Описание встроенного языка программирования ProtoEMU

Редакция от 10.10.2026 · docs.kb-agava.ru

Содержание

1 Область применения 3
2 Структура и выполнение программы 3
  2.1 Имена программ и источников 3
  2.2 Основная функция 4
  2.3 Циклическое выполнение 4
  2.4 Интервал обработки 4
3 Синтаксис и типы 5
  3.1 Лексические правила 5
  3.2 Основные типы 5
  3.3 Преобразования 6
  3.4 Операторы 6
  3.5 Управляющие конструкции 6
4 Переменные 7
  4.1 Локальное и глобальное состояние 7
  4.2 Наблюдаемые переменные 7
  4.3 Имена источников 8
5 Функции 8
  5.1 Объявление и вызов 8
  5.2 Передача параметров 8
  5.3 Перегрузка и значения по умолчанию 9
6 Классы, методы и дескрипторы 9
7 Работа со значениями Value 10
  7.1 Назначение 10
  7.2 Методы чтения, записи и проверки 10
  7.3 Присваивание и операции 11
  7.4 Использование источников и переменных 11
8 Стандартные функции и типы 11
  8.1 Математика 11
  8.2 Строки 12
  8.3 Массивы 12
  8.4 Дата и время 13
  8.5 Файлы 13
9 Порядок использования 14
  9.1 Создание и запуск программы 14
  9.2 Изменение программы 14
10 Прикладные примеры 14
  10.1 Линейное преобразование с ограничением 14
  10.2 Пороговый автомат с гистерезисом 15
  10.3 Счётчик и выходной параметр 15
  10.4 Битовое поле 16
  10.5 Скользящее среднее 16
  10.6 Генератор синусоидального сигнала 16
  10.7 Объектное состояние 17

10.8 Строки, календарное время и файл 18
11 Диагностика и ограничения 18
11.1 Проверка документа 18
11.2 Исполнение 19
11.3 Применимость примеров 19

Данное руководство описывает синтаксис, систему типов, функции, методы и порядок написания программ в ProtoEMU. Рассматриваются взаимодействие с регистрами, обработка сигналов и сохранение состояния между циклами. Материал предназначен для пользователей, знакомых с основами программирования.

Синтаксис языка программирования ProtoEMU близок к C++, однако набор доступных типов и функций отличается. Материал содержит правила работы в редакторе, справочные таблицы и готовые примеры для обработки значений регистров и отображения результатов на виджетах.

1 Область применения

Язык программирования ProtoEMU является языком со статической типизацией: каждой переменной назначается тип, а соответствие типов проверяется до запуска программы. Синтаксис близок к C++.

В ProtoEMU программы применяются для вычисления значений регистров, реализации логических условий, формирования модельных сигналов, фильтрации и подготовки данных для виджетов. Для этих задач доступны следующие средства:

Средства	Содержание	Пример
Язык	Типизация, выражения, функции и классы	`if`, `for`, `double`, `class`
Стандартные функции и типы	Строки, массивы, математика, файлы, дата и время	`string`, `array<double>`, `sin`, `file`
Работа с источниками	Чтение и изменение значений регистров	`Value`, `Register1.ToDouble()`

При написании программ следует использовать функции и типы, описанные в этом руководстве. Подключение заголовков C++ через `#include`, а также `std::vector`, `std::cout`, `printf` и `sleep` не поддерживаются.

2 Структура и выполнение программы

2.1 Имена программ и источников

Программы размещаются в группе «Программы» дерева проекта. Функции и глобальные переменные одной включённой программы могут использоваться в других программах проекта. Каждую глобальную переменную объявляют один раз; при обращении к ней из нескольких программ используют одно и то же имя. Для разных переменных и функций выбирают разные имена. Перегрузка функций описана в подразделе 5.3.

К источнику данных обращаются по его имени в дереве проекта. Например, значение включённого регистра `Register1` можно прочитать следующим образом:

```
double input = Register1.ToDouble();
```

В тексте программы не нужно объявлять переменную `Register1`: это имя уже обозначает источник данных проекта. Работа со значениями источников описана в разделе 7.

2.2 Основная функция

Для программы с именем `Process` применяется структура:

```
void Process()
{
    // Действия одного цикла обработки.
}
```

Основная функция должна иметь имя программы, не принимать аргументов и иметь тип результата `void`. Например, для программы `Process` используется `void Process()`. Имя `main` не требуется.

Основную функцию размещают первой в тексте. Глобальные переменные, вспомогательные функции, классы, перечисления и пространства имён располагают после неё. Перед основной функцией не следует помещать комментарии с символом `{`.

Основная функция может использовать переменные и вспомогательные функции, объявленные ниже по тексту:

```
void Process()
{
    Counter++;
    ResultValue = Limit(double(Counter), 0.0, 100.0);
}

int Counter = 0;
double ResultValue = 0.0;

double Limit(double value, double lower, double upper)
{
    if (value < lower) return lower;
    if (value > upper) return upper;
    return value;
}
```

2.3 Циклическое выполнение

После запуска эмуляции основная функция включённой программы вызывается повторно. Один вызов должен выполнять один шаг алгоритма: прочитать входные значения, произвести вычисления и задать результат.

Повторение основной функции не требуется организовывать отдельным бесконечным циклом. Для накопления данных между вызовами используют глобальные переменные. Например, счётчик можно увеличивать на единицу при каждом вызове, а фильтр — дополнять одним новым отсчётом.

Запись значений во внешнее устройство зависит от настроек транспорта и регистра. После присваивания следует контролировать фактическое значение и сообщения приложения.

2.4 Интервал обработки

Свойство `ProcessInterval` группы «Программы» задаёт интервал в миллисекундах; значение по умолчанию — **500 мс**. Фактический промежуток между вызовами также зависит от длительности вычислений и может отличаться от заданного интервала. Настройка не обеспечивает точного периода измерений.

После изменения интервала следует перезапустить эмуляцию. Если алгоритм использует расчётный шаг времени, его задают отдельно и согласовывают с интервалом обработки.

3 Синтаксис и типы

3.1 Лексические правила

Регистр букв значим: `Counter` и `counter` — разные идентификаторы. Для имён программ, источников и переменных рекомендуется использовать латинские буквы, цифры и знак подчёркивания. Имя должно начинаться с буквы или подчёркивания. Пробелы и ключевые слова языка в именах применять не следует.

Объявления, присваивания и вызовы завершаются `;`. Блоки заключаются в `{ ... }`. Комментарии имеют форму `// ...` или `/* ... */`.

Не следует применять C++-суффикс `u` у целочисленного литерала. Беззнаковый тип задаётся объявлением `uint` либо явным преобразованием `uint(1)`.

```
int count = 10;
double gain = 1.25;
bool enabled = true;
string caption = "Register 1";
uint mask = 0x000F;
```

3.2 Основные типы

Тип	Назначение	Примечание
`bool`	Логическое значение	`true` или `false`
`int8`, `int16`, `int`, `int64`	Знаковые целые	`int` — 32 бита
`uint8`, `uint16`, `uint`, `uint64`	Беззнаковые целые	`uint` — 32 бита
`float`	Вещественное число	32 бита
`double`	Вещественное число	64 бита
`string`	Строка	Текстовые данные
`array<T>`	Массив элементов типа T	Упорядоченная последовательность элементов
`Value`	Сигнальное значение ProtoEMU	Содержимое определяется во время выполнения
`void`	Отсутствие результата	Тип результата функции

Поддерживаются перечисления и псевдонимы примитивных типов:

```
enum OperatingMode
{
    ModeIdle = 0,
    ModeActive = 1
}

typedef double Real;
```

В исполняемой программе эти объявления располагаются после основной функции.

3.3 Преобразования

Явное преобразование имеет вид `Тип(выражение)`:

```
double average = double(sum) / double(count);
int whole = int(average);
```

При делении двух целых результат также является целым. Для вещественного результата хотя бы один операнд должен быть вещественным. Сужение типа может приводить к потере диапазона или точности; диапазон проверяется до записи регистра.

3.4 Операторы

Группа	Обозначения	Пример
Арифметика	`+`, `-`, `*`, `/`, `%`	`y = gain * x + offset;`
Сравнение	`==`, `!=`, `<`, `<=`, `>`, `>=`	`temperature >= limit`
Логика	`&&`, `||`, `!`	`enabled && ready`
Побитовые операции	`&`, `|`, `^`, `~`, `<<`, `>>`	`flags & 0x000F`
Присваивание	`=`, `+=`, `-=`, `*=`, `/=`	`counter += 1;`
Инкремент и декремент	`++`, `--`	`counter++;`
Условное выражение	`условие ? a : b`	`ready ? 1 : 0`
Доступ к члену	`.`	`values.length()`
Разрешение области	`::`	`MathUtils::Limit(x)`

Таблица относится к примитивным типам и объектам с соответствующими операциями. Интерфейс `Value` описан отдельно в разделе 7. Логические операции над `bool` допускают короткое замыкание. Для сложных условий предпочтительны явные скобки.

3.5 Управляющие конструкции

```
if (value < lower)
{
    value = lower;
}
else if (value > upper)
{
    value = upper;
}

for (uint i = 0; i < samples.length(); ++i)
{
    sum += samples[i];
}

while (index < count)
{
    ++index;
}

do
{
    ++attempt;
}
while (attempt < maximum);
```

```
switch (mode)
{
case 0:
    result = 0;
    break;
case 1:
    result = 1;
    break;
default:
    result = -1;
    break;
}
```

`break` завершает ближайший цикл или `switch`; `continue` переходит к следующей итерации; `return` завершает функцию. Условие должно иметь логический тип. Локальные переменные следует явно инициализировать до чтения. Подробнее: операторы управления.

Блок демонстрирует синтаксис и предполагает предварительные объявления переменных. Полные программы представлены в разделе 10.

4 Переменные

4.1 Локальное и глобальное состояние

Локальная переменная существует в пределах своего блока. Глобальная переменная объявляется вне функций и сохраняет значение между вызовами программы. После проверки изменённого текста и нового запуска следует рассчитывать на начальные значения, указанные в объявлениях. Для сохранения данных между сеансами работы требуется отдельная запись, например в файл. Подробнее: глобальные сущности.

```
void CountCycles()
{
    int localCount = 0;
    localCount++;           // При каждом вызове получается 1.
    TotalCycles++;          // Состояние сохраняется между вызовами.
}

int TotalCycles = 0;
```

Для состояния фильтра, генератора или автомата следует использовать глобальные переменные. Локальное объявление с C++-модификатором `static` для этой цели не применяется.

4.2 Наблюдаемые переменные

После успешной проверки документа глобальные переменные появляются в группе «Переменные» дерева проекта. Их можно использовать как источники виджетов.

Для наблюдения и изменения значений через виджеты доступны типы `int`, `uint`, `bool`, `float`, `double`, `string` и `Value`. Массивы, объекты классов и целые другой разрядности используют для вычислений в программе. Для отображения результата объявляют отдельную переменную одного из перечисленных типов.

Константы предназначены для чтения и не должны использоваться как управляемые источники. Для наблюдаемых переменных рекомендуется глобальная область без пространства имён (`namespace`). Это позволяет обращаться к ним по простому имени и выбирать их в качестве источников виджетов.

4.3 Имена источников

Для обращения к источнику используют его имя. Если регистр проекта называется `Register1`, чтение и запись имеют вид:

```
double input = Register1.ToDouble();
Register1 = 42;
```

Повторное глобальное объявление `Register1` недопустимо. Переименование источника требует исправления обращений в тексте. Переименование узла наблюдаемой переменной также не переписывает её объявление и ссылки автоматически.

5 Функции

5.1 Объявление и вызов

Функция задаёт именованное преобразование или действие. Сигнатура содержит тип результата, имя и типы параметров. Тело задаётся одновременно с объявлением; отдельные прототипы для разрешения вызовов не требуются. Подробнее: объявление функций.

```
double Scale(double input, double gain, double offset)
{
    return input * gain + offset;
}
```

Вызов оформляется как `Scale(x, 2.0, 10.0)`. Вспомогательные функции могут принимать параметры. Требование отсутствия аргументов относится только к основной функции программы.

5.2 Передача параметров

Форма	Назначение	Пример
По значению	Передача значения	`double input`
`const T &in`	Входной параметр без разрешения изменения	`const string &in text`
`T &out`	Передача результата вызывающей стороне	`double &out result`
`T &inout`	Чтение и изменение объекта	`array<double> &inout values`

Параметр `&inout` предназначен для ссылочных типов, поддерживающих дескрипторы, например массивов и скриптовых классов. Для выдачи примитивного значения применяются результат функции или `&out`. Подробнее: параметры по ссылке.

```
bool Divide(double numerator, double denominator, double &out result)
{
    if (denominator == 0.0)
    {
        result = 0.0;
```

```
        return false;
    }
    result = numerator / denominator;
    return true;
}
```

Исходное значение `&out` не следует использовать как входное. Проверка делителя предотвращает деление на ноль, но не заменяет общую проверку исходных данных.

5.3 Перегрузка и значения по умолчанию

Одноимённые функции могут различаться типами или числом параметров. Параметры со значениями по умолчанию располагаются после обязательных:

```
double Gain(double value, double factor = 1.0)
{
    return value * factor;
}
```

При перегрузке следует учитывать неявные преобразования `Value`: явные `ToInt()` и `ToDouble()` однозначно задают тип аргумента.

6 Классы, методы и дескрипторы

Метод — функция объекта, вызываемая через `.` и работающая с его состоянием. Скриптовые классы поддерживают конструкторы, методы и управление доступом.

```
class Accumulator
{
    double total = 0.0;

    void Add(double value)
    {
        total += value;
    }

    double GetTotal() const
    {
        return total;
    }
}
```

Модификатор `const` у метода обозначает отсутствие изменения объекта через этот метод. Глобальный экземпляр класса сохраняет состояние между циклами.

Дескриптор вида `T@` содержит ссылку на объект, может быть `null` и присваивается с помощью `@`. Для примитивных типов и `Value` дескрипторы не применяются. Подробнее: объекты и дескрипторы.

```
Accumulator accumulator;
Accumulator@ reference = @accumulator;

if (reference !is null)
{
    reference.Add(1.0);
}
```

Фрагмент демонстрирует синтаксис; полный пример приведён в подразделе 10.7. Проверка `reference !is null` необходима перед использованием дескриптора, которому мог не быть назначен объект.

7 Работа со значениями Value

7.1 Назначение

`Value` — тип для работы со значениями источников. Он может содержать число, логическое значение или строку. Доступны конструкторы без параметров и с аргументами `int`, `double`, `float`, `string`, `bool`:

```
Value numeric(12.5);
Value logical(true);
Value text("Ready");
```

Перед вычислением значение источника рекомендуется сначала проверить, затем явно преобразовать, например через `Register1.ToDouble()`.

7.2 Методы чтения, записи и проверки

Имена методов необходимо записывать с указанным регистром букв.

Метод	Результат	Назначение
`ToString()`	`string`	Текстовое представление
`ToDouble()`	`double`	Преобразование к вещественному двойной точности
`ToFloat()`	`float`	Преобразование к вещественному одинарной точности
`ToInt()`	`int`	Преобразование к знаковому целому
`ToBool()`	`bool`	Преобразование к логическому значению
`SetValue(int value)`	`void`	Установка целого
`SetValue(string value)`	`void`	Установка строки
`SetValue(double value)`	`void`	Установка вещественного
`SetValue(bool value)`	`void`	Установка логического
`IsError()`	`bool`	Проверка, содержит ли значение сообщение об ошибке
`IsNumber()`	`bool`	Проверка пригодности значения к числовой обработке

`IsError()` не является общей проверкой доступности регистра или любых ошибок численного расчёта. `IsNumber()` исключает строку, значение ошибки, неопределённое значение и NaN — специальное значение, обозначающее недопустимый числовой результат. Логическое значение рассматривается как числовое. Бесконечность отдельной проверкой не исключается.

При несовместимом содержимом `ToInt()` может возвращать 0, а `ToBool()` — `false`. Числовая строка не должна считаться корректным измерением только на основании успешного преобразования через `ToDouble()` или `ToFloat()`. Проверка предотвращает смешение недоступного значения с корректным нулём:

```
if (Register1.IsError() || !Register1.IsNumber()) return;
double input = Register1.ToDouble();
```

7.3 Присваивание и операции

Предоставляется присваивание `bool`, `int`, `double`, `float`, `string` и другого `Value`; доступны неявные преобразования к `double`, `float`, `int`, `bool`, `string`:

```
Register2 = 100;
Register2.SetValue(100);
Value snapshot;
snapshot = Register1;
```

Для арифметики рекомендуется явно задавать тип операндов:

```
double scaled = Register1.ToDouble() * 2.0;
Register2 = scaled;
```

Дополнительно доступны `+= int`, `-= int`, `|`, `&`, `|=`, `&=` и `!`. Для `|` и `&` числовые операнды преобразуются через `ToInt()`; `|` со строковым операндом может объединять строковые представления. Для битовых масок предпочтителен явный целый тип, как в подразделе 10.4.

Проверка равенства доступна при сравнении с `bool`, `int`, `double` и `string`. Для сравнения с порогом следует сначала получить число: `Register1.ToDouble() > threshold`.

Логические выражения следует строить из значений типа `bool`:

```
bool condition = Register1.ToBool() && Register2.ToBool();
```

7.4 Использование источников и переменных

При записи в числовой регистр присваивают числовой результат, соответствующий его типу и диапазону. Строковый результат выводят через отдельную глобальную переменную типа `string`, связанную с текстовым виджетом.

Значение входного регистра может изменяться между вызовами программы. Для хранения состояния алгоритма, например накопленной суммы или предыдущего отсчёта, используют самостоятельные глобальные переменные.

8 Стандартные функции и типы

8.1 Математика

Основные математические функции принимают и возвращают значения типа `float`:

Группа	Функции	Назначение
Тригонометрия	`sin`, `cos`, `tan`	Угол в радианах
Обратная тригонометрия	`asin`, `acos`, `atan`, `atan2`	У `atan2(y, x)` важен порядок аргументов
Гиперболические функции	`sinh`, `cosh`, `tanh`	Гиперболические преобразования
Логарифмы	`log`, `log10`	Натуральный и десятичный логарифмы
Степень и корень	`pow`, `sqrt`	Степень и квадратный корень
Округление	`ceil`, `floor`, `abs`, `fraction`	Округление вверх, вниз, модуль, дробная часть

Сравнение с допуском	`closeTo`	Перегрузки для `float` и `double`
Представление IEEE	`fpToIEEE`, `fpFromIEEE`	Битовое представление `float`/`double`

Вызов `sin(float(angle))` явно показывает переход от `double` к одинарной точности. Наличие `double` в языке не означает двойной точности каждой библиотечной функции. `closeTo` имеет форму `closeTo(a, b, epsilon)`; собственный критерий погрешности можно оформить отдельной функцией.

8.2 Строки

Для `string` доступны присваивание, сравнение, объединение, индексирование и следующие методы:

Метод	Назначение
`length()`, `isEmpty()`	Длина и проверка пустоты
`resize(uint length)`	Изменение длины
`substr(uint start = 0, int count = -1)`	Подстрока
`findFirst(const string &in text, uint start = 0)`	Первое совпадение; отсутствие обозначается -1
`findLast(const string &in text, int start = -1)`	Последнее совпадение
`insert(uint position, const string &in text)`	Вставка
`erase(uint position, int count = -1)`	Удаление фрагмента

Длина и позиции измеряются в байтах; для текста в UTF-8 число байтов может отличаться от числа отображаемых символов.

Глобальные функции форматирования и разбора:

```
string formatInt(int64 value, const string &in options = "", uint width = 0);
string formatUInt(uint64 value, const string &in options = "", uint width = 0);
string formatFloat(double value, const string &in options = "",
                   uint width = 0, uint precision = 0);
int64 parseInt(const string &in text, uint base = 10, uint &out byteCount = 0);
uint64 parseUInt(const string &in text, uint base = 10, uint &out byteCount = 0);
double parseFloat(const string &in text, uint &out byteCount = 0);
```

Это справочные сигнатуры, вставлять их как прототипы не требуется. `byteCount` позволяет проверить, какая часть строки разобрана. Методы `split` и `join` не поддерживаются.

8.3 Массивы

Массив объявляется как `array<T>`; разрешена также форма `T[]`. Индексация начинается с нуля, допустимый индекс удовлетворяет `index < values.length()`.

```
array<double> values = {1.0, 2.0, 3.0};
values.insertLast(4.0);
double first = values[0];
```

Метод	Назначение
`length()`, `isEmpty()`	Число элементов и проверка пустоты
`resize(uint length)`, `reserve(uint length)`	Размер и резервирование памяти

`insertLast(value)`, `insertAt(index, value)`	Добавление
`removeLast()`, `removeAt(index)`	Удаление
`sortAsc()`, `sortDesc()`	Сортировка при наличии сравнения
`reverse()`	Обращение порядка
`find(value)`	Поиск при наличии равенства; отсутствие обозначается -1

При изменении структуры массива во время обхода необходимо учитывать изменение индексов. Выход за границы приводит к исключению скрипта.

8.4 Дата и время

`datetime` предоставляет конструктор текущего времени и конструктор `datetime(year, month, day, hour, minute, second)`; последние три аргумента могут отсутствовать. Доступны свойства для чтения `year`, `month`, `day`, `hour`, `minute`, `second` и методы:

```
bool setDate(uint year, uint month, uint day);
bool setTime(uint hour, uint minute, uint second);
```

Прибавление и вычитание `int64` трактуют аргумент как число секунд. Разность двух `datetime` имеет тип `int64`. Конструктор без параметров даёт текущее время по UTC. Для отображения местного времени следует учитывать часовой пояс. Календарное время не следует использовать для точного измерения интервала между вызовами программы.

8.5 Файлы

Тип `file` предоставляет ввод и вывод:

Метод	**Назначение**
`open(path, mode)`	`r` — чтение, `w` — запись с заменой, `a` — добавление
`close()`	Закрытие
`getSize()`, `getPos()`, `setPos(position)`, `movePos(offset)`	Размер и позиция
`isEndOfFile()`	Конец файла
`readString(count)`, `readLine()`	Чтение текста
`writeString(text)`	Запись текста
`readInt(bytes)`, `readUInt(bytes)`, `readFloat()`, `readDouble()`	Чтение двоичных чисел
`writeInt(value, bytes)`, `writeUInt(value, bytes)`, `writeFloat(value)`, `writeDouble(value)`	Запись двоичных чисел

Успешный `open` возвращает 0; отрицательный результат означает ошибку. Для однозначного выбора файла рекомендуется указывать полный путь, например `D:/ProtoEMU/input.txt`. Необходимые каталоги следует создать заранее. Файловые операции увеличивают время выполнения программы; пример 10.8 прекращает чтение после первого успешного открытия.

9 Порядок использования

9.1 Создание и запуск программы

1. Создать источники и назначить допустимые уникальные имена.
2. Добавить программу в группу «Программы» и задать её имя.
3. Поместить первой `void ИмяПрограммы()`; дополнительные объявления расположить после неё.
4. Сохранить текст в редакторе и проверить документ.
5. При необходимости связать обнаруженные глобальные переменные с виджетами.
6. Задать `ProcessInterval`, сохранить документ и запустить эмуляцию.
7. Контролировать сообщения и фактические значения сигналов.

Для скрипта виджета применяется `void ИмяВиджетаOnTimer()` в свойстве `ScriptOnTimer` и включается `ScriptOnTimerEnabled`.

9.2 Изменение программы

Перед изменением текста остановите эмуляцию. После редактирования сохраните текст, выполните проверку документа и устраните найденные ошибки.

Если изменено имя программы, приведите в соответствие имя её основной функции. Если переименован источник или переменная, исправьте все обращения к ним в тексте. Для изменения типа или начального значения переменной отредактируйте её объявление. После успешной проверки сохраните документ и повторите запуск.

10 Прикладные примеры

Каждый пример является самостоятельным текстом программы. Имя узла соответствует первой функции. Примеры с `Register1` и `Register2` предполагают включённые регистры с такими именами; объявлять их в тексте программы не требуется.

Тип, диапазон и функция записи `Register2` должны соответствовать результату алгоритма. При совместном использовании примеров в одном проекте необходимо согласовать имена глобальных переменных и вспомогательных функций.

10.1 Линейное преобразование с ограничением

Имя программы: `ScaleSignal`. Вход масштабируется и ограничивается диапазоном UInt16. Исходные числовые данные предполагаются конечными.

```
void ScaleSignal()
{
    if (Register1.IsError() || !Register1.IsNumber()) return;

    double input = Register1.ToDouble();
    double output = Clamp(input * Gain + Offset, 0.0, 65535.0);
    Register2 = int(output);
}

double Gain = 2.0;
double Offset = 10.0;
```

```
double Clamp(double value, double lower, double upper)
{
    if (value < lower) return lower;
    if (value > upper) return upper;
    return value;
}
```

При входе 20 выход равен 50. Преобразование к `int` выполняется после ограничения и отбрасывает дробную часть неотрицательного результата.

10.2 Пороговый автомат с гистерезисом

Имя программы: `ThresholdControl`. Выход принимает 0 или 1. Между порогами сохраняется предыдущее состояние.

```
void ThresholdControl()
{
    if (Register1.IsError() || !Register1.IsNumber()) return;

    double input = Register1.ToDouble();
    if (input >= UpperThreshold) Active = true;
    else if (input <= LowerThreshold) Active = false;

    Register2 = Active ? 1 : 0;
}

bool Active = false;
double LowerThreshold = 45.0;
double UpperThreshold = 55.0;
```

Требуется `LowerThreshold < UpperThreshold`. Глобальную `Active` можно дополнительно привязать к виджету.

10.3 Счётчик и выходной параметр

Имя программы: `CycleCounter`. Регистры не требуются; `Counter`, `Ratio` и `DivisionValid` предоставляют данные для виджетов.

```
void CycleCounter()
{
    if (Counter < 1000000) Counter++;
    else Counter = 0;

    double result = 0.0;
    DivisionValid = SafeDivide(double(Counter), 10.0, result);
    Ratio = result;
}

int Counter = 0;
double Ratio = 0.0;
bool DivisionValid = false;

bool SafeDivide(double numerator, double denominator, double &out result)
{
    if (denominator == 0.0)
    {
        result = 0.0;
        return false;
```

```
    }
    result = numerator / denominator;
    return true;
}
```

Счётчик отражает число вызовов, а не точное время в секундах.

10.4 Битовое поле

Имя программы: `DecodeBits`. Младшие четыре бита входа копируются на выход; бит с индексом 2 дополнительно отображается через `Bit2Set`.

```
void DecodeBits()
{
    if (Register1.IsError() || !Register1.IsNumber()) return;

    uint word = uint(Register1.ToInt()) & 0xFFFF;
    Bit2Set = (word & (uint(1) << 2)) != 0;
    Register2 = int(word & 0x000F);
}

bool Bit2Set = false;
```

Биты индексируются с нуля. Для входа 13 выход равен 13, `Bit2Set` равен `true`.

10.5 Скользящее среднее

Имя программы: `MovingAverage`. Окно содержит пять последних допустимых отсчётов. В начальной фазе среднее вычисляется по накопленным отсчётам.

```
void MovingAverage()
{
    if (Register1.IsError() || !Register1.IsNumber()) return;

    Samples.insertLast(Register1.ToDouble());
    if (Samples.length() > WindowSize) Samples.removeAt(0);

    double sum = 0.0;
    for (uint i = 0; i < Samples.length(); ++i)
    {
        sum += Samples[i];
    }
    Filtered = sum / double(Samples.length());
}

array<double> Samples;
uint WindowSize = 5;
double Filtered = 0.0;
```

Массив хранит внутреннее состояние; на виджет выводится `Filtered`. Требуется `WindowSize > 0`. Для больших окон предпочтителен кольцевой буфер, избегающий сдвига элементов при удалении первого отсчёта.

10.6 Генератор синусоидального сигнала

Имя программы: `GenerateSine`. `Signal` предназначена для виджета; `StepSeconds` — модельное приращение времени за вызов.

```
void GenerateSine()
{
    Signal = Center + Amplitude * double(sin(Phase));
    Phase += 2.0f * Pi * Frequency * StepSeconds;
    if (Phase >= 2.0f * Pi) Phase = 0.0f;
}

const float Pi = 3.14159265f;
float Phase = 0.0f;
float Frequency = 0.1f;
float StepSeconds = 0.5f;
double Amplitude = 10.0;
double Center = 20.0;
double Signal = 0.0;
```

Расчётный период соответствует 20 вызовам. Его длительность в секундах зависит от фактического интервала между вызовами. Простая коррекция фазы рассчитана на приведённый небольшой шаг; для произвольных параметров необходима нормализация с сохранением остатка.

10.7 Объектное состояние

Имя программы: `Accumulate`. Объект хранит сумму между циклами, а `Total` предоставляет результат виджету.

```
void Accumulate()
{
    Accumulator@ reference = @State;
    if (reference is null) return;

    reference.Add(1.0);
    Total = reference.GetTotal();
}

class Accumulator
{
    double total = 0.0;

    void Add(double value)
    {
        total += value;
    }

    double GetTotal() const
    {
        return total;
    }
}

Accumulator State;
double Total = 0.0;
```

Для наблюдения состояния объекта выделена скалярная переменная `Total`.

10.8 Строки, календарное время и файл

Имя программы: `ReadConfiguration`. Пример читает строку из существующего `D:/ProtoEMU/input.txt` и формирует сообщение. После первого успешного открытия повторное чтение прекращается.

```
void ReadConfiguration()
{
    if (Loaded) return;

    datetime now;
    Stamp = formatUInt(now.year) + "-" + formatUInt(now.month)
          + "-" + formatUInt(now.day);

    file input;
    OpenResult = input.open("D:/ProtoEMU/input.txt", "r");
    if (OpenResult < 0)
    {
        StatusText = "Cannot open input file";
        return;
    }

    string line = input.readLine();
    input.close();
    StatusText = Stamp + ": " + line;
    Loaded = true;
}

bool Loaded = false;
int OpenResult = -1;
string Stamp = "";
string StatusText = "";
```

Файл подготавливается пользователем. При отказе открытия попытка повторяется в следующем цикле. Содержимое не исполняется как код; файл и каталог не изменяются. `StatusText` можно связать с текстовым виджетом.

11 Диагностика и ограничения

11.1 Проверка документа

Причина ошибки	Исправление
Не найден источник	Проверить имя источника и убедиться, что он включён
Повторное глобальное объявление	Проверить уникальность имён в программах проекта
Недоступный метод `Value`	Выбрать метод из раздела 7
Неподходящий тип аргумента	Выполнить явное преобразование
Неверная форма основной функции	Поместить первой `void ИмяПрограммы()` без параметров
Использована недоступная функция C++	Выбрать функцию или метод из этого руководства

При проверке документа сообщения указывают ошибку и её положение в тексте программы. Найдите соответствующую строку, внесите исправление и повторите проверку. Сохранение текста само по себе не подтверждает отсутствие ошибок.

11.2 Исполнение

Деление на ноль, выход за границы и доступ через `null` могут прервать функцию исключением. Проверки должны предшествовать соответствующим операциям.

Основная функция должна завершать один ограниченный шаг обработки. Бесконечный цикл может помешать обработке других программ и остановке эмуляции. Для ожидания события сохраняйте состояние между вызовами и проверяйте условие при следующем вызове основной функции.

При ошибке выполнения проверьте сообщения приложения и текущие значения источников. Перед исправлением программы остановите эмуляцию.

11.3 Применимость примеров

Примеры предполагают корректную конфигурацию транспорта, допустимые диапазоны и согласованные имена. `IsNumber()` не заменяет прикладную проверку конечности, диапазона и единиц измерения.

Перед использованием примера замените имена источников на имена вашего проекта, проверьте настройки регистров и при необходимости измените коэффициенты, пороги и размер окна. Затем выполните проверку документа и сопоставьте результат с ожидаемыми значениями. Для файлового примера предварительно создайте входной файл и укажите его полный путь.