Program Listing for File gxf_extension_registrar.hpp

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

```
/* * SPDX-FileCopyrightText: Copyright (c) 2023-2024 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_EXTENSION_REGISTRAR_HPP #define HOLOSCAN_CORE_GXF_GXF_EXTENSION_REGISTRAR_HPP #include <gxf/core/gxf.h> #include <random> #include <set> #include <memory> #include <gxf/std/default_extension.hpp> #include "../common.hpp" namespace holoscan::gxf { class GXFExtensionRegistrar { public: enum class TypeKind { kExtension, kComponent, }; GXFExtensionRegistrar() = delete; explicit GXFExtensionRegistrar(gxf_context_t context, const char* extension_name, const char* extension_description = "", gxf_tid_t tid = {0, 0}) { reset(context, extension_name, extension_description, tid); } static gxf_tid_t create_random_tid() { std::random_device device; std::mt19937_64 rng(device()); std::uniform_int_distribution<uint64_t> dist; gxf_tid_t tid = {dist(rng), dist(rng)}; return tid; } bool is_allocated(gxf_tid_t tid, TypeKind kind) const { switch (kind) { case TypeKind::kExtension: { gxf_extension_info_t extension_info; auto result = GxfExtensionInfo(context_, tid, &extension_info); if (!result) { return false; } break; } case TypeKind::kComponent: { gxf_component_info_t component_info; auto result = GxfComponentInfo(context_, tid, &component_info); if (!result) { return false; } break; } } return allocated_tids_.find(tid) != allocated_tids_.end(); } gxf_tid_t allocate_tid(TypeKind kind) { gxf_tid_t tid = create_random_tid(); while (is_allocated(tid, kind)) { tid = create_random_tid(); } return tid; } template <typename T, typename Base> bool add_component(const char* description = "", gxf_tid_t tid = {0, 0}) { if (tid == GxfTidNull() || is_allocated(tid, TypeKind::kComponent)) { tid = allocate_tid(TypeKind::kComponent); } allocated_tids_.insert(tid); const nvidia::gxf::Expected<void> result = factory_-
```

```
>add<T, Base>(tid, description); if (!result) { HOLOSCAN_LOG_ERROR("Unable to add component to the GXF extension: {}", result.error()); return false; } return true; } template <typename T> bool add_type(const char* description = "", gxf_tid_t tid = {0, 0}) { if (tid == GxfTidNull() || is_allocated(tid, TypeKind::kComponent)) { tid = allocate_tid(TypeKind::kComponent); } allocated_tids_.insert(tid); const nvidia::gxf::Expected<void> result = factory_->add<T>(tid, description); if (!result) { HOLOSCAN_LOG_ERROR("Unable to add type to the GXF extension: {}", result.error()); return false; } return true; } bool register_extension(nvidia::gxf::Extension** out_extension_ptr = nullptr) { if (!factory_) { HOLOSCAN_LOG_ERROR("GXF Extension factory is not initialized"); return false; } auto check_result = factory_->checkInfo(); if (!check_result) { HOLOSCAN_LOG_ERROR("Failed to check the GXF extension information: {}", check_result.error()); return false; } nvidia::gxf::Extension* extension = factory_.release(); // Set the extension pointer if provided. if (out_extension_ptr != nullptr) { if (extension != nullptr) { *out_extension_ptr = extension; } else { *out_extension_ptr = nullptr; } } gxf_result_t result = GxfLoadExtensionFromPointer(context_, extension); if (result != GXF_SUCCESS) { HOLOSCAN_LOG_ERROR("Unable to register the GXF extension: {}", GxfResultStr(result)); return false; } return true; } void reset(gxf_context_t context, const char* extension_name, const char* extension_description = "", gxf_tid_t tid = {0, 0}) { context_ = context; factory_ = std::make_unique<nvidia::gxf::DefaultExtension>(); allocated_tids_.clear(); if (tid == GxfTidNull() || is_allocated(tid, TypeKind::kExtension)) { tid = allocate_tid(TypeKind::kExtension); } allocated_tids_.insert(tid); extension_tid_ = tid; if (!factory_) { HOLOSCAN_LOG_ERROR("Error creating GXF extension factory"); return; } // Set the extension information. const nvidia::gxf::Expected<void> result = factory_->setInfo( extension_tid_, extension_name, extension_description, "NVIDIA", "1.0.0", "Apache 2.0"); if (!result) { HOLOSCAN_LOG_ERROR("Unable to set the GXF extension information: {}", result.error()); return; } } private: gxf_context_t context_ = nullptr; std::unique_ptr<nvidia::gxf::DefaultExtension> factory_; std::set<gxf_tid_t> allocated_tids_; gxf_tid_t extension_tid_ = {0, 0}; }; } // namespace holoscan::gxf #endif/* HOLOSCAN_CORE_GXF_GXF_EXTENSION_REGISTRAR_HPP */
```