Major.Minor.Patch versioning format.
- **Returns**: The version string (e.g., "1.2.3").

`GetDeviceType()`
- **Signature**: `DeviceType GetDeviceType() const`
- **Description**: Retrieves the device type as a `DeviceType` enum.
- **Returns**: A `DeviceType` enum value.

`GetId()`
- **Signature**: `int GetId() const`
- **Description**: Retrieves the unique identifier of the device.
- **Returns**: The device ID as an integer.

`GetInfoString()`
- **Signature**: `std::string GetInfoString() const`
- **Description**: Retrieves detailed static information about the device, equivalent to `dxcli -i`.
- **Returns**: A formatted string with device specifications.

`GetNpuClock(int ch)`
- **Signature**: `uint32_t GetNpuClock(int ch) const`
- **Description**: Retrieves the current clock frequency of the specified NPU channel. The clock frequency may change dynamically depending on performance scaling settings.
- **Parameters**:
- `ch`: The NPU channel index.
- **Returns**: The clock frequency in megahertz (MHz).

`GetNpuVoltage(int ch)`
- **Signature**: `uint32_t GetNpuVoltage(int ch) const`
- **Description**: Retrieves the voltage level of the specified NPU channel. The voltage level can vary depending on power management settings and workload.
- **Parameters**:
- `ch`: The NPU channel index.
- **Returns**: The voltage level in millivolts (mV).

`GetStatusString()`
- **Signature**: `std::string GetStatusString() const`
- **Description**: Retrieves the real-time status of the device, equivalent to `dxcli -s`.
- **Returns**: A formatted string with real-time status.

`GetTemperature(int ch)`
- **Signature**: `int GetTemperature(int ch) const`

© Copyright 2026 DEEPX All Rights Reserved.

- **Description**: Retrieves the temperature of the specified NPU channel. Monitoring the temperature is crucial for ensuring the NPU operates within safe thermal limits.
- **Parameters**:
- `ch`: The NPU channel index.
- **Returns**: The temperature in degrees Celsius. Returns `INT16_MIN` when the channel index is out of range.

`MemoryClock()`
- **Signature**: `uint64_t MemoryClock() const`
- **Description**: Retrieves the memory clock frequency of the NPU. The memory clock speed affects data transfer rates and overall processing efficiency.
- **Returns**: The frequency in megahertz (MHz).

`MemoryFrequency()`
- **Signature**: `int MemoryFrequency() const`
- **Description**: Retrieves the memory operating frequency of the device. Higher frequencies typically result in better memory performance.
- **Returns**: The frequency in megahertz (MHz).

`MemorySize()`
- **Signature**: `int64_t MemorySize() const`
- **Description**: Retrieves the total memory size available for the NPU. The memory size determines the capacity for storing models, intermediate computations, and input data.
- **Returns**: The total memory size in bytes.

`MemorySizeStrBinaryPrefix()`
- **Signature**: `std::string MemorySizeStrBinaryPrefix() const`
- **Description**: Retrieves the total memory size as a string using binary units (IEC standard). Example: `"1.98 GiB"`.
- **Returns**: A formatted string.

`MemorySizeStrWithComma()`
- **Signature**: `std::string MemorySizeStrWithComma() const`
- **Description**: Retrieves the total memory size as a string in bytes, formatted with thousands separators for readability. Example: `"2,130,706,432 Byte"`.
- **Returns**: A formatted string.

`MemoryTypeStr()`
- **Signature**: `std::string MemoryTypeStr() const`
- **Description**: Retrieves the type of memory used in the device.
- **Returns**: A string (e.g., "LPDDR4").

© Copyright 2026 DEEPX All Rights Reserved.

`NpuStatusStr(int ch)`
- **Signature**: `std::string NpuStatusStr(int ch) const`
- **Description**: Retrieves the status of a specific NPU as a formatted string (voltage, clock, temperature). Example: `"NPU 0: voltage 825 mV, clock 800 MHz, temperature 46'C"`.
- **Parameters**:
- `ch`: The NPU index.
- **Returns**: A formatted string.

`PcieInfoStr(int spd, int wd, int bus, int dev, int func)`
- **Signature**: `std::string PcieInfoStr(int spd, int wd, int bus, int dev, int func) const`
- **Description**: Returns PCIe information (speed, generation, etc.) as a string.
- **Parameters**:
- `spd`, `wd`, `bus`, `dev`, `func`: PCIe configuration parameters.
- **Returns**: A formatted string with PCIe information.

`GetCoreUtilization(int coreId)` - **Signature**: `double GetCoreUtilization(int coreId) const` - **Description**: Retrieves the utilization of the specified NPU core. - **Parameters**: - `coreId`: The NPU core index (0 to 2). - **Returns**: Utilization percentage (0.0 ~ 100.0). Returns -1.0 if coreId is out of range. Returns 0.0 if monitoring data is unavailable.

`GetMemoryUsed()` - **Signature**: `uint64_t GetMemoryUsed() const` - **Description**: Retrieves the amount of NPU DRAM currently in use. - **Returns**: DRAM usage in bytes. Returns 0 if monitoring data is unavailable.

`GetMemoryFree()` - **Signature**: `uint64_t GetMemoryFree() const` - **Description**: Retrieves the amount of free NPU DRAM. - **Returns**: Free DRAM in bytes. Returns 0 if monitoring data is unavailable.

---

class dxrt::Tensor

This class abstracts a DXRT tensor object, which defines a data array composed of uniform elements.

CONSTRUCTOR

`Tensor(std::string name_, std::vector<int64_t> shape_, DataType type_, void *data_=nullptr)`
- **Description**: Constructs a Tensor object.
- **Parameters**:
- `name_`: The name of the tensor.
- `shape_`: A vector defining the tensor's shape (dimensions).

© Copyright 2026 DEEPX All Rights Reserved.

- `type_` : The tensor's data type.
- `data_` : An optional pointer to the tensor's data.

MEMBER FUNCTIONS

`data()`
- **Signature**: `void* &data()`
- **Description**: Accessor for the tensor's data pointer.
- **Returns**: A reference to the void pointer holding the tensor's data.

`data(int height, int width, int channel)`
- **Signature**: `void* data(int height, int width, int channel)`
- **Description**: Gets a pointer to a specific element by its index, assuming NHWC data layout.
- **Parameters**:
- `height` : The height index.
- `width` : The width index.
- `channel` : The channel index.
- **Returns**: A void pointer to the specified element.

`elem_size()`
- **Signature**: `uint32_t &elem_size()`
- **Description**: Accessor for the size of a single element in the tensor.
- **Returns**: A reference to the element size in bytes.

`name()`
- **Signature**: `const std::string &name() const`
- **Description**: Accessor for the tensor's name.
- **Returns**: A constant reference to the tensor's name string.

`phy_addr()`
- **Signature**: `uint64_t &phy_addr()`
- **Description**: Accessor for the physical address of the tensor's data.
- **Returns**: A reference to the physical address.

`shape()`
- **Signature**: `std::vector<int64_t> &shape()`
- **Description**: Accessor for the tensor's shape.
- **Returns**: A reference to the vector defining the tensor's dimensions.

`size_in_bytes()`
- **Signature**: `uint64_t size_in_bytes() const`
- **Description**: Calculates and returns the total size of the tensor's data in bytes based on its shape and

© Copyright 2026 DEEPX All Rights Reserved.

element size.
- **Returns**: The total size in bytes.

`type()`
- **Signature**: `DataType &type()`
- **Description**: Accessor for the tensor's data type.
- **Returns**: A reference to the `DataType` enum.

---

class dxrt::RuntimeEventDispatcher

A singleton class that provides a centralized event dispatching mechanism for runtime events such as device errors, warnings, and notifications. It supports custom event handlers and automatic logging of events with different severity levels. Access is thread-safe and should be done via the `GetInstance()` method.

NESTED ENUMS

`enum class LEVEL`
- **Description**: Event severity levels for categorizing runtime events.
- **Members**:
- `INFO` (1): Informational messages for normal operation events.
- `WARNING` (2): Warning messages for potential issues that don't stop execution.
- `ERROR` (3): Error messages for recoverable failures.
- `CRITICAL` (4): Critical errors that may cause system instability.

`enum class TYPE`
- **Description**: Event type categories for classifying the source of events.
- **Members**:
- `DEVICE_CORE` (1000): Events related to NPU core operations.
- `DEVICE_STATUS` (1001): Device status change events.
- `DEVICE_IO` (1002): Input/Output operation events.
- `DEVICE_MEMORY` (1003): Memory management events.
- `UNKNOWN` (1004): Unknown or unclassified event types.

`enum class CODE`
- **Description**: Specific event codes for identifying the exact nature of events.
- **Members**:
- `WRITE_INPUT` (2000): Input data write operation event.
- `READ_OUTPUT` (2001): Output data read operation event.
- `MEMORY_OVERFLOW` (2002): Memory overflow or capacity exceeded.

© Copyright 2026 DEEPX All Rights Reserved.

- `MEMORY_ALLOCATION` (2003): Memory allocation failure or issue.
- `DEVICE_EVENT` (2004): General device event notification.
- `RECOVERY_OCCURRED` (2005): Device recovery action taken.
- `TIMEOUT_OCCURRED` (2006): Operation timeout event.
- `THROTTLING_NOTICE` (2007): Device throttling notification.
- `THROTTLING_EMERGENCY` (2008): Device throttling emergency notification.
- `UNKNOWN` (2009): Unknown or unclassified event code.

STATIC MEMBER FUNCTIONS

`GetInstance()`
- **Signature**: `static RuntimeEventDispatcher& GetInstance()`
- **Description**: Returns the unique static instance of the `RuntimeEventDispatcher` class. This is the only way to access the dispatcher object.
- **Returns**: A reference to the `RuntimeEventDispatcher` instance.
- **Example**:

```
auto& dispatcher = dxrt::RuntimeEventDispatcher::GetInstance();
```

MEMBER FUNCTIONS

`DispatchEvent(LEVEL level, TYPE type, CODE code, const std::string& eventMessage)`
- **Signature**: `void DispatchEvent(LEVEL level, TYPE type, CODE code, const std::string& eventMessage)`
- **Description**: Dispatches a runtime event with the specified parameters. The event is logged and any registered custom event handler is invoked. Events are filtered based on the current level threshold set via `SetCurrentLevel`.
- **Parameters**:
- `level`: Severity level of the event (`INFO`, `WARNING`, `ERROR`, `CRITICAL`).
- `type`: Category of the event (`DEVICE_CORE`, `DEVICE_IO`, etc.).
- `code`: Specific event code identifying the exact event.
- `eventMessage`: Descriptive message providing event details.

`RegisterEventHandler(const std::function<void(LEVEL, TYPE, CODE, const std::string& message, const std::string& timestamp)>& handler)`
- **Signature**: `void RegisterEventHandler(const std::function<void(LEVEL, TYPE, CODE, const std::string& message, const std::string& timestamp)>& handler)`
- **Description**: Registers a user-defined callback function that will be invoked for each dispatched event. Only one handler can be registered at a time; subsequent calls will replace the previous handler. The handler is invoked synchronously but with minimal lock holding time to avoid blocking.

© Copyright 2026 DEEPX All Rights Reserved.

- **Parameters**:
- `handler`: Callback function with the signature `void(LEVEL, TYPE, CODE, const std::string& message, const std::string& timestamp)`.
- **Example**:

```
auto& dispatcher = dxrt::RuntimeEventDispatcher::GetInstance();
dispatcher.RegisterEventHandler(
    [](dxrt::RuntimeEventDispatcher::LEVEL level,
       dxrt::RuntimeEventDispatcher::TYPE type,
       dxrt::RuntimeEventDispatcher::CODE code,
       const std::string& message,
       const std::string& timestamp)
    {
        std::cout << "[" << timestamp << "] " << message << std::endl;
    }
);
```

`SetCurrentLevel(LEVEL level)`

- **Signature**: `void SetCurrentLevel(LEVEL level)`
- **Description**: Sets the minimum event level threshold. Events below this level may be filtered out by custom handlers.
- **Parameters**:
- `level`: Minimum severity level for events to be processed.

`GetCurrentLevel()`

- **Signature**: `LEVEL GetCurrentLevel() const`
- **Description**: Gets the current minimum event level threshold.
- **Returns**: The current minimum event severity level as a `LEVEL` enum value.

© Copyright 2026 DEEPX All Rights Reserved.

10.2 Python API Reference

10.2.1 Deprecated API Migration Guide

The following `InferenceEngine` methods have been deprecated and will be removed in a future release. Please update your code to use the recommended alternatives.

Naming Convention Changes (PascalCase → snake_case)

Deprecated Method	Replacement	Notes
`Run(input_feed_list, user_arg)`	`run(input_data, user_args=user_arg)`	Parameter renamed to `user_args` (plural). Note: `run_async`, `run_multi_input`, `run_async_multi_input` retain the singular `user_arg` form
`RunBatch(input_buffers, output_buffers, user_args)`	`run(input_buffers, output_buffers=output_buffers, user_args=user_args)`	Use `run()` with batch-formatted inputs
`RunAsync(input_feed_list, user_arg)`	`run_async(input_data, user_arg=user_arg)`	Parameter name kept as singular `user_arg` (only `run()` uses plural `user_args`)
`RunBenchMark(loop_cnt, input_feed_list)`	`run_benchmark(num_loops, input_data)`	Parameter names changed
`ValidateDevice(input_feed_list, device_id)`	`validate_device(input_data, device_id)`	
`RegisterCallBack(callback)`	`register_callback(callback)`	
`Wait(req_id)`	`wait(job_id)`	Parameter name changed: `req_id` → `job_id`
`is_PPU()`	`is_ppu()`	

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Method Renaming (Consistency Improvements)

Deprecated Method	Replacement	Notes
`input_size()`	`get_input_size()`	
`output_size()`	`get_output_size()`	
`task_order()`	`get_task_order()`	
`latency()`	`get_latency()`	
`inference_time()`	`get_npu_inference_time()`	
`get_outputs()`	`get_all_task_outputs()`	
`bitmatch_mask(index)`	`get_bitmatch_mask(index)`	
`get_num_tails()`	`get_num_tail_tasks()`	
`run_batch(input_buffers, output_buffers, user_args)`	`run(input_buffers, output_buffers=output_buffers, user_args=user_args)`	Consolidated into `run()`

Data Type Query Methods

Deprecated Method	Replacement	Notes
`input_dtype()`	`get_input_tensors_info()`	Access `'dtype'` key from returned dict list
`output_dtype()`	`get_output_tensors_info()`	Access `'dtype'` key from returned dict list
`get_input_data_type()`	`get_input_tensors_info()`	Access `'dtype'` key from returned dict list
`get_output_data_type()`	`get_output_tensors_info()`	Access `'dtype'` key from returned dict list

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Migration Examples

Before (deprecated):

```
engine = InferenceEngine("model.dxnn")

# Inference
outputs = engine.Run([input_buffer])

# Batch inference
batch_outputs = engine.RunBatch(input_buffers, output_buffers)

# Async inference
job_id = engine.RunAsync([input_buffer])
result = engine.Wait(job_id)

# Query info
size = engine.input_size()
dtypes = engine.input_dtype()
time = engine.inference_time()
```

After (recommended):

```
engine = InferenceEngine("model.dxnn")

# Inference
outputs = engine.run(input_buffer)

# Batch inference
batch_outputs = engine.run(input_buffers, output_buffers=output_buffers)

# Async inference
job_id = engine.run_async(input_buffer)
result = engine.wait(job_id)

# Query info
size = engine.get_input_size()
dtypes = [info['dtype'] for info in engine.get_input_tensors_info()]
time = engine.get_npu_inference_time()
```

This section describes the Python bindings of the DX-RT SDK. It provides a streamlined interface to the same core functionalities as the C++ SDK, making it ideal for rapid development, prototyping, and integration with Python-based AI workflows.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

class dx_engine.InferenceEngine

This class is the main Python wrapper for the DXRT Inference Engine. It provides an interface to load a compiled model and perform inference tasks, either synchronously or asynchronously, supporting both single and batch inference.

CONSTRUCTOR

`__init__(self, model_path: str, inference_option: Optional[InferenceOption] = None)`
- **Description**: Initializes the InferenceEngine by loading a compiled model from the specified file path.
- **Parameters**:
- `model_path` : `str` . Path to the compiled model file (e.g., `*.dxnn`).
- `inference_option` : `Optional[InferenceOption]` . An `InferenceOption` object for configuration. If `None` , default options are used.
- **Raises**: `RuntimeError` if the underlying C++ engine fails to initialize.
- **Example**:

```
from dx_engine import InferenceEngine, InferenceOption

# Load model from file path
engine = InferenceEngine("model.dxnn")

# Load model with custom options
option = InferenceOption()
option.devices = [0, 1]
option.bound_option = InferenceOption.BOUND_OPTION.NPU_ALL
engine2 = InferenceEngine("model.dxnn", option)
```

`from_buffer(cls, memory_buffer: np.ndarray, inference_option: Optional[InferenceOption] = None) -> InferenceEngine`
- **Description**: Alternative class method constructor that creates an InferenceEngine from a memory buffer without requiring a file path. This is useful for embedded systems, network-based model distribution, or when models are loaded from custom sources (e.g., encrypted storage).
- **Parameters**:
- `memory_buffer` : `np.ndarray` . Pre-allocated memory buffer containing the compiled model data. Must be a C-contiguous numpy array (dtype=uint8).
- `inference_option` : `Optional[InferenceOption]` . Configuration options for inference. If `None` , default options are used.
- **Returns**: A new `InferenceEngine` instance loaded from the memory buffer.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Raises**: - `TypeError` if `memory_buffer` is not a numpy array. - `ValueError` if `memory_buffer` is not C-contiguous. - `RuntimeError` if the C++ engine fails to initialize from the buffer. - **Example**:

```
import numpy as np
from dx_engine import InferenceEngine, InferenceOption

# Load model file into memory buffer
with open('model.dxnn', 'rb') as f:
    buffer = np.frombuffer(f.read(), dtype=np.uint8)

# Create engine from memory buffer
engine = InferenceEngine.from_buffer(buffer)

# With custom options
option = InferenceOption()
option.use_ort = True
engine2 = InferenceEngine.from_buffer(buffer, option)
```

- **Use Cases**: - Loading models from encrypted or compressed storage - Network-based model distribution - Embedded systems with models stored in ROM - Dynamic model loading without filesystem access

MEMBER FUNCTIONS

`dispose(self)`

- **Signature**: `def dispose(self) -> None`
- **Description**: Explicitly releases the underlying C++ resources held by the inference engine. This is automatically called when using a `with` statement, so manual invocation is typically **not** required.

`get_all_task_outputs(self)`

- **Signature**: `def get_all_task_outputs(self) -> List[List[np.ndarray]]`
- **Description**: Retrieves the outputs of all internal tasks in their execution order. This is useful for debugging the intermediate steps of a model.
- **Returns**: A list of lists, where each inner list contains the output `np.ndarray` objects for a single task.

`get_bitmatch_mask(self, index: int = 0)`

- **Signature**: `def get_bitmatch_mask(self, index: int = 0) -> np.ndarray`
- **Description**: Retrieves a bitmatch mask for a specific NPU task, which can be used for validation and debugging purposes.
- **Parameters**:
- `index` : `int` . The index of the NPU task.
- **Returns**: A boolean `np.ndarray` representing the bitmatch mask.

`get_compile_type(self)`

- **Signature**: `def get_compile_type(self) -> str`

© Copyright 2026 DEEPX All Rights Reserved.

- **Description**: Returns the compilation type or strategy of the loaded model (e.g., "debug", "release").
- **Returns**: The compilation type as a string.

`get_input_size(self)`
- **Signature**: `def get_input_size(self) -> int`
- **Description**: Gets the total expected size of all input tensors combined in bytes.
- **Returns**: The total input size as an integer.

`get_input_tensor_count(self)`
- **Signature**: `def get_input_tensor_count(self) -> int`
- **Description**: Returns the number of input tensors required by the model.
- **Returns**: The number of input tensors.

`get_input_tensor_names(self)`
- **Signature**: `def get_input_tensor_names(self) -> List[str]`
- **Description**: Returns the names of all input tensors in the order they should be provided.
- **Returns**: A list of input tensor names.

`get_input_tensor_sizes(self)`
- **Signature**: `def get_input_tensor_sizes(self) -> List[int]`
- **Description**: Gets the individual sizes of each input tensor in bytes, in their correct order.
- **Returns**: A list of integer sizes.

`get_input_tensor_to_task_mapping(self)`
- **Signature**: `def get_input_tensor_to_task_mapping(self) -> Dict[str, str]`
- **Description**: Returns the mapping from input tensor names to their target tasks within the model graph.
- **Returns**: A dictionary mapping tensor names to task names.

`get_input_tensors_info(self)`
- **Signature**: `def get_input_tensors_info(self) -> List[Dict[str, Any]]`
- **Description**: Returns detailed information for each input tensor.
- **Returns**: A list of dictionaries, where each dictionary contains keys: `'name'` (str), `'shape'` (List[int]), `'dtype'` (np.dtype), and `'elem_size'` (int).

`get_latency(self)`
- **Signature**: `def get_latency(self) -> int`
- **Description**: Returns the latency of the most recent inference in microseconds.
- **Returns**: The latency value as an integer.

`get_latency_count(self)`
- **Signature**: `def get_latency_count(self) -> int`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Description**: Returns the total count of latency values collected since initialization.
- **Returns**: The number of measurements.

`get_latency_list(self)`
- **Signature**: `def get_latency_list(self) -> List[int]`
- **Description**: Returns a list of recent latency measurements in microseconds.
- **Returns**: A list of integers.

`get_latency_mean(self)`
- **Signature**: `def get_latency_mean(self) -> float`
- **Description**: Returns the mean (average) of all collected latency values.
- **Returns**: The mean latency as a float.

`get_latency_std(self)`
- **Signature**: `def get_latency_std(self) -> float`
- **Description**: Returns the standard deviation of all collected latency values.
- **Returns**: The standard deviation as a float.

`get_model_version(self)`
- **Signature**: `def get_model_version(self) -> str`
- **Description**: Returns the DXNN file format version of the loaded model.
- **Returns**: The model version string.

`get_npu_inference_time(self)`
- **Signature**: `def get_npu_inference_time(self) -> int`
- **Description**: Returns the pure NPU processing time for the most recent inference in microseconds.
- **Returns**: The NPU inference time as an integer.

`get_npu_inference_time_count(self)`
- **Signature**: `def get_npu_inference_time_count(self) -> int`
- **Description**: Returns the total count of NPU inference time values collected.
- **Returns**: The number of measurements.

`get_npu_inference_time_list(self)`
- **Signature**: `def get_npu_inference_time_list(self) -> List[int]`
- **Description**: Returns a list of recent NPU inference time measurements in microseconds.
- **Returns**: A list of integers.

`get_npu_inference_time_mean(self)`
- **Signature**: `def get_npu_inference_time_mean(self) -> float`
- **Description**: Returns the mean (average) of all collected NPU inference times.
- **Returns**: The mean time as a float.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

`get_npu_inference_time_std(self)`
- **Signature**: `def get_npu_inference_time_std(self) -> float`
- **Description**: Returns the standard deviation of all collected NPU inference times.
- **Returns**: The standard deviation as a float.

`get_num_tail_tasks(self)`
- **Signature**: `def get_num_tail_tasks(self) -> int`
- **Description**: Returns the number of 'tail' tasks (tasks with no successors) in the model graph.
- **Returns**: The number of tail tasks.

`get_output_size(self)`
- **Signature**: `def get_output_size(self) -> int`
- **Description**: Gets the total size of all output tensors combined in bytes.
- **Returns**: The total output size as an integer.

`get_output_tensor_count(self)`
- **Signature**: `def get_output_tensor_count(self) -> int`
- **Description**: Returns the number of output tensors produced by the model.
- **Returns**: The number of output tensors.

`get_output_tensor_names(self)`
- **Signature**: `def get_output_tensor_names(self) -> List[str]`
- **Description**: Returns the names of all output tensors in the order they are produced.
- **Returns**: A list of output tensor names.

`get_output_tensor_sizes(self)`
- **Signature**: `def get_output_tensor_sizes(self) -> List[int]`
- **Description**: Gets the individual sizes of each output tensor in bytes, in their correct order.
- **Returns**: A list of integer sizes.

`get_output_tensors_info(self)`
- **Signature**: `def get_output_tensors_info(self) -> List[Dict[str, Any]]`
- **Description**: Returns detailed information for each output tensor.
- **Returns**: A list of dictionaries with keys: `'name'` (str), `'shape'` (List[int]), `'dtype'` (np.dtype), and `'elem_size'` (int).

`get_task_order(self)`
- **Signature**: `def get_task_order(self) -> np.ndarray`
- **Description**: Returns the execution order of tasks/subgraphs within the model.
- **Returns**: A numpy array of strings representing the task order.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

is_multi_input_model(self)
- **Signature**: `def is_multi_input_model(self) -> bool`
- **Description**: Checks if the loaded model requires multiple input tensors.
- **Returns**: `True` if the model has multiple inputs, `False` otherwise.

is_ppu(self)
- **Signature**: `def is_ppu(self) -> bool`
- **Description**: Checks if the loaded model utilizes a Post-Processing Unit (PPU).
- **Returns**: `True` if the model uses a PPU, `False` otherwise.

register_callback(self, callback: Optional[Callable[[List[np.ndarray], Any], int]])
- **Signature**: `def register_callback(self, callback: Optional[Callable[[List[np.ndarray], Any], int]]) -> None`
- **Description**: Registers a user-defined callback function to be executed upon completion of an asynchronous inference.
- **Parameters**:
- `callback`: A callable function or `None` to unregister. The callback receives the list of output arrays and the user argument.

run(self, input_data: Union[np.ndarray, List[np.ndarray], List[List[np.ndarray]]], output_buffers: Optional[Union[List[np.ndarray], List[List[np.ndarray]]]] = None, user_args: Optional[Union[Any, List[Any]]] = None)
- **Signature**: `def run(self, input_data, output_buffers=None, user_args=None) -> Union[List[np.ndarray], List[List[np.ndarray]]]`
- **Description**: Runs inference synchronously. This versatile method handles single-item, multi-input, and batch inference based on the format of `input_data`.
- **Parameters**:
- `input_data`: Input data in various formats (`np.ndarray`, `List[np.ndarray]`, `List[List[np.ndarray]]`).
- `output_buffers`: Optional pre-allocated buffers for the output.
- `user_args`: Optional user-defined argument or list of arguments for batch mode.
- **Returns**: The inference result(s). A `List[np.ndarray]` for single inference or a `List[List[np.ndarray]]` for batch inference.

run_async(self, input_data: Union[np.ndarray, List[np.ndarray]], user_arg: Any = None, output_buffer: Optional[Union[np.ndarray, List[np.ndarray]]] = None)
- **Signature**: `def run_async(self, input_data, user_arg=None, output_buffer=None) -> int`
- **Description**: Runs inference asynchronously for a single item. Batch processing is **not** supported with this method.
- **Parameters**:
- `input_data`: A single `np.ndarray` or a `List[np.ndarray]` for multi-input models.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- `user_arg` : An optional user-defined argument to be passed to the callback.
- `output_buffer` : An optional pre-allocated buffer for the output.
- **Returns**: An integer `job_id` for this asynchronous operation.

`run_async_multi_input(self, input_tensors: Dict[str, np.ndarray], user_arg: Any = None, output_buffer: Optional[List[np.ndarray]] = None)`
- **Signature**: `def run_async_multi_input(self, input_tensors, user_arg=None, output_buffer=None) -> int`
- **Description**: A convenience method to run asynchronous inference on a multi-input model using a dictionary of named tensors.
- **Parameters**:
- `input_tensors` : A dictionary mapping input tensor names to `np.ndarray` data.
- `user_arg` : An optional user-defined argument.
- `output_buffer` : An optional list of pre-allocated output arrays.
- **Returns**: An integer `job_id` .

`run_benchmark(self, num_loops: int, input_data: Optional[List[np.ndarray]] = None)`
- **Signature**: `def run_benchmark(self, num_loops: int, input_data: Optional[List[np.ndarray]] = None) -> float`
- **Description**: Runs a performance benchmark for a specified number of loops.
- **Parameters**:
- `num_loops` : The number of inference iterations to run.
- `input_data` : An optional list of `np.ndarray` to use as input for the benchmark.
- **Returns**: The average frames per second (FPS) as a float.

`run_multi_input(self, input_tensors: Dict[str, np.ndarray], output_buffers: Optional[List[np.ndarray]] = None, user_arg: Any = None)`
- **Signature**: `def run_multi_input(self, input_tensors, output_buffers=None, user_arg=None) -> List[np.ndarray]`
- **Description**: A convenience method to run synchronous inference on a multi-input model using a dictionary of named tensors.
- **Parameters**:
- `input_tensors` : A dictionary mapping input tensor names to `np.ndarray` data.
- `output_buffers` : An optional list of pre-allocated output arrays.
- `user_arg` : An optional user-defined argument.
- **Returns**: A list of `np.ndarray` objects containing the output.

`wait(self, job_id: int)`
- **Signature**: `def wait(self, job_id: int) -> List[np.ndarray]`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Description**: Waits for an asynchronous job (identified by `job_id`) to complete and retrieves its output.
- **Parameters**:
- `job_id`: The integer job ID returned from a `run_async` call.
- **Returns**: A list of `np.ndarray` objects containing the output from the completed job.

---

class dx_engine.InferenceOption

This class provides a Pythonic interface to configure inference options such as device selection and core binding. It wraps the C++ `InferenceOption` struct.

CONSTRUCTOR

`__init__(self)`
- **Signature**: `def __init__(self) -> None`
- **Description**: Initializes a new `InferenceOption` object with default values from the C++ backend.

PROPERTIES

`bound_option`
- **Description**: Gets or sets the NPU core binding strategy.
- **Type**: `InferenceOption.BOUND_OPTION` (Enum).

`devices`
- **Description**: Gets or sets the list of device IDs to be used for inference. An empty list means all available devices will be used.
- **Type**: `List[int]`.

`use_ort`
- **Description**: Gets or sets whether to use the ONNX Runtime for executing CPU-based tasks in the model graph.
- **Type**: `bool`.

`buffer_count`
- **Description**: Gets or sets the number of internal buffers allocated for inference. Higher values can improve throughput in pipelined inference scenarios by allowing more concurrent operations, but consume more memory. Default is `6` (delegate to the library default). When set explicitly, valid range is 1-100. - **Type**: `int`. - **Example**:

```
from dx_engine import InferenceEngine, InferenceOption

option = InferenceOption()
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
option.buffer_count = 8  # Allocate 8 buffers for higher throughput
engine = InferenceEngine("model.dxnn", option)
```

MEMBER FUNCTIONS

`get_bound_option(self)`
- **Signature**: `def get_bound_option(self) -> BOUND_OPTION`
- **Description**: Returns the current NPU core binding option.
- **Returns**: An `InferenceOption.BOUND_OPTION` enum member.

`get_devices(self)`
- **Signature**: `def get_devices(self) -> List[int]`
- **Description**: Returns the list of device IDs targeted for inference.
- **Returns**: A list of integers.

`get_use_ort(self)`
- **Signature**: `def get_use_ort(self) -> bool`
- **Description**: Returns whether ONNX Runtime usage is enabled.
- **Returns**: A boolean value.

`get_buffer_count(self)`
- **Signature**: `def get_buffer_count(self) -> int`
- **Description**: Returns the current buffer count configuration.
- **Returns**: An integer representing the number of buffers.

`set_bound_option(self, boundOption: BOUND_OPTION)`
- **Signature**: `def set_bound_option(self, boundOption: BOUND_OPTION)`
- **Description**: Sets the NPU core binding option.
- **Parameters**:
- `boundOption`: An `InferenceOption.BOUND_OPTION` enum member.

`set_devices(self, devices: List[int])`
- **Signature**: `def set_devices(self, devices: List[int])`
- **Description**: Sets the list of device IDs to be used for inference.
- **Parameters**:
- `devices`: A list of integers representing device IDs.

`set_use_ort(self, use_ort: bool)`
- **Signature**: `def set_use_ort(self, use_ort: bool)`
- **Description**: Enables or disables the use of ONNX Runtime for CPU tasks.
- **Parameters**:
- `use_ort`: A boolean value.

© Copyright 2026 DEEPX All Rights Reserved.

`set_buffer_count(self, buffer_count: int)`
- **Signature**: `def set_buffer_count(self, buffer_count: int)`
- **Description**: Sets the number of internal buffers for inference.
- **Parameters**:
- `buffer_count`: An integer value specifying the number of buffers (valid range: 1-100).

NESTED CLASSES

`class BOUND_OPTION(Enum)`
- **Description**: An enumeration defining how NPU cores are utilized.
- **Members**: `NPU_ALL`, `NPU_0`, `NPU_1`, `NPU_2`, `NPU_01`, `NPU_12`, `NPU_02`.

---

class dx_engine.Configuration

Provides access to the global DXRT configuration singleton, allowing for system-wide settings like enabling the profiler.

CONSTRUCTOR

`__init__(self)`
- **Signature**: `def __init__(self)`
- **Description**: Initializes the Configuration object by getting a reference to the underlying C++ singleton instance.

MEMBER FUNCTIONS

`get_attribute(self, item: ITEM, attrib: ATTRIBUTE)`
- **Signature**: `def get_attribute(self, item: ITEM, attrib: ATTRIBUTE) -> str`
- **Description**: Retrieves the value of a specific attribute for a configuration item.
- **Parameters**:
- `item`: The configuration category (e.g., `Configuration.ITEM.PROFILER`).
- `attrib`: The attribute to retrieve (e.g., `Configuration.ATTRIBUTE.PROFILER_SHOW_DATA`).
- **Returns**: The attribute value as a string.

`get_driver_version(self)`
- **Signature**: `def get_driver_version(self) -> str`
- **Description**: Returns the version of the installed device driver.
- **Returns**: The driver version string.

`get_enable(self, item: ITEM)`
- **Signature**: `def get_enable(self, item: ITEM) -> bool`

© Copyright 2026 DEEPX All Rights Reserved.

- **Description**: Checks if a specific configuration item is enabled.
- **Parameters**:
- `item`: The configuration category to check.
- **Returns**: `True` if enabled, `False` otherwise.

`get_pcie_driver_version(self)`
- **Signature**: `def get_pcie_driver_version(self) -> str`
- **Description**: Returns the version of the installed PCIe driver.
- **Returns**: The PCIe driver version string.

`get_version(self)`
- **Signature**: `def get_version(self) -> str`
- **Description**: Returns the version of the DXRT library.
- **Returns**: The library version string.

`load_config_file(self, file_name: str)`
- **Signature**: `def load_config_file(self, file_name: str)`
- **Description**: Loads configuration settings from a specified file.
- **Parameters**:
- `file_name`: The path to the configuration file.

`set_attribute(self, item: ITEM, attrib: ATTRIBUTE, value: str)`
- **Signature**: `def set_attribute(self, item: ITEM, attrib: ATTRIBUTE, value: str)`
- **Description**: Sets a string value for a specific attribute of a configuration item (e.g., setting `PROFILER_SAVE_DATA` to `"ON"`).
- **Parameters**:
- `item`: The configuration category.
- `attrib`: The attribute to set.
- `value`: The string value to assign.

`set_enable(self, item: ITEM, enabled: bool)`
- **Signature**: `def set_enable(self, item: ITEM, enabled: bool)`
- **Description**: Enables or disables a global configuration item, such as `PROFILER`.
- **Parameters**:
- `item`: The configuration category.
- `enabled`: A boolean value to enable (`True`) or disable (`False`) the item.

NESTED CLASSES

`class ITEM`
- **Description**: An enumeration-like class defining configuration categories.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Members**: `DEBUG`, `PROFILER`, `SERVICE`, `DYNAMIC_CPU_THREAD`, `TASK_FLOW`, `SHOW_THROTTLING`, `SHOW_PROFILE`, `SHOW_MODEL_INFO`, `CUSTOM_INTRA_OP_THREADS`, `CUSTOM_INTER_OP_THREADS`.

`class ATTRIBUTE`
- **Description**: An enumeration-like class defining attributes for configuration items.
- **Members**: `PROFILER_SHOW_DATA`, `PROFILER_SAVE_DATA`, `CUSTOM_INTRA_OP_THREADS_NUM`, `CUSTOM_INTER_OP_THREADS_NUM`.

---

class dx_engine.DeviceStatus

Provides an interface to query real-time status and static information about hardware devices.

CLASS METHODS

`get_current_status(cls, deviceId: int)`
- **Signature**: `def get_current_status(cls, deviceId: int) -> object`
- **Description**: Retrieves the status for the device with the specified ID. The status includes key metrics such as core utilization and temperature. Monitoring data is refreshed every 1 second.
- **Parameters**:
- `deviceId`: The integer ID of the device to query.
- **Returns**: An instance of `DeviceStatus`.
- **Raises**: `RuntimeError` if the specified device ID is invalid or does not exist.

`get_device_count(cls)`
- **Signature**: `def get_device_count(cls) -> int`
- **Description**: Returns the total number of hardware devices detected by the system. This includes devices that are initialized and ready for operation.
- **Returns**: The number of devices as an integer.

INSTANCE METHODS

`get_id(self)`
- **Signature**: `def get_id(self) -> int`
- **Description**: Returns the unique ID of the device associated with this `DeviceStatus` instance.
- **Returns**: The device ID as an integer.

`get_npu_clock(self, ch: int)`
- **Signature**: `def get_npu_clock(self, ch: int) -> int`
- **Description**: Returns the current clock frequency of a specific NPU core. The clock frequency may change dynamically depending on performance scaling settings.
- **Parameters**:

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- `ch` : The integer index of the NPU core.
- **Returns**: The clock speed in MHz.

`get_npu_voltage(self, ch: int)`
- **Signature**: `def get_npu_voltage(self, ch: int) -> int`
- **Description**: Returns the current voltage of a specific NPU core. The voltage level can vary depending on power management settings and workload.
- **Parameters**:
- `ch` : The integer index of the NPU core.
- **Returns**: The voltage in millivolts (mV).

`get_temperature(self, ch: int)`
- **Signature**: `def get_temperature(self, ch: int) -> int`
- **Description**: Returns the current temperature of a specific NPU core. Monitoring the temperature is crucial for ensuring the NPU operates within safe thermal limits.
- **Parameters**:
- `ch` : The integer index of the NPU core.
- **Returns**: The temperature in degrees Celsius. Returns `INT16_MIN` when the channel index is out of range.

`get_core_utilization(self, core_id: int)` - **Signature**: `def get_core_utilization(self, core_id: int) -> float` - **Description**: Retrieves the utilization of the specified NPU core. - **Parameters**: - `core_id` : The NPU core index (0 to 2). - **Returns**: Utilization percentage (0.0 ~ 100.0). Returns -1.0 if core_id is out of range.

`get_memory_used(self)` - **Signature**: `def get_memory_used(self) -> int` - **Description**: Retrieves the amount of NPU DRAM currently in use. - **Returns**: DRAM usage in bytes.

`get_memory_free(self)` - **Signature**: `def get_memory_free(self) -> int` - **Description**: Retrieves the amount of free NPU DRAM. - **Returns**: Free DRAM in bytes.

---

class dx_engine.RuntimeEventDispatcher

A singleton class that provides a centralized event dispatching mechanism for runtime events such as device errors, warnings, and notifications. It supports custom event handlers and automatic logging of events with different severity levels.

CONSTRUCTOR

`__init__(self)`
- **Description**: Obtains the singleton instance of the underlying C++ `RuntimeEventDispatcher` .

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Instantiating the Python class is the standard entry point.
- **Example**:

```
from dx_engine import RuntimeEventDispatcher
dispatcher = RuntimeEventDispatcher()
```

NESTED CLASSES

`class LEVEL(IntEnum)`
- **Description**: Event severity levels for categorizing runtime events.
- **Members**:
- `INFO` (1): Informational messages for normal operation events.
- `WARNING` (2): Warning messages for potential issues that don't stop execution.
- `ERROR` (3): Error messages for recoverable failures.
- `CRITICAL` (4): Critical errors that may cause system instability.

`class TYPE(IntEnum)`
- **Description**: Event type categories for classifying the source of events.
- **Members**:
- `DEVICE_CORE` (1000): Events related to NPU core operations.
- `DEVICE_STATUS` (1001): Device status change events.
- `DEVICE_IO` (1002): Input/Output operation events.
- `DEVICE_MEMORY` (1003): Memory management events.
- `UNKNOWN` (1004): Unknown or unclassified event types.

`class CODE(IntEnum)`
- **Description**: Specific event codes for identifying the exact nature of events.
- **Members**:
- `WRITE_INPUT` (2000): Input data write operation event.
- `READ_OUTPUT` (2001): Output data read operation event.
- `MEMORY_OVERFLOW` (2002): Memory overflow or capacity exceeded.
- `MEMORY_ALLOCATION` (2003): Memory allocation failure or issue.
- `DEVICE_EVENT` (2004): General device event notification.
- `RECOVERY_OCCURRED` (2005): Device recovery action taken.
- `TIMEOUT_OCCURRED` (2006): Operation timeout event.
- `THROTTLING_NOTICE` (2007): Device throttling notification.
- `THROTTLING_EMERGENCY` (2008): Device throttling emergency notification.
- `UNKNOWN` (2009): Unknown or unclassified event code.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

INSTANCE METHODS

`dispatch_event(self, level: LEVEL, type: TYPE, code: CODE, event_message: str)`

- **Signature**:

`def dispatch_event(self, level: LEVEL, type: TYPE, code: CODE, event_message: str) -> None`

- **Description**: Dispatches a runtime event with the specified parameters. The event is logged and any registered custom event handler is invoked. Events are filtered based on the current level threshold set via `set_current_level`.

- **Parameters**:

- `level`: Severity level of the event (`INFO`, `WARNING`, `ERROR`, `CRITICAL`).

- `type`: Category of the event (`DEVICE_CORE`, `DEVICE_IO`, etc.).

- `code`: Specific event code identifying the exact event.

- `event_message`: Descriptive string providing event details.

- **Raises**: `ValueError` if `event_message` is not a string.

`register_event_handler(self, handler: Callable[[int, int, int, str, str], None])`

- **Signature**:

`def register_event_handler(self, handler: Callable[[int, int, int, str, str], None]) -> None`

- **Description**: Registers a user-defined callback function that will be invoked for each dispatched event. Only one handler can be registered at a time; subsequent calls will replace the previous handler. The handler is invoked synchronously but with minimal lock holding time to avoid blocking.

- **Parameters**:

- `handler`: A callable with signature `handler(level, type, code, message, timestamp) -> None`.

- **Raises**: `ValueError` if `handler` is not callable.

- **Example**:

```
from dx_engine import RuntimeEventDispatcher

def my_handler(level, type, code, message, timestamp):
    print(f"[{timestamp}] Level {level}: {message}")

dispatcher = RuntimeEventDispatcher()
dispatcher.register_event_handler(my_handler)
```

`set_current_level(self, level: LEVEL)`

- **Signature**: `def set_current_level(self, level: LEVEL) -> None`

- **Description**: Sets the minimum event level threshold. Events below this level may be filtered out by custom handlers.

- **Parameters**:

- `level`: Minimum severity level for events to be processed.

© Copyright 2026 DEEPX All Rights Reserved.

get_current_level(self)
- **Signature**: `def get_current_level(self) -> LEVEL`
- **Description**: Gets the current minimum event level threshold.
- **Returns**: The current minimum event severity level as a `LEVEL` enum value.

Standalone Functions

dx_engine.inference_engine.parse_model(model_path: str)
- **Signature**: `def parse_model(model_path: str) -> str`
- **Description**: Module-level function (not a class method) that parses a `.dxnn` model file using the C++ backend and returns a string containing information about the model's structure and properties. It is defined at the top level of `dx_engine.inference_engine` and is not re-exported from the `dx_engine` package root, so it must be imported from its fully qualified module path.
- **Parameters**:
- `model_path` : The path to the compiled model file.
- **Returns**: A string with model information.
- **Raises**: `TypeError` if `model_path` is not a `str`.
- **Import**:

```
from dx_engine.inference_engine import parse_model
```

- **Example**:

```
from dx_engine.inference_engine import parse_model

info = parse_model("model.dxnn")
print(info)
```

> **NOTE**
>
> A separate function `dx_engine.utils.parse_model(model_path, options=None) -> int` also exists with a different signature and an `int` return code (0 on success, -1 on failure). The function documented here (returning `str`) is the recommended one for retrieving model information.

© Copyright 2026 DEEPX All Rights Reserved.

11. Change Log

11.1 DX-RT v3.4.1 (July 2026)

This is a documentation-only hotfix. The `libdxrt` and `libdxrt-bin` Debian packages and `dx_engine` Python wheels are unchanged and remain at v3.4.0.

Changed

- None

Fixed

- Corrected supported OS list in installation guide: removed Ubuntu 18.04 (EOL), added Ubuntu 26.04
- Clarified ONNX Runtime version recommendation wording in installation guide (`02_Installation_on_Linux.md`)
- Removed stray emoji characters from C++ and Python tutorial pages

Added

- Added bundled wheel installation guide (Option B) for virtual environments in installation guide
- Added info note documenting `/usr/share/libdxrt-bin/python/` bundled wheel directory usage

---

11.2 DX-RT v3.4.0 (July 2026)

Changed

- Debian package bundles `dx_engine` Python wheels for Python 3.8 – 3.14.
- CLI binary names updated for consistency: `dxrt-cli` → `dxcli`, `parse_model` → `dxparse`, `run_model` → `dxrun`
- Old names (`dxrt-cli`, `parse_model`, `run_model`) remain functional as backward-compatible aliases (symlinks on Linux/macOS, hard links on Windows)
- Python bindings now use dxrt_cxx_api.h (header-only) instead of direct C++ linkage
- libdxrt.so exports C symbols only (dxrt_*); internal C++ symbols hidden via version script

© Copyright 2026 DEEPX All Rights Reserved.

- Update minimum versions
- Driver : 2.4.0 -> 2.5.0
- PCIe Driver : 2.2.0 -> 2.4.0
- Firmware : 2.5.2 -> 2.7.0
- Code style cleanup across Python package (`inference_option.py` , `runtime_event_dispatcher.py` , `utils.py` , etc.)
- Upgrade extern/include/dxrt/extern/cxxopts.hpp from v3.1.1 to v3.3.1
- Increase firmware update sequence wait time from 2s to 4s.
- Replace top-level matplotlib imports with lazy loading via `ensure_dependencies()` in `plot.py` to provide install guidance instead of ImportError

Fixed

- Improved shared-memory performance.
- Fix SEGV in NFHLayer caused by dereferencing uninitialized `Request::_task` in profiler instrumentation
- Return `INT16_MIN` instead of `0` from `DeviceStatus::GetTemperature()` when channel index is out of range
- Improve dxrtd error messages when manually executed
- Fix missing separator between `devices` and `buffer_count` fields in `InferenceOption.__repr__` output
- Update ppcpu logic for hardness
- Fixed a critical crash (nullptr) in pyRunBenchmark by ensuring the memory lifetime of input buffers.
- Improve temperature range validation in dxtop.
- Fix multi-input dictionary handling issue.

Added

- Script to build `dx_engine` wheels per Python version.
- Documentation describing the Debian prebuilt directory structure.
- Script to generate the `prebuilt/` directory.
- Stable C ABI (dxrt_c_api.h, 103 functions) enabling prebuilt SDK distribution without recompilation
- Header-only C++ wrapper (dxrt_cxx_api.h) for single-include modern C++ usage
- SDK-compat bridge headers (legacy/) for backward compatibility with existing dxrt_api.h code
- H1M firmware compatibility check: distinguish H1M (LPDDR4) from H1 (LPDDR5/LPDDR5X) in firmware update and device detection; support mixed 4-pack / 6-pack H1M configurations

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Deprecated Python API migration guide added to Python API Reference documentation
- Add shared memory-based inter-process communication using memfd
- Add dynamic IPC infrastructure with packet-based protocol layer
- Add cross-platform support for Linux and Windows IPC transport
- Add new device monitoring APIs (memory usage, per-core temperature, and utilization)
- Add a tool (plot_html.py) to generate HTML visualization reports for profiling data (profiler.json)
- Profiler
- Add GetJobMetrics() API for comprehensive per-job profiling
- Enhance performance statistics with Coefficient of Variation (CoV) metric
- Add input dtype validation to prevent undefined behavior caused by mismatching NumPy array types.
- Add `libdxrt-bin` Debian packaging for pre-built binaries with amd64/arm64 auto-detection.

11.3 DX-RT v3.3.2 (May 2026)

Changed

- Removed redundant build artifacts and temporary directories from the Debian package.

Fixed

- Improve Python extension module linking for _pydxrt build.

Added

- Added conditional pip upgrade (v21.3+) to ensure build stability on legacy OS environments.

11.4 DX-RT v3.3.1 (April 2026)

Changed

- Change the version of pre-built onnxruntime(1.23.2 -> 1.22.0) and openvino(25.4 -> 25.1)
- Enhance uninstall script to remove debian packages, legacy binaries, and test files from previous versions

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Fixed

- Fix typos in the document
- Fix error in uninstall logic

Added

- Add libdxrt 3.3.1 debian package with updated build and install pipeline

---

11.5 DX-RT v3.3.0 (April 2026)

Changed

- Update minimum versions
- Driver : 1.8.0 -> 2.4.0
- PCIe Driver : 1.5.1 -> 2.2.0
- Firmware : 2.4.0 -> 2.5.2
- Update the Python module version to match the C++ Runtime version.

Fixed

- Fix PPU data transfer error during multi-process execution in H1 and Multiple M1 M.2 environments.
- Fix input data lifecycle issues in the Python Runtime Module.
- Fix intermittent interrupt exceptions in IPC Message Queue.

Added

- Add dxtop for No Service Mode
- Add an example that loads an entire model file into a memory buffer and performs inference directly using this memory buffer.
- Add python InferenceEngine from numpy array
- Add acceleration features for CPU operations (Requires separate option configuration and build)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11.6 DX-RT v3.2.0 (December 2025)

Changed

- Optimize PCIe DMA sequence for better performance.
- Update OS requirements in installation guide for debian

Fixed

- Optimize device memory footprint of PPU models

Added

- Add RuntimeEventDispatcher class for C++
- RuntimeEventDispatcher is a singleton class that provides centralized event dispatching and handling for runtime events from the DX-RT system, such as device errors, warnings, and notifications.
- Add Python wrapper for RuntimeEventDispatcher class -> dx_engine 1.1.4
- Add `--profiler` option to enable profiling mode in run_model.py
- Add `--buffer-count` option to configure inference buffer count in run_model.py
- Add build.sh options

```
--use_service_on  Enable the use of the service in the build.
--use_service_off Disable the use of the service in the build.
--use_ort_on      Enable the use of the ORT component in the build.
--use_ort_off     Disable the use of the ORT component in the build.
```

- Add `__version__` import to the main `dx_engine` module namespace
- Implement per-instance configuration of Input and Output buffer counts for the Inference Engine.
- Enable direct loading of the .dxnn model format from a memory buffer within the Inference Engine.
- Add RuntimeEventDispatcher for centralized event handling and logging.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11.7 DX-RT v3.1.0 (November 2025)

Changed

• Update minimum versions

```
- Driver: 1.7.1 → 1.8.0
- PCIe Driver: 1.4.1 → 1.5.1
- Firmware: 2.1.0 → 2.4.0
```

• Standardize command-line argument parsing across all examples

```
- Use argparse (Python) and cxxopts (C++)
- Unified argument format: -m/--model, -l/--loops, etc.
```

• Update SanityCheck.sh

```
- Remove redundant libdxrt.so location check (validated via run_model execution)
- Hide ONNX Runtime version when built with USE_ORT=OFF
```

• Enhance build and installation scripts

```
- Use 'python3 -m pip' instead of 'pip' for better reliability
- Add Python version compatibility check in build.sh
- Improve logging with color-coded messages
- Add uninstall.sh script for project cleanup
```

• Improve profiler functionality

```
- Add Flush and GetPerformanceData functions
- Group events by base name instead of individual job/request entries
- Limit duration details to 30 values per group for cleaner output
- Add memory usage tracking with high usage warnings
```

• Improve parse_model CLI tool with enhanced functionality

• Support dynamic shape output for tail CPU tasks

• Update C++ exception handling to translate exceptions into Python

Fixed

• Fix multiple bugs

```
- Multi-tasking bugs related to CPU offloading and PPU output buffer
- PPU model format and layout issues
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
- Multi-output and multi-tail configuration bugs
- Tensor mapping errors in non-ORT inference mode
- Output tensor mapping and memory address configuration
- run_model error when -f option and -l loop count exceeds 1024
- Bounding issues on service
```

Added

• Add support for V8 DXNN file format with PPU support

• Add model voltage profiler tool (run_model_prof.py)

```
- Requires firmware ≥ 2.2.0 and driver ≥ 1.7.1
```

• Add dxbenchmark CLI tool for performance comparison across multiple models

```
- Default execution of 30 loops when neither loop nor time options are specified
- Automatically creates result directory when result path is specified
```

• Add system requirements check in install.sh

```
- Verify RAM ≥ 8GB
- Verify architecture (x86_64 or aarch64)
- Exit with error if requirements not met
```

• Add model file format version range validation

```
- Error messages now indicate valid version range (min to max)
```

• Add time-based inference mode to run_model (-t, --time option)

• Add error handling for invalid firmware files and update conditions

• Add dxtop enhancements

```
- Display PCIe bus number
- Update cross-compilation script
```

• Add new Python APIs for device configuration and status retrieval

• Add new documentation files for Inference API, Multi-Input Inference, and Global Instance

• Add examples for asynchronous model inference with profiling capabilities

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11.8 DX-RT v3.0.0 (September 2025)

- Update the .dxnn file format to version 7 (from v6).
- Update C++ exception handling to translate exceptions into Python for improved error handling.
- Update the Python v6_converter with enhanced functionality.
- Add support for models with multiple heads, tails, inputs, and outputs, including new C++ and Python APIs.
- Add a new internal C++ converter for v6 models.
- Add new Python APIs for handling device configuration and status retrieval.
- Fix several multi-tasking bugs related to CPU offloading buffer management and PPU output buffer mis-pointing.
- Fix a bug in the process of setting the PPU model format and layout.
- Fix a critical bug affecting models with multi-output and multi-tail configurations.
- Fix tensor mapping errors that occurred in non-ORT inference mode.
- Fix a warning message in get_output_tensors_info and a vector access bug in _npuModel.
- Fix an issue that prevented error messages from being displayed.
- Fix flaws in output tensor mapping and memory address configuration.
- Enhance OS and architecture checks in installation scripts
- Update documentation to reflect changes in supported CPU architecture and OS requirements.
- Enhance build and uninstall scripts with common utilities and improved logging
- Add PCIe bus number display for dxtop
- Add profiling data memory usage tracking with high usage warnings.
- Update user guide document
- Force-disabled with a warning instead of throwing a runtime exception in builds that don't support USE_ORT.
- Add time-based inference mode to run_model (-t, --time option)
- Profiler now groups events by base name (before) instead of showing individual job/request entries
- Limited duration details to 30 values per group for cleaner output
- Fix run_model error when -f option and -l loop count exceeds 1024
- Fix bounding option issue on service
- Add error handling for invalid firmware files and update conditions.
- Add a function to check Python version compatibility in build.sh.
- Add new documentation files for Inference API, Multi-Input Inference, and Global Instance.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

• Add examples for asynchronous model inference with profiling capabilities in both C++ and Python.

• Update minimum versions

```
- Driver : 1.5.0 -> 1.7.1
- PCIe Driver : 1.4.0 -> 1.4.1
- Firmware : 2.0.5 -> 2.1.0
```

• Fix kernel panic issue caused by wrong NPU channel number

• Update DeviceOutputWorker to use 4 threads for 4 DMA channels (3 channels to 4 channels)

• Improve error message readability in install, build scripts

```
- Apply color to error messages
- Reorder message output to display errors before help messages
```

• Update Python Package version (v1.1.1 -> v1.1.2)

• Modify run_async_model and run_async_model_output examples

• Modify build.sh (print python package install info)

• removed some unnecessary items from header files

• use Pyproject.toml instead of setup.py (now setup.py is not recommended)

• Fix some rapidjson issue from clients.

• Remove bad using namespace std from model.h (some programs need change)

• Add usb inference module (tcp/ip) (MACRO : DXRT_USB_NETWORK_DRIVER)

• Add options to SanityCheck.sh

```
- Usage: sudo SanityCheck.sh [all(default) | dx_rt | dx_driver | help]
```

• The build compiler has been updated to version 14 for both USE_ORT=ON and USE_ORT=OFF configurations.

• Fix an issue where temporary files from the ONNX Runtime installation would accumulate.

• Fix a cross-compilation error related to the ncurses library for the dxtop utility.

• Add Sanity Check Features

```
- Dependency version check.
- Executable file check.
```

• Add APIs to the Configuration class for retrieving version information.

• PCIE details displayed on some device errors

• dxrt-cli --errorstat option added (this shows pcie detailed information)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Modify run_model logging to include host info (Linux only).
- Add Python examples for configuration and device status.
- Add Python API for configuration and device status. (dx-engine-1.1.1)
- Add functionality to query the framework & driver versions in the Configuration class.
- Add weight checksum info for service
- Add ENABLE_SHOW_MODEL_INFO build option and configuration item
- Update code for compatibility with v3 environment
- Enhance UI for better clarity, enabled dynamic data rendering, and added visual graphs for NPU Memory usage.
- Fix: fix dx-rt build error caused by pybind11 incompatibility with Python 3.6.9 on Ubuntu 18.04

```
- Support automatic installation of minimum required Python version (>= 3.8.2)
- Install Python 3.8.2 if the system Python version is not supported
- On Ubuntu 18.04, install via source build; on Ubuntu 20.04+, use apt install
- Added support in install.sh to optionally accept --python_version and --venv_path
for installation
- Added support in build.sh to accept and use --python_exec
- Added support in build.sh to optionally accept --venv_path and activate the
specified virtual environment
```

- The default build option for DX-RT has been changed from USE_ORT=OFF to USE_ORT=ON. If the inference engine option is not specified separately, use_ort will be enabled by default, activating the CPU task for .dxnn models.
- Add dxtop tool, a terminal-based monitoring tool for Linux environments. It provides real-time insights into NPU core utilization and DRAM usage per NPU device.

11.9 DX-RT v2.9.5 (May 2025)

- Added full support for Python run_model.
- Updated the run_model option and its description
- Improve the Python API

```
- InferenceOption is now supported identically to the C++ API.
   - set_devices(...) → devices = [0]
   - set_bound_option(...) → bound_option = InferenceOption.BOUND_OPTION.NPU_ALL
   - set_use_ort(...) → use_ort = True
- Callback functions registered via register_callback now accept user_arg of custom
types. (removed .value)
   - user_arg.value → user_arg
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
- run() now supports both single-input and batch-input modes, depending on the input
format.
```

• Modify the build.sh script according to cmake options.

```
- CMake option USE_ORT=ON, running build.sh --clean installs ONNX Runtime.
- CMake option USE_PYTHON=ON, running build.sh installs the Python package.
- CMake option USE_SERVICE=ON, running build.sh starts or restarts the service.
```

• Add dxrt-cli -v to display minimum driver & compiler versions

• Addressed multithreading issues by implementing additional locks, improving stability under heavy load.

• Fix crash on multi-device environments with more than 2 H1 cards. (>=8 devices)

• Resolved data corruption errors that could occur in different scenarios, ensuring data integrity.

• Fix profiler bugs.

• Addressed issues identified by static analysis and other tools, enhancing code quality and reliability.

• Add --use_ort flag to the run_model.py example for ONNX Runtime.

• Add run batch function. (Python & C++)

```
- batch inference with multiple inputs and multiple outputs.
```

• Minimum model file versions

```
- .dxnn file format version >= v6
- compiler version >= v1.15.2
```

• Minimum Driver and Firmware versions

```
- RT Driver Version >= v1.5.0
- PCIe Driver Version >= v1.4.0
- Firmware Version >= v2.0.5
```

11.10 DX-RT v2.8.2 (April 2025)

• Modify Inference Engine to be used with 'with' statements, and update relevant examples.

• Add Python inference option interface with the following configurations

• NPU Device Selection / NPU Bound Option / ORT Usage Flag

• Display dxnn versions in parse_model (.dxnn file format version & compiler version)

• Added instructions on how to retrieve device status information

© Copyright 2026 DEEPX All Rights Reserved.

• Driver and Firmware versions

```
- RT Driver >= v1.3.3
- Firmware >= v1.6.3
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.