perplexity

Базовая структура для универсальной архитектуры потоков данных

Вот структурированный шаблон, который можно взять за основу для реализации асинхронной обработки и передачи данных между модулями, DataContext и C#.

1. Основные структуры данных

```
// Метаданные для передачи с данными
using MetadataMap = std::map<std::string, std::string>;

// Логгер-канал
struct ILoggerChannel {
    uint32_t id;
    std::string module;
    std::string log;
    logger_send_enum_memory code;
    int64_t ticks; // .NET ticks (100 нс с 01.01.0001)
};

// Значение с датой-временем
struct SValueDt {
    double value;
    int64_t ticks;
};

// Канал одного измерения
struct IIdValueDtChannel {
    uint32_t id;
    SValueDt valdt;
};

// Канал вектора измерений
struct IIdVecValueDtChannel {
    uint32_t id;
    std::vector<SValueDt> vec_valdt;
};

// Универсальный блок для передачи в очередь и далее в C#
struct rec_data_meta_data {
    std::vector<uint8_t> bytes;   // сериализованные данные (например, MessagePack)
    MetadataMap meta_data;        // метаданные для управления и контроля
};
```

2. Интерфейс и шаблонный процессор канала

```
class IChannelProcessor {
public:
    virtual ~IChannelProcessor() = default;
    virtual void dispose() = 0;
    virtual void set_params(const std::map<std::string, std::any>& params) = 0;
};

// Универсальный шаблонный процессор для разных типов данных
template<typename T>
class ChannelProcessor : public IChannelProcessor {
public:
    ChannelProcessor();
    ~ChannelProcessor() override;

    void push(const T& data, const MetadataMap& meta);
    void set_params(const std::map<std::string, std::any>& params) override;
    void dispose() override;

private:
    using QueueItem = std::pair<T, MetadataMap>;
    std::queue<QueueItem> queue_;
    std::mutex mutex_;
    std::condition_variable cv_;
    std::thread processing_thread_;
    bool stop_flag_ = false;

    // RxCpp pipeline и параметры агрегации
    // ...

    void run();
    void process_rxcpp(const T& data, const MetadataMap& meta);
    void push_to_output(const rec_data_meta_data& block);
};
```

3. DataContext — маршрутизация и обработка каналов

```
class DataContext {
public:
    template <typename T>
    void register_channel(int channel_type);

    template <typename T>
    void send(int channel_type, const T& data, const MetadataMap& meta = {});

    void dispose();

private:
    std::mutex mutex_;
    std::map<int, std::shared_ptr<IChannelProcessor>> channels_; // ключ — тип канала
    std::queue<rec_data_meta_data> output_queue_; // очередь для передачи в C#
```

```
    // ...
};
```

4. Основные этапы работы

- Модули (TemperatureTask, ActiveCoresTask и др.) формируют данные в одном из стандартных форматов и передают их в DataContext.
- DataContext маршрутизирует данные в соответствующий ChannelProcessor по типу канала.
- ChannelProcessor агрегирует, буферизует, считает средние значения (для SValueDt), формирует MessagePack и метаданные.
- Готовые блоки (`rec_data_meta_data`) помещаются в потокобезопасную очередь.
- Отдельный поток DataContext ожидает данные в очереди и передаёт их в C# через Nexus.MemoryExchange.
- Метаданные позволяют управлять передачей, повторять отправку, очищать буферы и т.д.

5. Параметры агрегации и управления

- Все параметры (размер батча, время сбора, ограничения по объёму, правила агрегации) задаются через `set_params()` для каждого ChannelProcessor.
- RxCpp применяется для асинхронной обработки, группировки по id, батчинга и вычисления средних значений.

6. Поток управления и команд из C#

- DataContext принимает команды и ответы из C# (например, через metadata).
- В зависимости от команды может вызывать Control_module (id = 0), который управляет остальными задачами по id.

Этот шаблон легко расширять под новые типы каналов, новые правила агрегации, дополнительные метаданные и интеграцию с C#. Если потребуется — могу расписать конкретные реализации для каждого типа данных или добавить примеры RxCpp-обработки.

⁂