Program Listing for File gxf_parameter_adaptor.hpp

Return to documentation for file (include/holoscan/core/executors/gxf/gxf_parameter_adaptor.hpp)

```
/* * SPDX-FileCopyrightText: Copyright (c) 2022-2024 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_EXECUTORS_GXF_GXF_PARAMETER_ADAPTOR_HPP #define HOLOSCAN_CORE_EXECUTORS_GXF_GXF_PARAMETER_ADAPTOR_HPP #include <functional> #include <memory> #include <string> #include <type_traits> #include <typeindex> #include <typeinfo> #include <unordered_map> #include <vector> #include "../../arg.hpp" #include "../../common.hpp" #include "../../gxf/gxf_condition.hpp" #include "../../gxf/gxf_resource.hpp" #include "../../gxf/gxf_utils.hpp" #include "../../io_spec.hpp" #include "../../parameter.hpp" namespace holoscan::gxf { class GXFParameterAdaptor { public: using AdaptFunc = std::function<gxf_result_t(gxf_context_t context, gxf_uid_t uid, const char* key, const ArgType& arg_type, const std::any& any_value)>; inline static AdaptFunc none_param_handler = [](gxf_context_t context, gxf_uid_t uid, const char* key, const ArgType& arg_type, const std::any& any_value) { (void)context; (void)uid; (void)key; (void)arg_type; (void)any_value; HOLOSCAN_LOG_ERROR("Unable to handle parameter: {}", key); return GXF_FAILURE; }; static GXFParameterAdaptor& get_instance(); static gxf_result_t set_param(gxf_context_t context, gxf_uid_t uid, const char* key, ParameterWrapper& param_wrap) { auto& instance = get_instance(); const auto index = std::type_index(param_wrap.type()); const AdaptFunc& func = instance.get_param_handler(index); if (&func == &none_param_handler) { HOLOSCAN_LOG_ERROR("Unable to handle parameter: {}", key); return GXF_FAILURE; } return func(context, uid, key, param_wrap.arg_type(), param_wrap.value()); } static gxf_result_t set_param(gxf_context_t context, gxf_uid_t uid, const char* key, const ArgType& arg_type, std::any& any_value) { auto& instance = get_instance(); const auto index =
std::type_index(any_value.type()); const AdaptFunc& func =
instance.get_arg_param_handler(index); if (&func == &none_param_handler) {
HOLOSCAN_LOG_ERROR("Unable to handle parameter: {}", key); return
GXF_FAILURE; } return func(context, uid, key, arg_type, any_value); } template
<typename typeT> static void ensure_type() { auto& instance = get_instance();
instance.add_param_handler<typeT>(); } AdaptFunc&
get_param_handler(std::type_index index) { if (function_map_.find(index) ==
function_map_.end()) { HOLOSCAN_LOG_WARN("No parameter handler for type '{}'
exists", index.name()); return GXFParameterAdaptor::none_param_handler; } auto&
handler = function_map_[index]; return handler; } AdaptFunc&
get_arg_param_handler(std::type_index index) { if (arg_function_map_.find(index) ==
arg_function_map_.end()) { HOLOSCAN_LOG_WARN("No parameter handler for type
'{}' exists", index.name()); return GXFParameterAdaptor::none_param_handler; }
auto& handler = arg_function_map_[index]; return handler; } template <typename
typeT> void add_param_handler(AdaptFunc func) {
function_map_.try_emplace(std::type_index(typeid(typeT)), func);
arg_function_map_.try_emplace(std::type_index(typeid(typeT)), func); } void
add_param_handler(std::type_index index, AdaptFunc func) {
function_map_.try_emplace(index, func); arg_function_map_.try_emplace(index,
func); } template <typename typeT> void add_param_handler() { const AdaptFunc&
func = [](gxf_context_t context, gxf_uid_t uid, const char* key, const ArgType&
arg_type, const std::any& any_value) { (void)context; // avoid `-Werror=unused-but-set-
parameter` due to `constexpr` (void)uid; // avoid `-Werror=unused-but-set-parameter`
due to `constexpr` try { auto& param = *std::any_cast<Parameter<typeT>*>
(any_value); param.set_default_value(); // set default value if not set. if
(!param.has_value()) { if (param.flag() == ParameterFlag::kOptional) { return
GXF_SUCCESS; } else { HOLOSCAN_LOG_WARN( "Unable to get argument for key '{}'
with type '{}'", key, typeid(typeT).name()); return GXF_FAILURE; } } typeT& value =
param.get(); gxf_result_t result = set_gxf_parameter_value(context, uid, key,
arg_type, value); return result; } catch (const std::bad_any_cast& e) {
HOLOSCAN_LOG_ERROR("Bad any cast exception: {}", e.what()); } return
GXF_FAILURE; }; const AdaptFunc& arg_func = [](gxf_context_t context, gxf_uid_t uid,
const char* key, const ArgType& arg_type, const std::any& any_value) {
(void)context; // avoid `-Werror=unused-but-set-parameter` due to `constexpr` (void)uid;
// avoid `-Werror=unused-but-set-parameter` due to `constexpr` try { typeT value =
std::any_cast<typeT>(any_value); gxf_result_t result =
set_gxf_parameter_value(context, uid, key, arg_type, value); return result; } catch
(const std::bad_any_cast& e) { HOLOSCAN_LOG_ERROR("Bad any cast exception: {}",
e.what()); } return GXF_FAILURE; };
function_map_.try_emplace(std::type_index(typeid(typeT)), func);
arg_function_map_.try_emplace(std::type_index(typeid(typeT)), arg_func); } template
<typename typeT> static gxf_result_t set_gxf_parameter_value(gxf_context_t
context, gxf_uid_t uid, const char* key, const ArgType& arg_type, typeT& value) {
switch (arg_type.container_type()) { case ArgContainerType::kNative: { switch
(arg_type.element_type()) { case ArgElementType::kBoolean: { if constexpr
(std::is_same_v<typeT, bool>) { return GxfParameterSetBool(context, uid, key,
value); } break; } case ArgElementType::kInt8: { if constexpr (std::is_same_v<typeT,
int8_t>) { return GxfParameterSetInt8(context, uid, key, value); } break; } case
ArgElementType::kUnsigned8: { if constexpr (std::is_same_v<typeT, uint8_t>) { return
GxfParameterSetUInt8(context, uid, key, value); } break; } case
ArgElementType::kInt16: { if constexpr (std::is_same_v<typeT, int16_t>) { return
GxfParameterSetInt16(context, uid, key, value); } break; } case
ArgElementType::kUnsigned16: { if constexpr (std::is_same_v<typeT, uint16_t>) {
return GxfParameterSetUInt16(context, uid, key, value); } break; } case
ArgElementType::kInt32: { if constexpr (std::is_same_v<typeT, int32_t>) { return
GxfParameterSetInt32(context, uid, key, value); } break; } case
ArgElementType::kUnsigned32: { if constexpr (std::is_same_v<typeT, uint32_t>) {
return GxfParameterSetUInt32(context, uid, key, value); } break; } case
ArgElementType::kInt64: { if constexpr (std::is_same_v<typeT, int64_t>) { return
GxfParameterSetInt64(context, uid, key, value); } break; } case
ArgElementType::kUnsigned64: { if constexpr (std::is_same_v<typeT, uint64_t>) {
return GxfParameterSetUInt64(context, uid, key, value); } break; } case
ArgElementType::kFloat32: { if constexpr (std::is_same_v<typeT, float>) { return
GxfParameterSetFloat32(context, uid, key, value); } break; } case
ArgElementType::kFloat64: { if constexpr (std::is_same_v<typeT, double>) { return
GxfParameterSetFloat64(context, uid, key, value); } break; } case
ArgElementType::kComplex64: { // GXF Doesn't have parameter setter for
complex<float> or complex<double> if constexpr (std::is_same_v<typeT,
std::complex<float>>) { YAML::Node yaml_node; yaml_node.push_back(value);
YAML::Node value_node = yaml_node[0]; return
GxfParameterSetFromYamlNode(context, uid, key, &value_node, ""); } break; } case
ArgElementType::kComplex128: { // GXF Doesn't have parameter setter for
```

```
complex<float> or complex<double> if constexpr (std::is_same_v<typeT,
std::complex<double>>) { YAML::Node yaml_node; yaml_node.push_back(value);
YAML::Node value_node = yaml_node[0]; return
GxfParameterSetFromYamlNode(context, uid, key, &value_node, ""); } break; } case
ArgElementType::kString: { if constexpr (std::is_same_v<typeT, std::string>) { return
GxfParameterSetStr(context, uid, key, value.c_str()); } break; } case
ArgElementType::kHandle: { HOLOSCAN_LOG_ERROR("Unable to set handle
parameter for key '{}'", key); return GXF_FAILURE; } case
ArgElementType::kYAMLNode: { if constexpr (std::is_same_v<typeT, YAML::Node>) {
return GxfParameterSetFromYamlNode(context, uid, key, &value, ""); } else {
HOLOSCAN_LOG_ERROR("Unable to handle ArgElementType::kYAMLNode for key
'{}'", key); return GXF_FAILURE; } } case ArgElementType::kIOSpec: { if constexpr
(std::is_same_v<typeT, holoscan::IOSpec*>) { if (value) { auto gxf_resource =
std::dynamic_pointer_cast<GXFResource>(value->connector()); gxf_uid_t cid =
gxf_resource->gxf_cid(); return GxfParameterSetHandle(context, uid, key, cid); } else
{ // If the IOSpec is null, do not set the parameter. return GXF_SUCCESS; } } break; }
case ArgElementType::kResource: { if constexpr (std::is_same_v<typename
holoscan::type_info<typeT>::element_type, std::shared_ptr<Resource>> &&
holoscan::type_info<typeT>::dimension == 0) { auto gxf_resource =
std::dynamic_pointer_cast<GXFResource>(value); if (gxf_resource) { // Initialize GXF
component if it is not already initialized. if (gxf_resource->gxf_context() == nullptr) {
gxf_resource->gxf_eid( gxf::get_component_eid(context, uid)); // set Entity ID of the
component gxf_resource->initialize(); } return GxfParameterSetHandle(context, uid,
key, gxf_resource->gxf_cid()); } else { HOLOSCAN_LOG_TRACE("Resource is null for
key '{}'. Not setting parameter.", key); return GXF_SUCCESS; } }
HOLOSCAN_LOG_ERROR("Unable to handle ArgElementType::kResource for key '{}'",
key); break; } case ArgElementType::kCondition: { if constexpr
(std::is_same_v<typename holoscan::type_info<typeT>::element_type,
std::shared_ptr<Condition>> && holoscan::type_info<typeT>::dimension == 0) { auto
gxf_condition = std::dynamic_pointer_cast<GXFCondition>(value); if (gxf_condition) {
// Initialize GXF component if it is not already initialized. if (gxf_condition->gxf_context()
== nullptr) { gxf_condition->gxf_eid( gxf::get_component_eid(context, uid)); // set
Entity ID of the component gxf_condition->initialize(); } return
GxfParameterSetHandle(context, uid, key, gxf_condition->gxf_cid()); }
HOLOSCAN_LOG_ERROR("Unable to handle ArgElementType::kCondition for key
'{}'", key); } break; } case ArgElementType::kCustom: {
```

```
HOLOSCAN_LOG_ERROR("Unable to handle ArgElementType::kCustom for key '{}'",
key); return GXF_FAILURE; } } break; } case ArgContainerType::kVector: { switch
(arg_type.element_type()) { case ArgElementType::kBoolean: case
ArgElementType::kInt8: case ArgElementType::kUnsigned8: case
ArgElementType::kInt16: case ArgElementType::kUnsigned16: case
ArgElementType::kInt32: case ArgElementType::kUnsigned32: case
ArgElementType::kInt64: case ArgElementType::kUnsigned64: case
ArgElementType::kFloat32: case ArgElementType::kFloat64: case
ArgElementType::kComplex64: case ArgElementType::kComplex128: case
ArgElementType::kString: { // GXF Doesn't support std::vector<T> or
std::vector<std::vector<T>> parameter // types so use a workaround with
GxfParameterSetFromYamlNode. if constexpr (holoscan::is_one_of_v<typename
holoscan::type_info<typeT>::element_type, bool, int8_t, uint8_t, int16_t, uint16_t,
int32_t, uint32_t, int64_t, uint64_t, float, double, std::complex<float>,
std::complex<double>, std::string>) { if constexpr (holoscan::dimension_of_v<typeT>
== 1) { // Create vector of Handles YAML::Node yaml_node = YAML::Load("[]"); // Create
an empty sequence for (typename holoscan::type_info<typeT>::element_type item :
value) { yaml_node.push_back(item); } return
GxfParameterSetFromYamlNode(context, uid, key, &yaml_node, ""); } else if
constexpr (holoscan::dimension_of_v<typeT> == 2) { YAML::Node yaml_node =
YAML::Load("[]"); // Create an empty sequence for (std::vector<typename
holoscan::type_info<typeT>::element_type>& vec : value) { YAML::Node
inner_yaml_node = YAML::Load("[]"); // Create an empty sequence for (typename
holoscan::type_info<typeT>::element_type item : vec) {
inner_yaml_node.push_back(item); } if (inner_yaml_node.size() > 0) {
yaml_node.push_back(inner_yaml_node); } } return
GxfParameterSetFromYamlNode(context, uid, key, &yaml_node, ""); } } break; } case
ArgElementType::kHandle: { HOLOSCAN_LOG_ERROR("Unable to handle vector of
ArgElementType::kHandle for key '{}'", key); return GXF_FAILURE; } case
ArgElementType::kYAMLNode: { HOLOSCAN_LOG_ERROR("Unable to handle vector
of ArgElementType::kYAMLNode for key '{}'", key); return GXF_FAILURE; } case
ArgElementType::kIOSpec: { if constexpr (std::is_same_v<typeT,
std::vector<holoscan::IOSpec*>>) { // Create vector of Handles YAML::Node
yaml_node = YAML::Load("[]"); // Create an empty sequence for (auto& io_spec : value)
{ if (io_spec) { // Only consider non-null IOSpecs auto gxf_resource =
std::dynamic_pointer_cast<GXFResource>(io_spec->connector());
```

yaml_node.push_back(gxf_resource->gxf_cname()); } } return GxfParameterSetFromYamlNode(context, uid, key, &yaml_node, ""); } HOLOSCAN_LOG_ERROR("Unable to handle vector of std::vector<holoscan::IOSpec*>> for key: " "'{}'", key); break; } case ArgElementType::kResource: { if constexpr (std::is_same_v<typename holoscan::type_info<typeT>::element_type, std::shared_ptr<Resource>> && holoscan::type_info<typeT>::dimension == 1) { *// Create vector of Handles* YAML::Node yaml_node; for (auto& resource : value) { auto gxf_resource = std::dynamic_pointer_cast<GXFResource>(resource); *// Push back the resource's gxf_cname only if it is not null.* if (gxf_resource) { *// Initialize GXF component if it is not already initialized.* if (gxf_resource->gxf_context() == nullptr) { gxf_resource->gxf_eid(gxf::get_component_eid(context, uid)); *// set Entity ID of the component* gxf_resource->initialize(); } gxf_uid_t resource_cid = gxf_resource->gxf_cid(); std::string full_resource_name = gxf::get_full_component_name(context, resource_cid); yaml_node.push_back(full_resource_name.c_str()); } else { HOLOSCAN_LOG_TRACE("Resource item in the vector is null. Skipping it for key '{}'", key); } } return GxfParameterSetFromYamlNode(context, uid, key, &yaml_node, ""); } HOLOSCAN_LOG_ERROR("Unable to handle vector of ArgElementType::kResource for key '{}'", key); break; } case ArgElementType::kCondition: { if constexpr (std::is_same_v<typename holoscan::type_info<typeT>::element_type, std::shared_ptr<Condition>> && holoscan::type_info<typeT>::dimension == 1) { *// Create vector of Handles* YAML::Node yaml_node; for (auto& condition : value) { auto gxf_condition = std::dynamic_pointer_cast<GXFCondition>(condition); *// Push back the condition's gxf_cname only if it is not null.* if (gxf_condition) { *// Initialize GXF component if it is not already initialized.* if (gxf_condition->gxf_context() == nullptr) { gxf_condition->gxf_eid(gxf::get_component_eid(context, uid)); *// set Entity ID of the component* gxf_condition->initialize(); } gxf_uid_t condition_cid = gxf_condition->gxf_cid(); std::string full_condition_name = gxf::get_full_component_name(context, condition_cid); yaml_node.push_back(full_condition_name.c_str()); } else { HOLOSCAN_LOG_TRACE("Condition item in the vector is null. Skipping it for key '{}'", key); } } return GxfParameterSetFromYamlNode(context, uid, key, &yaml_node, ""); } HOLOSCAN_LOG_ERROR("Unable to handle vector of ArgElementType::kCondition for key '{}'", key); break; } case ArgElementType::kCustom: { HOLOSCAN_LOG_ERROR("Unable to handle vector of ArgElementType::kCustom type for key '{}'", key); return GXF_FAILURE; } } break; } case ArgContainerType::kArray: { HOLOSCAN_LOG_ERROR("Unable to handle

```
ArgContainerType::kArray type for key '{}'", key); break; } } return GXF_SUCCESS; }
private: GXFParameterAdaptor() { add_param_handler<bool>();
add_param_handler<int8_t>(); add_param_handler<int16_t>();
add_param_handler<int32_t>(); add_param_handler<int64_t>();
add_param_handler<uint8_t>(); add_param_handler<uint16_t>();
add_param_handler<uint32_t>(); add_param_handler<uint64_t>();
add_param_handler<float>(); add_param_handler<double>();
add_param_handler<std::string>(); add_param_handler<std::vector<bool>>();
add_param_handler<std::vector<int8_t>>();
add_param_handler<std::vector<int16_t>>();
add_param_handler<std::vector<int32_t>>();
add_param_handler<std::vector<int64_t>>();
add_param_handler<std::vector<uint8_t>>();
add_param_handler<std::vector<uint16_t>>();
add_param_handler<std::vector<uint32_t>>();
add_param_handler<std::vector<uint64_t>>();
add_param_handler<std::vector<float>>();
add_param_handler<std::vector<double>>();
add_param_handler<std::vector<std::string>>();
add_param_handler<std::vector<std::vector<bool>>>();
add_param_handler<std::vector<std::vector<int8_t>>>();
add_param_handler<std::vector<std::vector<int16_t>>>();
add_param_handler<std::vector<std::vector<int32_t>>>();
add_param_handler<std::vector<std::vector<int64_t>>>();
add_param_handler<std::vector<std::vector<uint8_t>>>();
add_param_handler<std::vector<std::vector<uint16_t>>>();
add_param_handler<std::vector<std::vector<uint32_t>>>();
add_param_handler<std::vector<std::vector<uint64_t>>>();
add_param_handler<std::vector<std::vector<float>>>();
add_param_handler<std::vector<std::vector<double>>>();
add_param_handler<std::vector<std::vector<std::string>>>();
add_param_handler<YAML::Node>(); add_param_handler<holoscan::IOSpec*>();
add_param_handler<std::vector<holoscan::IOSpec*>>();
add_param_handler<std::shared_ptr<Resource>>();
add_param_handler<std::vector<std::shared_ptr<Resource>>>();
add_param_handler<std::shared_ptr<Condition>>();
```

add_param_handler<std::vector<std::shared_ptr<Condition>>>(); }
std::unordered_map<std::type_index, AdaptFunc> function_map_;
std::unordered_map<std::type_index, AdaptFunc> arg_function_map_; }; } *//
namespace holoscan::gxf* #endif/*
HOLOSCAN_CORE_EXECUTORS_GXF_GXF_PARAMETER_ADAPTOR_HPP */