Program Listing for File gxf_utils.hpp

Return to documentation for file (include/holoscan/core/gxf/gxf_utils.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_GXF_GXF_UTILS_HPP #define HOLOSCAN_CORE_GXF_GXF_UTILS_HPP #include <gxf/core/gxf.h> #include <cstdlib> #include <iostream> #include <sstream> #include <string> #include <utility> #include <common/assert.hpp> #include <common/backtrace.hpp> #include <common/type_name.hpp> #include "holoscan/logger/logger.hpp" *// macro like GXF_ASSERT_SUCCESS, but uses HOLOSCAN_LOG_ERROR and includes line/filename info // Note: HOLOSCAN_GXF_CALL depends on GNU C statement expressions ({ }) // https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html* #define HOLOSCAN_GXF_CALL(stmt) \ ({ \ gxf_result_t code = (stmt); \ if (code != GXF_SUCCESS) { \ HOLOSCAN_LOG_ERROR("GXF call {} in line {} of file {} failed with '{}' ({})", \ #stmt, \ __LINE__, \ __FILE__, \ GxfResultStr(code), \ code); \ if (!std::getenv("HOLOSCAN_DISABLE_BACKTRACE")) { PrettyPrintBacktrace(); } \ } \ code; \ }) #define HOLOSCAN_GXF_CALL_FATAL(stmt) \ { \ gxf_result_t code = HOLOSCAN_GXF_CALL(stmt); \ if (code != GXF_SUCCESS) { throw std::runtime_error("failure during GXF call"); } \ } #define HOLOSCAN_GXF_CALL_MSG(stmt, ...) \ ({ \ gxf_result_t code = HOLOSCAN_GXF_CALL(stmt); \ if (code != GXF_SUCCESS) { HOLOSCAN_LOG_ERROR(__VA_ARGS__); } \ code; \ }) #define HOLOSCAN_GXF_CALL_MSG_FATAL(stmt, ...) \ { \ gxf_result_t code = HOLOSCAN_GXF_CALL_MSG(stmt, __VA_ARGS__); \ if (code != GXF_SUCCESS) { throw std::runtime_error("failure during GXF call"); } \ } *// Duplicate of HOLOSCAN_GXF_CALL but without a backtrace and logs a warning instead of an error.* #define HOLOSCAN_GXF_CALL_WARN(stmt) \ ({ \ gxf_result_t code = (stmt); \ if (code != GXF_SUCCESS) { \ HOLOSCAN_LOG_WARN("GXF call {} in line {} of file {} failed with

'{}' ({})", \ #stmt, \ __LINE__, \ __FILE__, \ GxfResultStr(code), \ code); \ } \ code; \ }) *// Duplicate of HOLOSCAN_GXF_CALL_MSG but logs a warning instead of an error.* #define HOLOSCAN_GXF_CALL_WARN_MSG(stmt, ...) \ ({ \ gxf_result_t code = HOLOSCAN_GXF_CALL_WARN(stmt); \ if (code != GXF_SUCCESS) { HOLOSCAN_LOG_WARN(__VA_ARGS__); } \ code; \ }) namespace holoscan::gxf { gxf_uid_t get_component_eid(gxf_context_t context, gxf_uid_t cid); std::string get_full_component_name(gxf_context_t context, gxf_uid_t cid); std::string create_name(const char* prefix, int index); std::string create_name(const char* prefix, const std::string& name); template <typename S> inline gxf_uid_t find_component_handle(gxf_context_t context, gxf_uid_t component_uid, const char* key, const std::string& tag, const std::string& prefix) { gxf_uid_t eid; std::string component_name; const size_t pos = tag.find('/'); if (pos == std::string::npos) { *// Get the entity of this component* const gxf_result_t result_1 = GxfComponentEntity(context, component_uid, &eid); if (result_1 != GXF_SUCCESS) { return 0; } component_name = tag; } else { component_name = tag.substr(pos + 1); *// Get the entity* gxf_result_t result_1_with_prefix = GXF_FAILURE; *// Try using entity name with prefix* if (!prefix.empty()) { const std::string entity_name = prefix + tag.substr(0, pos); result_1_with_prefix = GxfEntityFind(context, entity_name.c_str(), &eid); if (result_1_with_prefix != GXF_SUCCESS) { HOLOSCAN_LOG_WARN("Could not find entity (with prefix) '{}' while parsing parameter '{}' " "of component {}", entity_name.c_str(), key, component_uid); } } *// Try using entity name without prefix, if lookup with prefix failed* if (result_1_with_prefix != GXF_SUCCESS) { const std::string entity_name = tag.substr(0, pos); const gxf_result_t result_1_no_prefix = GxfEntityFind(context, entity_name.c_str(), &eid); if (result_1_no_prefix != GXF_SUCCESS) { HOLOSCAN_LOG_ERROR("Could not find entity '{}' while parsing parameter '{}' of component {}", entity_name.c_str(), key, component_uid); return 0; } else if (!prefix.empty()) { HOLOSCAN_LOG_WARN("Found entity (without prefix) '{}' while parsing parameter '{}' " "of component {} in a subgraph, however the approach is deprecated," " please use prerequisites instead", entity_name.c_str(), key, component_uid); } } } *// Get the type id of the component we are are looking for.* gxf_tid_t tid; const gxf_result_t result_2 = GxfComponentTypeId(context, ::nvidia::TypenameAsString<S>(), &tid); if (result_2 != GXF_SUCCESS) { return 0; } *// Find the component in the indicated entity* gxf_uid_t cid; const gxf_result_t result_3 = GxfComponentFind(context, eid, tid, component_name.c_str(), nullptr, &cid); if (result_3 != GXF_SUCCESS) { if (component_name == "<Unspecified>") { HOLOSCAN_LOG_DEBUG("Using an <Unspecified> handle in entity {} while parsing

```
parameter '{}'" " of component {}. This handle must be set to a valid component before graph activation", eid, key, component_uid); return 0; } else { HOLOSCAN_LOG_WARN( "Could not find component '{}' in entity {} while parsing parameter '{}' " "of component {}", component_name.c_str(), eid, key, component_uid); } return 0; } return cid; } bool has_component(gxf_context_t context, gxf_uid_t eid, gxf_tid_t tid = GxfTidNull(), const char* name = nullptr, int32_t* offset = nullptr, gxf_uid_t* cid = nullptr); gxf_uid_t add_entity_group(void* context, std::string name); } // namespace holoscan::gxf #endif/* HOLOSCAN_CORE_GXF_GXF_UTILS_HPP */
```