

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ**

А.Е. Платунов, Н.П. Постников

**ВЫСОКОУРОВНЕВОЕ ПРОЕКТИРОВАНИЕ
ВСТРАИВАЕМЫХ СИСТЕМ**

(Часть 2)

Учебное пособие



Санкт-Петербург

2013

Платунов А.Е., Постников Н.П. Высокоуровневое проектирование встраиваемых систем. – СПб.: НИУ ИТМО, 2013. – 172 с.

Учебное пособие посвящено вопросам проектирования встраиваемых систем. В 2 части пособия рассматриваются: организация вычислительного процесса во встраиваемых системах (ВсС), основные модели вычислений (МоС), которые используются в архитектурном проектировании ВсС, модель актуализации вычислительного процесса ВсС, примеры решения задач этапа высокоуровневого проектирования ВсС.

Для подготовки бакалавров и магистров по направлению 23.01.00 «Информатика и вычислительная техника»; по группе магистерских программ «Встроенные вычислительные системы» по программам подготовки «Проектирование встроенных вычислительных систем», «Системотехника интегральных вычислителей. Системы на кристалле» и «Сетевые встроенные системы».

Рекомендовано к печати Ученым советом факультета КТиУ, 9 апреля 2013 г., протокол № 4



В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена Программа развития государственного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

© Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, 2013

© А.Е. Платунов, Н.П. Постников, 2013.

Оглавление

ВВЕДЕНИЕ.....	5
1 МОДЕЛИ ВЫЧИСЛЕНИЙ ВСТРАИВАЕМЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ.....	8
1.1 АРХИТЕКТУРНАЯ АБСТРАКЦИЯ «МОДЕЛЬ ВЫЧИСЛЕНИЙ» В ПРОЕКТИРОВАНИИ ВСС	8
1.1.1 Поведенческий аспект архитектурной модели	8
1.1.2 Понятие «модель вычислений» в проектировании встраиваемых систем...	10
1.1.3 Совместимость и синхронизация в моделях вычислений	12
1.1.4 Базовые модели вычислений встраиваемых систем.....	15
1.2 СРЕДСТВА МОДЕЛИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ ВСТРАИВАЕМЫХ СИСТЕМ	20
1.2.1 Способы описания распределенных ВСС	20
1.2.2 Симуляция моделей вычислений	24
1.2.3 Инструментальный комплекс моделирования вычислительных процессов Ptolemy II	27
1.3 ОБЪЕКТНО-СОБЫТИЙНАЯ МОДЕЛЬ ВЫЧИСЛЕНИЙ РАСПРЕДЕЛЕННЫХ ВСС.....	36
1.3.1 Общие положения объектно-событийной модели.....	36
1.3.2 Элементы объектно-событийной модели	39
1.3.3 Расчет временных характеристик объектно-событийной модели	54
1.3.4 Расширение ОСМВ на основе денотативного описания	69
Выводы.....	78
2 ПРИМЕНЕНИЕ МЕТОДОВ И СРЕДСТВ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ.....	80
2.1 АНАЛИЗ ТЕХНИЧЕСКИХ РЕШЕНИЙ УРОВНЯ РЕАЛИЗАЦИИ ВСС.....	80
2.1.1 Пример подсистемы визуализации аналитического прибора.....	80
2.1.2 Пример проектирования драйвера периферийного устройства	84
2.1.3 Проектирование специализированных вычислителей системы железнодорожной автоматики «Тракт»	91
2.1.4 Особенности организации КТС «Тракт» систем железнодорожной автоматики	94
2.2 ПРИНЦИП АКТУАЛИЗАЦИИ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА В ПРОЕКТИРОВАНИИ ВСС	104
2.2.1 Модель актуализации вычислительного процесса	104
2.2.2 Фазы актуализации вычислительного процесса ВСС.....	108
2.3 ПРОТОТИП САПР НА БАЗЕ ОСМВ.....	111
2.3.1 Микроархитектура вычислителя NL3.....	112
2.3.2 Модели и параметры DPU	117
2.3.3 Функциональный блок FCUP (Filter Configurable Microprocessor).....	121

2.3.4	Автоматизированная система прикладного пользовательского программирования NL3.....	127
2.3.5	Средства прикладного пользовательского программирования вычислителей NL1 и NL2.....	134
2.4	ИНСТРУМЕНТАЛЬНЫЙ АСПЕКТ ПРОЕКТИРОВАНИЯ ВСС	140
2.4.1	Инструментальный аспект архитектурной модели ВСС	140
2.4.2	Инструментальный комплекс вложенной отладки распределенных ВСС ..	147
2.4.3	Совмещение целевой и инструментальной КМС в распределенных ВСС .	150
2.4.4	Динамические инструментальные компоненты	154
2.4.5	Инструментальный комплекс МЗР.....	158
	Выводы.....	165
	СПИСОК СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ.....	166
	ЛИТЕРАТУРА.....	167
	КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ.....	172

Введение

Стремительный рост потребности во встраиваемых вычислительных системах (ВсС) различного назначения заставляет разработчиков активно совершенствовать методы и средства проектирования. Встраиваемые (или встроенные) системы и сети (embedded systems & networks) можно определить как специализированные (заказные) микропроцессорные системы, непосредственно взаимодействующие с объектом контроля или управления и объединенные с ним конструктивно [22].

Активно растет доля ВсС со сложной внутренней организацией, которая проявляется в таких особенностях, как многопроцессорная гетерогенная архитектура, распределенный характер вычислений, широкий диапазон потенциально доступных разработчику вычислительных ресурсов. Большинство сегодняшних ВсС составляют распределенные информационно-управляющие системы (РИУС), в которых доля технических решений, характерных для иных классов вычислительных систем (ВС), не является доминирующей. Это позволяет сделать вывод об актуальности поиска и развития всего многообразия технических решений в области ВсС (а не только ограниченного их числа в рамках ряда канонических аппаратно-программных платформ), а также методов и средств их проектирования.

Процесс создания ВсС характеризуется высокой сложностью. Это определяется сочетанием таких условий проектирования, как нестандартность задачи, требование технической оптимальности решений (модель ограниченных вычислительных ресурсов), минимальные временные и финансовые бюджеты разработки, присутствие большого числа дополнительных требований и ограничений (надежность, ограничения реального времени, тяжелые условия эксплуатации и многое другое).

Ключевой особенностью создания ВсС является необходимость комплексного проектирования, охватывающего практически все уровни организации ВС. Однако сегодня в достаточной степени формализованы и автоматизированы лишь конечные и часть средних этапов маршрутов проектирования.

Таким образом, первоочередное значение приобретает развитие методов и средств высокоуровневого (архитектурного, HLD – High Level Design) проектирования ВсС, где центральное место занимает формирование цельного взгляда на организацию всех фаз вычислительного процесса, как собственно на цель проектирования.

Создание четкой системы понятий архитектурного уровня позволит разработчику ВсС эффективно работать не на уровне примеров реализаций, а на уровне принципов организации ВС/вычислительного процесса. Важнейшей задачей является создание системы архитектурных абстракций, в которых не противопоставляются аппаратная и программная составляющие ВС, но при этом охватываются все уровни технических решений.

Практическая польза от подобной формализованной системы понятий состоит в возможности развития на ее основе общей теории и методологии проектирования ВсС, создания эффективных методик и САПР архитектурного и сквозного проектирования ВсС.

Следует признать, что в подавляющем большинстве коллективов проектировщиков ВсС сегодня недостаточно высоко оценивается роль и трудоемкость этапов высокоуровневого проектирования, отсутствует адекватный технический язык для общения на этом уровне. Они оперируют лишь конкретными реализациями вычислительных механизмов (то есть «ассемблерными кодами», в которых трудно или невозможно проследить концептуальные моменты и решения). Мери Шоу в статье «Мы можем обучать информатике лучше» пишет: *«Давайте организуем наши курсы вокруг идей, а не вокруг артефактов. Это сделает наши цели более ясными как для студентов, так и для преподавателей. Машиностроительные институты не преподают проектирование бойлера, они преподают термодинамику. В то же время, как минимум два из основных курсов по информатике «Создание компиляторов» и «Операционные системы» являются артефактными динозаврами программирования»*. [73]. С момента опубликования статьи ситуация кардинально не изменилась, данный призыв может быть обоснованно распространен на область проектирования ВсС.

Система вычислительных архитектурных абстракций может привести эффективный язык общения в область проектирования ВсС, повысить «прозрачность» разработок, резко ускорить развитие вычислительных архитектур. Однако для этого необходимы усилия не только со стороны действующих разработчиков вычислительной техники, но и поддержка высшей школы в части модернизации учебного процесса на профильных кафедрах университетов.

2-ая часть учебного пособия посвящена рассмотрению вопросов организации вычислительного процесса во встраиваемых системах.

В разделе 1 подробно обсуждается фундаментальное понятие модели вычислений, рассматриваются варианты моделей вычислений, которые активно используются в архитектурном проектировании ВсС. Приведено описание и рекомендации по применению оригинальной объектно-событийной модели вычислений для ВсС.

В разделе 2 рассматривается модель актуализации вычислительного процесса ВсС, элементы САПР высокоуровневого проектирования ВсС, понятие и наполнение инструментального аспекта организации ВсС. Приводятся примеры решения задач этапа высокоуровневого проектирования ВсС с использованием подходов и методов, представленных в 1 и 2 частях пособия. Особое внимание уделено архитектурному проектированию встраиваемых вычислительных систем на уровне архитектурных абстракций (вычислительных и невычислительных), рассмотрению проектного

пространства ВсС и организации вычислительного процесса. Значительное внимание уделено аспектной модели процесса проектирования ВсС.

Учебное пособие предназначено для студентов, специализирующихся в области вычислительной техники и программирования. Освоение студентами вопросов высокоуровневого проектирования встроенных систем позволит им эффективно внедрять данную методологию в практическую деятельность коллективов разработчиков, в которых они будут работать после окончания учебы.

1 Модели вычислений встраиваемых вычислительных систем

1.1 Архитектурная абстракция «модель вычислений» в проектировании ВcC

1.1.1 Поведенческий аспект архитектурной модели

Одним из основных и традиционно играющим определяющую роль в процессе проектирования ВС является поведенческий аспект проектирования ВcC. Именно на разработку этого аспекта тратится большинство времени при создании современных ВС. В основе работы любой ВcC лежит концепция организации вычислительного процесса, которую часто называют моделью вычислений (МВ или МоС) [46]. Эффективность, как реализации (проектирования), так и самого процесса решения прикладной задачи в первую очередь определяется выбором МоС.

Как было определено в части 1 настоящего пособия, каждый аспект процесса проектирования представляет собой средства описания частной проблемы [23]. Эти средства в некоторых случаях можно рассматривать как самостоятельную математическую модель. *Поведенческий аспект проектирования является моделью вычислений.* Аспектная технология не ограничивает разработчика в выборе той или иной МоС, более того, в принципе, можно рассматривать несколько МоС, но тогда будет нарушено требование ортогональности аспектов, что приведет к усложнению контроля непротиворечивости моделей. Этого можно избежать, если иметь автоматизированные средства трансляции моделей друг в друга с сохранением параметров или эквивалентной верификации моделей [41, 59, 64, 70, 75].

Являясь важнейшим понятием системы абстракции, модель вычислений (или вычислительная модель, Model of Computation) по-разному определяется специалистами. Приведем некоторые определения:

- недвусмысленный формализм для представления спецификаций проекта и проектных решений (S.-Vincentelli, 1998);
- правила взаимодействия компонентов вычислительного процесса (E. Lee, 2003).

Анализ показывает, что понятие МоС целесообразно рассматривать в контексте автоматизации проектирования ВcC в двух направлениях:

1. Направление простых (по «принципу действия») моделей высокого уровня абстракции (процессы Хоара; сети процессов Кана (потокосовые модели); модель с непрерывным временем (НВ); дискретно-событийная (ДС) модель; синхронно-реактивная (СР) модель; конечные автоматы; другие).
2. Направление формализованного представления вычислителей как виртуальных машин независимо от их сложности и принадлежности к

уровню вычислительной иерархии (x86; MCS-51; ARM7...; Java – машина; C – машина; другие).

В рамках системы вычислительных абстракций предлагается определять *модель вычислений как математическую модель, демонстрирующую вычислительные возможности и правила использования субъекта вычислений (актора)* и рассматривать в качестве поведенческого аспекта в аспектной модели проектирования ВсС.

Анализ особенностей сложных распределенных ВсС показал необходимость учета следующих их свойств в выбираемой композиции МоС:

- заранее четко определенная функциональность;
- распределенность вычислительного процесса;
- функционирование различных частей системы в собственных, значительно различающихся масштабах времени;
- наличие слабо связанных вычислительных доменов, сложная топология;
- частое пребывание узлов системы в ожидании событий;
- разнородность элементной базы.

Далеко не все известные МоС имеют сегодня примеры удачных (эффективных) реализаций. Кроме того, в большинстве распространенных ВС соседствуют (иногда не согласованно между собой) многие МоС, воплощенные в таких их стандартных компонентах, как процессоры, ОС, трансляторы, СУБД и другие. В универсальных ЭВМ или системах на их основе с такой ситуацией можно соглашаться, жертвуя суммарной эффективностью и надежностью вычислений в угоду сокращению затрат на массовую разработку не критичных прикладных систем. В области ВсС и особенно РИУС адекватность, корректное сочетание и эффективная реализация выбранных МоС стоят на первом месте в ряду критериев проектирования.

На практике многие разработчики пытаются подменить понятие *модели вычислений* понятием *виртуальной машины*. Как было отмечено выше, под *виртуальной машиной* можно понимать *вычислитель, для которого определены правила поведения* (например, система команд, условия ввода команд, данных, получения результата, правила синхронизации процесса), *позволяющие однозначно описать алгоритм решения задачи*. Описание виртуальной машины демонстрирует лишь внешние свойства вычислителя и правила его использования, не касаясь его устройства. Виртуальная машина может иметь различные реализации, например, в виде конкретной ЭВМ или последовательности "ручных" расчетов. В этом смысле виртуальная машина – это "черный ящик", обладающий известной функциональностью и внешним интерфейсом.

С другой стороны, для эффективного использования виртуальной машины необходимы знания концепции организации вычислений, то есть МоС, на

которую ориентирована виртуальная машина. В этом случае виртуальная машина начинает тяготеть к МоС. Остается вопрос, какой круг задач покрывает такая МоС, и какова степень формальности описания. Но в большинстве случаев, уже по названию “машина” можно предположить, что имеется в виду некоторая конкретная реализация, что, в принципе, не может выступать в роли МоС.

В главе 2 части 1 пособия было показано, что принцип выделения и фиксации виртуальных машин для реализации тех или иных МоС в современной вычислительной технике – один из наиболее мощных технологических инструментов разработчика, позволяющий структурировать процесс проектирования, обеспечить переносимость и повторное использование наработок, масштабировать технические решения и проекты [26].

1.1.2 Понятие «модель вычислений» в проектировании встраиваемых систем

Ключевой задачей в процессе проектирования является обоснованный выбор той или иной модели целевой системы. Рассмотрение системы с точки зрения той или иной модели автоматически привнесет в систему свойства, присущие выбранной модели [46, 70, 72, 35]. *МоС можно представлять как строго определенную парадигму (набор правил), описывающую протекание вычислительного процесса, способы обмена данными, взаимодействия между отдельными функциональными элементами* [46].

Кроме того, МоС предлагает терминологию и примитивы, в базисе которых требуется выражать и описывать целевую систему. МоС описывает природу потоков данных, элементов синхронизации, роль времени в процессе выполнения системой целевой функции. Различные МоС по-разному описывают одни и те же процессы, протекающие в целевой системе. Для больших и сложных систем совершенно нормальное положение дел, когда различные части системы представляются различными МоС.

Важной и неотъемлемой частью модели вычислений является язык МоС. Как и любой другой язык, язык МоС определяется алфавитом, синтаксисом и семантикой. Алфавит представляет собой множество допустимых символов языка, которые могут быть скомбинированы различными способами. Правила комбинирования и допустимые комбинации определяются синтаксисом языка. Смысл и интерпретация тех или иных допустимых комбинаций символов алфавита определяется семантикой языка. С точки зрения языка его МоС представляет собой поведение некоторой абстрактной вычислительной машины (или просто абстрактного вычислителя) в рамках семантики языка. С этой точки зрения можно вести разговор о МоС традиционных языков программирования, таких как C, C++, Java, Pascal и т.д. В этом случае, оценивая применимость того или иного языка, а на самом деле МоС, на которой каждый конкретный язык базируется, для решения той или иной задачи, стоящей перед разработчиком,

можно обсуждать и оценивать характеристики системы, которые навязывает МоС.

Традиционно в процессе проектирования, практически на всех уровнях абстракции, система рассматривается как набор изолированных компонентов (черных ящиков), с определенной функциональностью. Выполняя свои функции в конкретные моменты времени, компоненты “поглощают” входные данные и “производят” выходные данные. Этими данными компоненты обмениваются посредством некой коммуникационной среды. На более абстрактных уровнях рассмотрения системы относительно текущего многие компоненты текущего уровня с коммуникациями превращаются в новые “черные” ящики, скрывая реализацию. На более конкретных уровнях рассмотрения системы относительно текущего становится более конкретной реализация каждого черного ящика, становятся видны составляющие его блоки и коммуникации между ними. Ценность и удобство МоС на определенном уровне абстракции заключается в том, чтобы она не была очень абстрактной или очень подробной. В первом случае модель будет не в состоянии описать процессы, происходящие в целевой системе, с интересующей разработчика точки зрения. Во втором случае для составления полной модели необходим будет такой объем данных, что сложность представления системы окажется неадекватной и бесполезной.

Если рассмотреть целевую систему на некотором уровне абстракции как набор взаимодействующих изолированных блоков (вычислительных компонентов системы), то МоС системы на данном уровне абстракции, используя выразительные средства языка, описывает следующие аспекты системы [70]:

- Поведение вычислительных компонентов;
- Взаимодействие вычислительных компонентов;
- Способы передачи данных и синхронизацию вычислений;
- Способы декомпозиции и агрегации вычислительных компонентов.

В настоящее время предлагается множество МоС ВсС [32, 35, 41, 59, 64, 66], обладающих определенными свойствами и имеющих свои достоинства и недостатки. Для каждой из МоС необходимо учесть следующие моменты:

- Адекватность описания целевой системы;
- Удобство моделирования;
- Формальность методов реализации и верификации.

МоС должна содержать характеристики системы, важные на данном уровне абстракции. Элементы модели (примитивы, языковые средства, требования и др.), а, следовательно, и вся модель в целом, не должны быть слишком абстрактными или слишком конкретными. То есть МоС должна быть в

состоянии *описать* целевую систему. Описание в данном контексте понимается как определенная степень спецификации системы, возможно не до конца формальной. Это требование к МоС можно сформулировать как *адекватность описания* целевой системы.

Кроме собственно описания целевой системы на заданном уровне абстракции, МоС должна обеспечивать разработчику средства работы с этим описанием. Разработчик должен иметь возможность доказывать истинность или ложность определенных утверждений относительно целевой системы, проверять соответствие определенным требованиям и ограничениям, накладываемым на целевую систему. Инструменты, предоставляемые моделью, должны позволять проводить оценку тех или иных характеристик целевой системы, проводить оптимизацию по выбранным параметрам. Все эти действия разработчика можно назвать *моделированием* целевой системы в терминах выбранной МоС.

После обработки и корректировки модели на данном уровне абстракции разработчик должен повышать “конкретность” модели, т.е. *реализовывать* модель. В процессе реализации могут потребоваться значительные ресурсы, чтобы, оставаясь в ограничениях текущей МоС, повысить конкретику представления целевой системы, возможно с переходом на другую МоС. Зачастую требуются достаточно формальные методы трансформации МоС, чтобы в конечном итоге у разработчика оставалась возможность проведения эквивалентной *верификации* получаемых моделей. Необходимы формальные методы трансформации требований, ограничений, спецификаций, функциональности вычислительных компонентов и коммуникаций между ними. Чем более формальные методы описанных процессов предлагает МоС, тем удобнее с ней работать, и результаты такой работы становятся более предсказуемыми.

1.1.3 Совместимость и синхронизация в моделях вычислений

МоС задает некоторые общие положения представляемого в ее рамках вычислительного процесса. Эти положения чаще всего касаются вопросов:

- совместности вычислений отдельных компонентов;
- взаимодействия и синхронизации между ними;
- понятия времени в МоС.

Под совместностью вычислений понимается *потенциальная возможность выполнения вычислений отдельными компонентами параллельно* [67]. Совместность совсем не означает параллелизм вычислений, а может проявляться, например, в псевдопараллельном выполнении вычислений (потoki в многопоточковой среде, задачи в ОСРВ). Совместность вычислений является принципиальным вопросом практически любой МоС, так как фиксация правил совместности вычислений, производимых отдельными компонентами, позволяет делать выводы о планировании вычислений (например,

диспетчеризации отдельных компонентов в псевдопараллельной среде) и о реактивности системы. Совместность вычислений является одним из признаков, по которым можно классифицировать МоС. Например, как Eles [67], так и Liu [56] выделяют:

- Модели вычислений с неявной спецификацией совместности (data-driven, event-driven, demand-driven);
- Модели вычислений с явной спецификацией совместности (control-driven, imperative).

Первый тип совместности предполагает, что спецификация системы дается без явного указания порядка, в котором производятся вычисления отдельными компонентами. Порядок определяется неявно из взаимозависимости компонентов. Если компонент $E1$ требует для вычислений данные, генерируемые на выходе компонента $E2$, то выполнение компонента $E1$ будет зависеть от выполнения $E2$. Модели вычислений с неявной спецификацией совместности типичны, в основном, для задач обработки потоков информации: обработки сигналов и изображений, сбора информации, обработки потоков запросов. Для этого типа совместности различают два стиля выполнения вычислений отдельными компонентами: запуск по наличию данных на входах компонента (data-driven или «push») и запуск по требованию данных с выхода компонента (demand-driven или «pull»). Большинство МоС, используемых при проектировании ВСС, реализуют стиль «push», некоторые позволяют реализовать как первый, так и второй. Модели, реализующие только запуск по требованию, имеют мало смысла, так как ВСС, в основном, являются событийно-ориентированными, то есть реагируют на события внешней среды, которые легче всего представить как входные данные для системы. Среди МоС, реализующих стиль «push», можно назвать: модель с дискретными событиями [60], сети процессов Кана [74], модели с потоками данных (SDF [39, 63], DDF [39]), модель с непрерывным временем [39], ОСМВ. Среди моделей, реализующих комбинированный подход к выполнению вычислений («push/pull»), можно назвать Click [79, 47], CI [79].

Второй тип совместности вычислений предполагает, что порядок выполнения вычислений отдельными компонентами задается явно в спецификации системы. В спецификации может быть указано, какие компоненты выполняются последовательно друг относительно друга, какие совместно (параллельно или псевдопараллельно). В качестве примера можно привести фрагмент спецификации на языке ESTEREL (синхронно-реактивная модель вычислений [49]):

```
run p1;  
[run p2 || run p3];  
run p4;
```

Пример взят из [67]. Здесь содержится явное указание порядка выполнения вычислений компонентами $p1$, $p2$, $p3$ и $p4$. Компоненты $p2$ и $p3$ выполняются совместно друг с другом, но до начала их выполнения должен завершить вычисления компонент $p1$. Компонент $p4$ выполняет вычисления только после завершения $p2$ и $p3$.

Среди МоС с явным указанием совместности можно назвать различные варианты конечных автоматов (Мили, Мура, взаимодействующие синхронные конечные автоматы [57], конечные автоматы со временем, конечные автоматы кодизайна [57]), синхронно-реактивную модель вычислений [56, 45].

На рис. 1.1 представлено обобщение вышесказанного, то есть классификация МоС по типу совместности вычислений отдельных компонентов. На рис. 1.1 приведены лишь некоторые примеры моделей вычислений, находящихся в той или иной группе, однако, существует большое количество МоС, и представить их все невозможно.

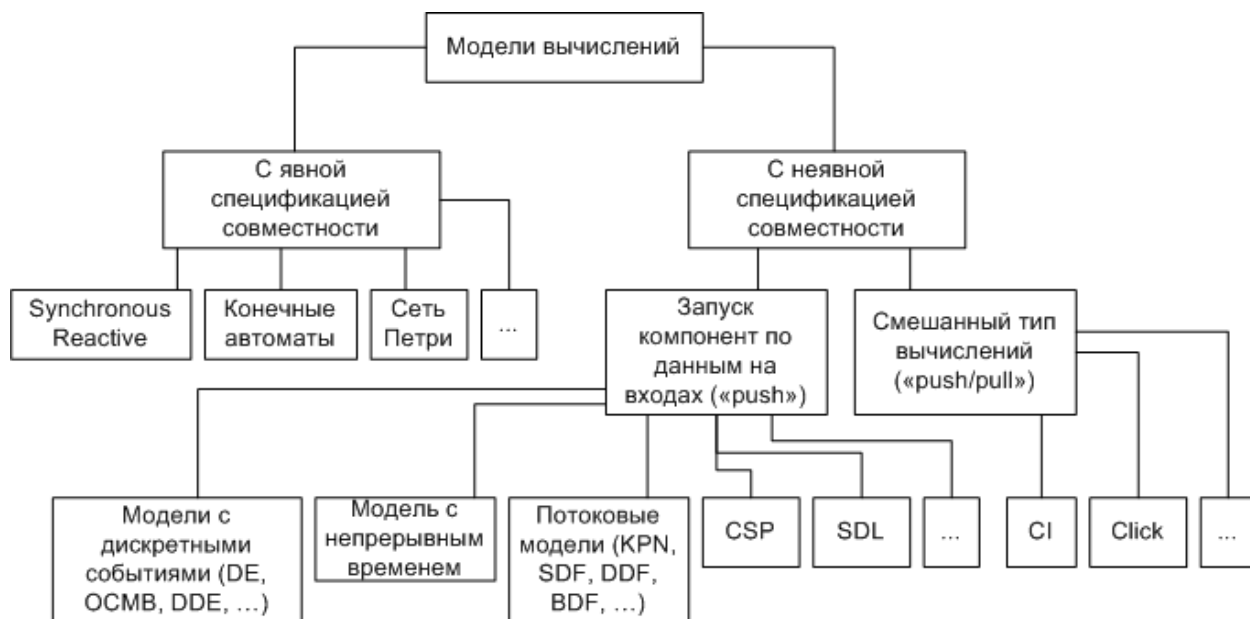


Рис. 1.1. Классификация моделей вычислений по совместности

Данная классификация условна и основана на обзоре литературы по данной тематике и представленных в различных работах соображениях, главным образом, основой для данной классификации можно считать [57, 67, 58].

Вопросы взаимодействия между отдельными компонентами вычислений ставятся некоторыми исследователями на первое место при различии МоС (например, Ли и возглавляемой им группой Ptolemy Project). Под термином «взаимодействие» понимается *передача результатов вычислений между компонентами вычислительного процесса, включающая способ синхронизации между ними для осуществления этой передачи*. Vincentelli и др. [57] выделяют несколько базовых способов, или «примитивов», взаимодействия в МоС, используемых при проектировании ВсС:

- *атомарное «чтение-модификация-запись»*. Доступ компонент к разделяемому буферу данных;
- *очередь неограниченной емкости*. На одну очередь должно быть не более одного компонента, принимающего данные. Если очередь пуста, то при попытке чтения из нее компонент блокируется, то есть приостанавливает вычисления;
- *очередь ограниченной емкости*. От предыдущего отличается тем, что при попытке записи в заполненную очередь компонент блокируется, то есть приостанавливает вычисления до освобождения места в очереди. Моделям вычислений, реализующим данный способ взаимодействия, свойственна проблема блокировок части системы из-за взаимного ожидания входящих в нее компонентов («deadlock»);
- *места сети Петри*. Один или несколько компонентов выдают элементы данных, и один или несколько компонентов их принимают. Размер буфера данных не ограничен. Элементы данных извлекаются из буфера при чтении.
- *«рандеву»*. Синхронизация операций записи и чтения одного или нескольких компонентов. В простейшем случае два компонента, то есть компонент, выдающий данные, и компонент, принимающий данные, ожидают готовности друг друга к взаимодействию. Некоторые МоС предлагают «множественное рандеву», когда для осуществления передачи одного элемента данных из набора компонентов, выдающих данные, и набора приемников не детерминированным способом выбираются два, между которыми происходит взаимодействие.

Обобщение способов взаимодействия приведено в табл. 1.1 (таблица взята из [57] с модификациями).

1.1.4 Базовые модели вычислений встраиваемых систем

Ниже представлен обзор пяти базовых МоС ВвС:

- мета-модель сигналов с тегами TSM.
- сеть обработки потоков данных;
- взаимодействующие конечные автоматы;
- модель дискретных событий;
- синхронная модель вычислений;

1.1.4.1 Мета-модель сигналов с тегами TSM

Для оценки различных аспектов МоС ВвС, а также как способ получения новых моделей Sangiovanni-Vincentelli и Lee предложили Tagged-Signal Model (TSM) [70, 62, 78, 77]. TSM не является какой-либо конкретной МоС встраиваемых систем. TSM представляет собой метамодель и позволяет оценить различные МоС ВвС с разных точек зрения.

Таблица 1.1. Коммуникационные примитивы

Примитив	Источников	Приемников	Буфер	Блокировка по чтению	Блокировка по записи	Извлекающее чтение
Не синхронизированный	несколько	несколько	возможно	нет	нет	нет
«Чтение-модификация-запись»	несколько	несколько	один элемент данных	да	да	нет
Не ограниченная очередь	один	один	не ограничен	да	нет	да
Ограниченная очередь	один	один	ограничен	да	да	да
Место сети Петри	несколько	несколько	не ограничен	нет	нет	да
Простое рандеву	один	один	один элемент данных	да	да	да
Множественное рандеву	несколько	несколько	один элемент данных	нет	нет	да

Основополагающим элементом TSM является *событие* (*event*) – пара значение/атрибут (*value/tag pair*). Если множество значений обозначить V , а множество атрибутов T , то событие есть элемент пространства $T \times V$. Атрибуты зачастую используются для описания поведения системы во времени. Сигнал s является множеством событий. При этом в общем случае на события, объединяющиеся в сигнал, не накладывается никаких дополнительных требований. Сигнал можно рассматривать просто как подмножество $T \times V$

$$s \subset T \times V. \quad (1.1)$$

Вычислительные компоненты (процессы) производят обработку сигналов. Конкретные МоС получаются, если в рамках TSM определить конкретику и соотношения между атрибутами событий, а также характером процессов МоС. Различная роль времени в моделях проявляется в характере множества атрибутов событий T . Можно выделить модели непрерывного времени и дискретного. Для непрерывного времени функциональность системы можно описывать в терминах дифференциальных уравнений. Для дискретного времени существует множество подходов к построению моделей. Если в рамках множества T удастся ввести отношение “порядка”, то можно говорить о

сигналах, как о последовательностях из множества V . Также необходимо сделать допущение, что для сигнала не может существовать двух событий с одинаковыми атрибутами.

1.1.4.2 Сеть обработки потоков данных (Dataflow Process Network)

В сети обработки потоков данных целевая система (обычно это программа в традиционном понимании) представляется в виде ориентированного графа, вершины которого представляют собой вычисления, а дуги представляют собой упорядоченные последовательности элементов данных [3, 6, 32, 61]. Граф обычно имеет иерархичную структуру, так что вершина одного графа сама по себе является другим ориентированным графом. Вершины могут представлять собой также процедуры на некоторых языках программирования.

Множество значений обозначим V и рассмотрим его подмножество s , которое является сигналом $s \in V$.

Рассмотрим несколько сигналов $s^1, s^2, s^3, \dots s^n$. Пусть I – набор входных сигналов $(s^1, s^2, \dots s^k)$, а O – набор выходных сигналов $(s^{k+1}, s^{k+2}, \dots s^n)$ некоторого вычислителя. Функциональность такого вычислителя в модели сети обработки потоков данных задается функцией f .

$$I = \prod_{i=1}^k s^i,$$

$$O = \prod_{i=k+1}^n s^i,$$

$$f : I \rightarrow O.$$

Передача данных между вычислительными вершинами осуществляется через буферы типа FIFO, а сами вычислительные вершины постоянно (вне времени) осуществляют обработку входных данных, порождая наборы выходных данных.

1.1.4.3 Взаимодействующие конечные автоматы (Communicating FSM)

Конечные автоматы всегда были привлекательной моделью для проектирования ВСС [1, 2, 4, 19, 33, 34]. Традиционный конечный автомат состоит из следующих элементов:

- Набора входных сигналов;
- Набора выходных сигналов;
- Конечного набора состояний с четко определенным начальным состоянием;
- Выходной функции, отображающей состояния и входные сигналы в выходные сигналы;

- Функции переходов, отображающей состояния и входные сигналы в состояния.

Традиционные конечные автоматы удобны для моделирования последовательного поведения, но практически неприменимы при моделировании параллельных вычислений или памяти. Собственно привлекательность конечных автоматов так высока, что многие авторы предлагают различные способы практического решения описанных проблем, чтобы расширить область применимости конечных автоматов при проектировании ВСС.

На самом деле конечные автоматы могут быть эффективно использованы при описании функциональности вычислительных компонент в различных МоС. Многие языки предоставляют встроенные синтаксические и семантические средства описания конечных автоматов. Для тех языков, в которых этих средства отсутствуют, различные авторы предлагают формальные методики реализации конечных автоматов доступными средствами языка.

1.1.4.4 Модель дискретных событий (Discrete-Event)

Неотъемлемой частью МоС дискретных событий является время. Происходящие в системе события снабжаются глобально упорядоченными временными метками, обозначающими момент времени, когда событие произошло. При моделировании таких систем обычно используются очереди событий, элементы которых отсортированы по возрастанию временной метки. Разработка аппаратных цифровых схем производится в такой модели. Многие языки описания аппаратуры базируются на модели дискретных событий. Модель дискретных событий эффективна для больших систем, в которых активность в каждый конкретный момент времени проявляется в локализованных участках системы.

Множество значений обозначим V , множество атрибутов обозначим T и рассмотрим подмножество их прямого произведения s , которое является сигналом из формулы (1.1).

Для множества атрибутов T должно быть определено отношение “порядка” и введена метрика пространства, для определения “расстояния” между атрибутами. Для каждого сигнала s определяется функция дискретизации

$$\partial(T \times V, T) \rightarrow V, \quad (1.2)$$

$$\partial(s, t) = v \Leftrightarrow (t', v) \in s, t' \leq t \text{ и } \text{dist}(t, t') = \min, \quad (1.3)$$

определяющая значение сигнала s к моменту времени $t \in T$. Для краткости функцию дискретизации можно записать $\partial(s, t) = s_t$.

Для множеств I и O определяем функцию дискретизации:

$$\partial(I, T) \equiv \prod_i \partial(s^i, T)$$

или

$$\partial(I, T) = I_T \equiv \prod_i s_T^i. \quad (1.4)$$

Функция вычислителя определяется следующим выражением

$$\forall t \in T \quad f : I_t \rightarrow O_t, \quad (1.5)$$

где t – временная метка события.

Модель дискретных событий вносит неоднозначность в случае одновременных событий или при учете задержек на вычислительных компонентах. Модель не накладывает никаких ограничений на реализацию вычислительных компонентов или поведение при одновременных событиях. Это делает симуляцию достаточно простой и наглядной, но реализацию и формальную верификацию модели относительно сложной.

1.1.4.5 Синхронная модель вычислений (Synchronous/Reactive)

Так же как в модели дискретных событий все события системы снабжаются меткой времени. Метки времени так же как и в модели дискретных событий глобально упорядочены. Принципиальным отличием синхронной модели от модели дискретных событий является то, что в синхронной МоС все события в системе синхронные, т.е. все сигналы системы имеют одинаковую метку времени [37]. Сами синхронные события в рамках одного временного цикла, в зависимости от вариаций МоС, могут быть полностью упорядочены, частично упорядочены или вовсе неупорядочены. С другой стороны, в отличие от модели дискретных событий в синхронной модели на каждом временном цикле происходят все события [50].

В течение временного цикла происходит обработка всех входных событий, активизируются все вычислительные компоненты системы, после чего начинается новый временной цикл. В рамках каждого конкретного временного цикла порядок обработки данных и активизация вычислительных компонентов модели определяется последовательностью обработки входных сигналов. Таким образом, временной цикл синхронной модели разбивается на множество так называемых микро-шагов.

Множество значений обозначим V , множество атрибутов обозначим T и рассмотрим подмножество их прямого произведения s , которое является сигналом из формулы (1.1).

Определяем функцию синхронизации sync , которая обеспечивает временное тактирование всей системы, т.е. упорядочивает элементы множества атрибутов событий: $\text{sync} : N \rightarrow T$.

Для каждого сигнала s определяется функция дискретизации:

$$\begin{aligned} \partial(T \times V, T) &\rightarrow V, \\ \partial(s, t) = v &\Leftrightarrow (t, v) \in s, \end{aligned}$$

определяющая значение сигнала s в момент времени $t \in T$. Для краткости функцию дискретизации можно записать:

$$\partial(s, t) = s_t.$$

В силу ограничений синхронной модели верно следующее утверждение:

$$\forall n \in N \text{ и } \forall s \subset T \times V \exists v : s_{sync(n)} = v.$$

Для множеств I и O определяем функцию дискретизации как в модели дискретных событий [выражения (1.4), (1.5)]. Функциональность вычислителя определяется следующим выражением

$$\forall n \in N \quad f : I_{sync(n)} \rightarrow O_{sync(n)},$$

где n -номер цикла синхронизации.

Такие циклические модели идеально подходят для разработки синхронных аппаратных схем или систем обработки сигналов. В случае, когда в системе невозможно выделить события, происходящие одновременно на всех сигналах, синхронная модель вычислений становится неприменима. Можно пытаться реализовать различные технические решения, позволяющие использовать синхронную модель при моделировании даже в этом случае, но реализация синхронной МоС в таких системах будет крайне неэффективна и потребует значительных ресурсов.

1.2 Средства моделирования вычислительных процессов встраиваемых систем

1.2.1 Способы описания распределенных ВСС

Как было показано в главе 1 части 1 настоящего пособия, в последнее время при проектировании сложных ВСС все шире применяется архитектурное описание проектируемой системы. Большинство разработчиков признают перспективность такого рассмотрения системы и пытаются развивать это направление в проектировании ВС. Видно, что различные разработчики совершенно по-разному понимают само понятие архитектуры ВС, однако во всех подходах демонстрируется, что описания одной лишь функциональности недостаточно для адекватного представления ВС в процессе ее эволюции при реализации.

1.2.1.1 Диаграммы потоков данных и управления

Функциональное и структурное описание системы можно представлять в виде широко распространенных в различных их проявлениях функциональной модели (Process Model) и модели управления (Control Model) [52].

Ядром функциональной модели можно рассматривать DFD (Data Flow Diagram – диаграмма потоков данных). Все изначально сформулированные

требования к целевой системе разбиваются на функциональные элементы, называемые процессами, и вся совокупность этих требований представляется в виде сети процессов, соединенных потоками данных. Основное назначение диаграммы потоков данных – показать, как каждый процесс преобразовывает входные потоки данных в выходные, и описать связи между процессами. Диаграмму можно дополнить необходимой информацией, которая включает правила описания алгоритма функционирования процесса (обработки входных данных) и спецификацию на потоки передачи данных (тип данных, формат, состав и т.д.). В случае необходимости отдельно добавляются и впоследствии учитываются требования РМВ.

В рамках описания модель управления описывает правила и законы управления процессами, составляющими сеть процессов функциональной модели, но не раскрывает содержания этих процессов. Задачей данной модели является указания правил, по которым процессы “активизируются” и выполняют свои функции. Эти правила могут быть выражены одним из множества способов, например таблицами истинности, диаграммами смены состояний (автоматная модель), таблицами активизации процессов и т.д.

Представленный способ описания ИУС является достаточно общепринятым на данный момент и с небольшими модификациями и дополнениями реализован в различных средствах архитектурного проектирования [3, 31, 35]. На самом деле можно заметить определенное сходство такого способа описания с одной из разновидностей сети обработки потоков данных (Dataflow Process Network). Преимущество такого способа заключается в том, что он интуитивно понятен большинству разработчиков и является одним из “естественных” способов представления системы, пусть даже иногда при этом страдает формальность описания.

1.2.1.2 Целевое прикладное программирование

Принципиально другой способ описания ИУС можно получить, если рассматривать систему с точки зрения целевого прикладного программирования. Обычно такое представление системы необходимо создавать, если спроектированная система предоставляет возможность доопределить свою [прикладную] функциональность конечному пользователю. Именно в этом случае конечный пользователь должен иметь представление о системе, как о некоем сложном устройстве, допускающем прикладное программирование. Другими словами, конечный пользователь имеет возможность “конфигурировать” целевую систему для выполнения конкретной (необходимой именно этому пользователю) функции управления. Под “конфигурированием” системы в частном случае можно понимать ее целевое прикладное программирование. Целевым это программирование называется потому, что таким образом определяется целевая функция системы. Если такой способ описания системы рассматривать не с точки зрения конечного пользователя, а уже на этапе проектирования самим проектировщиком, то его задача несколько видоизменяется. Вместо того, чтобы сразу спроектировать

конечный продукт, необходимо выделить функциональную платформу (базис), и целевую надстройку. Прикладное программирование в рамках целевой надстройки в дальнейшем позволит самому разработчику или конечному пользователю “конфигурировать” систему для решения конкретной задачи. Такой способ построения системы особенно привлекателен именно для РИУС.

Разработка сложного комплекса технических средств, что зачастую и представляет собой РИУС, требует значительных затрат и может окупаться только в случае значительного коэффициента повторного использования. Рассмотрение РИУС как совокупности функциональной платформы и целевой надстройки позволит разработчику резко повысить коэффициент повторного использования за счет ресурсоемкой функциональной платформы, которая не требует повторной разработки и может быть практически полностью повторно использована за счет изменения целевой надстройки.

Представленный способ описания ИУС предложен в стандарте IEC 1131-3 [68]. Указанный стандарт определяет три модели целевой системы: программную модель (software model), коммуникационную модель (communication model) и модель программирования (programming model). Совокупность этих моделей позволяет достаточно подробно описать целевую ИУС и дает возможность разработчику выразить целевую функцию управления в терминах и средствами системы.

Основными организационными элементами функционального описания целевой системы в стандарте IEC 1131-3 выступают функции (function), функциональные блоки (function block) и программы (programs). Не умоляя общности в качестве носителей функциональности целевой системы, можно рассмотреть один из перечисленных элементов. Для дальнейшего рассмотрения удобно в качестве такого элемента выбрать функциональный блок (ФБ). Рассмотрение именно ФБ также интересно и с точки зрения определенных его свойств. В отличие от функций или программ, представленных в IEC 1131-3 и выражающих только алгоритмическую сущность, ФБ подразумевает некоторую структурную составляющую [5, 30].

По аналогии с языками программирования переход к использованию ФБ можно расценивать как переход от процедурного программирования к объектно-ориентированному программированию. Такие свойства классов как полиморфизм и инкапсуляция играют огромную роль при рассмотрении РИУС с точки зрения множества объектов. Полиморфизм и инкапсуляция ФБ позволяет отодвинуть вопросы их реализации на как можно более поздние этапы разработки целевой системы [59, 76]. В свою очередь, такая возможность оставляет значительную свободу маневра при проектировании РИУС, выражаемую как в более свободном манипулировании архитектурной платформой, так и в возможности декомпозиции функциональности на программно-аппаратную реализацию на последних этапах проектирования. Указанные свойства ФБ позволяют проектировать систему на концептуальном

уровне, не вдаваясь в тонкости реализаций, что очень ценно при моделировании тех или иных процессов, протекающих в сложной РИУС.

1.2.1.3 Аналогия с «аппаратным» блоком

Еще один подход к описанию РИУС можно получить, если провести аналогию между функционированием РИУС и некоторой абстрактной «аппаратной» платы, решающей аналогичные задачи. В обоих случаях задачами систем являются получение входных сигналов/данных, их обработка и генерация выходных воздействий/данных. Соотношение между весовыми долями «информационности» и «управляемости» в таких платах как и в системах может быть разной и определяется спецификой решаемой задачи. Видно, что принципиальных различий на представленном уровне рассмотрения систем нет, кроме того, в связи с развитием элементной базы и повышением доступности ПЛИС, многие части целевой функциональности системы могут быть в конечном итоге реализованы не только с помощью традиционных микроконтроллеров. Эта возможность делает различия между абстрактным рассмотрением ИУС и аппаратных блоков не просто незначительным, но даже вредным, так как автоматически выводит из поля зрения разработчика огромное количество [потенциально эффективных] решений.

Отсутствие принципиальных отличий между функционированием РИУС и некоторой абстрактной «аппаратной» платы позволяет позаимствовать модели, понятия и средства описания из области разработки аппаратных блоков и схем [65, 71] в область разработки и описания РИУС. Такими полезными понятиями могут стать понятия микросхемы (аналог обсуждаемого выше ФБ), синхронных и асинхронных входов, механизмов тактирования и синхронизации.

Понятия слабой и сильной связи вычислителей (или ФБ) и способы реализации этих связей опять же можно заимствовать из типовых решений в области проектирования аппаратуры, где аналогами вычислителей можно рассматривать микросхемы или же отдельные блоки в составе одной микросхемы. Многие полезные понятия можно позаимствовать из современных динамично развивающихся языков описания аппаратуры.

Другим удачным заимствованием может стать представление о функционировании элементов системы совершенно независимо друг от друга. К сожалению, на этапе проектирования системы многие разработчики ставят проблемы псевдопараллельного исполнения и межпроцессорной синхронизации во главу угла, так как не могут избавиться от стереотипной модели исполнения функций на единственном процессоре. В общем случае это снижает эффективность проектируемой системы и, на самом деле, не отражает реальной ситуации. В реальной системе многие функции будут выполняться независимо и параллельно во многих случаях, например:

- в распределенных системах имеется несколько слабосвязанных вычислителей, которые по определению слабой связи выполняют задачи независимо и параллельно;

- проблемы синхронизации и псевдопараллельного выполнения могут быть решены в процессе реализации с использованием стандартных механизмов ОСРВ (если таковые будут использованы в ИУС), с помощью разработки простейших алгоритмов разделения процессорного времени, автоматическими средствами генерации комплексного кода с заданными параметрами временного отклика;
- даже в случае единственного центрального процессора в системе, выполняющего основную часть задачи, многие целевые функции могут быть решены периферийными контроллерами или аппаратными акселераторами.

Для простоты ниже будут рассмотрены лишь несколько аспектов архитектурного описания целевой системы. В качестве таких аспектов будут рассмотрены функциональный и структурный. Эти два аспекта зачастую играют ведущую роль при проектировании ВС. Кроме того, выше было показано, что большинство МоС представляют систему именно с точки зрения функциональности, структурной организации и коммуникаций. Можно попробовать рассмотреть различные способы представления ВС, чтобы попытаться выделить базовые примитивы, удобные для представления РИУС. Вместе с выбором способа описания будет осуществлен и выбор определенной МоС, так как они тесно взаимосвязаны и взаимозависимы.

Важной особенностью РИУС является работа различных частей системы в локальных масштабах времени, которые в свою очередь могут на порядки различаться для разных частей системы. Одним из способов формализации учета масштабов времени является рассмотрение каждого ФБ как некоторого *абстрактного вычислителя*. Для каждого ФБ можно составить полный перечень входных событий со всеми необходимыми параметрами: форматами данных, скоростью исполнения и т.д. Скорость работы самого низкого в иерархии ФБ определяется схемами тактирования, которые могут принадлежать самой системе или быть внешними для нее.

В разделе 1.3 предлагается способ описания распределенных ВсС (ОСМВ – объектно-событийная модель вычислений), основанный на представлении системы как множества ФБ (объектов), которые управляются асинхронными событиями, вызванными изменениями состояния объекта управления или самой системы.

1.2.2 Симуляция моделей вычислений

Важным вопросом при работе с моделями вычислений является построение эффективных симуляторов.

Можно реализовать простой и эффективный способ симуляции моделей, в котором симулятор организует все события в системе в единую очередь событий, сортируя ее по временным меткам. Извлекая из очереди событие с минимальной временной меткой, равной текущему модельному времени,

симулятор активизирует процесс, входному сигналу которого принадлежит событие. Если процесс генерирует в результате активизации события в одном из своих выходных сигналов, симулятор добавляет его в очередь, помещая на позицию, соответствующую временной метке события. Если в очереди не осталось событий с временными метками, равными текущему модельному времени, симулятор увеличивает модельное время, делая его равным минимальному значению временной метки событий в очереди.

Данный способ лежит в основе большого числа симуляторов моделей с дискретными событиями: симуляторов систем, описанных на языках VHDL [53] и Verilog [43, 42], систем на основе дискретной логики, входящих в состав САПР систем на ПЛИС, симуляторов моделей массового обслуживания, телекоммуникационных протоколов [39].

Однако, системам в рамках МоС с дискретными событиями присущ ряд проблем. Одна из них связана с потенциальной недетерминированностью поведения системы, в которой процессы преобразуют сигналы без задержки. Вторая также проявляется в виде недетерминированности поведения, но связана с наличием одновременных событий. Ниже предлагается иллюстрация обеих проблем на одном примере.

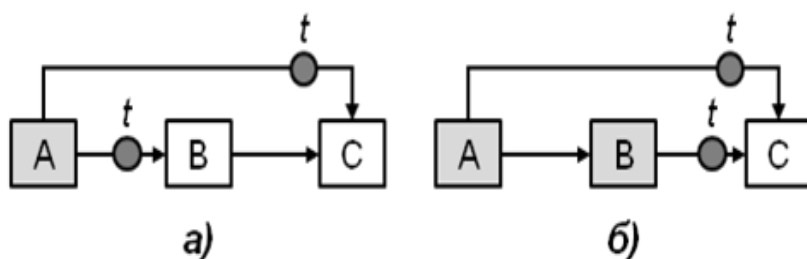


Рис. 1.2. Недетерминированность поведения системы без задержек (пример взят из [57])

На рис. 1.2 представлен пример системы, состоящей из трех процессов. Процесс В преобразует входной сигнал без задержки, то есть временная метка события на выходе равна временной метке соответствующего события на входе. Пусть процесс А одновременно генерирует события по двум своим выходам (рис. 1.2, а). Оба события имеют одну и ту же временную метку t . Возникает вопрос: какой процесс должен быть активирован первым, В или С? Если первым будет активирован процесс С, то в момент активации на его входах будет присутствовать только одно событие. Если же первым будет активирован процесс В, то в момент активации С на его входах будут присутствовать оба события с одинаковой временной меткой t (так как В преобразует входной сигнал без задержки). Интуитивно понятно, что если два события имеют одинаковую временную метку, то в момент активации процесса оба должны одновременно присутствовать на его входах. Однако, МоС с дискретными событиями формально не фиксирует такое поведение. Решение данной проблемы отдано на откуп симулятору модели, и в разных симуляторах

результаты моделирования системы, заданной одной и той же спецификацией, могут быть различными.

Кроме того, модель с дискретными событиями не дает ответа на второй вопрос: сколько раз должен быть активирован процесс С в момент, когда на двух его входах присутствуют события с одинаковыми временными метками (рис. 1.2б)? Однако данный вопрос однозначно решается почти всеми симуляторами, построенными на основе глобальной очереди событий. Поскольку события все-таки разные, процесс активируется по каждому событию, то есть два раза. Неоднозначность появляется как раз при таком решении, какое из двух событий должно первым активировать процесс С? Если процесс имеет внутреннее состояние, зависящее от потока входных событий, то эта неоднозначность может повлечь за собой различные состояния процесса, а значит и системы.

Другая проблема, связанная с моделированием в рамках МоС с дискретными событиями, касается сложности моделирования больших распределенных высокопроизводительных систем, выражающейся в больших вычислительных затратах и, опять же, в некоторых случаях в недетерминированности и неадекватности моделей. Модель вычислений с дискретными событиями хорошо подходит для моделирования систем, имеющих довольно большое число редко активируемых компонентов [57]. Для моделирования поведения такой системы достаточно активировать отдельные компоненты при изменении состояния входного сигнала, вместо того, чтобы поочередно запускать все компоненты (как в синхронных моделях вычислений) для вычисления состояния системы.

Модели с дискретными событиями могут использоваться при гетерогенном моделировании, то есть при моделировании иерархических систем, различные компоненты которых реализованы в разных МоС. Однако при этом следует учитывать некоторые моменты. В первую очередь, сложность моделирования интенсивных потоков событий. В МоС, в которых активация компонентов происходит часто, особенно в синхронных моделях вычислений (синхронно-реактивной, с непрерывным временем, DT), использование компонента, реализованного в МоС с дискретными событиями, может потребовать значительных вычислительных ресурсов.

При использовании в модели с дискретными событиями компонентов, реализованных в других моделях вычислений, следует учитывать, что активация процессов в МоС с дискретными событиями происходит только при возникновении события на одном из его входов. Поэтому вычисление внутреннего состояния компонента между событиями на его входах может потребовать либо модификации симулятора моделей с дискретными событиями, либо многопоточной симуляции. Во втором случае актуальна проблема синхронизации высокоуровневой модели (с дискретными событиями) и низкоуровневой (реализация компонента модели верхнего уровня).

1.2.3 Инструментальный комплекс моделирования вычислительных процессов Ptolemy II

Ptolemy II представляет собой комплекс программ, библиотек Java-классов и механизмов [69]. Комплекс предназначен для построения и исследования моделей различного рода систем, в основном для встраиваемых систем, а также для академических исследований в области моделирования.

На основе идей и подходов, реализованных в Ptolemy II, активно развиваются средства моделирования как во многих исследовательских HLD САПР, так и в продуктах индустриального назначения [44].

Кратко возможности и особенности системы Ptolemy II представлены ниже.

Ptolemy II создан и постоянно развивается группой исследователей Ptolemy Project из университета Калифорнии в Беркли, возглавляемой с профессором Edward A. Lee [69, 40]. Комплекс разрабатывается как open-source ресурс и распространяется свободно. Ptolemy II является третьим по счету инструментом, который разрабатывается группой Ptolemy Project, после Gabriel (1986-1991) и Ptolemy Classic (1990-1997). Большое количество идей для Ptolemy II было взято из Ptolemy Classic, который представлял собой средство моделирования и проектирования систем обработки данных и управляющих систем. Ptolemy Classic позволял создавать в графической среде гетерогенные модели систем с использованием разных моделей вычислений, исполнять эти модели на симуляторах различных аппаратных и программных платформ, а также генерировать из моделей код на ассемблере (для DSP), C и VHDL.

Ptolemy II начал разрабатываться в 1996 году и был призван сместить акцент на моделирование с использованием технологий, предоставляемых платформой Java™. В новой среде были введены понятия доменного полиморфизма, «режимных» (modal) моделей, а также типового полиморфизма. Был также разработан механизм разрешения типов данных в модели, множество моделей вычислений, технология распределенного моделирования, а также мощный язык выражений, который может быть использован, в том числе, и в исполняющейся модели.

Ptolemy II обеспечивает моделирование вычислительных систем различной природы. На основе комплекса созданы специализированные подсистемы для моделирования:

- беспроводных сетей и распределенных систем, объединенных радио-, оптическими или акустическими каналами связи (пакет VisualSense);
- гибридных систем, объединяющих в себе дискретную логику управления и модели с непрерывным временем. Пакет HyVisual включает в себя модели вычислений FSM (finite state machines) и CT (continuous-time), библиотеку акторов, поддерживающих эти модели вычислений и средства визуализации моделей.

Каждая подсистема представляет собой подмножество классов, механизмов и акторов, объединенное одной управляющей программой.

Ptolemy II создан для моделирования гетерогенных вычислительных систем, то есть систем, объединяющих компоненты, описываемые в разных моделях вычислений: дискретную и аналоговую технику, сети и телекоммуникационные протоколы, стохастические и криптографические системы, радиокомпоненты, компоненты, реализующие обработку сигналов и изображений и т.д. Для этого комплекс поддерживает моделирование в рамках различных моделей вычислений, и объединение отдельных моделей в гетерогенную иерархию с последующим комплексным моделированием. В настоящее время комплекс поддерживает следующие модели вычислений:

- CT (continuous time) – модели с непрерывным временем;
- DE (discrete-event) – модели с дискретными событиями;
- DDE (distributed discrete events) – системы с частично упорядоченным множеством дискретных событий;
- FSM (finite-state machines) – конечные автоматы, в модели отсутствует понятие времени;
- PN (process networks) – сети процессов Кана (Kahn), в модели отсутствует понятие времени;
- SDF (synchronous dataflow) – модели для представления систем обработки сигналов, в моделях отсутствует понятие времени;
- DDF (dynamic dataflow) – расширение SDF, позволяющее компонентам изменять количество потребляемых и генерируемых элементов данных за одну итерацию в процессе исполнения модели;
- HDF (heterogeneous dataflow) – расширение SDF, позволяющее сохранить статическое планирование и другие свойства моделей в ее рамках и избегать ограничений на неизменность количества потребляемых и генерируемых компонентами элементов данных;
- PSDF (parameterized synchronous dataflow) – расширение SDF с параметризуемой дисциплиной планирования вычислений;
- DT (discrete time) – периодически запускаемые статически планируемые процессы с дискретным временем. Аналог SDF, в котором присутствует понятие времени;
- SR (synchronous-reactive) – синхронно-реактивная модель вычислений;
- CI (component interaction) – модель вычислений со стилем взаимодействия компонентов «push/pull»;
- CSP (communicating sequential processes) – взаимодействующие последовательно выполняемые процессы;

- Giotto – поддержка одноименного языка и его семантики. Один из вариантов синхронной модели, в котором компоненты могут активироваться с разной частотой;
- TM (timed multitasking) – вариант взаимодействия задач ОСРВ;
- Wireless – модель вычислений для исследования систем, объединенных беспроводными каналами связи;
- GR – поддержка 3-D графики.

Комплекс Ptolemy II является open-source-проектом, что способствует распространению и широкому использованию подходов и механизмов, реализованных в нем. Комплекс является масштабируемым практически в любом аспекте, включая поддержку новых моделей вычислений. Исходный текст комплекса оформлен аккуратно и в соответствии с принятым в проекте стилем кодирования, что позволяет легко разобраться в исходном коде любого Java-класса. Кроме того, весь исходный код документирован с помощью утилиты Javadoc, что позволяет, избегая тщательного анализа исходных текстов, быстро перемещаться по систематизированной документации в поиске требуемой информации о том или ином классе. Комплекс снабжен тремя томами подробного описания всех возможностей, реализованных механизмов и моделей вычислений, а также техники моделирования и рекомендациями по применению и расширению возможностей комплекса Ptolemy II. Лицензия Ptolemy II предоставляет право свободного применения всего или частей комплекса в любых разработках, включая коммерческие продукты.

Система Ptolemy II включает в себя графическую среду разработки Vergil, позволяющую схематично представлять модели, аннотировать и конфигурировать их, проверять их корректность и запускать на исполнение (рис. 1.3).

Vergil включает редактор моделей в акторном представлении (Graph editor), редактор конечных автоматов (рис. 1.4), текстовый редактор, редактор иконок акторов, позволяет просматривать HTML-страницы и редактировать текст файлов.

Компонентами моделей Ptolemy II по большей части являются акторы, атомарные функциональные преобразователи сигналов, поведение которых описано на Java. Каждый такой актор представляет собой Java-класс, который может быть в процессе моделирования расширен и дополнен различными механизмами Ptolemy II. Акторы могут быть составными, то есть являться контейнерами других моделей, обеспечивая, таким образом, их иерархичность. Комплекс включает обширную библиотеку акторов, систематизированную по группам выполняемых ими преобразований (рис. 1.5).

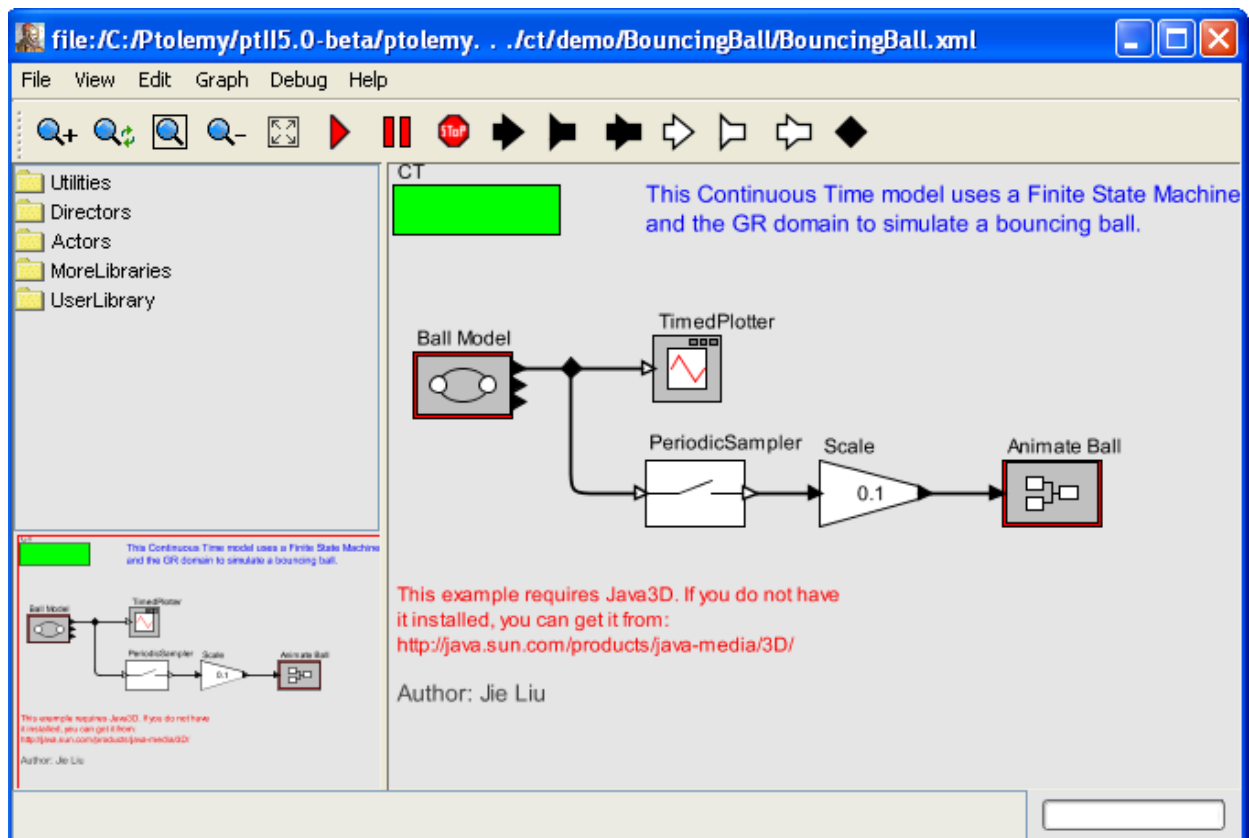


Рис. 1.3. Редактор моделей Vergil

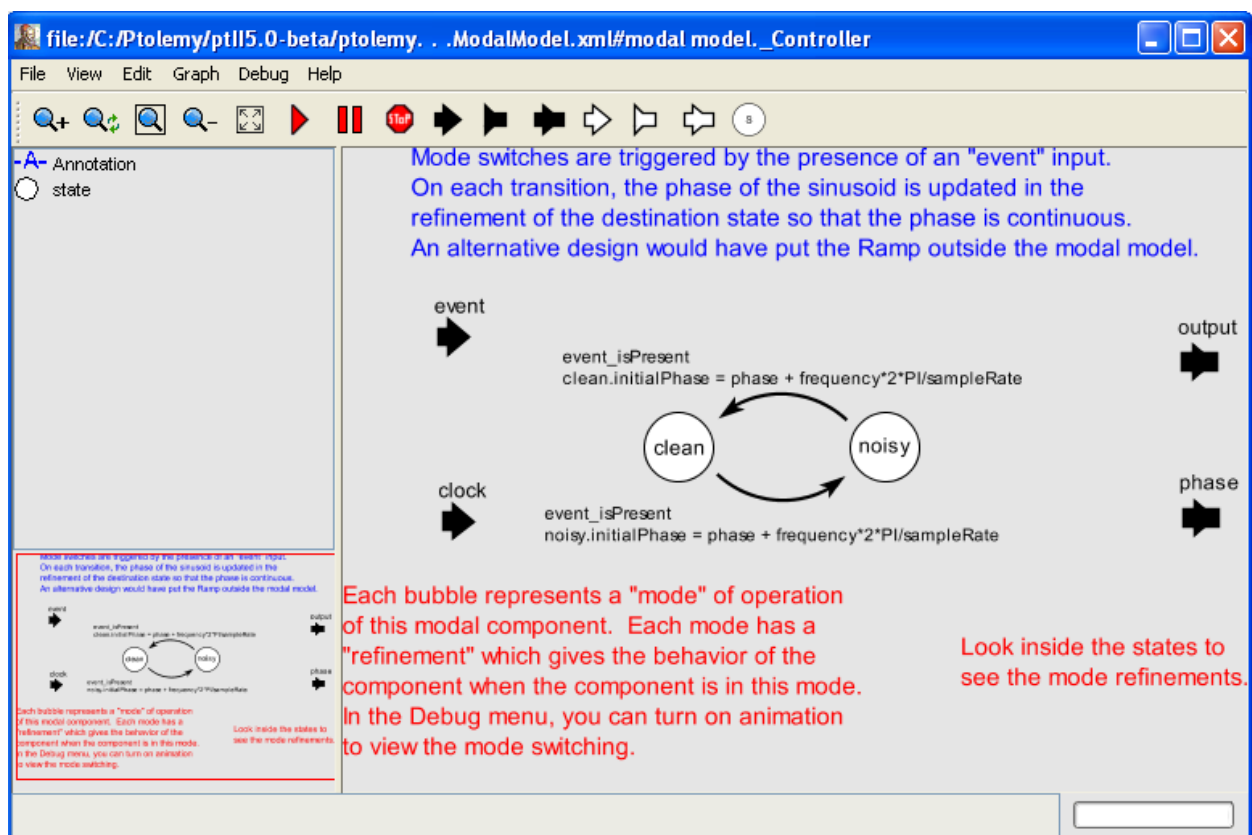


Рис. 1.4. Редактор конечных автоматов Vergil

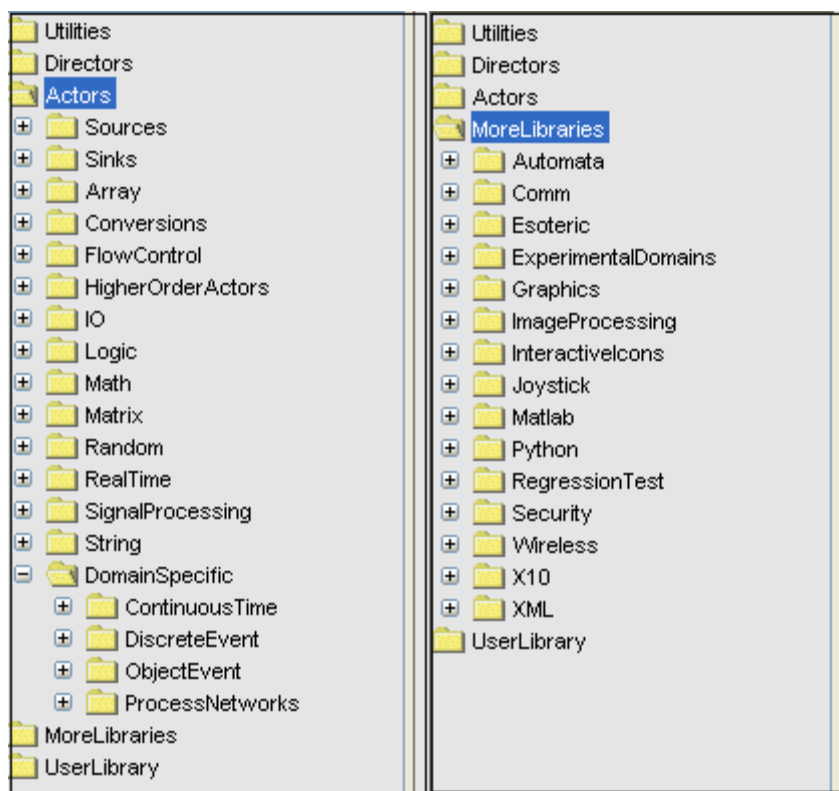


Рис. 1.5. Библиотека акторов Ptolemy II

Большинство акторов стандартной библиотеки Ptolemy II обладает полиморфизмом функциональности относительно модели вычислений, в которой они используются. Этот механизм является одним из базовых механизмов Ptolemy II и обеспечивает широкие возможности поведенческого описания в рамках почти любой модели вычислений пользователем в графическом редакторе Vergil, не прибегая к кодированию на Java или существенной модификации стандартных акторов.

Еще одним базовым механизмом является полиморфизм функциональности акторов относительно типа преобразуемых данных. Большинство акторов могут реализовывать свое поведение над сигналами различных типов. Этот механизм в значительной степени опирается на существующую в Ptolemy II систему типов и механизм их разрешения при инициализации модели. Он реализован за счет построения иерархии классов, которые представляют элементы данных различных типов, и реализации множества операций, смысл которых инвариантен к типу данных.

Типы данных, с которыми может работать модель в Ptolemy II, организованы в систему, которая сравнима по сложности с системой типов C++ или Java (рис. 1.6). Помимо простых типов данных (*int*, *double*, *long*, *unsignedByte*, *complex*, *fixedpoint*, *boolean*, *string*, *scalar*, *matrix* и т.д.), существуют составные типы (массивы, матрицы, записи и объекты). Компоненты модели, то есть акторы, обмениваются друг с другом в процессе ее исполнения элементами данных, имеющими тот или иной тип. С помощью встроенного в Ptolemy II языка выражений и механизма разрешения типов

данные, имеющие один тип, могут быть явно или неявно преобразованы к другому типу.

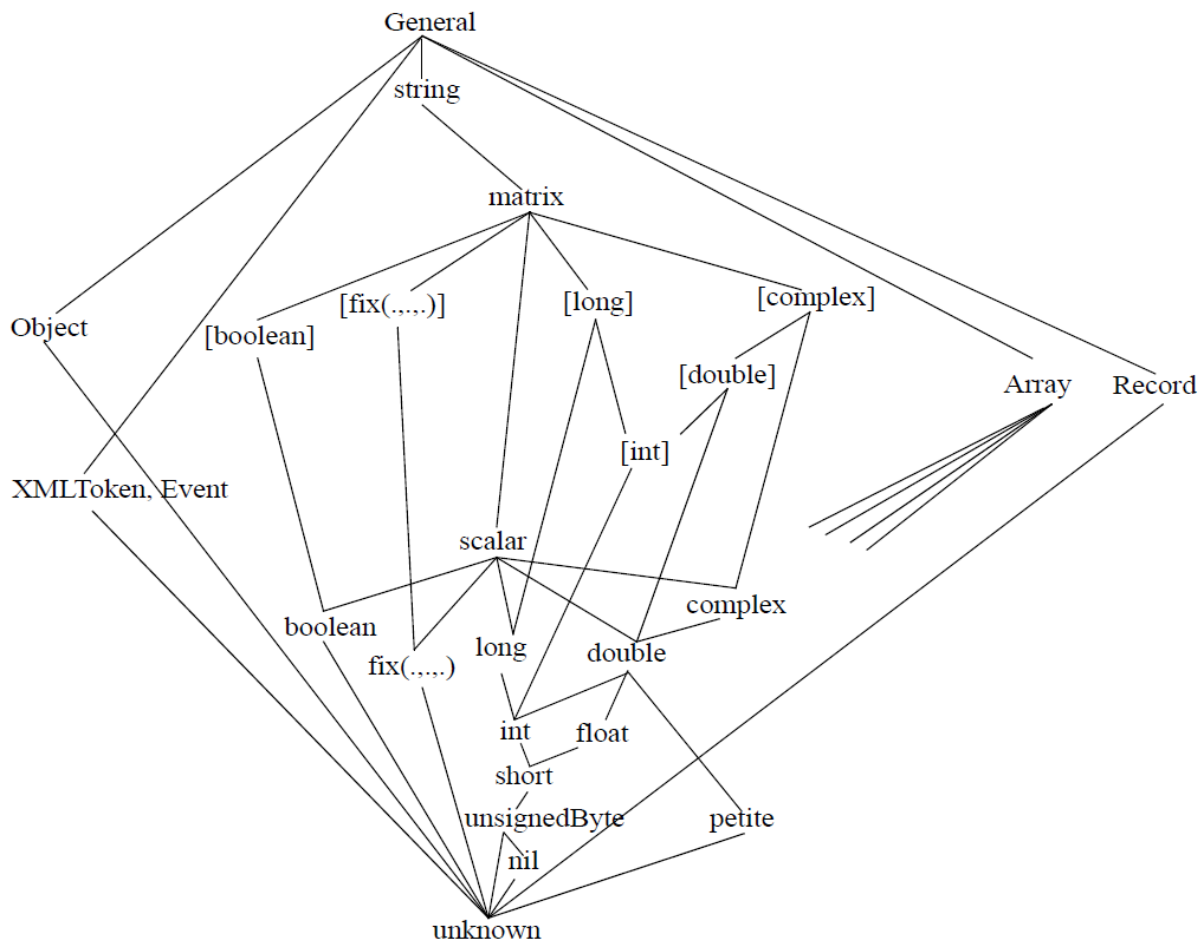


Рис. 1.6. Граф типов данных Ptolemy II. Типы данных, которые находятся ниже на рисунке, могут быть преобразованы к связанным с ними линиями типам, которые находятся выше них

Массивы в Ptolemy II определяются как упорядоченные наборы данных одного типа, имеющие одно измерение. Матрицы представляют собой двумерные упорядоченные наборы данных одного типа. Ptolemy II поддерживает матрицы, состоящие из элементов следующих типов: *boolean*, *complex*, *double*, *fixedpoint*, *int* и *long*. Другие типы элементов в матрицах не разрешены. Для матриц предусмотрено множество операций векторно-матричной алгебры и функций, практически все из которых присутствуют в MATLAB (MathWorks Ltd.). Записи в Ptolemy II состоят из именованных полей, которые могут иметь разные типы. Доступ к отдельным полям осуществляется с помощью операции '.' (как в C++ или Java). Интересно отметить, что для данных, имеющих тип «запись», предусмотрены бинарные операции пересечения и объединения по именам полей.

Механизм разрешения типов Ptolemy II позволяет автоматически проверять корректность преобразований данных в модели в целом, не указывая вручную конкретные типы или ограничения на порты отдельных акторов. Разрешение

типов происходит в процессе инициализации модели и устанавливает для каждого порта актора конкретный тип данных, которые он воспринимает, исходя из информации о зависимостях между типами данных портов, декларируемой акторами. Ограничения, накладываемые на порт, могут быть менее строгими, чем указание его конкретного типа, благодаря возможности преобразования данных одних типов к другим (рис. 1.6).

Например, ограничение на порт может заключаться в том, что тип данных, которые он воспринимает, может быть больше или равен конкретному типу. Под «больше» понимается то, что данный тип соединен нисходящим путем в графе на рис. 1.6. В случае несоответствия ограничениям, то есть невозможности разрешения типов, инициализация модели завершается с ошибкой. Ошибка может быть устранена вручную.

В Ptolemy II разработан довольно мощный язык выражений, позволяющий не только выполнять математические операции над данными, получая значение результата, но и создавать переменные различных типов, в том числе и произвольных составных, выполнять преобразования типов, выполнять сложные встроенные процедуры обработки данных и создавать собственные, производить ввод/вывод и управлять вычислительным процессом. В языке выражений предусмотрен набор стандартных именованных констант, которые могут быть использованы в выражениях. Это числовые константы: *PI*, *pi*, *E*, *e*, *true*, *false*, *i*, *j*, *NaN*, *Infinity*, *PositiveInfinity*, *NegativeInfinity*, *MaxUnsignedByte*, *MinUnsignedByte*, *MaxInt*, *MinInt*, *MaxLong*, *MinLong*, *MaxDouble*, *MinDouble*; и строковые константы *PTII*, *HOME* и *CWD*. Эти константы имеют заранее определенное значение. Константы с другими значениями могут быть определены с помощью литералов, задающих как значение, так и (либо явно, либо неявно) тип константы. Например, *2.0* даст константу типа *double*, *2i* (или *2j*) даст константу, имеющую тип *complex*. Константы составных типов также задаются с помощью литералов:

- массивы с использованием фигурных скобок с элементами, разделяемыми запятыми, например: {2, 3, 4, 5};
- матрицы задаются с помощью квадратных скобок и деления столбцов запятыми, а строк знаками ‘;’, например: [1,2,3; 4,5,6] – матрица из двух строк и трех столбцов. В языке выражений также поддержан способ задания матриц в стиле MATLAB ([r;q:r]);
- записи задаются с помощью фигурных скобок, в которых перечислены имена полей и их значения, например: {first_name = "Edward", data = {0, 1, 2}, associated_record = {a = 1, b = 2 + PI * j}}.

В языке выражений предусмотрено задание переменных и функций с произвольными именами. Операции, предусмотренные в языке выражений, включают как алгебраические (над числами, строками, массивами, записями и матрицами), так и логические: сравнения, побитовые (&|~#>><<), преобразования типов и условного вычисления (‘ ? : ’). Также в языке

предусмотрены комментарии в стиле C++ (`/* */`). Достаточно объемная библиотека стандартных функций, включающая также и статические методы стандартных классов Java (`java.lang.Math`, `java.lang.Double` и т.п.), позволяет реализовать достаточно сложные вычисления.

Выражения могут быть использованы в редакторе выражений Vergil при задании значений параметров в отдельных акторах, а также в качестве элементов данных (рис. 1.7).

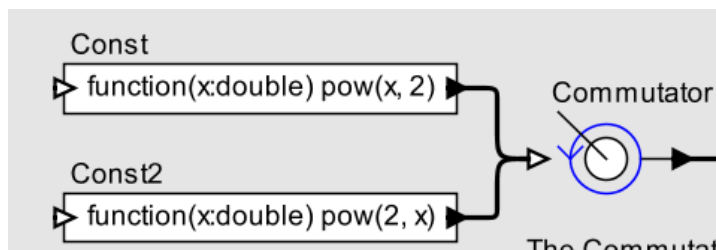


Рис. 1.7. Использование языка выражений в акторах

В Ptolemy II существует группа акторов, называемая «компонентами высшего порядка» (Higher-Order Components, HOC). Эти акторы выполняют действия над структурой и функциональной организацией модели в процессе ее исполнения. Примеры акторов этого типа:

- **ModelReference** – актор, представляющий собой ссылку на модель, заданную с помощью URL. Активация этого актора представляет собой команду на исполнение модели, на которую он ссылается;
- **MultiInstanceComposite** – составной актор, создающий заданное количество копий самого себя в процессе инициализации модели;
- **ModalModel** – актор, реализующий модель, которая может работать в нескольких «режимах». Каждый такой «режим» представляет собой произвольную модель с теми же портами, что у **ModalModel**, а переход между ними осуществляется с помощью конечного автомата. Актор может быть использован для моделирования гибридных систем (дискретная логика в сочетании с непрерывной динамикой).

В Ptolemy II реализован механизм объявления и наследования классов, реализующий механизмы из объектно-ориентированного подхода для составных акторов (рис. 1.8). Составной актор может реализовываться в графическом редакторе в рамках какой-либо модели вычислений, а затем может быть преобразован в описание «класса». Экземпляры этого «класса» затем могут быть использованы в модели как обычные составные акторы. Любые изменения в описании класса ведут к изменениям в его экземплярах. Этот подход удобен, когда в модели верхнего уровня иерархии используется несколько однотипных компонентов, преобразующих различные потоки данных.

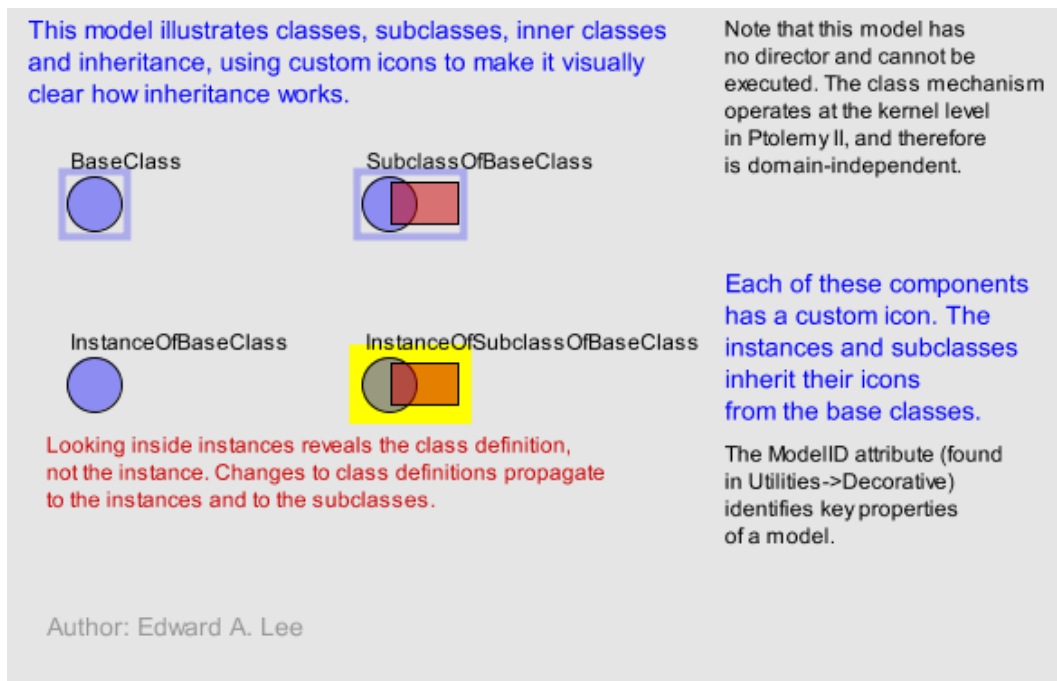


Рис. 1.8. "Классы" составных акторов

Кроме того, механизм позволяет строить иерархию «классов», «наследуя» от базовых «классов» все элементы модели нижнего уровня, добавляя в них новые компоненты и «перегружая» часть наследуемых.

Реализация Ptolemy II на базе Java™ позволяет пользоваться всеми возможностями, которые предоставляет эта платформа. В частности, стандартная библиотека Ptolemy II включает акторы, позволяющие:

- пользовательский ввод и интерактивное управление моделированием и структурой модели в процессе ее исполнения;
- ввод, вывод и обработку сигналов и изображений, включая преобразование видеосигналов в реальном масштабе времени;
- файловый ввод-вывод и доступ к устройствам инструментального ПК;
- доступ к ресурсам сети Internet и распределенное моделирование;
- интеграцию с другими инструментами и языками.
- Модели в Ptolemy II представляют собой Java-приложения, выполнением которых управляет один из специально созданных классов. В зависимости от используемых моделей вычислений, приложения могут быть как однопотоковыми, так и многопотоковыми (например, при использовании модели вычислений PN). Такой подход позволяет гибко использовать как сами модели, так и предоставляемые ими результаты. Например, можно получить готовую отлаженную модель, включающую средства отображения результатов моделирования, и оформить ее в виде самостоятельной программы (отдельного исполняемого модуля). Или можно создать Java-

апплет и опубликовать в Internet, обеспечив возможность удаленной загрузки готовой модели, ввода исходных данных и получения результатов моделирования на любом ПК, в котором реализована поддержка Java™ (рис. 1.9).

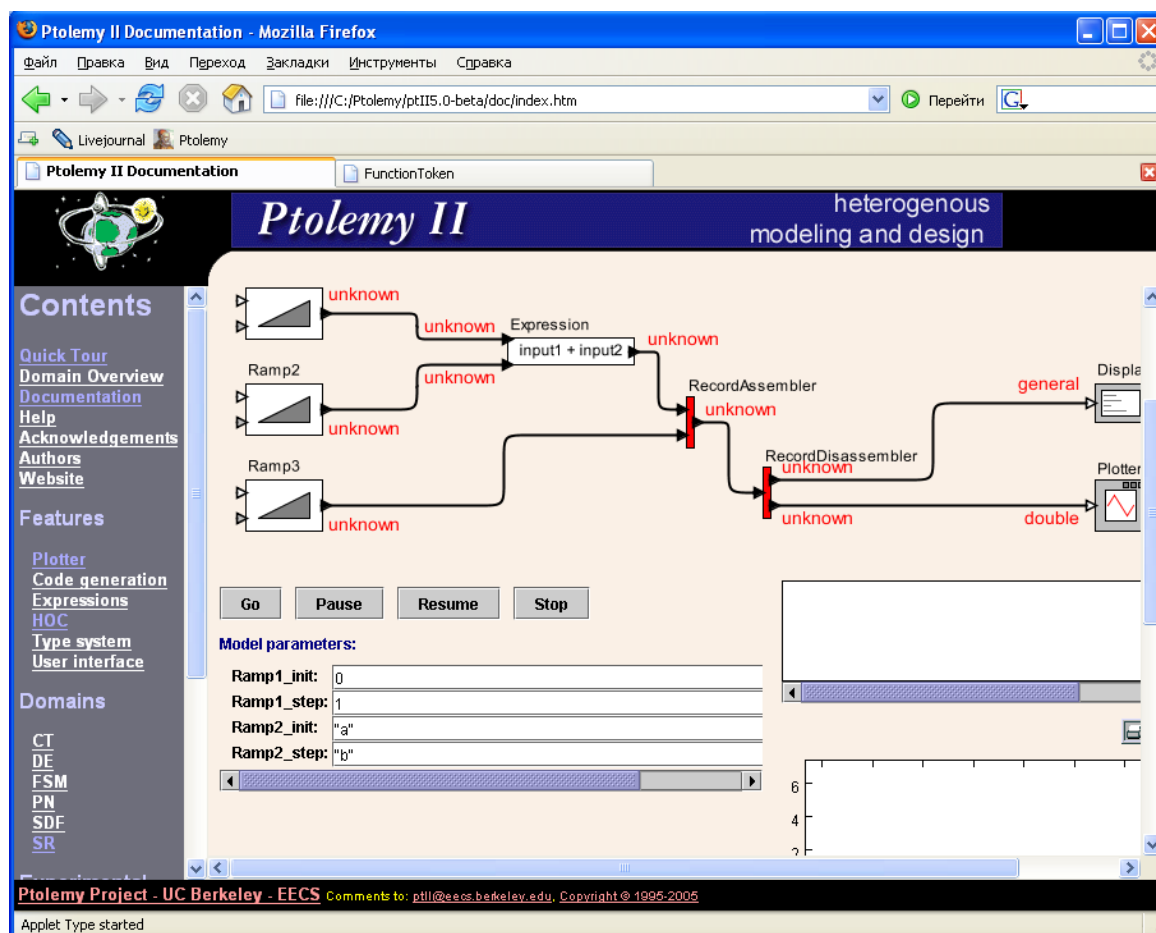


Рис. 1.9. Оформление модели Ptolemy II в виде апплета

Наконец, можно исключить весь графический ввод-вывод модели, существенно снизив вычислительную нагрузку на операционную систему, и запустить модель в виде консольной программы Java, обеспечив вывод результатов моделирования в файл. Этот подход удобен, если объем вычислений и время моделирования существенны, в отличие от анимации и диалога с пользователем.

1.3 Объектно-событийная модель вычислений распределенных ВсС

1.3.1 Общие положения объектно-событийной модели

Проанализировав принципы функционирования различных РИУС, можно сделать вывод, что зачастую для алгоритма управления объектом нужны не непосредственно сами данные определенных входов, т.е. состояние входа, а

события, происходящие на этих входах, одним из атрибутов которого является собственно состояние входа. Природа конкретного события может быть различной, она связана с особенностями объекта управления (внешней среды). *Локальным временным масштабом объекта управления (внешней среды)* называется минимальный интервал времени между событиями, которые эта среда производит. Временной масштаб возникновения тех или иных событий зависит от объекта управления (внешней среды) и может значительно различаться для разных частей системы.

Событием для РИУС может быть нажатие оператором на кнопки управления, выход напряжения за установленные границы, пропадание или нарушение несущей частоты, возникновение импульса заданной формы и т.д. Несомненно, что для выделения определенных событий необходимы значительные вычислительные ресурсы и малые временные масштабы, но в целом само событие при этом может быть достаточно простым и иметь относительно большой временной масштаб. Примером такого события может служить выход за определенные границы частоты входного сигнала. Определение значения входной частоты требует значительных вычислительных ресурсов и производительности микроконтроллера. Само изменение частоты может происходить очень медленно, что и определяет относительно большой временной масштаб события.

Управление объектом также очевидно производится в виде событий. Даже управляя статическим выходом, нет необходимости постоянно обновлять значение этого выхода, если оно не изменялось. Опять же, как и в случае входных событий именно форма выходного сигнала может потребовать вычислительной мощности. Например, выходным сигналом управления является гармонический сигнал определенной частоты. Само изменение выходной частоты происходит относительно редко, в то время как поддержание формы гармонического сигнала заданной частоты потребует значительных вычислительных затрат.

Очевидно, что все внутренние механизмы тактирования системы опять же генерируют события. Даже если на вход системы поступает сигнал не событийный по природе, зачастую его необходимо преобразовывать и производить предобработку, чтобы выделить действительно произошедшие изменения в объекте управления.

Структурная схема системы, для которой распознавание входных событий и формирование выходных сигналов является наиболее ресурсоемкой частью, приведена на рис. 1.10.

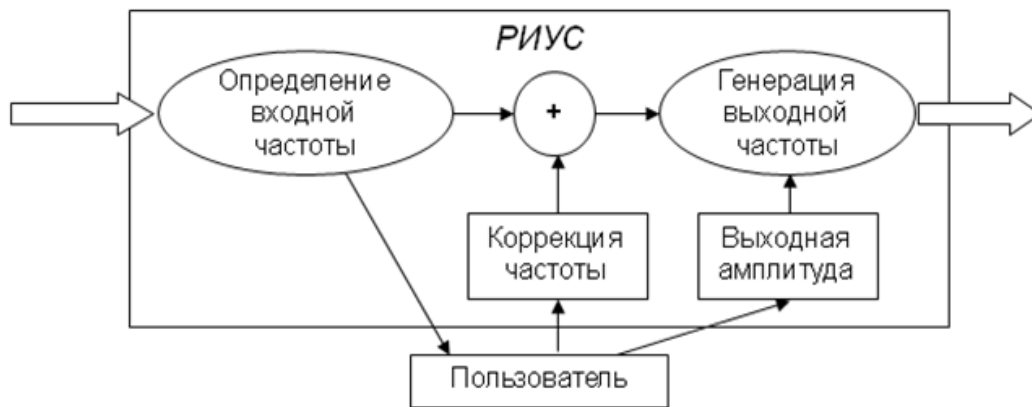


Рис. 1.10. Структурная схема РИУС, решающей задачу коррекции формы сигнала

Ядром системы является именно блок преобразования входной частоты с учетом корректирующего значения. Видно, что это относительно простая задача, не требующая значительных вычислительных ресурсов. Большинство событий в системе имеют большой временной масштаб. Малым временным масштабом и значительной ресурсоемкостью обладают блоки определения входной частоты и генерации выходной частоты. В общем случае, чтобы не нагружать все остальные части системы, такие блоки должны быть локализованы и реализованы на специальных узкоспециализированных вычислителях. Особенно это актуально для РИУС, где эти два блока могут быть разнесены на значительные расстояния.

На описанном примере демонстрируется влияние временных масштабов на построение системы. В принципе *уменьшение временных масштабов ведет к значительному удорожанию системы*. Уменьшение временных масштабов требует повышения тактовой частоты системы, реализации специфических алгоритмов, использованию более производительной элементной базы, что приводит к увеличению энергопотребления, возрастанию массы и габаритов системы, повышению технологичности.

Основными элементами предлагаемой объектно-событийной модели вычислений (ОСМВ) РИУС являются объекты. В рамках описания модели понятия “объект” и “функциональный блок” являются синонимами, так что для более точного отражения места в модели и чтобы не возникла терминологическая путаница (объект применяется достаточно широко и в других смыслах), в дальнейшем будет использовано понятие функциональный блок. ФБ, обладая свойствами полиморфизма и инкапсуляции, позволяют абстрагироваться от специфики реализации частей системы и сосредоточиться на функциональных характеристиках отдельных частей.

ФБ осуществляют взаимодействие посредством событий. Задачей ФБ становится преобразование событий, результатом чего является порождение новых событий, распространяющихся по системе и в конечном итоге достигающих внешней среды или объекта управления. Первичное событие также порождается внешней средой или объектом управления. Следует

отметить, что ФБ самостоятельно не может выступать инициатором события. Единственный способ для ФБ сгенерировать событие – обработать входные события. При этом для каждого выхода ФБ любое входное событие не может сгенерировать более одного выходного события, и одно входное событие может породить несколько выходных событий по разным выходам, но не более одного на каждый выход. В некоторых случаях помимо внешней среды инициаторами событий могут являться схемы тактирования и другие времязадающие механизмы.

Для восприятия входных событий и генерации выходных событий каждый ФБ имеет соответствующие интерфейсы. Интерфейсы передачи событий состоят из портов. Порты могут быть двух типов: входные и выходные. ФБ воспринимает события по входным портам и генерирует выходные события по выходным портам. Каждый выходной порт может быть связан с несколькими входными портами. Каждый входной порт может быть связан только с одним выходным. Порты, не имеющие связи с другими портами [на данном уровне иерархии], являются внешними портами системы [на данном уровне иерархии]. Входные порты ФБ могут быть двух типов: синхронные и асинхронные. Появление события на синхронном порте “запускает” ФБ и он начинает выполнять свою целевую функцию, обрабатывая событие и генерируя выходные события. Появление события на асинхронном порте не активизирует ФБ. Аналогией такого поведения в языках описания аппаратных проектов является понятие “списка чувствительности”, который регламентирует перечень сигналов системы, при изменении которых процесс активизируется.

Характер событий на синхронных портах ФБ очень сильно влияет на функционирование системы в целом. В каждом конкретном случае разработчик вправе применить те или иные способы распределения вычислительных и временных ресурсов ФБ при реализации параллельной обработки нескольких событий, для выполнения заданных временных соотношений.

1.3.2 Элементы объектно-событийной модели

При использовании предлагаемой ОСМВ разработчик имеет дело со следующими сущностями:

- *Объект (функциональный блок).* ФБ являются носителями поведенческих аспектов проектируемой системы. В узком смысле они являются реализацией алгоритмов обработки данных и формирования сигналов управления. Обладают свойствами полиморфизма и инкапсуляции, позволяют создавать иерархические модели, зачастую являясь композицией более мелких ФБ. Каждый отдельно взятый ФБ можно рассматривать как самостоятельную систему.
- *Событие.* Все элементы модели связаны некоторыми сигналами. Изменение (активизация) сигнала называется событием. События характеризуются временем возникновения и, в случае необходимости, данными (состоянием)

сигнального входа. События переносят данные между ФБ и активизируют последние для выполнения тех или иных действий.

- *Порт.* Порт является механизмом передачи событий между ФБ. Обмен событиями ФБ осуществляет именно с помощью портов. Порты могут быть двух типов: входные и выходные. ФБ воспринимает события по входным портам и генерирует выходные события по выходным портам. Входные порты ФБ могут быть двух типов: синхронные и асинхронные. Событие на синхронном порте активизирует ФБ.
- *Узел.* Физический носитель вычислительных ресурсов целевой системы, реализующий “отображенные” на него ФБ и порты. При реализации модели узлы системы обеспечивают выполнение следующих задач:
 - о Предоставление необходимых ресурсов реализациям ФБ;
 - о Поддержка всех необходимых форматов данных событий;
 - о Реализация необходимого количества портов ФБ;
 - о Обеспечение синхронизации событий.

ФБ реализует часть целевой функции, обладает набором входов/выходов (входных и выходных портов), назначен (отображен) на конкретный узел (набор узлов) целевой РИУС. В принципе, работать с моделью можно и не проводя отображения ФБ на целевые узлы. ФБ можно сформировать двумя способами:

1. ФБ является выражением конкретного алгоритма, заданного тем или иным способом;
2. ФБ является композицией других, более мелких ФБ.

Имея в виду эти способы получения ФБ, можно выделить достаточно важный класс ФБ – атомарных. *Атомарным (примитивным) функциональным блоком (АФБ)* называется ФБ, который обладает законченной и конкретной функциональностью и имеет некоторую реализацию (возможно несколько различных реализаций) в рамках целевой вычислительной платформы. Очевидно, что АФБ не нуждается в дальнейшей структурной декомпозиции на более мелкие ФБ. Иерархичность модели реализуется за счет представления более крупных ФБ (на верхних уровнях иерархии) в виде композиции более мелких.

Для абстрагирования от специфики программной или аппаратной реализации частей системы необходимо выбрать подходящие средства описания. Очень важным отличием аппаратных реализаций от программных реализаций является параллелизм. Параллелизм в аппаратных реализациях проявляется двояко: параллелизм во времени и параллелизм в использовании ресурсов. Программа же трактуется как упорядоченная во времени последовательность операций, при этом традиционно ось времени единственная и все операции выполняются строго последовательно. При

реализации программных алгоритмов вполне возможно разделение таких ресурсов системы, как процессорное время (отдельные функции могут выполняться последовательно и независимо), так и память (отдельные функции, выполняясь последовательно, могут использовать одни и те же ячейки памяти).

Аппаратная схема трактуется как набор подсистем, существующих и работающих одновременно и выполняющих одно и то же действие во времени. Так как традиционно вся функциональность аппаратной реализации существует одновременно (параллельность во времени), невозможно так же легко как в случае программной реализации обеспечить разделение ресурсов памяти кода и данных (параллельность в ресурсах). Очевидно, что в особых случаях можно применить различные техники распараллеливания вычислений, чтобы сделать одновременным выполнение функций в программной реализации и обеспечить последовательность в аппаратной реализации.

Переход от “разделяемых” локальных переменных к строго выделенным ресурсам имеет смысл только в случае, если интерпретатор имеет возможность обрабатывать наборы данных за одну инструкцию, так называемые SIMD (Single-Instruction Multiple-Data) процессоры. В противном случае сокращения времени выполнения функций не добиться. Еще нужно иметь в виду, что для объединения вычисления функций необходимо, чтобы к определенному моменту были готовы данные для обеих ветвей алгоритма. Имея в распоряжении только один последовательный интерпретатор, нет необходимости создавать отдельные копии кода функции в памяти. В РИУС зачастую дело обстоит иначе. В распоряжении разработчиков имеется множество вычислителей, и данные для функций могут быть готовы совершенно в различные моменты времени, так что объединить их выполнение на практике будет невозможно.

Тот или иной способ реализации вычислительного процесса не влияет на общую модель системы на высоких уровнях рассмотрения. Данное утверждение позволяет сделать вывод о том, что представление на уровне ФБ позволяет абстрагироваться от специфики реализации отдельных вычислительных механизмов и дает возможность описать систему без предварительного разбиения на программную и аппаратную составляющие. На последнем этапе проектирования при генерации кода программного компонента и аппаратного компонента ФБ могут быть заменены аналогичными элементами моделирования в выбранных языках реализации. Например, при генерации кода программного компонента на языке C/C++ ФБ адекватно замещается объектом, при генерации кода аппаратного компонента на языке VerilogHDL ФБ замещается модулем (module). Во всех случаях иерархия ФБ моделей осуществляется за счет использования реализаций более мелких ФБ при описании структуры более крупных ФБ. То же самое можно утверждать про создание объектов/модулей в упомянутых языках.

Для проведения моделирования необходимо абстрагироваться от способа внутренней организации конкретного ФБ и тонкостей его реализации.

Модельными параметрами ФБ должны явиться свойства, не зависящие от его организации и реализации. Такими важными при моделировании параметрами могут стать временные характеристики различного рода.

Рассмотрим несколько сигналов $s^1, s^2, s^3, \dots s^{m+n}$. Пусть I – набор входных сигналов ($s^1, s^2, \dots s^m$), а O – набор выходных сигналов ($s^{1+m}, s^{2+m}, \dots s^{n+m}$) некоторого ФБ. Временные характеристики ФБ определяются тремя параметрами.

- *Сложность генерации выходного события τ* . Данный параметр представляет собой матрицу, состоящую из n строк и m столбцов. Элемент матрицы $\tau_{i,j}$ представляет собой количество событий, которые необходимо обработать по входу j , чтобы сформировать событие на выходе i . Очевидно, по построению, что

$$\forall i (0 < i \leq n) \quad \forall j (0 < j \leq m) \quad \tau_{i,j} \geq 0, \tau_{i,j} \in \mathbb{Z}. \quad (1.6)$$

- *Минимальный интервал следования входных событий*. Для каждого ФБ имеются ограничения на следование входных событий. Эти ограничения связаны с физикой происходящих внутри блока процессов и практически ни при каких условиях не могут быть нарушены. К таким ограничениям относятся максимальная скорость работы ЦАП или АЦП, максимальная частота тактирования аппаратных схем и различных микроконтроллеров. Для АФБ этот параметр определяется по документации и может быть только ухудшен. Параметр представляет собой числовые значения временных интервалов для каждого входа ФБ.
- *Минимальный интервал генерации выходных событий*. Обладая определенной производительностью и реализуя некоторый алгоритм обработки событий, каждый ФБ в состоянии генерировать выходные события с интенсивностью, ограниченной сверху. Например, ФБ, делящий входную частоту φ_{in} в N раз, не в состоянии выдавать выходные события чаще, чем $\frac{\varphi_{in}}{N}$.

Временным масштабом функционального блока называется минимальный временной интервал следования входных и выходных событий.

Временной масштаб функционального блока играет огромную роль при выделении временных доменов в ОСМВ, а значит и в целевой системе. *Временным доменом функциональных блоков* называется множество связанных друг с другом портами ФБ, сформированное по критерию рассогласования временных масштабов элементов (1.7):

$$\sum_{f_i \in D} (M(f_i) - \bar{M}(D))^2, \quad (1.7)$$

где D – множество функциональных блоков (временной домен), $M(f_i)$ – временной масштаб ФБ f_i , $\bar{M}(D)$ – среднее значение временного масштаба домена. При декомпозиции функциональности задачей разработчика

распределенной системы становится выделение временных доменов [минимизация критерия (1.7)], которые при реализации будут локализованы на соответствующих узлах. Критерий выделения временных доменов ФБ является одним из методов структурной декомпозиции сложных ФБ при аналитическом построении модели. Декомпозицию ФБ необходимо проводить до тех пор, пока не будут получены АФБ.

Отталкиваясь от понятия временного масштаба, можно формализовать понятия сильной и слабой связи для вычислителей РИУС. Сам коммуникационный канал, осуществляющий связь между вычислителями, является одним из ФБ системы. Вычислители, которые данный канал связывает, тоже являются ФБ. Для всех ФБ определен временной масштаб. Чтобы вычислители были равноправными и можно было вести речь о сильной или слабой связи между ними, их временные масштабы должны совпадать. При сильной связи временной масштаб коммуникационного канала сравним с временными масштабами связываемых вычислителей, при слабой – значительно больше. Необходимо обратить внимание, что учитывается временной масштаб ФБ, решающего целевую задачу, а не частоты тактирования микропроцессоров и канала связи.

При переходе между уровнями иерархии в процессе проектирования временной масштаб изменяется. Временной масштаб ФБ не может быть меньше, чем максимальный временной масштаб ФБ, из которых он состоит. Сбалансированным функциональным разбиением по критерию сильной/слабой связи можно считать такое разбиение на ФБ, при котором временной масштаб сильно связанных доменов примерно одинаков и меньше, чем временной масштаб ФБ, осуществляющих слабую связь (1.8)

$$\frac{\overline{M}(D_i)}{\overline{M}(D_j)} < k, \quad (1.8)$$

где $\overline{M}(D)$ – среднее значение временного масштаба домена, k – показатель связности временных доменов ФБ.

Локализованный на одном узле временной домен характеризуется сильной связью между составляющими его ФБ. Собственно сильная связь характеризуется близкими значениями временного масштаба, что приводит к упорядочиванию временных меток событий в рамках одного сильно связанного домена. Слабая связь между такими доменами огрубляет временной масштаб, увеличивая его, и упорядоченность временных меток событий имеет совершенно другой характер. Таким образом можно сделать вывод о локальной упорядоченности временных меток событий в рамках сильно связанного домена ФБ. О глобальной упорядоченности временных меток событий в рамках всей системы можно говорить, только учитывая погрешность временных масштабов слабых связей.

Важной характеристикой ФБ является его *временная передаточная матрица*. Основной задачей ФБ является обработка входных событий и генерация выходных. На эту обработку узел, на который “отображен” ФБ, тратит определенные вычислительные ресурсы, а, следовательно, и время. Т.е. для каждой пары из всевозможных наборов вход $\rightarrow$ выход ФБ можно ввести такой параметр как время распространения события. Нужно сразу сказать, что такой параметр имеет смысл только для пар, в качестве левой части у которых указан синхронный порт ФБ (события на асинхронных портах не активизируют ФБ, и, следовательно, не передаются на выходы ФБ). Кроме того, для относительно больших ФБ, объединяющих в себе несвязанные друг с другом функции системы, можно заметить, что событие по синхронному входу далеко не всегда генерирует события по всем выходам. Зачастую это всего лишь небольшое подмножество выходов ФБ. Временные передаточные матрицы ФБ могут быть получены двумя способами: синтетическим способом (снизу вверх) и аналитическим способом (сверху вниз).

Временной масштаб может быть значительно увеличен, так как микроконтроллеры исполняют алгоритм, выраженный многими тысячами инструкций микроконтроллера, а помимо частоты тактирования в канале данных могут существовать определенные протоколы, которые увеличивают масштаб времени. С другой стороны следует различать расчетный и фактический временные масштабы ФБ. Расчетный временной масштаб достигается, когда все ресурсы ФБ направлены на обработку событий. На практике такие ситуации встречаются достаточно редко и фактический временной масштаб (временной масштаб, который должен обеспечить ФБ, чтобы решить целевую задачу) бывает меньше, а порой значительно меньше расчетного. Очевидно, что говорить о тех или иных фактических временных масштабах имеет смысл только при отображении ФБ на те или иные элементы вычислительной платформы.

Такое отображение нужно для получения фактических значений параметров ФБ. К тому же кроме задачи *декомпозиции функциональности на заданную топологию целевых узлов* (платформу), самостоятельный интерес представляет задача *выбора оптимальной топологии и состава целевых узлов для заданной модели* (функциональной декомпозиции).

При проектировании всей системы целиком на начальном этапе она рассматривается как один большой ФБ, временные параметры которого можно взять из спецификаций на систему. По этим временным параметрам необходимо сформировать первую временную передаточную матрицу. В дальнейшем при структурной декомпозиции ФБ, анализируя получаемое каждый раз разбиение, разработчик, опираясь на приведенные критерии (1.7) и (1.8), экспертные оценки и свой собственный опыт, составляет оптимальные временные передаточные матрицы мелких ФБ. При этом разработчик должен обеспечить корректность декомпозиции матрицы, т.е. временные передаточные матрицы мелких ФБ не должны противоречить временной передаточной матрице

основного ФБ. Двигаясь таким образом, каждый раз анализируя разбиение, разработчик получает оптимальные (желаемые) временные передаточные матрицы АФБ.

Если разработчик начнет двигаться от АФБ к более крупным блокам, то на каждом шагу он будет получать временные передаточные матрицы синтетическим способом. При таком способе получения рассматриваемых характеристик их значения получаются с помощью конкретных аналитических выражений и инструментальных средств.

АФБ, обладающий законченной и конкретной функциональностью и отображенный на конкретный целевой узел, позволяет достаточно точно оценить его ресурсоемкость и другие параметры (процессорное время, ресурсы памяти, объем логических ячеек, энергопотребление и т.д.). Основываясь на таких конкретных данных, разработчик имеет возможность составить фактическую (для данного отображения) временную передаточную матрицу. В дальнейшем при композиции мелких ФБ в более крупные, очередные временные передаточные матрицы будут синтезированы автоматически. При этом такие синтезированные временные передаточные матрицы будут содержать фактические данные, характерные именно для рассматриваемого отображения на целевые узлы.

Преимуществом первого способа является то, что для его осуществления нет необходимости производить отображение ФБ на целевые узлы. Разработчик конкретизирует функциональность отдельных частей системы, одновременно формируя временные требования на реализацию. Недостатком следует считать принципиальную возможность того, что для финального разбиения на АФБ не найдется реализации в элементной базе выбранной А-платформы, удовлетворяющей желаемым временным передаточным матрицам.

Преимуществами второго способа получения временных передаточных матриц являются:

- оперирование фактическими временами, полученными при анализе АФБ;
- относительно формальный метод синтеза крупных временных передаточных матриц.

К недостаткам такого метода следует отнести необходимость рассмотрения полностью проработанной модели, отображенной на целевые узлы системы, и необходимость проектирования снизу вверх, что противоречит основным тенденциям стиля проектирования в настоящее время. Кроме того, полученная синтетическим методом временная передаточная матрица всей системы целиком может не удовлетворять требованиям к системе. Следует указать, что синтетический метод получения временных передаточных матриц будет отлично работать при оценке модели с целью проведения верификации.

Единственным выходом в сложившейся ситуации видится совмещение указанных двух способов. Проводя аналитическое построение частных

временных передаточных матриц, помимо опыта и экспертных оценок, разработчик может использовать синтетический метод получения фактических временных передаточных матриц для сложных частей модели, значительно упрощая их и наискорейшим образом приводя к АФБ. Для известных разработчику А-платформ и знакомых композиций ФБ, экспертные оценки и опыт могут дать прекрасный результат и без синтеза фактических временных передаточных матриц.

Необходимо отметить, что использовать непосредственно временную передаточную матрицу ФБ не слишком удобно. Проблема заключается в том, что один и тот же АФБ, отображенный на разные узлы целевой системы, будет иметь различные временные передаточные матрицы. Это связано с тем, что в принципе одна и та же ресурсоемкость реализации алгоритма для разных физических вычислителей реально может потребовать различных ресурсов. Инвариантной к отображению ФБ на элементы вычислительной платформы является описанная выше матрица сложности генерации выходного события [см. формулу (1.6)].

Если рассмотреть пример стека виртуальных машин (табл. 1.2), то видно, что эффективность реализации алгоритма АФБ в конечном итоге зависит от эффективности реализации каждого уровня стека в базисе нижележащего уровня. Кроме того, на этапе оценки эффективности такого рода реализаций необходимо учитывать качество используемых инструментальных средств.

Таблица 1.2. Пример стека виртуальных машин

Уровень представления алгоритма	Количество событий
Средства функционального блока	1
Прикладной язык	10
Язык высокого уровня	100
Система команд	1000
Структурно-алгоритмический базис	5000
Механизмы тактирования	10000

Видно, что эффективность реализации каждого уровня в представленном стеке виртуальных машин непосредственно влияет на конечную эффективность реализации алгоритмов ФБ. Связь с временными параметрами А-платформы имеет только самый нижний уровень стека, а именно механизмы тактирования.

Особенности предлагаемой ОСМВ в том, что описанный стек виртуальных машин принадлежит не к модели, а к А-платформе, на которую данная модель должна быть отображена. ФБ может только косвенно влиять на представленный стек виртуальных машин, а именно путем выбора прикладного языка или языка

высокого уровня, в терминах которого выражен ФБ. Однако замена прикладного языка приводит просто к конфигурированию А-платформы (смене МоС).

Матрица сложности генерации выходного события ФБ зависит только от алгоритма и средств его выражения. Такие матрицы никак не зависят от вычислительной платформы, на которую будет отображен ФБ, и, следовательно, не изменяются при различных отображениях модели на целевые узлы.

Может сложиться ситуация, когда один ФБ должен быть отображен на множество целевых узлов системы. Такая ситуация может быть вызвана двумя фактами:

- При понижении детализации рассмотрения модели, ФБ объединяются, образуя все более крупные структурные элементы модели. В конце концов, сложится ситуация, когда один ФБ включает в себя задачи, решаемые на разных узлах целевой системы (например, таким блоком может быть сама РИУС в целом). В этом случае нет смысла говорить об отображении ФБ на некоторый конкретный узел целевой системы;
- В модели системы существуют АФБ, отображенные на несколько узлов целевой системы. При этом эти блоки действительно являются атомарными, то есть не подразумевают дальнейшей [разумной] функциональной декомпозиции.

В отличие от первого случая, который на самом деле имеет место и очень важен при моделировании, второй случай является результатом ошибочного понимания узла целевой системы. АФБ по определению имеет законченную строго определенную функциональность и некоторую реализацию. Невозможно существование АФБ, отображенного на несколько целевых узлов системы по той причине, что помимо конечной функциональности такой блок должен обладать средствами синхронизации. Заблуждения о том, что АФБ, отображенные на множество целевых узлов, существуют, связаны с неадекватным восприятием структуры вычислительной платформы, а именно ее декомпозиции на целевые узлы.

В общем случае границей узла не являются границы корпуса микросхемы или платы в составе системы. Такое “примитивное” восприятие целевого узла верно только для очень простой по функциональности элементной базы А-платформы. В настоящее время элементная база настолько функционально развита и многообразна, что в рамках одного кристалла могут существовать несколько практически независимых целевых узлов. Каждый целевой узел характеризуется независимой системой тактирования, собственной виртуальной машиной, интерфейсами обмена событиями и средствами сохранения событий.

Пример неочевидной декомпозиции архитектурной платформы на целевые узлы приведен на рис. 1.11–1.12. Модель системы состоит из трех ФБ, два из которых (“Ф1” и “Ф2”) выполняют функции на удаленных друг от друга независимых вычислителях, а третий (“К”) обеспечивает передачу данных

между ними (рис. 1.11). Модель должна быть отображена на два независимых микроконтроллера, связанных последовательным каналом.

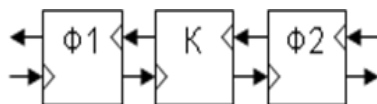


Рис. 1.11. Модель двух независимых вычислителей, связанных последовательным каналом

На первый взгляд не очень понятно, как АФБ “К” может быть отображен на предлагаемую топологию целевых узлов, так как он должен быть отображен и на uC1 и на uC2, что невозможно. Данное противоречие изображено на рис. 1.12.



Рис. 1.12. Невозможность отображения ФБ “К” на систему из двух вычислителей

Данное противоречие возникло из-за того, что границы целевых узлов очерчены некорректно. Функциональный блок “К” использует определенные ресурсы микроконтроллеров, и именно их внутреннюю периферию. Большинство периферийных интерфейсных расширителей являются самостоятельными вычислителями, пусть даже и не всегда поддающимися непосредственному программированию. Исходя из этого, необходимо детализировать предложенное представление микроконтроллеров, чтобы получить корректное разбиение вычислительной платформы на целевые узлы. Корректное представление целевых узлов и отображение на них модели представлено на рис. 1.13.

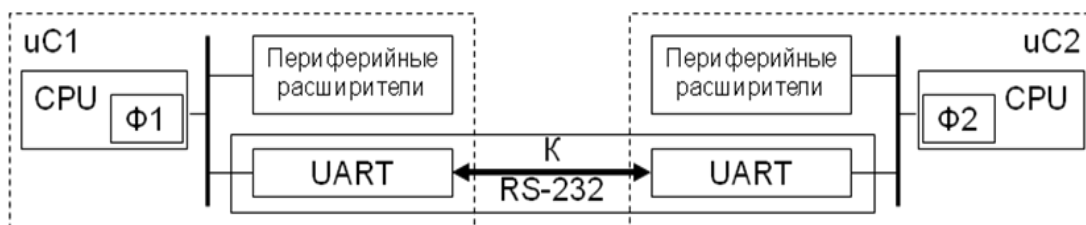


Рис. 1.13. Границы целевых узлов в системе двух независимых микроконтроллеров

Представленное на рис. 1.13 отображение модели показывает, что помимо традиционных вычислительных узлов (вычислительных ядер микроконтроллеров), рассматривается такой целевой узел вычислительной платформы, как асинхронный дуплексный канал связи. Хотя данный вычислительный узел не предусматривает программирования, однако имеет возможность конфигурирования скорости передачи данных, размера битовой посылки, контроля целостности и т.д. Аналогичными вычислительными узлами

могут являться и другие коммуникационные интерфейсы, причем не только точка-точка, но и с множественным доступом (CAN, Ethernet, RS-485 и др.).

В некоторых системах может оказаться полезным рассмотреть небольшие изменения поведения ФБ, который при этом по-прежнему выполняет свою целевую функцию. Если рассматривать абстрактный ФБ, то для его превращения в блок, предназначенный для работы целевой системы, необходимо выполнить два действия:

- “программировать” (определить алгоритм ФБ);
- “конфигурировать” (определить режим работы ФБ).

Первое действие непосредственно в целевой системе может быть не реализовано. Таким образом мы получим статическую ИУС, функциональность которой действительно жестко определена и не допускает перепрограммирования. Если предусмотреть возможность изменять алгоритм ФБ или даже количество самих ФБ, получим динамическую ИУС, которую легко можно будет превратить в систему управления различными объектами, динамически перестраивать функциональность при кардинальной смене условий функционирования, легко обновлять программное обеспечение системы, при обнаружении в нем ошибок и т.д.

Конфигурирование ФБ должно быть определено обязательно еще в архитектурной модели системы, так как изменение режимов алгоритма может повлиять на передаточные матрицы ФБ, что отразится на общих характеристиках системы в целом, т.е. будет влиять на другие аспекты проектирования.

После выбора конкретного “отображения” ФБ на целевые узлы перед разработчиками встает проблема реализации этих ФБ. Реализации подлежат только примитивные ФБ. На этом этапе можно выделить два типа ФБ:

- вычислительные ФБ;
- транспортные ФБ.

Вычислительный ФБ выполняет традиционные вычислительные функции обработки данных. Он может быть отображен на традиционный последовательный интерпретатор, микросхему программируемой логики или специализированный периферийный контроллер. Любой из этих элементов может служить целевым узлом отображения вычислительного ФБ. Производительностью таких узлов может стать скорость интерпретатора, объем доступных логических ячеек и т.д. Конфигурацией таких блоков может быть установка энергосберегающих режимов с деградацией скорости, объемы памяти и т.д.

Транспортный ФБ выполняет традиционные коммуникационные функции, реализуя сеть, связывающую целевые узлы. При этом сама сеть является одним из целевых узлов, на который ФБ и отражен. ФБ, отображаемые на целевой узел-сеть, представляют собой те или иные протоколы, которые целевая сеть

может поддерживать. Настройкой таких блоков может быть выбор битовой скорости канала связи, настройка форматов пакетов, контроля ошибок и т.д.

Событие

События в модели имеют сложную структуру. Каждое событие снабжено меткой времени и в качестве атрибута, при необходимости, имеет некоторые данные. Если рассмотреть предлагаемую модель как конкретизацию TSM [70], то получается, что событие e является элементом сигнала s

$$e \in s \subset T \times V,$$

где T – множество меток времени, V – значения данных. В общем случае данные события являются состоянием входа, по которому произошло событие. Типы данных, их объем, формат представления могут значительно варьироваться от модели к модели и даже внутри одной модели от одного ФБ к другому. В общем случае типы и объемы данных зависят от прикладной области и целевой задачи системы. В связи с этим можно выделить несколько возможных множеств данных V^i :

$$s^i \subset T \times V^i.$$

События распространяются по системе от внешней среды (объекта управления) или выходного порта ФБ к внешней среде (объекту управления) или входному порту другого ФБ. Между ФБ события распространяются мгновенно, без затрат времени. Наличие события активизирует определенный ФБ, который обрабатывает данные этого события и генерирует выходные события, опираясь на временную метку входного события и результаты обработки данных. Активизированный ФБ (занятый обработкой события) может различным образом реагировать на вновь поступающие события на синхронные порты. Поведение может варьироваться от полного временного игнорирования событий по своим синхронным портам, до независимой одновременной обработки поступающих событий. Конкретный тип поведения зависит от степени реализации параллельности для данного ФБ.

Согласно формуле (1.5) функциональность вычислителя определяется следующим выражением

$$\forall t \in T \quad f : I_t \rightarrow O_t,$$

при этом само действие функции f является композиций преобразования данных и преобразования меток времени

$$f(T \times V) = f_t(T) \times f_v(V), \quad (1.9)$$

$$f_t : T \rightarrow T, \quad f_v : V^i \rightarrow V^j. \quad (1.10), (1.11)$$

Предлагаемая декомпозиция функции предполагает независимость процедур обработки данных от значений этих самых данных. Время обработки события не зависит от значений атрибутов, а зависит только от самого события

$$\forall e \in s \subset T \times V \quad \|e - f(e)\|_T = \text{const} . \quad (1.12)$$

Данный критерий [см. формулу (1.12)] позволяет выделять управляющие входы и входы данных различных элементов для формирования ФБ ОСМВ.

Согласно формуле (1.9) поведение ФБ можно рассматривать как две независимых операции. Преобразование данных является частью общей функциональной декомпозиции. Функция преобразования времени является характеристикой самого ФБ и может зависеть от конкретной реализации. Данная функция определяет масштабы времени ФБ, время обработки входных сигналов и соотношение входных и выходных сигналов. Введение в модель функции f_t позволяет учесть временные аспекты моделирования, что очень важно для получения полной картины функционирования системы. Кроме того, за реализацию временных параметров отвечает та часть функциональности, которая преобразует временной атрибут события ОСМВ. В дальнейшем более подробно будет рассмотрена функция f_t , так как именно она определяет характеристики распространения событий в модели.

Порт

Как было сказано выше, функционирование системы происходит с помощью передачи событий между ФБ. Механизмом обмена событиями между ФБ называется порт. Порт воспринимает событие, сгенерированное одним из ФБ (для каждого порта генерировать события может только один ФБ), и передает его на входы других ФБ, которых в общем случае может быть несколько.

Порт предназначен для восприятия события от ФБ источника и передачи его нескольким ФБ потребителям. Порт должен передавать как само событие, так и его атрибуты. Порты, как и ФБ, должны быть отображены на целевые узлы системы. Так как порт соединяется с входами/выходами ФБ, то каждый из этих ФБ хотя бы частично должен быть “отображен” на тот же самый узел целевой системы. Если ФБ не в состоянии воспринять распространяемое портом событие или событие предназначается для асинхронного входа ФБ, то задачей порта становится буферизация события. Глубина буфера порта – одно событие. Если есть необходимость более “глубокой” буферизации событий, для этого в модель системы необходимо вводить специальные ФБ.

Для передачи события между ФБ порт при реализации требует определенных структурных ресурсов. Ресурсы необходимы для размещения данных события и его атрибутов. Объем ресурсов, требуемых для размещения события, зависит от типов элементов множеств T и V , которые составляют событие

$$e = \{t, v\} \in T \times V, t \in T, v \in V . \quad (1.13)$$

Важно рассмотреть различные способы реализации портов, описанных в ОСМВ. На рис. 1.14 представлена часть модели целевой системы,

демонстрирующая возможности порта Р при передаче события между функциональными блоками ФБ А и ФБ В.

Для восприятия события через входной порт ФБ В должен предоставить определенный интерфейс. Интерфейсом в данном случае является регламентированный набор действий, чтобы обеспечить получение события. При реализации необходимо “уведомить” ФБ-потребитель о возникновении события, которое необходимо обработать.

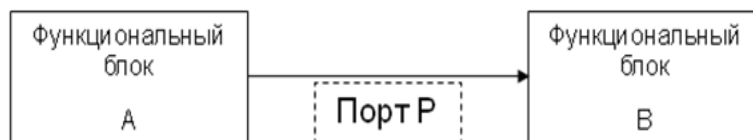


Рис. 1.14. Взаимодействие ФБ посредством порта

При реализации ФБ А и В могут “отображаться” на узлы программной (SW) и аппаратной (HW) природы. В обоих случаях факт передачи события реализуется различными традиционными механизмами. Примеры таких механизмов представлены в табл. 1.3.

Таблица 1.3. Реализация механизма передачи события

	Передача события {t,v} [см. формулу (1.13)]
SW→SW	Традиционный вызов процедуры, данные события передаются через стек или общую память, фактом вызова является выполнение инструкции перехода на подпрограмму.
SW→HW	Обращение (запись или чтение) специального управляющего регистра, данными события являются данные операции.
HW→SW	Запрос прерывания, работа канала ПДП. Данные либо не передаются (как в случае запроса прерывания), либо располагаются в фиксированных адресах памяти (как в случае ПДП).
HW→HW	Изменение состояния контакта или шины. Данными является состояние или последовательность (комбинация) состояний.

При реализации портов необходимо особое внимание уделить начальному состоянию буферов событий. Базовой МоС для ОСМВ можно рассматривать модель дискретных событий. Для работы такой модели необходимо реализовать функцию дискретизации сигнала, выражение (1.3)

$$\partial(s, t) = v \Leftrightarrow (t', v) \in s, t' \leq t \text{ и } \text{dist}(t, t') = \min.$$

Проблема в том, что в начальные моменты работы системы эта функция для некоторых сигналов вообще может не существовать. Т.е. может сложиться ситуация, когда в терминах функции дискретизации имеем

$$\neg \exists t' \leq t : (t', v) \in s.$$

В этом случае для ФБ не будут определены данные для проведения преобразования f_v , хотя преобразование f_t может быть осуществлено ввиду их взаимной независимости согласно формуле (1.9). Предлагается три способа решения данной проблемы.

- “*Безопасное состояние*”. Инициализация портов значениями “по умолчанию”. Данные значения сигналов должны соответствовать некоторым “безопасным” состояниям системы. Очевидно, что в некоторых случаях такое нарушение целостности данных внешних событий может привести к серьезным последствиям;
- “*Первоначальный опрос*”. Начальный опрос “производителей” событий. Данный механизм позволяет порту не просто быть пассивным приемником событий, но и по своей инициативе спрашивать у источника событий данные последнего события. В общем случае весь процесс распространения события распадается на 2 фазы: ФБ-источник уведомляет порт о возникновении события (формируется элемент множества T), порт обращается к источнику за данными (передача элемента множества V). При этом в работоспособной системе в случае буферизации источником данных последнего события вторая фаза обращения может производиться в произвольное время. В вычислительной технике существует огромное множество механизмов, реализующих именно такой двухфазный обмен. Недостатком подхода является необходимость дополнительных средств формирования фаз обмена и возможные сложности синхронизации пары $\{t, v\}$.
- “*Неинициализированные порты*”. ФБ не производит никаких действий, пока не будут получены все необходимые для этого события. Т.е. если среди портов, необходимых для формирования выходного события, присутствуют неопределенные, то действие не производится. Этот подход позволяет значительно сэкономить ресурсы целевых узлов, так как, в случае ошибок или других неполадок внешних устройств сигналы от них можно считать неинициализированными и действий не предпринимать. Недостатком подхода может стать то, что из-за одного неинициализированного сигнала ФБ может простаивать сколь угодно долго.

В каждом конкретном случае в зависимости от задачи или от инициатора событий разработчик имеет возможность выбрать один из перечисленных способов решить проблему. В некоторых случаях можно использовать комбинации способов, например, неинициализированные порты с первоначальным опросом.

Узел

Целевым узлом является определенный физический структурный элемент ИУС, который предоставляет свои вычислительные ресурсы для реализации “отображенных” на него ФБ и портов. Для различных реализаций одной и той же архитектуры перечень целевых узлов может значительно варьироваться. Основной временной характеристикой узла является минимальной фактическое

время следования входных событий, отображенных на него ФБ. Узлу необходимо некоторым образом разделять доступные ему ресурсы между всеми отображенными на него ФБ. Конкретная реализация параллельной работы отдельных ФБ в рамках одного узла влияет на фактические значения сложности генерации выходного события τ и минимальное время генерации выходных событий для всех ФБ.

1.3.3 Расчет временных характеристик объектно-событийной модели

При расчете характеристик ОСМВ РИУС используются временные параметры ФБ, а именно: сложность генерации выходного события τ [см. формулу (1.6)], минимальный интервал следования входных событий γ , и минимальный интервал генерации выходных событий ω . Эти три параметра дают полную временную характеристику ФБ $\{\tau, \gamma, \omega\}$.

Будем рассматривать ФБ, для которого I – набор входных сигналов ($s^1, s^2, \dots s^m$), а O – набор выходных сигналов ($s^{1+m}, s^{2+m}, \dots s^{n+m}$). Для такого блока имеем:

$$\tau = \begin{vmatrix} \tau_{1,1} & \dots & \tau_{1,m} \\ \vdots & \ddots & \vdots \\ \tau_{n,1} & \dots & \tau_{n,m} \end{vmatrix},$$

$$\gamma = \begin{vmatrix} \gamma_1 \\ \vdots \\ \gamma_m \end{vmatrix},$$

$$\omega = \begin{vmatrix} \omega_1 \\ \vdots \\ \omega_n \end{vmatrix}.$$

Важными требованиями к характеристикам ФБ являются

$$\forall j (0 < j \leq m) \sum_{i=1}^n \tau_{j,i} \neq 0, \quad (1.14)$$

т.е., все входные сигналы обрабатываются ФБ, пусть даже просто транслируются в выходной без изменений (нет пустых столбцов).

$$\forall j (0 < j \leq m) \sum_{i=1}^n \tau_{i,j} \neq 0,$$

все выходные сигналы генерируются ФБ (нет пустых строк).

При невыполнении одного из требований соответствующую строку или столбик необходимо исключить из матрицы τ и скорректировать столбцы γ и ω .

Можно рассчитать оптимальное поведение ФБ при благоприятных внешних условиях. Под благоприятными внешними условиями понимается максимальная доступная частота поступления внешних событий на все входы ФБ. Рассмотрим в таких условиях формирование ФБ некоторого выходного события s^i ($0 < i \leq n$).

Для формирования выходного события s^i необходимо обработать некоторое количество входных событий. Количество событий, необходимых для формирования выходного события s^i , определяется i -ой строкой матрицы τ . Ввиду благоприятных внешних условий все необходимые входные события поступают с максимально допустимой интенсивностью, заданной параметром γ . Таким образом, поток входных событий, необходимый для формирования s^i , будет следовать и обрабатываться не менее, чем $\tau_{i,j} \times \gamma_j$ для выходного события j .

В общем случае для формирования выходного события может потребоваться несколько различных входных событий. В этом случае время будет зависеть от степени параллельности выполнения этих событий ФБ. Интересны два крайних случая: полностью параллельное и полностью последовательное исполнение всех разнотипных входных событий. В первом случае функциональный блок называется *аппаратно реализованным*, во втором – *программно*. Таким образом получаем *критерий программно-аппаратной декомпозиции – степень параллельности во времени обработки событий ФБ*. Необходимо отметить, что можно применять различные известные технические решения, которые обеспечат псевдопараллельную обработку событий в программной реализации и конвейерно-последовательную обработку в аппаратной. Существование таких решений не отменяют сформулированный критерий, так как при его формулировке понятие “программной” и “аппаратной” реализации использовалось в традиционной трактовке [75].

Для ФБ последовательного исполнения входных событий минимальное время формирования выходного события составит

$$\omega_i = \sum_{j=1}^n \tau_{i,j} \gamma_j ,$$

что в матричном виде можно записать как

$$\omega = \tau \times \gamma . \quad (1.15)$$

Операцию “ $\times$ ” можно назвать *операцией последовательной интерпретации*. Применение такой операции к характеристикам ФБ при моделировании подразумевает “программную” реализацию данного ФБ. Можно заметить, что операция “ $\times$ ” является стандартной операцией умножения столбца на матрицу.

Для ФБ параллельного исполнения входных событий минимальное время формирования выходного события составит

$$\omega_i = \max_{j=1..n}(\tau_{i,j}\gamma_j), \quad (1.16)$$

что в матричном виде можно записать как

$$\omega = \tau \otimes \gamma, \quad (1.17)$$

где введенная операция “ $\otimes$ ” определяется выражением (1.16) и называется *операцией параллельной интерпретации*. Применение такой операции к характеристикам ФБ при моделировании подразумевает “аппаратную” реализацию данного ФБ.

В дальнейшем в тех местах выражений, где нет разницы между введенными операциями, будет использоваться символ “ $\bullet$ ”. Подстановка того или иного знака операции зависит от способа реализации ФБ.

Из представленных рассуждений и выражений (1.15) и (1.17) вытекает, что

$$\omega = \tau \bullet \gamma. \quad (1.18)$$

Видно, что формула (1.18) может быть использована для получения значений минимальных интервалов генерации выходных событий при известных значениях минимальных интервалов следования входных событий.

При анализе временных характеристик и матриц времен генерации выходных событий очень полезными являются операции сравнения. Интересует сравнение характеристик как сигналов, так и самих характеристических матриц ФБ. Для проведения такого рода сравнений определим некоторые операции.

Пусть ω и γ векторы-столбцы размером n элементов. Для таких столбцов определим операции сравнения $<, >, =, \leq, \geq$ следующим образом

$$\omega * \gamma \Leftrightarrow \forall i \in [1..n] \quad \omega_i * \gamma_i,$$

где $*$ обозначает любую из перечисленных операций сравнения. Аналогичным образом определим операции сравнения для матриц. Пусть τ^A и τ^B матрицы размером n столбцов m строк, ω вектор-столбец размером n . Тогда, по определению для некоторой реализации $\bullet$

$$\tau^A * \tau^B \Leftrightarrow \forall \omega \quad (\tau^A \bullet \omega) * (\tau^B \bullet \omega).$$

Можно доказать утверждение о том, что никакой ФБ не может уменьшить временной масштаб входных событий.

□ **Теорема 1.1.** “О не уменьшении временного масштаба ФБ”. Для выражения (1.18) можно доказать, что

$$\max_{i=1..n}(\omega_i) \geq \max_{i=1..m}(\gamma_i) \text{ и } \min_{i=1..n}(\omega_i) \geq \min_{i=1..m}(\gamma_i).$$

Аналогичным образом получаем

$$\min_{i=1..n}(\omega_i) = Mn = \omega_k = \bigodot_{i=1..m} \tau_{k,i} \gamma_i \geq \tau_{k,j} \gamma_j \geq \gamma_j \geq \min_{i=1..m}(\gamma_i).$$

Временной масштаб ФБ определяется его входными и выходными событиями. Если обозначить $M(\tau, \gamma, \omega)$ временной масштаб ФБ $\{\tau, \gamma, \omega\}$, то из теоремы 1.1 вытекает, что

$$M(\tau, \gamma, \omega) = \min(\min_{i=1..n}(\omega_i), \min_{i=1..m}(\gamma_i)) = \min_{i=1..m}(\gamma_i).$$

Временные характеристики, получаемые при аппаратной реализации ФБ не хуже, чем при программной реализации при всех прочих равных условиях. Данное утверждение доказывает следующая теорема.

□ **Теорема 1.2.** “О преимуществе параллельной интерпретации”.

$$\forall \gamma \quad \forall \tau \quad \tau \times \gamma \geq \tau \otimes \gamma.$$

.

■

К сожалению, обе операции (“ $\times$ ” и “ $\otimes$ ”) описывают некий идеальный случай реализации ФБ. Однако эти две реализации занимают граничные точки спектра, и все остальные реализации будут находиться между ними. Т.е. степень параллельности интерпретации входных событий при программно-аппаратной реализации функционального блока находится “между” операциями “ $\times$ ” и “ $\otimes$ ”, что позволяет получать граничные оценки реализации.

Для формирования выходных событий необходима сложная взаимосвязь входных. Чтобы посчитать скорость формирования ФБ реакции, нужно задаться вопросом: сколько выходных событий N может сформировать ФБ $\{\tau, \gamma, \omega\}$ за время t_0 , при наличии k_i входных событий за это же время?

Каждый из входных сигналов обеспечивает формирование некоторого количества интересующих выходных событий, а именно j -ый входной сигнал обеспечит формирование $\frac{k_j}{\tau_{i,j}}$ выходных i -ых событий. В этом случае учитывая ограничение на количество событий сверху, задаваемое временным масштабом выхода, получаем

$$N_i = \min\left(\frac{t_0}{\omega_i}, \min_{j=1..m}\left(\frac{k_j}{\tau_{i,j}}\right)\right). \quad (1.19)$$

Время формирования одного события равно $\Delta_i = \frac{t_0}{N_i}$.

Можно попробовать переписать выражение для Δ_i , используя выражение (1.).

$$\Delta_i = \frac{t_0}{N_i} = \frac{t_0}{\min(\frac{t_0}{\omega_i}, \min_{j=1..m}(\frac{k_j}{\tau_{i,j}}))} = \frac{1}{\min_{j=1..m}(\frac{1}{\omega_i}, \frac{k_j/t_0}{\tau_{i,j}})} =$$

$$= \max_{j=1..m}(\frac{1}{1/\omega_i}, \frac{\tau_{i,j}}{k_j/t_0}) = \max(\omega_i, \max_{j=1..m}(\tau_{i,j} \frac{t_0}{k_j})) = \max(\omega_i, t_0 \max_{j=1..m}(\tau_{i,j} \frac{1}{k_j}))$$

Заметим, что

$$t_0 \max_{j=1..m}(\tau_{i,j} \frac{1}{k_j}) = (t_0 \cdot \tau \otimes \begin{pmatrix} \frac{1}{k_1} \\ \frac{1}{k_2} \\ \vdots \\ \frac{1}{k_m} \end{pmatrix})_i. \quad (1.190)$$

Отсюда, если наряду со столбцом k определить столбец k^{-1} следующим образом

$$k_i^{-1} = \frac{1}{k_i},$$

то время формирования событий функциональным блоком можно рассчитать по формуле

$$\Delta = \max(\omega, t_0 \cdot (\tau \otimes k^{-1})) = \max(\tau \bullet \gamma, t_0 \cdot (\tau \otimes k^{-1})), \quad (1.201)$$

где символ “ $\bullet$ ” заменяет операцию реализации ФБ. Выражение (1.202) является основным для расчета временной передаточной матрицы.

При эволюции модели в результате проектирования понижается или повышается уровень абстракции рассмотрения системы. Создаются ФБ, представляющие собой композицию более мелких ФБ. В конечном итоге все ФБ должны стать атомарными. При составлении сложных ФБ из более мелких блоков параметры $\{\tau, \gamma, \omega\}$ таких блоков можно рассчитать, используя информацию о составляющих их блоках.

Рассмотрим декомпозицию ФБ С на два более мелких А и В (см. рис. 1.15). Пусть изображенные ФБ С, А и В определяются параметрами $\{\tau^C, \gamma, \omega\}$, $\{\tau^A, \gamma, \varepsilon\}$ и $\{\tau^B, \varepsilon, \omega\}$ соответственно. Необходимо получить аналитический способ расчета параметров ФБ С по известным параметрам ФБ А и В. Наряду с параметрами γ и ω , основной интерес представляет вычисление τ^C , так как именно эта матрица играет основную роль при расчете временных характеристик ФБ, в частности и модели в целом. В случае, когда ФБ состоит из большого количества малых, получение аналитического вида матрицы τ^C качественно упрощает процесс моделирования и верификации.

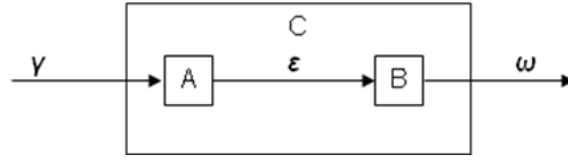


Рис. 1.15. Композиция ФБ

Согласно формуле (1.18) имеем

$$\omega = \tau^B \bullet \varepsilon = \tau^B \bullet (\tau^A \bullet \gamma). \quad (1.213)$$

Опять же согласно формуле (1.18) τ^C определяется как $\omega = \tau^C \bullet \gamma$.

Для начала рассмотрим случай, когда оба внутренних блоки имеют одинаковую реализацию (программную или аппаратную).

В случае программной реализации выражение (1.190) принимает вид:

$$\omega = \tau^B \times (\tau^A \times \gamma) = \tau^B \times \tau^A \times \gamma = (\tau^B \times \tau^A) \times \gamma = \tau^C \times \gamma. \quad (1.224)$$

Прделанные преобразования возможны, так как операция “ $\times$ ” представляет не что иное, как матричное умножение. Получаем, что τ^C определяется следующим образом:

$$\begin{aligned} \tau^C &= \tau^B \times \tau^A \\ \tau_{i,j}^C &= \sum_k \tau_{i,k}^B \tau_{k,j}^A. \\ \omega &= \tau^C \times \gamma \end{aligned}$$

В случае аппаратной реализации выражение (1.214) принимает вид:

$$\omega_i = (\tau^B \otimes (\tau^A \otimes \gamma))_i = \max_k (\tau_{i,k}^B \cdot (\tau^A \otimes \gamma)_k) = \max_k (\tau_{i,k}^B \cdot \max_j (\tau_{k,j}^A \gamma_j)).$$

Видно, что $\tau_{i,k}^B$ никак не зависит от j , поэтому переменную можно внести под функцию максимума

$$\omega_i = \max_k (\max_j (\tau_{i,k}^B \tau_{k,j}^A \gamma_j)).$$

Порядок поиска максимумов в выражении (1.224) не важен, поэтому справедливо

$$\omega_i = \max_j (\max_k (\tau_{i,k}^B \tau_{k,j}^A \gamma_j)).$$

Видно, что переменная γ_j не зависит от k и может быть вынесена из-под максимума:

$$\omega_i = \max_j (\max_k (\tau_{i,k}^B \tau_{k,j}^A) \gamma_j). \quad (1.25)$$

Если сравнить выражение (1.5) и определение операции параллельной интерпретации (1.16) и (1.17), то можно выделить некую матрицу:

$$\omega_i = \max_j (\tau'_{i,j} \gamma_j)$$

$$\tau'_{i,j} = \max_k (\tau_{i,k}^B \tau_{k,j}^A).$$

В этом случае, аналогично операции последовательной интерпретации, можно определить τ^C следующим образом (определив операцию параллельного умножения матриц “ $\otimes$ ”):

$$\tau^C = \tau^B \otimes \tau^A$$

$$\tau_{i,j}^C = \max_k (\tau_{i,k}^B \tau_{k,j}^A).$$

$$\omega = \tau^C \otimes \gamma$$

В случае разноименных операций $\bullet$ в выражении (1.213), для общего вида матрицы τ^C можно получить только оценки сверху и снизу, так как конкретный результат зависит от значений временных масштабов входных событий. Перед выводом оценок можно доказать вспомогательную теорему.

□ **Теорема 1.3.**

$$\forall i \in [1...n] \forall j \in [1...m] \quad a_{i,j} \geq 0,$$

тогда

$$\max_{i=1..n} \sum_{j=1}^m a_{i,j} \leq \sum_{j=1}^m \max_{i=1..n} a_{i,j}. \quad (1.236)$$

$$\max_{i=1..n} \sum_{j=1}^m a_{i,j} = \max_{i=1..n} \sum_{j=1}^m (M_j + \xi_{i,j}) = \max_{i=1..n} (\sum_{j=1}^m M_j + \sum_{j=1}^m \xi_{i,j}) = \sum_{j=1}^m M_j + \max_{i=1..n} \sum_{j=1}^m \xi_{i,j}. \quad (1.27)$$

Можно также доказать теорему об оценке матриц ФБ смешанных реализаций для выражения (1.213).

□ **Теорема 1.4.** “Об оценке смешанных реализаций ФБ”.

Для любых τ^A , τ^B и γ верны следующие верхние и нижние оценки смешанных реализаций:

$$\tau^C \times \gamma \geq \tau^A \otimes (\tau^B \times \gamma) \geq \tau^C \otimes \gamma, \text{ где } \tau^C = (\tau^A \otimes \tau^B), \quad (1.28)$$

$$\tau^C \times \gamma \geq \tau^A \times (\tau^B \otimes \gamma) \geq \tau^C \otimes \gamma, \text{ где } \tau^C = (\tau^A \times \tau^B). \quad (1.29)$$

Необходимо обратить внимание на то, что вариантов объединения “мелких” матриц значительно больше, чем изображено на рис. 1.15. Например, для представленной на рис. 1.16 топологии необходимо применить следующую технику получения матрицы τ^C .

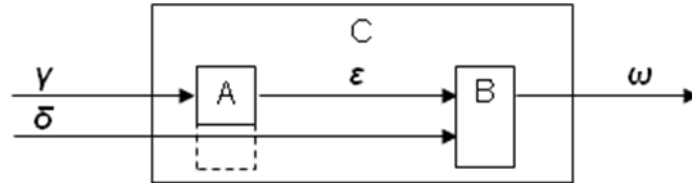


Рис. 1.16. Композиция ФБ

Необходимо “расширить” ФБ А, чтобы он просто ретранслировал сигнал δ на выход. На рис. 1.16 границы “расширенного” ФБ А обозначены пунктирной линией. Техника такого преобразования следующая.

Рассмотрим исходные правила преобразования сигналов:

$$\omega = \tau^B \bullet \begin{vmatrix} \varepsilon \\ \delta \end{vmatrix} = \tau^B \bullet \begin{vmatrix} \tau^A \bullet \gamma \\ \delta \end{vmatrix}.$$

Преобразуем матрицу τ^A так, чтобы на входе она получала расширенный столбец $\begin{vmatrix} \gamma \\ \delta \end{vmatrix}$ и после его преобразования не изменяла нижнюю часть столбца. Делается это следующим образом:

$$\begin{vmatrix} \varepsilon \\ \delta \end{vmatrix} = \tilde{\tau}^A \bullet \begin{vmatrix} \gamma \\ \delta \end{vmatrix}$$

$$\tilde{\tau}^A = \left[\begin{array}{c|cccc} \tau^A & 0 & 0 & \dots & 0 \\ \hline 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{array} \right].$$

Очевидно, что при любой операции $\bullet$, δ -часть столбца останется неизменной. В таком случае конечную матрицу τ^C следует вычислять следующим образом:

$$\tau^C = \tau^B \bullet \tilde{\tau}^A$$

$$\omega = \tau^C \bullet \begin{vmatrix} \gamma \\ \delta \end{vmatrix}.$$

Подводя итог, можно сказать, что предложенная ОСМВ учитывает основные особенности РИУС, перечисленные в [9, 10, 25], а разработанный математический аппарат может быть положен в основу инструментальных систем автоматизации проектирования и моделирования распределенных ВсС на базе ОСМВ.

При реализации той или иной МоС целевой системы коммуникационная среда (КМС) должна адекватно отражать протекающие в системе процессы и удовлетворять требованиям, предъявляемым моделью к стилю взаимодействия

объектов. ОСМВ РИУС легла в основу КМС проектов КТЖ-2 и СПКОПиУ-01Ф [20]. Будучи на первый взгляд совершенно разными, обе системы являются яркими представителями РИУС и хорошо описываются предложенной ОСМВ.

К основной КМС РИУС, т.е. КМС, действующей в рамках одной модели, можно выдвинуть следующие требования:

- Обеспечение контроля целостности КМС;
- Идентификацию, отключение и подключение узлов без потери общей работоспособности;
- Распространение события в пределах всей КМС, поддержка различных режимов адресации;
- Распространение синхронных событий;
- Распространение асинхронных событий;
- Передача всех параметров событий (данных и меток времени).

В различных задачах на первое место могут выходить одни требования, а другие вообще могут не иметь никакого значения. При реализации конкретной КМС лучше все эти требования иметь в виду, чтобы получить легко адаптируемую систему, в случае необходимости ее повторного использования.

В принципе, КМС должна поддерживать различные топологии информационных каналов связи и скрывать эту информацию от прикладной задачи (в роли прикладной задачи тут выступают объекты, выполняющие целевую задачу). То же самое можно сказать и о сетевых узлах. В случае сокрытия целевой топологии от прикладной задачи, для последней не имеет значения сколько именно сетевых узлов и какого они типа установлены в системе. Для прикладной задачи важен обмен данными с конкретными элементами ввода/вывода, а не способы обеспечения этого обмена. Для того чтобы скрыть от целевой задачи как топологию коммуникационных интерфейсов, так и состав узлов системы, КМС должна решать этот вопрос самостоятельно. В идеальном случае все события в системе, порожденные некоторым объектом, должны распространяться КМС только для соответствующих синхронных портов других объектов. С другой стороны, в случае динамического алгоритма управления, роль, перечень и расположение портов могут меняться в процессе функционирования системы.

При рассмотрении ОСМВ РИУС можно заметить, что топология связи портов (а именно между портами КМС должна передавать события) в общем случае кардинально отличается от топологии целевых узлов в физической сети. Модель системы, представленная на рис. 1.17, демонстрирует вышесказанное.



Рис. 1.17. Соотношение топологий коммуникационной среды и целевых узлов

Объекты, представленные в модели, отображаются на целевые сетевые узлы А, Б и В, которые имеют свои собственные адреса в рамках КМС. При другом “отображении” модели на целевые узлы, например, узел Б будет иметь другой адрес или изменится тип узла В и текущие, отображенные на него объекты, будут отображены на независимые узлы, адресация/топология объектов изменится и, если эта информация не будет скрыта от объектов инициаторов событий, то реконфигурации нужно будет подвергнуть всю систему целиком. В некоторых случаях такой метод приемлем, и для экономии ресурсов задачи маршрутизации можно перенести на канальный уровень и предоставить целевым процессам возможность непосредственно адресовать необходимые им порты для передачи событий.

Для построения относительно сложных РИУС, особенно с возможностями пользовательского прикладного программирования такой способ видится неперспективным, ввиду размытости задачи сетевого уровня КМС и ее значительного проникновения в соседние уровни.

В качестве альтернативного подхода при реализации КМС РИУС предлагается явным образом выделять функциональный уровень аналогичный сетевому уровню модели OSI/ISO [54, 55]. Выделение и реализация данного уровня позволит провести сквозную идентификацию портов системы в пределах модели и адресацию событий в терминах “прикладных” имен портов.

На первый взгляд может показаться, что выделение такого функционального уровня приводит к значительным затратам ресурсов, что особенно заметно для небольших неинтеллектуальных контроллеров периферийного ввода/вывода и тем самым является практически неприменимым. Несомненно, реализация дополнительной функциональности должна требовать дополнительных ресурсов. Однако объем этих дополнительных ресурсов зависит от вариантов реализации.

Во-первых, реализовывать такой функциональный уровень можно не для всех устройств системы, а только для нескольких центральных узлов, отвечающих за маршрутизацию событий. На самом деле сложно ожидать от слабого периферийного узла эффективной маршрутизации событий. От периферийных контроллеров в этом случае может потребоваться только способность идентифицировать свой тип.

Во-вторых, этот функциональный уровень можно реализовать динамически или статически. При динамической реализации требуется больше ресурсов, но система продолжает функционировать в различных конфигурациях сетевых узлов, при статической реализации система обладает способностью

определить, совпадает ли текущая конфигурация с заданной при проектировании системы. Оба варианта реализации значительно повышают надежность системы в целом.

С точки зрения предлагаемой ОСМВ каждое событие должно быть направлено на определенный порт. Порт располагается на некотором узле, на который был отображен объект, обладающий портом. Отсюда следует, что для адресации некоторого порта необходимо иметь пару значений {NODE,PORT}, где NODE адресует узел, на который отображен объект, а PORT идентифицирует собственно порт в рамках узла. При отсутствии средств контроля топологии КМС становится опасно пересылать сообщение просто на адрес {NODE,PORT}. Потенциально возможны следующие ситуации, которые приведут к нарушению работы системы в целом:

- Узел с адресом NODE может отсутствовать в системе;
- Узел с адресом NODE может быть другого типа, чем предполагает отправитель сообщения;
- Порт PORT может отсутствовать в узле NODE;
- Порт PORT может принадлежать другому объекту, способы идентификации объектов в рамках узла и портов в рамках того же узла никак не связаны друг с другом.

Именно для того, чтобы избежать перечисленных недоразумений, необходимо обеспечить целостность КМС. Целостность можно обеспечить, реализовав различные уровни идентификации объектов ОСМВ, а именно:

- Идентификация типов узлов в рамках системы. Это позволит определять физическую топологию и состав системы, работать с узлами как с устройствами (определение соответствия типа адресуемого устройства заданному, изменение идентификаторов, сетевых адресов, уникальных настроек, обновление программного обеспечения и т.д.). В рамках целевой задачи такая информация не нужна, поэтому выполнить перечисленные выше действия зачастую бывает невозможно.
- Системная идентификация объектов в рамках узла. Идентификация необходима, чтобы определять класс объекта, режимы работы объекта, идентифицировать объект и т.д.
- Идентификация отдельных портов в рамках объектов одного класса. В принципе, это не обязательное требование, однако желательно его придерживаться с целью сохранения независимости от реализации отдельных объектов.
- Прикладная идентификация объектов узла в рамках всей системы. Данный механизм позволяет скрыть особенности физической реализации коммуникационной подсистемы.

Как ни странно, все эти способы идентификации реализуются каждый раз при создании КМС, но сами реализации могут кардинально отличаться. В основном отличие состоит в том динамически или статически происходит каждый уровень идентификации и насколько этот процесс в конечном итоге автоматизирован.

В табл. 1.4 приведены сравнительные характеристики КМС проектов КТЖ-2 и СПКОПиУ-01Ф.

Таблица 1.4. Сравнительные характеристики КМС проектов КТЖ-2 и СПКОПиУ-01Ф

Система	КТЖ-2	СПКОПиУ-01Ф
Канальный уровень	CAN 2.0B	CAN 2.0B
Скорость передачи данных	50 кб/с	50 кб/с
Топология информационного канала	моноканал	кольцо (цепочка соединений точка-точка)
Подчиненность в рамках системы	Мастер сети + пассивные устройства ввода/вывода	Одноранговая сеть
Степень “программируемости” пользователем узлов сети	2 (Настройка параметров функционирования узла)	4 (Создание собственных алгоритмов функционирования узла)
Определение физической топологии	статическое (пользовательское)	динамическое
Отображение прикладного идентификатора	статическое (пользовательское)	динамическое

Для решения инструментальных задач при разработке РИУС КМС должна обеспечивать передачу больших объемов данных, представляющих собой собранную статистику, обновления ПО и т.д. Основная сложность, которую необходимо преодолеть, это реализация модели потоков данных в КМС, которая в целевом режиме базируется на ОСМВ.

В пособии рассматривается способ инкапсуляции инструментальной КМС в целевую и оценивается эффективность реализации предложенного совмещения МоС.

Задача совмещения моделей вычислений будет решаться при следующих условиях:

- необходимо передавать значительные объемы данных, на порядки превосходящие объемы данных единичной транзакции целевой КМС;
- передача потока данных должна осуществляться посредством событий;
- необходимо учесть доставку данных нескольким потребителям одновременно;
- необходимо учесть надежность КМС при выполнении транзакции.

Весь поток данных должен быть передан и получено подтверждение о его корректности. Рассматривая общий случай, будем разбивать весь поток на фрагменты и требовать подтверждения приема фрагментов.

Таким образом имеются 3 параметра задачи: n -величина фрагментации исходного потока, k -количество абонентов КМС, p -вероятность успешного прохождения транзакции до назначения (надежность КМС).

Эффективностью реализации совмещения МоС будем называть отношение $\left(\frac{V(n,k,p)}{n}\right)^{-1}$, где $V(n,k,p)$ – среднее количество транзакции для передачи фрагмента потока размера n . Нахождение оптимального решения заключается в нахождении максимума выражения (1.24) как функции от n при заданных параметрах k и p .

$$\nu(n,k,p) = \frac{n}{V(n,k,p)} \quad (1.24)$$

Очевидно, что для поэлементного подтверждения потока данных $n=1$, для передачи точка-точка $k=1$. В связи с этим интересными характеристиками оказываются отношения (1.25) - эффективность фрагментации и (1.26) - эффективность широковещания при варьировании надежности коммуникационной среды.

$$\varphi(n,k,p) = \frac{n \cdot V(1,k,p)}{V(n,k,p)} \quad (1.25)$$

$$\beta(n,k,p) = \frac{k \cdot V(n,1,p)}{V(n,k,p)} \quad (1.26)$$

Указанные выражения также могут быть использованы в качестве частных критериев при оценке эффективности решения задачи.

Для получения $\nu(n,k,p)$ необходимо построить $V(n,k,p)$. Процесс передачи фрагмента происходит в следующем порядке: пересылаются n транзакций данных, после чего ожидается k подтверждений полученных транзакций. Передача возобновляется с самой ранней некорректно принятой (неподтвержденной) транзакции.

Для дальнейших расчетов необходимо вычислить вероятность неуспешного прохождения подряд N транзакций при заданной надежности коммуникационной среды $P_{fail}(N, p)$. Легко показать, что

$$P_{fail}(N, p) = 1 - P_{success}(N, p) = 1 - p^N$$

Как было сказано выше, общий объем транзакций для передачи фрагмента равен $n+k$. Данное количество транзакций можно разбить на два блока $n-1$ транзакция в прямом направлении (“голова”, head) и $1+k$ транзакция в обратном направлении, кроме первой (“хвост”, tail). Общий объем транзакций для фрагмента n в этом случае рассчитывается по формуле (1.27):

$$V(n, k, p) = HEAD(n-1, p) + TAIL(1+k, p) \quad (1.27)$$

Для $TAIL()$ принципиальным является корректная передача всех транзакций. Рассчитаем средний объем транзакций для $TAIL(N, p)$ при заданной надежности КМС.

$$\begin{aligned} TAIL(N, p) &= \\ &= N \cdot P_{success}(N, p) + 2 \cdot N \cdot P_{fail}(N, p) \cdot P_{success}(N, p) + 3 \cdot N \cdot P_{fail}(N, p)^2 \cdot P_{success}(N, p) + \dots = \\ &= \frac{N \cdot P_{success}(N, p)}{P_{fail}(N, p)} \cdot \sum_{i=1}^{\infty} i \cdot P_{fail}(N, p)^i \end{aligned} \quad (1.28)$$

Из теории рядов известно, что:

$$\sum_{i=1}^{\infty} i \alpha^i = \frac{\alpha}{(1-\alpha)^2} \quad (1.29)$$

Используя (1.29) перепишем (1.28):

$$TAIL(N, p) = \frac{N \cdot P_{success}(N, p)}{P_{fail}(N, p)} \cdot \frac{P_{fail}(N, p)}{(1 - P_{fail}(N, p))^2} = \frac{N}{P_{success}(N, p)} = N \cdot p^{-N} \quad (1.30)$$

По формуле полной вероятности распишем $HEAD(N, p)$:

$$\begin{aligned} HEAD(N, p) &= N + (1-p) \cdot HEAD(N, p) + p \cdot (1-p) \cdot HEAD(N-1, p) + \dots \\ &\dots + p^{N-1} \cdot (1-p) \cdot HEAD(1, p) \end{aligned}$$

$$p \cdot HEAD(N, p) = N + p \cdot (1-p) \cdot HEAD(N-1, p) + \dots + p^{N-1} \cdot (1-p) \cdot HEAD(1, p) \quad (1.31)$$

$$p \cdot HEAD(N, p) = N + (1-p) \cdot \sum_{i=1}^{N-1} p^{N-i} \cdot HEAD(i, p) = N + (1-p) \cdot p^N \cdot \sum_{i=1}^{N-1} \frac{HEAD(i, p)}{p^i}$$

Для выражения $HEAD(N, p)$, используя (1.31), представим рекуррентную формулу (1.32)

$$\begin{aligned} HEAD(1, p) &= TAIL(1, p) = p^{-1} \\ HEAD(N, p) &= \frac{N}{p} + (1-p) \cdot p^{N-1} \cdot \sum_{i=1}^{N-1} \frac{HEAD(i, p)}{p^i} \end{aligned} \quad (1.32)$$

Имея выражение (1.30) для среднего количества транзакций $TAIL$ и (1.32) для $HEAD$, используя (1.27), можно рассчитать эффективность реализации совмещения МоС (1.24):

$$\nu(n, k, p) = \frac{n}{HEAD(n-1, p) + TAIL(1+k, p)},$$

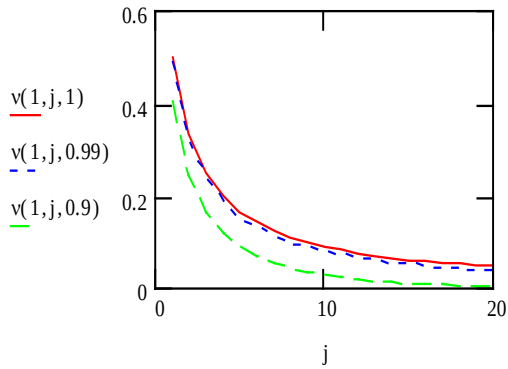


Рис. 1.18

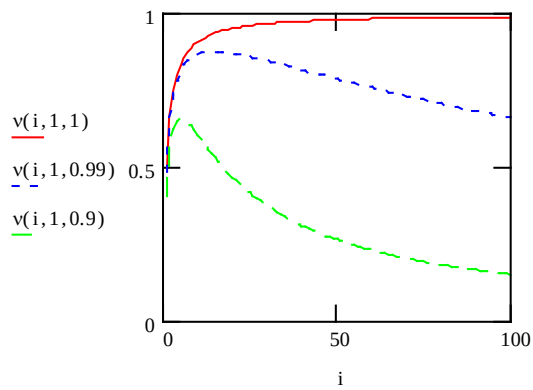


Рис. 1.19

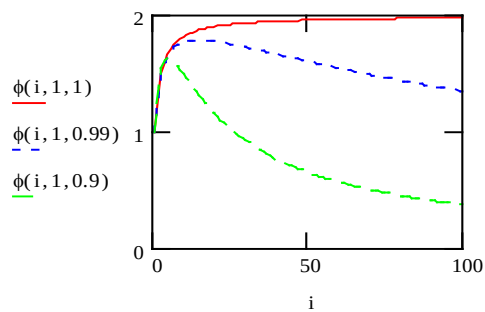


Рис. 1.20

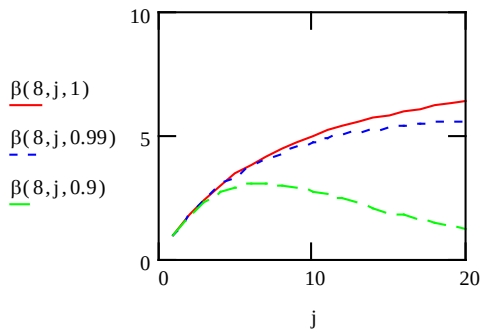


Рис. 1.21

Рис. 1.18 – эффективность совмещения МоС как функция количества абонентов КМС, рис. 1.19 – эффективность совмещения МоС как функция размера фрагментации потока данных, рис. 1.20 – эффективность фрагментации (1.25), рис. 1.21 – эффективность широковещания (1.26).

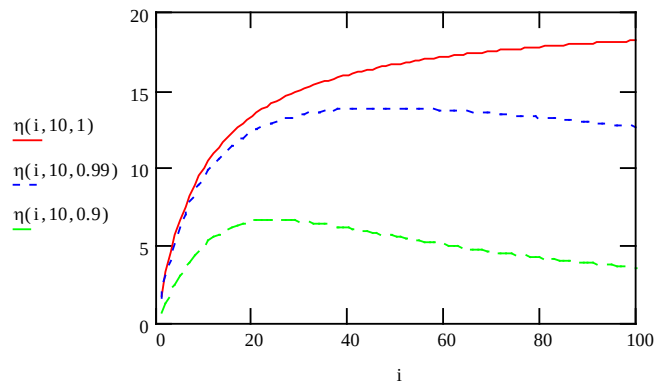


Рис. 1.22. Сравнение количества транзакций при широковещательной доставке обновлений ПО

Самостоятельный интерес представляет сравнение количества транзакций при широковещательной доставке обновлений ПО (1.33). Можно построить график (рис. 1.22) отношения индивидуальной поочередной конфигурации 10 контроллеров в модели потоков данных и широковещательной конфигурации для различной фрагментации образа обновлений ПО средствами ОСМВ.

$$\eta(n, k, p) = \frac{k \cdot n \cdot V(1, 1, p)}{V(n, k, p)} = \frac{k \cdot n \cdot v(1, 1, p)^{-1}}{n \cdot v(n, k, p)^{-1}} = \frac{k \cdot v(n, k, p)}{v(1, 1, p)} \quad (1.33)$$

1.3.4 Расширение ОСМВ на основе денотативного описания

Операционная семантика ОСМВ обладает очевидными достоинствами, однако она затрудняет сравнение этой модели с другими МоС и применение ее при создании гетерогенных моделей. Для применения ОСМВ в комплексном моделировании разработано ее расширение на основе денотативного описания, которое использует метамодель «Модель сигналов с тэгами», Tagged Signal Model, (TSM). Термин «денотативная» означает, что модель формулирует вычисления в стиле теории доменов Скотта, описывавшей вычисления в терминах свойств функциональных преобразований сигналов (таким образом,

обозначавшей эти свойства), в отличие от операционного подхода (Тьюринг, Марков, Чёрч).

Далее приводится денотативное описание объектно-событийных моделей ВсС и способы вычисления временных характеристик объектно-событийных моделей встроенных систем на основе денотативного описания [17, 18].

1.3.4.1 Денотативное описание объектно-событийных моделей ВсС

Как было показано в параграфе 1.1.4.5, модель TSM основана на представлении вычислительной системы как совокупности *сигналов*. Сигнал, в свою очередь, является множеством *событий* вида $e = \{t, v\}$, где v есть некоторое значение, элемент данных, а t есть ассоциированная с ним некоторая метка (*тэг*). Событие $e = \{t, \varepsilon\}$ называется «пустым». В зависимости от свойств множества меток T , понятие сигнала может иметь различный смысл. Компонент вычислительного процесса (*актор*) представляет собой множество его поведений. На основе теоретико-множественного подхода строится математическое описание соединений между акторами, входных и выходных сигналов, композиций акторов произвольной топологии, включая обратную связь. Модель описывает также процедуру абстракции системы процессов в один процесс и затрагивает вопросы эквивалентных структурных преобразований системы акторов. TSM формально задает понятия времени, его метрик, детерминизма, каузальности, монотонности и непрерывности акторов, другие понятия, используемые при анализе ВсС.

В рамках денотативного описания ОСМВ (ДОМВ) в качестве модели времени выбрано множество неотрицательных действительных чисел $\mathbf{R}_+$, то есть для ОСМВ множество тэгов $T = \mathbf{R}_+$. Из приведенного выше определения сигнала следует, что функциональный сигнал есть функция $s: T \rightarrow V$, приписывающая каждой метке некоторое значение. Область определения этой функции $\text{dom}(s)$ есть начальный сегмент T (сигнал может быть известен лишь до некоторого момента). Если $\text{dom}(s) = T$, то сигнал называется *целым* (он до конца известен).

Между сигналами можно задать отношение префикса, являющееся обобщением префиксов строк и последовательностей. Для любых двух сигналов s_1 и s_2 , s_1 есть *префикс* s_2 (обозначается $s_1 \sqsubseteq s_2$), если $\text{dom}(s_1) \subseteq \text{dom}(s_2)$ и для всех $t \in \text{dom}(s_1)$: $s_1(t) = s_2(t)$. Иными словами, выражение $s_1 \sqsubseteq s_2$ означает, что график функции s_1 является частью графика s_2 до некоторого t .

Для описания сигналов в объектно-событийных системах была задана функция $\delta: S \rightarrow T$, характеризующая минимальную разность между метками непустых событий в сигнале. Обозначим $D_s = \{|t - t'|: t, t' \in \text{dom}(s), s(t) \neq \varepsilon, s(t') \neq \varepsilon, t \neq t'\}$ множество всех интервалов времени между непустыми событиями в сигнале s . Тогда

$$\delta(s) = \begin{cases} \infty, & \text{если } D_s = \emptyset, \\ \inf(D_s), & \text{иначе,} \end{cases}$$

где $\inf(D_s)$ есть точная нижняя грань множества D_s . Для сигналов, содержащих меньше двух непустых событий, значение функции равно бесконечности. Если сигнал содержит больше одного непустого события, то $\delta(s)$ дает минимальный интервал между ними.

Функция $\delta(s)$ позволяет задать отличительную характеристику сигналов на синхронных входах и выходах ФБ. Для таких сигналов

$$0 < \tau_s \leq \delta(s), \quad (1.34)$$

где τ_s есть атрибут порта (γ_j для синхронного входа, ω_i для выхода). Этот атрибут характеризует проектное ограничение на минимальный интервал времени между интерпретируемыми на входе или генерируемыми на выходе ФБ инструкциями (возможности платформы, требования ТЗ) и является характеристикой ФБ.

В соответствии с вышесказанным, сигналы в объектно-событийной системе, за исключением подаваемых на ее внешние асинхронные входы, обладают свойством (1.34) и являются дискретно-событийными (ДС). Стоит отметить, что свойство (1.34) является более строгим, чем определение ДС сигнала, и не все ДС сигналы обладают этим свойством.

Асинхронный вход ФБ преобразует подаваемый на него сигнал таким образом, что ФБ в каждый момент времени может прочитать значение последнего непустого события в поданном сигнале. Множество непустых событий на асинхронном входе будет $E = \{(t, s(t'))\}$, где $t, t' \in \text{dom}(s)$, $t \leq t'$ и не существует $t'' \in \text{dom}(s)$ такого, что $t' < t'' \leq t$ и $s(t'') \neq \varepsilon$. Очевидно, множество E непрерывно и минимальный интервал между событиями на асинхронном входе равен 0.

Поведение ФБ есть функция $\sigma: P \rightarrow S$, приписывающая каждому порту (входу или выходу) ФБ наблюдаемый на нем сигнал (P есть множество портов ФБ, S множество сигналов в системе). Сам функциональный блок есть множество поведений.

Для любого поведения ФБ σ минимальный интервал между событиями на синхронном входе p_j будет равен, с учетом (1.34), $\tilde{\gamma}_j = \delta(\sigma(p_j))$. Минимальный интервал между генерируемыми на выходе p_i событиями будет $\tilde{\omega}_i = \delta(\sigma(p_i))$. Эти значения, в отличие от γ_j и ω_i , являются характеристиками сигналов, а не самого ФБ. Векторы $\tilde{\Gamma} = \|\tilde{\gamma}_j\|$ и $\tilde{\Omega} = \|\tilde{\omega}_i\|$ характеризуют совокупность сигналов, связанных с ФБ в конкретной системе, в отличие от векторов $\Gamma = \|\gamma_j\|$ и $\Omega = \|\omega_i\|$, представляющих собой проектные ограничения временных характеристик ФБ.

Математический аппарат ОСМВ позволяет вычислить вектор $\tilde{\Omega}$ по заданному вектору $\tilde{\Gamma}$.

Обозначим $f \uparrow D$ сужение области определения функции f до ее подмножества $D \subseteq \text{dom}(s)$. Тогда для ФБ с множеством входных портов P_{in} и выходных P_o можно задать функцию

$$F_{fb} : \{\sigma \uparrow P_{in}\} \rightarrow \{\sigma \uparrow P_o\}, \quad (1.35)$$

такую, что для любых двух поведений ФБ σ_1, σ_2 : если $\sigma_1 \uparrow P_{in} = \sigma_2 \uparrow P_{in}$, то $\sigma_1 \uparrow P_o = \sigma_2 \uparrow P_o$. То есть, если поведения на входе ФБ совпадают, то они должны совпадать и на выходе.

Отношение префикса $\sqsubseteq$ можно задать и для поведений. Поведения ФБ σ_1 и σ_2 связаны этим отношением, то есть $\sigma_1 \sqsubseteq \sigma_2$, если для всех портов ФБ $p \in P$: $\sigma_1(p) \sqsubseteq \sigma_2(p)$. Отношение $\sqsubseteq$ делает множество поведений ФБ частично упорядоченным. С учетом наименьшего элемента $\sigma_\perp$ (такого, что для любого $p \in P$: $\sigma_\perp(p)$ есть «пустой» сигнал $s_\perp : \emptyset \rightarrow V$, не содержащий событий), это множество становится полурешеткой, что позволяет применить к нему результаты теории решеток и, в частности, топологии Скотта. Этот подход используется в модели сигналов с тэгами для формального исследования свойств различных моделей вычислений.

С точки зрения этого подхода, ФБ обладают следующими свойствами: детерминизм (функциональность, (1.35)), монотонность (префикс сигналов на входе дает префикс на выходе), непрерывность (монотонность при целых сигналах), рецептивность (ФБ интерпретирует любой набор сигналов, не нарушающий условие (1.34)) и строгая каузальность (событие на выходе может быть сгенерировано строго после события на синхронном входе).

1.3.4.2 Вычисление временных характеристик ОСМВ ВвС на основе денотативного описания

Рассмотрим способ вычисления временных характеристик ОСМВ ВвС в рамках денотативного описания на основе свойств полного частично-упорядоченного множества атрибутов компонентов модели и теоремы Кнастера-Тарского о неподвижной точке. Предложенный способ учитывает иерархичность моделей, допускает произвольные функции вычисления атрибутов отдельных компонентов и использование обратной связи в модели.

ОСМВ предоставляет способ расчета вектора Ω , если известен вектор Γ , позволяя, таким образом, вычислить выходные характеристики ФБ по заданным входным. Если все инструкции интерпретируются функциональным блоком последовательно (например, при программной реализации), то

$$\Omega = T\Gamma, \quad (1.36)$$

то есть обычное умножение матрицы T на вектор Γ . Если разнотипные (то есть по разным синхронным входам) инструкции интерпретируются ФБ параллельно (например, при аппаратной реализации), то

$$\Omega = T \otimes \Gamma, \quad (1.37)$$

где $\otimes$ представляет собой операцию, результат которой есть вектор с элементами $\omega_i = \max_{j=1..M} \tau_{ij} \gamma_j$.

Математический аппарат ОСМВ также позволяет вычислить временные характеристики модели (композиции функциональных блоков), однако способ их вычисления требует, чтобы зависимость между атрибутами Γ и Ω всех ФБ в модели выражалась уравнениями (1.36) либо (1.37), использует нетривиальные преобразования матриц T функциональных блоков и не дает ответа при наличии обратной связи в модели.

Операции по нетривиальному (в зависимости от топологии конкретной модели и параллельной или последовательной интерпретации инструкций отдельными ФБ) объединению матриц T могут оказаться сложными в реализации и требовать больших затрат памяти при автоматизированном проектировании в рамках ОСМВ. Способ требует хранения в памяти «плоских» (то есть без учета иерархии) матриц T всех ФБ и получение одной, учитывающей матрицы каждого атомарного ФБ, матрицы T для всей проектируемой системы. В то же время современные ВСС могут выражаться тысячами и десятками тысяч примитивных элементов платформы (АФБ). Это означает, что применение такого способа расчета временных характеристик может затруднить в ряде случаев практическое использование ОСМВ для проектирования ВСС.

Частично упорядоченное множество атрибутов ФБ

Атрибут синхронного входа (γ_j) или выхода (ω_i) представляет собой минимальный интервал времени между событиями в сигнале (на заданном входе или выходе ФБ). Если в качестве модели времени использовать множество $T = \mathbf{R}_+ \cup \{\infty\}$ неотрицательных действительных чисел плюс бесконечность (что является довольно простой и интуитивно понятной моделью времени), то, очевидно, любое значение этих атрибутов будет являться элементом этого множества (отрезок времени).

Обозначим β_j некоторый атрибут (синхронного входа или выхода), равный минимальному интервалу между событиями в сигнале, $\beta_j \in T$. Между двумя значениями β_{j_1} и β_{j_2} можно установить отношение $\preceq$ такое, что

$$\beta_{j_1} \preceq \beta_{j_2} \Leftrightarrow \beta_{j_2} \leq \beta_{j_1}, \quad (1.38)$$

то есть оно инвертирует обычное отношение чисел $\leq$.

Обозначим $\beta_{\perp} = \infty$ интервал между событиями в сигнале, который содержит менее двух событий (например, начальный сброс или однократное нажатие кнопки). Тогда множество всех значений β_j будет полностью упорядочено отношением $\preceq$ и будет иметь наименьший элемент $\beta_{\perp}$.

Возьмем вектор $\mathbf{v}$, содержащий M элементов β_j . Очевидно, $\mathbf{v} \in T^M$. Установим между любыми двумя векторами $\mathbf{V}_1, \mathbf{V}_2 \in T^M$ отношение $\preceq$ такое, что

$$\mathbf{V}_1 \preceq \mathbf{V}_2 \Leftrightarrow \forall j = 1 \dots M : \beta_{j1} \preceq \beta_{j2}, \quad (1.39)$$

где β_{j1} и β_{j2} есть элементы первого и второго векторов соответственно. То есть два вектора связаны отношением $\preceq$ тогда и только тогда, когда все их соответствующие элементы связаны отношением (1.38). Отношение (1.39) рефлексивно ($\mathbf{V}_1 \preceq \mathbf{V}_1$), антисимметрично (если $\mathbf{V}_1 \preceq \mathbf{V}_2$ и $\mathbf{V}_2 \preceq \mathbf{V}_1$, то $\mathbf{V}_1 = \mathbf{V}_2$) и транзитивно (если $\mathbf{V}_1 \preceq \mathbf{V}_2$ и $\mathbf{V}_2 \preceq \mathbf{V}_3$, то $\mathbf{V}_1 \preceq \mathbf{V}_3$) и делает множество векторов T^M частично упорядоченным.

Множество T^M содержит вектор $\mathbf{V}_{\perp}$, все элементы которого равны $\beta_{\perp}$. Легко показать, что $\mathbf{V}_{\perp}$ является наименьшим элементом T^M с отношением $\preceq$, что превращает это множество в полное частично-упорядоченное множество. Каждое его направленное подмножество имеет точную верхнюю грань (так как любой элемент вектора β_j ограничен значением 0), а все множество имеет наименьший элемент $\mathbf{V}_{\perp}$.

Функция вычисления атрибутов модели

Вектор Ω может быть вычислен по вектору Γ с помощью соотношения (1.36) или (1.37). Абстрагируемся от их сути и заменим их функцией вычисления атрибутов выходов по атрибутам синхронных входов $F_{\Omega} : T^M \rightarrow T^N$ (M есть число синхронных входов, N число выходов ФБ). Единственное свойство (1.36) и (1.37), которое оставим, будет непрерывность F_{Ω} (в самом деле, легко показать, что преобразования (1.36) и (1.37) непрерывны).

Рассмотрим произвольную композицию конечного числа ФБ с функциями вычисления атрибутов $F_{\Omega 1}, F_{\Omega 2}, \dots, F_{\Omega n}$. Каждая из этих функций вычисляет атрибуты соответствующих ФБ $\Omega_1, \Omega_2, \dots, \Omega_n$ по их атрибутам $\Gamma_1, \Gamma_2, \dots, \Gamma_n$. Элементы этих векторов представляют собой атрибуты выходов или синхронных входов соответствующих ФБ. Если некоторое ω_i является атрибутом выхода ФБ с номером l , то оно вычисляется с помощью функции $F_{\Omega l}$ при наличии атрибутов всех его синхронных входов, то есть, если можно сформировать вектор Γ_l . Тогда, если есть атрибуты всех синхронных входов (то есть можно сформировать все векторы $\Gamma_1, \Gamma_2, \dots, \Gamma_n$), то с помощью соответствующих функций можно вычислить атрибуты всех выходов.

Разделим все множество синхронных входов композиции на множество «внешних» и «внутренних». «Внутренний» вход отличается тем, что он имеет соединение с одним из выходов ФБ в композиции. Например, для композиции на рис. 1.23 (все входы в которой синхронные) «внутренними» являются входы, атрибуты которых обозначены γ_2 , γ_3 и γ_5 , а внешними являются входы с атрибутами γ_1 и γ_4 .

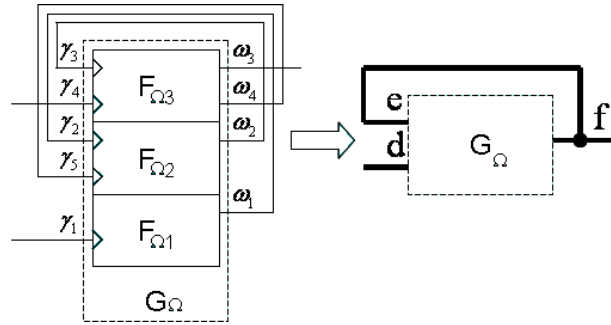


Рис. 1.23. Приведение произвольной композиции ФБ к виду с обратной связью

Сформируем из атрибутов «внешних» синхронных входов вектор $\mathbf{d}$. Сформируем из атрибутов «внутренних» синхронных входов вектор $\mathbf{e}$ и дополним его при необходимости элементами со свободными значениями так, чтобы число элементов этого вектора совпадало с числом всех выходов композиции. Для примера на рис. 1.23 вектор $\mathbf{e}$ придется дополнить одним элементом, так как число выходов на единицу больше числа «внутренних» синхронных входов. Сформируем из атрибутов всех выходов композиции вектор $\mathbf{f}$ таким образом, чтобы его элемент с номером i был атрибутом выхода, соединенного с «внутренним» синхронным входом, атрибут которого является элементом с номером i вектора $\mathbf{e}$. Для этого, возможно, потребуется продублировать некоторые элементы вектора $\mathbf{f}$, если выход соединен с несколькими «внутренними» входами. Для примера на рис. 1.23 векторы будут иметь значения

$$\mathbf{d} = \begin{bmatrix} \gamma_1 \\ \gamma_4 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} \gamma_2 \\ \gamma_3 \\ \gamma_5 \\ x \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_4 \\ \omega_3 \end{bmatrix},$$

где x есть свободный элемент, который может принимать любое значение.

Выше было замечено, что если известны атрибуты всех синхронных входов, то с помощью соответствующих функций $F_{\Omega 1}$, $F_{\Omega 2}$, ..., $F_{\Omega n}$ можно вычислить атрибуты всех выходов. Обозначим G_{Ω} функцию, вычисляющую вектор $\mathbf{f}$, содержащий атрибуты всех выходов композиции, по векторам $\mathbf{d}$ и $\mathbf{e}$:

$$\mathbf{f} = G_{\Omega}(\mathbf{d}, \mathbf{e}). \quad (1.40)$$

При этом если некоторый элемент ω_i вектора $\mathbf{f}$ является атрибутом выхода ФБ с номером l , то он вычисляется с помощью функции $F_{\Omega l}$. G_{Ω} можно рассматривать как суперпозицию функций $F_{\Omega 1}, F_{\Omega 2}, \dots, F_{\Omega n}$. Очевидно, если все функции $F_{\Omega 1}, F_{\Omega 2}, \dots, F_{\Omega n}$ непрерывны, то G_{Ω} также непрерывна.

Обозначим $G_d(\mathbf{e}) = G_{\Omega}(\mathbf{d}, \mathbf{e})$ значение функции G_{Ω} при заданных атрибутах «внешних» синхронных входов $\mathbf{d}$.

Вследствие того, что все «внутренние» входы в композиции соединены с некоторыми выходами, а свободные элементы вектора $\mathbf{e}$ могут принимать любые значения, $\mathbf{e} = \mathbf{f}$ и

$$\mathbf{e} = G_d(\mathbf{e}). \quad (1.41)$$

Вычисление (а при наличии явной обратной связи в композиции, и *вычислимость*) атрибутов всех выходов в композиции сводится к нахождению (и существованию) неподвижной точки функции G_d , удовлетворяющей уравнению (1.41).

Множество E всех значений векторов $\mathbf{e}$ с отношением $\preceq$ является полным частично упорядоченным множеством. Теорема о неподвижной точке, которая считается одним из вариантов теоремы Кнастера-Тарского [78], формулирует условие существования и позволяет найти неподвижную точку функции (1.41).

Теорема 1.5. Пусть дано полное частично-упорядоченное множество $(E, \preceq)$. Тогда для любой непрерывной функции $G_d : E \rightarrow E$ существует неподвижная точка $\mathbf{e} = G_d(\mathbf{e})$. Причем, эта точка может быть найдена вычислением последовательности

$$\mathbf{e}_{\perp}, G_d^1(\mathbf{e}_{\perp}), G_d^2(\mathbf{e}_{\perp}), \dots, G_d^n(\mathbf{e}_{\perp}),$$

для некоторого n , в которой $G_d^i(\mathbf{e}_{\perp}) = G_d(G_d^{i-1}(\mathbf{e}_{\perp}))$, $G_d^0(\mathbf{e}_{\perp}) = \mathbf{e}_{\perp}$, а $\mathbf{e}_{\perp}$ есть минимальный элемент E .

Функция G_d непрерывна, так как непрерывна G_{Ω} [78], и ее неподвижная точка всегда может быть найдена за конечное число шагов. Для того, чтобы вычислить атрибуты всех выходов композиции, достаточно взять вектор атрибутов «внешних» синхронных выходов $\mathbf{d}$, в качестве начального значения атрибутов «внутренних» входов взять вектор $\mathbf{e}_{\perp}$ (все элементы которого будут равны $\beta_{\perp}$), и итеративно вычислять значения функции G_d . На некотором шаге последовательность вычисленных значений сойдется к неподвижной точке.

Обозначим неподвижную точку (1.41) $\text{fix}(G_d)$. Если вспомнить, что $\mathbf{e} = \mathbf{f}$, то можно переписать (1.40):

$$\mathbf{f} = G_{\Omega}(\mathbf{d}, \text{fix}(G_d)) = F_{\Omega}(\mathbf{d}). \quad (1.42)$$

Функция $F_{\Omega} : D \rightarrow E$ (D есть множество всех значений векторов $\mathbf{d}$) представляет собой функцию вычисления атрибутов всех выходов композиции при заданном значении атрибутов «внешних» синхронных входов (см. рис. 1.24). F_{Ω} непрерывна, если непрерывна G_{Ω} [78].

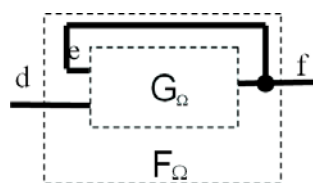


Рис. 1.24. Функция (1.42) вычисления атрибутов

Предложенный способ не требует конкретного вида функций $F_{\Omega_1}, F_{\Omega_2}, \dots, F_{\Omega_n}$ вычисления атрибутов отдельных ФБ, а требует только, чтобы они были непрерывны. Зависимость между атрибутами выходов и входов отдельных ФБ может выражаться соотношением (1.36) или (1.37) и потенциально любым другим непрерывным соотношением.

Способ не зависит от порядка интерпретации инструкций (последовательного или параллельного) функциональными блоками модели. Это позволяет задавать способы расчета атрибутов для смешанных (параллельно-последовательных) вариантов порядка интерпретации [18] и использовать их для расчета атрибутов отдельных ФБ при вычислении атрибутов для всей модели [17].

Математический аппарат ОСМВ не позволяет вычислить атрибуты выходов при наличии обратной связи в модели. Для функциональных блоков со строгой зависимостью выходов от входов (событие на выходе не может быть сгенерировано, пока не поступят инструкции по всем необходимым входам), которую задает матрица T , это и не требовалось. Легко показать, что любая композиция таких ФБ с обратной связью приведет к блокировке (deadlock). Тем не менее, обратная связь представляет собой базовый подход в проектировании управляющих систем, и ОСМВ в его отсутствие вряд ли получит широкое использование [18]. Актуальность этого вопроса подчеркивается тем, что аналогичная проблема существует и в синхронных языках (Statecharts, Esterel) [48]. В них она решается введением «нестрогих» компонентов, позволяющих генерацию событий на выходах, если известно состояние только части входов. Исследование в области формального обоснования «нестрогих» функциональных блоков могло бы решить эту проблему в ОСМВ. Способ вычисления атрибутов таких ФБ, очевидно, может отличаться от (1.36) и (1.37). Предложенный способ вычисления временных характеристик композиции, тем не менее, делает возможным применение таких ФБ в моделях.

В отличие от математического аппарата ОСМВ, предложенный способ учитывает иерархию модели в том смысле, что при изменении одной из моделей на более низком уровне абстракции требуется пересчитать только ту часть высокоуровневой модели, которая непосредственно зависит от нее, а не

всю модель целиком. Способ не требует построения и хранения в памяти матриц T , что делает его более экономичным при вычислении характеристик крупных моделей, с точки зрения используемой памяти.

Выводы

1. Модель вычислений (MoC) ВcC является важнейшим (поведенческим) аспектом архитектурного проектирования.
2. На основе анализа базовых MoC [70] и типовых вычислительных архитектур, реализуемых в виде виртуальных языковых машин и различных процессорных ядер, рекомендуется расширение трактовки MoC на область таких вычислительных архитектур.
3. MoC классифицируются по совместности вычислений:
 - с неявной спецификацией совместности (data-driven, event-driven, demand-driven);
 - с явной спецификацией совместности (control-driven, imperative).
4. Существует возможность эффективно использовать элементы параллельной модели на основе функциональных блоков (по аналогии с аппаратными блоками ВС) для представления распределенных ВcC.
5. На основе анализа традиционных подходов к описанию распределенных ВcC и учета требований перспективных методик проектирования сформулированы базовые понятия объектно-событийной модели вычислений (ОСМВ) РИУС.
6. Показаны типовые методы для синтетического и аналитического способов построения ОСМВ. Структурно-функциональная декомпозиция модели опирается на принцип выделения временных доменов функциональных блоков. Свойство полиморфизма структурных элементов модели позволяет абстрагироваться от особенностей реализации отдельных узлов системы.
7. Важное значение имеют частные критерии аналитического построения модели целевой системы. К ним относятся критерий выделения временных доменов ФБ, критерий сильной/слабой связи, критерий определения управляющих входов и критерий программно-аппаратной реализации.
8. При расчете временных характеристик учитывается специфика программной и аппаратной реализации отдельных элементов ОСМВ, которая формальным образом влияет на эволюцию модели в процессе проектирования.
9. Рассмотренные базовые механизмы функционирования основных элементов ОСМВ позволяют получать эффективные реализации модели на современной элементной базе.
10. Для расчета временных характеристик ОСМВ представлен математический аппарат, позволяющий получать оценки программной и аппаратной реализации отдельных элементов, рассчитывать реактивные свойства,

выявлять критические участки модели. Математический аппарат может быть положен в основу САПР и средств моделирования ВСС.

11. В качестве примера продемонстрированы оценки эффективности фрагментации потока данных и широковещательных обменов при решении параметризованной задачи совмещения моделей вычислений в рамках коммуникационной системы (КМС) ВСС. Представлены частные критерии оценки протоколов КМС распределенных ВСС.
12. Для применения ОСМВ в комплексном моделировании разработано ее расширение на основе денотативного описания, которое в отличие от операционного подхода описывает вычисления в терминах свойств функциональных преобразований сигналов. Дан способ вычисления временных характеристик ОСМВ ВСС в рамках денотативного описания на основе свойств полного частично-упорядоченного множества атрибутов компонент модели и теоремы Кнастера-Тарского о неподвижной точке. Способ учитывает иерархичность моделей, допускает произвольные функции вычисления атрибутов отдельных компонент и использование обратной связи в модели.

2 Применение методов и средств проектирование встраиваемых вычислительных систем

2.1 Анализ технических решений уровня реализации ВcС

В данной главе на примерах поясняется техника формирования и анализа проектных решений ВcС на основе методологии высокоуровневого проектирования. Использована задача проектирования вычислительной системы MiniLab семейства сканирующих зондовых микроскопов (СЗМ) (подсистема визуализации консоли физического узла и проектирование драйвера периферийного интерфейса). Показано получение архитектурных абстрактных моделей и их реализаций с различной функциональной декомпозицией по шкале HW/SW. Представлен анализ проекта специализированных гетерогенных вычислителей МЗМ системы железнодорожной автоматики управления движением «Тракт».

Приборный контроллер СЗМ MiniLab в качестве примера будет использован в разделах 2.1 и 2.3, поэтому отметим особенности его архитектуры (рис. 2.1):

- гетерогенная архитектура центрального вычислителя;
 - запатентованный специализированный VLIW процессор цифровой обработки сигналов на базе ПЛИС(FPGA);
 - алгоритмизация эксперимента посредством Java Virtual Machine;
- стандартные программные интерфейсы связи с программным обеспечением персонального компьютера;
 - USB SIC Device Class: фотокамера для сканирования, обработки и получения изображений;
 - USB Audio Device Class: состояние в реальном времени топологии схемы цифровой обработки сигналов;
 - VISA/IVI: совместимость с LabVIEW и Matlab;
- компактный дизайн, энергосберегающие технологии.

2.1.1 Пример подсистемы визуализации аналитического прибора

Рассмотрим варианты проектирования подсистемы визуализации в составе физического измерительного узла СЗМ в проекте MiniLab.

В начальном варианте (рис. 2.2) управление выводом графической информации на ЖКИ было реализовано в виде системной программной задачи. Интерфейс к ЖКИ был реализован в виде простейшего блока в ПЛИС, который отвечал за преобразование полученных по шине I²C команд в команды управления ЖКИ.

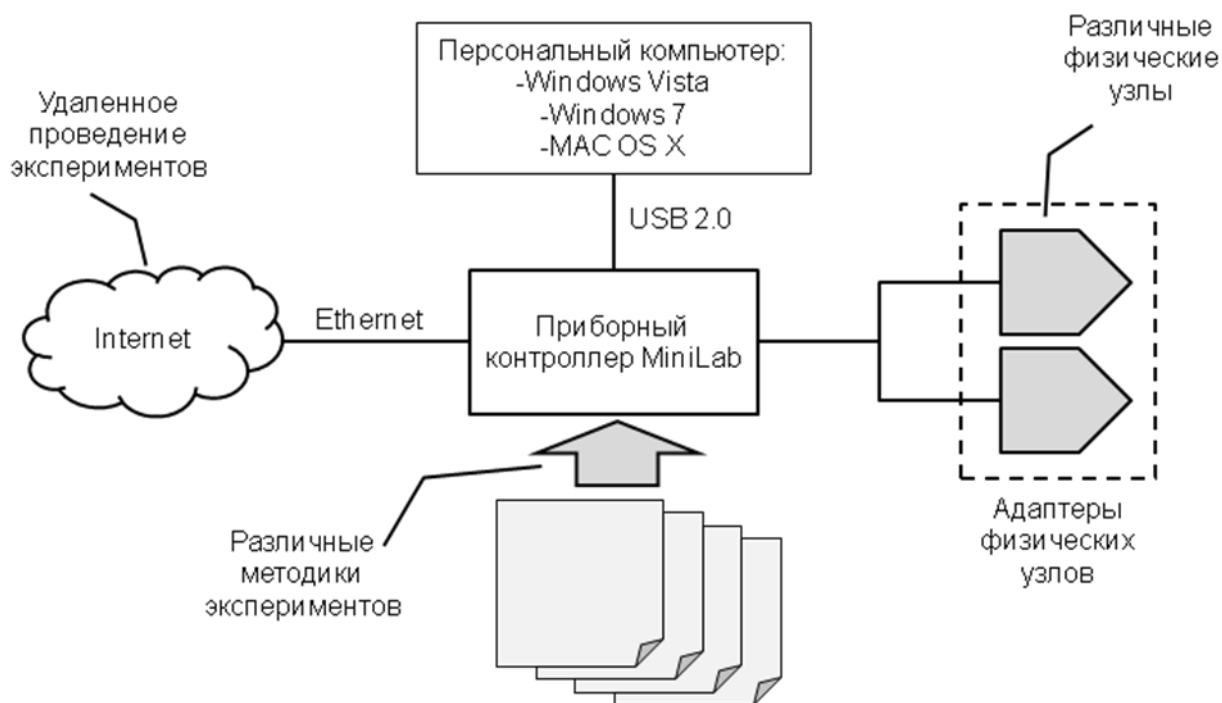


Рис. 2.1. Приборный контроллер СЗМ MiniLab

Скорость обновления графической информации на ЖКИ была невысокой, вследствие необходимости реализовывать в драйвере ЖКИ все действия по преобразованию данных в низкоуровневые команды внутреннего контроллера ЖКИ и передаче этих команд по низкоскоростному интерфейсу I2C. Помимо этого, быстро сокращающиеся размеры памяти кода в микроконтроллере привели к необходимости пересмотра применяемого подхода к отображению информации на ЖКИ.

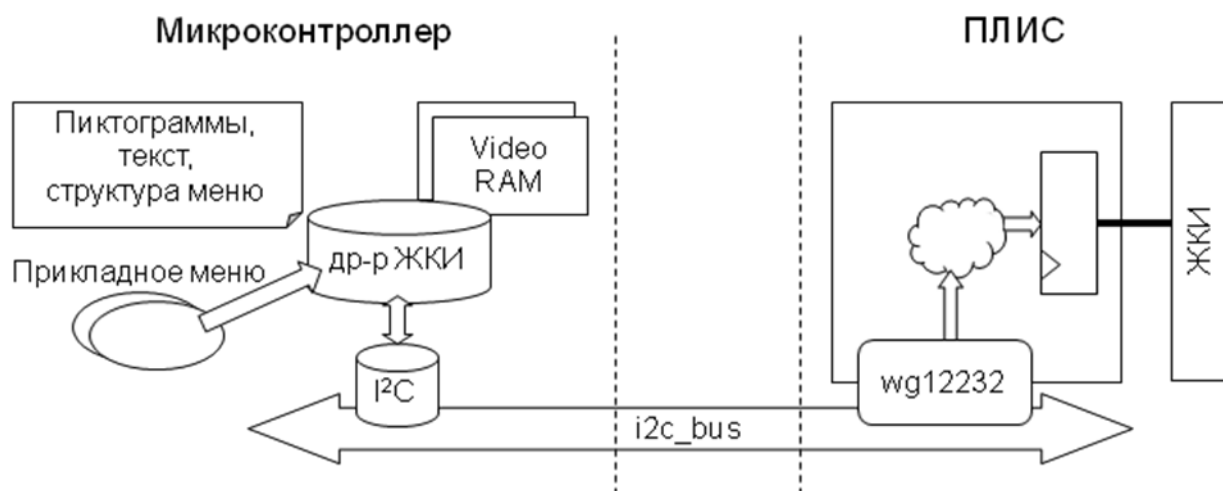


Рис. 2.2. Вариант 1 подсистемы управления индикатором

Было принято решение отказаться от возможности вывода графических примитивов, таких как “точка”, “линия”, средствами микроконтроллера и ограничиться выводом текстовых строк и растровых изображений.

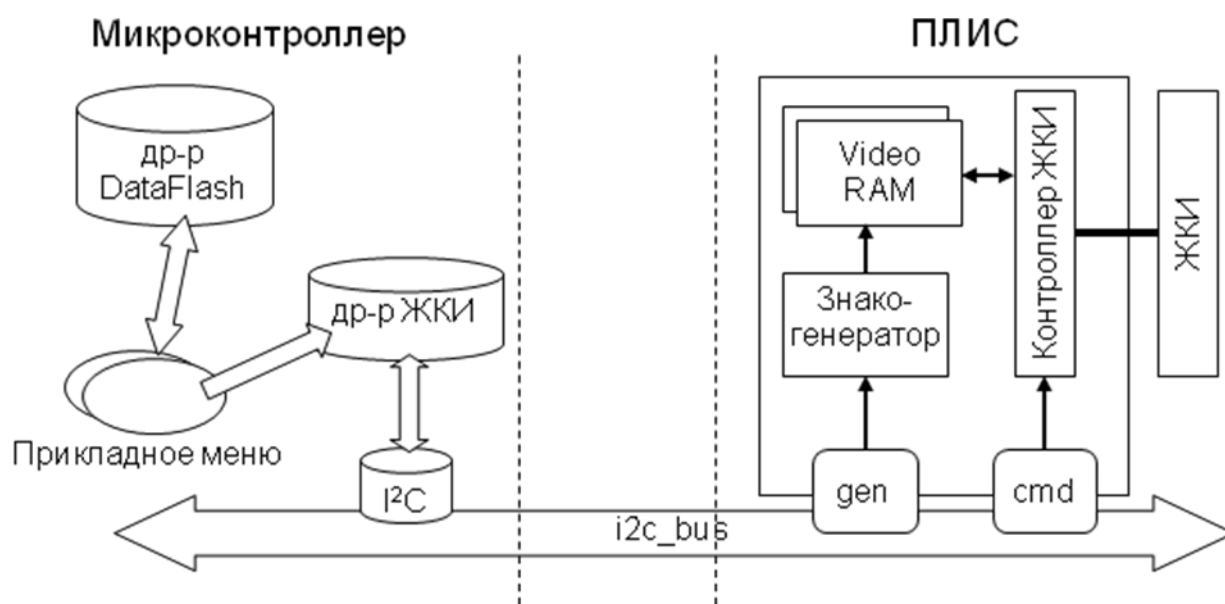


Рис. 2.3. Вариант 2 подсистемы управления индикатором

Графические ресурсы меню стали располагаться в DataFlash, видеопамять и вся логика управления ЖКИ (в том числе знакогенератор) перенесены в ПЛИС (рис. 2.3).

Сравнение вариантов представлено в табл. 2.1.

Таблица 2.1. Использование вычислительных ресурсов в системе отображения физического узла СЗМ

	Начальная версия (1)	Рабочая версия (2)
Микроконтроллер		
Прикладные алгоритмы	10 КБ (2 КБ)	4 КБ (2 КБ)
Видеопамять	1,5 КБ	—
Знакогенерация	6 КБ (2 КБ)	—
Управление ЖКИ		900 байт (6 байт)
ПЛИС		
Прикладные алгоритмы	—	—
Видеопамять	—	488x9 (1 блок памяти)
Знакогенерация	—	100 регистров, 4 счетчика
Управление ЖКИ	15 регистров, 2 счетчика	250 регистров, 7 счетчиков

В начальной версии производилось периодическое копирование обновленных данных из видеопамати во внутренний контроллер ЖКИ. Все

операции по обмену с ЖКИ по шине I²C выражались с помощью единственной функции WritePort(N,char*), следующего формата:

1 байт

DEV_ADDR	L2	L1	C_D
----------	----	----	-----

N байт

cmd[] или data[]

где

DEV_ADDR – адрес устройства на шине I²C,

L1, L2 – выбор внутреннего контроллера ЖКИ,

C_D – выбор типа передаваемых данных (команды или данные),

cmd[] – массив команд,

data[] – массив данных.

В рабочей версии протокол был усовершенствован в плане оптимизации передаваемых данных: WriteData(x,y,N,char*,Repeat) – запись блока данных с повтором, начиная с заданных координат; WriteCharArray(x,y,N,char) – запись набора символов, начиная с заданных координат; WritePort(cmd) – запись команды напрямую во внутренний контроллер ЖКИ.

Формат команды WriteData(x,y,N,char*,Repeat):

1 байт

DEV_ADDR	M	XY	L
----------	---	----	---

1 байт

1 байт

1 байт

1 байт

N байт

[Mask]	[X]	[Y]	[LEN]	data[]
--------	-----	-----	-------	--------

Формат команды WriteCharArray(x,y,N,char):

1 байт

DEV_ADDR	–	XY	L
----------	---	----	---

1 байт

1 байт

1 байт

N байт

[X]	[Y]	[LEN]	char[]
-----	-----	-------	--------

Формат команды WritePort(cmd):

1 байт

DEV_ADDR	L2	L1	C_D
----------	----	----	-----

1 байт

command

где

DEV_ADDR – адрес устройства на шине I²C,

M – признак наличия поля Mask,

XY – признак наличия полей X и Y,

L – признак наличия поля LEN,

Mask – битовая маска записываемых данных,

X и Y – координаты для вывода данных,

LEN – количество повторений полей data[] и char[],

data[] – растровая информация,
char[] – коды символов,
command – низкоуровневые данные контроллера ЖКИ.

Ниже приведено сравнение (табл. 2.2) рассмотренных подходов.

Таблица 2.2. Сравнение версий реализации подсистемы визуализации

	Начальная версия	Рабочая версия
Скорость отображения	Низкая	Высокая
Объем ресурсов МК	Много	Средне
Объем ресурсов ПЛИС	Очень мало	Значительно
Нагрузка на шину I2C	Высокая	Низкая
Графические примитивы	Средне	Мало

2.1.2 Пример проектирования драйвера периферийного устройства

В качестве примера рассматривается фаза высокоуровневого проектирования драйвера интерфейса контроллерной сети в стандарте 1-Wire. Необходимо определить набор примитивов (API) драйвера с учетом особенностей протокола, «физики» интерфейса, номенклатуры подключаемых устройств, предполагаемой статистики транзакций, особенностей прикладной задачи. Проектирование включает ряд шагов анализа и синтеза.

- Особенности шины и организация ввода/вывода в стандарте 1-Wire bus [36]
 - Варианты типового обмена: Reset 1-Wire bus; Select 1-Wire device(s); Perform a device specific operation.
 - Сигналы: RESET, PRESENCE PULSE; WRITE-ONE; WRITE-ZERO; READ; STRONG PULL-UP; PROGRAM.
 - Типы кадров: INIT; ADDRESS; DATA.
- Анализ номенклатуры и функциональности устройств 1-Wire

Номенклатура устройств 1-Wire сегодня насчитывает 60 типов микросхем в 43 группах по функциональности. На рис. 2.4 представлен формат идентификатора контроллера 1-Wire, на рис. 2.5 – элементы архитектуры контроллера DS2433.

8-Bit CRC Code		48-Bit Serial Number		8-Bit Family Code (01h)	
MSB	LSB	MSB	LSB	MSB	LSB

Рис. 2.4. Формат идентификатора контроллера 1-Wire

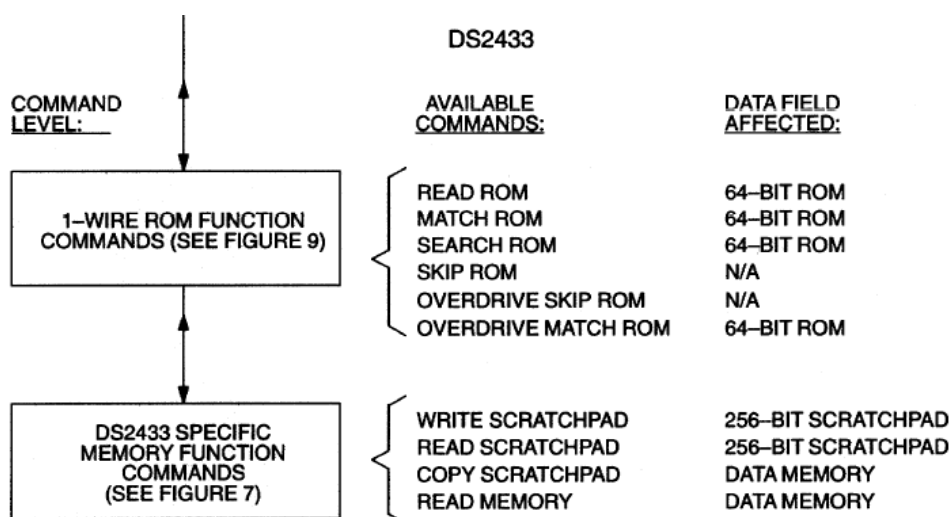


Рис. 2.5. Элементы архитектуры контроллера DS2433

Таблица 2.3. Адресация (ROM)

		DS 2780	DS 2762	DS 2740	DS 2502	DS 2450	DS 2438	DS 2433	DS 2431	DS 2422
READ ROM	(33) Rx8	+	+	+	+	+	+	+	+	+
READ ROM	(39) Rx8	+	+	+						
READ ROM	(0F) Rx8									
MATCH ROM	(55) Tx8	+	+	+	+	+	+	+	+	+
SWAP	(AA) Tx8		+							
OVER-DRIVE MATCH	(69) Tx8					+		+	+	+
SEARCH ROM	(F0) Sx8	+	+	+	+	+	+	+	+	+
ALARM SEARCH	(EC) Sx8					+				+
SKIP ROM	(CC)	+	+	+	+	+	+	+	+	+
RESUME	(A5)	+		+					+	+
OVER-DRIVE SKIP	(3C)					+		+	+	+

3. Анализ алгоритмов функционирования устройств 1-Wire

Выделяются устройства 1-Wire со стандартными операциями, связанными с обслуживанием ROM и устройства специального назначения (таймеры, АЦП

и др.). Одно устройство может сочетать в себе группы функциональности. На рис. 2.7 приведен фрагмент блок-схемы для контроллера DS2433.

4. Анализ кадра протокола обмена 1-Wire устройств.

- Базовые примитивы INIT: RP - reset pulse + presence pulse.
- Базовые примитивы ADDRESS (табл. 2.3, рис. 2.6):
- (hex) – код команды; Tx8 – передача 8 байтов; Rx8 – прием 8 байтов; Sx8 – передача и прием 8 байтов (команда триплет).
- В кадре для DS1821 отсутствует фаза ADDRESS.
- Базовые примитивы DATA: (hex) – код команды; Tx – передача байтов; Rx – прием байтов; RxC – прием байтов с проверкой; W[01] – ожидание подтверждения в виде смены считываемого бита с 0 на 1; W[AA] – ожидание подтверждения в виде чередующихся битов 0 и 1; W – ожидание в течение заданного времени; PUW – strong pullup и ожидание в течение заданного времени; PP – импульс программирования; {...} – возможно повторение более 1 раза.

8-Bit CRC Code		48-Bit Serial Number		8-Bit Family Code (01h)	
MSB	LSB	MSB	LSB	MSB	LSB

Рис. 2.6. Общий формат адреса 1-Wire устройства

В табл. 2.4 и 2.5 приведена исходная информация и статистика по использованию при обмене примитивов DATA каждого типа. На рис. 2.9 приведен пример базового примитива DATA.

5. Настройки шины 1-Wire устройств

- Проверка возможностей: Overdrive; Presence Mask; Active Pull-up; Strong Pull-up; Program Pulse.
- Статическое конфигурирование шины: Presence Mask; Active Pull-up.
- Динамическое конфигурирование шины: Overdrive; Strong Pull-up; Program Pulse.

6. Структура запроса ввода/вывода

На основе анализа предыдущей информации формируется структура запросов драйвера (рис. 2.8) с соответствующим составом полей.

INIT	ADDRESS	DATA
------	---------	------

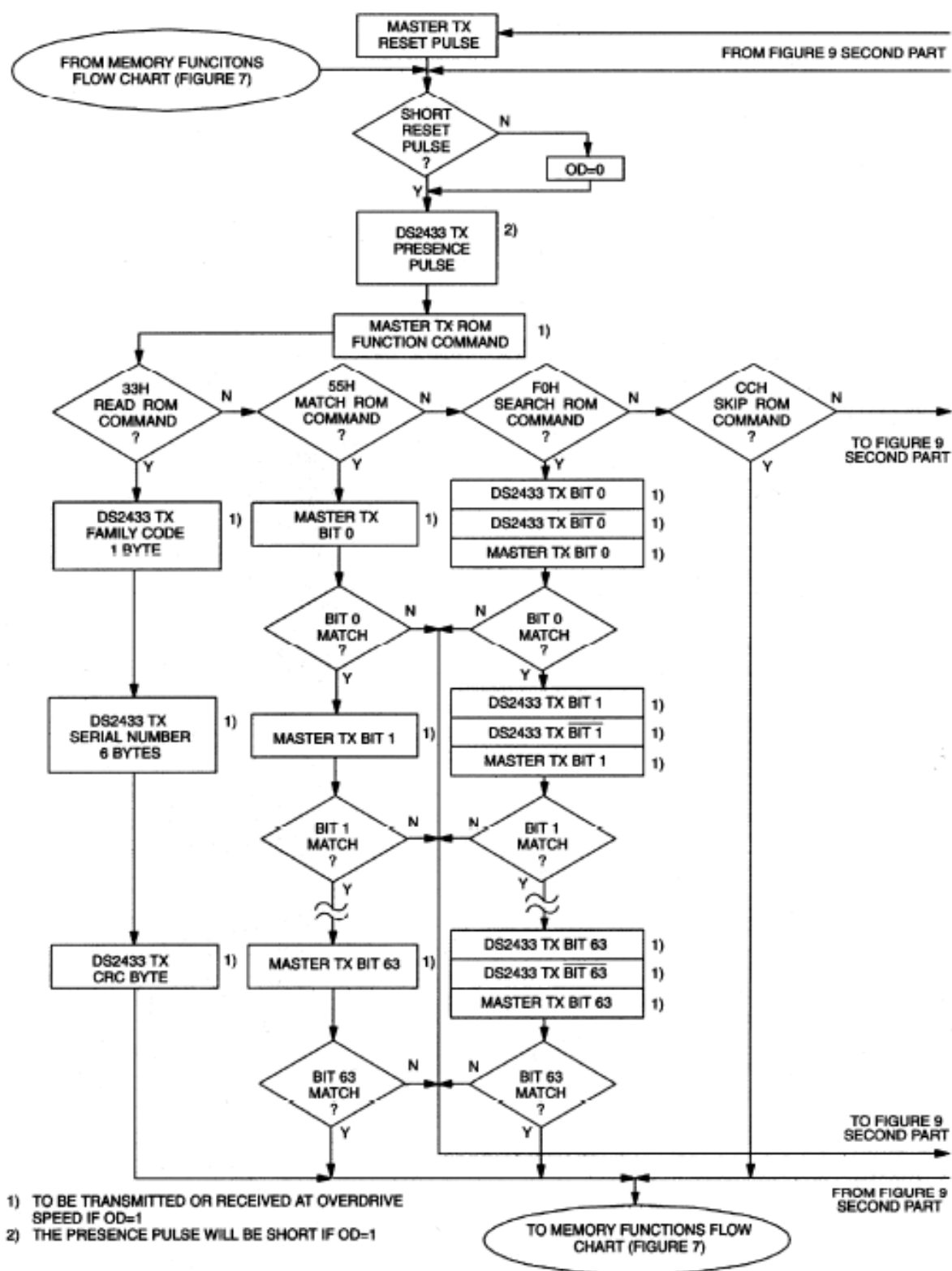


Рис. 2.7. Функции работы с ROM DS2433

Таблица 2.4. Базовые примитивы DATA в работе 1-Wire устройств

	DS 2780	DS 2762	DS 2740	DS 2502	DS 2450	DS 2438	DS 2433	DS 2431	DS 2422
(hex)									
(hex) PUW									
(hex) Tx	4	4	1			1			4
(hex) Rx							1	1	1
(hex) RxC									
(hex) Tx RxC							1	1	1
(hex) Tx Rx	1	1	1		1	1	1	1	1
(hex) W[01]						2			
(hex) Tx W[01]						2			
(hex) {Tx RxC Rx}				3					
(hex) Tx W[AA]									1
(hex) Tx PUW W[AA]							1	1	
(hex) Tx RxC PUW W[01]					1				
(hex) Tx {Tx RxC}					1				
(hex) Tx {Tx RxC PP RxC}				2					
(F5) ???									

Таблица 2.5. Статистика использования базовых примитивов DATA

Примитив	Устройства	Команды	Процент
(hex)	19 / 19	106 / 106	100% / 100%
Tx	19 / 19	58 / 106	100% / 55%
Rx	19 / 19	39 / 106	100% / 37%
RxC	9 / 19	18 / 106	47% / 17%
W[01]	6 / 19	17 / 106	32% / 16%
W[AA]	5 / 19	5 / 106	26% / 5%
W	0 / 19	0 / 106	0% / 0%
PUW	9 / 19	13 / 106	47% / 12%
PP	2 / 19	4 / 106	11% / 4%
{...}	5 / 19	11 / 106	26% / 10%

31...24	23...16	15...8	7...0
fMode	cmdA	idcd	cmdD
ID[0]	ID[1]	ID[2]	ID[3]
ID[4]	ID[5]	ID[6]	ID[7]
iMode	-	Count / Timeout	
Data / Data pointer / Device handle			

fMode						
ovdrv	rst	-	aCmd	aMode	dCmd	dItem

iMode					
-	-	wMode	iType	-	dItem

Рис. 2.8. Структура запроса ввода/вывода

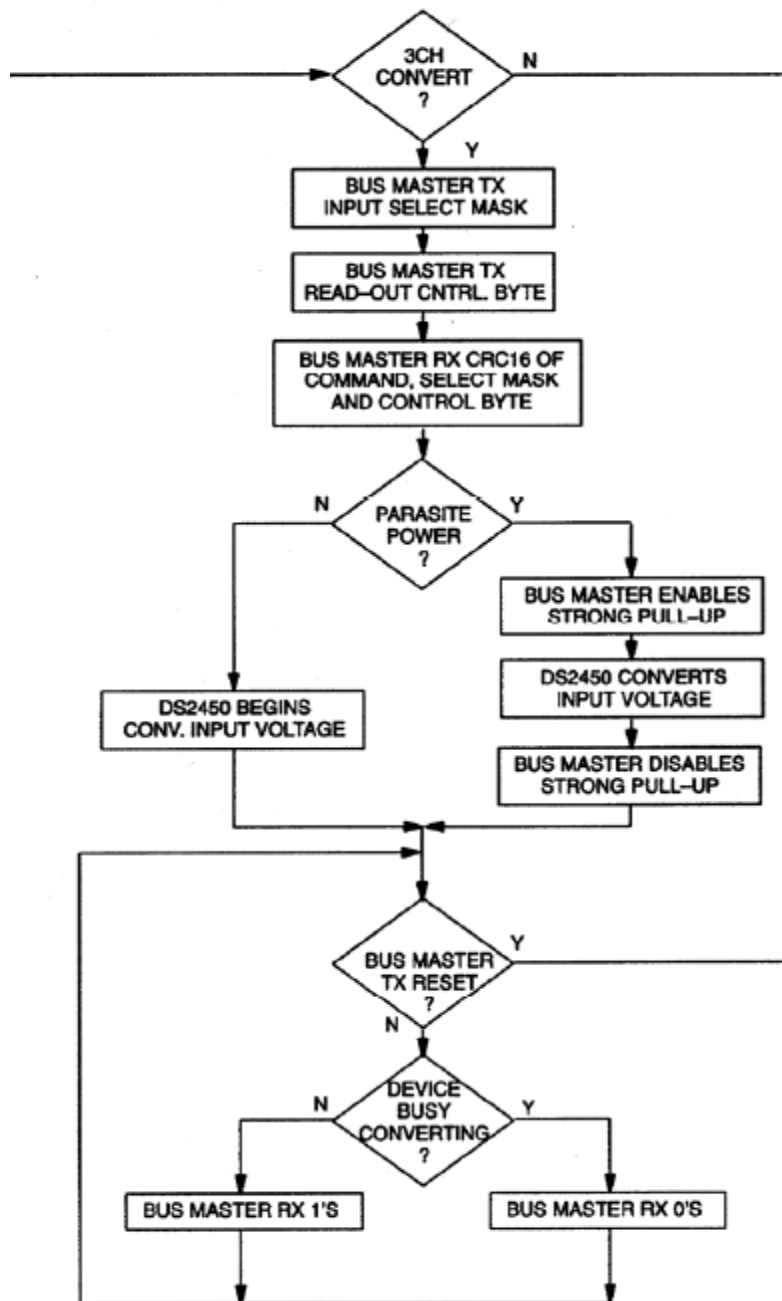


Рис. 2.9. Пример базового примитива DATA

Поля запроса ввода/вывода:

- fMode – конфигурация кадра:
 - o ovdrv – обмен данными должен осуществляться в режиме overdrive;
 - o rst – не выдавать 1-Wire RESET в начале INIT фазы;
 - o aCmd – не передавать поле cmdA в начале ADDRESS фазы;
 - o aMode – режим обработки поля ID[7:0] (Tx/Rx/Sx/Nx);
 - o dCmd – не передавать поле cmdD в начале DATA фазы
 - o dItem – в запросе присутствует элемент ввода/вывода DATA;

- cmdA – команда адресации (aCmd=0);
- idcd – глубина конфликта в режиме адресации Sx;
- ID[7:0] – уникальный идентификатор (адрес);
- cmdD – функциональная команда (dCmd=0);
- iMode – конфигурация элемента ввода/вывода:
 - iType – тип элемента (Tx/Rx/Check/Wait);
 - wMode – режим ожидания (Normal / Strong Pull-up / Program / Ack) или режим доступа к данным (data / data pointer);
 - dItem – в запросе присутствует еще элемент ввода/вывода DATA;
- Count / Timeout – размер данных / время ожидания⁴
- Data / Data Pointer / Device Handle – данные / драйвер ожидания.

Требования к реализации запроса ввода/вывода:

- Обеспечение целостности обмена в пределах стандартного кадра 1-Wire на уровне драйвера.
- Возможность непосредственно указывать состав фаз запроса ввода/вывода (INIT / ADDRESS / DATA).
- Автоматическое переключение в режим overdrive и обратно.
- Автоматические ожидания в режиме Strong Pull-up и Programming.
- Использование полей запроса ввода/вывода фазы DATA для хранения небольших объемов данных (до 4-х байтов) позволяет осуществлять ввод/вывод больших структур данных с контролем CRC без копирования блоков памяти.

7. Перечень операций драйвера 1-Wire:

- QueryDriverCapability – получение информации о возможностях драйвера.
- BusSetup – настройка параметров статической конфигурации шины.
- DataExchange – обмен данными согласно запросу ввода/вывода.

2.1.3 Проектирование специализированных вычислителей системы железнодорожной автоматики «Тракт»

2.1.3.1 Многоуровневая система обеспечения функциональной безопасности транспортной автоматики

Необходимый уровень функциональной безопасности микропроцессорных систем управления на железнодорожном транспорте может быть обеспечен только комплексом технических и организационных решений, которые должны

учитывать как архитектурную иерархию, так и многоэтапность процесса проектирования и производства.

Многоуровневая организация систем управления движением поездов вытекает из сложности выполняемых функций и пространственной протяженности и является естественной. Под функциональной безопасностью такого рода систем понимается их свойство выполнять целевую функцию или (в случае сбоев или отказов) переходить в безопасное для объекта состояние.

Сложность систем транспортной автоматики не позволяет решить проблему функциональной безопасности только на уровне "выходного каскада". В системе должны присутствовать элементы контроля функционирования в ряде точек информационного тракта, увязанные в контуры контроля безопасности функционирования подсистем по иерархическому принципу.

Реализация контрольных механизмов и механизмов принятия решения на сегодняшний день достаточно хорошо известна и сводится к нескольким группам технических решений. Такие типовые решения предполагают аппаратную, программную или смешанную реализацию в зависимости от уровня иерархии контролируемых процессов.

Открытым, в значительной степени, остается вопрос оценки качества контроля при функционировании системы. Здесь перспективным представляется подход, в основе которого лежит формализованное представление архитектурных элементов вычислительной системы, направленное на проектирование системы в целом из механизмов, инвариантно к их последующей реализации [21, 24, 25].

Необходимо отметить возрастающее влияние на уровень функциональной безопасности микропроцессорных систем ряда новых факторов, таких как:

- высокая сложность готовых аппаратных и программных компонент, для которых трудно гарантировать достаточную верификацию (т.е. соответствие заявленным спецификациям);
- требование "прозрачности" проекта в такой степени, чтобы обеспечить плавную замену комплектующих по мере их морального старения в рамках длительного жизненного цикла системы;
- возможность повторного использования аппаратных и программных компонентов в сочетании с поддержанием корректного инструментария.

Противоречивый характер перечисленных факторов эффективно может быть учтен при проектировании безопасных систем только в рамках иерархии контрольных и блокирующих механизмов.

Перечисленные подходы к проектированию позволили обеспечить высокие функциональные и эксплуатационные характеристики систем управления движением поездов на основе комплекса технических средств "Тракт" в сочетании с требуемым уровнем безопасности и надежности.

2.1.3.2 Организация и принцип функционирования комплекса технических средств «ТРАКТ-ЛП»

Комплекс технических средств "ТРАКТ-ЛП" (в дальнейшем – КТС ТРАКТ-ЛП) входит в состав системы "ТРАКТ". КТС ТРАКТ-ЛП предназначен для обеспечения гарантированного функционирования аппаратно-программных средств системы диспетчерской централизации "ТРАКТ" на линейных пунктах железных дорог в условиях воздействия неблагоприятных внешних климатических и техногенных факторов, в том числе грозových разрядов, электромагнитных полей, аварий электропитания, при решении задач безопасного управления и контроля [7, 8].

КТС ТРАКТ-ЛП (рис. 2.10) позволяет строить распределенные иерархические системы железнодорожной автоматики в составе центрального поста (ЦП), линейных постов (ЛП) и коммуникационной системы.

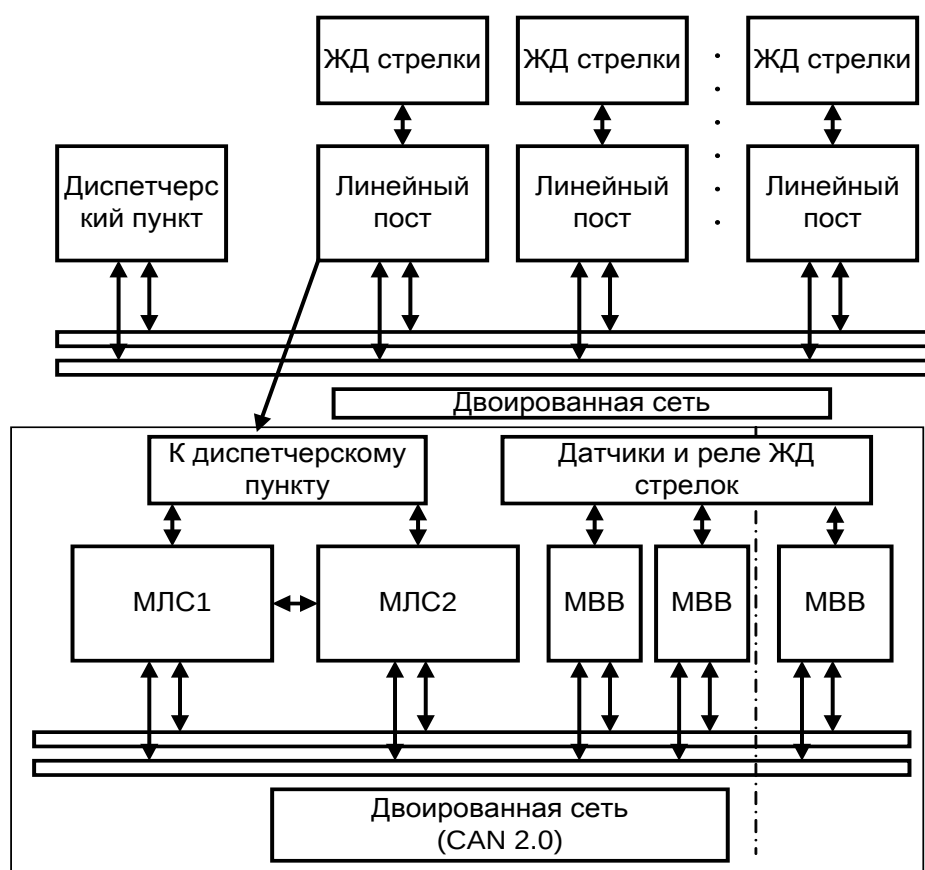


Рис. 2.10. Система железнодорожной автоматики "КТС ТРАКТ"

КТС ТРАКТ-ЛП взаимодействует с ЦП посредством двоированной коммуникационной системы с радиальной, цепочечной или кольцевой топологией на основе выделенных каналов связи (физические линии или каналы с аппаратурой уплотнения). Предусмотрена возможность использования линий связи и протоколов, эксплуатирующихся в настоящее время других систем железнодорожной автоматики.

КТС ТРАКТ-ЛП можно условно разделить на пять систем, каждая из которых включает в свой состав аппаратный и программный компонент:

- центральный вычислитель (МАСТЕР);
- периферийные модули ввода-вывода;
- контроллерная локальная сеть;
- внешняя коммуникационная система;
- система электропитания.

Дополнительно можно выделить сервисные средства для конфигурирования, обслуживания и ремонта КТС ТРАКТ-ЛП, интегрированные в аппаратуру и программное обеспечение модулей ЛП.

Рассмотрим общие принципы организации КТС ТРАКТ-ЛП, его центрального вычислителя (модуля МАСТЕР версии МЗМ) [11] и модулей ввода-вывода.

2.1.4 Особенности организации КТС «Тракт» систем железнодорожной автоматики

2.1.4.1 Общие принципы организации КТС ТРАКТ-ЛП

Аппаратное и программное обеспечение ЛП строится по модульному принципу (рис. 2.11). Используется архитектура многомашинных микропроцессорных систем с несколькими уровнями обеспечения безопасности функционирования, с резервированием и возможностью перераспределения функций, с развитыми средствами диагностики. Используются прогрессивные конструктивы, высоконадежная система энергопитания, развитые средства индикации состояния системы, возможно ведение протокола функционирования. Особое внимание уделено технологичности комплексирования КТС ТРАКТ-ЛП «под задачу» как в части аппаратных средств, так и в программном обеспечении.

Вычислительная система представляет собой локальную сеть контроллеров с двоированной магистралью в стандарте CAN-2.0. Основной и резервный комплекты модулей имеют выход на обе шины сети. Двоирование сети осуществляется на верхних уровнях протокола, что значительно расширяет зону контроля функционирования системы и позволяет при необходимости перераспределять вычислительные ресурсы (например, при частичном снижении скорости передачи в результате обрыва или замыкания одного из проводов магистрали). В состав каждого комплекта входит ведущий вычислительный модуль (МАСТЕР) и определенное количество периферийных модулей ввода и вывода, определяемое числом точек контроля и управления конкретного объекта. Предусмотрена возможность расширения ресурсов ЛП за счет подключения расширителя ввода-вывода в рамках базовой двоированной локальной сети. Модули МАСТЕРА основного и резервного комплектов имеют

дополнительный интерфейс для передачи полномочий в сети в случае различного рода отказов.

Основу элементной базы вычислительной системы составляют две платформы: процессор для встроенных применений Intel386EX и контроллер для встроенных применений AM186CC. Ядро модулей строится с использованием высоко интегрированных схем программируемой логики FLASH-памяти и быстродействующей статической памяти. Элементная база имеет встроенные средства контроля, защитные средства системного уровня, обладает свойствами повышенной помехоустойчивости, минимальным энергопотреблением.

Основу элементной базы вычислительной системы составляют две платформы: процессор для встроенных применений Intel386EX и контроллер для встроенных применений AM186CC. Ядро модулей строится с использованием высоко интегрированных схем программируемой логики FLASH-памяти и быстродействующей статической памяти. Элементная база имеет встроенные средства контроля, защитные средства системного уровня, обладает свойствами повышенной помехоустойчивости, минимальным энергопотреблением.

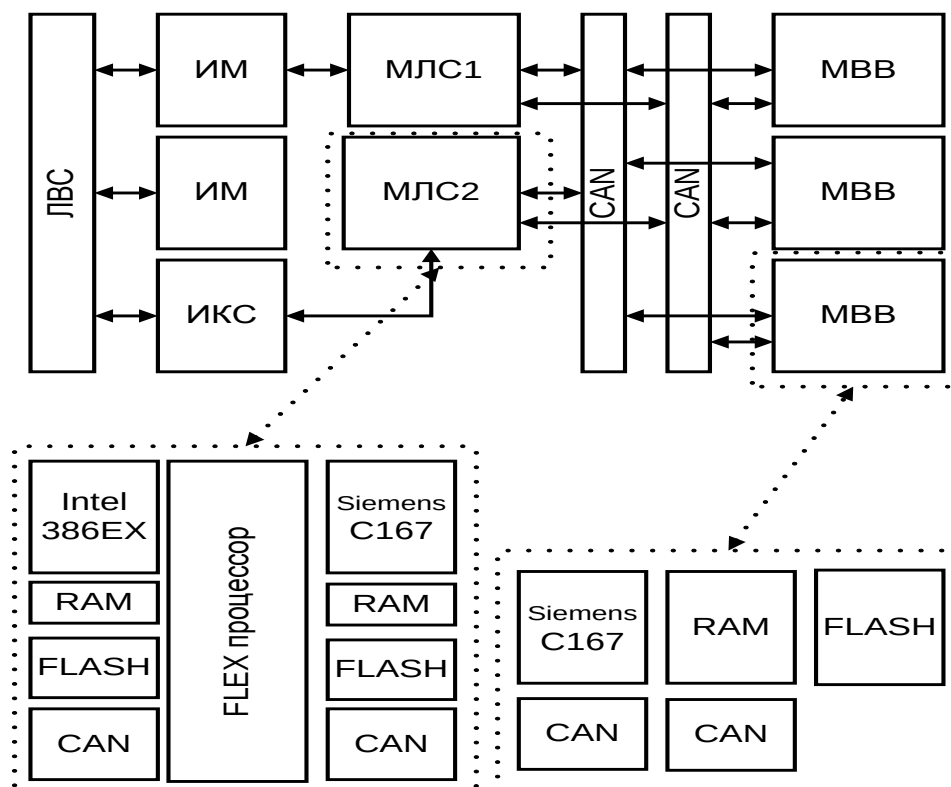


Рис. 2.11. Модули МЛС (МАСТЕР) и МВВ

Программное обеспечение КТС ТРАКТ-ЛП построено по модульному принципу на основе стандартных сертифицированных протоколов обмена и

программных компонент реального времени. На критических участках используется n-версионное программирование.

2.1.4.2 Центральный вычислитель КТС ТРАКТ-ЛП

Ведущий процессорный модуль – МАСТЕР, представляет собой высокопроизводительное вычислительное ядро, обладающее высокой безопасностью и надежностью функционирования. МАСТЕР координирует работу модулей ввода и вывода, осуществляет связь с оборудованием ЦП посредством коммуникационной системы, проверяет корректность поступающей информации, обеспечивает прохождение ответственных команд. МАСТЕР также выполняет функции конфигулятора системы, диагностирование на уровне модулей ЛС, инструментальные функции.

МАСТЕР имеет многопроцессорную несимметричную структуру, включающую двоированный процессор Intel 386 EX (процессор для встроенных применений статического типа с интегрированной периферией и средствами контроля и диагностики), оперативную память с контролем и исправлением ошибок на микросхемах FLASH и SRAM, FLEX-логику контроля и диагностики, сервисный процессор C167CR / AM186CC, коммуникационные контроллеры штатного и инструментального режимов, традиционные средства поддержки реального масштаба времени, локальную инженерную консоль.

Контроль и обработка сбоев, неисправностей, программных коллизий и их частичное исправление обеспечиваются на четырех уровнях архитектуры МАСТЕРА:

- использование встроенных средств элементной базы;
- схемотехника модуля (двоирование процессора и магистралей ЛС, применение в качестве схем контроля кодовых векторов на шинах основных процессоров «самопроверяющихся тестеров N/2» на основе безопасных комбинационных схем с несимметричными отказами, использование двоированных схем блокировки выдачи ответственных команд в случае сбоев в работе ядра системы, сигнатурный анализ критических участков программы в реальном времени, использование специальных схем управления памятью, локальное резервное питание памяти, три уровня контроля питания на плате, схема контроля генератора и др.);
- в случае обнаружения ошибки в работе ядра МАСТЕРА останов основных процессоров вычислительной системы и многоступенчатая блокировка выдачи ответственных команд аппаратным микропроцессорным супервизором с последующей обработкой информации с контрольных схем сервисным процессором, запуск аппаратно-программных процедур диагностики ядра модуля;
- специальная технология программирования, использующая алгоритмически несимметричное дублирование функций и традиционные механизмы повышения достоверности функционирования.

Средства программируемой логики и сервисный процессор в составе МАСТЕРА позволяют реализовывать операционные, тестирующие, диагностирующие блоки различной сложности, обеспечивая требуемую глубину диагностики (в сочетании с программными средствами), перераспределение ресурсов, изменение правил функционирования интерфейсов и ряд других свойств по требованию заказчика.

Модуль МАСТЕРА полностью программно конфигурируем как на этапе производства, так и в процессе эксплуатации (предусмотрены блокировки от несанкционированного доступа), что позволило исключить предварительное программирование микросхем памяти и логики, применение разъемных соединений, значительно повысило надежность и технологичность изделия.

В состав интерфейсов МАСТЕРА помимо двух CAN-магистралей и двух модемных каналов входят интерфейс координации работы МАСТЕРА 1 и МАСТЕРА 2 в ЛП, интерфейс расширения РС-104 (обеспечивает совместимость с оборудованием ПК), тестовый интерфейс в стандарте IEEE 1149.1 (JTAG), инструментальные интерфейсы основного и сервисного процессоров RS232C, локальная инженерная консоль с двухстрочным ЖКИ и функциональной мини-клавиатурой.

2.1.4.3 Периферийный модуль КТС ТРАКТ-ЛП

В состав КТС ТРАКТ-ЛП входят периферийные модули трех типов:

- ввода на 32 контролируемых объекта (256 объектов при матричном включении);
- вывода на 32 объекта управления (с повышенной надежностью);
- вывода для ответственных объектов управления (с повышенной безопасностью) на 16 выходов.

Во всех периферийных модулях используется вычислительная платформа C167CR со встроенной FLASH-памятью. Модули имеют выход на обе CAN-магистралы для каждого из процессоров. Модули оснащаются конфигурационным ППЗУ и часами реального времени (по согласованию с Заказчиком).

Модуль вывода построен по двоированной схеме с перекрестными шинами управления исполнительными устройствами на уровне выходных регистровых схем (на основе FLEX-логики) и специальной схемой выходного каскада.

Модуль вывода ответственных команд построен в двоированном варианте с регистровой и контролирующей схемой на основе FLEX-логики. Выходной каскад выполнен по схеме с повышенной безопасностью. Отказ любого элемента каждого из вычислительных каналов приводит к блокировке выходного каскада. Включение выходного каскада возможно только при условии полной работоспособности обоих вычислительных трактов модуля.

Модуль ввода построен по однопроцессорной схеме. Процессор обслуживает гальванически развязанные входы в линейном или матричном варианте. Предусмотрены схемы контроля работоспособности входных каскадов и схемы их защиты от перенапряжения.

2.1.4.4 Организация программного обеспечения КТС ТРАКТ-ЛП

Организация программного обеспечения МАСТЕРА

Программное обеспечение МАСТЕРА делится на четыре части для каждого из процессоров:

- технологическое ПО;
- инструментальное ПО;
- системное ПО;
- прикладное ПО.

Технологическое ПО предназначено для обеспечения загрузки программ в МАСТЕР, тестирования и отладки на этапе производства, а также во время проведения пуско-наладочных и ремонтных работ. В состав технологического ПО МАСТЕРА входят следующие компоненты:

- набор тестовых прошивок для FLEX логики;
- набор рабочих прошивок для FLEX логики;
- технологический монитор для инструментальной ЭВМ;
- набор тестовых прошивок для FLASH 386EX и AM186CC;
- набор рабочих прошивок для FLASH 386EX и AM186CC;
- программатор JTAG.

Инструментальная ЭВМ и технологический монитор используется на этапе пуско-наладки. На этапе тестирования и ремонта можно использовать как инженерную консоль, так и инструментальный ПК.

Инструментальное ПО предназначено для разработки и корректировки прикладного и системного ПО МАСТЕРА. В состав инструментального ПО входят следующие компоненты:

- инструментальный монитор для инструментальной ЭВМ;
- инструментальный монитор для МАСТЕРА (отдельный для каждого процессора);
- компилятор;
- программатор JTAG.

Системное ПО предназначено для обеспечения безопасного и надежного функционирования прикладной программы. В состав системного ПО входят следующие компоненты:

- система контроля загрузки FLEX логики;
- система контроля правильности массивов во FLASH памяти;
- система рестарта при сбоях;
- система восстановления программы после сбоя;
- система запоминания состояния системы при сбое;
- система контроля питания;
- система контроля чтения/записи в ОЗУ МАСТЕРА;
- система контроля работоспособности 386EX;
- драйвер CAN;
- драйверы синхронного и асинхронного каналов;
- драйверы RTC;
- драйвер инженерной консоли;
- система ON-LINE диагностики и ведения протокола во FLASH памяти.

Прикладное ПО состоит из модулей связи с верхним уровнем, модулей связи с периферийными контроллерами и модуля анализа команд верхнего уровня.

Операционная система МАСТЕРА для основного процессора Intel 386EX

Основу программного обеспечения мастера составляет ОС QNX 4.23 фирмы QNX Software. Эта система является распределенным, отказоустойчивым микроядром, сделанным по спецификации POSIX. Система сертифицирована на соответствие стандарту POSIX. Минимальный размер микроядра около 10-ти Кбайт. Система легко ставится в ПЗУ. Позволяет организовывать сети, обладающие механизмами маршрутизации, с разными протоколами обмена и на базе различных физических каналов передачи

Микроядро QNX осуществляет четыре основные функции: обмен между процессами, низкоуровневые сетевые коммуникации, переключение процессов и диспетчеризация прерываний. Имеется 14 основных системных вызовов для обращения к вышеперечисленным функциям. Эти функции очень компактны в реализации и занимают всего 7 Кб кода. Микроядро обеспечивает необходимый набор средств для нормальной работы ОС реального времени.

Примитивы для планирования процессов соответствуют стандарту POSIX 1003.4 (реальное время). QNX обеспечивает три типа планирования процессов: карусель, FIFO и адаптивное планирование.

Применение операционной системы на базе стандарта POSIX позволяет повысить общую надежность мастера за счет стандартного API с хорошо продуманным набором системных вызовов.

Для сервисного процессора AM186CC разработана операционная система PnPDOS-3.3.

Инструментальное программное обеспечение МАСТЕРА

Система QNX является так называемой self-hosted системой, т.е. в состав системы входит инструментарий для разработки. Для программирования основного процессора Intel 386EX МАСТЕРА используется компилятор WATCOM 10.6, считающийся самым мощным и надежным профессиональным компилятором для процессоров Intel 80x86.

Для программирования сервисного процессора AM186CC МАСТЕРА используется инструментальная утилита ColdLoader 2.10a.

Программное обеспечение модулей ввода-вывода, построенных на базе C167CR, по структуре аналогично ПО сервисного процессора МАСТЕРА.

2.1.4.5 Организация модуля МАСТЕР версии МЗМ

Структура серийно выпускаемого варианта МАСТЕР МЗМ представлена на рис. 2.12. Рассмотрим более подробно состав вычислительных механизмов, реализованных средствами ПЛИС в данном вычислителе.

На рис. 2.14 представлена метамодель спроектированной системы, на рис. 2.15 представлена диаграмма реализации. Если рассматривать прошивку в общем, она представляет собой структурированный набор механизмов. При этом механизмы могут быть как служебными, так и целевыми. Последние в свою очередь подразделяются на механизмы 186, механизмы 386 и общие механизмы.

Целевые механизмы непосредственно реализуют требования к системе. Служебные механизмы предназначены для поддержки и обеспечения работы целевых механизмов и непосредственным образом не обеспечивают целевой функциональности, однако выполняют это косвенно.

На основе метамодели можно определить и классифицировать базовые механизмы, реализованные в ПЛИС (рис. 2.16 и 2.17):

- Механизмы 186
 - Коммутатор последовательных каналов (Шифр механизма – SCC).
 - Система прерываний Am186CC (4 линии запроса прерывания) (Шифр механизма – i10).
- Механизмы 386
 - Система прерываний 386EX (5 линий запроса прерывания) (Шифр механизма – i100).

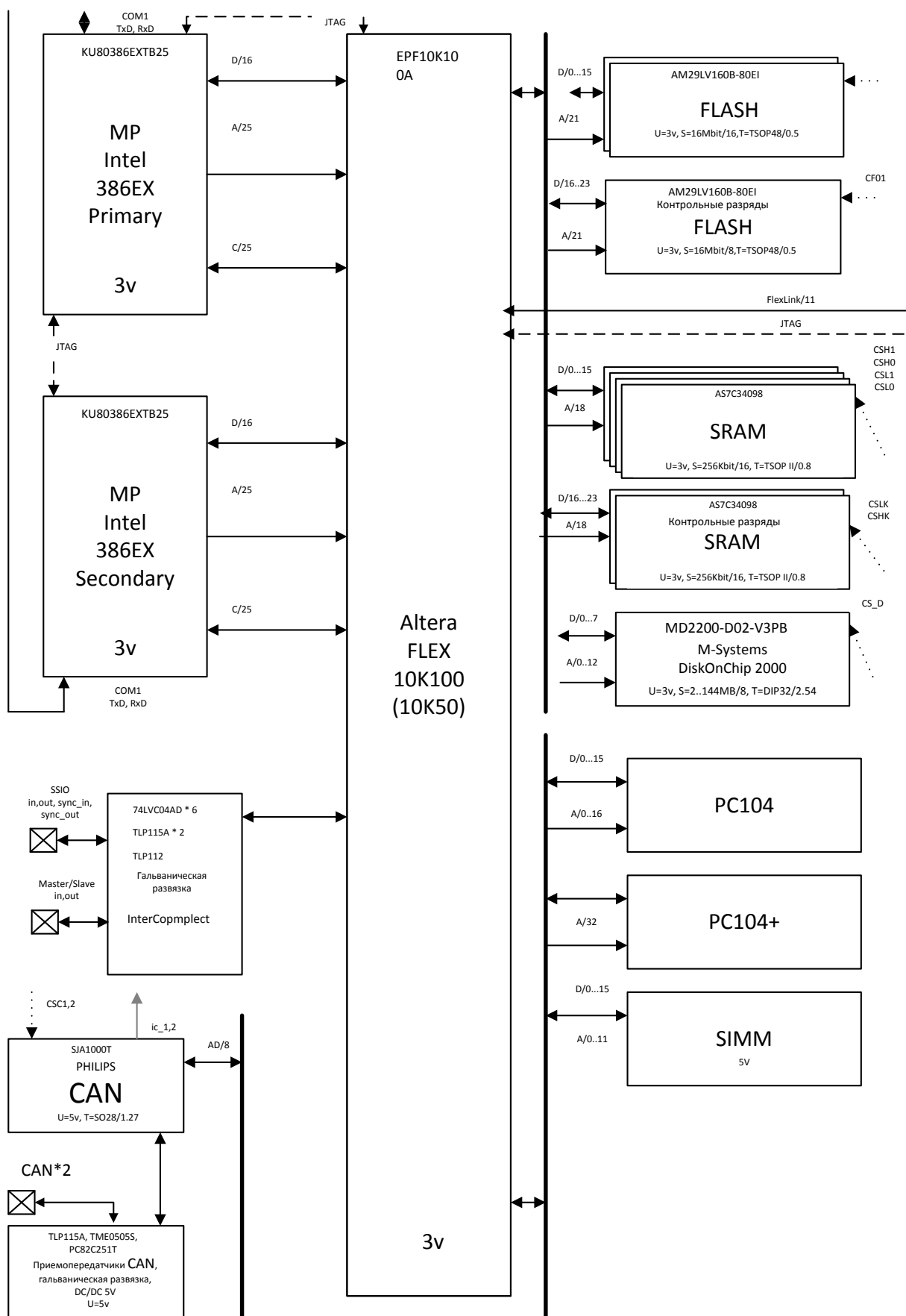
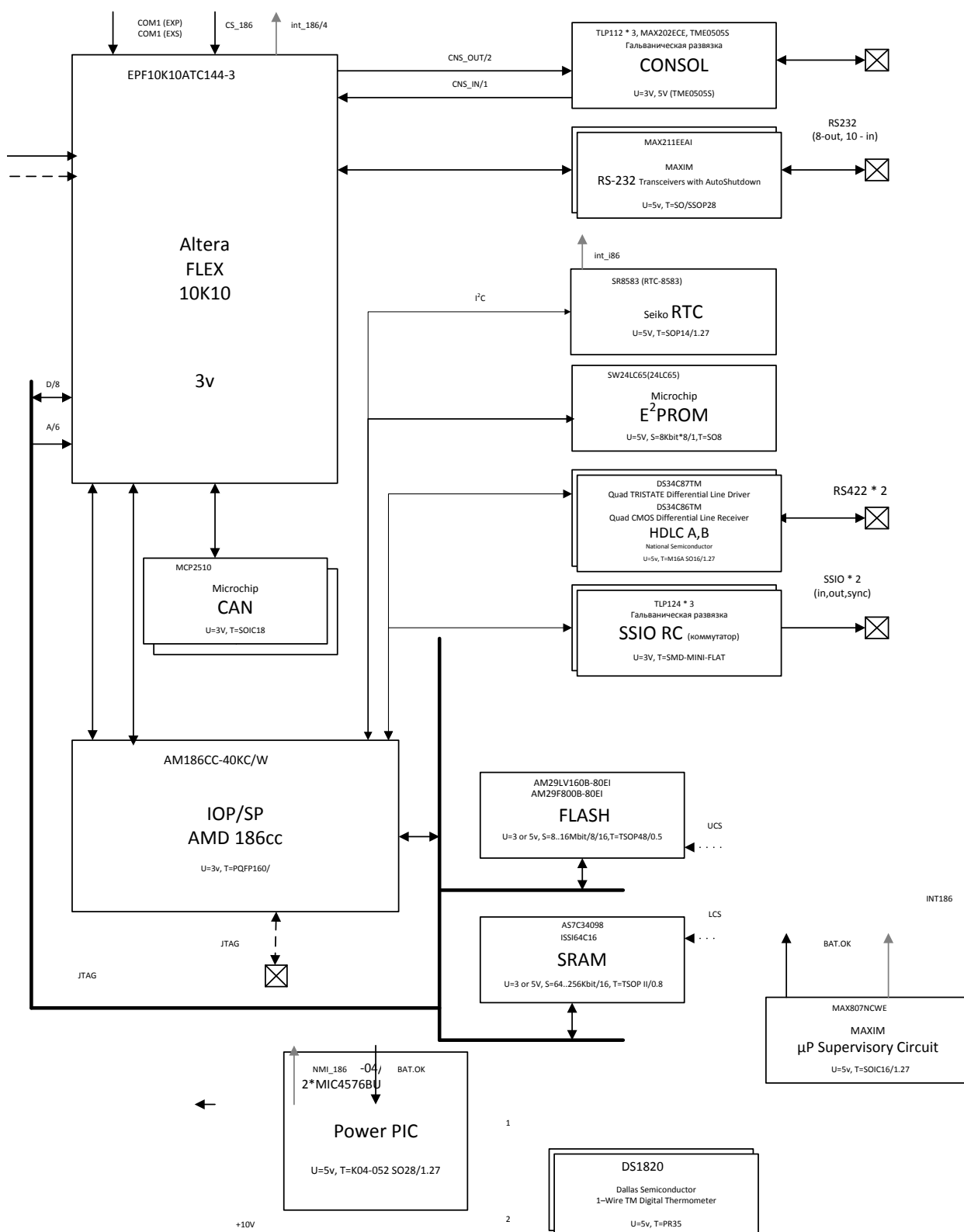


Рис. 2.12а Структурная схема модуля МАСТЕР версии МЗМ (левая часть)



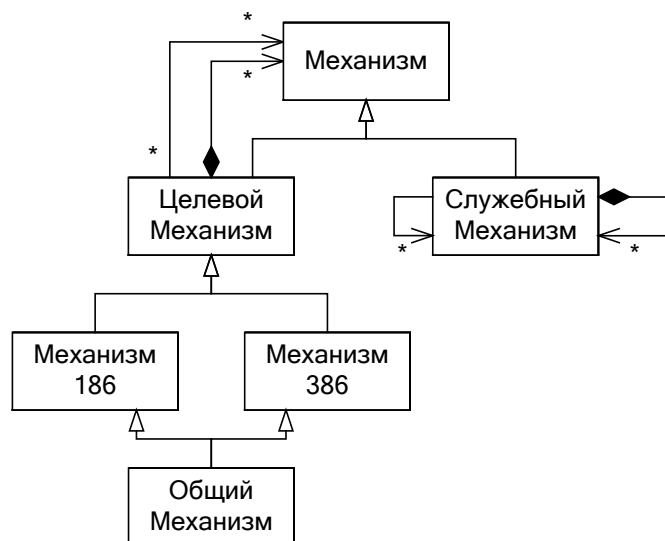


Рис. 2.14. Мета модель МАСТЕРА М3М

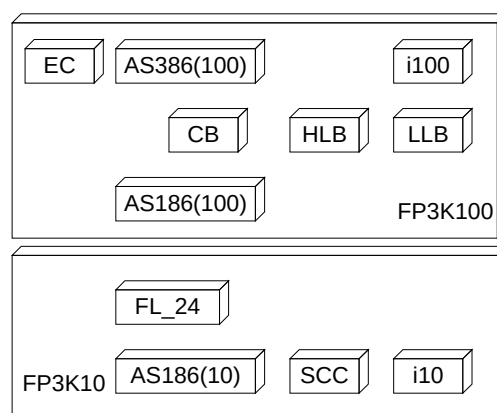


Рис. 2.15. Диаграмма реализации МАСТЕРА М3М

- Общие механизмы
 - Блок констант (Шифр механизма – CB).
 - Канал обмена сети верхнего уровня (Шифр механизма – HLB).
 - Канал обмена общего назначения (Шифр механизма – LLB).
- Служебные механизмы
 - Адресный селектор 186 (Шифр механизма – AS186).
 - Адресный селектор 386 (Шифр механизма – AS386).
 - FlexLink (Шифр механизма – FL_24).
 - Блок управления EX (Шифр механизма – EC).

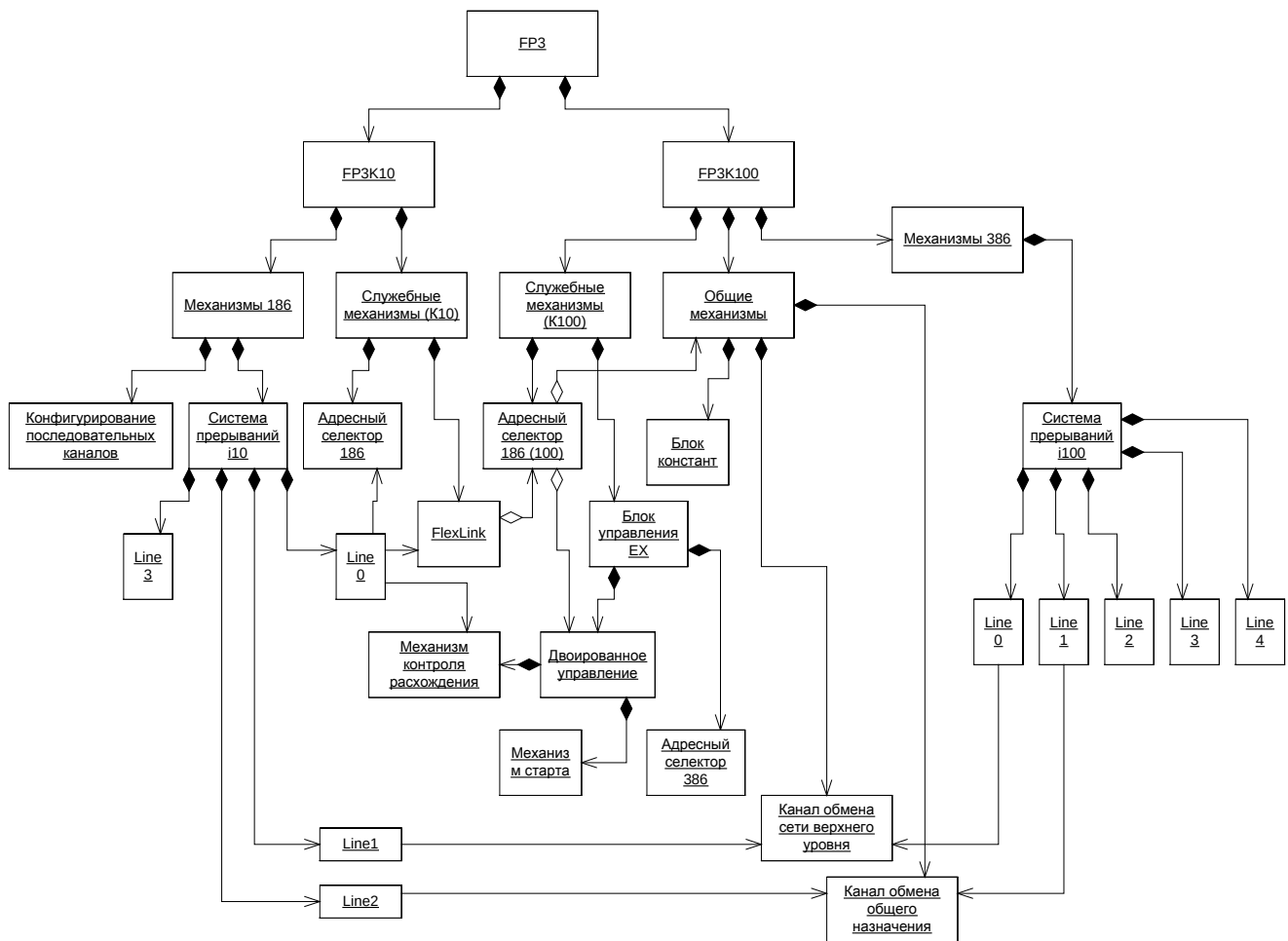


Рис. 2.16. Диаграмма классов механизмов FLEX-логики (ПЛИС) МАСТЕРА МЗМ

2.2 Принцип актуализации вычислительного процесса в проектировании ВсС

2.2.1 Модель актуализации вычислительного процесса

Первый документ, определяющий целевую задачу любой ВсС — это техническое задание (ТЗ). Для ВсС постановщиком ТЗ является прикладной специалист. ТЗ на разработку вычислительных систем всегда является неполным, поэтому в их разработке присутствует этап проектирования. ТЗ содержит информацию о том, «что» должна выполнять система (часто неформально) и «как» она должна это делать. Неполнота ТЗ заключается в том, что даются не все указания «как», а указания «что» позволяют двоякую трактовку и реализацию. Потенциально разработчики свободны в способе реализации ВсС, лишь бы она выполняла требуемые от нее функции. Указания «как» являются ограничением этой свободы: элементная база, минимальная производительность, инструментальные средства, энергопотребление, стоимость, надежность и др. Задача этапа проектирования состоит в добавлении недостающих указаний «как» и в выборе конкретных «что».

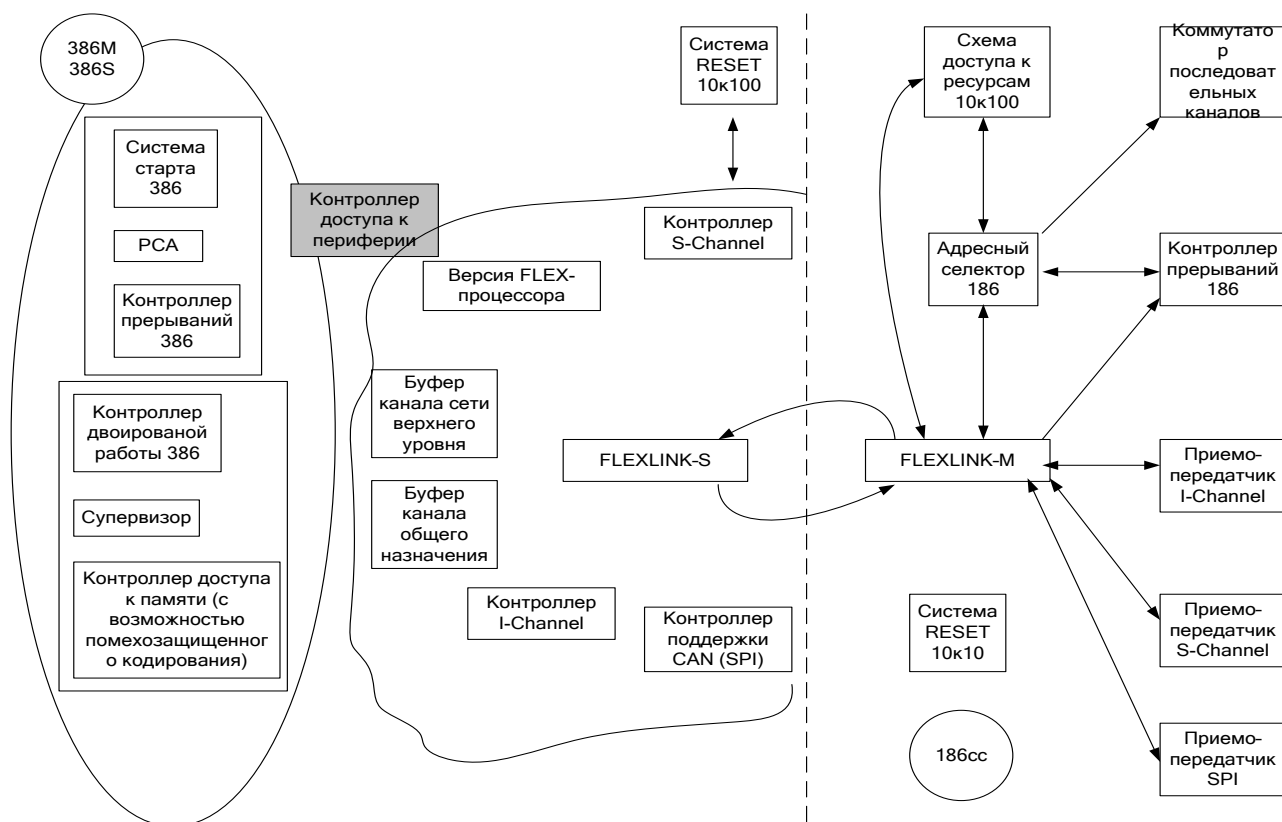


Рис. 2.17. Обобщенный набор механизмов FLEX-логики (ПЛИС) МАСТЕРА МЗМ

ТЗ, дополненное всеми недостающими указаниями, назовем *целевым алгоритмом* (ЦА). Дополнить ТЗ можно по-разному, поэтому ЦА, реализующих ТЗ, может быть несколько. Задача разработчиков ВСС заключается в реализации конкретного ЦА вычислительными средствами ВСС. Для реализации ЦА требуется применение промежуточных языков. В процессе передачи «желания» человека машине ЦА претерпевает ряд преобразований из одного языка в другой, часть из которых выполняют разработчики ВСС, а часть выполняется автоматизировано инструментальными средствами и самой ВСС. Очевидно, что этот процесс не является полностью автоматизированным, поскольку инструментальные машины, с помощью которых выполняются преобразования ЦА, тоже не понимают человеческого языка. Реализация ЦА заключается в выполнении функций ТЗ с учетом его ограничений. Весь процесс преобразований «запускается» не для создания ВСС, не для создания программы, а для того, что хочет получить человек. ЦА претерпевает множество изменений, пока не преобразуется в требуемые физические сигналы: свет, движение, электричество и др. Назовем процесс преобразования ЦА в «желаемое» человеком *процессом актуализации*.

В качестве модели процесса актуализации ЦА будем использовать ориентированный граф (*граф актуализации*). Процесс состоит из разветвленной системы преобразований информации, начинающейся с ЦА и заканчивающейся конечными физическими сигналами. Вершинами графа являются *трансляторы*, преобразующие информацию из одной формы представления в другую. Дуги

графа — это формы представления ЦА. Таким образом, последовательность трансляторов, соединенных дугами, задает последовательность преобразований ЦА. Поскольку действия разработчиков ВcС не являются формальными, мы их исключим из графа актуализации. Исходным представлением информации будем считать *целевую программу* (ЦП) — ЦА, записанный на каком-нибудь языке (программирования), для которого есть инструментальные средства. Сузив граф актуализации, мы оставили в нем лишь автоматические трансляторы, что позволит рассмотреть его формально и получить численные характеристики. На рис. 2.18 представлен пример типичного графа актуализации для ВcС с программным управлением.

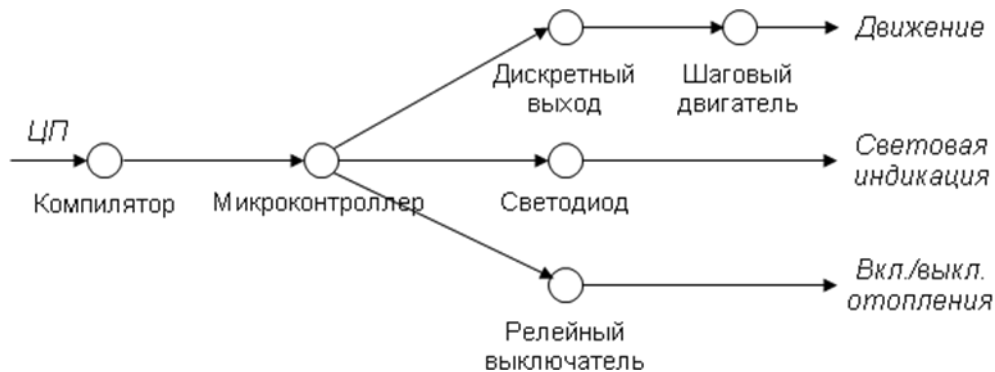


Рис. 2.18. Пример графа актуализации

Граф актуализации можно использовать для решения двух задач:

- Создание графа актуализации и ВcС по нему. Это является одним из способов проектирования ВcС.
- Создание графа по существующей ВcС. На основе графа можно получить характеристики для повторного использования трансляторов, модернизации и рефакторинга ВcС.

Предлагаемая модель не является одной из моделей жизненного цикла вычислительных систем. Они моделируют этапы существования систем, а процесс актуализации соответствует всему процессу преобразования информации и является статическим. Трансляторы, участвующие в этом процессе, могут показаться принципиально разными, но это обусловлено несовершенством вычислительных возможностей современной элементной базы.

2.2.1.1 Трансляторы в модели актуализации вычислительного процесса

Понятие *транслятор* базируется на TSM-модели [70]. У транслятора есть входные и выходные потоки: по входным он получает команды (сигналы), а по выходным генерирует реакции на команды. Транслятор будем называть *программируемым*, если он поддерживает входные сигналы, приводящие к появлению выходных сигналов (реакций). Транслятор будем называть *конфигурируемым*, если он поддерживает входные сигналы, приводящие к смене реакций на программирующие команды. Потоки, по которым передаются

команды, приводящие к реакциям или к сменам реакций, будем называть программирующими (*программами*) и конфигурирующими (*конфигурациями*). Любой поток состоит из команд одного типа. На графе актуализации программы будем обозначать сплошными дугами, а конфигурации — пунктирными. Трансляторы будем обозначать кругами. Для актуализации целевой программы (ЦП) ВсС некоторым трансляторам даются программирующие команды, чтобы они выполнили действия. Некоторые из этих трансляторов не способны самостоятельно актуализировать полученные команды. Они разбивают команды на части и передают их для актуализации другим трансляторам. Так команды передаются от транслятора к транслятору до тех пор, пока не дойдут до тех трансляторов, которые могут сами актуализировать полученные команды. Их суммарные реакции как раз и составляют действия ВсС, которые она должна выполнять в соответствии с ЦП. Потоки являются формами представления частей ЦП, а трансляторы — объектами, преобразующими эти формы представления, разделяющими и соединяющими их части. Трансляторы преобразуют программы и конфигурации в новые программы и конфигурации.

Если один транслятор программируется или конфигурируется другим транслятором, то он является его *ресурсом*. Отношение между трансляторами можно определять не через необходимость актуализации их команд. Допустим, есть транслятор А с ресурсом Б и нужно получить реакцию ресурса. Непосредственно запрограммировать ресурс Б невозможно, потому что к нему нет доступа. Но его можно запрограммировать через транслятор А. Даже, если известны языки входных потоков ресурса Б, ему можно давать не любые команды, а только те, которые для него генерирует транслятор А, обрабатывая собственные входные команды. Аналогичным образом можно передавать команды через цепочку трансляторов. Все трансляторы, к которым можно получить доступ через команды транслятора, являются его непосредственными или опосредованными ресурсами.

Часто ВсС получают информацию из своего окружения: температура воздуха, напряжение во входных электрических цепях, действия оператора и др. С одной стороны, эти сигналы можно поделить на программирующие и конфигурирующие. С другой стороны, потоки, по которым передаются эти сигналы, не являются формой представления ЦП. Получается, что у некоторых трансляторов не все входные потоки являются формой представления ЦП. Назовем такие трансляторы *смешанными*, а такие потоки *внешними*. Транслятор, у которого присутствуют только внешние потоки, назовем *0-транслятором*. Не следует путать, например, кнопку или датчик с 0-транслятором. В графе 0-транслятор не является ничьим ресурсом, но сам программирует и конфигурирует другие трансляторы. Смешанные трансляторы могут по внешним входам получать некоторую информацию извне, а по другим — команды на передачу этой информации по графу актуализации. Сама ВсС обычно является смешанным транслятором.

Любой транслятор является программируемым (0-программируемым). Если транслятор конфигурируемый, то он одновременно является и программируемым, и конфигурируемым. Конфигурируемый транслятор может менять реакции, но без подачи программирующей команды эти реакции не будут генерироваться.

Если в актуализации участвует большое число трансляторов, то можно вводить иерархию, выделяя подграфы. Граф актуализации критически зависит от иерархических уровней. Информация графа включает в себя трансляторы, потоки данных и все их команды. При введении иерархии меняется состав трансляторов и, что важно, состав их команд. Фактически команда останется прежней, но ее влияние на новый транслятор будет другим: команда может поменять тип. Начиная с самого верхнего представления ВcС, как одного транслятора, можно опускаться на более детальные уровни, на которых будут появляться новые трансляторы и команды. При этом будут детализироваться не только трансляторы, но и их ресурсы. Так можно опускаться вплоть до логических элементов и электрических сигналов на их входных и выходных линиях.

Модель позволяет неограниченно детализировать граф актуализации. При этом большинству разработчиков не нужен уровень детализации одного порядка для разных фрагментов графа: разработчики инструментального программного обеспечения хорошо представляют фазу разработки, разработчики встроенного программного обеспечения — подграф программных трансляторов фазы исполнения. Чтобы иметь одинаковое (по уровню) представление обо всех частях графа актуализации, необходимо участвовать в разработке всех его трансляторов. Для современных ВcС это большая редкость, поэтому на практике граф актуализации представляется фрагментарно и отображает лишь то, что важно для разработчика, лишь то, на что он может повлиять. Формально создание полного графа актуализации затруднительно, поскольку в граф входят трансляторы, разработанные сторонними фирмами-разработчиками: электронные компоненты, библиотеки функций и др. Даже если известны их входные и выходные потоки, то внутренние связи являются закрытой информацией.

2.2.2 Фазы актуализации вычислительного процесса ВcС

Разделим процесс актуализации на 2 фазы и пометим вершины графа (трансляторы) в соответствии с принадлежностью к одной из фаз:

- фаза разработки (Design-Time, DT);
- фаза исполнения (Run-Time, RT).

Обе фазы участвуют в процессе актуализации ЦП, но по-разному. Фаза разработки совпадает с этапом разработки ВcС. Все формы представления ЦП на этой фазе предназначены для подготовки к его исполнению. Ни одна из форм представления на фазе разработки непригодна для ее исполнения ВcС, как

транслятором. Трансляторы фазы исполнения функционируют во время работы ВсС, при каждом ее запуске. Основное отличие трансляторов этих двух фаз заключается во времени их работы. Работа RT-трансляторов — это работа системы. Работа DT-трансляторов происходит вне ВсС и до ее готовности. Очевидно, что в графе актуализации ЦП сначала попадает в фазу разработки, а затем в фазу исполнения.

DT-трансляторы преобразуют целевой алгоритм в форму, которая исполняется ВсС. Уже давно используется технология, по которой эта форма получается один раз, а исполняется много раз. Чтобы каждый раз не проделывать работу фазы разработки нужны энергонезависимые хранилища информации. Без этих хранилищ полученную форму ЦП невозможно было бы сохранить для повторного использования и при каждом старте нужно было бы программировать ВсС заново. Для большинства окружающих нас вычислительных систем это считается очевидной нормой, но раньше это было не так и сейчас для некоторых видов ВсС это не так (Evaluation Boards).

Разделение процесса актуализации на фазы позволяет дать новую трактовку широко распространенными в области разработки ВсС определениям. Одни трансляторы актуализируют целевой алгоритм внутри одной фазы, а другие — переводят между фазами. Транслятор, переводящий потоки в фазу исполнения, называется *компилятором*. Т.е. компилятор — это тот транслятор, чьи выходные потоки могут быть актуализированы ВсС. Связанный подграф, состоящий из DT-трансляторов и заканчивающийся компилятором, называется *инструментальной цепочкой*. У ВсС, если она, например, многопроцессорная система, может быть несколько инструментальных цепочек. Для каждого из ее процессоров может быть своя инструментальная цепочка. *Программное обеспечение ВсС* (ПО ВсС) — это совокупность выходных потоков всех компиляторов, т.е. все потоки, приходящие в фазу исполнения. Традиционно под программным обеспечением понимается несколько другое, то, что создают программисты. Программисты, как трансляторы, вынесены за рамки графа актуализации, поэтому генерируемая ими форма представления ЦА, не попадает непосредственно в фазу исполнения.

2.2.2.1 Организация фазы исполнения ВсС

Каждый RT-транслятор обладает специфической архитектурой и содержит набор команд. Изучать и использовать язык каждого транслятора для актуализации ЦП неэффективно. Для повышения эффективности программирования вводятся специальные трансляторы, через которые осуществляется программирование других трансляторов. Другими словами, на фазе исполнения выделяют такие трансляторы, чтобы остальные RT-трансляторы являлись их ресурсами. Назовем такие трансляторы *базовыми (БТ)*. Выгода от использования БТ очевидна: программирование их ресурсов выполняется только с помощью языков БТ, а их в системе гораздо меньше, чем ресурсов. Наибольшее распространение получили следующие БТ:

микропроцессоры, микроконтроллеры и программируемые логические интегральные схемы (ПЛИС).

Рассмотрим задачу программирования произвольного транслятора Т. Это можно сделать через его входной поток (2.19, а, поток А). Если транслятор Т доступен, как ресурс БТ, то его можно запрограммировать с помощью команд БТ (2.19, б, потоки Б-В). Формируется новая задача: как запрограммировать БТ, чтобы генерируемый им поток В оказался аналогичным потоку А. Можно ввести промежуточный транслятор Д. Его задачей будет обеспечение актуализации тех функций транслятора Т, которые требуются для актуализации ЦП (2.19, в, потоки Г-Е-Ж). Такой транслятор назовем *драйвером*.

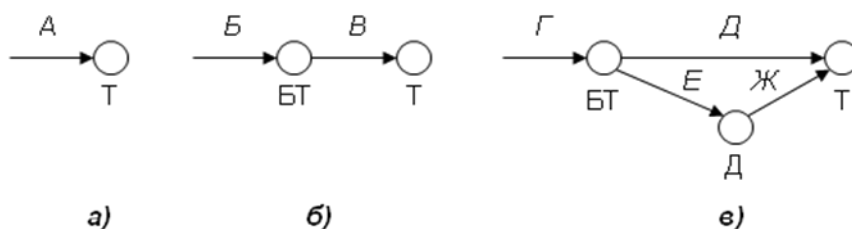


Рис. 2.19. Способы программирования трансляторов

Например, последовательный канал можно программировать через регистры специального назначения, поддерживаемые микроконтроллером, а можно через драйвер последовательного канала. Вызов функций драйвера обеспечивает микроконтроллер, поэтому драйвер является ресурсом микроконтроллера. В некоторых случаях при наличии драйвера непосредственное программирование ресурса (рис. 2.19, в, потоки Г-Д) может быть запрещено и единственным вариантом останется программирование через драйвер (рис. 2.19, в, потоки Г-Е-Ж).

Введение драйверов усложняет структуру графа актуализации, но для программирования ресурса нужно определить лишь первый программный поток. В первом случае это будет поток А, во втором — Б, а в третьем — Г. При этом поток Г через БТ может привести к программированию через поток Е и/или Д. В первом случае нужно реализовать функции для транслятора Т в его входном потоке, во втором — в потоке БТ, а в третьем случае эти функции уже реализованы драйвером и в потоке Г их нужно запустить на исполнение. Тем самым упрощается программный поток для БТ. Использование термина драйвер интуитивно более понятно для программирования микроконтроллеров и микропроцессоров. Для ПЛИС аналогом драйверов являются *мегафункции*. Упрощение входных потоков, которые нужно задать разработчикам ВсС, привело к увеличению сложности ее структуры. До тех пор, пока эта сложность не превысит простоту программирования потоков, будут пользоваться этим методом.

Таким образом, для упрощения программирования ресурсов ВсС вводится сеть драйверов, актуализирующая частные задачи, входящие в актуализацию ЦП. Это является одним из способов организации фазы исполнения графа

актуализации. Дальнейшим развитием драйверов является появление устойчивых комбинаций драйверов, предоставляющих не только доступ к ресурсам ВсС, но и управляющих вычислительным процессом. Сначала это были библиотеки функций, затем они преобразовались в *операционные системы* (ОС). Программирование для ПК достигло уже такого уровня, что число функций, которые он может выполнять, столь велико, что программисты специализируются лишь на некоторых типах драйверов: разработке баз данных, WEB-интерфейсов, пользовательских приложений, приложений клиент-сервер и др. Программированием процессора ПК, как БТ, уже никто не занимается, потому что реализация функций, которые целесообразно решать с учетом вычислительной мощности современных ПК, слишком сложна.

Для упрощения актуализации ЦП на фазу исполнения могут быть добавлены не только драйверы, но и новые БТ. У них может быть новая архитектура, новые входные команды, более предпочтительные для актуализации ЦП. Программно реализованный БТ будем называть *виртуальной машиной* (ВМ). Если доступ к ресурсам обеспечивается с помощью драйверов, то они скрывают особенности ресурсов: ресурсы могут быть разными (разные модели устройств), а функции работы с ними будут одинаковыми. Программирование получается аппаратно-независимым, но зависимым от драйверов. Программирование ВМ аналогичным образом является независимым от используемого БТ, но зависимым от самой ВМ (см. рис. 2.20).



Рис. 2.20. Добавление виртуальной машины на фазу исполнения

Ни один транслятор не функционирует мгновенно и это влияет на производительность фазы исполнения: чем больше в ней будет трансляторов, тем меньше будет производительность ВсС. Добавление драйверов и ВМ с целью организации вычислительного процесса понижает производительность ВсС. Добавление драйверов вносит незначительно понижение производительности, потому что драйверы добавляются между БТ и ресурсами: БТ программируется с первоначальной производительностью, а ресурсы уже с пониженной. Добавление ВМ вносит значительное понижение производительности, потому что исходный БТ сам становится ресурсом и его программирование замедляется, а, значит, замедляется доступ ко всем его ресурсам.

2.3 Прототип САПР на базе ОСМВ

В разделе рассматривается организация и использование прототипа САПР на базе разработанных объектно-событийных МоС в задаче создания СППП для семейства сложных аналитических приборов с использованием оригинальной

архитектуры NL3 и для ВсС с ограниченными ресурсами на платформах NL1, NL2.

2.3.1 Микроархитектура вычислителя NL3

Процессор NL3 сочетает в себе элементы суперскалярной и VLIW архитектуры, основной обрабатывающей единицей выступает DPU (Data Processing Unit) – реализация функционального блока ОСМВ. Типовой состав DPU аналитического прибора (открытый список) включает:

- блоки цифровой обработки (вычислитель поворота координат; коррелятор; КИХ/БИХ-фильтр);
- блоки общего назначения (сумматор; табличная выборка; умножитель; память; мультиплексор);
- специализированные блоки (приемопередатчики внешних устройств; осциллограф; отладчик).

Организация вычислений производится посредством статического и динамического конфигурирования композиции DPU, для чего разработан прототип САПР, включающий графическую систему подготовки нет-листов, компилятор на основе системы неравенств, библиотеки логических ФБ, Design-Time и Run-Time библиотеки драйверов DPU (раздел 2.3.4).

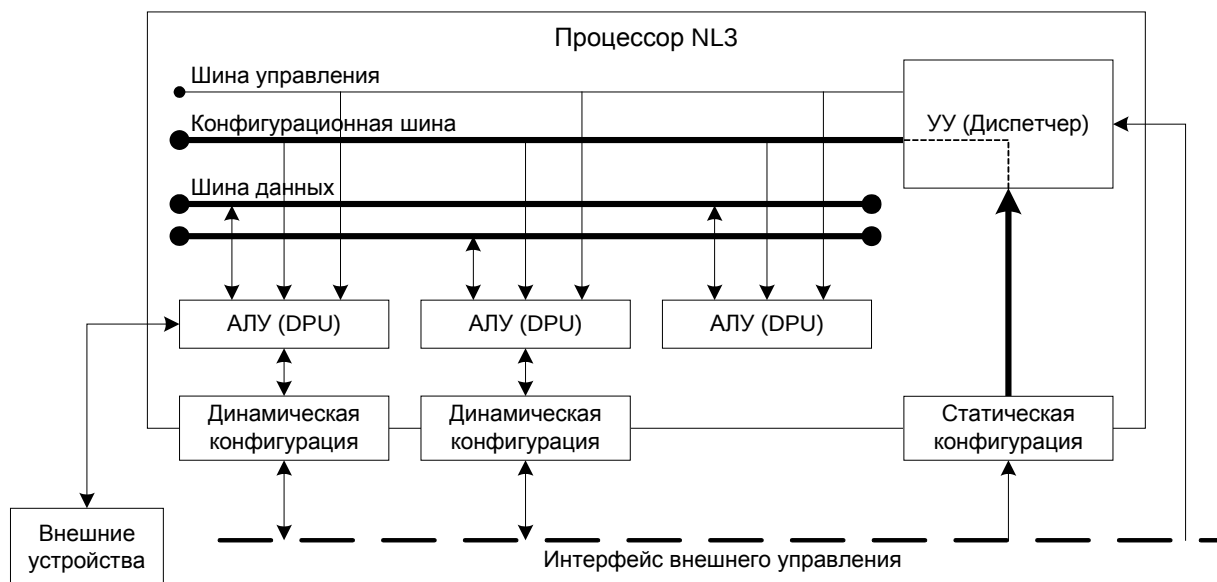


Рис. 2.21. Организация вычислителя NL3

Основу архитектуры NL3 (рис. 2.21) составляют DPU (рис. 2.22), которые выполняют следующие задачи:

- Вычисления
- Обмен данными
 - Обмен данными с шинами данных
 - Загрузка данных с конфигурационной шины

- Обмен данными с ВУ

DPU имеют порты – буферы для ввода/вывода данных.

- Направление:

- Входные
- Выходные

- Тип:

- Данных
- Конфигурационные

DPU функционируют независимо друг от друга.

Любые действия в вычислителе выполняются по команде.

- Типы команд

- Загрузка значения во входной порт
- Запуск вычислений
- Выгрузка значения выходного порта

Команды передаются по шинам: конфигурационная шина; шина управления.

Организация шины данных (рис. 2.23).

- Задача: загрузка и выгрузка данных DPU

- В один момент времени может выгружать только один DPU, а загружать – много DPU
- Разделяемый ресурс

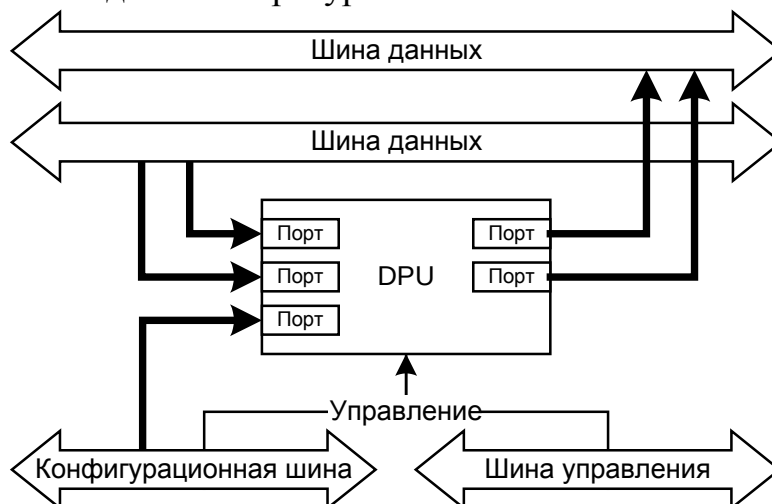


Рис. 2.22. Data Processing Unit (DPU)

- Данные кодируются как 32-битные вещественные числа с фиксированной запятой
 - Диапазон значений: [-1, +1]
- К DPU могут быть подключены не все линии шины данных
 - Выравнивание на старшие разряды

- Уменьшение разрядности подключения = понижение точности
- На неподключенных линиях шины данных удерживается ноль
- Каждый DPU подключен к одной или двум шинам данных

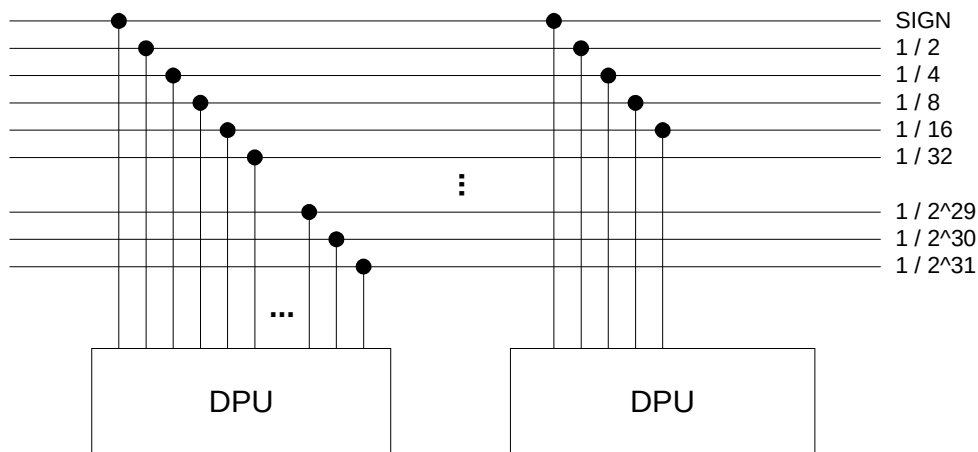


Рис. 2.23. Организация шины данных

Конфигурационная шина / шина управления (рис. 2.24).

- *Конфигурационная шина* передает индивидуальные команды и константы
 - Разрядность: 72 бита
 - К каждому DPU подключены только те линии, которые нужны для управления и передачи данных
 - Возможно совместное использование
 - Разделяемый ресурс
- *Шина управления* передает широковещательные команды
 - Разрядность: 3 бита
 - К каждому DPU подключены все линии шины
 - Типы команд:
 - Start of Time Slot (SOS)
 - End of Time Slot (EOS)
 - Hold

Важно эффективное подключение DPU к конфигурационной шине.

Диаграмма работы DPU под управлением диспетчера приведена на рис. 2.25. В диспетчере используются следующие правила работы:

- Слот – максимальная длительность обработки входных данных
 - Определяется FSR

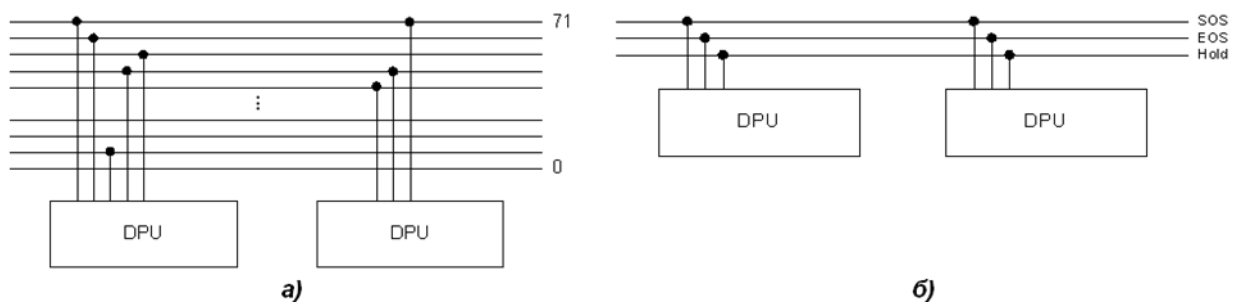


Рис. 2.24. Конфигурационная шина (а), шина управления (б)

- Статическая конфигурация (программа) – последовательность значений (инструкций), выставляемых на конфигурационную шину
- Размер = $F_{\text{диспетчер}} / FSR$
- Не меняется от слота к слоту
- Изменение требует остановки диспетчера
- Задачи диспетчера:
 - Управление конфигурационной шиной в течение слота
 - Формирование команд на шине управления в начале и в конце слота

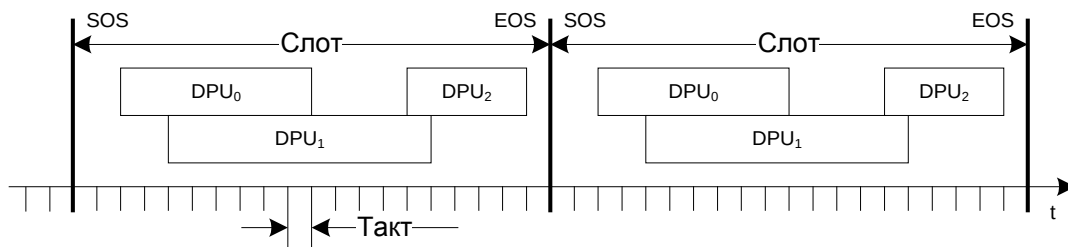


Рис. 2.25. Диаграмма работы DPU под управлением диспетчера

Система динамического конфигурирования DPU.

- Динамическая конфигурация DPU – область памяти, в которой находятся параметры работы и текущие данные DPU
 - Может быть изменена во время работы диспетчера
 - Доступна на чтение и запись для DPU и интерфейса внешнего управления
 - Снабжена механизмами синхронизации

Интерфейс внешнего управления NL3. Задачи:

- Запуск и остановка программ на NL3
 - Загрузка статической конфигурации диспетчера
 - Управление диспетчером
 - Инициализация динамических конфигураций DPU
- Управление и контроль над работой DPU
 - Запись и чтение динамических конфигураций DPU в время работы NL3
- Не является необходимым элементом архитектуры NL3

- Отсутствие = отсутствие конфигурирования (динамического управления)

Таблица 2.6. Сравнение NL3 с перспективными процессорными архитектурами

		NL3	Суперскалярная архитектура	VLIW архитектура
Исполнительные блоки (DPU)				
Количество		Много	Много	Много
Время работы	Однотипных	Разное	Одинаковое	Одинаковое
	Разнотипных	Разное	Разное	Одинаковое
Регистровые банки		Нет	Нет	Есть
Программирование				
Общая инструкция		Да	Нет	Да
Планирование		Статическое	Динамическое	Статическое
Плотность кода		Низкая	Высокая	Низкая
Перекрытие инструкций		Да	Нет	Нет

Реализация процессора NL3

В табл. 2.6 показано, как NL3 позиционируется по отношению к наиболее близким процессорным архитектурам.

Приведем основные параметры существующей реализации процессора:

- Разрядность
 - Шина данных 32 бита
 - Конфигурационная шина 72 бита
 - Шина управления 3 бита
- Частота
 - FSR = 1 МГц
 - Фдиспетчер = 200 МГц
- Аппаратная база
 - ПЛИС XC3S400A
- Состав DPU
 - Цифровой обработки
 - Вычислитель поворота координат
 - Коррелятор

- КИХ/БИХ-фильтр
- Общего назначения
 - Сумматор
 - Табличная выборка
 - Умножитель
 - Память
 - Мультиплексор
- Специализированные
 - Приемопередатчики ВУ
 - Осциллограф
 - Отладчик

Так как часть свойств NL3 выходит за рамки суперскалярной и VLIW архитектур, поясним эти моменты:

- У DPU различных типов отличается время работы. Вместе с тем время работы однотипных DPU тоже может отличаться. В суперскалярных и VLIW архитектурах активация исполнительного блока – это инструкция. NL3 поддерживает более низкоуровневое программирование. Для активации DPU требуется несколько команд: загрузка, выгрузка и начало вычислений. Можно варьировать количество тактов между этими командами и благодаря этому время работы даже однотипных DPU может отличаться.
- Длинные инструкции передаются по конфигурационной шине. К каждому DPU подключается только часть линий шины. Одни и те же линии могут быть подключены к нескольким DPU, что соответствует наложению команд для разных DPU в рамках одной инструкции. Если в VLIW архитектуре каждый фрагмент длинной инструкции предназначен для отдельного исполнительного блока, то у NL3 эти фрагменты могут пересекаться.
- DPU поддерживают конвейеризацию. Вычисления начинаются по специальной команде, так что в DPU можно загружать входные данные до выгрузки результатов предыдущих вычислений. Загрузка входных данных требует определенного времени, но она не портит значения, посчитанные ранее.

2.3.2 Модели и параметры DPU

Модель DPU позволяет получить метрики функциональных блоков DPU, по которым компилятор составляет систему неравенств. Система неравенств строится в несколько этапов, начиная с «общих», присущих всем DPU. Выделяются DPU с активацией по загрузке входных портов и по началу цикла, что отражается в системе неравенств.

Модель DPU требует предоставления перечня численных параметров, которые определяют временные характеристики работы устройства. Создана методика получения параметров временных диаграмм работы DPU, которые сводятся в специальный технологический документ – паспорт DPU.

2.3.2.1 Параметры DPU

В табл. 2.7 представлены основные параметры DPU, которые необходимо определять для создания паспорта.

Таблица 2.7. Параметры DPU

Параметр	Краткое описание
TSTABLE	Интервал использования входного порта в процессе вычислений.
TOLD	Интервал сохранения старого значения в выходном порте после начала вычислений.
TEXEC	Время вычисления выходного порта.
Δ CMD	Время упреждения команды загрузки/выгрузки порта.
Δ XCHG	Время работы терминала.
Δ EXEC	Задержка начала вычислений.

Параметр T_{STABLE} определенного входного порта может быть разным для разных выходных портов и должен представлять собой таблицу размером $N_{IN} \times N_{OUT}$, однако модель использует самое большое значение из этой таблицы.

Интервал T_{OLD} может начинаться в момент начала вычислений или после загрузки определенного входного порта. Полное количество параметров T_{OLD} представляет собой таблицу $(1+N_{IN}) \times N_{OUT}$.

2.3.2.2 Анализ структуры DPU

Один из способов получения значений временных параметров модели DPU – создание и анализ структурную схему DPU на уровне регистровых передач. Используя исходные тексты DPU, необходимо выделить регистры и уяснить их взаимодействие в процессе выполнения целевой функции. Опуская вычислительные подробности, нужно выделить скелет структуры обработки данных. Рассмотрим создание структурной схемы DPU на примере `dpu_accum`, фрагменты HDL-кода которого приведены ниже.

```
module
    dpu_accum ( dp_clk, dp_load, dp_init, dp_neg, dp_data, dp_attr, dp_oe,
dp_value, dp_vattr);
    reg [] acc; // результат работы блока (накапливает входные данные)
    reg    car;
    reg [] dp_value; // выходной регистр на шину (непосредственно значение
шины)
```

```

always @(posedge dp_clk) dp_value <= ~dp_oe ? 0 : acc[];
reg [] data; // входные данные с шины
always @(posedge dp_clk) data <= ~dp_load ? 0 : dp_data;
reg sub, init; // режимы работы
always @(posedge dp_clk) {sub,init} <= ~dp_load ? 0 : {dp_neg,dp_init};
reg [] arg; // приходится делать регистром, т.к. иначе не трассируется
reg arg_sub, arg_init;
always @(posedge dp_clk) arg <= SIGNEXT(data) ^ {...}{sub};
always @(posedge dp_clk) {arg_sub,arg_init} <= {sub,init};
wire [] rhi = (arg_init ? 0 : {acc[HI],car}) + {arg[HI],car};
wire [] rlo = {1'b0,arg_init ? 0 : acc[LO],arg_sub} +
{1'b0,arg[LO],arg_sub};
always @(posedge dp_clk) {acc[HI],car,acc[LO]} <= {rhi[],rlo[]};
endmodule

```

Выделяем основные регистры и соединяем их линиями передачи данных (рис. 2.26). Желательно расположить граф так, чтобы он начинался от шины и заканчивался ей же.

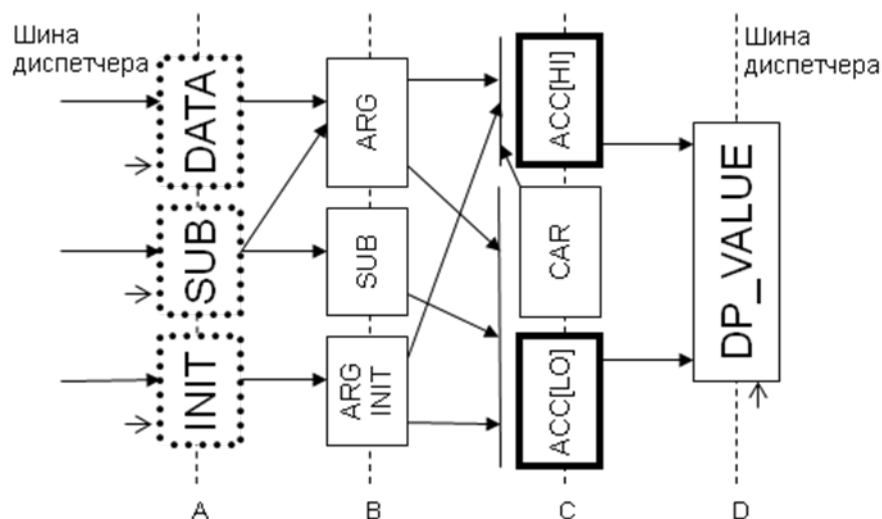


Рис. 2.26. Структурная схема dpu_accum

Для удобства “одноранговые” группы регистров размещены на одинаковых тактах, помеченных буквами A...D. Можно заметить, что на такте С в рассматриваемом DPU существует цикл (новое значение такта С зависит от предыдущего значения на том же такте).

Модель DPU требует явного выделения двух типов объектов: входных и выходных портов. В рассматриваемом случае выделение портов может показаться неочевидным, однако будем считать, что все входные порты представляют собой тройку INIT/SUB/DATA (обведены жирными точками), а выходной порт-результат – пара ACC[HI]/ACC[LO] (выделены жирной линией).

Все отсчеты интервалов должны производиться от моментов доступности соответствующих данных в регистрах (т.е. моментом попадания данных в порт считается первый такт, когда данные доступны для чтения из порта).

2.3.2.3 Определение параметров DPU

Параметры запуска вычислений

Первым параметром, который необходимо определить, является параметр запуска вычислений. Модель предоставляет три варианта: SOS, EOS, загрузка последнего входного порта. Так как первые два варианта нам не подходят, то останавливаемся на загрузке последнего входного порта. Никакой специальной задержки начала вычислений не предусматриваем, поэтому $\Delta_{EXEC} = 0$.

Время исполнения

Несмотря на то, что мы указали моментом начала исполнения загрузку последнего порта, реально расчет выходного значения начался вскоре после загрузки первого порта. Загрузка последнего порта просто завершает процесс формирования результата. Легко понять, что для рассматриваемого блока $T_{EXEC}=3$. На первом такте данные группы А переходят в регистры группы В, на втором – в регистры группы С, а на третьем происходит корректировка результата регистром CAR, после чего результат доступен в выходном порту. Данные рассуждения необходимо провести для всех выходных портов.

Стабильность входных портов

Для получения корректного результата последний входной порт требуется удерживать стабильным 2 такта (второй такт требуется для выполнения коррекции регистром CAR). Остальные входные порты для вычислений требуются в течение только 1 такта собственно того, когда они стали доступны. Однако нужно учесть, что в представленной реализации DPU, в отличие от модели, все входные порты физически располагаются в одних и тех же регистрах. Т.е. чтобы обеспечить стабильность последнего входного порта (по которому запускаются вычисления) нужно указать избыточную стабильность первого порта, которая защитит от изменения данных во входных регистрах группы А. Для рассматриваемого блока можно сказать, что $T_{STABLE1}=2$ и $T_{STABLEx}=1$ для всех $x>1$.

Удержание значений выходных портов

Модель DPU говорит о том, что даже после запуска вычислений еще какое-то время может быть доступным старое значение выходного порта. Время показывает, сколько тактов, включая такт запуска вычислений/обновления порта, остается доступным предыдущее значение выходного порта. Несмотря на то, что запуск вычислений был указан по загрузке последнего порта, реальные вычисления в рассматриваемом DPU начинаются сразу после загрузки первого порта. Загрузка не всех входов не приводит к получению результата, однако приводит к уничтожению предыдущего значения выходного порта. Так как загрузка любого порта одинаково нарушает целостность

результата, можно найти параметр T_{OLD} для любого из них. Из структуры видно, что на первом такте данные входных портов будут размещены в регистрах группы В (результат все еще будет доступен), а на следующем такте (втором) значения регистров группы С будут обновлены, тем самым уничтожая предыдущий результат. $T_{OLD}=2$ для любого входного порта и вычислений. Ввиду ограничений на порядок загрузки портов и запуска вычислений по последнему, достаточно указать T_{OLD} только для первого порта, так как именно это обращения является разрушающим.

Параметры терминалов

Относительно несложно определяются параметры входного и выходного терминалов. Для входного терминала команда и данные поступают синхронно, и время доступности данных во входном порте после наличия команды на шине составляет 1 такт (захват данных регистрами уровня А). То есть данные во входном порту будут доступны на следующем такте (такте N+1) после нахождения их на шине (на такте N). Для входного терминала в рассматриваемом блоке $\Delta_{XCHG}=\Delta_{CMD}=1$.

Аналогичным образом определяются параметры выходного терминала. Для передачи данных выходного порта (группа регистров С) на шину (регистр группы D) требуется один такт (т.е. $\Delta_{XCHG}=1$), команда выгрузки данных на шину также выполняется на следующем такте, после ее присутствия на шине ($\Delta_{CMD}=1$).

2.3.3 Функциональный блок FCUP (Filter Configurable Microprocessor)

В рамках процессора NL3 независимо друг от друга существует набор проблемно-ориентированных функциональных блоков. FCUP – это один из функциональных блоков и его функция заключается в реализации механизмов цифровой фильтрации.

Базовая структура

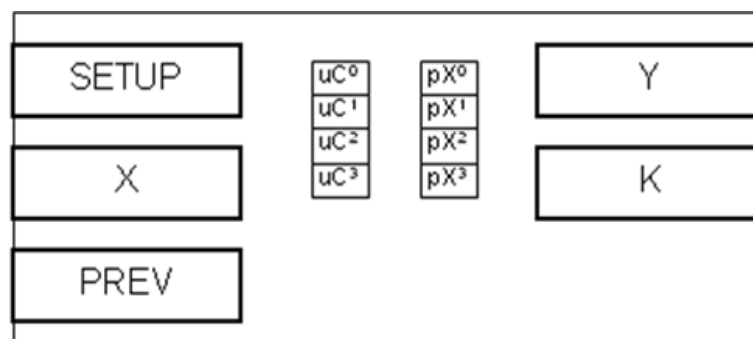


Рис. 2.27. Структура FCUP

Ядро цифрового фильтра FCUP (рис. 2.27) представляет собой микропрограммируемый автомат, способный реализовать множество независимых цифровых фильтров (КИХ/БИХ) или других вычислительных

структур схожей функциональности. Тип фильтра, размер окна входных данных и прочие параметры полностью определяются микропрограммой, хранимой в области памяти uC^1 . Скользящее окно входных данных формируется в области памяти pX^j . Входные данные получаются из входного порта X , а назначение окна входных данных микропрограмме осуществляется по значению входного порта $SETUP$. Результат фильтрации на каждом окне входных данных записывается в выходные порты Y и K : нормированное значение в диапазоне $[-1, +1]$ в порт Y и целочисленный масштабный множитель в порт K . Значение фильтрации, посчитанное на предыдущем шаге, подается на входной порт $PREV$.

$$y(n) = FCUP_{SETUP}(x(n), y(n-1)) = ky(n)$$

Основные особенности реализации:

- возможность реализации независимой работы нескольких фильтров разного типа;
- возможность начальной инициализации и сброса фильтра;
- Real-Time смена параметров фильтрации;
- оптимизация по скорости получения результата от момента загрузки очередного элемента входных данных (зависит от микропрограммы);
- поддержка холостых циклов.

Конкретный фильтр определяется микропрограммой, задаваемой при конфигурировании блока. Все микропрограммы независимы и должны работать с собственными наборами данных.

Алгоритм работы

В процессе исполнения микропрограммы автомат активизирует определенные блоки подчиненной структуры обработки данных. Конкретный вид микропрограммы определяет тип реализуемого с помощью нее фильтра. Структура обработки данных представлена на рис. 2.28.

На каждом такте работы автомата происходит передача данных между прямоугольниками, согласно указанным стрелкам. Стрелки на диаграмме обозначают направления передачи данных на каждом такте работы автомата.

При наличии надписи у стрелки соответствующая передача осуществляется при истинном условии и при передаче данных выполняется действие, указанное после наклонной черты. В круглых скобках указано начальное значение регистров при запуске микропрограммы. Регистр аккумулятора (АСС) является конвейерным с количеством стадий равным 2 для точного результата, и 1 – для предварительного. При одновременной записи/чтении X сначала выполняется запись, т.е. считывается записываемое значение.

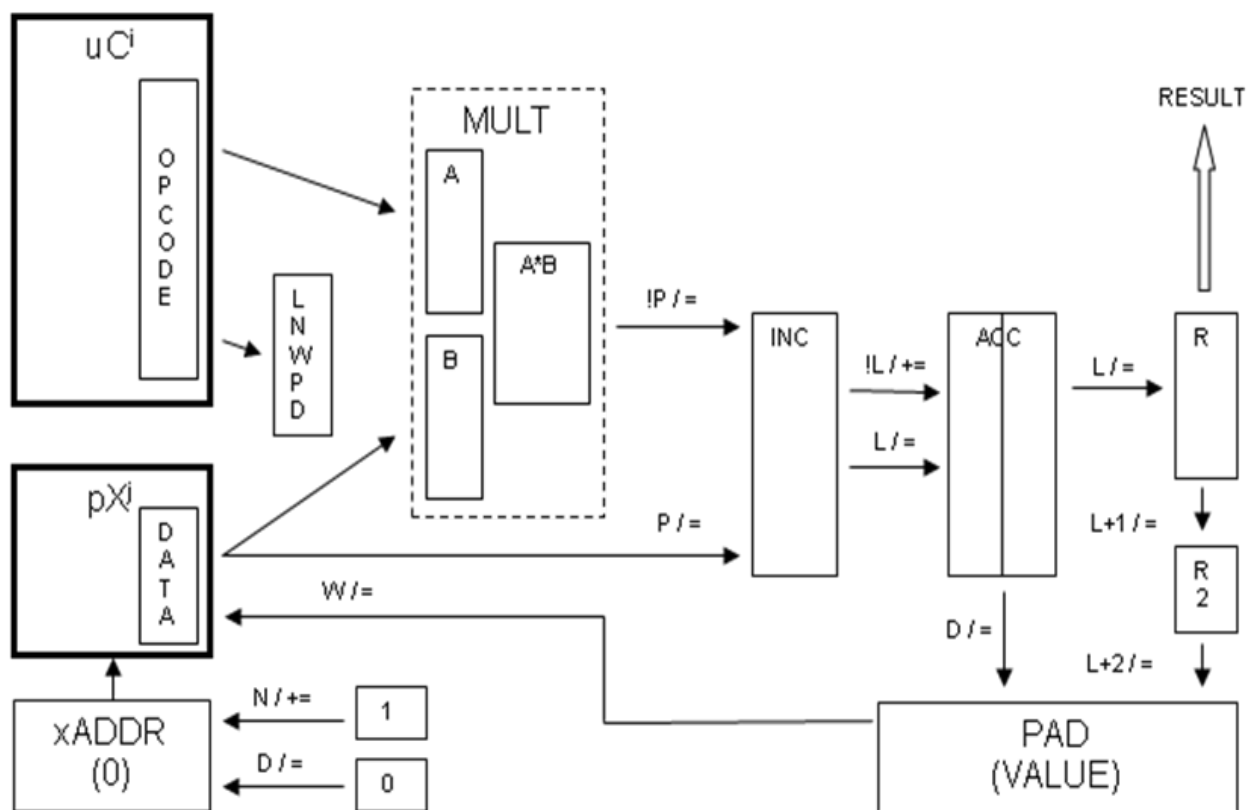


Рис. 2.28. Структура обработки данных

Для фильтрации данных необходимо загрузить очередной элемент в порт VALUE.

Микропрограмма представляет собой последовательность управляющих слов (до 128 шт.). Исполнение любой микропрограммы всегда начинается со слова номер 0. На первой инструкции микропрограммы запрещено активировать любые управляющие сигналы. Для завершения микропрограммы необходимо использовать инструкцию с активным управляющим сигналом D, после чего будет выполнена еще только одна инструкция.

Формат каждого управляющего слова представлен в табл. 2.8.

Таблица 2.8. Форматы управляющих слов

Номера битов	Имя поля	Описание
31..5	A	Коэффициент MULT.A в формате [-1..1)
4	L	Управляющий сигнал L (загрузка аккумулятора)
3	N	Управляющий сигнал N (следующий элемент X)
2	W	Управляющий сигнал W (запись элемента X)
1	P	Управляющий сигнал P (обход умножителя)
0	D	Управляющий сигнал D (завершение работы)

Применение

Основное назначение функционального блока – это реализация КИХ- и БИХ-фильтров. Все операции умножения и сложения вычисляются с помощью команд микрокода (см. табл. 2.9).

Таблица 2.9. КИХ- и БИХ-фильтры

КИХ-фильтр	БИХ-фильтр
$y(n) = \sum_{i=0}^P b_i x(n-i)$	$y(n) = \sum_{i=0}^P b_i x(n-i) - \sum_{i=1}^Q a_i y(n-i)$

Функциональный блок FCUP поддерживает работу с числами в диапазоне $[-1, +1]$. Это ограничение накладывается и на коэффициенты a_i и b_i . Поэтому вводится нормирующий множитель k , а в фильтре используются нормированные коэффициенты.

Таблица 2.10. КИХ- и БИХ-фильтры с нормированными коэффициентами

КИХ-фильтр	БИХ-фильтр
$\mathfrak{y}(n) = \sum_{i=0}^P \mathfrak{b}_i x(n-i)$ $b_i = k \mathfrak{b}_i$	$\mathfrak{y}(n) = \sum_{i=0}^P \mathfrak{b}_i x(n-i) - \sum_{i=1}^Q \mathfrak{a}_i y(n-i)$ $b_i = k \mathfrak{b}_i$ $a_i = k \mathfrak{a}_i$

$$y(n) = k \mathfrak{y}(n)$$

Таким образом, фильтр работает с нормированными коэффициентами (табл. 2.10), а масштабный множитель сохраняет в выходной порт К. В дальнейшем, чтобы получить корректное значение фильтра, нужно перемножить значения портов Y и К.

Как видно из формулы, для вычисления значения $y(n)$ требуется произвести множество операций умножения и сложения. Время вычисления прямо пропорционально количеству коэффициентов a_i и b_i . Расчеты будут занимать значительное время, а функциональные блоки, для которых эти вычисления выполняются, будут простаивать в ожидании готовности результата FCUP. На рис. 2.29 показана схема типового расчета КИХ-фильтра.

Для ускорения готовности результата была предложена оптимизация микропрограммы. Рассмотрим эту оптимизацию на примере КИХ-фильтра. На шаге n вычисляется значение фильтра:

$$y(n) = b_0 x(n) + b_1 x(n-1) + \dots + b_P x(n-P)$$

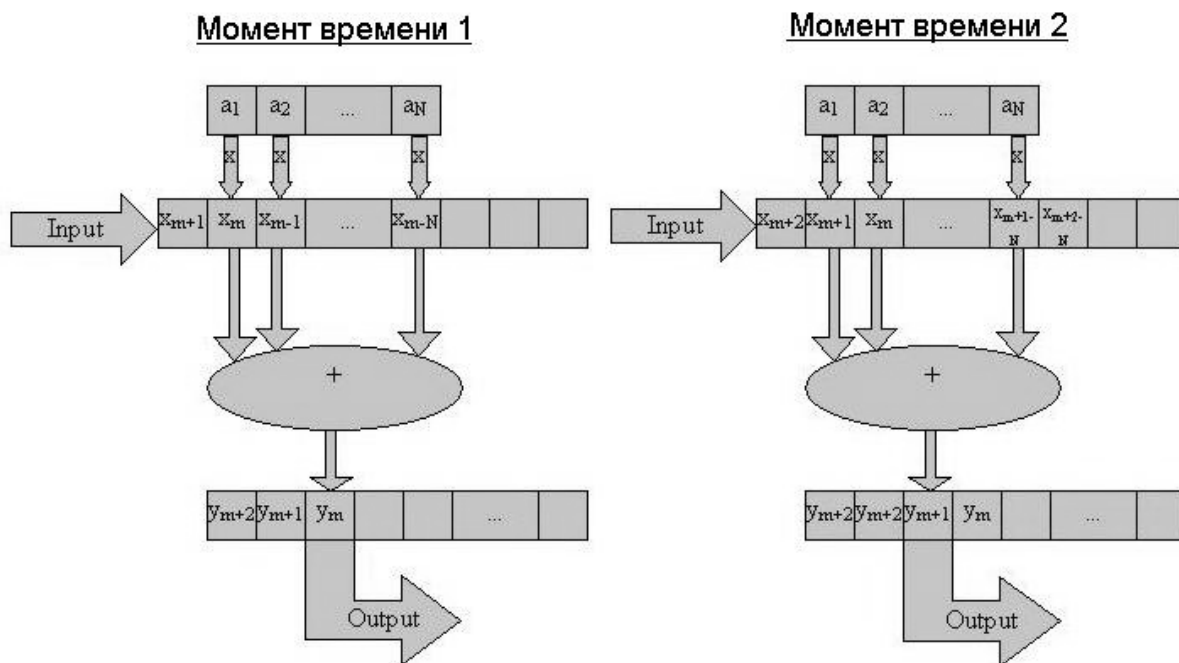


Рис. 2.29. Схема типового расчета КИХ-фильтра

Тогда значение фильтра на шаге $n+1$ будет таким:

$$y(n+1) = b_0 x(n+1) + b_1 x(n) + \dots + b_P x(n - (P-1))$$

Обозначим $S(n)$ частичную сумму фильтра:

$$S(n) = b_1 x(n) + \dots + b_P x(n - (P-1))$$

$$y(n+1) = b_0 x(n+1) + S(n)$$

Значение $S(n)$ может быть посчитано на шаге n . Тогда на шаге $n+1$ для вычисления значения фильтра придется выполнить только одну операцию сложения и умножения. Таким образом, если на шаге n сделать подготовку к шагу $n+1$, то на шаге $n+1$ фильтр можно будет посчитать очень быстро. Суммарное время работы фильтра не изменится, но готовность результата на очередном шаге наступит гораздо раньше. Схема оптимизированного расчета показана на рис. 2.30.

Конфигурирование

Функциональный блок поддерживает конфигурирование на фазе Design-Time и Run-Time. Сравнение возможностей конфигурирования на этих фазах приведено в табл. 2.11.

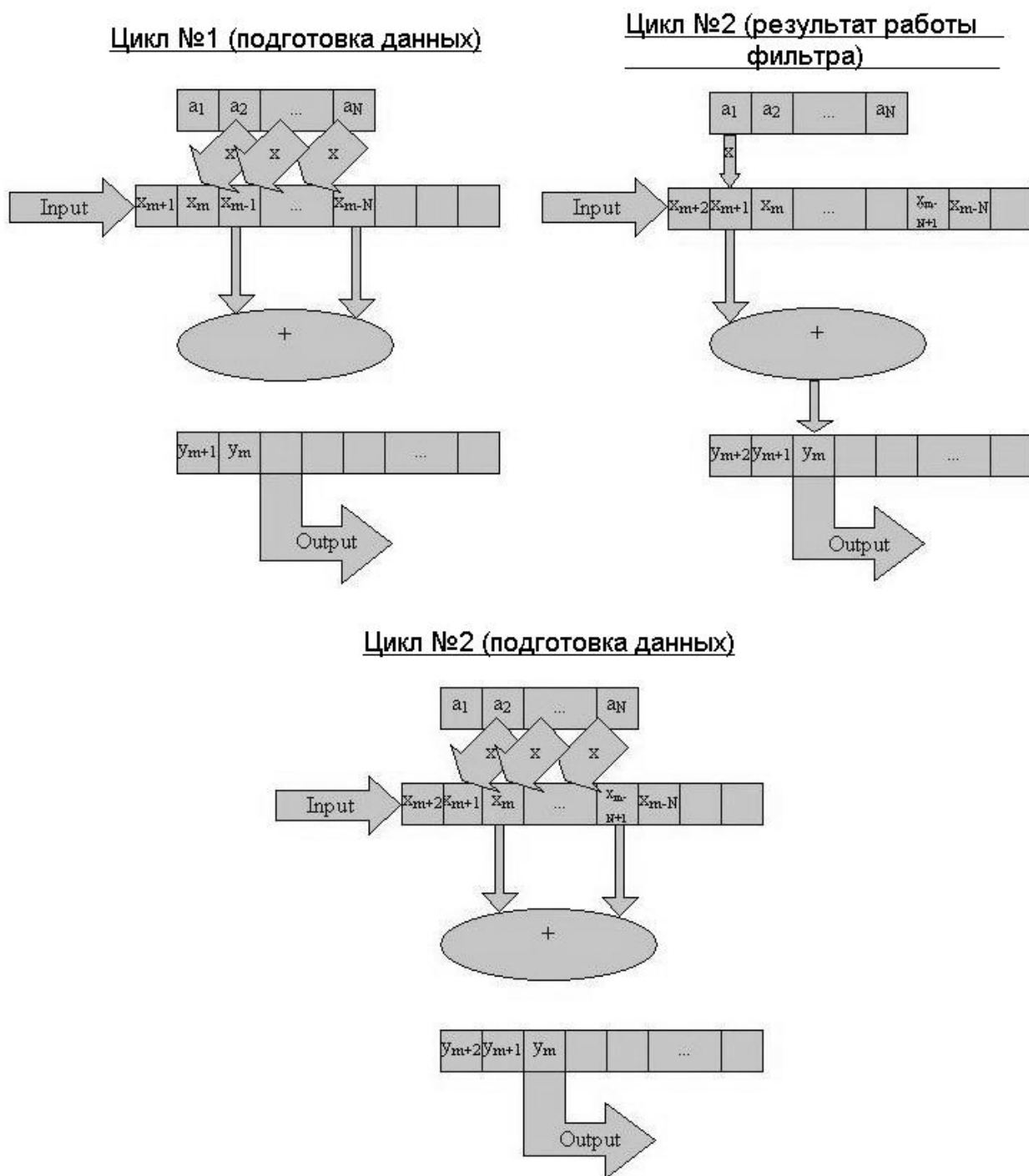


Рис. 2.30. Схема оптимизированного расчета КИХ-фильтра

Таблица 2.11. Сравнение возможностей конфигурирования на фазах DT и RT

Конфигурирование Design-Time	Конфигурирование Run-Time
• P • Q • $b_i, i \in [1, P]$ • $a_i, i \in [0, Q]$ • ΔT	• $b_i, i \in [1, P]$ • $a_i, i \in [0, Q]$

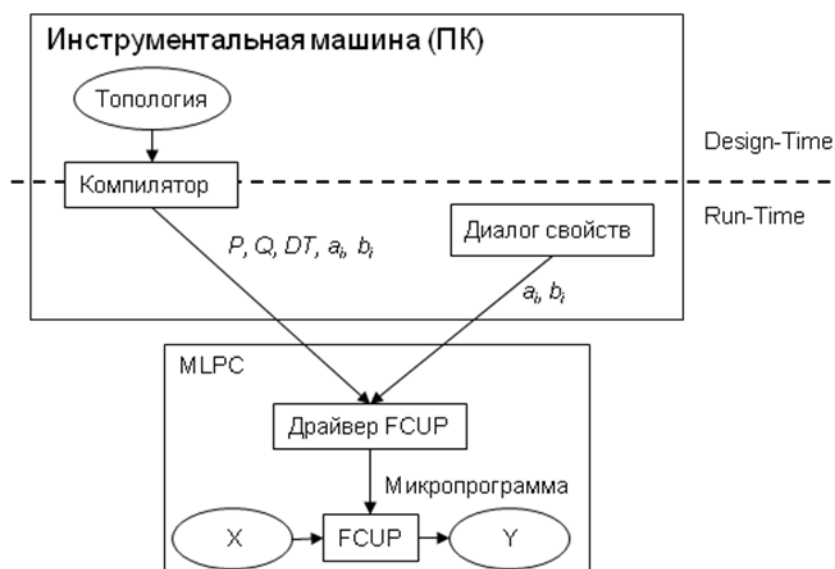


Рис. 2.31. Возможности конфигурирования фильтра

Цифровой фильтр работает в рамках какой-либо топологии. Время его работы определяется его микропрограммой. В процессе компиляции топологии это влияет на порядок работы других функциональных блоков, входящих в топологию. Параметры P и Q , задающие количество коэффициентов фильтра, непосредственным образом влияют на время исполнения микропрограммы, поэтому они могут быть установлены только на фазе Design-Time. Также на время вычисления влияет период дискретизации входного сигнала ΔT . Изменение этих параметров влечет за собой перекомпиляцию топологии, что невозможно на фазе Run-Time. Поэтому на фазе Run-Time можно менять только коэффициенты самого фильтра, но не его характер/тип. Возможности конфигурирования показаны на рис. 2.31.

Особенности формирования микрокода для фильтра скрыты в драйвере функционального блока FCUP, работающего в контроллере MLPC. Драйвер поддерживает несколько шаблонов микрокода: по одному шаблону на каждый тип фильтра. Для конфигурирования драйвера нужны параметры P , Q , ΔT , а для формирования микрокода – коэффициенты a_i , b_i . Драйвер помещает коэффициенты a_i , b_i в нужные фрагменты шаблона микрокода и заменяет им текущий микрокод функционального блока.

2.3.4 Автоматизированная система прикладного пользовательского программирования NL3

2.3.4.1 Инструментальная цепочка процессора NL3

Организация вычислений в NL3 производится посредством статического и динамического конфигурирования композиции DPU, для чего разработан прототип САПР (рис. 2.32), включающий графическую систему подготовки нет-листов, компилятор на основе системы неравенств, библиотеки логических ФБ, Design-Time и Run-Time библиотеки драйверов DPU.

Топология задает схему обработки цифровых сигналов. Изначально эта схема представляется в графическом виде, похожем на электрическую принципиальную схему. Дизайнер графических схем преобразует ее в текстовую форму, которая уже обрабатывается компилятором. Все графические примитивы топологии преобразуются в логические функциональные блоки. Каждый логический функциональный блок может быть реализован одним или несколькими функциональными блоками DPU, которые в противовес называются физическими функциональными блоками. Каждый DPU может реализовать функциональность целого ряда логических функциональных блоков. Например, все сумматоры в топологии могут быть реализованы на одном DPU типа ACCUM.

Разработана библиотека логических функциональных блоков, которая проводит соответствие между этими блоками и DPU, на которых они будут реализованы. Кроме этого есть библиотека драйверов DPU. Она предоставляет компилятору метрики DPU, на базе которых реализуются логические функциональные блоки топологии. Опираясь на эту информацию компилятор выполняет укладку (fitting) топологии и генерирует результат укладки в бинарном виде, который представляет собой исполняемый код процессора NL3. Топология в бинарном виде передается драйверам DPU, работающим в MLPC.

У драйверов DPU на инструментальной машине и в MLPC не случайно одинаковое название. Это по сути одни и те же драйверы, предоставляющие метрики DPU и интерфейсы управления. Перечень DPU, входящих в состав процессора NL3, является проблемно-ориентированным. Поэтому перечень драйверов DPU в контроллере MLPC должен соответствовать перечню драйверов DPU на инструментальной машине. Это достигается благодаря механизму самоидентификации: процессор NL3 предоставляет информацию о входящих в его состав DPU. По этой информации komponуются обе библиотеки драйверов DPU.

2.3.4.2 Система неравенств, описывающая топологию DPU

Модель DPU позволяет получить метрики функциональных блоков DPU, по которым компилятор составляет систему неравенств. В дальнейшем по этой системе неравенств производится укладка (fitting) топологии.

В системе неравенств неизвестными являются такты цепей, соединяющих логические блоки. Такт цепи – это такт, на котором эта цепь «срабатывает», т.е. ее значение находится на шине данных процессора NL3. У DPU есть входные и выходные порты. Цепи, подключенные к этим портам, называются входными и выходными соответственно.

Система неравенств строится в несколько этапов. Сначала строятся «общие» неравенства, присущие всем DPU.

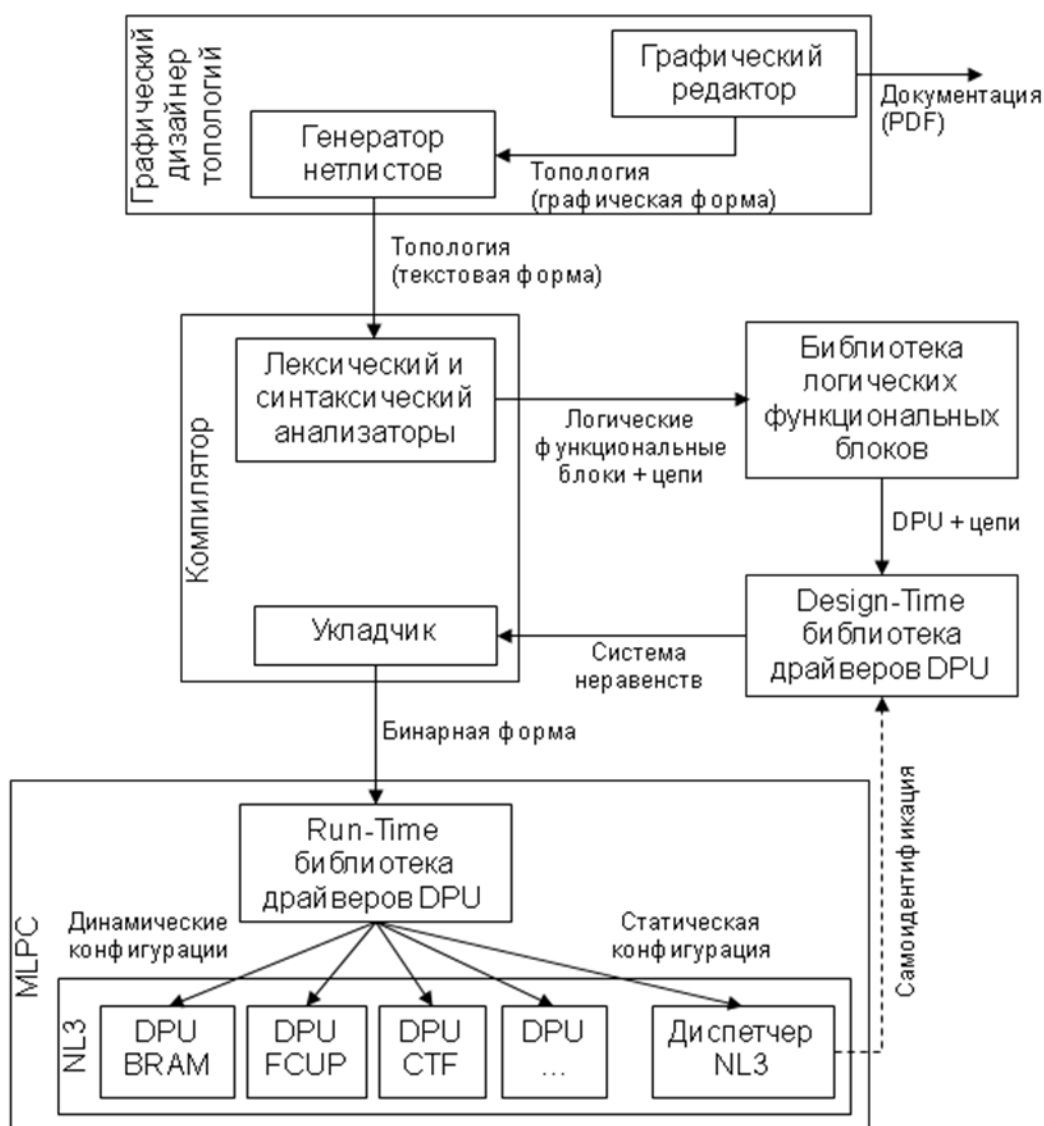


Рис. 2.32. Инструментальная цепочка процессора NL3

По способу активации DPU делятся на 2 типа: с активацией по загрузке входных портов и по началу цикла. Для каждой группы строятся свои системы неравенств. DPU, активирующиеся по загрузке входных портов, могут работать во время цикла или между циклами. В зависимости от этого тоже строятся разные системы неравенств.

Если на одном DPU реализуются несколько логических блоков, то они не могут функционировать одновременно. Между ними всегда должен быть порядок: сначала работает один логический блок, затем другой. Это формирует неравенства между цепями, принадлежащими разным логическим блокам.

В табл. 2.12 приведен пример системы неравенств, описывающей топологию композиции DPU процессора NL3.

Таблица 2.12. Математическая модель вычислительного процесса NL3

Общие неравенства	
Ограничение для входных цепей	$n_{in}[i] \geq \Delta_{CMD}^{in} - \Delta_{XCHG}^{in} + i$ $n_{in}[i] \leq CYCLE - \Delta_{XCHG}^{in} - N_{in} + i$
Определение порядка загрузки входных цепей	$n_{in}[i+1] \geq n_{in}[i] + 1$
Ограничение для выходных цепей	$n_{out}[i] \geq \Delta_{CMD}^{out}$ $n_{out}[i] \leq CYCLE - 1$
Неравенства по типу активации	
Активация по загрузке входных портов (работа в цикле)	$n_{out}[i] \geq n_{in}[N_{in} - 1] + \Delta_{EXEC} + T_{EXEC}[i] + \Delta_{XCHG}^{in} + \Delta_{XCHG}^{out}$
Активация по загрузке входных портов (работа между циклами)	$n_{out}[i] \leq n_{in}[N_{in} - 1] + \Delta_{EXEC} + T_{OLD}[i] + \Delta_{XCHG}^{in} + \Delta_{XCHG}^{out} - 1$
Активация по началу цикла	$n_{in}[i] \geq T_{STABLE}[i] + \Delta_{EXEC}$ $n_{out}[i] \geq \Delta_{EXEC} + T_{EXEC}[i]$
Парные неравенства	
Следование входных цепей	$n_{in}^{last}[i] \geq n_{in}^{first}[N_{in}^{first} - 1] + \Delta_{EXEC} + T_{STABLE}[i] + 1$
Следование выходных цепей	$n_{out}^{first}[i] \leq n_{in}^{last}[N_{in}^{last} - 1] + \Delta_{EXEC} + \Delta_{XCHG}^{in} + \Delta_{XCHG}^{out} + T_{OLD}[i] - 1$

2.3.4.3 Пример топологии DPU

На рис. 2.33 приведена топология, выполняющая функцию табличного генератора сигналов.

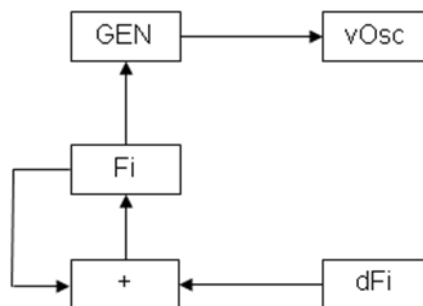


Рис. 2.33. Топология табличного генератора сигналов

```

TABLE GEN (N=128);
VSCOPE vOsc (N=1);
CELL Fi (Breakable=true);
ADD ADD;
CELL_INPUT dFi (Value = 0x00010000);

(GEN.res0, vOsc.ch0);
(Fi.dout, GEN.index, ADD.a2);
(ADD.value, Fi.din);
(dFi.dout, ADD.a1);

```

В топологии присутствуют 5 логических функциональных блоков и 5 цепей. Перечень логических функциональных блоков приведен в табл. 2.13.

Таблица 2.13. Перечень логических функциональных блоков

Название	Тип	Функция
GEN	Блок табличной выборки	Генерация сигнала согласно загруженным в него таблицам
vOsc	Цифровой осциллограф	Отображение генерируемого сигнала в реальном времени
Fi	Ячейка памяти	Накапливаемая фаза
+	Сумматор	
dFi	Ячейка памяти	Частота генерации

Принцип работы топологии заключается в логике работы табличного генератора. Ему на входной порт подается индекс элемента в его таблицах, которых требуется выгрузить на выходной порт. Если подавать разные индексы, то табличный генератор будет генерировать разные значения. Текущий индекс хранится в ячейке Fi. На каждом цикле работы процессора индекс инкрементируется на dFi. Поэтому dFi – это фактически скорость перебора значений табличного генератора, т.е. частота генерации. Индекс переполняется, но в генераторе он нормируется по размеру таблицы.

В результате перехода от логических блоков к DPU в топологии получается 6 элементов и 6 цепей. Логический блок табличного генератора GEN внутри себя содержит 2 элемента DPU: непосредственно табличный генератор и конфигурационную ячейку. В одном табличном генераторе могут храниться несколько таблиц. Конфигурирующая ячейка нужна для выбора конкретной таблицы. После назначения цепей топология принимает вид, показанный на рис. 2.34.

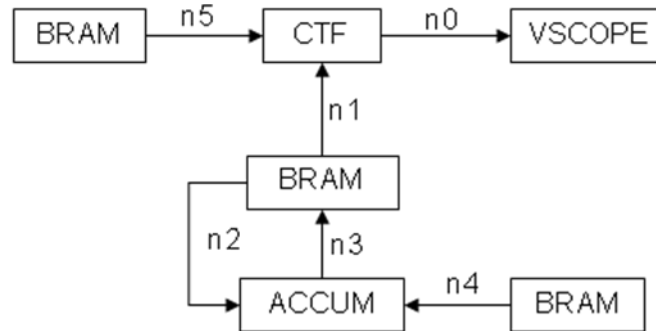


Рис. 2.34. Топология генератора с назначенными цепями

По получившейся топологии строится система неравенств и граничные условия, исходя из длительности цикла 200 тактов.

$$\begin{cases} n_0 - n_4 \geq 1 \\ n_5 - n_0 \geq 6 \\ n_2 - n_0 \geq -2, \\ n_1 - n_3 \geq 1 \\ n_2 - n_1 \geq 5 \end{cases}, \quad n_0 \in [4, 196], n_1 \in [3, 197], n_2 \in [3, 198], n_3 \in [1, 198], n_4 \in [3, 197], n_5 \in [3, 196]$$

В результате укладки находится решение, по которому топология отработывает за 12 тактов:

$$n_0 = 4, n_1 = 6, n_2 = 11, n_3 = 5, n_4 = 3, n_5 = 10$$

2.3.4.4 Распределение функций в средствах прикладного пользовательского программирования приборного контроллера

Многоуровневая организация вычислительного процесса в ВcС зондового сканирующего микроскопа MiniLab позволяет демонстрировать применение системы архитектурных абстракций и принципа актуализации вычислительного процесса. На рис. 2.35 показаны доступные через средства прикладного пользовательского программирования (СППП) уровни, которые охватывают всю иерархию распределенной гетерогенной ВcС прибора и позволяют гибко балансировать параметры целевого физического эксперимента посредством шкалы «Design/Run-Time».

В соответствии с текущей ситуацией производится ранжирование по критериям распределения функций и выбирается наиболее “удобное” распределение. При необходимости быстрой реакции и прогнозируемого времени вычислений максимальная часть пользовательского алгоритма решается на как можно более низком уровне. Если в качестве примера рассматривать задачу сканирования в СЗМ, то основная часть пользовательского алгоритма реализуется в топологии аппаратных элементов.

На долю “User Level”-программирования приходится лишь незначительная часть по передаче результатов.

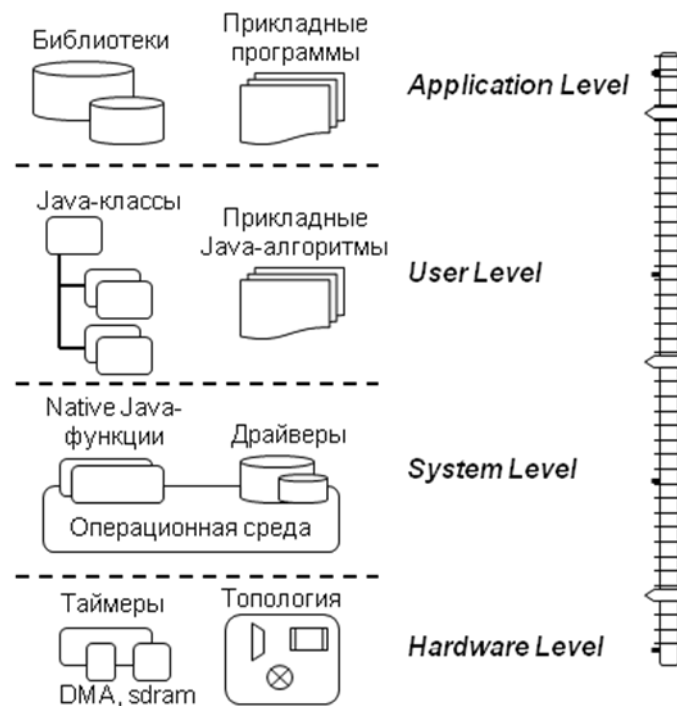


Рис. 2.35. Распределение функций в СППП MiniLab

Напротив, в случае сложности или невозможности низкоуровневого решения задачи увеличивается доля программирования уровней “User Level” и “System Level”. Здесь разработчик также может определять баланс функциональности между нативными функциями и JAVA-классами.

Критерии распределения функций в СППП MiniLab:

- скорость вычислений;
- оперативность реакции;
- возможности разработчиков (время разработки, сложность, возможности);
- ресурсы вычислительные;
- гибкость модификации, масштабируемость, (“evaluate”);
- удобство и доступность использования;
- стоимость использования;
- инструментальная поддержка;
- повторное использование (блоков, драйверов, подходов);
- интеграция в другие системы.

2.3.5 Средства прикладного пользовательского программирования вычислителей NL1 и NL2

2.3.5.1 Виртуальная машина NL1

На основе 3-уровневой архитектурной модели [14, 15, 16] разработана виртуальная машина 2-го уровня NL1. Действия, выполняемые в каждом ее состоянии, описывают соответствующий список инструкций. В автомате может быть до 16 списков, но должен присутствовать хотя бы один. Списки индексируются с нуля. NL1 всегда начинает исполнение с нулевого списка. Особенностью реализации NL1 является то, что селектор списка инструкций хранится в памяти ввода/вывода, т.е. является обычной ячейкой памяти.

Система команд NL1 состоит из 19 инструкций. Все инструкции в списках инструкций интерпретируются последовательно без пропусков. Инструкций, управляющих порядком исполнения, нет. Инструкции оперируют константами и ячейками памяти. В памяти ввода/вывода представлены ячейки следующих типов:

- ячейки для управления процессом исполнения списков;
- ячейки для хранения временных результатов;
- ячейки, представляющие периферийные устройства (Peripheral).

Драйверы всех вычислительных ресурсов контроллера реализованы на системном уровне и предоставляют интерфейс контроля и управления через ячейки ввода/вывода.

Инструкции виртуальной машины NL1 (см. табл. 2.14) имеют фиксированное количество операндов и их типы тоже зафиксированы. Операндами инструкций могут быть числовые константы и адреса ячеек памяти. При описании будут использованы следующие обозначения:

- A – значение A (числовая константа);
- [A] – значение ячейки с адресом A.

Таблица 2.14. Инструкции виртуальной машины NL1

Символьная мнемоника	Функция	Описание
MOVE(S,R)	$[R] = [S]$	Копирование значения ячейки S в ячейку R.
IMOVE(V,R)	$[R] = V$	Помещение константы V в ячейку R.
GE(A,B,R)	$[R] = [A] \geq [B]$	Сохранение в ячейке R признака того, что значение ячейки A больше или равно значению ячейки B.

LE(A,B,R)	$[R] = [A] \leq [B]$	Сохранение в ячейке R признака того, что значение ячейки A меньше или равно значению ячейки B.
GT(A,B,R)	$[R] = [A] > [B]$	Сохранение в ячейке R признака того, что значение ячейки A больше значения ячейки B.
LT(A,B,R)	$[R] = [A] < [B]$	Сохранение в ячейке R признака того, что значение ячейки A меньше значения ячейки B.
MUX2(S,R,A,B)	$[R] = [S] ? [B] : [A]$	Мультиплексор с ячейками памяти на входе. В зависимости о значения ячейки S выбирается ячейка A или B и копируется в ячейку R. Значение ячейки S должно быть либо 0, либо 1.
IMUX2(S,R,V,W)	$[R] = [S] ? W : V$	Мультиплексор с константами на входе. В зависимости о значения ячейки S выбирается константа V или W и помещается в ячейку R. Значение ячейки S должно быть либо 0, либо 1.
DMUX2(S,C,A,B)	$[S] ? [B] : [A] = [C]$	Демультимплексор с адресами ячеек на выходе. В зависимости о значения ячейки S выбирается ячейка A или B и в нее производится копирование ячейки C. Значение ячейки S должно быть либо 0, либо 1.
IEQ(V,A,R)	$[R] = [A] \equiv V$	Сохранение в ячейке R признака того, что значение ячейки A равно значению константы V.
EQ(A,B,R)	$[R] = [A] \equiv [B]$	Сохранение в ячейке R признака того, что значение ячейки A равно значению ячейки B.
OR(A,B,R)	$[R] = [A] \parallel [B]$	Сохранение в ячейке R результата операции логического ИЛИ над значениями ячеек A и B.
AND(A,B,R)	$[R] = [A] \&\& [B]$	Сохранение в ячейке R результата операции логического И над значениями ячеек A и B.
NOT(A,R)	$[R] = ![A]$	Сохранение в ячейке R результата операции логического отрицания над значением ячейки A.
ADD(A,B,R)	$[R] = [A] + [B]$	Сохранение в ячейке R результата сложения значений ячеек A и B.

SUB(A,B,R)	$[R] = [A] - [B]$	Сохранение в ячейке R результата вычитания значений ячейки B из A.
------------	-------------------	--

Доступ к периферийным устройствам в памяти ввода/вывода сделан таким, чтобы на прикладном уровне не нужно было реализовывать каких-либо протоколов обмена с ними. Поэтому периферийные устройства представлены данными: прочитанные данные и данные для записи.

Инструментальные средства

Для программирования NL1 разработан ассемблер NL2HEX. Он преобразует списки инструкций из текстового вида в формат кода для виртуальной машины. Преобразование выполняется «один в один». Для ячеек памяти ввода/вывода есть возможность определения символьных имен. Также разработан симулятор NL1Sim для отладки прикладных алгоритмов. Перечень периферийных устройств симулятора не совпадает с периферийными устройствами в целевых системах. Инструментальная машина не имеет такого же перечня периферийных блоков, как и у целевого контроллера. К тому же периферийные устройства представлены своими данными, тогда как прикладной алгоритм содержит только логику работы с ними. Данные периферийных блоков можно подменить данными других периферийных блоков, которые доступны в инструментальной машине.

2.3.5.2 Виртуальная машина NL2

Проектирование эффективной ВВМ необходимо начинать с анализа типовых прикладных задач. Типовые задачи локальных регуляторов состоят из следующих элементарных действий:

- вычисление логических термов;
- регистровые передачи;
- вычисление арифметических выражений;
- проверка условий.

При этом сам процесс вычислений носит циклический характер или активизируется по некоторым событиям. Ввиду отсутствия в прикладных алгоритмах сложных структурных операторов предлагается управляющую процедуру разбить на линейные фрагменты, исполняемые последовательно. Линейный фрагмент программы, закодированный в последовательность примитивов виртуальной машины, назовем *нетлистом*. Вся прикладная программа в таком случае представится в виде набора нетлистов, между которыми передается управление. Аналогичным образом во множество нетлистов может быть переведен алгоритм, выраженный в автоматной форме.

Для реализации элементарных действий, перечисленных выше, с учетом критериев эффективности была разработана специализированная виртуальная машина NL2. Виртуальная машина NL2 — это регистровая машина

гарвардского типа. Вычислительное ядро NL2 поддерживает одну инструкцию с конфигурируемым микрокодом. Микрокод определяет типы операндов и систему логических функций. При выполнении инструкции может быть использовано до 4 входных и 4 выходных операндов. NL2 предоставляет доступ к двум адресным пространствам данных. Через пространство памяти доступны специализированные ячейки памяти, в том числе и интерфейсы сопряжения с объектом. Регистры предназначены для хранения временных результатов и обеспечивают более высокое быстродействие машины за счет более высокой скорости доступа. Виртуальная машина NL2 является масштабируемой на этапе реализации и характеризуется четверкой параметров *NARC*, называемой *профилем*. Профиль определяет разрядность машины *N* (размер ячеек памяти и регистров), размеры адресных пространств памяти (*A*) и регистров (*R*), а также ширину представления короткой константы (*C*).

0..15	Конфигурация АЛУ	0
16..31	Конфигурация входов ($K_{\text{входов}}$)	$K_{\text{выходов}}$
32..47	Логическая функция	
48..63		
64..79		
80..95		

Рис. 2.36. Формат микрокода инструкции виртуальной машины NL2

На рис. 2.36 представлен упрощенный формат микрокода. Данные микрокода занимают 12 байт. Конфигурация ядра содержит количество активных входов и выходов и режимы работы ядра виртуальной машины NL2. Типы операндов задаются в конфигурации входов и выходов. Они определяют адресное пространство для загрузки и сохранения операндов. Система логических функций, задающая действия микрокода, позволяет определять традиционные логические операции (AND, NOT, XOR и др.), арифметические операции («+», «-»), операции пересылки данных (MOV, SWAP, XCHG и др.), логические признаки («больше», «равно», «меньше или равно» и др.), мультиплексирование, любую логическую функцию 4-х входов. Уникальность микрокода определяется именно логической функцией, остальные параметры микрокода могут быть изменены в режиме исполнения за счет указания префикса инструкции (*параметризация микрокода*). Параметризация микрокода дает возможность использовать один и тот же микрокод при изменении режима выборки или сохранения операндов, что позволяет значительно уменьшить количество микрокодов в прикладной задаче.

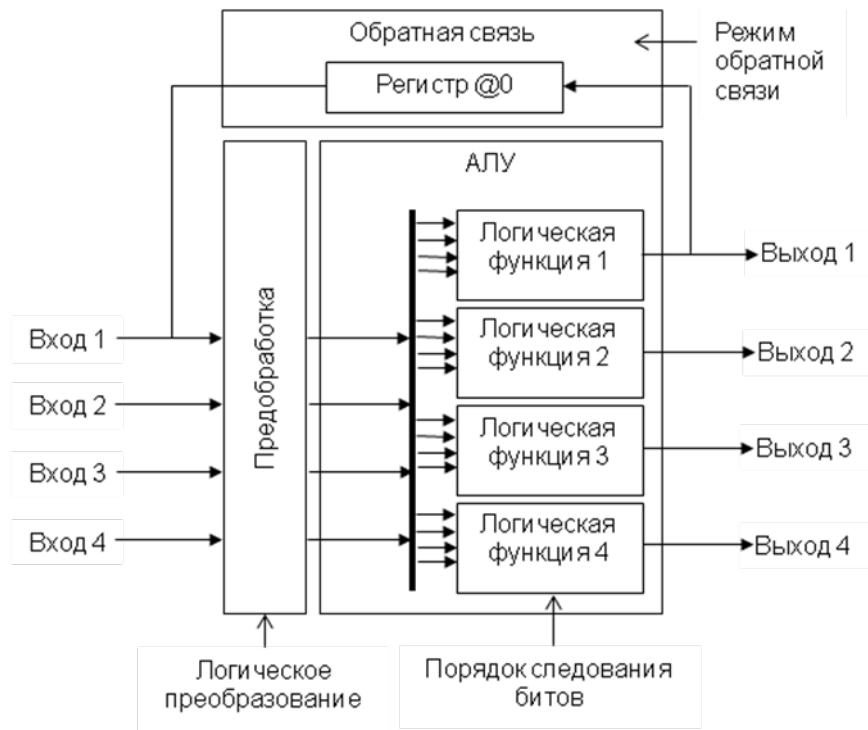


Рис. 2.37. Структура ядра виртуальной машины NL2

На рис. 2.37 показана структура ядра виртуальной машины NL2. Видно, что для интерпретации микрокода инструкции ядро получает до 4-х входных операндов. Все они поступают на входы 4-х логических функций, каждая из которых формирует один выходной результат. Таким образом формируются 4 выходных операнда. Для выполнения логических и арифметических операций ядро виртуальной машины содержит блок предварительной обработки и блок обратной связи. Все логические функции применяются последовательно ко всем битам входных операндов, поэтому сложность реализации виртуальной машины не зависит от размера машинного слова N . Алгоритм работы АЛУ NL2 представлен несложным выражением (2.1).

В формуле (2.1) N — это разрядность машины, IN — входные данные, OUT — выходные данные, а $table$ — система логических функций микрокода $\mu CODE$. Из выражения (2.1) видно, что для любой системы логических функций микрокода инструкция будет исполняться одинаковое время.

$$\begin{aligned}
 & \text{for } (n = 0; n < N; ++N) \\
 & \{ \\
 & \quad \text{int } p = 0; \\
 & \quad \text{for } (i = 0; i < 4; ++i) p|_i = IN^i|_n; \\
 & \quad \text{for } (i = 0; i < 4; ++i) OUT^i|_n = \mu CODE.table^p|_i; \\
 & \}
 \end{aligned} \tag{2.1}$$

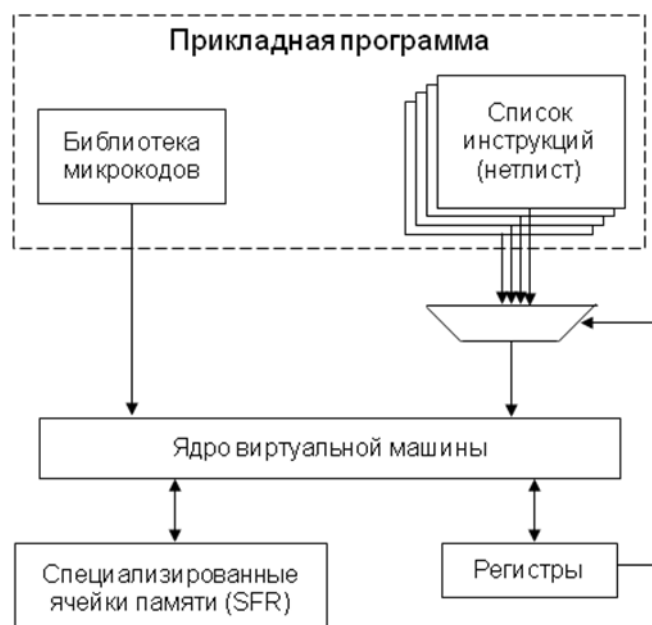


Рис. 2.38. Прикладное программное обеспечение для виртуальной машины NL2

Так как использован всего один функциональный примитив виртуального микроконтроллера, то характеристика n определяется только эффективностью реализации выражения (2.1) в каждом конкретном случае. Возможность использования до 4-х пар операндов в одной инструкции и реализация уникальных систем логических функций позволяют сделать характеристику k значительно больше 1. Еще больше повышает эффективность виртуальной машины возможность выбора специфических микрокодов для конкретной задачи.

Прикладное программное обеспечение

Для создания прикладной программы необходимо выполнить 2 этапа (рис. 2.38). На первом этапе требуется анализ задачи и создание оптимального набора библиотеки микрокодов. Анализ задачи заключается в выделении в прикладном алгоритме логических таблиц и часто используемых последовательностей элементарных действий. Удачный выбор микрокодов является залогом удовлетворения критерия компактности представления прикладной задачи (повышение показателя k). На втором этапе нужно с использованием микрокодов закодировать прикладной алгоритм. Это можно сделать в рамках инструментальных средств, предоставляющих возможность программирования на уровне ассемблера или с использованием языков высокого уровня. Важно понимать, что для каждого конкретного случая ассемблер уникален, и, следовательно, изучение набора инструкций конкретной задачи не имеет смысла. В результате прикладное программное обеспечение состоит из библиотеки микрокодов инструкций и списка нетлистов.

Применение

Типовыми применениями ВВМ NL1 и NL2 являются автономные контроллеры систем кондиционирования, программируемые таймеры

управления наружным освещением, контроллеры управления периферийными подсистемами в системах «интеллектуальный дом» и др. [20].

Пример параметров реализации виртуальной машины NL2 приведен в табл. 2.15.

Таблица 2.15. Параметры реализации виртуальной машины NL2

Базовый микроконтроллер	PIC18 (8-бит,RISC,16МГц)
Профиль NL2	NARC = { 8, 8, 4, 4 }
Макс. количество уникальных микрокодов	64
Макс. количество нетлистов	256
Занятый объем памяти кода базового микроконтроллера	1 Кбайт (500 строк С-кода)
Занятый объем памяти данных базового микроконтроллера	40 байт
Время исполнения инструкции	500мкс
Инструментальные средства	дизайнер микрокодов, ассемблер, встроенный загрузчик

2.4 Инструментальный аспект проектирования ВСС

2.4.1 Инструментальный аспект архитектурной модели ВСС

Использование альтернативных методик проектирования, общие положения которых были изложены выше, заставляет разработчиков совершенно по-иному взглянуть на инструментальное обеспечение процесса проектирования. В конце концов, сам инструментарий начинает играть огромную роль и даже заметно влиять на проектируемую систему. При таком положении дел целесообразным становится выделение инструментальной составляющей, как самостоятельного аспекта процесса проектирования ВСС.

В данном разделе идет речь об инструментальных технологиях и средствах, необходимых для эффективного проектирования современных ВСС. Формулируется ряд принципов:

- сопряженного проектирования инструментальной и целевой компоненты ВСС (частично рассматривался в главе 2 части 1 учебного пособия);
- вложенной отладки распределенных ВСС;
- взаимной инкапсуляции инструментальной и целевой коммуникационных систем.

Принципы положены в основу базового метода организации инструментальных комплексов ВСС. Изложена технология динамических

инструментальных компонент, приведены примеры реализации двух комплексов вложенной отладки распределенных ВСС. Рассматриваются варианты реализации отдельных инструментальных средств и приведены решения распространенных инструментальных задач.

2.4.1.1 Инструментальная модель ВСС

Повышение сложности ВСС, использование новых технологий, большие объемы памяти, высокопроизводительные интерфейсы, распределенная структура, использование сложных программных и аппаратных компонентов расширяют требования к инструментарию встраиваемых систем. К ВСС предъявляются повышенные требования по надежности, безопасности и соблюдению РМВ. Кроме того, в случае, если ВСС является бортовой системой управления, на первое место могут выйти такие требования, как энергопотребление и габаритные размеры. В главе 1 части 1 учебного пособия [23] была продемонстрирована необходимость разработки новых методик проектирования и, следовательно, новых инструментальных средств. Кроме привычных средств разработки (компиляторы, отладчики, трансляторы, синтезаторы) необходимо применять средства анализа моделей системы и принятия решения на основе критериев проектирования. Для эффективной работы с моделями различного рода должен существовать отдельный класс инструментальных средств, предназначенный для сбора характеристик и оценки параметров каждого конкретного элемента модели.

В современных условиях, когда именно инструментальная инфраструктура зачастую определяет уровень сложности разработок коллектива, совершенно правомерным становится выделение инструментального аспекта проектирования. Инструментальный аспект включает в себя технологии и средства верификации, реализации и отладки проектируемой системы. Именно в рамках инструментального аспекта коэффициент повторного использования наработок для коллектива разработчиков имеет решающее значение.

В рамках инструментального аспекта проектирования ВСС *инструментальную модель можно определить как перечень инструментальных задач и способ их взаимодействия при обработке потоков инструментальных данных (инструментальных объектов).*

Разнообразие инструментальных требований и их специфика ставят перед разработчиком на первый взгляд взаимоисключающие задачи, например, обеспечение развитой функциональности и минимизация используемых ресурсов, полный доступ к ресурсам системы и обеспечение безопасности функционирования системы. Любое стороннее вмешательство в систему, даже просто с целью наблюдения, приводит к изменениям характеристик последней. Инструментарий как раз и осуществляет именно такое вмешательство, тем самым, нарушая функционирование целевой системы. *Те возмущения, которые инструментарий вынужденно вносит в различные аспекты целевой системы можно называть инструментальной погрешностью.* Оценка

инструментальной погрешности осуществляется на основании характеристических функций аспектов проектирования.

Инструментальная погрешность является неустранимой в принципе. Однако при проектировании, как целевой системы, так и инструментальных средств можно делать попытки учета инструментальной погрешности. Учет инструментальной погрешности можно обеспечить двумя способами:

- разработать на этапе проектирования систему так, чтобы даже при наличии возмущений, вызванных разумной инструментальной активностью, характеристики системы не выходили за рамки выдвигаемых требований, т.е. необходимо спроектировать систему с определенным “архитектурным запасом”;
- при инструментальном воздействии на систему с целью получения определенных характеристик учитывать возмущения, вызванные оказанным воздействием, корректируя соответствующим образом полученные характеристики, приводя их к “идеальным”.

Применение стандартных, широко распространенных инструментальных средств отладки ВcС приводит к тому, что:

- система отлаживается по частям (программная и аппаратная реализации), возможности комплексной отладки сильно ограничены;
- система отлаживается только на одном или двух уровнях (как правило, это уровень исходных текстов на ЯВУ и уровень процессов в рамках ОСРВ).

Несмотря на указанные недостатки традиционных инструментальных средств, по-прежнему не имеют широкого распространения средства совместного проектирования аппаратуры и ПО (технология HW/SW CoDesign). Проблема состоит в том, что при проектировании системы в комплексе необходимо учитывать множество факторов одновременно. Неформальное, интуитивное, отданное на откуп экспертам разбиение проекта на две крупные части, аппаратуру и программу, позволило в свое время решить главную задачу: многократно уменьшить сложность проектирования, делая его рентабельным. Такое разбиение инструментальных средств позволило значительно повысить эффективность реализаций аппаратных и программных компонентов системы, однако, вопрос комплексной инструментальной поддержки процесса проектирования, особенно на начальных этапах проектирования, существующие инструментальные средства не решают.

2.4.1.2 Инструментальные средства ВcС

Инструментальные средства (или инструментарий) - это комплекс технических средств, позволяющий производить процесс разработки, отладки и тестирования ВcС в рамках известных методик. Инструментарий направлен на автоматизацию этапов проектирования, отладки и тестирования, выполняя рутинные действия и обеспечивая выполнение наиболее сложных и ответственных операций. Основные цели инструментария – исключение из

процесса разработки элементов случайности, уменьшение количества ошибок, обеспечение повторяемости результатов. Важной задачей при создании инструментария становится повышение степени его повторного использования, чтобы повысить эффективность разработок.

Как было отмечено выше, одним из ключевых свойств РИУС является нелокализованность целевой функции. Зачастую эта функция распределяется между несколькими вычислительными узлами целевой системы. Отладка любой современной ВСС является непростым делом, а некоторые характерные черты РИУС (пространственная распределенность, ограниченность ресурсов, различные вычислительные платформы, уникальные каналы связи, физическая недоступность частей системы в период тестовых испытаний и т.д.) делают задачу отладки РИУС еще сложнее.

Имеет смысл более четко определить ключевые свойства ВСС с точки зрения инструментальной поддержки:

- *Специализация.* Каждая ВСС предназначена для решения конкретной задачи, которая формулируется в ТЗ и требованиях на систему. В связи с этим нет прямой необходимости обеспечивать универсальность как системы в целом, так и отдельных ее составляющих.
- *Завершенность.* Спроектированная целевая система представляет собой самодостаточный комплекс (не случайно) связанных друг с другом объектов, решающих общую задачу. Обычно в такой системе не предусматриваются различного рода “открытые” интерфейсы для расширения функциональности.
- *Иерархичность.* Каждый объект в составе комплекса имеет различную степень важности, подчиненности и ответственности. В связи с этим система в целом имеет иерархическую структуру. В некоторых случаях иерархичность структуры значительно затрудняет доступ к отдельным компонентам системы извне.
- *Сложность.* Современные распределенные информационно-управляющие системы, в общем случае, предназначены для решения достаточно сложной задачи, что приводит к сложности (в широком смысле слова) и самой системы.
- *Распределенность.* Как было сказано выше, распределенность системы понимается в смысле распределенности функциональности по целевым вычислительным узлам. Пространственная распределенность также во многих случаях может значительно влиять на инструментальные средства.

Необходимо отметить *тяготение реализации инструментальных задач к модели вычислений потоков данных*. Эта особенность может затруднить задачу проектирования инструментальных средств, так как появляется необходимость совмещения МоС при решении целевых и инструментальных задач.

Для определения рабочего режима системы, сбора необходимых параметров и характеристик на этапе разработки и пуско-наладки разработчик должен иметь доступ к некоторым контрольным точкам внутри системы. Таким образом, при внешнем взгляде на РИУС и отладке только на уровне внешних для системы интерфейсов, она теряет свое свойство распределенности и представляется неким “черным ящиком” с логически локализованной функциональностью и заранее заданными характеристиками.

На данном этапе необходимо уяснить для себя особенность свойства распределенности. Вместе с распределенностью целевой функции системы можно говорить и о *распределенности инструментальной функции*, т.е. распределенности системы с точки зрения инструментального аспекта. Данное свойство можно очертить, потребовав распределенности целевой функции, нуждающейся в отладке. Если в системе используется мощное вычислительное ядро с подключенными к нему стандартными модулями или интеллектуальными датчиками, данная система теряет свойство инструментальной распределенности (для конкретного разработчика этой системы). Действительно, разработчик эффективно может влиять только на ядро системы, он не имеет возможности отлаживать используемые периферийные контроллеры, максимум, что он может делать – прослушивать каналы связи, т.е. интерфейсы системы с внешним миром.

Итак, система является распределенной (в контексте инструментального аспекта), если целевая функциональность, требующая инструментальной поддержки, является распределенной. Так как требование распределенности предъявляется именно к целевой функциональности, автоматически вытекает требование распределенности в рамках границ системы.

Помимо проблем комплексного обеспечения процесса проектирования инструментальными средствами, существуют проблемы инструментальной поддержки отладки и конфигурирования отдельных узлов сложных РИУС. Специализация конкретных систем, особенности реализации, сложные требования приводят к тому, что стоимость программно-аппаратного инструментального комплекса сопровождения отладки РИУС может превысить стоимость разработки самой целевой системы [13].

На последних этапах проектирования системы, связанных с реализацией отдельных компонентов, а также на этапах отладки необходимо внешнее “нецелевое” вмешательство в функционирование системы. Данное инструментальное вмешательство должно быть подготовлено на этапе проектирования и в систему должны быть заложены все необходимые механизмы. Можно условно выделить два типа такого инструментального вмешательства: “инициализация” и “отладка”.

“Инициализация”. Это этап “оживления” аппаратных средств и проверки некоторых базовых вычислительных механизмов. Система в этом состоянии еще мало напоминает целевую систему, предназначенную для решения поставленной задачи. На данном этапе решается следующая задача:

относительно простыми средствами обеспечивается доступ к механизмам и устройствам, заложенным в систему, и проверяется их базовая функциональность. Инструментальная поддержка этого этапа заключается либо в использовании дорогостоящего внешнего оборудования (например, внутрисхемных эмуляторов), либо используются самые простые средства, и успех зависит от квалификации разработчика. Элементами простого инструментария можно считать различные холодные загрузки, JTAG-каналы доставки и тестирования, эмуляторы ПЗУ [27]. Особенность этапа состоит в том, что в состав целевой системы не входят инструментальные функции, так как собственно целевой системы еще не существует. В процесс инициализации входят не только программирование и конфигурирование энергонезависимых ресурсов системы, но также некоторые начальные тесты, такие как проверка качества монтажа печатных плат и базовой функциональности отдельных микросхем в составе модулей системы. Как было сказано, целевой системы на данном этапе не существует и проверка осуществляется независимо в рамках модулей, возможно, совершенно иными способами, чем при дальнейшей отладке.

“Отладка”. На этом этапе происходит проверка работоспособности системы с постоянным повышением уровня и сложности тестов. Повышение уровня связано с иерархичностью целевой системы, сложность объективно повышается в связи с повышением уровня тестирования.

В общем случае для РИУС можно выделить три уровня отладки: непосредственная отладка, прямая (удаленная) отладка, вложенная отладка [13]. Каждый из перечисленных уровней требует реализации предыдущих, однако, возможны системы комплексной отладки, когда одновременно применяются несколько уровней.

Непосредственная отладка. Отлаживаемые компоненты и средства отладки взаимодействуют друг с другом непосредственно. В этом случае отладочные средства и отлаживаемые объекты располагаются в пределах одной и той же вычислительной платформы и тесно друг с другом связаны. Связь заключается во взаимном управлении отладочных средств и объектов отладки. Отлаживаемые модули могут выполняться под постоянным контролем средств отладки или средства отладки активизироваться время от времени по командам отлаживаемых объектов. В данный класс инструментария входят такие средства как интерактивные отладчики, эмуляторы, профилировщики и т.д. Кроме того именно к этому классу инструментальных средств относится подавляющее большинство компиляторов и линковщиков.

В связи с тесным взаимодействием компонентов и наличием общей вычислительной платформы такой способ отладки позволяет получать наибольшее количество информации об отлаживаемой системе, но имеет значительные недостатки:

- значительная ресурсоемкость реализации;

- объективная сложность генерации внешних воздействий;
- недостаточная адекватность отлаживаемой модели целевой системе;
- значительная инструментальная погрешность.

Особенно значимы последние два недостатка, которые не позволяют широко использовать указанный метод отладки. Этих недостатков в значительной степени лишен следующий метод отладки – прямая (удаленная) отладка.

Прямая (удаленная) отладка. Отлаживаемые компоненты удалены от средств отладки, но между ними существует прямая связь. Прямая отладка сводится к проведению непосредственной отладки целевой системы через выделенный инструментальный интерфейс с использованием инструментальных агентов. *Инструментальный агент* – элемент целевой системы, который при необходимости может быть удален из системы в конечной реализации, отвечающий за поддержку непосредственной отладки и связь по выделенному инструментальному интерфейсу с отладочными средствами, располагающимися на другом вычислителе целевой системы или на специальной инструментальной ЭВМ (хост-машине). Данный подход позволяет резко сократить инструментальную погрешность, снизить ресурсоемкость и решить проблему эмуляции окружения отлаживаемого объекта, так как система помещена в реальные условия эксплуатации и расположена на целевой вычислительной платформе. Однако нельзя не отметить, что наличие удаленной инструментальной платформы, ослабление контроля (связанное с необходимостью снижения инструментальной погрешности) и другие особенности порождают новый класс проблем. К ним относятся доставка компонентов, конфигурирование, управление, обмен данными между целевой системой и хост-машиной. При этом не нужно забывать о необходимости выделенного инструментального канала связи для проведения прямой отладки. Несмотря на перечисленные сложности, использование данного подхода остается крайне эффективным, что подтверждается наличием большого количества средств прямой отладки. Эти средства можно разбить на две группы.

- Непосредственно отладчики. Это программные комплексы и инструментальные агенты, использующие различные инструментальные каналы и обладающие различной мощностью, предназначенные для непосредственной отладки целевой системы.
- Анализаторы каналов связи, генераторы трафика. Данные средства позволяют искусственным образом разбить систему на подсистемы путем анализа взаимодействия системы с внешним миром и генерированием внешних воздействий. При этом в целевой системе не увеличивается вредное влияние инструментария, что позволяет использовать такие средства [потенциально] в любых количествах.

Метод прямой отладки хоть и эффективен, но не всегда реализуем. Для осуществления прямой отладки необходимо иметь доступ к целевой системе через выделенный инструментальный канал, что далеко не всегда возможно, особенно для РИУС. В случае отсутствия выделенного инструментального канала хотя бы у одного из модулей в составе целевой системы, для решения задач отладки такого модуля необходимо использовать метод вложенной отладки.

Вложенная отладка. Данный способ отладки предназначен для обеспечения прямой отладки модулей, не имеющих собственных выделенных инструментальных каналов. Обмен данными с такими модулями происходит с помощью построения инструментальной КМС с использованием как выделенных инструментальных каналов, так и целевых каналов связи.

2.4.2 Инструментальный комплекс вложенной отладки распределенных ВСС

К инструментальному комплексу вложенной отладки РИУС предъявляются следующие требования.

Универсальность, широкая область применения. Так как коллективом разрабатывается широкий спектр устройств различной сложности и сами РИУС характеризуются разнообразием элементной базы, то инструментальная система должна обеспечивать работу как с малыми системами (PIC, MCS51 и т.д.), так и со сложными контроллерными сетями и мощными вычислителями. Отсюда следует, что инструментарий не должен навязывать жесткие протоколы и правила поведения внутри отлаживаемой системы, однако в этом случае даже при сохранении работоспособности инструментальной системы, ее мощность резко понижается. Понижение мощности связано с низким уровнем вложенной тестовой поддержки.

Простота реализации базовых механизмов обеспечения вложенной отладки. Простота здесь подразумевается как высокая скорость изготовления соответствующих механизмов, так и низкая требовательность к ресурсам. Простота реализации базовых механизмов необходима для применения стандартной инструментальной среды при отладке целевой системы на ранней стадии разработки при получении основных характеристик и параметров реализации.

Мощность. Инструментальная система в целом должна обладать определенной “мощностью”. Т.е. система должна быть не в роли необходимого звена, но и в некотором смысле достаточного. В идеале в каждом конкретном целевом проекте инструментарий должен быть в состоянии решать возникающие перед разработчиком задачи.

Открытая масштабируемая архитектура. Мощность требует наличия открытой и масштабируемой архитектуры. Масштабируемость архитектуры вытекает из тезиса потенциальной возможности решения любого вопроса. Решенные однажды проблемы порождают новые, возможно более сложные, и

сложность задачи, в конце концов, превысит возможности инструментария. Для преодоления такого барьера необходимо иметь масштабируемую (расширяемую) систему. Архитектура должна быть открытой, чтобы обеспечить простоту масштабирования инструментария потенциально любым разработчиком, однако в рамках базовых принципов. Отсюда вытекает требование создания шаблонов инструментальных моделей в рамках коллектива, использующего аспектную модель проектирования. Зафиксированные шаблоны в контексте инструментальной АСМ позволят значительно повысить коэффициент повторного использования инструментария.

Совместимость с существующими средствами отладки. Данное свойство позволит просто перейти на вновь создаваемую инструментальную среду. Экономически невыгодно и практически невозможно строить всю инструментальную цепочку в рамках конкретного проекта самостоятельно. Инструментальный комплекс должен обеспечивать как можно более тесную интеграцию с уже существующими инструментальными средствами.

Повторное использование. Данное требование очевидно, однако неочевидно как его добиться. Его реализация тесно связано с архитектурой системы. Эта связь не столько с масштабируемостью и открытостью архитектуры, сколько с ее распределенностью. Скорость работы системы можно достичь, перенеся большинство функций в целевую систему, оставив хосту задачу взаимодействия с пользователем. Однако в этом случае крайне затруднено повторное использование реализованного таким образом механизма. В случае обратного распределения функций приобретает возможность простого повторного использования, однако вместе с тем и катастрофическая потеря скорости.

Надежность и безопасность. Под надежностью понимается степень достоверности получаемых с помощью инструментария результатов, готовность системы, реакция системы на непредвиденные обстоятельства. Сюда же можно отнести адаптируемость инструментальной среды. Безопасность инструментария понимается как надежная адекватная реакция на непреднамеренные неверные действия пользователя или системы, а также противодействие намеренным разрушающим действиям. Удобной возможностью является введение различных привилегий и уровней доступа к возможностям инструментального комплекса.

Возможность отладки целевой системы в широком смысле включает в себя:

1. Конфигурирование системы. Идентификация элементов системы, доставка конфигурационных данных, ведение архивов, контроль версий и т.д.
2. Мониторинг системы. Отслеживание состояний целевой системы и поведения отдельных процессов в системе.
3. Сбор статистики по состояниям.

4. Генерирование и прогон тестов по проверке функциональности.

Имеются следующие принципиальные противоречия (табл. 2.16), которые необходимо решать в рамках задачи создания инструментального комплекса вложенной отладки.

Таблица 2.16. Требования к инструментальному комплексу вложенной отладки

Специализация целевой системы	Универсальность инструментальной системы
Отладка именно целевой системы	Внедрение в целевую систему инструментальных структур
Безопасность целевой системы	Мощность встроенных инструментальных средств
Оптимальность решения инструментальных задач	Возможность повторного использования

Возможность проведения вложенной отладки, требует построения выделенного [виртуального] инструментального канала на основе топологии целевой КМС. При этом нужно учитывать перечисленные выше противоречия.

Для решения поставленных задач необходимо использовать существующие в системе каналы передачи данных для совместной передачи данных целевого и инструментального характера. Эта задача должна быть решена, чтобы разработчики архитектуры целевой системы в целом и целевого программного обеспечения в частности могли не обращать внимание на наличие инструментальных компонентов в составе активных устройств и проблемы совместного использования канала передачи данных. Архитектура инструментальной среды должна сделать эти вопросы “прозрачными” для упомянутых выше специалистов. В общем случае топология связей инструментальной системы может не совпадать с топологией целевой системы. Скорее всего, инструментальная топология проще целевой, но при этом в сеть связываются все объекты целевой системы.

Одной из проблем, стоящей перед инструментальной КМС вложенной отладки, является проблема совместного использования среды передачи и совмещения МоС. В большинстве случаев связи между активными устройствами в составе целевой системы строятся на основе контроллеров, обеспечивающих стандартный интерфейс, однако возможны и исключения. Стандартные контроллеры связи обычно обеспечивают до двух-трех уровней модели ISO/OSI (физический, канальный и сетевой) [54]. Физический уровень имеет дело с передачей битов по физическим каналам. Канальный уровень отвечает за целостность передачи фрейма по каналу связи. Сетевой уровень служит для образования единой транспортной системы, объединяющей несколько сетей с различными принципами передачи информации между конечными узлами. Именно с сетевого уровня могут различаться целевая и

инструментальная сети. Физический и каналный уровни обеспечат доставку в пределах одного канала передачи данных, и тут сложно выявить различия, особенно, если эти уровни реализованы с помощью контроллеров стандартных сетей передачи данных. А вот на сетевом уровне можно выделить различия в топологии, правилах маршрутизации, иерархии объектов и т.д. Можно выделить два принципиальных способа решения задачи совмещения целевой и инструментальной КМС: инкапсуляция инструментальной КМС в архитектуру целевой и инкапсуляция целевой КМС в архитектуру инструментальной.

2.4.3 Совмещение целевой и инструментальной КМС в распределенных ВСС

2.4.3.1 Инкапсуляция инструментальной коммуникационной среды в целевую

Инструментальная информация курсирует в пределах системы посредством целевой КМС (рис. 2.39). В каждом активном устройстве, имеется процесс, отвечающий за обработку инструментальных пакетов и сбор отладочной информации.

Достоинства способа:

- сохранение целевой топологии системы, скоростей работы сети;
- малая вносимая инструментальная погрешность;
- достаточно простое исключение инструментальной части.



Рис. 2.39. Инкапсуляция инструментальной КМС в целевую

Недостатки способа:

- внесение равноправного процесса, отвечающего за отладку, в архитектуру ПО каждого вычислительного узла целевой системы;
- изменение *целевого* стека протоколов при изменении *инструментальных средств и задач отладки*;
- зависимость алгоритмов системы отладки от конкретной архитектуры целевой системы;
- привнесение целевого стека протоколов на инструментальную хост-систему, перенос специализированной информации в хост-систему;

- сложность полного мониторинга КМС в отладочных целях.

Этот способ удобно использовать в случае сбора статистики и тестирования готовой целевой системы. Рассматриваемая архитектура не оказывает значительного влияния на функционирование сети системы. Более того, при расчете характеристик целевой системы можно зарезервировать часть ресурсов для решения инструментальных задач. Система может существовать автономно, без подключения к инструментальной хост-машине. На таком уровне развития системы инструментальные задачи настолько специализируются (диагностика, сбор статистики и т.д.), что их трудно называть “идеологически” инструментальными задачами. Они становятся целевыми задачами определенной степени важности. Данный способ практически неприменим на начальных этапах разработки целевой системы по следующим причинам:

- На начальных этапах отладки системы целевая задача слабо проработана, не говоря уже об ее реализации. Разработка целевой архитектуры займет достаточное время, в течение которого необходимо работать с системой для проверки жизнеспособности аппаратуры, решения задач начальной инициализации, измерения пропускных способностей каналов и т.д.
- Даже при разработанной целевой архитектуре потребуется значительное время, чтобы обеспечить уникальную инструментальную среду на хост-машине. Это время можно резко сократить при применении открытой масштабируемой архитектуры на хост-машине, но даже в этом случае использование нового не отлаженного компонента в составе инструментальной системы может затруднить отладку системы в целом.
- Архитектура целевой системы, особенно КМС, может не удовлетворять требованиям, предъявляемым к инструментальной системе.

2.4.3.2 Инкапсуляция целевой коммуникационной среды в инструментальную

Основной КМС, связывающей активные объекты системы, является инструментальная КМС, по которой по мере развития системы начинают передаваться и “целевые” пакеты (рис. 2.40). Данная архитектура не подразумевает наличие целевой задачи, последняя может отсутствовать полностью, что не мешает разработчикам располагать мощным стандартным средством отладки и всеми наработками прошлых проектов в виде стандартной инструментальной среды.

Достоинства способа:

- минимальные затраты на развертывание сети и проведение начальных тестов;
- использование стандартных надежных отладочных средств;

- сохранение алгоритмов инструментальной среды;
- полный мониторинг целевой сети;
- легкая и независимая модернизация целевой архитектуры.



Рис. 2.40. Инкапсуляция целевой КМС в инструментальную

Недостатки способа:

- внесение драйвера-фильтра сетевого уровня в архитектуру ПО каждого вычислительного узла целевой системы;
- значительное влияние на функционирование целевой сети;
- потенциальная зависимость от хост-машины (можно применять различные решения для построения инструментальной архитектуры сети);
- сравнительно сложное выделение инструментальной составляющей (при необходимости).

Описанный способ удобно использовать на начальных этапах разработки системы. Работоспособность инструментальной системы слабо зависит (или почти не зависит) от степени готовности целевой задачи. Реализовав необходимые простейшие базовые механизмы, которые вытекают из базовых требований к используемой инструментальной системе, ориентируясь по стандартным шаблонам или используя готовые наработки, разработчик оказывается в знакомой инструментальной среде. Эта среда позволит с легкостью решать задачи инициализации и начального конфигурирования системы, надежной и безопасной доставки, запуска процессов, сбора информации и т.д. При постепенном развитии целевой составляющей круг задач может расширяться за счет мониторинга и управления целевыми процессами. Простота такого расширения обеспечивается открытой масштабируемой архитектурой инструментальной среды. При развитии целевой системы, а особенно ее коммуникационной части, инструментальная среда с рассматриваемой архитектурой, хоть и решает задачи отладочного плана, но оказывает значительное влияние на показатели работоспособности целевой системы. Это связано со значительным влиянием, оказываемым на функциональность целевой сети при ее инкапсуляции в инструментальную. Вызвано это не столько наличием дополнительного программного обеспечения

в каждом активном устройстве и его зависимостью от хост-системы, сколько кардинальными отличиями топологий и дисциплин инструментальной и целевой сетей.

Используя любой из описанных способов совмещения целевой и инструментальной КМС, целевая система, снабженная некоторыми инструментальными функциями, представляется в следующем виде (рис. 2.41).

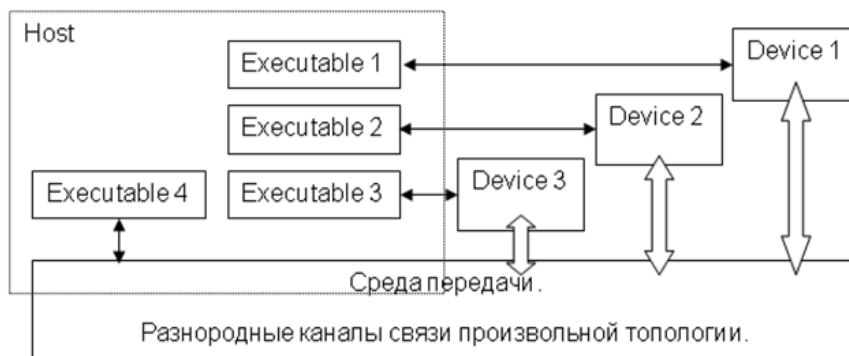


Рис. 2.41. Целевая система с инструментальными функциями

Как упоминалось выше, большинство инструментальных средств располагаются в области прямой (удаленной) отладки. Такие средства рассчитаны на отладку одного активного устройства, а не системы в целом, особенно, если система имеет распределенную архитектуру. Большинство предлагаемых методик отладки рассчитаны на конфигурацию Host $\leftrightarrow$ Target или Host $\leftrightarrow$ Target1, Target2 ... TargetN, где все цели равноправны и подключены к хост-машине непосредственно. Иерархичность целевой системы и транзит данных удаленных объектов не рассматривается. При отладке РИУС необходимо решать две проблемы: отладка всех активных объектов (как изолированных устройств) и взаимодействие активных объектов друг с другом через среду передачи. Для решения первой задачи отладочная среда должна обеспечивать виртуальную изоляцию конкретного устройства (выделение специального логического инструментального канала связи), чтобы вернуть разработчика в привычную для него конфигурацию Host $\leftrightarrow$ Target со всеми вытекающими удобствами, такими как доставка, конфигурирование, вывод отладочной информации и т.д. Для решения второй задачи отладки отладочная среда должна обладать возможностями сетевого анализатора. Очевидно, что при развитии системы изоляция активных устройств невозможна, но такую изоляцию можно поддерживать в конкретных отладочных целях.

На рис. 2.41 Executable 1,2,3 программы-мониторы соответствующих устройств целевой системы, связанные с инструментальными агентами. С каждым инструментальным агентом монитор связан собственным виртуальным инструментальным каналом, и тем самым обеспечивается логическая связь Host $\leftrightarrow$ Target. Executable 1,2,3 могут быть как одна и та же программа, запущенная для разных устройств целевой системы (например эмулятор терминала), так и процессы в составе сложного отладочного комплекса. Процесс Executable 4 выполняет задачи представления данных мониторинга

сети. Такая архитектура достаточно просто реализуется при использовании обоих способов совмещения целевой и инструментальной КМС.

2.4.4 Динамические инструментальные компоненты

В рамках предложенной инструментальной модели построения инструментального комплекса (инструментальные агенты и внешние компоненты, связанные выделенным интерфейсом) предлагается способ организации инструментальных компонентов для РИУС, позволяющий упростить разработку резидентного инструментария и повысить эффективность повторного использования инструментального комплекса [28]. Как было сказано выше, инструментальный комплекс представляет собой множество инструментальных функций, выполняющих обработку данных определенного типа. При этом очевидно, что в конкретной целевой системе в конкретный момент времени активированы далеко не все заложенные в нее инструментальные возможности.

Каждая из функций инструментальной модели представляет собой достаточно крупную задачу, которую можно представить как набор более мелких задач. Одни из них могут быть выполнены только на борту целевой системы, другие только на инструментальной хост-машине. Опираясь на определение инструментальной модели, важной составляющей инструментального комплекса можно выделить собственно обрабатываемые инструментальные данные. Инструментальный комплекс в конечном итоге решает определенную инструментальную задачу, функционально распределенную между целевой системой и инструментальной машиной. Критерием оценки качества (оптимальности) конкретного разделения функций между внешними и резидентными компонентами могут служить скорость работы, загрузка инструментального канала, минимизация использованных ресурсов целевой системы и т.д.. Проблемы декомпозиции инструментальной задачи по целевым вычислительным узлам сродни проблемам декомпозиции целевой функциональности и должны решаться схожими методами, но в рамках инструментального аспекта целевой системы.

В минимальной конфигурации инструментальный комплекс состоит только из распределенного ядра, скрывающего в себе конкретный инструментальный интерфейс. Соккрытие конкретного инструментального интерфейса в ядре является важным моментом, так как остальные (собственно инструментальные) функции комплекса становятся независимыми от реализации инструментального канала связи и топологии КМС и пользуются интерфейсом, предоставляемым ядром инструментального комплекса. Нарастивание функциональности инструментального комплекса происходит благодаря возможности ядра создавать резидентные компоненты и обрабатываемые ими данные. Динамическое создание резидентных инструментальных функций (в узком смысле – инструментальных агентов) и данных позволяет значительно

сэкономить ресурсы целевой системы и повысить безопасность ее функционирования.

Значительным преимуществом инструментального комплекса является обеспечение доступа к различным ресурсам целевой РИУС (структуры ОСРВ, данные системного менеджера памяти и т.д.). Представленная технология позволяет реализовать механизм “инструментальных мостов”. Механизм заключается в том, что для интересующих сугубо целевых функций создаются простейшие функции-wrapper’ы, которые “экспортируют” системную целевую функцию в инструментальный комплекс. Такие инструментальные мосты могут быть с легкостью динамически “наведены” практически в любую точку целевой системы и уничтожены после выполнения инструментальной задачи. Мостами разработчик может обеспечить себе доступ к таким важным блокам целевой системы, как файловая подсистема, подсистема ввода/вывода или менеджер памяти. Пример реализации технологии динамических инструментальных компонентов приведен в [28].

Представленная архитектура инструментальных средств позволяет создавать мощные инструментальные комплексы для отладки сложных РИУС. Простота использования и относительная нетребовательность (гибкие политики) к ресурсам целевой системы позволяют применять этот подход для обеспечения инструментальной поддержки широкого класса РИУС. Предлагаемая архитектура позволяет учесть большинство инструментальных требований на этапе проектирования в рамках инструментального аспекта и формально оценить оптимальность распределения инструментальных функций между инструментальной машиной и целевой системой.

Апробация описанной архитектуры ядра инструментального комплекса показала его высокую универсальность и эффективность при разработке достаточно сложных РИУС. Простота использования и легкая масштабируемость позволила независимым группам разработчиков создать мощный инструментальный программный комплекс в сжатые сроки.

Далее демонстрируется использование представленных принципов на примере решения задачи обновления ПО распределенных ВСС.

Современные условия вынуждают коллективы разработчиков ВСС резко сокращать длительность этапа проектирования и в кратчайшие сроки выходить на этап пуска-наладки и эксплуатации. При этом разработчику приходится жертвовать функциональностью и качеством создаваемого ПО, чтобы уложиться в заданные сроки. Одним из способов исправить ситуацию является предоставление возможности обновления ПО микропроцессорных модулей системы после того, как она была сдана в эксплуатацию и установлена на объекте [29].

В общем случае проблема обновления ПО микропроцессорного модуля сводится к использованию инструментального канала ICP/ISP для замены

содержимого (обычно это объектный код программы микроконтроллера) энергонезависимой памяти самого микроконтроллера или внешних микросхем.

Даже будучи относительно неудобным, описанный способ позволит решить проблему обновления программного обеспечения компактных одномодульных систем. Для более сложных РИУС такой способ становится очень ресурсоемким и сложным, как для разработчика, так и для конечного пользователя.

Для отдельных микроконтроллеров производители предлагают средства IAP. К таким средствам относятся различные аппаратные решения, предусмотренные производителями микроконтроллеров для перепрограммирования ресурсов энергонезависимой памяти прямо из целевой задачи без перевода микроконтроллера в специальный режим работы.

Таким образом, задачу обновления ПО РИУС разработчик решает на двух уровнях: на системном уровне и на уровне микропроцессорного узла системы.

На первом уровне необходимо обеспечить взаимодействие частей системы в рамках единого инструментального протокола, идентификацию типов узлов и версий ПО, доставку новых версий ПО до вычислительных узлов. При этом общая работоспособность системы не должна быть нарушена.

На уровне вычислительного узла необходимо обеспечить защиту ПО, проверку целостности доставленного ПО, перепрограммирование энергонезависимых ресурсов микроконтроллера.

Так как обновление должно быть полностью доставлено и сохранено перед началом его обработки в периферийном модуле, необходимо учитывать требования малой ресурсоемкости. Технически задача решается сокращением объема обновления с использованием относительно несложных алгоритмов сжатия данных без потерь. На персональном компьютере разработчика создается файл-обновление, который потом нужно без изменений доставить в периферийный модуль. Файл-обновление самодостаточен, что позволяет периферийному модулю в дальнейшем самостоятельно восстановить объектный код ПО.

Инструментальная модель автоматизированного комплекса обновления ПО РИУС представлена на схеме (рис. 2.42).

Реализация резидентных инструментальных средств обновления ПО РИУС в полном объеме была получена для периферийных модулей проекта КТЖ-2. Все ПО периферийного модуля разбито на две части (рис. 2.43): начальный загрузчик и системное ПО (собственно эту часть ПО имеет возможность обновлять разработчик или пользователь). Передача и сохранение обновления реализуется через специально созданный программный интерфейс работы с хранилищем обновлений (SSI).

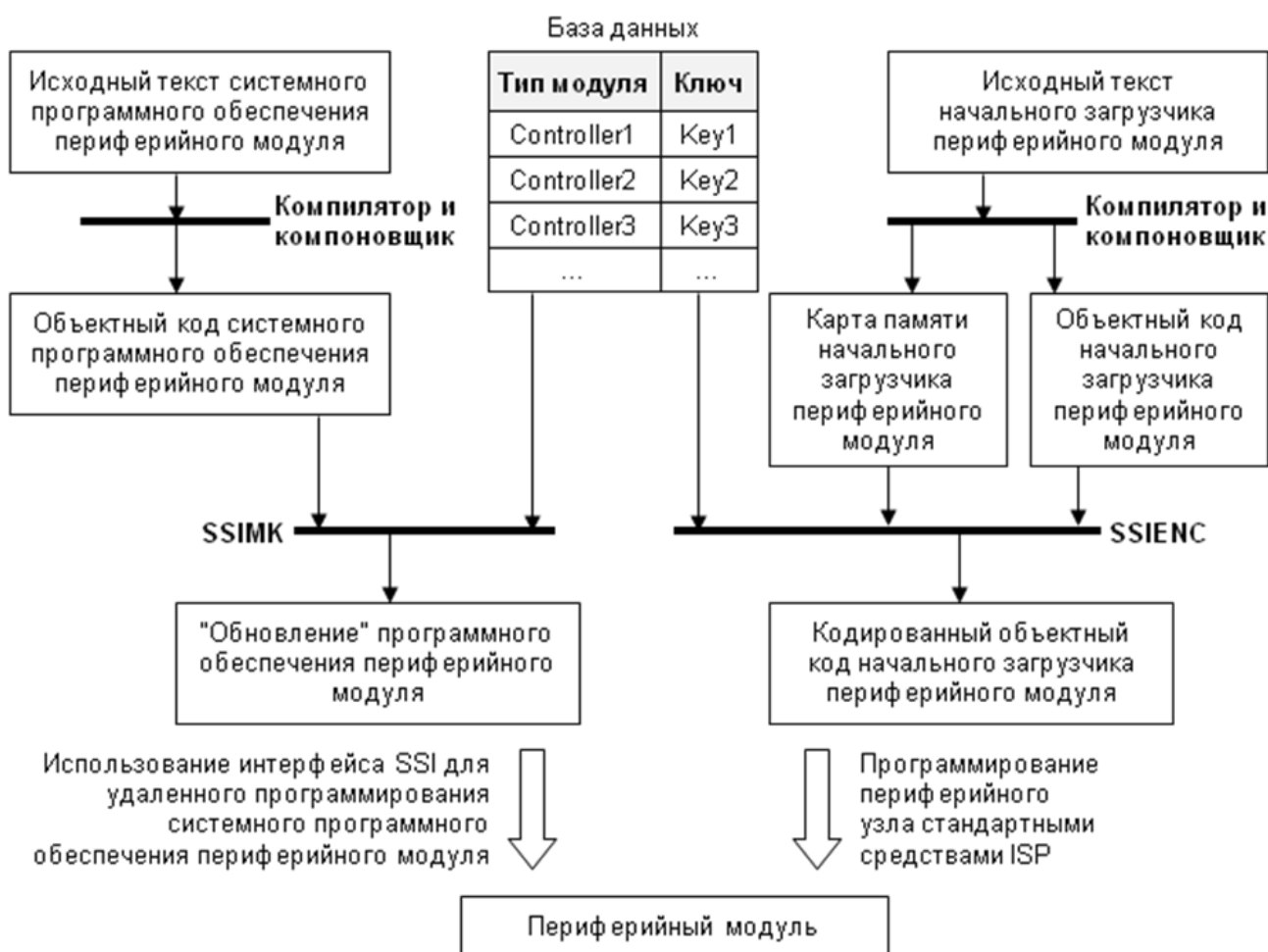


Рис. 2.42. Инструментальная модель автоматизированного комплекса обновления ПО РИУС

Этот интерфейс содержит всего 5 функций (SsiUnlock, SsiAllocate, SsiStoreItem, SsiGetItemCount, SsiLock), но при этом полностью обеспечивает решение поставленной задачи. Интерфейс SSI экспортируется через собственный инструментальный канал узла (реализуется в начальном загрузчике) и через целевой протокол сети CAN (реализуется в системном ПО).

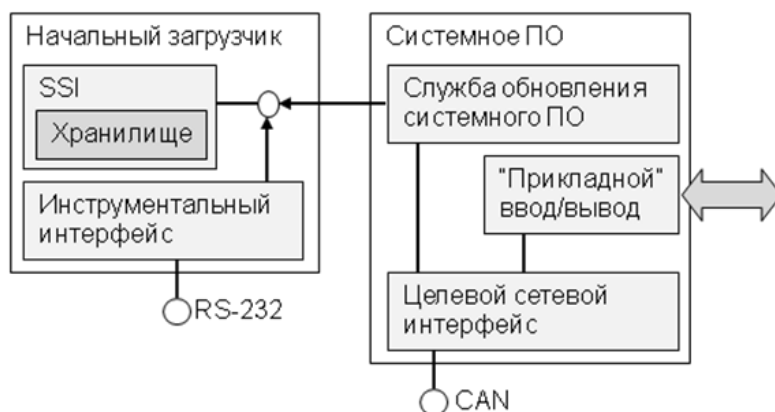


Рис. 2.43. Структура ПО периферийного модуля

Технические характеристики периферийных модулей, относящиеся к работе механизмов обновления ПО, приведены в табл. 2.17.

Таблица 2.17. Технические характеристики периферийного модуля

Объем кода обновляемого программного обеспечения	56 кб
Объем кода начального загрузчика	4 кб
Объем резидентной части загрузчика в RAM	200 б
Объем хранилища	64 кб
Типичный объем обновления	10-20 кб
Задержка при рестарте модуля: - с декодированием обновления - без декодирования обновления	15-20 сек нет

Передача обновлений периферийным модулям происходит в широкополосном режиме, что очень важно, так как в системе обычно присутствует много однотипных модулей. Передача обновлений в КМС осуществляется в наименее приоритетном режиме, что практически не влияет на временные параметры выполнения системой целевой задачи. Служба, реализованная в центральном вычислителе, позволяет в реальном масштабе времени в полуавтоматическом режиме отслеживать версии ПО периферийных модулей и производить при необходимости процедуру загрузки обновлений. Разработанная технология обновления программного обеспечения может быть легко реализована в различных КМС.

2.4.5 Инструментальный комплекс МЗР

Инструментальная система МЗР проектировалась как развитие ряда продуктов на основе диалектов языка Форт для решения задач отладки прототипов и опытных образцов ВсС, в том числе, распределенных [9, 12]. В проектировании учитывались следующие требования:

- решение комплекса проблем, связанных с отладкой, настройкой, тестированием и программированием ВсС;
- интеграция различных инструментальных программ в рамках одной среды, в том числе:
- загрузка программного обеспечения в целый спектр контроллеров;
- эмуляция терминалов различных типов;
- поддержка автоматизированных систем тестирования;
- преобразование файлов различных форматов и др.;

- объединение всех инструментальных программ через механизм скриптового программирования;
- предоставление Framework для реализации новых инструментальных программ;
- система должна быть расширяемой, надежной, обладать свойством кроссплатформенности;

Дополнительно при проектировании инструментальной системы МЗР были учтены нефункциональные требования:

- ориентация на компактные группы разработчиков ВсС, которые включают высококвалифицированных специалистов;
- малый срок разработки новых инструментальных средств (влияет на баланс функциональности и сервиса).



Рис. 2.44. Структура МЗР

Инструментальная система МЗР состоит из следующих блоков (рис. 2.44):

- интерпретатор скриптового языка (на основе языка FORTH);
- промежуточный уровень, обеспечивающий связь инструментальных функций и интерпретатор скриптового языка;
- набор инструментальных функций, реализующих различные инструментальные задачи;
- уровень абстракции от типа сетевой подсистемы;
- различные сетевые подсистемы, используемые инструментальными функциями (последовательные каналы, сокеты, именованные каналы).

Более подробно организация основных подсистем МЗР представлена ниже. Там же показано разделение представления системы по аспектам проектирования.

2.4.5.1 Сетевая подсистема РМЗР для связи с контроллерами

Сетевая подсистема РМЗР предназначена для решения инструментальных задач. Он ориентирован на реализацию в микропроцессорных устройствах с ограниченными вычислительными ресурсами (с небольшим быстродействием и объемом памяти данных, таких как микроконтроллеры семейств MCS51, PIC micro, Fujitsu FM216 и т.п.).

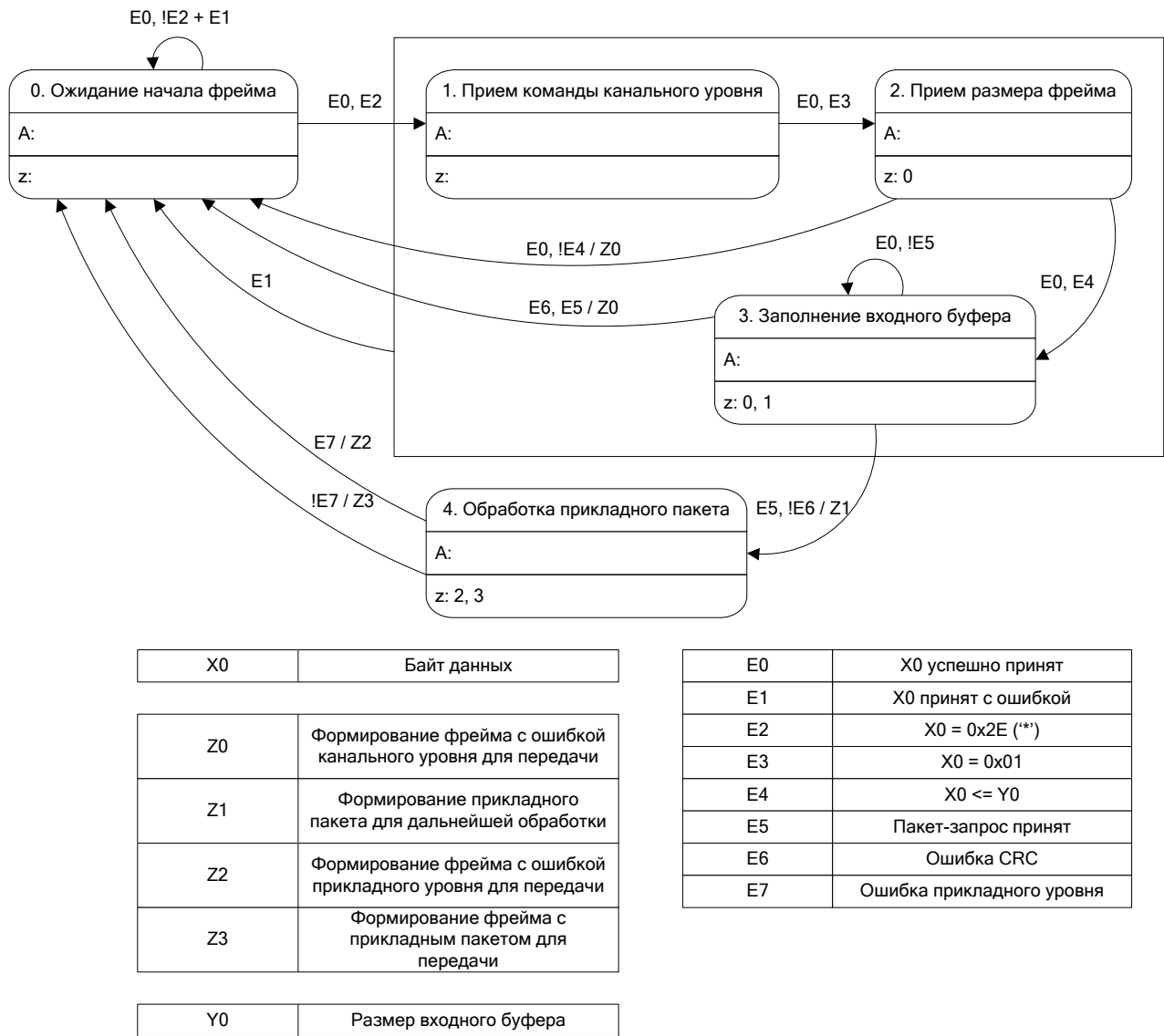


Рис. 2.45. Конечный автомат канального уровня подчиненного устройства

В соответствии с моделью OSI стек протоколов, реализованный в РМЗР, содержит протоколы только двух уровней:

- канальный уровень – пакетный протокол CP_P (Channel Protocol, Packet);

- прикладной уровень – пакетный протокол AP_P (Application Protocol, Packet).

На рис. 2.45 представлен пример автоматной модели протокола канального уровня подчиненного устройства.

Остальные уровни модели не представлены в силу простоты взаимодействия. РМЗР поддерживает модель взаимодействия MASTER-SLAVE. В обмене участвуют два устройства: инструментальная машина (ПК) и целевая система. При этом роль подчиненного устройства отводится целевой системе.

На канальном уровне данные передаются в виде фреймов (пакетов). Обмен происходит по принципу «ЗАПРОС-ОТВЕТ»: каждому запросу, сделанному мастером на канальном уровне, соответствует ответ целевой системы. При отсутствии ответа или при ошибочном ответе производятся перезапросы. Содержимое фреймов защищается циклическим кодом.

Канальный уровень инкапсулирует в себя прикладной пакет. Действия канального уровня с прикладным пакетом определяются полем КОМАНДА. Если подчиненное устройство корректно обработало пакет ЗАПРОС, то оно в пакете ОТВЕТ вернет код команды, т.е. копию поля КОМАНДА. В противном случае подчиненное устройство в поле КОМАНДА вернет код ошибки. Коды ошибок и команд отличаются старшим битом.

На прикладном уровне целевая система представлена в виде множества "целей" (рис. 2.46). Цель можно трактовать как виртуальную машину с некоторым набором команд, поставляемых вместе с данными через прикладной уровень протокола обмена. Наборы команд каждой из целей могут отличаться друг от друга, но возможны и пересечения.

Базовым вариантом цели прикладного уровня является представление целевой системы в виде логического адресного пространства некоторой адресуемой памяти. Команды, поддерживаемые этой памятью, переносятся в команды цели. Эти команды позволяют читать данные (read) и записывать данные (write). Для работы с некоторыми типами памяти требуются команды стирания данных (erase, erase_sector). Команды работы с памятью формируют архитектуру памяти. Эта архитектура применима к микроконтроллерам, а команды работы с памятью являются ее инструкциями. В этом случае никто не мешает расширить перечень инструкций. Если память является частью машины Фон-Неймана, то естественной выглядит команда передачи управления (jmp) или сброса (reset). При необходимости можно расширять и настраивать протокол для различных приложений путем создания новых целей.

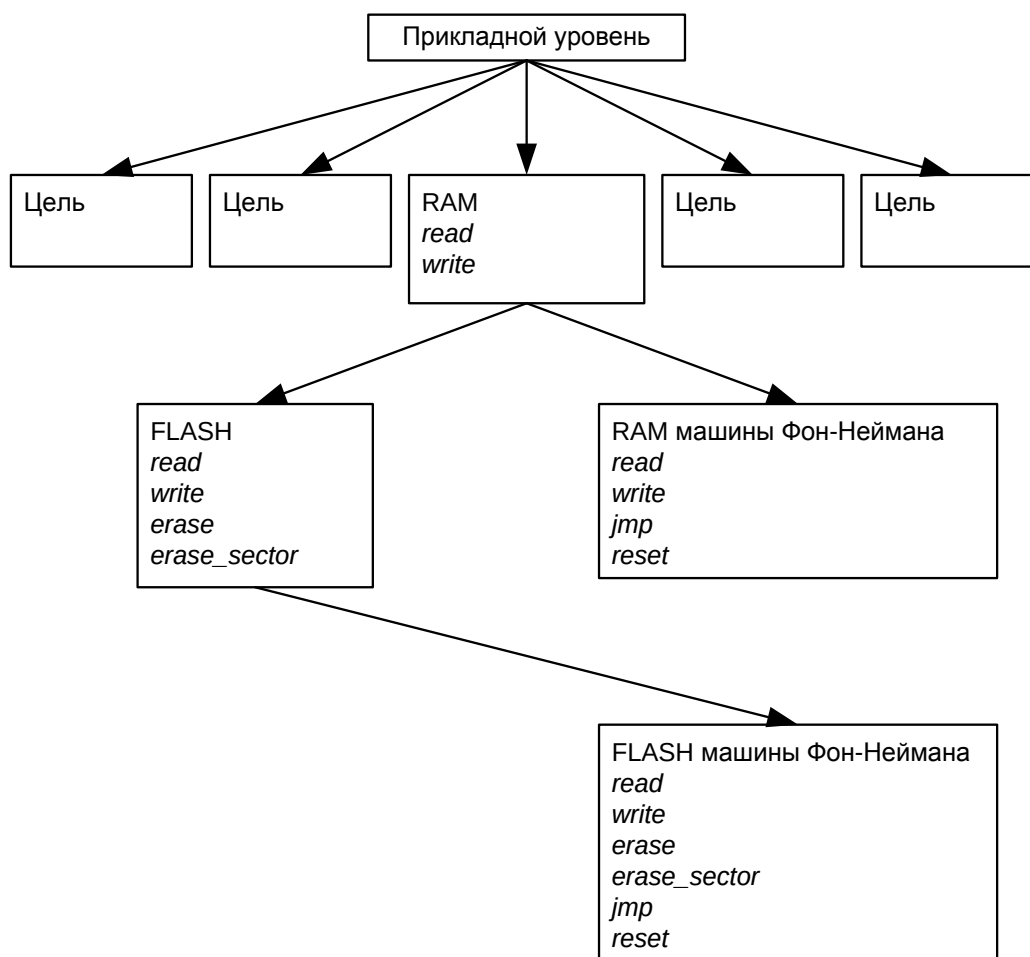


Рис. 2.46. Иерархия целей

2.4.5.2 Подсистема создания загрузочных модулей

Загрузочный модуль – файл, содержащий всю необходимую информацию для программирования различных устройств памяти контроллера (рис. 2.47). В контроллере загрузочный модуль хранится в специально отведенном устройстве памяти.

Контрольный блок предназначен для хранения информации, необходимой для инициализации различных устройств памяти контроллера после старта. В контрольном блоке хранится программа, позволяющая производить операции с целями в рамках идеологии РМЗР. Контрольный блок защищен CRC.

В целевой системе работает интерпретатор команд, производящий различные операции над принятыми блоками (копирование блоков памяти, программирование различных устройств энергонезависимой памяти, подсчет CRC и т.п.).

Блок данных содержит несколько бинарных модулей, полученных в процессе компиляции, с помощью различных инструментальных средств. В виде бинарного модуля могут быть представлены как программы, так и конфигурационные данные. Блок данных защищен CRC. По факту

возникновения ошибки действие интерпретатора команд прекращается и вызывается обработчик ошибки.

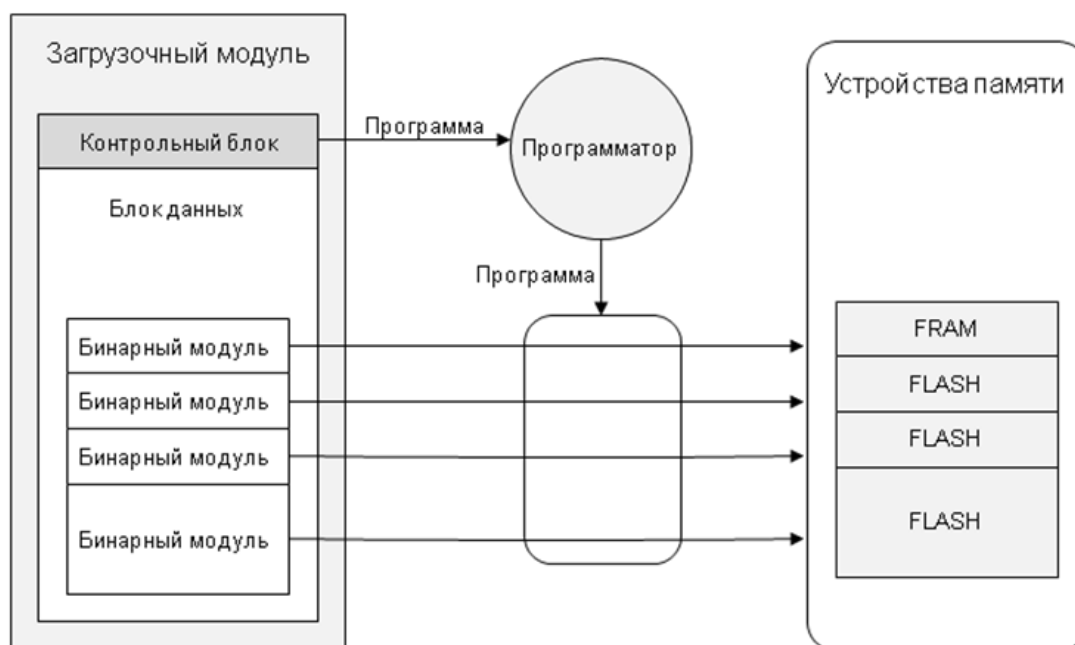


Рис. 2.47. Доставка загрузочного модуля в целевую систему

2.4.5.3 Мультитерминал

Мультитерминал предназначен для исполнения форт-команд интерпретатора МЗР, посланных удаленным устройством по последовательному каналу.

Мультитерминал использует дополнительный драйвер последовательного канала и работает независимо от команд Openchannel, Closechannel, wsio, rsio, различных загрузчиков и функций для работы с протоколом РМЗР.

После запуска мультитерминал отображает все принятые через коммуникационный порт символы. При нажатии на различные клавиши мультитерминал отправляет коды нажатых клавиш по последовательному каналу. При появлении специально сформированной ESC-последовательности мультитерминал извлекает из нее команду от удаленного устройства и при удачной проверке CRC передает ее интерпретатору форт-команд МЗР.

2.4.5.4 Элементы аспектного проектирования в МЗР

Аспект расширяемости

Аспект расширяемости являлся одним из важнейших аспектов при создании МЗР. Аспект расширяемости реализован на верхнем уровне самим используемым Framework (множество инструментальных функций объединяется интерпретатором скриптового языка). На нижнем уровне расширяемость обеспечивается простыми механизмами передачи параметров через стек и входной поток, принятыми в языке FORTH.

Аспекты сложности и повторного использования

В связи с тем, что МЗР необходимо дополнять новыми функциями в процессе работы над новыми проектами, система должна быть простой (принцип KISS). Упрощение системы производится созданием фреймворка, позволяющего создавать сравнительно сложные инструментальные системы из простых компонентов, объединенных скриптовым языком программирования. Необходимо заметить, что простота фреймворка позволила получить первую работающую версию инструментальной системы всего за несколько дней. Требование простоты обеспечивается в инструментальной системе отказом от графического интерфейса пользователя в пользу текстовой командной строки.

Простота реализации – один из главных аргументов в пользу выбора языка FORTH. В связи с тем, что в данной системе не предполагается написание очень сложных скриптовых программ (это доказано практикой, на протяжении десятка лет), язык FORTH позволяет легко решать различные задачи.

На начальных этапах реализация МЗР изобиловала использованием объектно-ориентированных особенностей языка C++. Практика показала, что использование сложных иерархий классов отрицательно сказывается на повторном использовании, в связи с чем последующие реализации МЗР были очень сильно упрощены в сторону структурного программирования.

Аспект переносимости

Как показала многолетняя практика, многим заказчикам требуется инструментарий, работающий на самых различных платформах (Windows, Unix, Mac OS и так далее). Обеспечение переносимости в инструментальной системе выполняется отказом от графического интерфейса пользователя в пользу текстовой командной строки и выполнения реализации ядра и большинства инструментальных функций в рамках стандарта POSIX. Такое решение позволяет с минимальными переделками быстро получить реализации инструментальной системы для MS Windows, Cygwin, Linux и Mac OS. Уровневая архитектура системы позволяет легко отделить части, зависящие от особенностей конкретной операционной системы.

Выводы

- 1 В терминах системы архитектурных вычислительных абстракций и ОСМВ представлен ряд специализированных виртуальных машин NL, перекрывающих широкий диапазон аппаратных и программных реализаций с различной производительностью и конфигурируемой функциональностью.
- 2 В качестве примеров использования системы архитектурных абстракций и в частности понятия проектного пространства ВсС рассмотрены модель актуализации вычислительного процесса ВсС и организация СППП на основе архитектур машин NL.
- 3 Семейство виртуальных машин NL должно быть обеспечено инструментальными средствами комплексирования, конфигурирования и программирования. Разнообразие архитектур машин NL, идеология инструментария и успешный опыт применения во многих проектах подтвердили эффективность представленных методов и средств проектирования ВсС.
- 4 В пособии рассмотрена микроархитектура машины NL3, приведена технология работы с DPU NL3, показана организация автоматизированной инструментальной цепи СППП на примере решения задач приборного комплекса MiniLab.
- 5 Дано понятие инструментального аспекта и инструментальной модели проектирования ВсС. Инструментальный аспект определяет уровень сложности доступных коллективу разработок и является важнейшим фактором успешного проведения процесса проектирования. За счет повышения степени повторного использования инструментария можно значительно повысить эффективность проектирования.
- 6 Приведены примеры инструментальных решений ВсС, демонстрирующие эффективность проектирования на основе описанной методологии. Показаны базовые механизмы организации инструментального комплекса вложенной отладки распределенных ВсС. На основании распределенности инструментальной модели приведены понятия непосредственной, прямой (удаленной) и вложенной отладки ВсС.

Список сокращений и обозначений

АСМ	аспектная модель
ВС	вычислительная система
ВсС	встраиваемая или встроенная ВС, embedded system
ВВМ	виртуальная вычислительная машина
ВПО	встроенное программное обеспечение
ИУС	информационно-управляющая система
МоС	модель вычислений
ОС	операционная система
ОСРВ	операционная система реального времени
ПЛИС	программируемая логическая интегральная схема
ПК	персональный компьютер
ПО	программное обеспечение
РИУС	распределенная ИУС
РМВ	реальный масштаб времени
САПР	система автоматизированного проектирования
СБИС	сверхбольшие интегральные схемы
СРВ	система реального времени
ТЗ	техническое задание
УСО	устройство сопряжения с объектом
ЯВУ	язык высокого уровня
API	application programming interface
DFD	data flow diagram
DT	Design-Time
FSM	finite state machine
HW	hardware
IAP	in-application programming
ICP	in-circuit programming
ISP	in-system programming
RT	Run-Time
SOC, СнК	System-On-a-Chip, система на кристалле
SW	software

Литература

1. Балашов Е.П., Пузанков Д.В. Проектирование информационно-управляющих систем. М. Радио и связь. 1987.
2. Баранов С.И. Синтез микропрограммных автоматов. Л. Энергия, Ленингр. отд-ние. 1979.
3. Барский А.Б., Шилов В.В. Потокковая вычислительная система: программирование и оценка эффективности. М. Новые технологии. 2003. 24 с.
4. Варшавский В.И., Розенблюм Л.Я., Цирлин Б.С. и др. Автоматное управление асинхронными процессами в ЭВМ и дискретных системах. М. Наука. 1981.
5. Воеводин В.В. Информационная структура алгоритмов. М. Изд-во МГУ. 1997. 139 с.
6. Воеводин В.В. Математические модели и методы в параллельных процессах. М. Наука. 1986. 296 с.
7. Гавриков В.О., Григорьев В.В., Платунов А.Е., Шубинский И.Б. Микропроцессорная система диспетчерской централизации «Тракт» нового поколения. // Информационные технологии на железнодорожном транспорте: докл. конф. СПб. 1997. С. 163–173.
8. Гавриков В.О., Платунов А.Е., Никифоров Н.Л. Комплекс технических средств для систем железнодорожной автоматики. // Автоматика, телемеханика и связь. № 11. 1998. С. 5–10.
9. Ключев А.О. Методы и инструментальное обеспечение разработки распределенных информационно-управляющих систем с программируемой архитектурой. // Тезисы кандидатской диссертации. СПб ГИТМО(ТУ). Санкт-Петербург. 1999.
10. Ключев А.О., Кустарев П.В., Платунов А.Е. Распределенные системы управления. // Сб. тезисов ДИМЭБ. СПб. 1997. С. 216–217.
11. Ключев А.О., Кустарев П.В., Платунов А.Е., Скорубский В.И., Стародубцев Э.В. Высоконадежный управляющий модуль М386.2. // Научно-технические разработки СПб ГИТМО(ТУ). СПб. СПб ГИТМО (ТУ). 2001. С. 218–219.
12. Ключев А.О., Платунов А.Е. Встроенные инструментальные средства современных микроконтроллеров. // Электронные компоненты. № 7. 2002. С. 94–97.
13. Ключев А.О., Постников Н.П. Технология сквозного проектирования информационно-управляющих систем. // Тезисы докладов XXX научно-технической конференции ППС. СПб. ГИТМО(ТУ). 1999. 75 с.
14. Ковязин Р.Р., Постников Н.П. Создание локальных регуляторов на базе виртуальной машины с динамическим набором инструкций. // Научно-технический вестник СПбГУ ИТМО. Выпуск 32. Информационные

- технологии: теория, методы, приложения. СПб. СПбГУ ИТМО 2006. С. 55–62.
15. Ковязин Р.Р., Постников Н.П. Модель DPU проблемно-ориентированного процессора NL3. // Сборник тезисов докладов конференции молодых ученых, Выпуск 5. СПб. СПбГУ ИТМО. 2010. С. 18–19.
 16. Ковязин Р.Р., Постников Н.П. Разработка проблемно-ориентированных процессоров. // Сборник тезисов докладов конференции молодых ученых. Выпуск 5. СПб: СПбГУ ИТМО. 2010. С. 16–17.
 17. Лукичев А.Н. Временные характеристики функциональных блоков при дискретно-событийном моделировании встроенных систем. // Сборник докладов ИММОД-2007. Том 1. СПб. ФГУП ЦНИИ технологии судостроения. 2007. С. 172–176 .
 18. Лукичев А.Н. Денотативно-объективная модель вычислений для встроенных систем. // Автореферат диссертации на соискание ученой степени кандидата технических наук, Санкт-Петербург, 2008.
 19. Лупанов О.Б. Асимптотические оценки сложности управляющих систем. М. Изд-во МГУ. 1984.
 20. Официальный сайт ООО «ЛМТ». URL: <http://lmt.ifmo.ru>.
 21. Платунов А.Е. Архитектурные абстракции в технологии сквозного проектирования встроенных вычислительных систем. // Науч.-техн. вестн. СПб ГИТМО (ТУ). СПб. СПб ГИТМО (ТУ). Вып. 6: Информационные, вычислительные и управляющие системы. 2002. С. 76–83.
 22. Платунов А.Е. Встраиваемые системы управления. // Control Engineering Россия – Т43, №1, 2013, с.16-24
 23. Платунов А.Е., Постников Н.П. Высокоуровневое проектирование встраиваемых систем. Часть 1. Учебное пособие. СПб.: НИУ ИТМО, 2011. – 121 с
 24. Платунов А.Е., Постников Н.П. Единое проектное пространство плюс аспектная технология – перспективная парадигма проектирования встраиваемых систем. // Научно-технический вестник СПбГУ ИТМО. Вып. 11. Актуальные проблемы анализа и синтеза сложных технических систем. СПб. СПбГУ ИТМО. 2003. С.121–128.
 25. Платунов А.Е., Постников Н.П. Перспективы формализации методов проектирования встроенных систем. // Электронные компоненты. № 1. 2005. С. 24–29.
 26. Платунов А.Е., Постников Н.П. Формализация архитектурного проектирования информационно-управляющих систем. // Тезисы докладов XXXI научно-технической конференции ППС. СПб. ГИТМО(ТУ). 2000. С. 122.
 27. Платунов А.Е., Постников Н.П., Чистяков А.Г. Механизмы граничного сканирования в неоднородных микропроцессорных системах. // Chip News. № 10. 2000. С.8–13.
 28. Постников Н.П. Динамические инструментальные компоненты. Научно-технический вестник СПбГИТМО(ТУ). // Информационные,

- вычислительные и управляющие системы. СПб. СПбГИТМО(ТУ). № 6. 2002. С.83–88.
29. Постников Н.П. Обновление программного обеспечения распределенных микропроцессорных систем. // Компоненты и технологии. №3. СПб. 2004, С.142–144.
30. Райли Д. Абстракция и структуры данных. Вводный курс. Пер. С англ. М. Мир. 1993. 752 с.
31. Топорков В.В. Функциональность недетерминированной модели распределенных вычислений. // Информационные технологии. №12. 2001. С. 2–5.
32. Топорков В.В. Реализуемость потоковых моделей распределенных программ. // Программирование. №5. 2001. С.18–25.
33. Шалыто А.А. SWITCH – технология. Алгоритмизация и программирование задач логического управления. СПб. Наука. 1998. 628с.
34. Шеннон К. Работы по теории информации и кибернетике. М. Изд-во иностр. литер. 1963.
35. Agha G.A. Actors: A Model of Concurrent Computation in Distributed Systems // The MIT Press Series in Artificial Intelligence. MIT Press. 1986. 204 p.
36. Application Note 155. 1-Wire Software Resource Guide Device Description // Maxim Integrated Products. 2008. URL: <http://www.maxim-ic.com/app-notes/index.mvp/id/155>.
37. Berry G., Gonthier G. The Esterel synchronous programming language: Design, semantics, implementation // Science of Programming. 1992. Vol. 19. № 2. P. 87–152.
38. Boulytchev D., Lomov D. An Empirical Study of Retargetable Compilers. In Perspectives of System Informatics, Springer Berlin. Heidelberg. 2001.
39. Brooks C., Lee E.A., Liu X., Neuendorffer S., Zhao Y., Zheng H. Heterogeneous Concurrent Modeling and Design in Java. // Technical Memorandum UCB/ERL M04/27 - University of California. Berkeley. July 29, 2004. Vol. 1, 2, 3.
40. Buck J.T., Ha S., Lee E.A., Messerschmitt D.G. Ptolemy: A mixed-paradigm simulation/prototyping platform in C++. // In Proceedings of the C++ At Work Conference, Santa Clara, CA. November 1991.
41. Clarke E.M., Emerson E.A., Sistla A.P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. // ACM Trans. on Programming Languages and Systems. Vol. 8. April 1986. P. 244–263.
42. Clifford E. Cummings. Correct Methods For Adding Delays To Verilog Behavioral Models. HDLCON. 1999.
43. Clifford E. Cummings. Verilog Coding Styles for Improved Simulation Efficiency. International Cadence User Group Conference. October 5-9, 1997.
44. Densmore D., Passerone R., Sangiovanni-Vincentelli A. A Platform-Based Taxonomy for ESL Design. IEEE Design and Test of Computers, September 2006.
45. Edwards S. A. The Specification and Execution of Heterogeneous Synchronous Reactive Systems. // PhD thesis. University of California. Berkeley. 1997.

46. Edwards S., Lavagno L., Lee E.A., Sangiovanni-Vincentelli A. Design of embedded systems: Formal models, validation, and synthesis. // Proceedings of the IEEE. March 1997.
47. Elaine Cheong. Design and Implementation of TinyGALS: A Programming Model for Event-Driven Embedded Systems. // Memorandum No. UCB/ERL M03/14 – Electronics Research Laboratory, College of Engineering, University of California. Berkeley. May 23, 2003.
48. Gerard Berry. The Foundations of Esterel. // Proof, Language, and Interaction: Essays in Honour of Robin Milner. Editors: Gordon Plotkin, Colin Stirling, Mads Tofte. MIT Press. 2000.
49. Gérard Berry. The Foundations of Esterel. Ecole des Mines de Paris. INRIA. 1999.
50. Halbwachs N., Caspi P., Raymond P., Pilaud D. The synchronous data flow programming language Lustre. // Proceedings of the IEEE. September 1991.
51. Hardware-Software Codesign. // IEEE Design & Test of Computers, January-March 2000. P. 92–99.
52. Hatley D.J., Pirbhai I.A. Strategies for Real-Time System Specification. Dorset House. 1988.
53. IEEE Standard VHDL Language Reference Manual. Institute of Electrical and Electronic Engineers, Inc. December 29, 2000.
54. Information Processing Systems. OSI Reference Model. The Basic Model. 1994.
55. Information Technology. Open Systems Interconnection. Basic Reference Model: Naming and Addressing. 1994.
56. Jie Liu. Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems. // PhD thesis – university of California at Berkeley. 2001.
57. Lavagno L., Sangiovanni-Vincentelli A., Sentovich E. Models of Computation for Embedded System Design. // NATO ASI Proceedings on System Synthesis. Il Ciocco, Italy. August 1998.
58. Lee E.A. Concurrent Models of Computation for Embedded Software. // UCB ERL Technical Memorandum M05/2. Department of Electrical Engineering and Computer Sciences, University of California. Berkeley, January 4, 2005.
59. Lee E.A. Model-Driven Development – From Object-Oriented Design to Actor-Oriented Design. // Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop). Chicago. Sept. 24, 2003.
60. Lee E.A. Modeling Concurrent Real-Time Processes Using Discrete Events. // Annals of Software Engineering. Special Volume on Real-Time Software Engineering. 1998.
61. Lee E.A., Parks T.M. Dataflow process networks. // Proceedings of the IEEE, May 1995. URL: <http://ptolemy.eecs.berkeley.edu/papers/processNets>.
62. Lee E.A., Sangiovanni-Vincentelli A. A denotational framework for comparing models of computation. IEEE Transactions on CAD, December 1998.
63. Lee E.A., Sangiovanni-Vincentelli A. Comparing Models of Computation. Proceedings of ICCAD. San Jose, California, USA. November 10-14, 1996.

64. Maciel P., Barros E., Rosenstiel W. A Petri Net Model for Hardware/Software Codesign. // In Design Automation for Embedded Systems. Vol. 4. October 1999. P. 243–310.
65. Martin G., Chang H., et al. Surviving the SOC Revolution: A Guide to Platform Based Design. Kluwer Academic Publishers. September 1999.
66. Peterson J. L. Petri Net Theory and the Modeling of Systems. // Prentice-Hall Inc., Englewood Cliffs, NJ. 1981.
67. Petru Eles. System Design and Methodology: Modeling and Design of Embedded Systems. Institutionen för Datavetenskap (IDA). Linköpings Universitet. 2004.
68. Programmable Controllers – Part 3: Programming Languages. IEC 61131-3, Ed. 2.0. 2003
69. Ptolemy II. URL: <http://ptolemy.berkeley.edu/ptolemyII>.
70. Sangiovanni-Vincentelli A., Lee E.A. A framework for comparing models of computation. // IEEE Trans. Computer-Aided Design Integrated Circuits. December 1998.
71. Semiconductor Reuse Standard V2.0. Motorola Inc. 1999.
72. Sgroi M., Lavagno L., Sangiovanni-Vincentelli A. Formal Models for Embedded System Design. // IEEE Design & Test of Computers. April-June 2000. P. 2–15.
73. Shaw M. We can teach software better. // Computing Research News 4(4):2-12. September 1992.
74. Thomas M. Parks. Bounded Scheduling of Kahn Process Networks. // Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California. Berkeley. December 1995.
75. Wirth N. Hardware Compilation: Translating Programs into Circuits. // IEEE Computer. No. 31(6). 1998. P. 25–31.
76. Wolf W. Object-oriented cosynthesis of distributed embedded systems. // ACM Transactions on Design Automation of Electronic Systems. July 1996.
77. Xiaojun Liu, Edward A. Lee. CPO semantics of timed interactive actor networks. // Technical Report UCB/EECS-2007-131. EECS Department, University of California. Berkeley. November 2007.
78. Xiaojun Liu. Semantic Foundation of the Tagged Signal Model. // Technical Report UCB/EECS-2005-31. Electrical Engineering and Computer Sciences. University of California at Berkeley. 2005.
79. Yang Zhao. A Model of Computation with Push and Pull Processing. // Technical Memorandum UCB/ERL M03/51. Electronics Research Laboratory. University of California at Berkeley. December 16, 2003.



В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России с присвоением категории «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра ВТ НИУ ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России.

Традиционно основной упор в подготовке специалистов на кафедре делается на фундаментальную базовую подготовку в рамках общепрофессиональных и специальных дисциплин, охватывающих наиболее важные разделы вычислительной техники.

Кафедра является одной из крупнейших в университете. Учебными курсами и научно-исследовательскими работами руководят 8 профессоров и 13 доцентов. На кафедре обучаются более 500 студентов и 30 аспирантов.

Кафедра имеет собственные компьютерные классы и специализированные исследовательские лаборатории, оснащенные современной вычислительной и оргтехникой, уникальным инструментальным и технологическим оборудованием, измерительными приборами и программным обеспечением.

В 2007-2008 гг. коллективом кафедры была успешно реализована инновационная образовательная программа СПбГУ ИТМО по научно-образовательному направлению «Встроенные вычислительные системы».

Начиная с 2009 года кафедра вычислительной техники является активным участником реализации программы развития национального исследовательского университета ИТМО, вошла в состав крупнейшего в НИУ ИТМО научно-исследовательского центра «Интеллектуальные системы управления и обработки информации».

Алексей Евгеньевич Платунов

Николай Павлович Постников

**ВЫСОКОУРОВНЕВОЕ ПРОЕКТИРОВАНИЕ
ВСТРАИВАЕМЫХ СИСТЕМ
(часть 2)**

Учебное пособие

В авторской редакции

Редакционно-издательский отдел НИУ ИТМО

Зав. РИО

Н.Ф. Гусарова

Лицензия ИД № 00408 от 05.11.99

Подписано к печати

Заказ №

Тираж 100 экз.

Отпечатано на ризографе

Платунов А.Е., Постников Н.П.

ВЫСОКОУРОВНЕВОЕ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ

(Часть 2)

УЧЕБНОЕ ПОСОБИЕ



Санкт-Петербург
2013

Редакционно-издательский отдел
Санкт-Петербургского национального
исследовательского университета
информационных технологий, механики и
оптики
197101, Санкт-Петербург, Кронверкский пр., 49

