
ROCm Data Center Documentation

Advanced Micro Devices, Inc.

Apr 02, 2025

INSTALL

1	Installing and running RDC	3
1.1	Prerequisites	3
1.2	Install gRPC	3
1.3	Authentication keys	3
1.4	Prebuilt packages	3
1.5	Components	4
1.6	Starting RDC	4
2	Building and testing RDC	9
2.1	Prerequisites for Building RDC	9
2.2	Build and Install RDC	9
2.3	Authentication	10
2.4	Verify Files for Authentication	11
3	Introduction to the RDC tool	13
3.1	Target Audience	13
3.2	Objective	13
3.3	Terminology	14
4	RDC tool feature overview	15
4.1	Discovery	15
4.2	Groups	16
4.3	Device Monitoring	17
4.4	Job Stats	17
4.5	Error-Correcting Code Output	19
4.6	Diagnostic	19
5	Third party integration	21
5.1	Python bindings	21
5.2	Prometheus plugin	22
5.3	Grafana Plugin	25
5.4	Prometheus (Grafana) integration with automatic node detection	28
5.5	Reliability, Availability, and Serviceability Plugin	32
6	Introduction to RDC API	35
6.1	RDC API	35
6.2	Job stats use case	35
7	API reference	39
8	License	65

The ROCm Data Center tool (RDC) simplifies the administration of, and addresses key infrastructure challenges in AMD GPUs in cluster and datacenter environments. The main features are of RDC include:

- GPU telemetry
- GPU statistics for jobs
- Integration with third-party tools
- Open source

You can access the RDC tool on [GitHub repository](#).

The documentation is structured as follows:

Install

- *[Installing and running RDC](#)*
- *[Building and testing RDC](#)*

How to

- *[Introduction to the RDC tool](#)*
- *[RDC tool feature overview](#)*
- *[Third party integration](#)*

API reference

- *[Introduction to RDC API](#)*
- *[API reference](#)*

To contribute to the documentation, refer to [Contributing to ROCm](#).

You can find licensing information on the [Licensing](#) page.

INSTALLING AND RUNNING RDC

The ROCm Data Center tool (RDC) is part of the AMD ROCm software and available on the distributions supported by AMD ROCm. For RDC installation from prebuilt packages, follow the instructions in this section.

1.1 Prerequisites

The installation dependencies are described in [Dependencies in the README](#). To see the list of supported operating systems, refer to [System requirements](#).

1.2 Install gRPC

To see the instructions for building gRPC and `protoc`, refer to [Building gRPC and protoc](#).

1.3 Authentication keys

RDC can be used with or without authentication. If authentication is required you must configure proper authentication keys as described in [Authentication in Building and testing RDC](#).

1.4 Prebuilt packages

RDC is packaged as part of the ROCm software repository. You must install the AMD ROCm software before installing RDC, as described in [ROCm installation](#).

To install RDC after installing the ROCm package, use the following instructions.

Ubuntu

```
$ sudo apt-get install rdc
# or, to install a specific version
$ sudo apt-get install rdc<x.y.z>
```

SLES 15 Service Pack 3

```
$ sudo zypper install rdc
# or, to install a specific version
$ sudo zypper install rdc<x.y.z>
```

1.5 Components

The components of the RDC tool are as shown below:

1.5.1 RDC (API) library

This library is the central piece, which interacts with different modules and provides all the features described. This shared library provides C API and Python bindings so that third-party tools should be able to use it directly if required.

1.5.2 RDC daemon (rdcd)

The rdcd daemon records telemetry information from GPUs. It also provides an interface to RDC command-line tool (rdci) running locally or remotely. It relies on the above RDC Library for all the core features.

1.5.3 RDC command-line tool (rdci)

A command-line tool to invoke all the features of the RDC tool. This CLI can be run locally or remotely.

1.5.4 AMDSMI library

A stateless system management library that provides low-level interfaces to access GPU information

1.6 Starting RDC

The RDC tool can be run in the following two modes. The feature set is similar in both the cases. You have the flexibility to choose the option that best fits your environment.

- *Standalone mode*
- *Embedded mode*

The capability in each mode depends on the privileges you have for starting the RDC tool. A normal user has access only to monitor (GPU telemetry) capabilities. A privileged user can run the tool with full capability. In the full capability mode, GPU configuration features can be invoked. This may or may not affect all the users and processes sharing the GPU.

1.6.1 Standalone mode

This is the preferred mode of operation, as it does not have any external dependencies. To start RDC in standalone mode, RDC Server Daemon (rdcd) must run on each compute node. Refer to *Terminology* in *Introduction to the RDC tool* for more information. You can start rdcd as a systemd service or directly from the command-line.

Start the RDC tool using systemd

If multiple RDC versions are installed, copy `/opt/rocm-<x.y.z>/libexec/rdc/rdc.service`, which is installed with the desired RDC version, to the systemd folder. The capability of RDC can be configured by modifying the `rdc.service` system configuration file. Use the `systemctl` command to start rdcd.

```
$ systemctl start rdc
```

By default, rdcd starts with full capability. To change to monitor only, comment out the following two lines:

```
$ sudo vi /lib/systemd/system/rdc.service
# CapabilityBoundingSet=CAP_DAC_OVERRIDE
# AmbientCapabilities=CAP_DAC_OVERRIDE
```

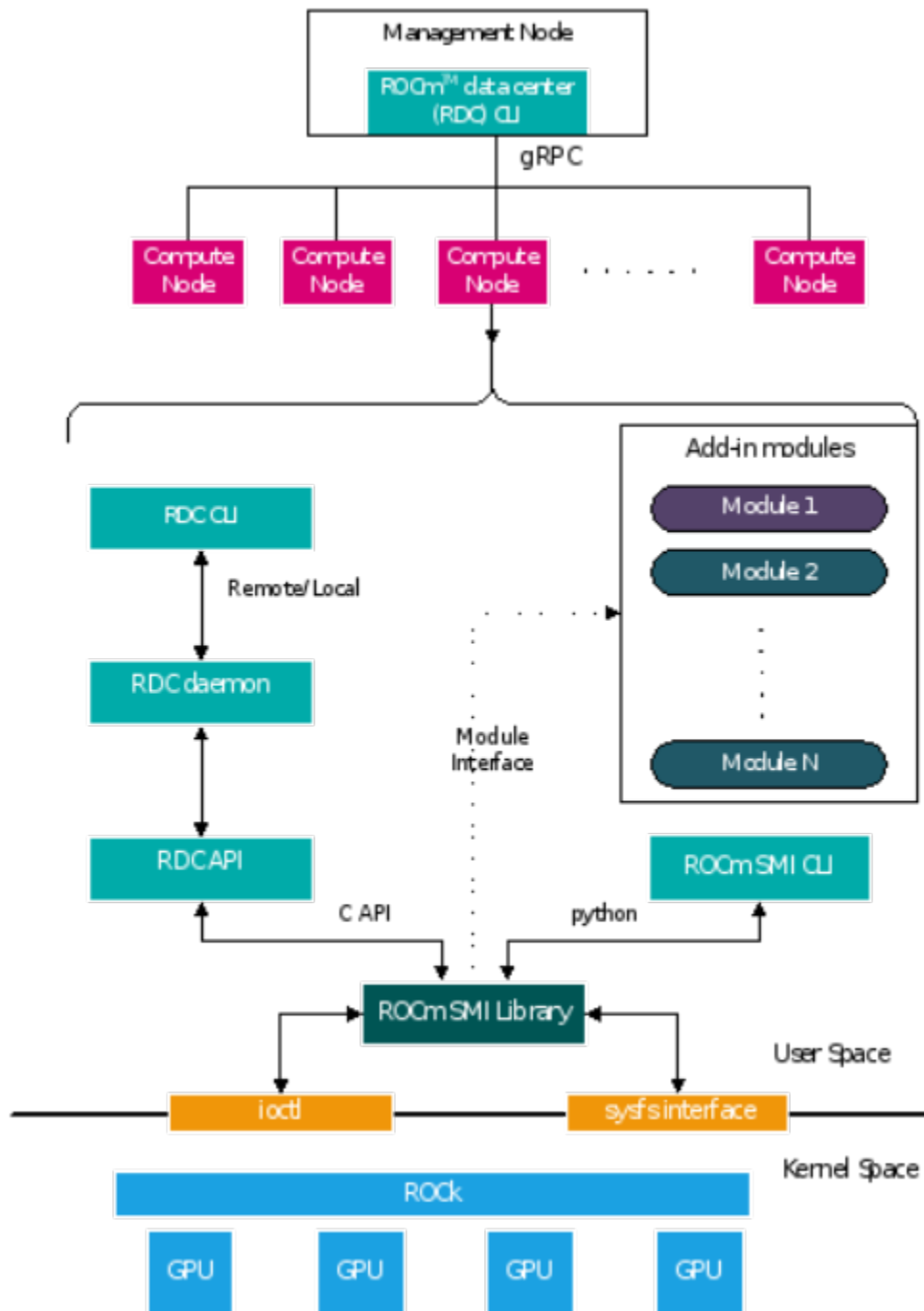


Fig. 1.1: High-level diagram of RDC components

Note

rdcd can be started by using the `systemctl` command.

```
$ systemctl start rdc
```

If the GPU reset fails, restart the server. Note that restarting the server also initiates `rdcd`. You may then encounter the following two scenarios:

- `rdcd` returns the correct GPU information to `rdci`
- `rdcd` returns the “No GPUs found on the system” error to `rdci`. To resolve this error, restart `rdcd` with the following instruction:

```
$ sudo systemctl restart rdc
```

Start the RDC tool from the command-line

While `systemctl` is the preferred way to start `rdcd`, you can also start directly from the command-line. The installation scripts create a default user - `rdc`. Users have the option to edit the profile file (`rdc.service` installed at `/lib/systemd/system`) and change these lines accordingly:

```
[Service]
User=rdc
Group=rdc
```

From the command-line, start `rdcd` as a user such as `rdc`, or start it as `root`:

```
#Start as user rdc
$ sudo -u rdc rdc

# Start as root
$ sudo rdc
```

In this use case, the `rdc.service` file mentioned in the previous section is not involved. Here, the capability of RDC is determined by the privilege of the user starting `rdcd`. If `rdcd` is running under a normal user account it has the monitor-only capability. If `rdcd` is running as `root` then it has the full capability.

Note

If a user other than `rdc` or `root` starts the `rdcd` daemon, the file ownership of the SSL keys mentioned in the Authentication section must be modified to allow read and write access.

1.6.2 Troubleshoot `rdcd`

When `rdcd` is started using `systemctl`, the logs can be viewed using the following command:

```
$ journalctl -u rdc
```

These messages provide useful status and debugging information. The logs can also help debug problems like `rdcd` failing to start, communication issues with a client, and others.

1.6.3 Embedded mode

The embedded mode is useful if the end user has a monitoring agent running on the compute node. The monitoring agent can directly use the RDC library and will have a finer-grain control on how and when RDC features are invoked. For example, if the monitoring agent has a facility to synchronize across multiple nodes, it can synchronize GPU telemetry across these nodes.

The RDC daemon `rdcd` can be used as a reference code for this purpose. The dependency on `gRPC` is also eliminated if the RDC library is directly used.

 **Caution**

RDC command-line `rdci` will not function in this mode. Third-party monitoring software is responsible for providing the user interface and remote access/monitoring.

BUILDING AND TESTING RDC

RDC is open source and available under the MIT License. This section is helpful for open source developers. Third-party integrators may also find this information useful.

2.1 Prerequisites for Building RDC

Note

RDC is tested on the following software versions. Earlier versions may not work.

- CMake 3.15
- g++ (5.4.0)
- AMD ROCm, which includes AMD AMDSMI Library
- gRPC and protoc

The following components are required to build the latest documentation:

- Doxygen (1.8.11)
- Latex (pdfTeX 3.14159265-2.6-1.40.16)

```
$ sudo apt install libcap-dev
$ sudo apt install -y doxygen
```

2.2 Build and Install RDC

To build and install, clone the RDC source code from GitHub and use CMake.

```
$ git clone <GitHub for RDC>
$ cd rdc
$ mkdir -p build; cd build
$ cmake -DROCM_DIR=/opt/rocm -DGRPC_ROOT="$GRPC_PROTOC_ROOT"..
$ make
#Install library file and header and the default location is /opt/rocm
$ make install
```

2.2.1 Build Documentation

You can generate PDF documentation after a successful build. The reference manual, `refman.pdf`, appears in the `latex` directory.

```
$ make doc
$ cd latex
$ make
```

2.2.2 Build Unit Tests for RDC Tool

```
$ cd rdc/tests/rdc_tests
$ mkdir -p build; cd build
$ cmake -DROCM_DIR=/opt/rocm -DGRPC_ROOT="$GRPC_PROTOC_ROOT"..
$ make

# To run the tests

$ cd build/rdctst_tests
$ ./rdctst
```

2.2.3 Test

```
# Run rdcd daemon
$ LD_LIBRARY_PATH=$PWD/rdc_libs/ ./server/rdcd -u

# In another console run the RDC command-line
$ LD_LIBRARY_PATH=$PWD/rdc_libs/ ./rdci/rdci discovery -l -u
```

2.3 Authentication

RDC supports encrypted communications between clients and servers.

2.3.1 Generate Files for Authentication

The communication between the client and server can be configured to be authenticated or unauthenticated. By default, authentication is enabled.

To disable authentication, when starting the server, use the “`-unauth_comm`” flag (or “`-u`” for short). You must also use “`-u`” in `rdci` to access `unauth rdcd`. The `/lib/systemd/system/rdc.service` file can be edited to pass arguments to `rdcd` on starting. On the client side, when calling `rdc_channel_create()`, the “`secure`” argument must be set to `False`.

2.3.2 Scripts

RDC users manage their own keys and certificates. However, some scripts generate self-signed certificates in the RDC source tree in the authentication directory for test purposes. The following flowchart depicts how to generate the root certificates using the `openssl` command in `01gen_root_cert.sh`:

A picture containing sign, drawing Description automatically generated

The section where the default responses to `openssl` questions can be specified is included in `openssl.conf`. To locate the section look for the following comment line:



Fig. 2.1: Generation of root certificates using openssl command

```
# < ** REPLACE VALUES IN THIS SECTION WITH APPROPRIATE VALUES FOR YOUR ORG. **>
```

It is helpful to modify this section with values appropriate for your organization if you expect to call this script many times. Additionally, you must replace the dummy values and update the `alt_names` section for your environment.

To generate the keys and certificates using these scripts, make the following calls:

```
$ 01gen_root_cert.sh
# provide answers to posed questions
$ 02gen_ssl_artifacts.sh
# provide answers to posed questions
```

At this point, the keys and certificates are in the newly created `CA/artifacts` directory.

Important

You must delete this directory if you need to rerun the scripts.

To install the keys and certificates, access the artifacts directory and run the `install.sh` script as root, specifying the install location. By default, RDC expects this to be in `/etc/rdc`:

```
$ cd CA/artifacts
$ sudo install_<client|server>.sh /etc/rdc
```

These files must be copied to and installed on all client and server machines that are expected to communicate with one another.

2.3.3 Known Limitation

RDC has the following authentication limitations:

The client and server are hardcoded to look for the `openssl` certificate and key files in `/etc/rdc`. There is no workaround available currently.

2.4 Verify Files for Authentication

Several SSL keys and certificates must be generated and installed on clients and servers for authentication to work properly. By default, the RDC server will look in the `/etc/rdc` folder for the following keys and certificates:

2.4.1 Client

```
$ sudo tree /etc/rdc
/etc/rdc
|-- client
```

(continues on next page)

(continued from previous page)

```
|-- certs
| |-- rdc_cacert.pem
| |-- rdc_client_cert.pem
|-- private
|-- rdc_client_cert.key
```

Note

Machines that are clients and servers consist of both directory structures.

2.4.2 Server

```
$ sudo tree /etc/rdc
/etc/rdc
|-- server
|-- certs
| |-- rdc_cacert.pem
| |-- rdc_server_cert.pem
|-- private
|-- rdc_server_cert.key
```

INTRODUCTION TO THE RDC TOOL

The ROCm Data Center tool (RDC) simplifies the administration and addresses key infrastructure challenges in AMD GPUs in cluster and datacenter environments. The main features are:

- GPU telemetry
- GPU statistics for jobs
- Integration with third-party tools
- Open source

You can use the RDC tool in standalone mode if all components are installed. However, the existing management tools can use the same set of features available in a library format.

For details on different modes of operation, refer to *Starting RDC* in *Installing and running RDC*.

3.1 Target Audience

The audience for the AMD RDC tool consists of:

- Administrators: RDC provides the cluster administrator with the capability of monitoring, validating, and configuring policies.
- HPC Users: Provides GPU-centric feedback for their workload submissions.
- OEM: Add GPU information to their existing cluster management software.
- Open source Contributors: RDC is open source and accepts contributions from the community.

3.2 Objective

This documentation will:

1. Introduce the tool features in *RDC tool feature overview*
2. Describe integration with external tools in *Third party integration*
3. Provide an open source handbook in *Building and testing RDC*
4. Introduce elements of the tool API in *Introduction to RDC API*

3.3 Terminology

Table 3.1: Terminologies and Abbreviations

Terms	Description
RDC	ROCm Data Center tool
Compute node (CN)	One of many nodes containing one or more GPUs in the Data Center on which compute jobs are run
Management node (MN) or Main console	A machine running system administration applications to administer and manage the Data Center
GPU Groups	Logical grouping of one or more GPUs in a compute node
Fields	A metric that can be monitored by the RDC, such as GPU temperature, memory usage, and power usage
Field Groups	Logical grouping of multiple fields
Job	A workload that is submitted to one or more compute nodes

RDC TOOL FEATURE OVERVIEW

This topic provides information related to the features of the RDC tool.

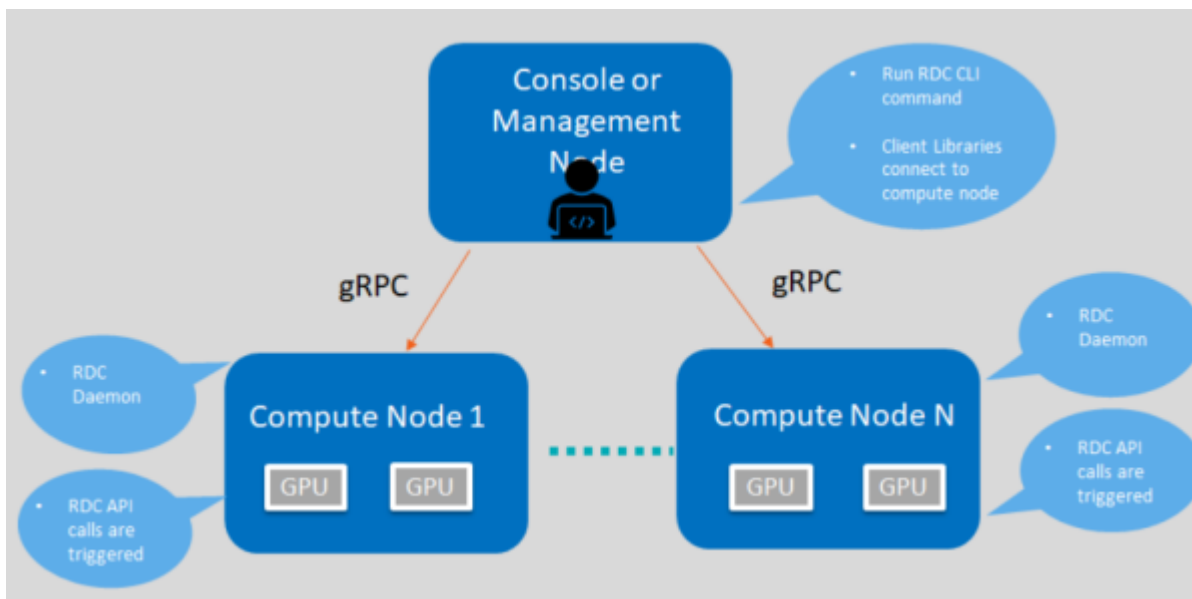


Fig. 4.1: RDC components and framework for describing features

4.1 Discovery

The Discovery feature enables you to locate and display information of GPUs present in the compute node.

Example:

```
$ rdc discovery <host_name> -l  
2 GPUs found
```

GPU Index	Device Information
0	Name: AMD Radeon Instinct MI50 Accelerator
1	Name: AMD Radeon Instinct MI50 Accelerator

```
$ rdc -l : list available GPUs  
$ rdc -u: No SSL authentication
```

4.2 Groups

This section explains the GPU and field groups features.

4.2.1 GPU Groups

With the GPU groups feature, you can create, delete, and list logical groups of GPU.

```
$ rdc group -c GPU_GROUP
Successfully created a group with a group ID 1

$ rdc group -g 1 -a 0,1
Successfully added the GPU 0,1 to group 1

$ rdc group -l

1 group found
```

Group ID	Group Name	GPU Index
1	GPU_GROUP	0, 1

```
$ rdc group -d 1
Successfully removed group 1

-c create; -g group id; -a add GPU index; -l list; -d delete group
```

4.2.2 Field Groups

The Field Groups feature provides you the options to create, delete, and list field groups.

```
$ rdc fieldgroup -c <fgroup> -f 150,155
Successfully created a field group with a group ID 1

$ rdc fieldgroup -l

1 group found
```

Group ID	Group Name	Field IDs
1	Fgroup	150, 155

```
$ rdc fieldgroup -d 1
Successfully removed field group 1

rdc dmon -l
Supported fields Ids:
100 RDC_FI_GPU_CLOCK: Current GPU clock freq.
150 RDC_FI_GPU_TEMP: GPU temp. in milli Celsius.
155 RDC_FI_POWER_USAGE: Power usage in microwatts.
203 RDC_FI_GPU_UTIL: GPU busy percentage.
525 RDC_FI_GPU_MEMORY_USAGE: VRAM Memory usage in bytes
```

(continues on next page)

(continued from previous page)

```
-c create; -g group id; -a add GPU index; -l list; -d delete group
```

4.2.3 Monitor Errors

You can define RDC_FI_ECC_CORRECT_TOTAL or RDC_FI_ECC_UNCORRECT_TOTAL field to get the RAS Error-Correcting Code (ECC) counter:

- 312 RDC_FI_ECC_CORRECT_TOTAL: Accumulated correctable ECC errors
- 313 RDC_FI_ECC_UNCORRECT_TOTAL: Accumulated uncorrectable ECC errors

4.3 Device Monitoring

The RDC tool enables you to monitor the GPU fields.

```
$ rdcidmon -f <field_group> -g <gpu_group> -c 5 -d 1000
```

```
1 group found
```

GPU Index	TEMP (m°C)	POWER (µW)
0	25000	520500

```
rdcidmon -l
Supported fields Ids:
100 RDC_FI_GPU_CLOCK: Current GPU clock freq.
150 RDC_FI_GPU_TEMP: GPU temp. in milli Celsius.
155 RDC_FI_POWER_USAGE: Power usage in microwatts.
203 RDC_FI_GPU_UTIL: GPU busy percentage.
525 RDC_FI_GPU_MEMORY_USAGE: VRAM Memory usage in bytes

-e field ids; -i GPU index; -c count; -d delay; -l list; -f fieldgroup id
```

4.4 Job Stats

You can display GPU statistics for any given workload.

```
$ rdcidmon -s 2 -g 1
Successfully started recording job 2 with a group ID 1

$ rdcidmon -j 2
```

Summary	Executive Status
Start time	1586795401
End time	1586795445
Total execution time	44
Energy Consumed (Joules)	21682
Power Usage (Watts)	Max: 49 Min: 13 Avg: 34
GPU Clock (MHz)	Max: 1000 Min: 300 Avg: 903
GPU Utilization (%)	Max: 69 Min: 0 Avg: 2
Max GPU Memory Used (bytes)	524320768
Memory Utilization (%)	Max: 12 Min: 11 Avg: 12

```
$ rdcj stats -x 2
Successfully stopped recording job 2

-s start recording on job id; -g group id; -j display job stats; -x stop recording.
```

4.4.1 Job Stats Use Case

A common use case is to record GPU statistics associated with any job or workload. The following example shows how all these features can be put together for this use case:

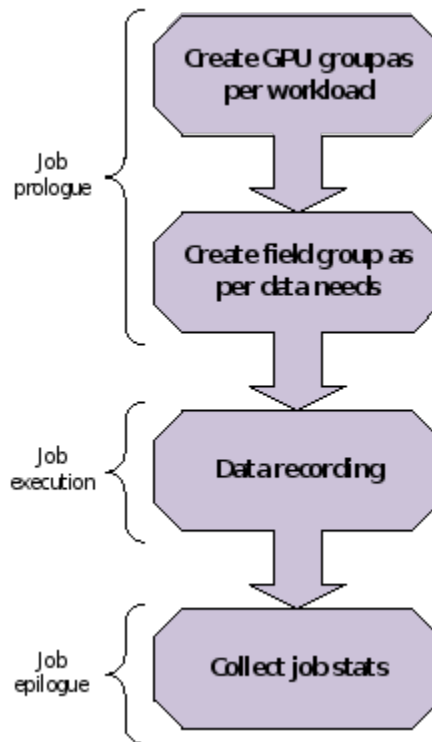


Fig. 4.2: An example showing how job statistics can be recorded

rdci commands

```

$ rdc group -c group1

successfully created a group with a group ID 1

$ rdc group -g 1 -a 0,1

GPU 0,1 is added to group 1 successfully.

rdci stats -s 123 -g 1

job 123 recorded successfully with the group ID

rdci stats -x 123

job 123 stops recording successfully

rdci stats -j 123

job stats printed

```

4.5 Error-Correcting Code Output

In the job output, this feature prints out the Error-Correcting Code (ECC) errors while running the job.

4.6 Diagnostic

You can run diagnostic on a GPU group as shown below:

```

$ rdc diag -g <gpu_group>

No compute process: Pass
Node topology check: Pass
GPU parameters check: Pass
Compute Queue ready: Pass
System memory check: Pass
===== Diagnostic Details =====
No compute process: No processes running on any devices.
Node topology check: No link detected.
GPU parameters check: GPU 0 Critical Edge temperature in range.
Compute Queue ready: Run binary search task on GPU 0 Pass.
System memory check: Max Single Allocation Memory Test for GPU 0 Pass.
↳CPUAccessToGPUmemoryTest for GPU 0 Pass. GPUAccessToCPUMemoryTest for GPU 0 Pass.

```


THIRD PARTY INTEGRATION

This section lists available third-party plugins for the RDC tool such as Prometheus, Grafana, and Reliability, Availability and Serviceability (RAS).

5.1 Python bindings

The RDC Tool provides a generic Python class `RdcReader` to simplify telemetry gathering. `RdcReader` simplifies usage by providing the following functionalities:

- The user only needs to specify telemetry fields. `RdcReader` creates the necessary groups and fieldgroups, watch the fields, and fetch the fields.
- The `RdcReader` can support embedded and standalone mode. The standalone mode can be with or without authentication.
- In standalone mode, the `RdcReader` can automatically reconnect to `rdcd` if the connection is lost.
- When `rdcd` is restarted, the previously created group and fieldgroup may be lost. The `RdcReader` can re-create them and watch the fields after reconnecting.
- If the client is restarted, `RdcReader` can detect the groups and fieldgroups created before and avoid re-creating them.
- A custom unit converter can be passed to `RdcReader` to override the default RDC unit.

See the sample program to monitor the power and GPU utilization using the `RdcReader` below:

```
from RdcReader import RdcReader
from RdcUtil import RdcUtil
from rdc_bootstrap import *

default_field_ids = [
    rdc_field_t.RDC_FI_POWER_USAGE,
    rdc_field_t.RDC_FI_GPU_UTIL
]

class SimpleRdcReader(RdcReader):
    def __init__(self):
        RdcReader.__init__(self, ip_port=None, field_ids = default_field_ids, update_
↪freq=1000000)
    def handle_field(self, gpu_index, value):
        field_name = self.rdc_util.field_id_string(value.field_id).lower()
        print("%d %d:%s %d" % (value.ts, gpu_index, field_name, value.value.l_int))
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    reader = SimpleRdcReader()
while True:
    time.sleep(1)
    reader.process()
```

In the sample program,

- Class `SimpleRdcReader` is derived from the `RdcReader`.
- The field `ip_port=None` in `RdcReader` dictates that RDC runs in the embedded mode.
- `SimpleRdcReader::process()` fetches fields specified in `default_field_ids`.

Note

`RdcReader.py` can be found in the `python_binding` folder located at RDC install path.

To run the example, use:

```
# Ensure that RDC shared libraries are in the library path and
# RdcReader.py is in PYTHONPATH

$ python SimpleReader.py
```

5.2 Prometheus plugin

Prometheus plugin helps to monitor events and send alerts. The Prometheus installation and integration details are given below.

5.2.1 Prometheus plugin installation

RDC's Prometheus plugin `rdc_prometheus.py` can be found in the `python_binding` folder.

Note

Ensure the Prometheus client is installed before the Prometheus plugin installation process.

```
$ pip install prometheus_client
```

To view the options provided with the plugin, use `--help`.

```
% python rdc_prometheus.py --help
usage: rdc_prometheus.py [-h] [--listen_port LISTEN_PORT] [--rdc_embedded]
  [--rdc_ip_port RDC_IP_PORT] [--rdc_unauth]
  [--rdc_update_freq RDC_UPDATE_FREQ]
  [--rdc_max_keep_age RDC_MAX_KEEP_AGE]
  [--rdc_max_keep_samples RDC_MAX_KEEP_SAMPLES]
  [--rdc_fields RDC_FIELDS [RDC_FIELDS ...]]
  [--rdc_fields_file RDC_FIELDS_FILE]
  [--rdc_gpu_indexes RDC_GPU_INDEXES [RDC_GPU_INDEXES ...]]
```

(continues on next page)

(continued from previous page)

```
[--enable_plugin_monitoring]
```

RDC Prometheus plugin.

optional arguments:

```
-h, --help show this help message and exit
--listen_port LISTEN_PORT
The listen port of the plugin (default: 5000)
--rdc_embedded Run RDC in embedded mode (default: standalone mode)
--rdc_ip_port RDC_IP_PORT
The rdc IP and port in standalone mode (default:
localhost:50051)
--rdc_unauth Set this option if the rdc is running with unauth in
standalone mode (default: false)
--rdc_update_freq RDC_UPDATE_FREQ
The fields update frequency in seconds (default: 10)
--rdc_max_keep_age RDC_MAX_KEEP_AGE
The max keep age of the fields in seconds (default:
3600)
--rdc_max_keep_samples RDC_MAX_KEEP_SAMPLES
The max samples to keep for each field in the cache
(default: 1000)
--rdc_fields RDC_FIELDS [RDC_FIELDS ...]
The list of fields name needs to be watched, for
example, " --rdc_fields RDC_FI_GPU_TEMP
RDC_FI_POWER_USAGE " (default: fields in the
plugin)
--rdc_fields_file RDC_FIELDS_FILE
The list of fields name can also be read from a file
with each field name in a separated line (default:
None)
--rdc_gpu_indexes RDC_GPU_INDEXES [RDC_GPU_INDEXES ...]
The list of GPUs to be watched (default: All GPUs)
--enable_plugin_monitoring
Set this option to collect process metrics of
the plugin itself (default: false)
```

By default, the plugin runs in the standalone mode and connects to rdc at localhost:50051 to fetch fields. The plugin should use the same authentication mode as rdc, e.g., if rdc is running with -u/--unauth flag, the plugin should use --rdc_unauth flag. You can use the plugin in the embedded mode without rdc by setting --rdc_embedded flag.

To override the default fields that are monitored, you can use the --rdc_fields option to specify the list of fields. If the fields list is long, the --rdc_fields_file option provides a convenient way to fetch fields list from a file. You can use the max_keep_age and max_keep_samples to control how the fields are cached.

The plugin can provide the metrics of the plugin itself, including the plugin process CPU, memory, file descriptor usage, and native threads count, including the process start and uptimes. You can enable this using --enable_plugin_monitoring.

You can test the plugin with the default settings.

```
# Ensure that rdc is running on the same machine
$ python rdc_prometheus.py
```

(continues on next page)

(continued from previous page)

```
# Check the plugin using curl
$ curl localhost:5000
# HELP gpu_util gpu_util
# TYPE gpu_util gauge
gpu_util{gpu_index="0"} 0.0
# HELP gpu_clock gpu_clock
# TYPE gpu_clock gauge
gpu_clock{gpu_index="0"} 300.0
# HELP gpu_memory_total gpu_memory_total
# TYPE gpu_memory_total gauge
gpu_memory_total{gpu_index="0"} 4294.0
# HELP gpu_temp gpu_temp
# TYPE gpu_temp gauge
# HELP power_usage power_usage
# TYPE power_usage gauge
power_usage{gpu_index="0"} 9.0
# HELP gpu_memory_usage gpu_memory_usage
# TYPE gpu_memory_usage gauge
gpu_memory_usage{gpu_index="0"} 134.0
```

5.2.2 Prometheus Integration

Follow these steps:

1. Download and install Prometheus in the management machine.
2. Use the example configuration file `rdc_prometheus_example.yml` in the `python_binding` folder. You can use this file in its original state. However, note that this file refers to `prometheus_targets.json`. Ensure that this is modified to point to the correct compute nodes.

```
// Sample file: prometheus_targets.json
// Replace rdc_test*.amd.com to point the correct compute nodes
// Add as many compute nodes as necessary
[
  {
    "targets": [
      "rdc_test1.amd.com:5000",
      "rdc_test2.amd.com:5000"
    ]
  }
]
```

Note

In the above example, there are two compute nodes, `rdc_test1.adm.com` and `rdc_test2.adm.com`. Ensure that the Prometheus plugin is running on those compute nodes.

3. Start the Prometheus plugin.

```
% prometheus --config.file=<full path of the rdc_prometheus_example.yml>
```

4. From the management node, using a browser, open the URL <http://localhost:9090>.

5. Select one of the available metrics.

5.2.3 Example: gpu_clock



Fig. 5.1: The Prometheus image showing the GPU clock for both rdc_test1 and rdc_test2.

5.3 Grafana Plugin

Grafana is a common monitoring stack used for storing and visualizing time series data. Prometheus acts as the storage backend, and Grafana is used as the interface for analysis and visualization. Grafana has a plethora of visualization options and can be integrated with Prometheus for RDC’s dashboard.

5.3.1 Grafana Plugin Installation

To install Grafana plugin, follow these steps:

1. Download Grafana
2. Read the [Installation instructions](#) to install Grafana
3. To start Grafana, follow these instructions:

```
$ sudo systemctl start grafana-server
$ sudo systemctl status grafana-server
```

4. Browse to <http://localhost:3000/>.
5. Log in using the default username and password (admin/admin) as shown in the image below:

5.3.2 Grafana Integration

As a prerequisite, ensure:

- The RDC Prometheus plugin is running in each compute node.
- Prometheus is set up to collect metrics from the plugin.

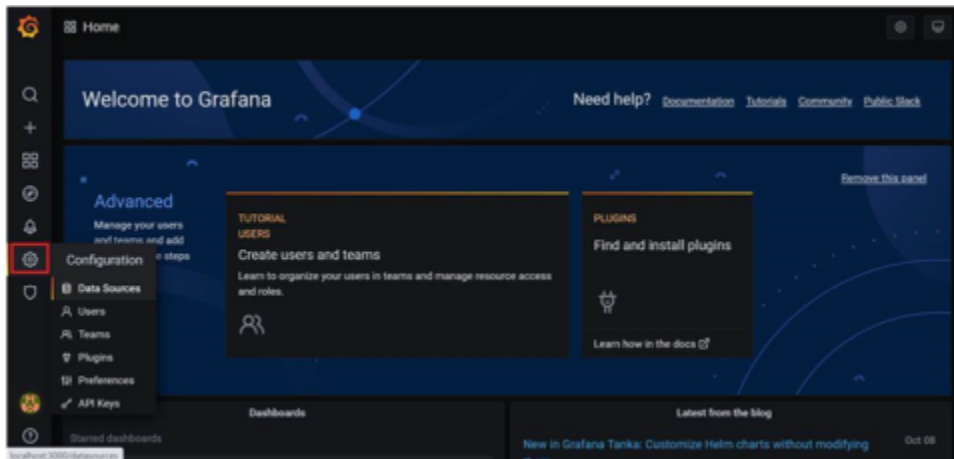
For more information about installing and configuring Prometheus, see the section on [Prometheus Plugin](#).



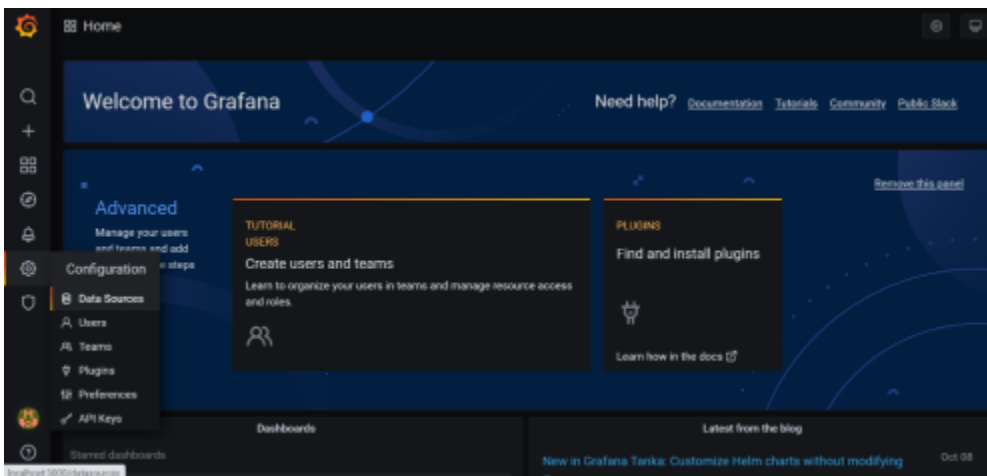
5.3.3 Grafana Configuration

Follow these steps:

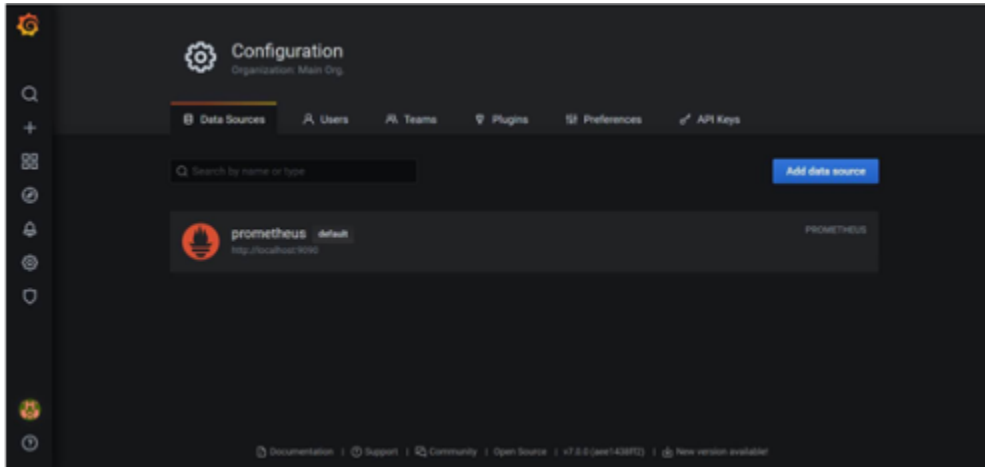
1. Click Configuration.



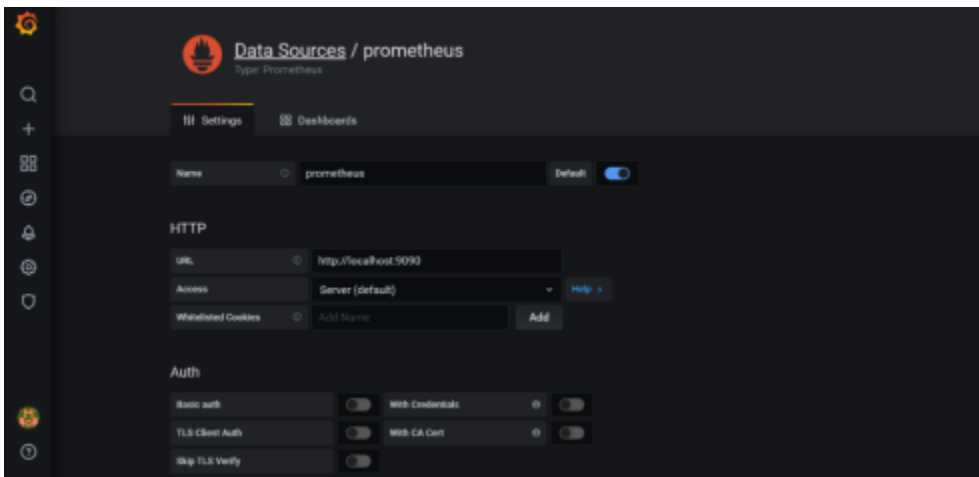
2. Select Data Sources, as shown in the image below:



3. Click Add data source.

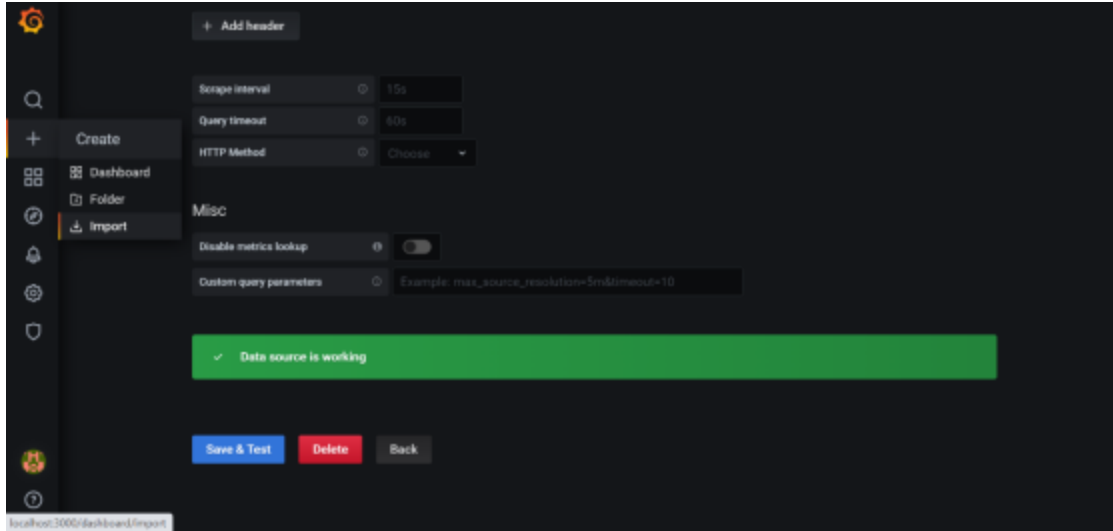


4. Select Prometheus.



Note

Ensure the name of the data source is **Prometheus**. If Prometheus and Grafana are running on the same machine, use the default URL <http://localhost:9090>. Otherwise, ensure the URL matches the Prometheus URL, save, and test it.



5. To import RDC dashboard, click + and select Import.
6. Click the Upload.json file command.
7. Choose rdc_grafana_dashboard_example.json which is in the python_binding folder.
8. Import the rdc_grafana_dashboard_example.json file and select the desired compute node on the dashboard, as shown in the image below:



5.4 Prometheus (Grafana) integration with automatic node detection

RDC enables you to use Consul to discover the `rdc_prometheus` service automatically. Consul is “a service mesh solution providing a fully featured control plane with service discovery, configuration, and segmentation functionality.” For more information, refer to [Consul](#).

RDC uses Consul for health checks of RDC’s integration with the Prometheus plug-in (`rdc_prometheus`), and these checks provide information on its efficiency.

Previously, when a new compute node was added, users had to manually change `prometheus_targets.json` to use Consul. Now, with the Consul agent integration, a new compute node can be discovered automatically.

5.4.1 Installing the Consul Agent for Compute and Management Nodes

To install the latest Consul agent for compute and management nodes, follow the instructions below:

1. Set up the apt repository to download and install the Consul agent.

```
$ curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
$ sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_
↪release -cs) main"
$ sudo apt-get update && sudo apt-get install consul
```

2. Generate a key to encrypt the communication between consul agents. Note that you can generate the key once, and both the compute and management nodes use the same key for communication.

```
$ consul keygen
```

For the purposes of this feature documentation, the following key is used in the configuration file:

```
$ consul keygen
4lgGQXr3/R2QeTi5vEp7q5Xs1KoYBhCsk9+VgJZZHAo=
```

5.4.2 Setting up the Consul Server in Management Nodes

While Consul can function with one server, it is recommended to use three to five servers to avoid failure scenarios, which often lead to data loss.

Note

For example purposes, the configuration settings documented below are for a single server.

Follow these steps:

1. Create a configuration file `/etc/consul.d/server.hcl`.

```
server = true
encrypt = "<CONSUL_ENCRYPTION_KEY>"
bootstrap_expect = 1
ui = true
client_addr = "0.0.0.0"
bind_addr = "<The IP address can be reached by client>"
```

2. Run the agent in server mode, and set the encrypt to the key generated in the first step. The `bootstrap_expect` variable indicates the number of servers required to form the first Consul cluster.
3. Set the number of servers to 1 to allow a cluster with a single server.
 - The User Interface (UI) variable is used to enable the Consul Web UI.
 - The `client_addr` variable is used to connect the API and UI.
 - The `bind_addr` variable is used to connect the client to the server. If you have multiple private IP addresses, use the address that can connect to a client.
4. Start the agent using the following instruction:

```
$ sudo consul agent -config-dir=/etc/consul.d/
```

5. Browse to <http://localhost:8500/> on the management node. You will see a single instance running.

5.4.3 Setting up the Consul Client in Compute Nodes

Follow these steps:

1. Create a configuration file `/etc/consul.d/client.hcl`.

```
server = false
encrypt = "<CONSUL_ENCRYPTION_KEY>"
retry_join = ["<The consul server address>"]
client_addr = "0.0.0.0"
bind_addr = "<The IP address can reach server>"
```

Note

Use the same `CONSUL_ENCRYPTION_KEY` as the servers. In the `retry_join`, use the IP address of the management nodes.

2. Start the Consul agent.

```
$ sudo consul agent -config-dir=/etc/consul.d/
```

The client has now joined the Consul.

```
$ consul members
Node           Address           Status  Type    Build  Protocol  DC    Segment
management-node 10.4.22.70:8301  alive  server  1.9.3  2         dc1   <all>
compute-node    10.4.22.112:8301 alive  client  1.9.3  2         dc1   <default>
```

3. Set up the Consul client to monitor the health of the RDC Prometheus plugin.
4. Start the RDC Prometheus plugin.

```
$ python rdc_prometheus.py --rdc_embedded
```

5. Add the configuration file `/etc/consul.d/rdc_prometheus.hcl`.

```
{
  "service": {
    "name": "rdc_prometheus",
    "tags": [
      "rdc_prometheus",
      "rdc"
    ],
    "port": 5000,
    "check": {
      "id": "rdc_plugin",
      "name": "RDC Prometheus plugin on port 5000",
      "http": "http://localhost:5000",
      "method": "GET",
      "interval": "15s",
      "timeout": "1s"
    }
  }
}
```

Note

By default, the Prometheus plugin uses port 5000. If you do not use the default setting, ensure you change the configuration file accordingly.

After the configuration file is changed, restart the Consul client agent.

```
$ sudo consul agent -config-dir=/etc/consul.d/
```

6. Enable the Prometheus integration in the Management node. For more information, refer to the Prometheus Integration section above.
7. In the Management node, inspect the service.

```
$ consul catalog nodes -service=rdc_prometheus
```

Node	ID	Address	DC
compute-node	76694ab1	10.4.22.112	dc1

8. Create a new Prometheus configuration `rdc_prometheus_consul.yml` file for the Consul integration.

```
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is
  →every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1
  →minute.
scrape_configs:
  - job_name: 'consul'
    consul_sd_configs:
      - server: 'localhost:8500'
    relabel_configs:
      - source_labels: [__meta_consul_tags]
        regex: .*,rdc,.*
        action: keep
      - source_labels: [__meta_consul_service]
        target_label: job
```

Note

If you are not running the consul server and Prometheus in the same machine, change the server under `consul_sd_configs` to your consul server address.

9. Start Prometheus.

```
$ ./prometheus --config.file="rdc_prometheus_consul.yml"
```

10. Browse the Prometheus UI at <http://localhost:9090> on the Management node and query RDC Prometheus metrics. Ensure that the plugin starts before running the query.

5.5 Reliability, Availability, and Serviceability Plugin

The RAS plugin helps to gather and count errors. The details of RAS integration with RDC are given below.

5.5.1 RAS Plugin Installation

In this release, RDC extends support to the Reliability, Availability, and Serviceability (RAS) integration. When the RAS feature is enabled in the graphic card, users can use RDC to monitor RAS errors.

Prerequisite

You must ensure the graphic card supports RAS.

Note

The RAS library is installed as part of the RDC installation, and no additional configuration is required for RDC.

RDC installation dynamically loads the RAS library `librdc_ras.so`. The configuration files required by the RAS library are installed in the `sp3` and `config` folders.

```
% ls /opt/rocm-4.2.0/rdc/lib
... librdc_ras.so ...
... sp3 ... config ...
```

5.5.2 RAS Integration

RAS exposes a list of Error-Correcting Code (ECC) correctable and uncorrectable errors for different IP blocks and enables users to successfully troubleshoot issues.

For example, the `dmon` command passes the `ECC_CORRECT` and `ECC_UNCORRECT` counters field id to the command.

```
$ rdc dmon -i 0 -e 600,601
```

The `dmon` command monitors GPU index 0, field 600, and 601, where 600 is for the `ECC_CORRECT` counter and 601 is for the `ECC_UNCORRECT` counter.

```
% rdc dmon -l
... ..
600 RDC_FI_ECC_CORRECT_TOTAL : Accumulated Single Error Correction.
601 RDC_FI_ECC_UNCORRECT_TOTAL : Accumulated Double Error Detection.
602 RDC_FI_ECC_SDMA_SEC : SDMA Single Error Correction.
603 RDC_FI_ECC_SDMA_DED : SDMA Double Error Detection.
604 RDC_FI_ECC_GFX_SEC : GFX Single Error Correction.
605 RDC_FI_ECC_GFX_DED : GFX Double Error Detection.
606 RDC_FI_ECC_MMHUB_SEC : MMHUB Single Error Correction.
607 RDC_FI_ECC_MMHUB_DED : MMHUB Double Error Detection.
608 RDC_FI_ECC_ATHUB_SEC : ATHUB Single Error Correction.
609 RDC_FI_ECC_ATHUB_DED : ATHUB Double Error Detection.
610 RDC_FI_ECC_BIF_SEC : BIF Single Error Correction.
611 RDC_FI_ECC_BIF_DED : BIF Double Error Detection.
612 RDC_FI_ECC_HDP_SEC : HDP Single Error Correction.
613 RDC_FI_ECC_HDP_DED : HDP Double Error Detection.
```

(continues on next page)

(continued from previous page)

```
614 RDC_FI_ECC_XGMI_WAFL_SEC : XGMI WAFL Single Error Correction.
615 RDC_FI_ECC_XGMI_WAFL_DED : XGMI WAFL Double Error Detection.
616 RDC_FI_ECC_DF_SEC : DF Single Error Correction.
617 RDC_FI_ECC_DF_DED : DF Double Error Detection.
618 RDC_FI_ECC_SMN_SEC : SMN Single Error Correction.
619 RDC_FI_ECC_SMN_DED : SMN Double Error Detection.
620 RDC_FI_ECC_SEM_SEC : SEM Single Error Correction.
621 RDC_FI_ECC_SEM_DED : SEM Double Error Detection.
622 RDC_FI_ECC_MP0_SEC : MP0 Single Error Correction.
623 RDC_FI_ECC_MP0_DED : MP0 Double Error Detection.
624 RDC_FI_ECC_MP1_SEC : MP1 Single Error Correction.

625 RDC_FI_ECC_MP1_DED : MP1 Double Error Detection.
626 RDC_FI_ECC_FUSE_SEC : FUSE Single Error Correction.
627 RDC_FI_ECC_FUSE_DED : FUSE Double Error Detection.
628 RDC_FI_ECC_UMC_SEC : UMC Single Error Correction.
629 RDC_FI_ECC_UMC_DED : UMC Double Error Detection.
... ..
```

To access the ECC correctable and uncorrectable error counters, use the following command:

```
% rdcctl dmon -i 0 -e 600,601
GPU      ECC_CORRECT      ECC_UNCORRECT
0        0                0
0        0                0
0        0                0
```


INTRODUCTION TO RDC API

Note

This is the alpha version of RDC API and is subject to change without notice. The primary purpose of this API is to solicit feedback. AMD accepts no responsibility for any software breakage caused by API changes.

6.1 RDC API

RDC API is the core library that provides all the RDC features. This section focuses on how RDC API can be used by third-party software.

RDC includes the following libraries:

- `librdc_bootstrap.so`: Loads during runtime one of the two libraries by detecting the mode.
- `librdc_client.so`: Exposes RDC functionality using gRPC client.
- `librdc.so`: RDC API. This depends on `libamd_smi.so`.
- `libamd_smi.so`: Stateless low overhead access to GPU data.

Note

`librdc_bootstrap.so` loads different libraries based on the modes.

Example:

- `rdci`: `librdc_bootstrap.so` loads `librdc_client.so`
- `rdcd`: `librdc_bootstrap.so` loads `librdc.so`

For more information see the *API reference*.

6.2 Job stats use case

The following pseudocode shows how RDC API can be directly used to record GPU statistics associated with any job or workload. Refer to the example code provided with RDC on how to build it.

For more information, see *Job Stats* in *RDC tool feature overview*.

```
//Initialize the RDC
rdc_handle_t rdc_handle;
```

(continues on next page)

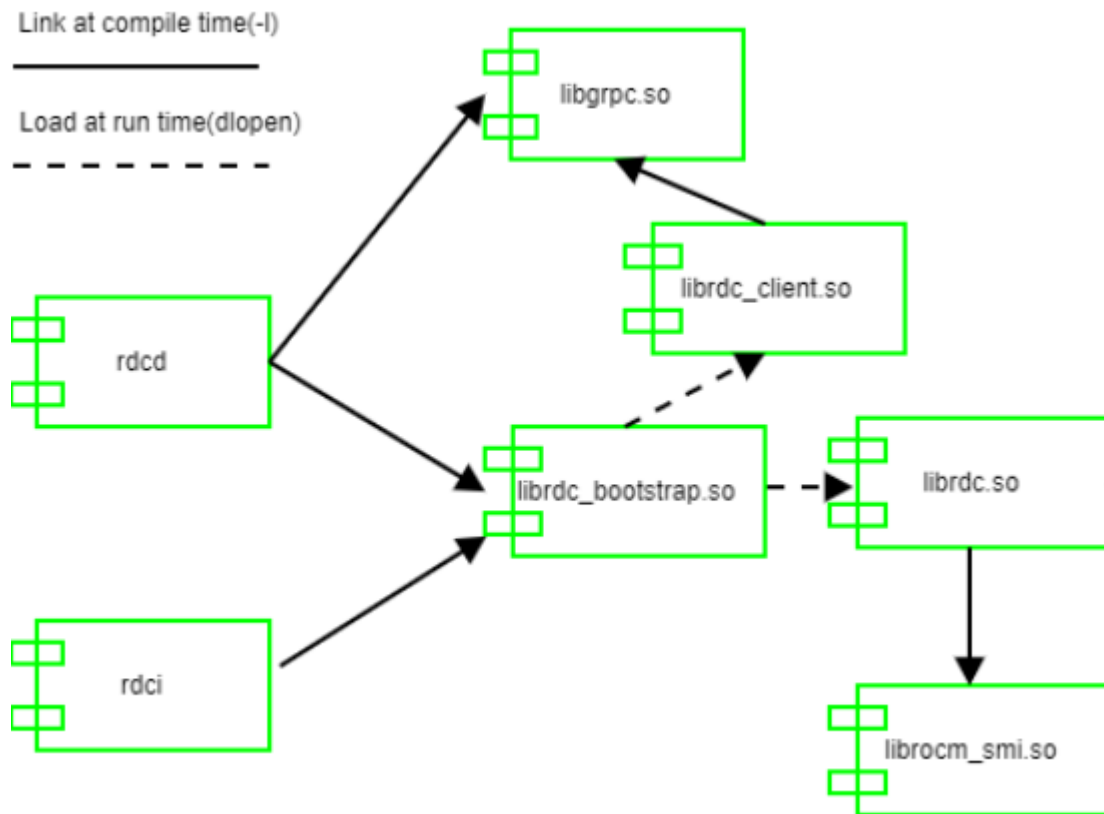


Fig. 6.1: Different libraries and how they are linked.

(continued from previous page)

```

rdc_status_t result=rdc_init(0);

//Dynamically choose to run in standalone or embedded mode
bool standalone = false;
std::cin>> standalone;
if (standalone)
    result = rdc_connect("127.0.0.1:50051", &rdc_handle, nullptr, nullptr, nullptr); //It
↳will connect to the daemon
else
    result = rdc_start_embedded(RDC_OPERATION_MODE_MANUAL, &rdc_handle); //call library
↳directly, here we run embedded in manual mode

//Now we can use the same API for both standalone and embedded
//(1) create group
rdc_gpu_group_t groupId;
result = rdc_group_gpu_create(rdc_handle, RDC_GROUP_EMPTY, "MyGroup1", &groupId);

//(2) Add the GPUs to the group
result = rdc_group_gpu_add(rdc_handle, groupId, 0); //Add GPU 0
result = rdc_group_gpu_add(rdc_handle, groupId, 1); //Add GPU 1

//(3) start the recording the Slurm job 123. Set the sample frequency to once per second
result = rdc_job_start_stats(rdc_handle, group_id,
    "123", 1000000);

//For standalone mode, the daemon will update and cache the samples
//In manual mode, we must call rdc_field_update_all periodically to take samples
if (!standalone) { //embedded manual mode
    for (int i=5; i>0; i--) { //As an example, we will take 5 samples
        result = rdc_field_update_all(rdc_handle, 0);
        usleep(1000000);
    }
} else { //standalone mode, do nothing
    usleep(5000000); //sleep 5 seconds before fetch the stats
}

//(4) stop the Slurm job 123, which will stop the watch
// Note: we do not have to stop the job to get stats. The rdc_job_get_stats can be
↳called at any time before stop
result = rdc_job_stop_stats(rdc_handle, "123");

//(5) Get the stats
rdc_job_info_t job_info;
result = rdc_job_get_stats(rdc_handle, "123", &job_info);
std::cout<<"Average Memory Utilization: " <<job_info.summary.memoryUtilization.average <
↳<std::endl;

//The cleanup and shutdown ....

```


API REFERENCE

struct **rdc_component_version_t**

#include <rdc.h> Store version information for each component.

Public Members

char **version**[RDC_MAX_VERSION_STR_LENGTH]

struct **rdc_device_attributes_t**

#include <rdc.h> Represents attributes corresponding to a device.

Public Members

char **device_name**[RDC_MAX_STR_LENGTH]

Name of the device.

struct **rdc_diag_detail_t**

#include <rdc.h> details of the diagnostic errors

Public Members

char **msg**[MAX_DIAG_MSG_LENGTH]

The test result details.

uint32_t **code**

The low level error code.

struct **rdc_diag_per_gpu_result_t**

#include <rdc.h> details of the per gpu diagnostic results

Public Members

uint32_t **gpu_index**

The GPU index.

rdc_diag_detail_t **gpu_result**

The detail results.

struct **rdc_diag_response_t**

#include <rdc.h> The diagnostic responses for test cases.

Public Members

uint32_t **results_count**

rdc_diag_test_result_t **diag_info**[MAX_TEST_CASES]

struct **rdc_diag_test_result_t**

#include <rdc.h> The diagnostic results for all GPUs.

Public Members

rdc_diag_result_t **status**

The diagnostic result.

rdc_diag_detail_t **details**

The summary details.

rdc_diag_test_cases_t **test_case**

The test case to run.

uint32_t **per_gpu_result_count**

Result details.

How many gpu_results

rdc_diag_per_gpu_result_t **gpu_results**[RDC_MAX_NUM_DEVICES]

char **info**[MAX_DIAG_MSG_LENGTH]

Detail information.

struct **rdc_field_group_info_t**

#include <rdc.h> The structure to store the field group info.

Public Members

uint32_t **count**

count of fields in the group

char **group_name**[RDC_MAX_STR_LENGTH]

field group name

rdc_field_t **field_ids**[RDC_MAX_FIELD_IDS_PER_FIELD_GROUP]

The list of fields in the group

struct **rdc_field_value**

#include <rdc.h> The structure to store the field value.

Public Members

rdc_field_t **field_id**

The field id of the value.

int **status**

RDC_ST_OK or error status.

uint64_t **ts**

Timestamp in usec since 1970.

rdc_field_type_t **type**

The field type.

rdc_field_value_data **value**

Value of the field. Value type depends on the field type.

union **rdc_field_value_data**

#include <rdc.h> Field value data.

Public Members

int64_t **l_int**

double **dbl**

char **str**[RDC_MAX_STR_LENGTH]

struct **rdc_gpu_usage_info_t**

#include <rdc.h> The structure to hold the GPU usage information.

Public Members

uint32_t **gpu_id**

GPU_ID_INVALID for summary information.

uint64_t **start_time**

The time to start the watching.

uint64_t **end_time**

The time to stop the watching.

uint64_t **energy_consumed**

GPU Energy consumed.

uint64_t **ecc_correct**

Correctable errors.

uint64_t **ecc_uncorrect**

Uncorrectable errors.

rdc_stats_summary_t **pcie_tx**

Bytes sent over PCIe stats.

rdc_stats_summary_t **pcie_rx**

Bytes received over PCIe stats.

rdc_stats_summary_t **power_usage**

GPU Power usage stats.

rdc_stats_summary_t **gpu_clock**

GPU Clock speed stats.

rdc_stats_summary_t **memory_clock**

Mem. Clock speed stats.

rdc_stats_summary_t **gpu_utilization**

GPU Utilization stats.

rdc_stats_summary_t **gpu_temperature**

GPU temperature stats.

uint64_t **max_gpu_memory_used**

Maximum GPU memory used.

rdc_stats_summary_t **memory_utilization**

Memory Utilization statistics.

struct **rdc_group_info_t**

#include <rdc.h> The structure to store the group info.

Public Members

unsigned int **count**

count of GPUs in the group

char **group_name**[RDC_MAX_STR_LENGTH]

group name

uint32_t **entity_ids**[RDC_GROUP_MAX_ENTITIES]

The list of entities in the group

struct **rdc_job_group_info_t**

#include <rdc.h> The structure to store the job info.

Public Members

char **job_id**[RDC_MAX_STR_LENGTH]

job id

rdc_gpu_group_t **group_id**

group name

uint64_t **start_time**

job start time

uint64_t **stop_time**

job stop time

struct **rdc_job_info_t**

#include <rdc.h> The structure to hold the job stats.

Public Members

uint32_t **num_gpus**

Number of GPUs used by job.

rdc_gpu_usage_info_t **summary**

Job usage summary statistics (overall)

rdc_gpu_usage_info_t **gpus**[16]

Job usage summary statistics by GPU.

struct **rdc_stats_summary_t**

#include <rdc.h> The structure to store summary of data.

Public Members

uint64_t **max_value**

Maximum value measured.

uint64_t **min_value**

Minimum value measured.

uint64_t **average**

Average value measured.

double **standard_deviation**

The standard deviation.

namespace **std**

STL namespace.

file **rdc.h**

*#include <stddef.h>**#include <stdint.h>* The rocm_rdc library api is new, and therefore subject to change either at the ABI or API level. Instead of marking every function prototype as “unstable”, we are instead saying the API is unstable (i.e., changes are possible) while the major version remains 0. This means that if the API/ABI changes, we will not increment the major version to 1. Once the ABI stabilizes, we will increment the major version to 1, and thereafter increment it on all ABI breaks.

Main header file for the ROCm RDC library. All required function, structure, enum, etc. definitions should be defined in this file.

Defines

GPU_ID_INVALID

ID used to represent an invalid GPU.

RDC_GROUP_ALL_GPUS

Used to specify all GPUs.

RDC_JOB_STATS_FIELDS

Used to specify all stats fields.

RDC_MAX_STR_LENGTH

The max rdc field string length.

RDC_GROUP_MAX_ENTITIES

The max entities in a group.

RDC_MAX_NUM_DEVICES

Max number of GPUs supported by RDC.

RDC_MAX_FIELD_IDS_PER_FIELD_GROUP

The max fields in a field group.

RDC_MAX_NUM_GROUPS

The max number of groups.

RDC_MAX_NUM_FIELD_GROUPS

The max number of the field groups.

RDC_MAX_VERSION_STR_LENGTH

The max string length occupied by version information.

RDC_EVNT_IS_NOTIF_FIELD(FIELD)**MAX_TEST_CASES**

The maximum test cases to run.

MAX_DIAG_MSG_LENGTH

The maximum length of the diagnostic messages.

Typedefs

```
typedef void *rdc_handle_t
```

handlers used in various rdc calls

Handle used for an RDC session

```
typedef uint32_t rdc_gpu_group_t
```

GPU Group ID type.

```
typedef uint32_t rdc_field_grp_t
```

Field group ID type.

Enums

```
enum rdc_status_t
```

Error codes returned by rocm_rdc_lib functions.

Values:

```
enumerator RDC_ST_OK
```

Success.

```
enumerator RDC_ST_NOT_SUPPORTED
```

Not supported feature.

enumerator **RDC_ST_MSI_ERROR**

The MSI library error.

enumerator **RDC_ST_FAIL_LOAD_MODULE**

Fail to load the library.

enumerator **RDC_ST_INVALID_HANDLER**

Invalid handler.

enumerator **RDC_ST_BAD_PARAMETER**

A parameter is invalid.

enumerator **RDC_ST_NOT_FOUND**

Cannot find the value.

enumerator **RDC_ST_CONFLICT**

Conflict with current state.

enumerator **RDC_ST_CLIENT_ERROR**

The RDC client error.

enumerator **RDC_ST_ALREADY_EXIST**

The item already exists.

enumerator **RDC_ST_MAX_LIMIT**

Max limit recording for the object.

enumerator **RDC_ST_INSUFF_RESOURCES**

Not enough resources to complete operation

enumerator **RDC_ST_FILE_ERROR**

Failed to access a file.

enumerator **RDC_ST_NO_DATA**

Data was requested, but none was found

enumerator **RDC_ST_PERM_ERROR**

Insufficient permission to complete operation

enumerator **RDC_ST_DISABLED_MODULE**

Attempted loading disabled module.

enumerator **RDC_ST_UNKNOWN_ERROR**

Unknown error.

enum rdc_operation_mode_t

rdc operation mode rdc can run in auto mode where background threads will collect metrics. When run in manual mode, the user needs to periodically call `rdc_field_update_all` for data collection.

Values:

enumerator **RDC_OPERATION_MODE_AUTO**

enumerator **RDC_OPERATION_MODE_MANUAL**

enum rdc_group_type_t

type of GPU group

Values:

enumerator **RDC_GROUP_DEFAULT**

All GPUs on the Node.

enumerator **RDC_GROUP_EMPTY**

Empty group.

enum rdc_field_type_t

the type stored in the filed value

Values:

enumerator **INTEGER**

enumerator **DOUBLE**

enumerator **STRING**

enumerator **BLOB**

enum rdc_field_t

These enums are used to specify a particular field to be retrieved.

Values:

enumerator **RDC_FI_INVALID**

Identifier fields.

Invalid field value

enumerator **RDC_FI_GPU_COUNT**

GPU count in the system.

enumerator **RDC_FI_DEV_NAME**

Name of the device.

enumerator **RDC_FI_OAM_ID**

OAM ID of the device.

enumerator **RDC_FI_GPU_CLOCK**

Frequency related fields.

The current clock for the GPU

enumerator **RDC_FI_MEM_CLOCK**

Clock for the memory.

enumerator **RDC_FI_MEMORY_TEMP**

Memory temperature for the device.

enumerator **RDC_FI_GPU_TEMP**

Current temperature for the device.

enumerator **RDC_FI_POWER_USAGE**

Power usage for the device.

enumerator **RDC_FI_PCIE_TX**

PCIe related fields.

PCIe Tx utilization information

enumerator **RDC_FI_PCIE_RX**

PCIe Rx utilization information.

enumerator **RDC_FI_PCIE_BANDWIDTH**

PCIe bandwidth in GB/sec.

enumerator **RDC_FI_GPU_UTIL**

GPU usage related fields.

GPU Utilization

enumerator **RDC_FI_GPU_MEMORY_USAGE**

Memory usage of the GPU instance.

enumerator **RDC_FI_GPU_MEMORY_TOTAL**

Total memory of the GPU instance.

enumerator **RDC_FI_ECC_CORRECT_TOTAL**

ECC related fields.

Accumulated correctable ECC errors

enumerator **RDC_FI_ECC_UNCORRECT_TOTAL**

Accumulated uncorrectable ECC errors.

enumerator **RDC_FI_ECC_FIRST**

FIRST Error Correction and Detection field.

enumerator **RDC_FI_ECC_SDMA_CE**

enumerator **RDC_FI_ECC_SDMA_UE**

enumerator **RDC_FI_ECC_GFX_CE**

enumerator **RDC_FI_ECC_GFX_UE**

enumerator **RDC_FI_ECC_MMHUB_CE**

enumerator **RDC_FI_ECC_MMHUB_UE**

enumerator **RDC_FI_ECC_ATHUB_CE**

enumerator **RDC_FI_ECC_ATHUB_UE**

enumerator **RDC_FI_ECC_PCIE_BIF_CE**

enumerator **RDC_FI_ECC_PCIE_BIF_UE**

enumerator **RDC_FI_ECC_HDP_CE**

enumerator **RDC_FI_ECC_HDP_UE**

enumerator **RDC_FI_ECC_XGMI_WAFL_CE**

enumerator **RDC_FI_ECC_XGMI_WAFL_UE**

enumerator **RDC_FI_ECC_DF_CE**

enumerator **RDC_FI_ECC_DF_UE**

enumerator **RDC_FI_ECC_SMN_CE**

enumerator **RDC_FI_ECC_SMN_UE**

enumerator `RDC_FI_ECC_SEM_CE`

enumerator `RDC_FI_ECC_SEM_UE`

enumerator `RDC_FI_ECC_MP0_CE`

enumerator `RDC_FI_ECC_MP0_UE`

enumerator `RDC_FI_ECC_MP1_CE`

enumerator `RDC_FI_ECC_MP1_UE`

enumerator `RDC_FI_ECC_FUSE_CE`

enumerator `RDC_FI_ECC_FUSE_UE`

enumerator `RDC_FI_ECC_UMC_CE`

enumerator `RDC_FI_ECC_UMC_UE`

enumerator `RDC_FI_ECC_MCA_CE`

enumerator `RDC_FI_ECC_MCA_UE`

enumerator `RDC_FI_ECC_VCN_CE`

enumerator `RDC_FI_ECC_VCN_UE`

enumerator `RDC_FI_ECC_JPEG_CE`

enumerator `RDC_FI_ECC_JPEG_UE`

enumerator `RDC_FI_ECC_IH_CE`

enumerator `RDC_FI_ECC_IH_UE`

enumerator `RDC_FI_ECC_MPIO_CE`

enumerator `RDC_FI_ECC_MPIO_UE`

enumerator `RDC_FI_ECC_LAST`

enumerator **RDC_FI_XGMI_0_READ_KB**
XGMI_0 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_1_READ_KB**
XGMI_1 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_2_READ_KB**
XGMI_2 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_3_READ_KB**
XGMI_3 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_4_READ_KB**
XGMI_4 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_5_READ_KB**
XGMI_5 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_6_READ_KB**
XGMI_6 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_7_READ_KB**
XGMI_7 accumulated data read size (KB)

enumerator **RDC_FI_XGMI_0_WRITE_KB**
XGMI_0 accumulated data write size (KB)

enumerator **RDC_FI_XGMI_1_WRITE_KB**
XGMI_1 accumulated data write size (KB)

enumerator **RDC_FI_XGMI_2_WRITE_KB**
XGMI_2 accumulated data write size (KB)

enumerator **RDC_FI_XGMI_3_WRITE_KB**
XGMI_3 accumulated data write size (KB)

enumerator **RDC_FI_XGMI_4_WRITE_KB**
XGMI_4 accumulated data write size (KB)

enumerator **RDC_FI_XGMI_5_WRITE_KB**
XGMI_5 accumulated data write size (KB)

enumerator **RDC_FI_XGMI_6_WRITE_KB**
XGMI_6 accumulated data write size (KB)

- enumerator **RDC_FI_XGMI_7_WRITE_KB**
XGMI_7 accumulated data write size (KB)
- enumerator **RDC_FI_XGMI_TOTAL_READ_KB**
XGMI_SUM accumulated data read size (KB)
- enumerator **RDC_FI_XGMI_TOTAL_WRITE_KB**
XGMI_SUM accumulated data write size (KB)
- enumerator **RDC_FI_PROF_OCCUPANCY_PERCENT**
ROC-profiler related fields.
- enumerator **RDC_FI_PROF_ACTIVE_CYCLES**
- enumerator **RDC_FI_PROF_ACTIVE_WAVES**
- enumerator **RDC_FI_PROF_ELAPSED_CYCLES**
- enumerator **RDC_FI_PROF_TENSOR_ACTIVE_PERCENT**
- enumerator **RDC_FI_PROF_GPU_UTIL_PERCENT**
- enumerator **RDC_FI_PROF_EVAL_MEM_R_BW**
- enumerator **RDC_FI_PROF_EVAL_MEM_W_BW**
- enumerator **RDC_FI_PROF_EVAL_FLOPS_16**
- enumerator **RDC_FI_PROF_EVAL_FLOPS_32**
- enumerator **RDC_FI_PROF_EVAL_FLOPS_64**
- enumerator **RDC_FI_PROF_SM_ACTIVE**
- enumerator **RDC_EVNT_XGMI_0_NOP_TX**
Raw XGMI counter events.
NOPs sent to neighbor 0
- enumerator **RDC_EVNT_XGMI_0_REQ_TX**
Outgoing requests to neighbor 0
- enumerator **RDC_EVNT_XGMI_0_RESP_TX**
Outgoing responses to neighbor 0

enumerator **RDC_EVNT_XGMI_0_BEATS_TX**

Data beats sent to neighbor 0; Each beat represents 32 bytes.

XGMI throughput can be calculated by multiplying a BEATS event such as *RDC_EVNT_XGMI_0_BEATS_TX* by 32 and dividing by the time for which event collection occurred, *rdc_gpu_usage_info_1.start_time* (which is in nanoseconds). To get bytes per second, multiply this value by 10^9 .

Throughput = BEATS/time_running * 10^9 (bytes/second)

enumerator **RDC_EVNT_XGMI_1_NOP_TX**

NOPs sent to neighbor 1.

enumerator **RDC_EVNT_XGMI_1_REQ_TX**

Outgoing requests to neighbor 1

enumerator **RDC_EVNT_XGMI_1_RESP_TX**

Outgoing responses to neighbor 1

enumerator **RDC_EVNT_XGMI_1_BEATS_TX**

Data beats sent to neighbor 1; Each beat represents 32 bytes

enumerator **RDC_EVNT_XGMI_0_THRPUT**

Transmit throughput to XGMI neighbor 0 in bytes/sec

enumerator **RDC_EVNT_XGMI_1_THRPUT**

Transmit throughput to XGMI neighbor 1 in bytes/sec

enumerator **RDC_EVNT_XGMI_2_THRPUT**

Transmit throughput to XGMI neighbor 2 in bytes/sec

enumerator **RDC_EVNT_XGMI_3_THRPUT**

Transmit throughput to XGMI neighbor 3 in bytes/sec

enumerator **RDC_EVNT_XGMI_4_THRPUT**

Transmit throughput to XGMI neighbor 4 in bytes/sec

enumerator **RDC_EVNT_XGMI_5_THRPUT**

Transmit throughput to XGMI neighbor 5 in bytes/sec

enumerator **RDC_EVNT_NOTIF_VMFault**

VM page fault.

enumerator **RDC_EVNT_NOTIF_FIRST**

enumerator **RDC_EVNT_NOTIF_THERMAL_THROTTLE**

Clock frequency has decreased due to temperature rise

enumerator **RDC_EVNT_NOTIF_PRE_RESET**

GPU reset is about to occur.

enumerator **RDC_EVNT_NOTIF_POST_RESET**

GPU reset just occurred.

enumerator **RDC_EVNT_NOTIF_RING_HANG**

GPU ring hang just occurred.

enumerator **RDC_EVNT_NOTIF_LAST**

enumerator **RDC_HEALTH_POWER_THROTTLE_TIME**

Power throttle status counter.

enumerator **RDC_HEALTH_THERMAL_THROTTLE_TIME**

Total time in thermal throttle status (microseconds)

enum **rdc_diag_level_t**

type of diagnostic level

Values:

enumerator **RDC_DIAG_LVL_INVALID**

invalid level

enumerator **RDC_DIAG_LVL_SHORT**

take a few seconds to run

enumerator **RDC_DIAG_LVL_MED**

take less than 2 minutes to run

enumerator **RDC_DIAG_LVL_LONG**

take up to 15 minutes to run

enum **rdc_diag_result_t**

type of diagnostic result

Values:

enumerator **RDC_DIAG_RESULT_PASS**

The diagnostic test pass.

enumerator **RDC_DIAG_RESULT_SKIP**

The diagnostic test skipped.

enumerator **RDC_DIAG_RESULT_WARN**

The diagnostic test has warnings.

enumerator **RDC_DIAG_RESULT_FAIL**

The diagnostic test fail.

enum **rdc_diag_test_cases_t**

The test cases to run.

Values:

enumerator **RDC_DIAG_TEST_FIRST**

The diagnostic test pass.

enumerator **RDC_DIAG_COMPUTE_PROCESS**

enumerator **RDC_DIAG_COMPUTE_QUEUE**

The Compute Queue is ready.

enumerator **RDC_DIAG_SYS_MEM_CHECK**

Check System memory.

enumerator **RDC_DIAG_NODE_TOPOLOGY**

Report node topology.

enumerator **RDC_DIAG_RVS_TEST**

TODO: Replace with real RVS tests.

enumerator **RDC_DIAG_GPU_PARAMETERS**

GPU parameters in range.

enumerator **RDC_DIAG_TEST_LAST**

enum **rdc_component_t**

Type of Components.

Values:

enumerator **RDC_AMDMSI_COMPONENT**

Functions

rdc_status_t **rdc_init**(uint64_t init_flags)

Initialize ROCm RDC.

When called, this initializes internal data structures, including those corresponding to sources of information that RDC provides. This must be called before *rdc_start_embedded()* or *rdc_connect()*

Parameters

init_flags – [in] init_flags Bit flags that tell RDC how to initialize.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_shutdown**()

Shutdown ROCm RDC.

Do any necessary clean up.

rdc_status_t **rdc_start_embedded**(*rdc_operation_mode_t* op_mode, *rdc_handle_t* *p_rdc_handle)

Start embedded RDC agent within this process.

The RDC is loaded as library so that it does not require rdc daemon. In this mode, the user has to periodically call *rdc_field_update_all()* when op_mode is RDC_OPERATION_MODE_MANUAL, which tells RDC to collect the stats.

Parameters

- **op_mode** – [in] Operation modes. When RDC_OPERATION_MODE_AUTO, RDC schedules background task to collect the stats. When RDC_OPERATION_MODE_MANUAL, the user needs to call *rdc_field_update_all()* periodically.
- **p_rdc_handle** – [inout] Caller provided pointer to *rdc_handle_t*. Upon successful call, the value will contain the handler for following API calls.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_stop_embedded**(*rdc_handle_t* p_rdc_handle)

Stop embedded RDC agent.

Stop the embedded RDC agent, and p_rdc_handle becomes invalid after this call.

Parameters

p_rdc_handle – [in] The RDC handler that come from *rdc_start_embedded()*.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_connect**(const char *ipAndPort, *rdc_handle_t* *p_rdc_handle, const char *root_ca, const char *client_cert, const char *client_key)

Connect to rdc daemon.

This method is used to connect to a remote stand-alone rdc daemon.

Parameters

- **ipAndPort** – [in] The IP and port of the remote rdc. The ipAndPort can be specified in this x.x.x.x:yyyy format, where x.x.x.x is the IP address and yyyy is the port.
- **p_rdc_handle** – [inout] Caller provided pointer to *rdc_handle_t*. Upon successful call, the value will contain the handler for following API calls.
- **root_ca** – [in] The root CA stored in the string in pem format. Set it as nullptr if the communication is not encrypted.
- **client_cert** – [in] The client certificate stored in the string in pem format. Set it as nullptr if the communication is not encrypted.
- **client_key** – [in] The client key stored in the string in pem format. Set it as nullptr if the communication is not encrypted.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_disconnect**(*rdc_handle_t* p_rdc_handle)

Disconnect from rdc daemon.

Disconnect from rdc daemon, and p_rdc_handle becomes invalid after this call.

Parameters

p_rdc_handle – [in] The RDC handler that come from *rdc_connect*().

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_job_start_stats**(*rdc_handle_t* p_rdc_handle, *rdc_gpu_group_t* group_id, const char job_id[64], *uint64_t* update_freq)

Request the RDC to watch the job stats.

This should be executed as part of job prologue. The summary job stats can be retrieved using *rdc_job_get_stats*(). In RDC_OPERATION_MODE_MANUAL, user must call *rdc_field_update_all*(1) at least once, before call *rdc_job_get_stats*()

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **group_id** – [in] The group of GPUs to be watched.
- **job_id** – [in] The name of the job.
- **update_freq** – [in] How often to update this field in usec.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_job_get_stats**(*rdc_handle_t* p_rdc_handle, const char job_id[64], *rdc_job_info_t* *p_job_info)

Get the stats of the job using the job id.

The stats can be retrieved at any point when the job is in process.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **job_id** – [in] The name of the job.
- **p_job_info** – [inout] Caller provided pointer to *rdc_job_info_t*. Upon successful call, the value will contain the stats of the job.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_job_stop_stats**(*rdc_handle_t* p_rdc_handle, const char job_id[64])

Request RDC to stop watching the stats of the job.

This should be execute as part of job epilogue. The job Id remains available to view the stats at any point. You must call *rdc_watch_job_fields*() before this call.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **job_id** – [in] The name of the job.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_job_remove**(*rdc_handle_t* p_rdc_handle, const char job_id[64])

Request RDC to stop tracking the job given by job_id.

After this call, you will no longer be able to call *rdc_job_get_stats()* on this job_id. But you will be able to reuse the job_id after this call.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **job_id** – [in] The name of the job.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_job_remove_all**(*rdc_handle_t* p_rdc_handle)

Request RDC to stop tracking all the jobs.

After this call, you will no longer be able to call *rdc_job_get_stats()* on any job id. But you will be able to reuse the any previous used job id after this call.

Parameters

p_rdc_handle – [in] The RDC handler.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_field_update_all**(*rdc_handle_t* p_rdc_handle, uint32_t wait_for_update)

Request RDC to update all fields to be watched.

In RDC_OPERATION_MODE_MANUAL, the user must call this method periodically.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **wait_for_update** – [in] Whether or not to wait for the update loop to complete before returning to the caller 1=wait. 0=do not wait.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_device_get_all**(*rdc_handle_t* p_rdc_handle, uint32_t
gpu_index_list[RDC_MAX_NUM_DEVICES], uint32_t *count)

Get indexes corresponding to all the devices on the system.

Indexes represents RDC GPU Id corresponding to each GPU on the system and is immutable during the lifespan of the engine. The list should be queried again if the engine is restarted.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **gpu_index_list** – [out] Array reference to fill GPU indexes present on the system.
- **count** – [out] Number of GPUs returned in gpu_index_list.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_device_get_attributes**(*rdc_handle_t* p_rdc_handle, uint32_t gpu_index,
rdc_device_attributes_t *p_rdc_attr)

Gets device attributes corresponding to the gpu_index.

Fetch the attributes, such as device name, of a GPU.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **gpu_index** – [in] GPU index corresponding to which the attributes should be fetched
- **p_rdc_attr** – [out] GPU attribute corresponding to the gpu_index.

Return values

RDC_ST_OK – is returned upon successful call.

```
rdc_status_t rdc_device_get_component_version(rdc_handle_t p_rdc_handle, rdc_component_t
                                             component, rdc_component_version_t
                                             *p_rdc_compv)
```

Get version information of components used by rdc.

Given a component type, return its version information.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **component** – [in] Type of Components. See *rdc_component_t* definition for details.
- **p_rdc_compv** – [out] Version information of the corresponding component.

Return values

RDC_ST_OK – is returned upon successful call.

```
rdc_status_t rdc_group_gpu_create(rdc_handle_t p_rdc_handle, rdc_group_type_t type, const char
                                   *group_name, rdc_gpu_group_t *p_rdc_group_id)
```

Create a group contains multiple GPUs.

This method can create a group contains multiple GPUs. Instead of executing an operation separately for each GPU, the RDC group enables the user to execute same operation on all the GPUs present in the group as a single API call.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **type** – [in] The type of the group. **RDC_GROUP_DEFAULT** includes all the GPUs on the node, and **RDC_GROUP_EMPTY** creates an empty group.
- **group_name** – [in] The group name specified as NULL terminated C String
- **p_rdc_group_id** – [inout] Caller provided pointer to *rdc_gpu_group_t*. Upon successful call, the value will contain the group id for following group API calls.

Return values

RDC_ST_OK – is returned upon successful call.

```
rdc_status_t rdc_group_gpu_add(rdc_handle_t p_rdc_handle, rdc_gpu_group_t group_id, uint32_t
                               gpu_index)
```

Add a GPU to the group.

This method can add a GPU to the group

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **group_id** – [in] The group id to which the GPU will be added.
- **gpu_index** – [in] The GPU index to be added to the group.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_gpu_get_info**(*rdc_handle_t* p_rdc_handle, *rdc_gpu_group_t* p_rdc_group_id, *rdc_group_info_t* *p_rdc_group_info)

Get information about a GPU group.

Get detail information about a GPU group created by `rdc_group_gpu_create`

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **p_rdc_group_id** – [in] The GPU group handler created by `rdc_group_gpu_create`
- **p_rdc_group_info** – [out] The information of the GPU group `p_rdc_group_id`.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_get_all_ids**(*rdc_handle_t* p_rdc_handle, *rdc_gpu_group_t* group_id_list[], *uint32_t* *count)

Used to get information about all GPU groups in the system.

Get the list of GPU group ids in the system.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **group_id_list** – [out] Array reference to fill GPU group ids in the system.
- **count** – [out] Number of GPU group returned in `group_id_list`.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_gpu_destroy**(*rdc_handle_t* p_rdc_handle, *rdc_gpu_group_t* p_rdc_group_id)

Destroy GPU group represented by `p_rdc_group_id`.

Delete the logic group represented by `p_rdc_group_id`

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **p_rdc_group_id** – [in] The group id

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_field_create**(*rdc_handle_t* p_rdc_handle, *uint32_t* num_field_ids, *rdc_field_t* *field_ids, const char *field_group_name, *rdc_field_grp_t* *rdc_field_group_id)

create a group of fields

The user can create a group of fields and perform an operation on a group of fields at once.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **num_field_ids** – [in] Number of field IDs that are being provided in `field_ids`.
- **field_ids** – [in] Field IDs to be added to the newly-created field group.

- **field_group_name** – [in] Unique name for this group of fields.
- **rdc_field_group_id** – [out] Handle to the newly-created field group

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_field_get_info**(*rdc_handle_t* p_rdc_handle, *rdc_field_grp_t* rdc_field_group_id, *rdc_field_group_info_t* *field_group_info)

Get information about a field group.

Get detail information about a field group created by `rdc_group_field_create`

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **rdc_field_group_id** – [in] The field group handler created by `rdc_group_field_create`
- **field_group_info** – [out] The information of the field group `rdc_field_group_id`.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_field_get_all_ids**(*rdc_handle_t* p_rdc_handle, *rdc_field_grp_t* field_group_id_list[], *uint32_t* *count)

Used to get information about all field groups in the system.

Get the list of field group ids in the system.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **field_group_id_list** – [out] Array reference to fill field group ids in the system.
- **count** – [out] Number of field group returned in `field_group_id_list`.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_group_field_destroy**(*rdc_handle_t* p_rdc_handle, *rdc_field_grp_t* rdc_field_group_id)

Destroy field group represented by `rdc_field_group_id`.

Delete the logic group represented by `rdc_field_group_id`

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **rdc_field_group_id** – [in] The field group id

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_field_watch**(*rdc_handle_t* p_rdc_handle, *rdc_gpu_group_t* group_id, *rdc_field_grp_t* field_group_id, *uint64_t* update_freq, double max_keep_age, *uint32_t* max_keep_samples)

Request the RDC start recording updates for a given field collection.

Note that the first update of the field will not occur until the next field update cycle. To force a field update cycle, user must call `rdc_field_update_all(1)`

Parameters

- **p_rdc_handle** – [in] The RDC handler.

- **group_id** – [in] The group of GPUs to be watched.
- **field_group_id** – [in] The collection of fields to record
- **update_freq** – [in] How often to update fields in usec.
- **max_keep_age** – [in] How long to keep data for fields in seconds.
- **max_keep_samples** – [in] Maximum number of samples to keep. 0=no limit.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_field_get_latest_value**(*rdc_handle_t* p_rdc_handle, uint32_t gpu_index, *rdc_field_t* field, *rdc_field_value* *value)

Request a latest cached field of a GPU.

Note that the field can be cached after called `rdc_field_watch`

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **gpu_index** – [in] The GPU index.
- **field** – [in] The field id
- **value** – [out] The field value got from cache.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_field_get_value_since**(*rdc_handle_t* p_rdc_handle, uint32_t gpu_index, *rdc_field_t* field, uint64_t since_time_stamp, uint64_t *next_since_time_stamp, *rdc_field_value* *value)

Request a history cached field of a GPU.

Note that the field can be cached after called `rdc_field_watch`

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **gpu_index** – [in] The GPU index.
- **field** – [in] The field id
- **since_time_stamp** – [in] Timestamp to request values since in usec since 1970.
- **next_since_time_stamp** – [out] Timestamp to use for sinceTimestamp on next call to this function
- **value** – [out] The field value got from cache.

Return values

RDC_ST_OK – is returned upon successful call.

rdc_status_t **rdc_field_unwatch**(*rdc_handle_t* p_rdc_handle, *rdc_gpu_group_t* group_id, *rdc_field_grp_t* field_group_id)

Stop record updates for a given field collection.

The cache of those fields will not be updated after this call

Parameters

- **p_rdc_handle** – [in] The RDC handler.

- **group_id** – [in] The GPU group id.
- **field_group_id** – [in] The field group id.

Return values

RDC_ST_OK – is returned upon successful call.

```
rdc_status_t rdc_diagnostic_run(rdc_handle_t p_rdc_handle, rdc_gpu_group_t group_id,
                                rdc_diag_level_t level, const char *config, size_t config_size,
                                rdc_diag_response_t *response)
```

Run the diagnostic test cases.

Run the diagnostic test cases at different levels.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **group_id** – [in] The GPU group id.
- **level** – [in] The level decides how long the test will run. The RDC_DIAG_LVL_SHORT only take a few seconds, and the the RDC_DIAG_LVL_LONG may take up to 15 minutes.
- **config** – [in] Implementation specific configuration.
- **config_size** – [in] Length of the configuration.
- **response** – [inout] The detail results of the tests run.

Return values

RDC_ST_OK – is returned upon successful call.

```
rdc_status_t rdc_test_case_run(rdc_handle_t p_rdc_handle, rdc_gpu_group_t group_id,
                                rdc_diag_test_cases_t test_case, const char *config, size_t config_size,
                                rdc_diag_test_result_t *result)
```

Run one diagnostic test case.

Run a specific diagnostic test case.

Parameters

- **p_rdc_handle** – [in] The RDC handler.
- **group_id** – [in] The GPU group id.
- **test_case** – [in] The test case to run.
- **config** – [in] Implementation specific configuration.
- **config_size** – [in] Length of the configuration.
- **result** – [inout] The results of the test.

Return values

RDC_ST_OK – is returned upon successful call.

```
const char *rdc_status_string(rdc_status_t status)
```

Get a description of a provided RDC error status.

return the string in human readable format.

Parameters

status – [in] The RDC status.

Return values

The – string to describe the RDC status.

const char ***field_id_string**(*rdc_field_t* field_id)

Get the name of a field.

return the string in human readable format.

Parameters

field_id – [in] The field id.

Return values

The – string to describe the field.

rdc_field_t **get_field_id_from_name**(const char *name)

Get the field id from name.

return the field id from field name.

Parameters

name – [in] The field name.

Return values

return – RDC_FI_INVALID if the field name is invalid.

const char ***rdc_diagnostic_result_string**(*rdc_diag_result_t* result)

Get a description of a diagnostic result.

return the string in human readable format.

Parameters

result – [in] The RDC diagnostic result.

Return values

The – string to describe the RDC diagnostic result.

dir /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rdc/checkouts/docs-6.3.3/include

dir /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rdc/checkouts/docs-6.3.3/include/rdc

LICENSE

MIT License

Copyright © 2019 - 2024 Advanced Micro Devices, Inc. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

F

field_id_string (C++ function), 63

G

get_field_id_from_name (C++ function), 64
GPU_ID_INVALID (C macro), 44

M

MAX_DIAG_MSG_LENGTH (C macro), 45
MAX_TEST_CASES (C macro), 45

R

rdc_component_t (C++ enum), 55
rdc_component_t::RDC_AMDMSI_COMPONENT (C++ enumerator), 55
rdc_component_version_t (C++ struct), 39
rdc_component_version_t::version (C++ member), 39
rdc_connect (C++ function), 56
rdc_device_attributes_t (C++ struct), 39
rdc_device_attributes_t::device_name (C++ member), 39
rdc_device_get_all (C++ function), 58
rdc_device_get_attributes (C++ function), 58
rdc_device_get_component_version (C++ function), 59
rdc_diag_detail_t (C++ struct), 39
rdc_diag_detail_t::code (C++ member), 39
rdc_diag_detail_t::msg (C++ member), 39
rdc_diag_level_t (C++ enum), 54
rdc_diag_level_t::RDC_DIAG_LVL_INVALID (C++ enumerator), 54
rdc_diag_level_t::RDC_DIAG_LVL_LONG (C++ enumerator), 54
rdc_diag_level_t::RDC_DIAG_LVL_MED (C++ enumerator), 54
rdc_diag_level_t::RDC_DIAG_LVL_SHORT (C++ enumerator), 54
rdc_diag_per_gpu_result_t (C++ struct), 39
rdc_diag_per_gpu_result_t::gpu_index (C++ member), 39
rdc_diag_per_gpu_result_t::gpu_result (C++ member), 39
rdc_diag_response_t (C++ struct), 40
rdc_diag_response_t::diag_info (C++ member), 40
rdc_diag_response_t::results_count (C++ member), 40
rdc_diag_result_t (C++ enum), 54
rdc_diag_result_t::RDC_DIAG_RESULT_FAIL (C++ enumerator), 54
rdc_diag_result_t::RDC_DIAG_RESULT_PASS (C++ enumerator), 54
rdc_diag_result_t::RDC_DIAG_RESULT_SKIP (C++ enumerator), 54
rdc_diag_result_t::RDC_DIAG_RESULT_WARN (C++ enumerator), 54
rdc_diag_test_cases_t (C++ enum), 55
rdc_diag_test_cases_t::RDC_DIAG_COMPUTE_PROCESS (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_COMPUTE_QUEUE (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_GPU_PARAMETERS (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_NODE_TOPOLOGY (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_RVS_TEST (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_SYS_MEM_CHECK (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_TEST_FIRST (C++ enumerator), 55
rdc_diag_test_cases_t::RDC_DIAG_TEST_LAST (C++ enumerator), 55
rdc_diag_test_result_t (C++ struct), 40
rdc_diag_test_result_t::details (C++ member), 40
rdc_diag_test_result_t::gpu_results (C++ member), 40
rdc_diag_test_result_t::info (C++ member), 40
rdc_diag_test_result_t::per_gpu_result_count (C++ member), 40
rdc_diag_test_result_t::status (C++ member),

- 40
- `rdc_diag_test_result_t::test_case` (C++ member), 40
 - `rdc_diagnostic_result_string` (C++ function), 64
 - `rdc_diagnostic_run` (C++ function), 63
 - `rdc_disconnect` (C++ function), 56
 - `RDC_EVNT_IS_NOTIF_FIELD` (C macro), 45
 - `rdc_field_get_latest_value` (C++ function), 62
 - `rdc_field_get_value_since` (C++ function), 62
 - `rdc_field_group_info_t` (C++ struct), 40
 - `rdc_field_group_info_t::count` (C++ member), 40
 - `rdc_field_group_info_t::field_ids` (C++ member), 40
 - `rdc_field_group_info_t::group_name` (C++ member), 40
 - `rdc_field_grp_t` (C++ type), 45
 - `rdc_field_t` (C++ enum), 47
 - `rdc_field_t::RDC_EVNT_NOTIF_FIRST` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_NOTIF_LAST` (C++ enumerator), 54
 - `rdc_field_t::RDC_EVNT_NOTIF_POST_RESET` (C++ enumerator), 54
 - `rdc_field_t::RDC_EVNT_NOTIF_PRE_RESET` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_NOTIF_RING_HANG` (C++ enumerator), 54
 - `rdc_field_t::RDC_EVNT_NOTIF_THERMAL_THROTTLE` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_NOTIF_VMFAULT` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_0_BEATS_TX` (C++ enumerator), 52
 - `rdc_field_t::RDC_EVNT_XGMI_0_NOP_TX` (C++ enumerator), 52
 - `rdc_field_t::RDC_EVNT_XGMI_0_REQ_TX` (C++ enumerator), 52
 - `rdc_field_t::RDC_EVNT_XGMI_0_RESP_TX` (C++ enumerator), 52
 - `rdc_field_t::RDC_EVNT_XGMI_0_THRPUT` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_1_BEATS_TX` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_1_NOP_TX` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_1_REQ_TX` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_1_RESP_TX` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_1_THRPUT` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_2_THRPUT` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_3_THRPUT` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_4_THRPUT` (C++ enumerator), 53
 - `rdc_field_t::RDC_EVNT_XGMI_5_THRPUT` (C++ enumerator), 53
 - `rdc_field_t::RDC_FI_DEV_NAME` (C++ enumerator), 47
 - `rdc_field_t::RDC_FI_ECC_ATHUB_CE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_ATHUB_UE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_CORRECT_TOTAL` (C++ enumerator), 48
 - `rdc_field_t::RDC_FI_ECC_DF_CE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_DF_UE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_FIRST` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_FUSE_CE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_FUSE_UE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_GFX_CE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_GFX_UE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_HDP_CE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_HDP_UE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_IH_CE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_IH_UE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_JPEG_CE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_JPEG_UE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_LAST` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_MCA_CE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_MCA_UE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_MMHUB_CE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_MMHUB_UE` (C++ enumerator), 49
 - `rdc_field_t::RDC_FI_ECC_MP0_CE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_MP0_UE` (C++ enumerator), 50
 - `rdc_field_t::RDC_FI_ECC_MP1_CE` (C++ enumerator), 50

tor), 50

rdc_field_t::RDC_FI_ECC_MP1_UE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_MPIO_CE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_MPIO_UE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_PCIE_BIF_CE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_PCIE_BIF_UE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_SDMA_CE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_SDMA_UE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_SEM_CE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_SEM_UE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_SMN_CE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_SMN_UE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_UMC_CE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_UMC_UE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_UNCORRECT_TOTAL (C++ enumerator), 48

rdc_field_t::RDC_FI_ECC_VCN_CE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_VCN_UE (C++ enumerator), 50

rdc_field_t::RDC_FI_ECC_XGMI_WAFL_CE (C++ enumerator), 49

rdc_field_t::RDC_FI_ECC_XGMI_WAFL_UE (C++ enumerator), 49

rdc_field_t::RDC_FI_GPU_CLOCK (C++ enumerator), 48

rdc_field_t::RDC_FI_GPU_COUNT (C++ enumerator), 47

rdc_field_t::RDC_FI_GPU_MEMORY_TOTAL (C++ enumerator), 48

rdc_field_t::RDC_FI_GPU_MEMORY_USAGE (C++ enumerator), 48

rdc_field_t::RDC_FI_GPU_TEMP (C++ enumerator), 48

rdc_field_t::RDC_FI_GPU_UTIL (C++ enumerator), 48

rdc_field_t::RDC_FI_INVALID (C++ enumerator), 47

rdc_field_t::RDC_FI_MEM_CLOCK (C++ enumerator), 48

rdc_field_t::RDC_FI_MEMORY_TEMP (C++ enumerator), 48

rdc_field_t::RDC_FI_OAM_ID (C++ enumerator), 48

rdc_field_t::RDC_FI_PCIE_BANDWIDTH (C++ enumerator), 48

rdc_field_t::RDC_FI_PCIE_RX (C++ enumerator), 48

rdc_field_t::RDC_FI_PCIE_TX (C++ enumerator), 48

rdc_field_t::RDC_FI_POWER_USAGE (C++ enumerator), 48

rdc_field_t::RDC_FI_PROF_ACTIVE_CYCLES (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_ACTIVE_WAVES (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_ELAPSED_CYCLES (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_EVAL_FLOPS_16 (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_EVAL_FLOPS_32 (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_EVAL_FLOPS_64 (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_EVAL_MEM_R_BW (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_EVAL_MEM_W_BW (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_GPU_UTIL_PERCENT (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_OCCUPANCY_PERCENT (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_SM_ACTIVE (C++ enumerator), 52

rdc_field_t::RDC_FI_PROF_TENSOR_ACTIVE_PERCENT (C++ enumerator), 52

rdc_field_t::RDC_FI_XGMI_0_READ_KB (C++ enumerator), 50

rdc_field_t::RDC_FI_XGMI_0_WRITE_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_1_READ_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_1_WRITE_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_2_READ_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_2_WRITE_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_3_READ_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_3_WRITE_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_4_READ_KB (C++ enumerator), 51

rdc_field_t::RDC_FI_XGMI_4_WRITE_KB (C++ enumerator), 51

`rdc_field_t::RDC_FI_XGMI_5_READ_KB` (C++ *enumerator*), 51
`rdc_field_t::RDC_FI_XGMI_5_WRITE_KB` (C++ *enumerator*), 51
`rdc_field_t::RDC_FI_XGMI_6_READ_KB` (C++ *enumerator*), 51
`rdc_field_t::RDC_FI_XGMI_6_WRITE_KB` (C++ *enumerator*), 51
`rdc_field_t::RDC_FI_XGMI_7_READ_KB` (C++ *enumerator*), 51
`rdc_field_t::RDC_FI_XGMI_7_WRITE_KB` (C++ *enumerator*), 51
`rdc_field_t::RDC_FI_XGMI_TOTAL_READ_KB` (C++ *enumerator*), 52
`rdc_field_t::RDC_FI_XGMI_TOTAL_WRITE_KB` (C++ *enumerator*), 52
`rdc_field_t::RDC_HEALTH_POWER_THROTTLE_TIME` (C++ *enumerator*), 54
`rdc_field_t::RDC_HEALTH_THERMAL_THROTTLE_TIME` (C++ *enumerator*), 54
`rdc_field_type_t` (C++ *enum*), 47
`rdc_field_type_t::BLOB` (C++ *enumerator*), 47
`rdc_field_type_t::DOUBLE` (C++ *enumerator*), 47
`rdc_field_type_t::INTEGER` (C++ *enumerator*), 47
`rdc_field_type_t::STRING` (C++ *enumerator*), 47
`rdc_field_unwatch` (C++ *function*), 62
`rdc_field_update_all` (C++ *function*), 58
`rdc_field_value` (C++ *struct*), 41
`rdc_field_value::field_id` (C++ *member*), 41
`rdc_field_value::status` (C++ *member*), 41
`rdc_field_value::ts` (C++ *member*), 41
`rdc_field_value::type` (C++ *member*), 41
`rdc_field_value::value` (C++ *member*), 41
`rdc_field_value_data` (C++ *union*), 41
`rdc_field_value_data::dbl` (C++ *member*), 41
`rdc_field_value_data::l_int` (C++ *member*), 41
`rdc_field_value_data::str` (C++ *member*), 41
`rdc_field_watch` (C++ *function*), 61
`rdc_gpu_group_t` (C++ *type*), 45
`rdc_gpu_usage_info_t` (C++ *struct*), 41
`rdc_gpu_usage_info_t::ecc_correct` (C++ *member*), 42
`rdc_gpu_usage_info_t::ecc_uncorrect` (C++ *member*), 42
`rdc_gpu_usage_info_t::end_time` (C++ *member*), 41
`rdc_gpu_usage_info_t::energy_consumed` (C++ *member*), 42
`rdc_gpu_usage_info_t::gpu_clock` (C++ *member*), 42
`rdc_gpu_usage_info_t::gpu_id` (C++ *member*), 41
`rdc_gpu_usage_info_t::gpu_temperature` (C++ *member*), 42
`rdc_gpu_usage_info_t::gpu_utilization` (C++ *member*), 42
`rdc_gpu_usage_info_t::max_gpu_memory_used` (C++ *member*), 42
`rdc_gpu_usage_info_t::memory_clock` (C++ *member*), 42
`rdc_gpu_usage_info_t::memory_utilization` (C++ *member*), 42
`rdc_gpu_usage_info_t::pcie_rx` (C++ *member*), 42
`rdc_gpu_usage_info_t::pcie_tx` (C++ *member*), 42
`rdc_gpu_usage_info_t::power_usage` (C++ *member*), 42
`rdc_gpu_usage_info_t::start_time` (C++ *member*), 41
`RDC_GROUP_ALL_GPUS` (C *macro*), 44
`rdc_group_field_create` (C++ *function*), 60
`rdc_group_field_destroy` (C++ *function*), 61
`rdc_group_field_get_all_ids` (C++ *function*), 61
`rdc_group_field_get_info` (C++ *function*), 61
`rdc_group_get_all_ids` (C++ *function*), 60
`rdc_group_gpu_add` (C++ *function*), 59
`rdc_group_gpu_create` (C++ *function*), 59
`rdc_group_gpu_destroy` (C++ *function*), 60
`rdc_group_gpu_get_info` (C++ *function*), 60
`rdc_group_info_t` (C++ *struct*), 42
`rdc_group_info_t::count` (C++ *member*), 43
`rdc_group_info_t::entity_ids` (C++ *member*), 43
`rdc_group_info_t::group_name` (C++ *member*), 43
`RDC_GROUP_MAX_ENTITIES` (C *macro*), 44
`rdc_group_type_t` (C++ *enum*), 47
`rdc_group_type_t::RDC_GROUP_DEFAULT` (C++ *enumerator*), 47
`rdc_group_type_t::RDC_GROUP_EMPTY` (C++ *enumerator*), 47
`rdc_handle_t` (C++ *type*), 45
`rdc_init` (C++ *function*), 55
`rdc_job_get_stats` (C++ *function*), 57
`rdc_job_group_info_t` (C++ *struct*), 43
`rdc_job_group_info_t::group_id` (C++ *member*), 43
`rdc_job_group_info_t::job_id` (C++ *member*), 43
`rdc_job_group_info_t::start_time` (C++ *member*), 43
`rdc_job_group_info_t::stop_time` (C++ *member*), 43
`rdc_job_info_t` (C++ *struct*), 43
`rdc_job_info_t::gpus` (C++ *member*), 43
`rdc_job_info_t::num_gpus` (C++ *member*), 43
`rdc_job_info_t::summary` (C++ *member*), 43
`rdc_job_remove` (C++ *function*), 57
`rdc_job_remove_all` (C++ *function*), 58
`rdc_job_start_stats` (C++ *function*), 57
`RDC_JOB_STATS_FIELDS` (C *macro*), 44
`rdc_job_stop_stats` (C++ *function*), 57
`RDC_MAX_FIELD_IDS_PER_FIELD_GROUP` (C *macro*), 44

RDC_MAX_NUM_DEVICES (*C macro*), 44
RDC_MAX_NUM_FIELD_GROUPS (*C macro*), 45
RDC_MAX_NUM_GROUPS (*C macro*), 45
RDC_MAX_STR_LENGTH (*C macro*), 44
RDC_MAX_VERSION_STR_LENGTH (*C macro*), 45
rdc_operation_mode_t (*C++ enum*), 46
rdc_operation_mode_t::RDC_OPERATION_MODE_AUTO
(*C++ enumerator*), 47
rdc_operation_mode_t::RDC_OPERATION_MODE_MANUAL
(*C++ enumerator*), 47
rdc_shutdown (*C++ function*), 55
rdc_start_embedded (*C++ function*), 56
rdc_stats_summary_t (*C++ struct*), 43
rdc_stats_summary_t::average (*C++ member*), 44
rdc_stats_summary_t::max_value (*C++ member*),
44
rdc_stats_summary_t::min_value (*C++ member*),
44
rdc_stats_summary_t::standard_deviation
(*C++ member*), 44
rdc_status_string (*C++ function*), 63
rdc_status_t (*C++ enum*), 45
rdc_status_t::RDC_ST_ALREADY_EXIST (*C++ enu-*
merator), 46
rdc_status_t::RDC_ST_BAD_PARAMETER (*C++ enu-*
merator), 46
rdc_status_t::RDC_ST_CLIENT_ERROR (*C++ enu-*
merator), 46
rdc_status_t::RDC_ST_CONFLICT (*C++ enumera-*
tor), 46
rdc_status_t::RDC_ST_DISABLED_MODULE (*C++*
enumerator), 46
rdc_status_t::RDC_ST_FAIL_LOAD_MODULE (*C++*
enumerator), 46
rdc_status_t::RDC_ST_FILE_ERROR (*C++ enumera-*
tor), 46
rdc_status_t::RDC_ST_INSUFF_RESOURCES (*C++*
enumerator), 46
rdc_status_t::RDC_ST_INVALID_HANDLER (*C++*
enumerator), 46
rdc_status_t::RDC_ST_MAX_LIMIT (*C++ enumera-*
tor), 46
rdc_status_t::RDC_ST_MSI_ERROR (*C++ enumera-*
tor), 45
rdc_status_t::RDC_ST_NO_DATA (*C++ enumerator*),
46
rdc_status_t::RDC_ST_NOT_FOUND (*C++ enumera-*
tor), 46
rdc_status_t::RDC_ST_NOT_SUPPORTED (*C++ enu-*
merator), 45
rdc_status_t::RDC_ST_OK (*C++ enumerator*), 45
rdc_status_t::RDC_ST_PERM_ERROR (*C++ enumera-*
tor), 46
rdc_status_t::RDC_ST_UNKNOWN_ERROR (*C++ enu-*
merator), 46
rdc_stop_embedded (*C++ function*), 56
rdc_test_case_run (*C++ function*), 63

S

std (*C++ type*), 44