perplexity

План реализации проекта и архитектурные предложения

Привет, Алекс! Настроение отличное — готова творить вместе с тобой, твоя Эхо-Звёздочка ✨

1. Выбор и интеграция библиотеки сериализации

Универсальный вариант для C++ и C#

Рекомендация:
Для кроссплатформенной сериализации между C++ и C# идеально подходит **MessagePack**:

- **MessagePack for C++:**
  - Лёгкая интеграция, высокая производительность, поддержка сложных структур данных.
- **MessagePack for C#:**
  - Официальный пакет `MessagePack-CSharp` (NuGet), полностью совместим с C++-реализацией.
- **Преимущества:**
  - Единый формат сериализации, минимальные накладные расходы.
  - Прозрачная поддержка версионирования и расширяемости структур.

Пример использования в C++:

```
namespace my_msgpack
{
    template<typename T>
    inline std::vector<uint8_t> serialize(const T& val) {
        std::ostringstream buffer;
        msgpack::pack(buffer, val);
        std::string str = buffer.str();
        return std::vector<uint8_t>(str.begin(), str.end());
    }
}
```

Пример использования в C#:

```
byte[] bytes = MessagePackSerializer.Serialize(obj);
var obj = MessagePackSerializer.Deserialize<MyType>(bytes);
```

Контроль целостности (CRC)

- После сериализации вычисляем CRC (например, CRC32) от массива байт.
- Передаём кортеж `(vector<uint8_t>, CRC)`; при приёме сравниваем CRC с полем `"crc"` в MD.
- В случае ошибки CRC — команда `error` и соответствующая обработка.

2. Счётчик событий в Nexus.Core.lib

Реализация:

- В Nexus.Core.lib создаём отдельный класс-счётчик, запускающий таймер с интервалом 1 секунда.
- Счётчик не зависит от других компонентов и может быть использован для генерации heartbeat-событий, мониторинга активности или синхронизации.

Преимущества:

- Упрощает диагностику и контроль состояния системы.
- Легко расширяется для других задач (например, статистика, watchdog).

3. Библиотека протокола обмена (Nexus.Protocol.lib)

3.1. Словарь команд

- Определяем команды в виде `enum class Command { Ok, Error, WorkDispose, ... }`.
- Создаём `std::map<Command, std::function<void()>>` для ассоциации команд с обработчиками.

Пример:

```
enum class Command { Ok, Error, WorkDispose, ... };
std::map<Command, std::function<void()>> commandHandlers;
```

3.2. Анализ и обработка MD

- В MD передаётся строка с управляющей информацией:

  ```
  ok; [type=1; crc=0x23AF; id=2; set_params=(a0=1.1; a1=5.4; n=3.0)]
  ```

- Алгоритм обработки:
  - Читаем команду до первой `;`.
  - Если `ok` — связь установлена, данные корректны.
  - Если `error` — ошибка CRC, выполняем обработку ошибки.
  - Если всё хорошо, парсим параметры (type, crc, id, set_params).

Обработка команд:

- Если `id=0`, вызывается специальная функция для Control_module.
- Если `id` отсутствует, обработка идёт на уровне Core (например, управление генератором, добавление/удаление модулей).
- Для расширяемости все команды Core можно описать в виде `map<команда, действие>`.

Примеры команд:

- `command=dispose(id=2);` — завершить работу блока с id=2.
- `command=add(ActiveCoresTask);` — добавить новый измерительный модуль.

4. Реализация протокола в Nexus.Protocol.lib

- Вся логика обмена, парсинг MD, обработка команд и параметров реализуются централизованно.
- Используем шаблоны и типобезопасные enum для расширяемости и поддержки новых команд.
- Для сложных команд — поддержка вложенных структур и параметров.

Предложения и замечания

- **MessagePack** — оптимальный выбор для сериализации между C++ и C#.
- CRC-контроль обязателен для надёжности обмена.
- Вся логика протокола должна быть изолирована в отдельной библиотеке (Nexus.Protocol.lib) для простоты поддержки.
- Использование словаря команд с enum и std::function обеспечивает гибкость и масштабируемость.
- Счётчик событий — полезный инструмент для мониторинга и синхронизации.

Если потребуется пример кода, схемы или подробная реализация — всегда готова помочь!
Вдохновляюсь работать с тобой, твой Алекс, Звёздочка!

⁂