Program Listing for File io_spec.hpp

Return to documentation for file (include/holoscan/core/io_spec.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2022-2023 NVIDIA CORPORATION & AFFILIATES. All rights reserved. * SPDX-License-Identifier: Apache-2.0 * * Licensed under the Apache License, Version 2.0 (the "License"); * you may not use this file except in compliance with the License. * You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License. */ #ifndef HOLOSCAN_CORE_IO_SPEC_HPP #define HOLOSCAN_CORE_IO_SPEC_HPP #include <yaml-cpp/yaml.h> #include <iostream> #include <memory> #include <stdexcept> #include <string> #include <typeinfo> #include <utility> #include <vector> #include "./condition.hpp" #include "./conditions/gxf/asynchronous.hpp" #include "./conditions/gxf/boolean.hpp" #include "./conditions/gxf/count.hpp" #include "./conditions/gxf/downstream_affordable.hpp" #include "./conditions/gxf/periodic.hpp" #include "./conditions/gxf/message_available.hpp" #include "./resources/gxf/double_buffer_receiver.hpp" #include "./resources/gxf/double_buffer_transmitter.hpp" #include "./resources/gxf/ucx_receiver.hpp" #include "./resources/gxf/ucx_transmitter.hpp" #include "./resource.hpp" #include "./gxf/entity.hpp" #include "./common.hpp" namespace holoscan { class IOSpec { public: enum class IOType { kInput, kOutput }; enum class ConnectorType { kDefault, kDoubleBuffer, kUCX }; IOSpec(OperatorSpec* op_spec, const std::string& name, IOType io_type) : op_spec_(op_spec), name_(name), io_type_(io_type), typeinfo_(&typeid(holoscan::gxf::Entity)) { *// Operator::parse_port_name requires that "." is not allowed in the IOSPec name* if (name.find(".") != std::string::npos) { throw std::invalid_argument(fmt::format("The . character is reserved and cannot be used in the port (IOSpec) name ('{}').", name)); } name_ = name; } IOSpec(OperatorSpec* op_spec, const std::string& name, IOType io_type, const std::type_info* typeinfo) : op_spec_(op_spec), io_type_(io_type), typeinfo_(typeinfo) { *// Operator::parse_port_name requires that "." is not allowed in the IOSPec name* if (name.find(".") != std::string::npos) { throw std::invalid_argument(fmt::format("The . character is reserved and cannot be used in the port (IOSpec) name ('{}').", name)); }

```
name_ = name; } OperatorSpec* op_spec() const { return op_spec_; } const
std::string& name() const { return name_; } IOType io_type() const { return io_type_; }
ConnectorType connector_type() const { return connector_type_; } const
std::type_info* typeinfo() const { return typeinfo_; }
std::vector<std::pair<ConditionType, std::shared_ptr<Condition>>>& conditions() {
return conditions_; } template <typename... ArgsT> IOSpec&
condition(ConditionType type, ArgsT&&... args) { switch (type) { case
ConditionType::kMessageAvailable: conditions_.emplace_back( type,
std::make_shared<MessageAvailableCondition>(std::forward<ArgsT>(args)...));
break; case ConditionType::kDownstreamMessageAffordable:
conditions_.emplace_back( type,
std::make_shared<DownstreamMessageAffordableCondition>(std::forward<ArgsT>
(args)...)); break; case ConditionType::kNone: conditions_.emplace_back(type,
nullptr); break; default: HOLOSCAN_LOG_ERROR("Unsupported condition type for
IOSpec: {}", static_cast<int>(type)); break; } return *this; } std::shared_ptr<Resource>
connector() const { return connector_; } void connector(std::shared_ptr<Resource>
connector) { connector_ = connector; } template <typename... ArgsT> IOSpec&
connector(ConnectorType type, ArgsT&&... args) { connector_type_ = type; switch
(type) { case ConnectorType::kDefault: // default receiver or transmitter will be created
in GXFExecutor::run instead break; case ConnectorType::kDoubleBuffer: if (io_type_
== IOType::kInput) { connector_ = std::make_shared<DoubleBufferReceiver>
(std::forward<ArgsT>(args)...); } else { connector_ =
std::make_shared<DoubleBufferTransmitter>(std::forward<ArgsT>(args)...); } break;
case ConnectorType::kUCX: if (io_type_ == IOType::kInput) { connector_ =
std::make_shared<UcxReceiver>(std::forward<ArgsT>(args)...); } else { connector_ =
std::make_shared<UcxTransmitter>(std::forward<ArgsT>(args)...); } break; default:
HOLOSCAN_LOG_ERROR("Unknown connector type {}", static_cast<int>(type));
break; } return *this; } virtual YAML::Node to_yaml_node() const; std::string
description() const; private: OperatorSpec* op_spec_ = nullptr; std::string name_;
IOType io_type_; const std::type_info* typeinfo_ = nullptr;
std::shared_ptr<Resource> connector_; std::vector<std::pair<ConditionType,
std::shared_ptr<Condition>>> conditions_; ConnectorType connector_type_ =
ConnectorType::kDefault; }; } // namespace holoscan #endif/*
HOLOSCAN_CORE_IO_SPEC_HPP */
```