МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра компьютерных технологий

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Синицына Даниила Владимировича

Тема работы:

РАЗРАБОТКА ЯДРА ОБЛАЧНОГО IDE ПРОЦЕСС-ОРИЕНТИРОВАННОГО РАСШИРЕНИЯ ЯЗЫКА СИ

«К защите допущен»
Заведующий кафедрой,
д.т.н, доцент
Зюбин В.Е. /……..
(ФИО) / (подпись)
«……»……20…г.

Руководитель ВКР
д.т.н, доцент
зав. каф. КТ ФИТ НГУ
Зюбин В.Е. /……...
(ФИО) / (подпись)
«……»……20…г.

Соруководитель ВКР
ассистент КИСТ ВКИ,
соруководитель
Башев В.И. /……..
(ФИО) / (подпись)
«……»……20…г.

Новосибирск, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра компьютерных технологий
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Зюбин В.Е.
(фамилия, И., О.)

…….
(подпись)

«……» ..... 20…г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Синицыну Даниилу Владимировичу, группы 19207
(фамилия, имя, отчество, номер группы)

Тема «Разработка ядра облачного IDE процесс-ориентированного расширения языка Си»
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от…..№……
Срок сдачи студентом готовой работы…..... 20.. г.
Исходные данные (или цель работы): унификация языков Reflex и Industrial-C и разработка ядра web среды разработки для полученного языкового средства
Структурные части работы: анализ предметной области, выбор способа унификации языков, описание синтаксиса, выбор средств реализации ядра, практическая реализация, экспериментальное исследование на практической задаче.
Консультанты по разделам ВКР отсутствуют

Руководитель ВКР
зав. каф. КТ ФИТ НГУ
д.т.н., доцент
Зюбин В.Е. /……..
(ФИ О) / (подпись)
«…» ..... 20…г.

Задание принял к исполнению
Синицын Д.В. /……..
(ФИО студента) / (подпись)
«…» ..... 20…г.

Соруководитель ВКР
ассистент КИСТ ВКИ, соруководитель
Башев В.И. /……..
(ФИ О) / (подпись)
«…» ..... 20…г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4
Глава 1. Анализ предметной области, предъявление требований к синтаксису унифицированного языка 7
1.1 Предметная область 7
1.2 Анализ специфики языка Reflex, а также его преимуществ и недостатков 8
1.3 Анализ специфики языка Industrial-C, а также его преимуществ и недостатков 8
1.4 Анализ различий и общих черт в синтаксисе языков Industrial-C и Reflex 9
1.5 Требования к разрабатываемому инструментарию 11
1.6 Основные выводы главы 12
Глава 2. Язык Reflex 14
2.1 Общие сведения о синтаксисе языка 14
2.2 Константы, перечисления, порты ввода/вывода, векторы, регистры, биты 14
2.3 Гиперпроцессы, процессы и их состояния. Локальные и разделяемые переменные 15
2.4 Процесс-ориентированные конструкции языка 16
2.5 Структура программы 18
2.6 Семантика конструкций языка Reflex. Правила трансляции в язык Си 19
2.7 Выбор подхода для обеспечения кроссплатформенности 27
2.8 Основные выводы по главе 28
Глава 3. Выбор стека технологий и практическая реализация ядра 29
3.1 Подбор стека технологий для практической реализации 29
3.2 Архитектура разрабатываемой системы 31
3.3 Структура модулей системы 33
3.4 Структура генерируемых файлов при генерации кода на языке Си 34
3.5 Генерация кода 35
3.6 Обеспечение кроссплатформенности системы 36
3.7 Реализация модуля выбора целевой платформы 36
3.8 Экспериментальное исследование на практической задаче 37
3.9 Основные выводы по главе 38
Заключение 39
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 41
Приложение А. Грамматика языка Reflex v2.1 в нотации XText 43

ВВЕДЕНИЕ

Использование языков общего назначения для программирования цифровых систем управления ведет приводит к увеличению сложности алгоритмов и архитектуры программ, что ведет к росту затрат на разработку и усложняет процесс их поддержки.

Для решения данной проблемы в Институте автоматики и электрометрии СО РАН был реализован специализированный язык программирования Reflex [1], а также ide RIDE для данного языка. Кроме того, для подобных целей был разработан и реализован язык Industrial-C, который ориентирован на программирование встраиваемых систем на базе архитектуры AVR. Данные языки программирования уже долгое время используются в практических задачах и успели зарекомендовать себя, как простые, удобные и эффективные решения [2]. Тем не менее, оба языка имеют свои недостатки, которые не позволяют использовать их при решении некоторых задач. Так, Reflex не поддерживает работу с периферией контроллера, что может стать проблемой при разработке ПО для некоторых типов устройств. Industrial-C не обладает кроссплатформенностью, что может усложнить его использование в проектах, требующих переносимости между различными операционными системами и архитектурами процессоров. Однако и Reflex, и IndustrialC ориентированы на программирование встраиваемых систем, используют модель гиперпроцесса для реализации алгоритмов, а также они имеют очень схожую семантику. Кроме того, недостатки одного покрываются преимуществами другого. Вследствие вышесказанного, было принято решение провести унификацию данных языков, в целях получения инструмента, который, как и Reflex позволит вести разработку встраиваемых систем без привязки к конкретной платформе, а также будет иметь низкий порог вхождения для разработчиков и, как и IndustrialC, позволит в удобной форме работать с периферией.

Цель данной работы – унификация языков Reflex/IndustrialC и разработка ядра облачной IDE для унифицированного языка (Reflex 2.1)

Для достижения поставленной цели решались следующие **задачи**:

1. анализ концепции процесс-ориентированного программирования;
2. анализ недостатков и преимуществ языков Reflex и Industrial-C, необходимый для понимания их различий и выявления возможности их унификации;
3. анализ инструментов для реализации ядра web-IDE;
4. составление требований к разрабатываемому языку и ядру облачной IDE;
5. разработка синтаксиса и семантики языка, правил перевода в язык Си;
6. разработка механизмов обеспечения платформонезависимости;
7. разработка архитектуры интегрированной среды разработки;

8. выбор инструментов для реализации ядра облачной IDE;
9. реализация ядра и описание синтаксиса языка Reflex 2.1;
10. экспериментальное исследование языка на тестовой задаче.

Научная новизна:

Разработан синтаксис процесс-ориентированного языка Reflex 2.1, обладающий кроссплатформенностью с точки зрения архитектуры разрабатываемых встраиваемых систем. Кроме того, в Reflex 2.1 были введены средства для обработки прерываний в рамках процесс-ориентированной модели гиперпроцесса. Эти новые возможности позволят повысить эффективность и универсальность языка Reflex при разработке систем управления.

Практическая ценность:

В результате проделанной работы было получено средство для разработки цифровых встраиваемых систем управления в виде языка программирования Reflex 2.1 и ядра для облачной ide на базе Eclipse/Ride/Theia, в которой в перспективе будет возможна комфортная разработка программных средств на языке Reflex 2.1.

Разработанный язык Reflex 2.1 обладает понятным и простым синтаксисом для обеспечения низкого порога вхождения для новых разработчиков, кроссплатформенностью для разработки программных средств не только под AVR архитектуру, как например в Industrial-C, но и под другие архитектуры микроконтроллеров. Более того, он обеспечивает простой механизм обработки прерываний, что позволяет внедрить быструю обработку внешних событий в рамках цикличности событийной модели процесс-ориентированных языков. Помимо этого, разрабатываемая среда разработки является облачной, что обеспечивает возможность внедрения механизма коллективной разработки в рамках одного проекта.

Все это делает Reflex 2.1 удобным и эффективным средством для разработки цифровых встраиваемых систем управления.

Структура работы:

Работа состоит из введения, 3 глав и заключения. Объем работы – 49 страниц.

В первой главе на основе литературных данных описывается предметная область и приводится анализ аналогов среди языков программирования, ориентированных на разработку ПО для встраиваемых систем, выявляются их преимущества и недостатки, а также предъявляются требования к разрабатываемому языку. Во второй главе описывается синтаксис языка Reflex 2.1 с учетом введенных корректировок и его семантика в виде правил трансляции в язык Си, а также предлагаемый способ обеспечения платформонезависимости. В третьей главе обосновывается выбор стека технологий, описывается практическая

реализация ядра web среды разработки Reflex 2.1 и генератора Си-кода, приводятся результаты эмпирического исследования языка на модельной задаче.

Глава 1. Анализ предметной области, предъявление требований к синтаксису унифицированного языка

1.1 Предметная область

1.1.1 Кибер-физические системы и ПЛК

Кибер-физическая система – это система, основанная на интеграции вычислений с физическими процессами [3]. На сегодня системы на базе микроконтроллеров широко используются в робототехнике [4], медицине [5] и других областях. Обычно такие системы состоят из объектов управления, датчиков, реагирующих на некоторые внешние события и отправляющих сигналы, а также центральный управляющий объект, который, чаще всего, представляет из себя программируемый логический контроллер (в дальнейшем ПЛК). ПЛК делегирует на себя управление всеми объектами управления в системе и обрабатывает события, происходящие во внешней среде посредством датчиков, и, в зависимости от природы этих событий, корректирует поведение объектов управления. Алгоритмам управления такими системами присущ ряд свойств, отличающий их от алгоритмов для ПО общего назначения [6]:

1. Открытость;
2. Событийность;
3. Неопределенная продолжительность работы;
4. Синхронизм – обеспечение синхронности выполнения алгоритма с протекающими на объекте управления физическими процессами. Необходимо понимать, что у управляющего устройства есть инерция, то есть некоторое время отклика на событие;
5. Логический параллелизм. Это свойство означает, что в окружающей среде физические процессы происходят независимо друг от друга. Алгоритм управления также должен структурно отражать параллелизм данных процессов.

С каждым днем количество объектов управления в кибер-физических системах в сфере промышленности непрерывно растет, в связи с чем растет и возрастает сложность разработки программного обеспечения для ПЛК в силу того, что растет сложность алгоритмов, а как следствие и объем кода, требуемого для обработки этих алгоритмов. В связи с этим необходимо иметь удобное средство для разработки программного обеспечения для контроллеров.

1.1.2 Процесс-ориентированный подход

Процесс-ориентированный подход давно зарекомендовал себя в качестве эффективного представления кибер-физической системы, как некоторой математической

модели. Основа данной парадигмы заключается в представлении всей системы, как некоторого гиперпроцесса, то есть совокупности процессов. При этом каждый процесс в данной идеологии представляет из себя расширенный конечный автомат и соответствует некоторому реальному объекту управления. Расширение конечного автомата заключается в добавлении остановочных состояний, средств синхронизации и возможность контроля состояний других процессов в автомат. Модель гиперпроцесса обеспечивает событийность, логический параллелизм и синхронизм алгоритмов управления, удовлетворяя всем требованиям задач автоматизации [7]. Для данного алгоритма присуща неопределенность продолжительности работы, которая обуславливается тем, что вышеописанный цикл имеет неограниченное число итераций.

Данная модель на текущее время уже имеет практические реализации, в виде разработанных в институте автоматики и электрометрии СО РАН языков программирования Reflex и Industrial-C.

1.2 Анализ специфики языка Reflex, а также его преимуществ и недостатков

На сегодняшний день язык Reflex является одним из активно развивающихся процесс-ориентированных языков [8]. Reflex представляет собой диалект языка Си.

Основной концепцией языка Reflex является модель гиперпроцесса, которая позволяет реализовать взаимодействие с множеством независимо протекающих на объекте управления процессов. Программа на данном языке представляет из себя упорядоченное множество процессов, которые по своей сути являются конечными автоматами с некоторым набором состояний и правилами перехода между этими состояниями. Reflex позволяет работать с сигналами, привязанными к физическим портам управляющего устройства [9].

К основным преимуществам Reflex относятся его кроссплатформенность, а также удобный синтаксис, что снижает порог входа для работы с данным языком у новых разработчиков. Однако у языка есть и некоторые недостатки. Например, отсутствие возможности работы с прерываниями. Также, при работе с архитектурой AVR (например, Arduino), отсутствует возможность работы с векторами и регистрами микроконтроллера.

1.3 Анализ специфики языка Industrial-C, а также его преимуществ и недостатков

Язык Industrial-C, как и Reflex был разработан в Институте автоматики и электрометрии СО РАН, как процесс-ориентированный язык, предназначенный для программирования систем управления.

В отличие от Reflex, данный язык рассчитан на работу с конкретной архитектурой микроконтроллеров, а именно с AVR. В следствие этого возможности языка предполагают обработку прерываний, а также работу с периферией в виде регистров, битов и векторов микроконтроллера. Кроме того, язык поддерживает практически весь синтаксис языка Си с сохранением семантики конструкций, за исключением работы с памятью через указатели. Из языка Reflex были заимствованы конструкции описания процессов и состояний, операций тайм-аутов и выражений для взаимодействия между процессами. Однако от Reflex язык отличается тем, что описывается не множеством процессов в рамках одного гиперпроцесса, а множеством гиперпроцессов.

К основным преимуществам данного языка относятся описанные выше возможности работы с периферией и прерываниями. Недостатком данного языка является высокий порог входа для новых разработчиков. Язык включает в себя синтаксис языка Си и работу с периферией, что требует определенного уровня знаний и навыков. Это означает, что для проектирования программы необходимо вкладывать больше времени и усилий. Кроме того, разработанный язык программирования не является кроссплатформенным, поскольку он был разработан для работы конкретно с архитектурой AVR.

1.4 Анализ различий и общих черт в синтаксисе языков Industrial-C и Reflex

1.4.1 Типы данных

Типы данных в обоих языках по большей части заимствованы из языка Си. Типы данных языков Reflex и Industrial-C приведены ниже в таблице 1.

Таблица 1 – Типы данных языков Reflex, Industrial-C, их семантическое соответствие в языке Си и описание

Тип данных языка Reflex	Тип данных языка Industrial-C	Тип данных языка Си при трансляции из Reflex	Тип данных языка Си при трансляции из Industrial-C	Описание
bool	bool	Char (диапазон значений 1-2)	_Bool из современных стандартов языка Си	Логическое

Продолжение таблицы 1

Int8, int16, int32, int64	char, short, int, long	Определения взяты из стандартной библиотеки stdint.h языка Си	char, short, int, long	Целое
Uint8, uint16, uint32, uint64	unsigned <любой целочисленный тип>		unsigned <аналогичный целочисленный тип>	Беззнаковое целое
Float, double	Float, double	Float, double	Float, double	Дробное с плавающей точкой
time	--	Uint32_t	--	Время
--	void	--	Void	Пустота
--	char	--	char	Символ
Input, output	--	int8_t	--	Системные порты

Из данных, представленных в таблице 1, можно сделать следующие выводы:

1. Логические и целочисленные (знаковые и беззнаковые) типы в языках Reflex и Industrial-C идентичны семантически, но различаются правила их трансляции в язык Си, что не создает проблем при унификации этих языков;
2. Типы данных с плавающей точкой абсолютно идентичны в обоих языках, как семантически, так и с точки зрения трансляции;
3. Язык Reflex содержит уникальный относительно Си и Industrial-C тип данных – time. А также типы input и output, которые семантически обозначают системные порты и при трансляции отображаются в 8-битные целочисленные переменные;
4. Язык Industrial-C содержит семантически уникальные относительно Reflex типы, а именно void и char.

1.4.2 Выражения

Утверждения if-else, switch-case, арифметические и логические выражения присутствуют как в языке Reflex, так и Industrial-C и заимствуются синтаксически и семантически из Си. Однако в отличие от Reflex, Industrial-C также использует конструкции языка Си, такие как for и return. Стоит отметить, что конструкция for не всегда имеет смысл, так как ее функциональность можно реализовать через зацикленное состояние с проверкой на выход.

Оба языка реализуют такие конструкции, как start/stop process, set state, process in state active/inactive, timeout их семантика в данных языках абсолютно идентична, может отличаться только синтаксис (например, в Reflex после ключевого слова timeout должен стоять литерал,

обозначающий время, то есть типа time, а в Industrial-C время в мс), что не является трудностью для унификации языков.

В языке Reflex есть уникальные конструкции относительно Industrial-C, такие как set next state, error process, reset timer, restart, stop, error, process in state error/stop.

В Industrial-C присутствует возможность создания гиперпроцессов и управления ими, это достигается введением таких конструкций как: start/stop hyperprocess. Для возможности запретить прерывания на момент исполнения кода в Industrial-C было добавлено ключевое слово atomic.

1.4.3 Структура программы

Программа на языке Industrial-C – набор объявлений:

- Переменных;
- Векторов, регистров, бит;
- Гиперпроцессов;
- Процессов.

Программа на языке Reflex описывается следующим образом:

1. Объявляется основное тело программы ключевым словом program;
2. Объявляются глобальные переменные, константы, перечисления и порты;
3. Объявляются процессы и их состояния.

Таким образом, исключая векторы, регистры и биты, а также ключевое слово program, программа на языке Reflex, является по сути программой на языке Industrial-C с одним единственным фоновым гиперпроцессом, в рамках которого работаю объявленные процессы.

1.4.4 Дополнительные возможности

Industrial-C предлагает возможность реализовывать вставки на языке Си, либо между $ и \n (однострочные вставки), либо между двумя ограничителями $$...$$. Однако использование этой возможности нежелательно и, чаще всего, при решении задач можно обойтись средствами языка Industrial-C.

Reflex предоставляет возможность аннотаций для задания дополнительной информации, необходимой кодогенератору.

1.5 Требования к разрабатываемому инструментарию

Исходя из анализа языков Reflex и Industrial-C, можно сформулировать следующие требования к разрабатываемому языку Reflex 2.1:

1. Язык должен являться процесс-ориентированным языком, реализуя механизм гиперпроцесса и обладать всеми вытекающими из процесс-ориентированной парадигмы свойствами;
2. Язык должен иметь Си-подобный синтаксис на основе существующих языков Reflex и Industrial-C;
3. Генерация исполняемого кода должна быть реализована через трансляцию в язык Си;
4. Должен быть предусмотрен механизм для обеспечения платформонезависимости, то есть возможность вести разработку под различные архитектуры микроконтроллеров;
5. В случае работы с высокотехнологичными микроконтроллерами, как, например, контроллеры AVR архитектуры, в языке должны присутствовать конструкции, позволяющие вести работу с периферией контроллера и обрабатывать прерывания, происходящие на портах контроллера;
6. Язык должен быть поддержан интегрированной средой разработки с возможностью удаленного доступа.

1.6 Основные выводы главы

Одними из ключевых различий между языками Reflex и Industrial-C можно назвать:

- Возможность в Industrial-C работать с прерываниями посредством объявления и привязки к гиперпроцессам векторов, битов и регистров;
- Описание в программе нескольких дополнительных гиперпроцессов в случае Industrial-C;
- Аннотации в Reflex, вставки на языке Си в Industrial-C.

Идея описания гиперпроцессов, активирующихся по прерыванию, существующая в языке Industrial-C можем быть перенесена и в язык Reflex. Данный метод позволит внедрить механизм обработки прерываний без отклонения от процесс-ориентированного подхода.

Вставки на языке Си являются избыточной возможностью, так как одним из ключевых требований является простота синтаксиса и семантики языка. Тем не менее, для повышения гибкости языка Reflex данный механизм также было решено перенести из IndustrialC без изменения синтаксиса и семантики.

Таким образом, так как в языке Reflex уже есть все необходимое для соблюдения предъявляемых функциональных требований, за исключением возможности обработки прерываний и работы с периферией управляющего устройства, было принято решение использовать Reflex в качестве основы для создания нового языка. В рамках данной работы Reflex должен быть дополнен необходимым функционалом для корректной обработки

прерываний и работы с периферией устройств, в виде векторов, регистров и битов контроллера, при этом кроссплатформенность должна сохранятся.

Глава 2. Язык Reflex

В данной главе будут рассмотрены основные конструкции языка Reflex 2.1. Описан их синтаксис и семантика на примере правил трансляции данных выражений в язык Си.

2.1 Общие сведения о синтаксисе языка

Синтаксис языка Reflex 2.1 во многом схож с описанным в [8] Reflex 2.0. Однако у некоторых конструкций были изменены правила трансляции в язык Си (управляющие команды, описания процессов и состояний, а также функции работы с интервалами времени). Также были добавлены новые специфические для данной версии конструкции, применяемые в описании программ, портируемых на платформу Arduino (векторы, биты, регистры, гиперпроцессы и прерывания).

2.2 Константы, перечисления, порты ввода/вывода, векторы, регистры, биты

Одной из ключевых констант в языке можно считать период активизации процессов. Для описания данной константы используется специальное ключевое слово ***clock <интервал>***. Для временных интервалов введены специализированные константы вида **0t** <***число_дней***> **d** <***число_часов***> **h** <***число_минут***> **m** <***число_секунд***> **s** <***число_миллисекунд***> **ms**. Но также возможно использование целочисленной константы. В этом случае период активизации процессов будет равен количеству миллисекунд, равному указанному числу. Примеры показаны ниже в листинге 1.

clock 0t1m50s; *// период активизации равен 1 минуте 50 секундам*

clock 120; *// период активизации равен 120 миллисекундам*

Для описания констант используется конструкция ***const*** <*тип*> <*имя*> = <*значение*>. В качестве типа константы может быть использован любой тип данных, присущий языку Reflex, а именно: логический (**bool**), целочисленный (**int8**, **int16**, **int32**, **int64**, **uint8**, **uint16**, **uint32**, **uint64**), вещественный (**float**, **double**), а также временной (**time**). Алфавит для описания имени константы состоит из латинских символов, цифр и знака подчеркивания. При этом нельзя для именования использовать зарезервированные слова языка Си и Reflex, а также идентификаторы, состоящие исключительно из цифр. В качестве значения должен выступать литерал, тип которого соответствует указанному типу константы.

Для описания портов ввода/вывода используется <***input/output***> <*имя*> <*адрес1*> <*адрес2*> <разрядность>. Если декларация порта начинается с ключевого слова **input**, то порт считается входным, то есть используется для считывания значений из портов целевого

устройства. При использовании слова **output**, считается, что порт выходной, соответственно используется для записи значения в порт. Именование портов происходит по тем же правилам, что и констант. Адреса порта – целочисленные значения, определяющие системный адрес порта. Разрядность – целое число, равное либо 8, либо 16.

Перечисления в Reflex имеют точно такую же структуру и семантику, что и в языке Си.

В языке есть возможность описывать глобальные переменные, доступные всем процессам. Для их декларации в программе используется либо конструкция <*тип*> <*имя*>, либо <*тип*> <*имя*> = <имя порта> [<номер бита>]. Такие переменные не инициализируются при начале работы программы, при этом их может использовать любой процесс. В случае указания знака равно и имени порта, данная переменная отображается на порт, при этом если далее в квадратных скобках указывается еще номер бита, то отображается не весь порт, а значение в конкретном бите порта.

В теле процессов, в языке Reflex поддерживается возможность указания вектора, регистра и бита для обработки прерываний. Если декларируется какая-либо из данных сущностей, то необходимо указать и две остальные. Вектор, регистр и бит декларируются ключевыми словами **vector**, **register** и **bit**, соответственно. После ключевого слова должен идти идентификатор соответствующего вектора, регистра или бита. В качестве идентификатора должно использоваться наименование, описанное в соответствующей спецификации целевого устройства, выбранного в редакторе кода при сборке приложения, в противном случае будет выброшена семантическая ошибка.

2.3 Гиперпроцессы, процессы и их состояния. Локальные и разделяемые переменные

Гиперпроцессы – это новая конструкция, введенная в рамках работы в язык Reflex. Создание нескольких гиперпроцессов необходимо для того, чтобы помимо фонового, было возможно описывать также гиперпроцессы, срабатывающие по прерыванию. Для описания гиперпроцесса в языке введено ключевое слово **hyperprocess**. Синтаксис следующий: **hyperprocess** <имя гиперпроцесса> {<тело гиперпроцесса>} или **hyperprocess** <имя гиперпроцесса> **interrupted** {**vector** <идентификатор вектора>**; register** <идентификатор регистра>**; bit** <идентификатор бита>**;**}. Первый вид конструкции используется для объявления фонового гиперпроцесса: он срабатывает по таймеру, указанному в теле программы. В программе может быть объявлен только один фоновый гиперпроцесс. Второй вид конструкции применяется для объявления, прерываемого гиперпроцесса, который срабатывает однократно по возникновении прерывания в случае, если прерывание разрешено (подробнее в 2.4.4 Прерывания.).

Для описания процессов в языке Reflex используется ключевое слово **process** <*имя процесса*> {<*тело процесса*>}. В теле процесса описываются локальные и разделяемые переменные, а также последовательно декларируются состояния процесса. Кроме того, язык позволяет определять векторы, регистры и биты, что позволяет включать или выключать прерывания на определенном векторе. Для включения или выключения прерывания на соответствующем векторе, необходимо описать соответствующий ему регистр и бит.

В языке существует два вида переменных: локальные и глобальные. Локальные переменные декларируются аналогично глобальным переменным в теле программы. Однако, в отличие от глобальных переменных, такие переменные декларируются в теле конкретного процесса, а значит и доступны исключительно внутри конкретного процесса, в котором они были объявлены. Разделяемые переменные объявляются с использованием ключевого слова **shared**. Такие переменные являются общими для нескольких процессов. При этом в программе обязательно должен быть процесс, предоставляющий данную переменную, в таком процессе переменная описывается конструкцией <*тип*> <*имя*> **shared**. Все остальные процессы считаются процессами, получающими переменную. Такие процессы обязаны декларировать такую переменную следующей конструкцией: **shared** <*имя переменной*> **from process** <*имя процесса*>. Фактически при этом и предоставляющий процесс, и получающий являются равноправными в использовании переменной.

Состояние процесса задается с помощью ключевого слова **state.** При этом используется следующая конструкция: **state** <*имя состояния*> {<*тело состояния*>}. Если состояние не приводит к переходу в другие состояния, то перед телом необходимо указать идентификатор "looped". Отсутствие данного идентификатора в таком случае будет считаться семантической ошибкой.

Тело состояния может включать в себя процесс-ориентированные управляющие процессами конструкции, конструкции проверки состояния процесса, тайм-ауты, базовые управляющие конструкции языка Си, такие как **switch case**, **if**. Конструкции циклов из Си, тернарные операторы, выражения для работы с указателями, выражения для работы с массивами, а также операторы **sizeof** и ‘,’ не поддерживаются языком Reflex, их использование приведет к синтаксической ошибке.

2.4 Процесс-ориентированные конструкции языка

Среди процесс-ориентированных конструкций языка Reflex можно выделить 3 категории.

2.4.1 Конструкции, управляющие процессами

Управляющие конструкции предназначены для управления состояниями процессов. В языке Reflex различают два вида таких конструкций:

- **set state** <*имя состояния*> и **set next state –** данный оператор переводит текущий процесс в указанное состояние. **set next state** переводит процесс в следующее по порядку состояние. Если текущее состояние было последним, среди перечисленных в теле процесса, то процесс переходит в начальное состояние;
- **start process** <*имя процесса*>, **stop process** <*имя процесса*> и **error process** <*имя процесса*> – этот оператор переводит указанные процессы в состояние START, STOP и ERROR, соответственно. Также может быть использована укороченная версия данных команд **restart, stop** и **error** для изменения состояния текущего процесса.

2.4.2 Конструкции проверки состояния процессов

В языке существует один вид конструкций для проверки нахождения процесса в конкретном состоянии. Данная конструкция имеет вид **process** <*имя процесса*> **in state** <*имя состояния*> и служит для проверки, что процесс находится в указанном состоянии. Конструкция служит для проверки, что указанный процесс находится в указанном состоянии, в таком случае данное выражение вернет логическое значение истины (**true**), в противном – лжи (**false**). Имя состояния может принимать одно из следующих значений: **inactive, active, error, stop**.

2.4.3 Конструкции работы с временными интервалами

Для работы с временными интервалами в языке присутствует две основные конструкции:

- **reset timer –** сбрасывает таймер текущего процесса;
- **timeout** <*интервал*> {<*последовательность выражений*>} – данную конструкцию можно использовать для того, чтобы выполнять блок кода внутри процесса по таймауту.

2.4.4 Прерывания

Для работы с прерываниями в язык добавлены 4 конструкции:

- **vector** <*идентификатор вектора*> **–** определение вектора;
- **register** <*идентификатор регистра*> **–** определение регистра;
- **bit** <*идентификатор бита*> **–** определение бита;
- **hyperprocess** <*идентификатор гиперпроцесса*> **interrupted {**<*тело гиперпроцесса*>**} –** данная конструкция определяет множество процессов, выполняющихся по

прерыванию. Стоит отметить, что в теле гиперпроцесса первые три строки должны содержать определения вектора, регистра и бита (именно в таком порядке). Вектор определяет вектор прерывания, по которому гиперпроцесс должен быть запущен, регистр и бит служат для механизма разрешения / запрета прерывания.

Важно учесть, что идентификаторы векторов, регистров и битов должны указываться в соответствии со спецификацией целевого устройства. Если на целевом устройстве архитектурно не заложены данные сущности, то их использование не приведет к ошибке, однако **interrupted** гиперпроцессы не будут исполнены.

2.4.5 Процесс-ориентированные конструкции для работы с гиперпроцессами

Для работы с прерываемыми гиперпроцессами в язык внедрены две новые конструкции:

- **start hyperprocess** <*идентификатор гиперпроцесса*> – данная конструкция служит для разрешения прерываний. Если данная команда не была применена к гиперпроцессу, то по умолчанию прерывания будут запрещены и гиперпроцесс не будет запущен по указанному в нем прерыванию. Если прерывания уже разрешены, то вызов команды ни к чему не приведет;
- **stop hyperprocess** <*идентификатор гиперпроцесса*> – данная конструкция служит для запрета прерываний.

2.5 Структура программы

Начиная с версии 2.1, программа на языке Reflex представляет собой непустое множество описаний гиперпроцессов, один из которых обязательно фоновый. Тело фонового гиперпроцесса начинается с описания периода активизации процессов, тело прерываемого гиперпроцесса – с описания вектора, регистра и бита прерываний. Затем в теле гиперпроцессов следует последовательное описание констант, портов, перечислений и глобальных переменных, которые доступны для использования в теле процессов. В конце тела гиперпроцесса последовательно описываются процессы. Пример программы на языке Reflex приведен ниже:

```
hyperprocess Program {
    //Пример описания периода активации процессов
    clock 0t50ms;
    //Пример входных/выходных портов
    input portA 2 2 8;
    output portB 2 4 8;
    //Пример констант
    const time SECOND = 0t1s;
    const bool ON = true;
```

```
        const bool OFF = false;
        int16 level;

        process Process1 {
            //Пример разделяемой переменной
            bool isOpen = portA[1] shared;
            state Init {
                start process Process2;
                stop;
            }
        }

        process Process2 {
            //Пример разделяемой переменной
            shared isOpen from process Process1;
            state Work {
                //Пример использования тайм-аута
                timeout 0t2s {
                    isOpen = true;
                    stop;
                }
            }
        }
    }
    hyperprocess Logging interrupted {
        //Пример описания вектора, регистра и бита прерываний для прерываемого гипер процесса
        vector vec;
        register reg;
        bit bt;

        output portB 2 4 8;

        process Process {
            state Log {
                portB = 1;
            }
        }
    }
```

Листинг 1 – Пример программы на языке Reflex 2.1

2.6 Семантика конструкций языка Reflex. Правила трансляции в язык Си

В данном пункте будет описана семантика всех конструкций языка Reflex на примере правил трансляции этих выражений в язык Си.

2.6.1 Трансляция программы на языке Reflex

Согласно процесс-ориентированному подходу, любая программа представляет из себя набор гиперпроцессов. В листинге 2 описано, как транслируется программа на языке Reflex в Си-код. Как можно заметить, метод main осуществляет вызов функции control_loop, которая является представлением фонового гиперпроцесса. Начинается выполнение с инициализации процессов, где первый объявленный процесс инициализируется состоянием START, все остальные – STOP. Далее вызываются функции init_time() и init_io(), которые предназначены для инициализации счетчиков времени и портов ввода/вывода соответственно. Затем в бесконечном цикле поочередно происходит:

1. Обновление счетчика времени;
2. Проверка наступления момента активизации процессов;
3. Если проверка, проведенная в пункте 2, успешна, то процесс переходит к обработке случаев, когда время выполнения алгоритма превышает ожидаемое. Затем осуществляется считывание переменных значений с портов ввода, поочередный запуск процессов и запись переменных в порты вывода.

Стоит отметить, что реализация функций init_time(), init_io(), а также функций записи и чтения значения на портах контроллера зависит от целевой платформы. Кроме того, перед методом main объявляется функция обработчик прерывания.

```
const uint8_t START = 0;
const uint8_t CONFIRMED = 253;
const uint8_t ERROR = 254;
const uint8_t STOP = 255;

uint8_t process1_state;
uint8_t process2_state;
uint8_t process3_state;
…
uint8_t processN_state;

uint32_t process1_time;
uint32_t process2_time;
uint32_t process3_time;
…
uint32_t processN_time;

enum process1_states {
      p1_state1 = START,
      p1_state2,
```

```
        …
        p1_stateN
}
enum process2_states {
 p2_state1 = START,
 p2_state2,
        …
 p2_stateN
}

…

enum processN_states {-\\-}


int main() {
            control_loop();
        }

void control_loop(void)    /* Control algorithm */
{
    process1_state = START;
    process2_state = STOP;
    process3_state = STOP;
    …
    processN_state = STOP;
    init_time();
    init_io();
    for (;;) {
        _r_cur_time = get_time();
        if (_r_cur_time - _r_next_act_time >= 0) {
            // Find next activation time
            _r_next_act_time += _r_CLOCK;
            if (_r_next_act_time - _r_cur_time > _r_CLOCK) {
                _r_next_act_time = _r_cur_time + _r_CLOCK;
            }
            input();
            process1();
            process2();
            process3();
```

```
                …
                processN();
                output();
            }
        }
    }
    //функция чтения всех значений на портах ввода
    void input(void) {
        _i_X_BIT0_PORT = read_byte(0, 0);
        _i_X_BIT1_PORT = read_byte(0, 1);
        _i_X_BIT2_PORT = read_byte(0, 2);
        _i_X_BIT3_PORT = read_byte(0, 3);
        _g_X_BIT0 = _i_X_BIT0_PORT;
        _g_X_BIT1 = _i_X_BIT1_PORT;
        _g_X_BIT2 = _i_X_BIT2_PORT;
        _g_X_BIT3 = _i_X_BIT3_PORT;
    }

    //Функция записи значений в порты вывода
    void output(void) {
        _o_U_BIT0_PORT = _g_U_BIT0;
        _o_U_BIT1_PORT = _g_U_BIT1;
        _o_U_BIT2_PORT = _g_U_BIT2;
        _o_U_BIT3_PORT = _g_U_BIT3;
        write_byte(1, 0, _o_U_BIT0_PORT);
        write_byte(1, 1, _o_U_BIT1_PORT);
        write_byte(1, 2, _o_U_BIT2_PORT);
        write_byte(1, 3, _o_U_BIT3_PORT);
    }
```

Листинг 2 – Программа на языке Reflex после трансляции в язык Си

Можно заметить, что в начале программы идет определение базовых констант, описывающих стандартные для языка Reflex состояния START, CONFIRMED, ERROR и STOP. Состояние CONFIRMED в текущей версии языка не используется, однако данное значение зарезервировано. Для каждого процесса также определяется переменная типа uint8_t, в которой хранится информация о его текущем состоянии и переменная типа uint32_t, предназначенная для хранения таймера процесса. Кроме того, для каждого процесса описывается перечисление, в котором хранятся именования всех состояний каждого из

процессов в порядке их определения. Первое объявленное в программе состояние для данного процесса всегда задается как START, что гарантирует, наличие такого состояния у каждого из процессов. Так происходит хранение всей информации о процессах и их состояниях.

Функции процессов, такие как process1(), process2(), представленные в листинге 2, реализуются в языке Си в виде вызова оператора **switch case**, в котором происходит проверка текущего состояния процесса и, на основе этого вызывается код, описанный в теле текущего состояния, транслированный в язык Си. Пример реализации такой функции представлен в листинге 3.

```
void process1() {
    switch (process1_state) {
        case p1_state1: {
                /*some code*/
           break;
        }
        case p1_state2: {
                /*some code*/
           break;
        }
        …
        case p1_stateN: {
                /*some code*/
           break;
        }
    }
}
```

Листинг 3 – Пример реализации функции process1()

Также стоит отметить, что переменные, порты и т.д., объявленные в гиперпроцессах, транслируются также в глобальные переменные программы. Отличие от фонового гиперпроцесса заключается в том, что для таких гиперпроцессов вместо функции control_loop заводятся функции обработчики прерываний, которые регистрируют код для обработки заданного для гиперпроцесса прерывания. Однако в отличие от control_loop, вызов происходит не в бесконечном цикле, а в одной итерации по прерыванию. В остальном эти функции работают аналогично, то есть представляют из себя последовательный вызов функции процессов.

2.6.2 Процесс-ориентированные конструкции

Такие конструкции, как **if else**, **switch case**, арифметические и логические операции транслируются в язык Си без изменений, поэтому наибольший интерес представляют специфические конструкции языка Reflex. Правила их трансляции описаны в таблице 2.

Таблица 2 – Правила трансляции процесс-ориентированных конструкций в язык Си

Конструкция в языке Reflex	Правило трансляции
set state stateName;	currentProcessName_state = stateName; currentProcessName_time = _r_cur_time;
restart;	currentProcessName_state = START; currentProcessName_time = _r_cur_time;
stop;	currentProcessName_state = STOP;
error;	currentProcessName_state = ERROR;
start process processName;	processName_state = START; processName_time = _r_cur_time;
stop process processName;	processName_state = STOP;
error process processName;	processName_state = ERROR;
set next state;	currentProcessName_state = nextStateName;
reset timer;	currentProcessName_time = _r_cur_time;
timeout interval {<*последовательность выражений*>}	if (cur_time - currentProcessName_time > interval) {<*транслированная последовательность*>}
process processName **in state active**	processName_state < ERROR
process processName **in state inactive**	processName_state >= ERROR
process processName **in state stop**	processName_state == STOP
process processName **in state error**	processName_state == ERROR

Продолжение таблицы 2

vector vec; **register** reg; **bit** bt;	#define currentProcessName_vec vec #define currentProcessName_reg reg #define currentProcessName_bit bit
start hyperprocess hyperprocessName	reg |= (1<<bit);
stop hyperprocess hyperprocessName	reg &= ~(1<<bit);

Примечание: в таблице 2 currentProcessName – имя процесса, в котором запускается транслируемая конструкция, nextStateName – имя следующего по списку состояния для процесса. При этом, если текущее состояние является последним в списке, используется имя первого состояния. reg, bit – регистр и бит, объявленные в гиперпроцессе с идентификатором hyperprocessName.

2.6.3 Типы данных, порты и переменные, временные интервалы

Типы данных транслируются в язык Си по следующим правилам:

- Типы данных int8, uint8, int16, uint16, int32, uint32, int64, uint64 транслируются в аналогичные с суффиксом _t типы, описанные в стандартной библиотеке Си stdint;
- Типы данных float и double транслируются без изменений;
- Тип boolean транслируется в char, при этом значение true транслируется в 1, а false в 0.

Порты транслируются в глобальные переменные типа int8_t и int16_t в зависимости от разрядности. Также и любые переменные, определенные в программе, транслируются в глобальные, и их тип определяется в соответствии с правилами трансляции типов данных. Для каждого порта определяется логика обработки. Например, для входного порта логика определяется в функции **input()**, а для выходного – **output()**. Примеры генерации кода при объявлении портов и привязке их к переменным в языке Reflex представлены ниже в таблице 3.

Таблица 3 – Примеры трансляции объявления портов и отображаемых на них переменных в языке Reflex

Код на языке Reflex	Сгенерированный код на языке Си, при трансляции

input X_BIT0_PORT 0 0 8;
int8 X_BIT0 = X_BIT0_PORT[1];
	`void input(void) {`
	`_i_X_BIT0_PORT = read_byte(0, 0);`
	`if (_i_X_BIT0_PORT & MASK1_S8) {`
	`_g_X_BIT0 = TRUE;`
	`}`
	`else {`
	`_g_X_BIT0 = FALSE;`
	`}`
	`}`

input X_BIT1_PORT 0 1 8;
int8 X_BIT1 = X_BIT1_PORT[];
	`void input(void) {`
	`_i_X_BIT1_PORT = read_byte(0, 1);`
	`_g_X_BIT1 = _i_X_BIT1_PORT;`
	`}`

output U_BIT0_PORT 1 0 8;
int8 U_BIT0 = U_BIT0_PORT[1];
	`void output(void) {`
	`if (_g_U_BIT0) {`
	`_o_U_BIT0_PORT |= MASK1_S8;`
	`} else {`
	`_o_U_BIT0_PORT &= ~MASK1_S8;`
	`}`
	`write_byte(1, 0, _o_U_BIT0_PORT);`
	`}`

output U_BIT1_PORT 1 1 8;
int8 U_BIT1 = U_BIT1_PORT[];
	`void output(void) {`
	`_o_U_BIT1_PORT = _g_U_BIT1;`
	`write_byte(1, 1, _o_U_BIT1_PORT);`
	`}`

Временные интервалы транслируются в числовые литералы, значение которых равняется количеству миллисекунд, соответствующему значению указанного интервала.

2.7 Выбор подхода для обеспечения кроссплатформенности

Для обеспечения платформонезависимости при условии работы языка с векторами, регистрами и битами управляющего устройства, было выделено 2 подхода.

Первый заключается в том, чтобы обеспечить портирование программ, написанных на языке Reflex 2.1 только под операционной системой FreeRtos, то есть реализовать модуль сборки программы под данную операционную систему и встроить в ядро среды разработки. FreeRTOS – это лидирующая на рынке операционная система реального времени для микроконтроллеров [10]. В таком случае в трансляционной семантике языка можно использовать Hardware Abstraction Layer, реализованный в данной операционной системе. Это избавит от необходимости реализовывать уровень аппаратных абстракций в рамках транслятора языка. Однако данный подход имеет несколько недостатков:

1. Завязка на конкретную операционную систему автоматически делает невозможным расширение языка с точки зрения увеличения числа архитектур контроллеров, поддерживаемых данным языком. Поэтому назвать это полноценной платформонезависимостью будет некорректно. Данный подход нарушает одно из ключевых функциональных требований, предъявляемых к языку
2. Реализация модуля сборки под конкретную операционную систему является достаточно трудоемкой задачей.

Второй путь решения предполагает описание для транслятора некоторых стандартных обозначений портов, регистров, векторов и битов контроллера, реализация специальных модулей расширения языка под конкретные архитектуры. Такие описания представляют из себя си-файлы с правилами трансляции стандартных обозначений в специализированные для данной архитектуры посредством ключевого слова define из языка Си и реализацией дополнительного функционала в языке, позволяющего разработчику указать в программе, на какую архитектуру программа будет портироваться. Данное решение позволит расширять список архитектур, на которые возможно портировать программу, написанную на языке Reflex 2.1. Для этого достаточно создать новый модуль расширения в код транслятора, который состоит из объявлений констант на языке Си. Недостатком данного подхода является то, что обеспечение хорошей платформонезависимости потребует немало времени, так как для каждой архитектуры придется описывать свой модуль расширения.

В связи с тем, что первый способ нарушает одно из требований к разрабатываемому языку, было принято решение использовать второй подход и реализовать hardware abstraction layer на уровне транслятора языка.

При этом необходимо обеспечить внедрение новых модулей расширения для возможности портирования программ под другие архитектуры. Для этого будет реализован модуль расширения языка, который позволит указать транслятору целевую платформу при портировании программы посредством LSP протокола.

2.8 Основные выводы по главе

По итогу данной главы:

1. Был описан синтаксис языка Reflex с учетом предлагаемых корректировок;
2. Была описана семантика языка на примере правил трансляции в язык Си, также с учетом корректировок, предлагаемых для внесения в язык;
3. Был предложен подход по обеспечению платформонезависимости в кодогенераторе. Для этого было решено внедрение нового модуля расширения в проект, позволяющего по протоколу LSP конкретизировать целевую платформу, на которую предполагается портирование программы.

Упомянутые выше корректировки представляют из себя внедрение в язык возможности работы с прерываниями, векторами, регистрами и битами процессора микроконтроллера. Данную возможность позволяет реализовать механизм **interrupted** гиперпроцессов. Предложенный механизм не отклоняется от процесс-ориентированного подхода, что, безусловно, является его преимуществом. Кроме того, в язык добавляется механизм обработки Си-кода в виде вставок, синтаксис и семантика которых перенимается из языка Industrial-C.

Глава 3. Выбор стека технологий и практическая реализация ядра

3.1 Подбор стека технологий для практической реализации

3.1.1 Среды разработки

Среды разработки всегда являются комплексом программных средств, предоставляющих возможность вести разработку программ на языках программирования. Они разделяются на 2 основных вида: онлайн (в дальнейшем web) и оффлайн среды. У обоих подходов есть как свои положительные стороны, так и негативные.

Оффлайн среда разработки является обычным, запускаемым на локальной рабочей станции приложением. Такие среды используют файловую систему рабочей станции для хранения исходных файлов программного обеспечения, написанных разработчиком. Такой подход гарантирует быстродействие при работе с кодом и стабильность системы независимо от качества интернет-соединения. Однако данный инструмент ненадежен в силу вероятности потери доступа к исходным файлам, например при поломке рабочей станции разработчика или программном сбое.

Web среда разработки представляет из себя комплекс программных средств, состоящий из онлайн редактора кода, а также ядро среды разработки. Под ядром среды разработки имеется в виду некоторый внешний модуль, который по определенному протоколу обменивается информацией с редактором и реализует функциональные возможности языка программирования, на котором ведется работа в редакторе. Таких модулей может быть несколько и, часто редактор кода предоставляет возможность подключения новых модулей в качестве дополнений. Новые модули могут работать с редактором на удаленном сервере по определенному протоколу. Главным преимуществом онлайн сред разработки является возможность дополнять их функционал, создавая новые модули расширения, обеспечивающие поддержку новых языков. Онлайн редактор кода представляет из себя web приложение для редактирования текстовых файлов, хранящихся в облачном хранилище. Хранение файлов в облаке является вторым ключевым преимуществом онлайн редактора кода, так как освобождает разработчика от хранения на своем устройстве исходных файлов реализованных программных средств. Это в свою очередь приводит к нескольким положительным эффектам. Во-первых, экономится память рабочей станции разработчика. Во-вторых, обеспечивает безопасность файлов, в том смысле, что поломка или сбой рабочей станции не повлечет за собой потерю доступа к файлам, так как они останутся на хранении в облачном хранилище редактора кода. Третьим преимуществом является то, что web среда не обязывает разработчика к установке какого-либо программного обеспечения на свой

компьютер и последующего периодического обновления. И заключительным преимуществом является доступность таких сред, так как для работы в онлайн редакторе необходимо только стабильное интернет-соединение, что в нынешних реалиях редко является проблемой.

В связи с простотой внедрения и реализации нового языка в существующие онлайн редакторы кода и надежностью таких инструментов, было принято решение придерживаться стратегии реализации среды для языка Reflex 2.1 именно в виде ядра web среды разработки.

3.1.2 Выбор подхода и инструментальных средств для реализации web-ide языка Reflex 2.1

При определении подхода и инструментальных средств для реализации онлайн среды разработки было отмечено, что, как и любое web приложение, ядро web среды разработки является серверным программным обеспечением, поэтому ключевым является выбор протокола взаимодействия данного ядра с существующими web средами разработки. В процессе исследования данного вопроса было выявлено два основных подхода.

Первый заключается в реализации своего собственного протокола взаимодействия и самой среды разработки. Такой подход является очень трудозатратным и сложным, так как разработчику необходимо не только написать большой объем кода, в рамках работ по реализации ядра среды разработки, но и грамотно продумать протокол взаимодействия данных приложений. Также, в таком случае становится невозможна интеграция ядра с уже существующими средами разработки, как Eclipse, Visual Studio Code, Theia и т.п. Из плюсов же можно выделить гибкость, так как самописный протокол взаимодействия никак не сковывает разработчика в реализации новой функциональности, и полный контроль над работой всей системы, так как поддержка такого проекта осуществляется одной командой разработчиков. Тем не менее, в силу высокой трудозатратности и невозможности дальнейшей интеграции с другими перспективными продуктами, от такого подхода пришлось отказаться.

Второй подход заключается в использовании уже существующего протокола взаимодействия ядра среды разработки с web редактором кода. Использование подобного протокола решает довольно распространенную в прошлом проблему MxN. Проблема заключается в том, что если взаимодействие онлайн редактора и языкового сервера не стандартизировано, то интеграция M языков в N редакторов кода требует реализации MxN модулей внедрения языка. Таким образом сложность внедрения языка в некоторое множество редакторов становится пропорциональна мощности данного множества. Однако уже долгое время Language Server Protocol (далее LSP), стал стандартом при реализации как языковых серверов, так и онлайн редакторов кода [11]. На сегодняшний день LSP поддерживается множеством легковесных IDE, таких как VS Code, Eclipse Theia, Atom, Vim. Таким образом

реализация ядра web ide при помощи LSP открывает возможности для использования его в большом количестве уже существующих популярных сред разработки [12].

В рамках LSP считается, что языковой сервер обменивается в одном потоке информацией в формате JSON с редактором кода. Выделяется 3 этапа взаимодействия языкового сервера с редактором:

1. Открытие пользователем файла – в этот момент файлы отправляются с редактора кода на языковой сервер;
2. Изменение файла – в момент любого изменения редактор отправляет информации об изменении на сервер;
3. Закрытие файла – в момент закрытия файла отправляется запрос на сервер о прекращении редактирования файла и процесс завершается.

Для реализации Language Server для языка Reflex 2.1, необходимо реализовать несколько основных модулей. Среди них можно выделить модуль взаимодействия по протоколу HTTP с текстовым редактором, парсер исходного кода в AST дерево [13], модуль кодогенерации. Данные компоненты придется либо реализовывать полностью вручную, либо использовать готовый технический инструмент. С одной стороны реализация вручную обеспечивает большую гибкость и полный контроль над сервером, с другой, уже существующие решения, в большинстве своем, являются намного более простыми и их функционала более чем достаточно для такого простого языка, как Reflex 2.1. От ручной реализации было решено отказаться в связи с тем, что высокая трудозатратность такой реализации является неоправданной на фоне обширного функционала существующих инструментов. В качестве же технического инструмента лучше всего подходит XText. XText – фреймворк java-подобного языка XTend для разработки языков программирования и предметно-ориентированных языков, предоставляющий полностью готовую инфраструктуру в виде парсера, классов для хранения AST, а также поддержку редактирования в Eclipse IDE и любом редакторе, работающем по протоколу LSP. Данное средство уже использовалось при разработке ядра web-среды разработки для языков Reflex и Industrial-C и предоставляет достаточные возможности для реализации ядра языка Reflex 2.1.

Таким образом, стек технологий для реализации ядра web среды разработки языка Reflex 2.1 включает в себя XText и среду разработки Eclipse. Взаимодействие языкового сервера с редактором кода осуществляется по протоколу LSP. В качестве тестового онлайн редактора кода выбран редактор Theia.

3.2 Архитектура разрабатываемой системы

Архитектура системы, по большей части совпадает с архитектурой, предоставляемой фреймворком XText по умолчанию. Среди основных компонентов стоит отметить:

1. **Редактор –** модуль, который отвечает за обработку написания пользователем текста, может представлять из себя либо стандартный Eclipse редактор кода, либо онлайн редактор кода, работающий по протоколу LSP с программой;
2. **Парсер –** данный модуль отвечает за обработку переданного кода и генерации на его основе синтаксического дерева (AST). В случае успешной генерации дерева, оно передается модулю семантического анализа для проверки. В случае ошибки, информация об этом возвращается в редактор и отображается пользователю, в случае успеха – дерево передается далее в кодогенератор;
3. **Модуль семантического анализа –** модуль, отвечающий за валидацию написанного на языке Reflex 2.1 кода;
4. **Обобщенный кодогенератор –** данный модуль служит для абстрагирования от конкретных реализаций кодогенерации. Он принимает на вход AST дерево и делегирует кодогенерацию кодогенератору для конкретных языка программирования и платформы;
5. **Кодогенератор имплементирующий** – данный модуль имплементирует генерацию кода на конкретном языке программирования под конкретную платформу по AST дереву. В общем случае таких модулей в проекте может быть несколько, на данный момент в рамках работы реализуются только генератор в язык Си и генератор в язык Си под платформу Ардуино;
6. **Модуль выбора целевой платформы** – модуль предоставляющий функционал по выбору конкретной целевой платформы. Он служит для генерации платформозависимых функций и определений портов, регистров, битов и векторов, в зависимости от платформы, выбранной пользователем.

Парсер и редактор реализуются при помощи XText автоматически [14]. Модуль семантического анализа и кодогенератор реализуются в рамках работы вручную. Ниже представлена наглядная диаграмма, представляющая архитектуру системы в виде диаграммы.

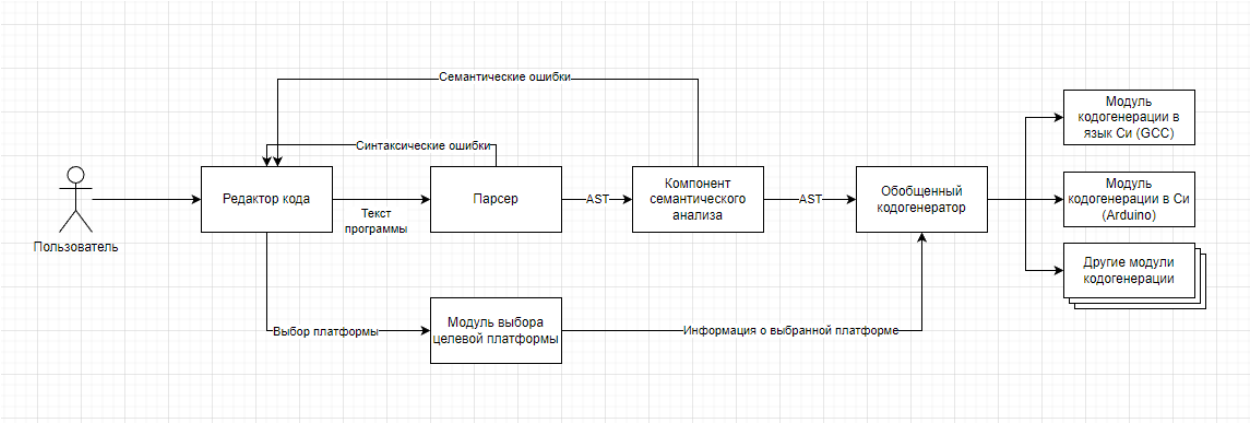

Рисунок 1 – Компоненты системы и их взаимодействие

3.3 Структура модулей системы

Для всех проектов, реализованных с помощью Xtext структура проекта является одинаковой. Проект состоит из слабосвязанных Eclipse плагинов. В следующей таблице описаны плагины, из которых состоит реализуемая система.

Таблица 4 – Структура плагинов в рамках проекта

Название плагина	Компонентная структура	Компонент реализован автоматически с помощью XText, либо уже реализован в рамках Reflex 1.0
ru.iaie.reflex	Парсер	+
	AST классы	+
	Компонент семантического анализа	+
	Обобщенный кодогенератор	+
ru.iaie.reflex.platform	Модуль выбора целевой платформы	-
ru.iaie.reflex.generators.r2c	Генератор кода для языка Си	+-
ru.iaie.reflex.ide	Компоненты поддержки LSP	+

Продолжение таблицы 4

ru.iaie.reflex.ui	Редактор кода средствами Eclipse IDE	+

Примечание: + означает, что компонент реализуется автоматически средствами XText фреймворка или уже реализован в рамках реализации ядра веб среды разработки для языка Reflex 1.0, - означает, что компонент реализуется вручную, +- означает, что в компоненте в рамках работы были сделаны доработки.

XText по умолчанию не позволяет создавать более одного кодогенератора, поэтому в институте автоматики и электрометрии СО РАН был реализован обобщенный кодогенератор, который будет вызываться по умолчанию, он в свою очередь производит поиск классов, имплементирующих интерфейс IReflexGenerator. Такие классы представляют из себя кодогенераторы под конкретный язык программирования, например Си или Python, и под конкретную платформу, например Arduino, STM32. У каждой такой имплементации обобщенный генератор вызывает метод генерации файлов.

Однако, так как в Reflex не было возможности выбора, под какую платформу происходит трансляция кода, было принято решение внедрить модуль выбора целевой платформы. Данный модуль предоставляет функционал пользователю средствами LSP, который позволяет в редакторе кода выбрать, какая платформа будет использоваться для портирования написанной программы. В зависимости от этого к проекту подключается специализированный модуль с объявлением существующих на устройстве портов, регистров, битов и векторов и реализацией платформозависимых функций.

3.4 Структура генерируемых файлов при генерации кода на языке Си

В процессе кодогенерации в язык Си в проекте создается 3 директории. Ниже подробнее описывается назначение каждой из директорий, а также файлов, которые в ней создаются:

usr – данная директория содержит 2 файла (usr.c и usr.h), они в свою очередь необходимы для предоставления пользователю возможности внедрять свои пользовательские функции в программу на языке Reflex. В файле usr.h хранятся объявления таких функций, в usr.c – реализация.

lib – данная директория содержит основные файлы, наполнение которых никак не зависит от целевой платформы и кода, написанного в редакторе. Среди них:

- platofrm.h – файл, в котором написаны объявления основных платформозависимых функций. Служит для декларации компилятору о том, какие функции должны быть

реализованы в файле platform.c в рамках генерации платформозависимого функционала;

- r_cnst.h – в данном файле хранятся определения типов и констант, общих для всех Reflex программ;
- r_main.h – файл, описывающий функции, реализация которых обязательна в main.c файле программы. На текущий момент это только функция control_loop, реализующая цикл обхода процессов и их состояний.

generated – директория, в которой находятся все файлы, генерация которых происходит в зависимости от написанного пользователем кода и платформы. Таких файлов всего три, это platform.c, main.c и ext.h. main.c – файл в котором реализована функция main(), иначе говоря основной файл программы. platform.c – файл описывающий реализацию всех платформозависимых функций программы, его реализация зависит от платформо-ориентированного генератора кода. ext.h – файл, в котором описаны все директивы include внешних для файла main.c библиотек.

3.5 Генерация кода

Процесс генерации кода и реализация данного процесса в общем смысле не претерпела изменений по сравнению с Reflex 2.0. Однако в процессе работы были замечены некоторые недоработки в процессе генерации кода, которые были устранены, в рамках работы. Далее поочередно представлены данные недоработки и то, как они были решены.

Первой недоработкой было выделено хранение состояний процессов и их таймеров в массивах. Такой способ хранения изначально использовался так как в некоторых старых моделях контроллеров было ограничение на количество объявлений переменных, однако в современных контроллерах такой проблемы нет, более того использование массива вместо переменной ухудшает быстродействие программы, так как на обращение к элементу массива по индексу уходит больше машинных команд, нежели на обращение к переменной. Данная недоработка была исправлена вынесением состояний процессов и их таймеров в отдельные переменные.

Второй недоработкой было то, что при генерации кода создавалось слишком много файлов, это затрудняло поиск, в каком месте в коде происходит ошибка при отладке. Для решения данной проблемы большая часть функций и объявлений переменных была вынесена в main.c файл. Из структуры генерируемых файлов исчезли следующие: r_lib.h, r_lib.c, cnst.h, gvar.h, io.h, io.c, xvar.h.

Последней было то, что процесс-ориентированные команды работы с таймерами и состояниями процессов транслировались в вызов функций, описанных в файлах r_lib.c и

r_lib.h, данное поведение было заменено на трансляцию в код реализаций функций обработки процессов. Правила трансляции таких команд описаны в пункте 2 главы 3.

Таким образом, были проведены общие оптимизационные и косметические улучшения процесса кодогенерации в рамках системы RIDE.

3.6 Обеспечение кроссплатформенности системы

Ключевым вопросом в данной работе стал способ обеспечения кроссплатформенности. Проблема заключается в том, что порты, регистры, биты и векторы именуются по-разному, в зависимости от целевого устройства. При решении данного вопроса было решено не уходить от существующих принципов языка Reflex, а именно микросервисной архитектуры системы, идеи вынесения платформо-зависимого функционала в отдельные кодогенераторы под конкретные платформы и хранения платформо-зависимого функционала в файле platform.c. Таким образом было принято решение расширения функционала обобщенного кодогенератора путем внедрения дополнительной логики, проверяющей какая целевая платформа выбрана пользователем на текущий момент и в зависимости от этого использовать конкретный кодогенератор. Также решено создать дополнительный модуль расширения системы, предоставляющий возможность выбора целевой платформы, таким модулем стал модуль выбора целевой платформы, описанный в 1 и 2 пунктах данной главы. По итогу генерация платформо-зависимого функционала и объявлений делегируется кодогенератору под конкретную платформу, а выбор данного кодогенератора зависит от выбранной пользователем платформы, что обеспечивается модулем выбора целевой платформы.

3.7 Реализация модуля выбора целевой платформы

Реализация модуля выбора целевой платформы в рамках данной работы представляла из себя создание компонента *PlatformChooser.class.* Данный компонент является интерфейсом и предоставляет возможность сохранения выбранной платформы, а также возможность получения выбранной платформы при помощи методов *void setPlatform(String platformName)* и *String getPlatoform()*. В рамках работы создана конкретная реализация, которая позволяет в редакторе нажать на файл Reflex программы правой кнопкой мыши и выбрать платформу в пункте Platform. Описан класс GeneratorChooser, в котором реализован метод Class<> chooseByName(String platformName). Данный метод позволяет получить класс конкретного кодогенератора по названию платформы. Он берет данную информацию из текстового файла generators_platforms.txt, каждая строка которого состоит из структуры вида: <имя класса генератора>=<название платформы>. По этой информации и переданному названию

конкретной платформы GeneratorChooser принимает решение о том, какой кодогенератор должен использоваться для генерации кода на языке Си.

3.8 Экспериментальное исследование на практической задаче

В качестве практической задачи была выбрана задача проведения ACD измерений напряжения. В системе присутствует некоторая линия напряжения 220 Вольт. На линии установлен понижающий трансформатор, необходимый для снижения амплитуды напряжения до значения безопасного для микроконтроллера. На определенном участке цепи стоит датчик, измеряющий напряжение. Когда напряжение проходит один период, датчик отправляет прерывание на порт микроконтроллера. Также в системе присутствует преобразователь аналогового сигнала в цифровой, чтобы было возможно производить измерения. Таким образом система выглядит следующим образом. В системе участвует микроконтроллер, управляющий ADC преобразователем, получающий от датчика прерывания и ведущий общение с внешним компьютером по COM порту. Внешний компьютер используется исследователем для управления процессом измерения. Необходимо написать программное обеспечение для ПЛК, которое должно позволять:

1. Разрешать/запрещать прерывания на ПЛК с помощью сигнала на COM порт;
2. Считывать микроконтроллером измерения с ADC-преобразователя и передавать их по COM порту на компьютер исследователя.

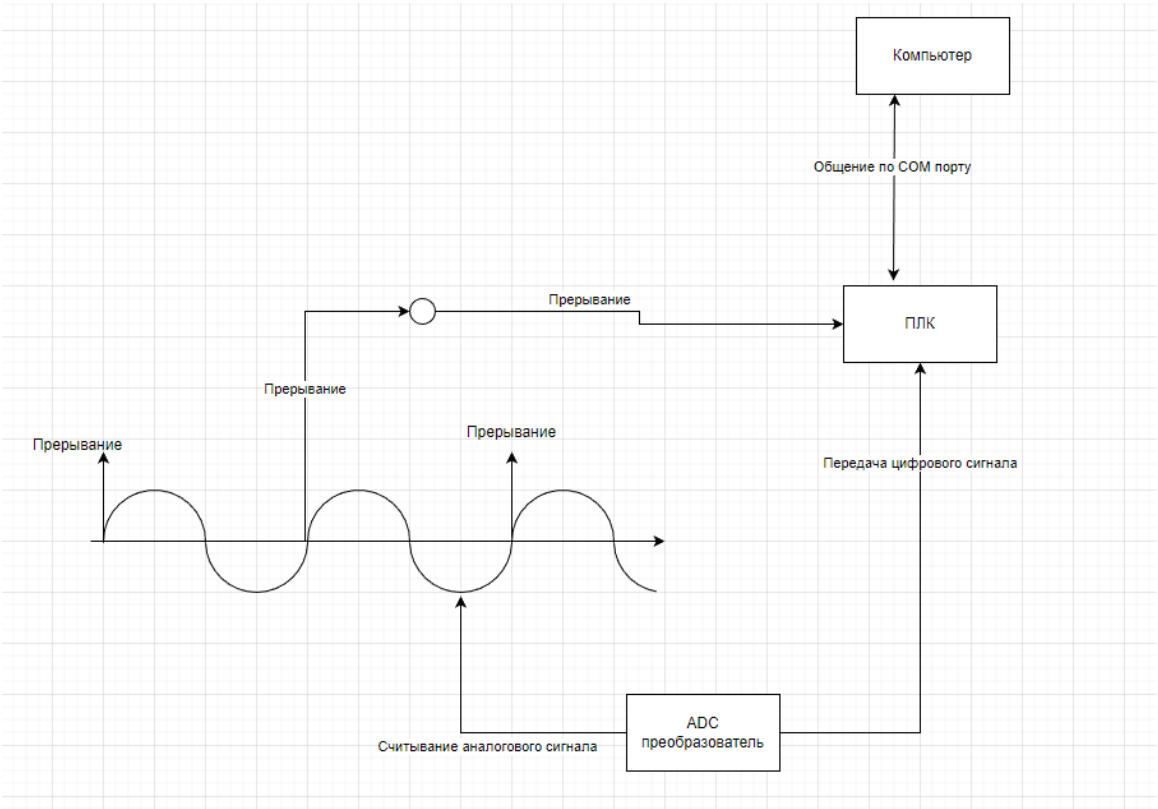

Рисунок 2 – Схема установки

Примечание: на схеме не отражен трансформатор и другие устройства, не принимающие участия в системе считывания и расчета сигнала.

В системе выделяется два основных процесса:

Процесс передачи информации между контроллером и компьютером (для краткости процесс 1) – С компьютера исследователь может в любой момент послать команду контроллеру на начало измерений. После этого контроллер должен разрешить прерывания на процессе 2. Контроллер во время сбора данных посылает их на компьютер по мере их появления. В любой момент с компьютера может прийти новый сигнал на остановку вычислений. В таком случае контроллер должен запретить прерывания на процессе 2.

Фоновый процесс сбора измерений (для краткости процесс 2) – Данный процесс следит за прерываниями, посылаемыми датчиком. Если прерывания разрешены, то при возникновении прерывания должен начаться цикл сбора измерений от ADC преобразователя сигнала. Далее на каждом прерывании происходит новое измерение. Когда прерывания становятся запрещены, процесс завершается и уходит в состояние останова.

Для данной задачи была написана программа на языке Reflex 2.1. Данный язык эффективно показал себя при реализации задачи. Новые возможности языка позволяют решать такие специфические задачи без больших трудозатрат.

3.9 Основные выводы по главе

В данной главе был обоснован выбор стека технологий для реализации ядра веб среды разработки. Описаны архитектура разрабатываемой системы, структура проекта, процесс генерации кода, процесс реализации модуля выбора целевой платформы, и проведено экспериментальное исследование системы на тестовой задаче по измерению напряжения.

Заключение

В данной работе предложена новая версия языка Reflex, представляющая из себя результат унификации данного языка с Industrial-C. В итоге данный инструмент является кроссплатформенным решением, открытым к расширению с точки зрения архитектуры портируемых устройств. Кроме того, он позволяет во всей полноте использовать специфические возможности различных моделей контроллеров.

В процессе данной работы были получены следующие результаты:

1. Исследована процесс-ориентированная парадигма программирования микроконтроллеров на примере языков Reflex и Industrial-C;
2. Синтаксис языка Reflex расширен конструкциями для работы с прерываниями, векторами и регистрами процессора. Тем самым получена новая версия языка.
3. Для новых конструкций описаны правила трансляции в язык Си;
4. В ядре веб среды разработки для языка Reflex расширена функциональность кодогенератора для поддержки новой версии языка;
5. В уже существующих ранее конструкциях внедрены изменения с точки зрения кодогенерации для улучшения быстродействия системы и лаконичности кода на языке Си, полученного путем трансляции исходного кода на языке Reflex;
6. Ядро веб среды разработки расширено новым модулем для выбора целевой платформы в процессе разработки. Данный модуль реализован по всем стандартам Language Server Protocol.

На текущий момент в ядре реализована работа только с платформой Arduino. В дальнейшем планируется поддержка языком других платформ, таких как STM32.

Результаты данной работы были представлены на Международной научной студенческой конференции (г. Новосибирск) и X Международной научной конференции «Математическое и компьютерное моделирование» (г. Омск).

Выпускная квалификационная работа выполнена мною самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для недопуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

_____	_____
ФИО студента	*Подпись студента*

« ____ »_____20 __г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Зюбин, В. Е. Язык «Рефлекс» – диалект Си для программируемых логических контроллеров / В. Е. Зюбин // Шестая международная научно-практическая конференция «Средства и системы автоматизации» CSAF. –2005. – Т. 6. – С. 1-6.
2. Liakh, T. Modeling and Verification using Different Notations for CPSs: The One-Water-Tank Case Study / A. Rozov, V. Zyubin, S. Staroletov, T. Baar, H. Schulte, I. Konyukhov, N. Shilov // 16th Conference on Computer Science and Intelligence Systems (FedCSIS). – 2021. – P. 485-488.
3. Lee, E.A. A cyber-physical systems approach / E.A. Lee, S.A. Seshia // Introduction to embedded systems. – 2011. – P. 1-4.
4. Масюк, В.М. Обзор и классификация современных микроконтроллеров в области мехатроники и робототехники / В.М. Масюк, В.И. Кодубенко, Л.С. Симонова // Материалы всерос. науч.-техн. конф. «Наукоемкие технологии в приборо- и машиностроении и развитие инновационной деятельности в вузе». – 2016. – Т. 5. – С. 40-44.
5. Ямпилов, С.С. Разработка устройства для проведения импедансной спектрометрии биологических объектов / С.С. Ямпилов, Б.Р. Галсанов, Е.И. Копылова // Вестник ВСГУТУ. – 2016. – № 6. – С. 90-94.
6. Zyubin, V. E. Hyper-automaton: A Model of Control Algorithms / V.E. Zyubin // Материалы международной научной конференции IEEE International Siberian Conference on Control and Communications (SIBC0N-2007). – 2007. – С. 51-57.
7. Розов, А. С. Адаптация процесс-ориентированного подхода к разработке встраиваемых микроконтроллерных систем / А.С. Розов, В.Е. Зюбин // Автометрия. – 2019. – Т. 55. – № 2. – С. 114-122.
8. Liakh, T.V. Reflex Language: a Practical Notation for Cyber-Physical Systems / T.V. Liakh, A.S. Rozov, V.E. Zyubin // System Informatics. – 2018. – V. 12. – P. 85-104.
9. Anureev, I. Towards safe cyber-physical systems: the Reflex language and its transformational semantics / I. Anureev // 2019 International Siberian Conference on Control and Communications (SIBCON). – 2019. – P. 1-6.
10. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions [Электронный ресурс] / - Режим доступа: https://freertos.org/index.html
11. Rask, J. K. The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions / J. K. Rask // License. – 2021. – P. 3-18.

12. Bünder, H. Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages / H. Bünder// MODELSWARD. – 2019. – P. 129-140.
13. Neamtiu, I. Understanding source code evolution using abstract syntax tree matching / I. Neamtiu, J.S. Foster, M. Hicks//Proceedings of the 2005 international workshop on Mining software repositories. – 2005. – P. 1-5.
14. Efftinge S., oAW xText: A framework for textual DSLs ./ S. Efftinge, M. Völter //Workshop on Modeling Symposium at Eclipse Summit. – 2006. – Т. 32. – №. 118.

Приложение А. Грамматика языка Reflex v2.1 в нотации XText

```
grammar ru.iaie.reflex.Reflex with org.eclipse.xtext.common.Terminals
generate reflex "http://www.iaie.ru/reflex/Reflex"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
Program:
  hyperprocess+=Hyperprocess
;
Hyperprocess:
  ("[" annotations+=Annotation "]")*
  "hyperprocess" name=ID "interrupted"? "{"
  (port=Port
  vector=Vector
  register=Register)?
  (clock=ClockDefinition)?
  (consts+=Const |
  enums+=Enum |
  functions+=Function |
  globalVars+=GlobalVariable |
  bits+=Bit |
  processes+=Process)*
  "}";
ClockDefinition:
  "clock" (intValue=INTEGER | timeValue=TIME) ";";
Process:
  ("[" annotations+=Annotation "]")*
  "process" name=ID "{"
  ((imports+=ImportedVariableList | variables+=ProcessVariable) ";")*
  states+=State*
  "}";
State:
  ("[" annotations+=Annotation "]")*
  "state" name=ID (looped?="looped")? "{"
  stateFunction=StatementSequence
  (timeoutFunction=TimeoutFunction)?
  "}";
```

```
Annotation:
  key=AnnotationKey ":" value=STRING | key=AnnotationKey;
AnnotationKey:
  ID "." ID | ID;
ImportedVariableList:
  "shared" (variables+=[ProcessVariable] "," variables+=[ProcessVariable])* "from"
"process" process=[Process];
ProcessVariable:
  (PhysicalVariable | ProgramVariable) (shared?="shared")?;
GlobalVariable:
  (PhysicalVariable | ProgramVariable) ";";
PhysicalVariable:
  type=Type name=ID "=" mapping=PortMapping;
PortMapping:
  port=[Port] "[" (bit=INTEGER)? "]";
ProgramVariable:
  type=Type name=ID;
TimeoutFunction:
  "timeout" (TimeAmountOrRef | "(" TimeAmountOrRef ")") body=Statement;
fragment TimeAmountOrRef:
  time=TIME | intTime=INTEGER | ref=[IdReference];
Function:
  returnType=Type name=ID "(" argTypes+=Type ("," argTypes+=Type)* ")" ";";
Port:
  type=PortType name=ID addr1=INTEGER addr2=INTEGER size=INTEGER ";";
Vector:
  "vector" name=ID ";"
;
Bit:
  "bit" name=ID ";"
;
Register:
  "register" name=ID ";"
;
enum PortType:
```

```
    INPUT='input' | OUTPUT='output';
  Const:
    "const" type=Type name=ID "=" value=Expression ";";
  Enum:
    "enum" identifier=ID "{" enumMembers+=EnumMember (',' enumMembers+=EnumMember)* "}";
  EnumMember:
    name=ID ("=" value=Expression)?;
    // Statements
  Statement:
    {Statement} ";" | CompoundStatement |
    StartProcStat | StopProcStat | ErrorStat | RestartStat | ResetStat
    | SetStateStat | IfElseStat | SwitchStat | Expression | StartHyperprocStat | StopHyperprocStat ";";
  StatementSequence:
    {StatementSequence} statements+=Statement*;
  CompoundStatement:
    {CompoundStatement} "{" statements+=Statement* "}";
  IfElseStat:
    "if" "(" cond=Expression ")"
    then=Statement
    (=> "else" else=Statement)?;
  SwitchStat:
    "switch" "(" expr=Expression ")" "{" options+=CaseStat* defaultOption=DefaultStat? "}";
  CaseStat:
    "case" option=Expression ":" "{" SwitchOptionStatSequence "}";
  DefaultStat:
    "default" ":" "{" SwitchOptionStatSequence "}";
  fragment SwitchOptionStatSequence:
    statements+=Statement* hasBreak?=BreakStat?;
  BreakStat:
    "break" ";";
  StartProcStat:
    "start" "process" process=[Process] ";";
  StopProcStat:
```

```
    {StopProcStat} "stop" ("process" (process=[Process]))? ";";
  StartHyperprocStat:
    "start" "hyperprocess" hyper=[Hyperprocess] ";"
  ;
  StopHyperprocStat:
    "stop" "hyperprocess" hyper=[Hyperprocess] ";"
  ;
  ErrorStat:
    {ErrorStat} "error" ("process" (process=[Process]))? ";";
  RestartStat:
    {RestartStat} "restart" ";";
  ResetStat:
    {ResetStat} "reset" "timer" ";";
  SetStateStat:
    {SetStateStat} "set" ((next?="next" "state") | ("state" state=[State])) ";";
  IdReference:
    PhysicalVariable | ProgramVariable | EnumMember | Const;
    // Expressions
  InfixOp:
    op=InfixPostfixOp ref=[IdReference];
  PostfixOp:
    ref=[IdReference] op=InfixPostfixOp;
  FunctionCall:
    function=[Function] "(" (args+=Expression ("," args+=Expression)*)? ")";
  CheckStateExpression:
    "process" process=[Process] "in" "state" qualfier=StateQualifier;
  enum StateQualifier:
    ACTIVE="active" | INACTIVE="inactive" | STOP="stop" | ERROR="error";
  PrimaryExpression:
    reference=[IdReference] | {PrimaryExpression} integer=INTEGER | {PrimaryExpression}
floating=FLOAT |
    {PrimaryExpression} bool=BOOL_LITERAL | {PrimaryExpression} time=TIME | "("
nestedExpr=Expression ")";
  UnaryExpression:
    PrimaryExpression |
```

```
        FunctionCall |
        PostfixOp |
        InfixOp |
        unaryOp=UnaryOp right=CastExpression;
    CastExpression:
        UnaryExpression |
        "(" type=Type ")" right=CastExpression;
    MultiplicativeExpression:
        CastExpression ({MultiplicativeExpression.left=current} mulOp=MultiplicativeOp
right=CastExpression)*;
    AdditiveExpression:
        MultiplicativeExpression ({AdditiveExpression.left=current} addOp=AdditiveOp
right=AdditiveExpression)*;
    ShiftExpression:
        AdditiveExpression ({ShiftExpression.left=current} shiftOp=ShiftOp
right=ShiftExpression)*;
    CompareExpression:
        CheckStateExpression | ShiftExpression ({CompareExpression.left=current}
cmpOp=CompareOp right=CompareExpression)*;
    EqualityExpression:
        CompareExpression ({EqualityExpression.left=current} eqCmpOp=CompareEqOp
right=EqualityExpression)*;
    BitAndExpression:
        EqualityExpression ({BitAndExpression.left=current} BIT_AND
right=BitAndExpression)*;
    BitXorExpression:
        BitAndExpression ({BitXorExpression.left=current} BIT_XOR right=BitXorExpression)*;
    BitOrExpression:
        BitXorExpression ({BitOrExpression.left=current} BIT_OR right=BitOrExpression)*;
    LogicalAndExpression:
        BitOrExpression ({LogicalAndExpression.left=current} LOGICAL_AND
right=LogicalAndExpression)*;
    LogicalOrExpression:
        LogicalAndExpression ({LogicalOrExpression.left=current} LOGICAL_OR
right=LogicalOrExpression)*;
```

```
AssignmentExpression:
  (assignVar=[IdReference] assignOp=AssignOperator)? expr=LogicalOrExpression;
Expression:
  AssignmentExpression;
enum InfixPostfixOp:
  INC="++" | DEC="--";
enum AssignOperator:
  ASSIGN="=" | MUL='*=' | DIV="/=" | MOD="+=" | SUB="-=" | CIN="<<=" | COUT=">>="
| BIT_AND="&=" | BIT_XOR="^=" |
  BIT_OR="|=";
enum UnaryOp:
  PLUS="+" | MINUS="-" | BIT_NOT="~" | LOGICAL_NOT="!";
enum CompareOp:
  LESS="<" | GREATER=">" | LESS_EQ="=<" | GREATER_EQ=">=";
enum CompareEqOp:
  EQ="==" | NOT_EQ="!=";
enum ShiftOp:
  LEFT_SHIFT=">>" | RIGHT_SHIFT="<<";
enum AdditiveOp:
  PLUS="+" | MINUS="-";
enum MultiplicativeOp:
  MUL="*" | DIV="/" | MOD="%";
terminal LOGICAL_OR:
  "||";
terminal LOGICAL_AND:
  "&&";
terminal BIT_OR:
  "|";
terminal BIT_XOR:
  "^";
terminal BIT_AND:
  "&";
  // Types
enum Type:
```

```
        VOID_C_TYPE="void" | FLOAT="float" | DOUBLE="double" | INT8="int8" |
INT8_U="uint8" | INT16="int16" |
        INT16_U="uint16" | INT32="int32" | INT32_U="uint32" | INT64="int64" |
INT64_U="uint64" | BOOL="bool" | TIME="time";
        // Literals
      terminal INTEGER:
        SIGN? (HEX | OCTAL | DECIMAL) (LONG | UNSIGNED)?;
      terminal FLOAT:
        DEC_FLOAT | HEX_FLOAT;
      terminal fragment DEC_FLOAT:
        DEC_SEQUENCE? '.' DEC_SEQUENCE (EXPONENT SIGN DEC_SEQUENCE)? (LONG
| FLOAT_SUFFIX)?;
      terminal fragment HEX_FLOAT:
        HEX_SEQUENCE? '.' HEX_SEQUENCE (BIN_EXPONENT SIGN DEC_SEQUENCE)?
(LONG | FLOAT_SUFFIX)?;
      terminal fragment DEC_SEQUENCE:
        ('0'..'9')+;
      terminal fragment HEX_SEQUENCE:
        ('0'..'9' | 'a'..'f' | 'A'..'F')+;
      terminal fragment BIN_EXPONENT:
        ('p' | 'P');
      terminal fragment EXPONENT:
        'e' | 'E';
      terminal fragment SIGN:
        '+' | '-';
      terminal fragment DECIMAL:
        "0" | ('1'..'9') ('0'..'9')*;
      terminal fragment OCTAL:
        '0' ('0'..'7')+;
      terminal fragment HEX:
        HEX_PREFIX HEX_SEQUENCE;
      terminal fragment HEX_PREFIX:
        '0' ('x' | 'X');
      terminal fragment LONG:
        "L" | "l";
```

```
terminal fragment FLOAT_SUFFIX:
  "F" | "f";
terminal fragment UNSIGNED:
  "U" | "u";
terminal TIME:
  ("0t" | "0T") (DECIMAL DAY)? (DECIMAL HOUR)? (DECIMAL MINUTE)? (DECIMAL SECOND)? (DECIMAL MILLISECOND)?;
terminal fragment DAY:
  "D" | "d";
terminal fragment HOUR:
  "H" | "h";
terminal fragment MINUTE:
  "M" | "m";
terminal fragment SECOND:
  "S" | "s";
terminal fragment MILLISECOND:
  "MS" | "ms";
terminal BOOL_LITERAL returns ecore::EBooleanObject:
  "true" | "false";
```