SystemC – AMS

Проектирование электронных систем с аналоговыми и смешанными сигналами

Руководство пользователя

Составил и перевел с английского
Алехин В.А.

Москва 2020

SystemC – AMC

Предисловие

Данное руководство пользователя составлено на основе материалов компаний Accellera и Coseda Technologies GmbH и предназначено в качестве вводного руководства для инженеров и архитекторов электронного уровня, которые хотели бы использовать расширения SystemC AMS для задач проектирования и проверки на системном уровне.

Основная цель - предоставить самообучающееся руководство по использованию расширений SystemC AMS, объяснив основы моделирования и примеры того, как начать проектирование на уровне системы AMS на более высоких уровнях абстракции. Предполагается, что пользователь имеет некоторые предварительные знания по моделированию и симуляции SystemC и C ++ в целом и знаком с проектированием и моделированием аналоговых / смешанных сигналов.

Изучив это руководство, читатель сможет начать использовать расширения SystemC AMS, и должен быть в состоянии:

• Ознакомьтесь с применимыми сценариями использования и требованиями расширений SystemC AMS.

• Понимать представленные модели вычислений и связанную с ними семантику выполнения.

• Использование языковых конструкций для создания моделей с дискретным и непрерывным временем на разных уровняхмабстракция.

• Объединение SystemC и расширений AMS для разработки системы со смешанным сигналом.

• Выполнять анализ во временной и частотной областях и отслеживать сигналы AMS.

Методология проектирования AMS, стиль моделирования и примеры, приведенные в данном руководстве пользователя, основаны на языковом стандарте Open SystemC Initiative AMS. Любая реализация симулятора, совместимая с этим Стандартом может быть использована для построения и выполнения этих примеров.

Этот материал является информативным руководством, предназначенным для разъяснения использования и предполагаемого поведения SystemC расширения AMS. Точное и полное определение расширений SystemC AMS стандартизировано в Справочном руководстве по языку AMS.

Составитель выполнил моделирование примеров и привел результаты.

Получить текущую версию программы SystemC AMS можно на сайте компании Accellera (http://accellera.org). В установочном каталоге Вы найдёте инструкцию по установке в Visual Studio и в Linux.

1. Глава 1. Введение

1.1. Мотивация

Существует растущая тенденция к более тесному взаимодействию между встроенными аппаратными / программными (HW / SW) системами и их аналоговой физической средой. Это приводит к системам, в которых цифровое HW / SW функционально переплетены с аналоговыми и смешанными блоками сигналов, такими как РЧ-интерфейсы, силовая электроника, датчики и приводы, как показано, например, в системе связи на рисунке 1.1. Такие системы называются Встроенные аналоговые / смешанные системы (E-AMS). Примерами систем E-AMS являются когнитивные радиостанции, сенсорные сети или системы для восприятия изображений. Задача разработки систем E-AMS заключается в понимании взаимодействия между HW / SW и аналоговой подсистемой и подсистемой со смешанным сигналом в архитектурном уровене. Это требует новых средств для моделирования и симуляции взаимодействующих аналоговых / смешанных подсистем и HW / SW подсистемы на функциональном и архитектурном уровне.

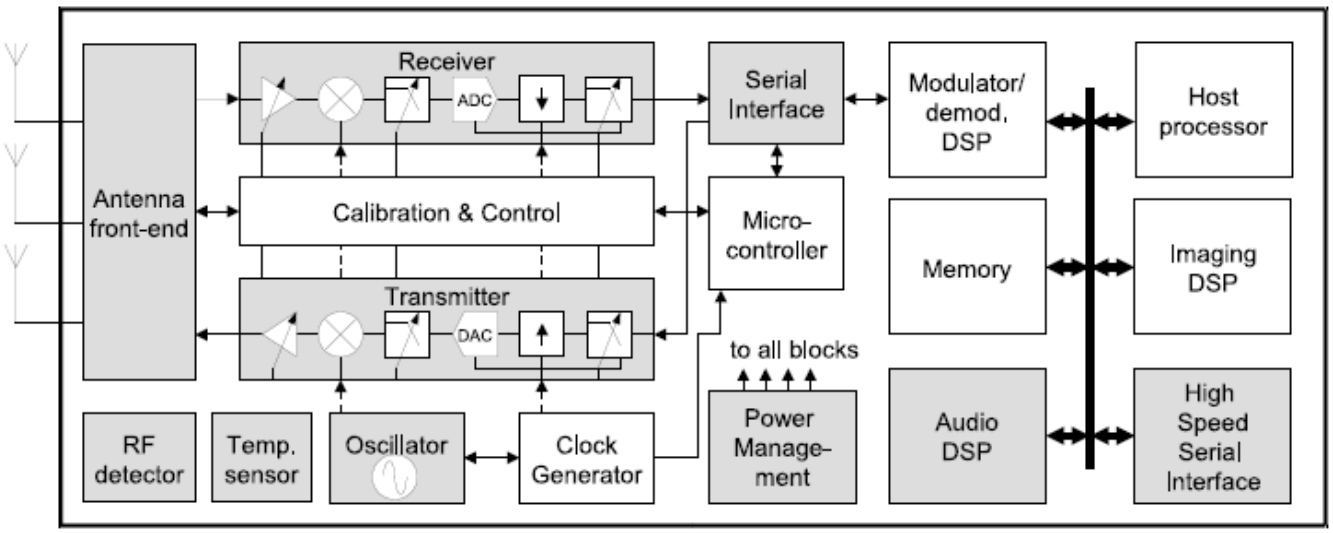


Рисунок 1.1. Система связи, пример встроенной аналоговой / смешанной архитектуры

SystemC поддерживает усовершенствование систем HW / SW вплоть до точного цикла, обеспечивая среду моделирования дискретных событий. Методология обобщенного моделирования коммуникации и синхронизации, построенная на этой платформе, также доступна: моделирование на уровне транзакций (TLM). Это позволяет дизайнерам выполнять абстрактное моделирование, симуляцию и проектирование систем HW / SW. Тем не мение, ядро моделирования SystemC не было разработано для моделирования и симуляции аналоговых систем с непрерывным временем и отсутствует поддержка методологии уточнения для описания аналогового поведения от функционального уровня до уровня реализации.

В ответ на потребности телекоммуникационной, автомобильной и полупроводниковой промышленности, AMS расширения введены на основе SystemC, чтобы обеспечить единую и стандартизированную методологию для моделирование E-AMS систем.

1.2. Расширения SystemC AMS

Расширения SystemC AMS основаны на стандарте языка SystemC IEEE 1666-2005 и определяют дополнительные языковые конструкции, которые вводят новую семантику выполнения и моделирование на уровне методологии системного проектирования и проверки систем со смешанными сигналами.

Определения классов, предоставляемые стандартом языка AMS, составляют основу для создания реализация библиотеки классов C ++, которая может использоваться в сочетании с IEEE 1666-2005 совместимым с реализацией SystemC. Такая реализация может использоваться для создания моделей системного уровня AMS и создания исполняемой спецификации, чтобы проверить и оптимизировать архитектуру системы AMS, изучить различные алгоритмы, и предоставить группе разработчиков программного обеспечения действительный виртуальный прототип всей системы AMS, включая также аналоговые функции. Для поддержки этих вариантов использования SystemC AMS расширения определяют необходимые формализмы моделирования для моделирования поведения на уровне системы AMS на разных уровни абстракции.

1.2.1. Варианты использования и требования

Как показано на рисунке 1.2, расширения SystemC AMS могут применяться для самых разных случаев использования, таких как:

- Исполняемая спецификация;
- Виртуальное прототипирование;
- Исследование архитектуры и
- Проверка интеграции.

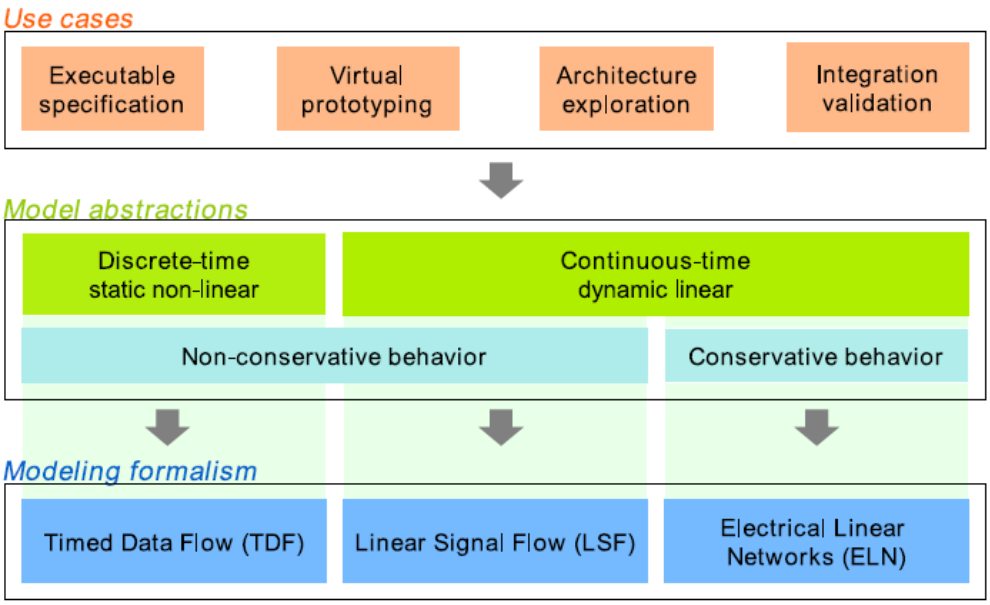


Figure 1.2. Use cases, model abstractions, and modeling formalisms

Рисунок 1.2. Варианты использования модельных абстракций и формализм моделирования

Исполняемая спецификация

Исполняемая спецификация создается для проверки правильности спецификации системных требований и создания исполняемого описания системы с использованием симуляции. Для этого варианта использования создаются модели на высоком уровене абстракции, которые не обязательно должны относиться к физической архитектуре или реализации системы. Поэтому модели называются функциональными или алгоритмическими моделями.

Расширения SystemC и AMS определяют как язык моделирования на уровне системы, так и их семантику исполнения для целей моделирования. Они полностью реализованы в виде библиотек C ++, которые связаны с скомпилированными моделями AMS для создания исполняемого описания системы. Этот целиком основанный на C ++ подход к моделированию обеспечивает уникальную гибкость, поскольку позволяет, например, легко интегрировать встроенные программное обеспечение, сторонние библиотеки и устаревший код в модели системы.

Виртуальное прототипирование

Вариант использования виртуального прототипирования нацелен на предоставление разработчикам программного обеспечения высокоуровневой модели, которая представляет аппаратную архитектуру и обеспечивает высокую скорость моделирования. Специально для E-AMS систем, в которых программное обеспечение или встроенное ПО взаимодействует напрямую с оборудованием AMS, важно взаимодействие с использованием расширения SystemC Translation-Modeling (TLM).

Использование моделирования потока временных данных для (сверх) дискретизированного непрерывного времени и поведения при обработке сигналов обеспечивает высокую скорость моделирования с соответствующей точностью. Таким образом, подсистема AMS может стать частью виртуального прототипа для дальнейшего развития подсистемы HW / SW.

Исследование архитектуры

Вариант использования исследования архитектуры оценивает, как определенные идеальные функции и алгоритмы во время фазы исполняемой спецификации могут быть отображены на предполагаемую архитектуру системы. Ключевые свойства архитектуры системы определены и должны соответствовать фактической требуемой функциональности.

Изучение архитектуры состоит из двух этапов: на первом этапе уточняется исполняемая спецификация, добавляя неидеальные свойства реализации, чтобы лучше понять их влияние на общее поведение системы. На втором этапе структура и интерфейсы архитектуры уточняются, чтобы получить более точную модель за счет введения архитектурных элементов и связи между этими элементами.

Проверка интеграции

После определения архитектуры и проектирования аналоговых и цифровых компонентов HW / SW эти компоненты будут интегрированы, и их правильность проверена в рамках всей системы. Для проверки правильности использования в этом случае интерфейсы всех подсистем должны быть точно смоделированы. Интерфейсы и типы данных, используемые в модели должны соответствовать физической реализации. Для аналоговых цепей это относится к электрическим узлам. В цифровых схемах, это относится к точным контактам шин. Для систем HW / SW могут подойти интерфейсы TLM.

1.2.2. Модельные абстракции

Расширения SystemC AMS добавляют новые методы абстракции для моделирования и симуляции на системном уровне систем AMS в существующую платформу SystemC. Абстракции модели, поддерживаемые расширением SystemC AMS основаны на хорошо известных методах абстрагирования аналоговых и смешанных сигналов. Как показано на рисунке 1.2, уровни абстракции различают поведение с дискретным временем и поведение с непрерывным временем и неконсервативное из консервативных описаний. Глава 7 представит доступные методы абстракции подробнее.

Описание с дискретным временем и с непрерывным временем

Моделирование в дискретном времени представляет сигналы (например, аудио- или видеопотоки) или физические величины (например, напряжения, токи и силы) как последовательности значений, определенных только в отдельные моменты времени. Значения могут быть либо действительными значениями, либо дискретными значениями (например, целыми или логическими значениями). Значения между временными точками формально не

определены, хотя принято считать их постоянными. Поведение тогда абстрагируется в качестве процедурных присвоений, включающих выборочные сигналы. Описание статической (алгебраической) нелинейности поведения (например, с использованием полиномов) поддерживается. Моделирование с дискретным временем особенно подходит для описания поведения, в котором преобладает обработка сигналов, для которой сигналы естественным образом дискретизируются. Это может быть также использовано для описания поведения непрерывного времени, при условии, что дискретная абстракция производит разумные приближения.

Моделирование в непрерывном времени становится ближе к физическому миру, поскольку сигналы и физические величины абстрагируются как вещественные функции времени. Время теперь рассматривается как непрерывное значение. Поведение тогда описывается с использованием математических уравнений, которые могут включать производные во временной области любого порядка (так называемые дифференциальные алгебраические уравнения (ОДУ) или обыкновенные дифференциальные уравнения (ОДУ)). Уравнения должны решается с помощью специального линейного или нелинейного решателя, который обычно требует сложных числовых или символических алгоритмов. Непрерывное моделирование особенно подходит для описания физического поведения, так как это может естественно объяснить динамические эффекты.

Неконсервативные и консервативные описания

Модели с непрерывным временем можно разделить на два класса: неконсервативные и консервативные модели.

Неконсервативное модели выражают поведение как направленные потоки сигналов или величин непрерывного времени, на которых применяются функции обработки, такие как фильтрация или интеграция. Нелинейные динамические эффекты могут быть правильно описаны, но взаимные эффекты и взаимодействия между блоками AMS, такие как импедансы или нагрузки, не являются естественно поддерживаемыми.

Консервативные модели - это наиболее подробные модели с непрерывным временем на уровне системы и схемы, так как законы сохранения энергии (законы Кирхгофа) должны быть выполнены. В результате решается система уравнений, которые являются более крупными и, возможно, более сложными, чем те, которые выводятся неконсервативными моделями.

1.2.3. Формализм моделирования

Расширения SystemC AMS определяют основные формализмы моделирования, необходимые для поддержки AMS поведенческого моделирования на разных уровнях абстракции. Эти формализмы моделирования реализуются с помощью различных моделей вычислений: синхронный поток данных (Timed Data Flow - TDF), линейный поток сигналов

(LSF) и электрические линейные сети (ELN).

Временной поток данных (TDF)

Семантика исполнения, основанная на TDF, вводит моделирование и моделирование в дискретном времени без накладных расходов на динамическое планирование, налагаемое ядром дискретных событий SystemC. Симуляция ускоряется за счет определения статического расписания, которое вычисляется до начала моделирования и которое выполняет функции обработки запланированных модулей TDF в соответствии с направлением потока данных.

Выборочные сигналы с дискретным временем, которые распространяются через модули TDF, могут представлять собой любой C ++ тип. Если, например, используется вещественный тип, такой как double, сигнал TDF может представлять напряжение или ток на данный момент времени. Комплексные значения могут использоваться для представления эквивалентного сигнала основной полосы частот. TDF моделирование представлено в главе 2.

Линейный поток сигналов (Linear Signal Flow - LSF)

Формализм Linear Signal Flow поддерживает моделирование непрерывного поведения, предлагая согласованный набор примитивных модулей, таких как сложение, умножение, интеграция или задержка. Модель LSF состоит из соединения таких примитивов с помощью вещественных сигналов во временной области, представляющих любые виды непрерывного количества времени. Модель LSF определяет систему линейных уравнений, которая решается в линейном DAE решателе. Моделирование LSF представлено в главе 3.

Электрические линейные сети (Electrical Linear Networks - ELN)

Моделирование электрических сетей поддерживается путем создания заранее определенных линейных сетевых примитивов, таких как резисторы или конденсаторы, которые используются в качестве макромоделей для описания непрерывных временных отношений между напряжениями и токами. Доступен ограниченный набор линейных примитивов и переключателей для моделирования электрического энергосберегающего поведения. Моделирование ELN представлено в главе 4.

1.2.4. Анализ во временной и частотной областях

Расширения SystemC AMS поддерживают анализ как во временной (переходной), так и в частотной областях, путем введение новой семантики исполнения и дополнительных функций для управления имитацией.

Моделирование во временной области может применяться к описаниям, сделанным с использованием моделей вычисления TDF, LSF или ELN. Анализ вычисляет поведение всей системы во временной области, возможно,

составленное из различных моделей вычислений, которые могут даже включать описания, определенные в области дискретных событий.

Семантика выполнения для моделирования во временной области моделей TDF, LSF и ELN описана в главе 2, 3 и 4 соответственно.

Моделирование в частотной области может применяться к одним и тем же описаниям, комбинируя различные модели вычислений, где анализ вычисляет поведение слабого сигнала в частотной области для общей системы. Помимо анализа в частотной области для слабого сигнала, также доступен анализ в шумовом диапазоне для слабого сигнала. Глава 5 опишет оба метода анализа более подробно.

Методы управления имитацией и трассировки сигналов для моделирования во временной и частотной областях представлены в главе 6. Кроме того, создание и базовая структура испытательных стендов объяснены в этой главе.

1.2.5. Языковая архитектура

Расширения SystemC AMS полностью совместимы со стандартом языка SystemC, как показано на рисунке 1.3. Стандарт языка AMS определяет семантику исполнения моделей TDF, LSF и ELN вычислений и даёт представление о лежащих в её основе технологий, таких как линейный решатель, планировщик и уровень синхронизации. В настоящее время интерфейсы и определения классов этих технологий определяется реализацией. Разработчик AMS (конечный пользователь) может использовать выделенные классы и интерфейсы для создания моделей TDF, LSF или ELN с использованием предопределённых модулей, портов, терминалов, сигналов и узлов.

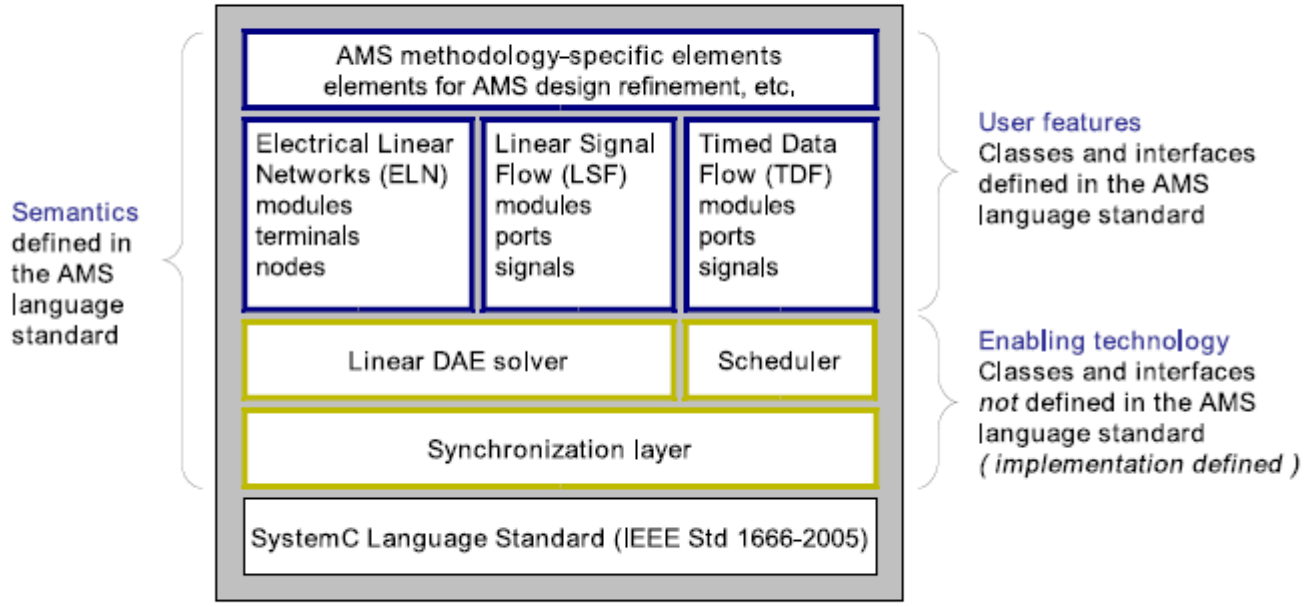
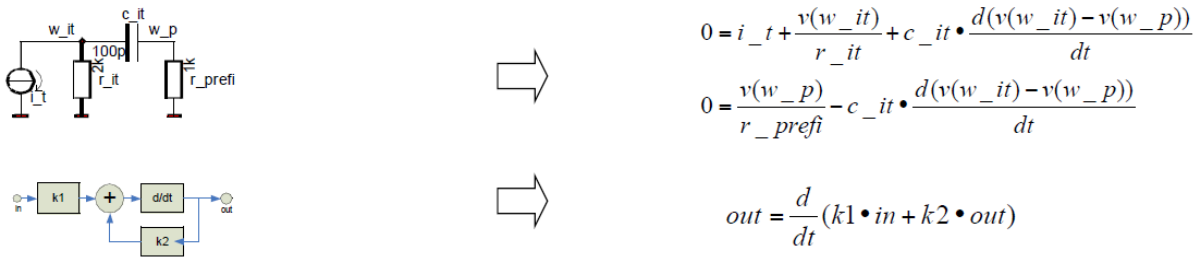


Figure 1.3. Architecture of the AMS language standard

Рисунок 1.3. Архитектура стандарта языка AMS

1.2.6. Чем отличаются аналоговые системы от цифровых систем?

Аналоговое уравнение не может быть решено с помощью коммуникации и синхронизации процессов.



$$0 = i\_t + \frac{v(w\_it)}{r\_it} + c\_it \bullet \frac{d(v(w\_it) - v(w\_p))}{dt}$$

$$0 = \frac{v(w\_p)}{r\_prefi} - c\_it \bullet \frac{d(v(w\_it) - v(w\_p))}{dt}$$

$$out = \frac{d}{dt}(k1 \bullet in + k2 \bullet out)$$

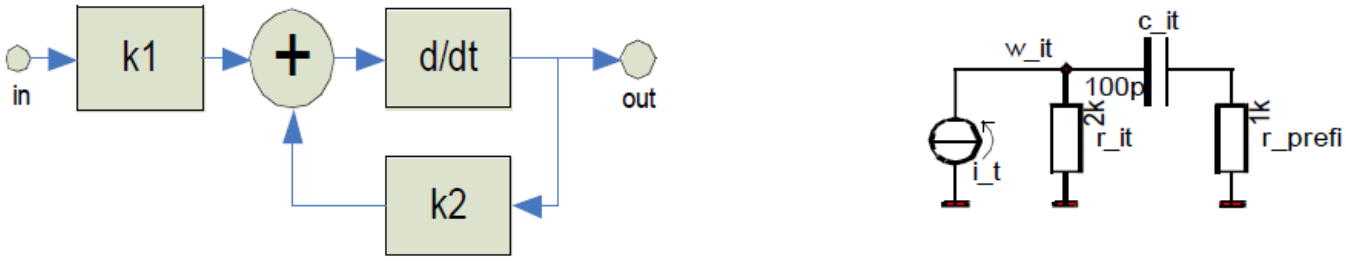
Рис. 1.4. Уравнения аналоговой системы

В общем, должна быть составлена система уравнений.

Состояние аналоговой системы постоянно меняется:

- значение между точками решения является непрерывным (линейным является только приближением первого порядка)
- значение для момента времени между двумя точками решения может быть оценено только после того, как была рассчитана вторая точка (в противном случае будет нестабильная экстраполяция)

1.2.7. Не консервативная и консервативная система



А) Б)

Рис. 1.5. Не консервативная (А) и консервативная (Б) система

В не консервативной системе выполняется следующее:

1. Абстрактное представление аналогового поведения;
2. График представляет собой непрерывное по времени (неявное) уравнение (системы)

В консервативной системе:

1. Схема представляет топологическую структуру моделируемой системы.
2. Узлы характеризуются двумя величинами: поперечным значением (например, напряжением) и продольным значением (например, током).
3. Для электрических систем применяются законы Кирхгофа (первый - KCL, и второй - KVL)
4. Для других физических областей применяются обобщённые версии законов Кирхгофа

1.2.8. Основы языка SystemC-AMS

Подробное изложение Стандарта расширения SystemC® AMS 2.0 можно найти в справочном руководстве по языку [2].

Краткий обзор основных конструкций имеется в Приложении к [1] и в приложении к данному пособию.

В этом разделе мы приведём лишь основные понятия, необходимые для начального понимания текстов программ.

1. Примитивный модуль представляет собой вклад уравнений в модель вычислений (MoC). Примитивы каждой модели вычислений MoC должны быть получены из определенного базового класса.
2. Канал, как правило, представляет собой ребро или переменную системы уравнений. Поэтому это не обязательно канал связи
3. Модули / каналы SystemC-AMS являются производными от базовых классов SystemC (sc_module, sc_prim_channel / sc_interface)
4. Нет разницы по сравнению с SystemC для иерархических описаний - они используют SC_MODULE / SC_CTOR

1.2.9. Имена символов и пространства имён

Реализация должна помещать каждый оператор и описание, определённые этим стандартом, в одно из следующих пространств имён: `sca_core`, `sca_tdf`, `sca_lsf`, `sca_eln`, `sca_ac_analysis` или `sca_util`.

Базовые классы базового языка должны быть помещены в пространство имён `sca_core`.

Для предопределённых моделей вычислений должны использоваться следующие пространства имён:

• Предопределённые классы для синхронизированного потока данных должны быть помещены в пространство имён `sca_tdf`.

• Предопределённые классы для линейного потока сигналов должны быть помещены в пространство имён `sca_lsf`.

• Предопределенные классы для электрических линейных сетей должны быть помещены в пространство имён `sca_eln`.

Предопределённые классы для анализа частотной области слабого сигнала должны быть помещены в пространство имён `sca_ac_analysis`. Утилиты должны быть размещены в пространстве имён `sca_util`.

Рекомендуется, чтобы реализация использовала вложенные пространства имён в `sca_core`, `sca_tdf`, `sca_lsf`, `sca_eln`, `sca_ac_analysis` и `sca_util`, чтобы свести к минимуму количество определённых реализаций имён в этих пространствах имён.

Для предопределённых примитивных модулей, которые используют порты для подключения к другой модели вычислений, пространство имён, связанное с подключённой моделью вычислений, должно использоваться как вложенное пространство имён, вложенное пространство имён `sca_de` должно использоваться для модулей или портов, которые используются для подключения к SystemC `discrete - event` каналы или порты.

Как правило, выбор внутренних, специфичных для реализации имён внутри реализации может привести к конфликты именования в приложении. Разработчик должен выбрать имена, которые вряд ли могут вызвать конфликты именования в приложении

Следует понимать и соблюдать такие правила:

1. Все символы SystemC-AMS имеют префикс `sca_`, а макросы - префикс `SCA_`

2. Все символы SystemC-AMS встроены в пространство имен. Эта концепция допускает расширяемость.

3. Символы, назначенные определённой модели вычисление MoC, находятся в соответствующем пространстве имен `sca_tdf`, `sca_lsf`, `sca_eln`.

4. Символы, относящиеся к базовой функциональности или общим

встроенным базовым классам, находятся в пространстве имен `sca_core.`

5. Символы утилит, такие как трассировка и типы данных, находятся в пространстве имён `sca_util.`

6. Символы, относящиеся к анализу частотной области слабого сигнала обозначаются `sca_ac_analysis.`

1.2.10. Модули SystemC-AMS

1. Модули AMS являются производными от модуля `sca_core :: sca_module,` который является производным от `sc_core :: sc_module`

Примечание: не все функции-члены `sc_core :: sc_module` могут быть использованы.

2. Модули AMS всегда являются примитивными модулями (модуль AMS не может содержать другие модули и / или каналы).

3. В иерархических описаниях все ещё используется `sc_core :: sc_module` (или SC_MODULE macro)

4. В зависимости от MoC, модули AMS предопределены или определены пользователем.

Языковые конструкции

• `sca_MoC:: sca_module` (или SCA_MoC _MODULE macro)

например модуль: `sca_tdf :: sca_module` (или `SCA_TDF_MODULE` macro)

1.2.11. Каналы SystemC AMS

1. Каналы AMS выводятся из `sca_core :: sca_interface,` который выводится из `sc_core: sc_interface`

2. Каналы AMS для потока данных времени и линейного потока сигналов:

- базируются на основе направленной связи
- используется для неконсервативной модели вычисления AMS
- Языковые конструкции:

`sca_MoC::sca_signal`

`например: sca_lsf::sca_signal, sca_tdf::sca_signal<T>`

3. Каналы AMS для электрических линейных сетей

• консервативная, ненаправленная связь

• характеризуется поперечным (напряжение) и сквозным значением (ток)

• языковые конструкции:

- `sca_MoC::sca_node / sca_MoC::sca_node_ref`
- `например: sca_eln::sca_node, sca_eln::sca_node_ref`

1.2.12. Языковой состав SystemC AMS - подведение итогов

• `sca_module` - базовый класс для примитива SystemC AMS;

• `sca_in / sca_out` – неконсервативный порт (направлено in / out);

• sca_terminal - консервативный терминал;

• sca_signal - неконсервативный (направленный) сигнал;

• sca_node / sca_node_ref - консервативный узел;

Модель вычислений MoC назначается пространством имен, например:

• sca_tdf :: sca: module - базовый класс для модулей примитивов потока данных по времени;

• sca_lsf :: sca_in - линейный входной поток сигналов

• sca_tdf: sca_in <int> - вход в TDF

• sca_eln :: sca_terminal - терминал электрической линейной сети

• sca_eln :: sca_node - узел электрической линейной сети

1.2.13. Композиция языкового элемента SystemC AMS - конвертер

• Элементы конвертера состоят из пространств имён обоих доменов:

• sca_tdf::sca_de::sca_in<T>- это порт примитивного модуля TDF, который можно подключить к sc_core::sc_signal<T> или к sc_core::sc_in<T>

• Аббревиатура: sca_tdf::sc_in<T>

sca_eln::sca_tdf::sca_voltage - это источник напряжения, который управляется входом TDF.

• Аббревиатура: sca_eln::sca_tdf_voltage

• sca_lsf::sca_de::sca_source - источник сигнала линейного тока, управляемый сигналом SystemC (sc_core::sc_signal<double>)

•

Аббревиатура: sca_lsf::sca_de_source

1.2.14. Подключение systemc-ams или systemc-ams.h

- systemc-ams включает SystemC и все определения классов, символов и макросов SystemC-AMS

• systemc-ams.h подключает systemc-ams и systemc.h и добавляет все символы следующих пространств имён в глобальное пространство имён (например, использование комплекс sca_util::sca_complex;)

- sca_ac_analysis
- sca_core
- sca_util •

Примечание. Символы пространств имён, связанных с MoC, не добавляются.

Глава 2. Моделирование временного потока данных (TDF)

2.1. Основы моделирования

Модель вычислений с временным потоком данных (TDF) основана на хорошо известном формализме моделирования синхронных данных Flow (SDF). В отличие от несвязанной модели вычисления SDF, TDF имеет дискретное время и стиль моделирования, который рассматривает данные как сигналы, отобранные во времени. Эти сигналы помечены в отдельных точках во времени и несут дискретные или непрерывные значения, такие как амплитуды.

На рисунке 2.1 показан основной принцип моделирования потока данных по времени. На этом рисунке три коммуникационные модули TDF, называемые A, B и C. Модель TDF состоит из набора подключенных модулей TDF, которые формируют ориентированный граф, называемый кластером TDF. Модули TDF являются вершинами графа, и сигналы TDF соответствуют его краям. Модуль TDF может иметь несколько входных и выходных портов TDF.

Модуль TDF, содержащий только выходные порты, также называется производителем (источником), а модуль TDF только с входными портами называется потребителем (приемником). Сигналы TDF используются для соединения портов разных модулей вместе.

Каждый модуль TDF содержит метод C ++, который вычисляет математическую функцию f (то есть fA, fB и fC), которая зависит от его прямых входов и возможных внутренних состояний. Таким образом, общее поведение кластера определяется как математическая композиция функций задействованных модулей TDF в соответствующем порядке, fC (fB (fA (...))), обозначенный {A→B→C} на рисунке 2.1.

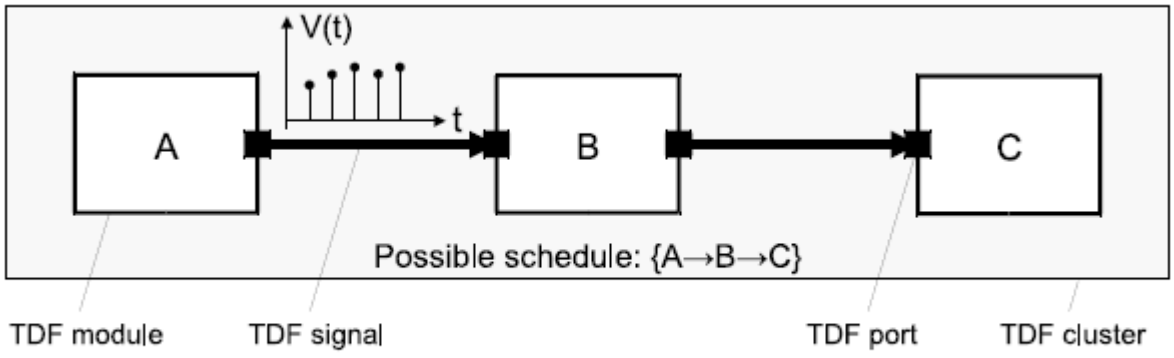


Figure 2.1. A basic TDF model with 3 TDF modules and 2 TDF signals

Рисунок 2.1. Базовая модель TDF с 3 модулями TDF и 2 сигналами TDF

Данная функция обрабатывается (или «запускается» в соответствии с формализмом SDF) тогда и только тогда, когда есть достаточно образцов, доступных на входных портах. В этом случае входные выборки считываются модулем TDF, где функция использует эти значения для вычисления одного

или нескольких результирующих значений, которые записываются в соответствующий выводной порт. В TDF число выборок, считываемых или записываемых в порты модуля, фиксируется во время моделирования, но число прочитанных и записанных образцов модулем TDF не обязательно равно. Отметка времени связана с каждой выборкой с использованием локального временного модуля TDF. Фиксированный интервал между двумя выборками называется временной шаг.

2.1.1. TDF модуль и атрибуты порта

Гибкость и выразительность моделирования TDF проистекают из способности определять атрибуты каждого модуля TDF и каждого из его портов.

В TDF возможно:

• Назначить определенный временной шаг модулю TDF (назначение временного шага модуля). Рисунок 2.2a показывает Модуль A TDF с шагом по времени модуля (Tm) 20 мкс.

• Назначить определенный временной шаг для данного порта модуля, принадлежащего кластеру (временной шаг порта назначение). На рисунке 2.2b показан модуль B TDF с шагом по времени входного порта TDF (Tp), равным 10 мкс.

• Назначить определенную скорость для данного порта модуля, принадлежащего кластеру (назначение скорости порта). На рисунке 2.2b показан модуль B TDF, где при активации каждого модуля считываются 2 выборки (скорость входного порта R установлен в 2, обозначено R: 2).

• Назначить конкретную задержку данному порту модуля, принадлежащего кластеру (назначение задержки порта). На рисунке 2.2c показан модуль C TDF, где при активации каждого модуля соответствующий образец, записывается на предыдущий временной шаг (задержка выходного порта D установлена на 1 отсчет, обозначенный D: 1).

• Назначить конкретное временное смещение для данного порта модуля, принадлежащего кластеру (временное смещение порта назначения). На рисунке 2.2d показан модуль D TDF со смещением времени модуля (Tpf), равным 1 мкс. Смещение времени можно назначить только специализированным портам для подключения к домену дискретных событий, так называемому преобразователю TDF портов.

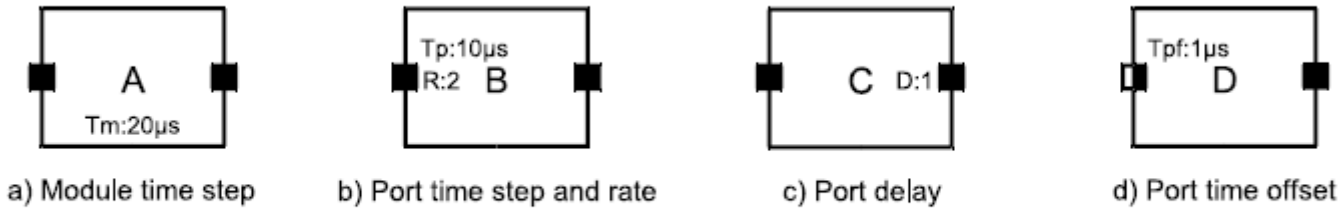


Figure 2.2. TDF module and port attributes

Рисунок 2.2. TDF модуль и атрибуты порта

При условии, что назначение атрибутов на портах и модулях модели TDF совместимо, порядок активации модулей TDF в кластере и количества сэмплов, которые они читают (потребляют) и записывают (производят) может быть статически определено до начала симуляции. Таким образом, и более формально, кластер TDF может быть определенным как набор подключенных модулей TDF, которые принадлежат одному и тому же статическому расписанию. Если назначения несовместимы, статическое расписание не может быть установлено, и кластер TDF считается не планируемым (см. также раздел 2.1.3). Поэтому после обязательной проверки согласованности кластера TDF расписание определяет последовательность, в которой выполняется алгоритмическое или процедурное описание каждого модуля TDF.

Основным преимуществом этого подхода является то, что выполнение моделей TDF не зависит от оценки /механизм обновления ядра дискретных событий SystemC, и, следовательно, может моделироваться более эффективно.

Модели TDF обрабатываются независимо с использованием механизма местного аннотирования времени. Взаимодействие между моделями TDF и чистые модели SystemC поддерживаются через определенные порты конвертера, как обсуждалось в Раздел 2.4

2.1.2. Топология модели TDF

На рисунке 2.3 показан пример модели TDF с характеристиками с множеством скоростей. Назначение скорости порта с значение скорости 2 (R: 2) выполнено на выходном порту модуля A. TDF. Порты без атрибута скорости считаются имеющими показатель 1 (графически не представлен). Когда модуль A активирован, 2 образца написаны. Поскольку оба модуля B и C читают один образец при каждой активации, возможный график для этого кластера TDF - это {A→B→C→B→C}

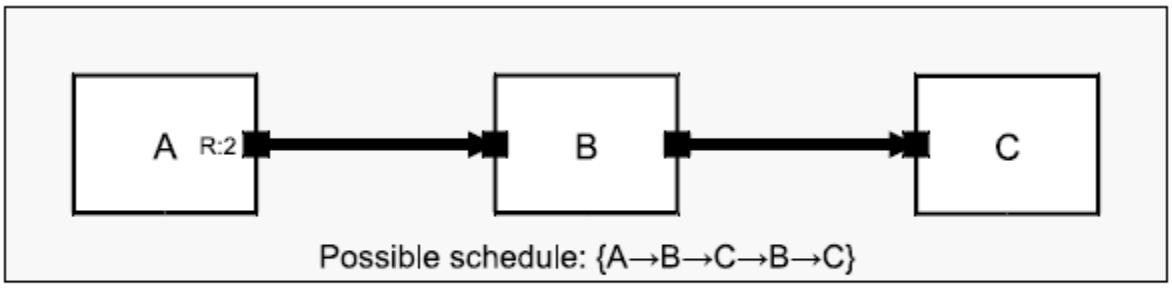


Figure 2.3. Multirate TDF model using port rate assignment

Рисунок 2.3. Модель многоскоростного TDF с использованием назначения скорости порта

Для обработки моделей TDF, содержащих циклы , необходимо ввести задержку для порта модуля принадлежность к одному из модулей цикла. Эта задержка порта должна быть определена во время разработки моделирования, чтобы сделать возможным статическое планирование. Простой пример приведен на рисунке 2.4, без цикла, на нем показан модуль A с задержкой в один отсчет, связанный с выходным портом (D: 1). Возможное расписание {A→B}, но также {B→A}, поскольку при первой активации модуля B входной порт модуля B считывает образец уже доступный благодаря назначенной задержке, определенной на этапе разработки.

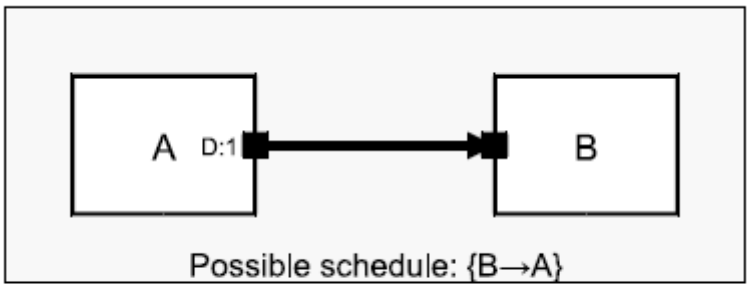


Figure 2.4. TDF model with port delay

Рисунок 2.4. Модель TDF с задержкой порта

Начальное значение выборки порта с задержкой определяется конструктором соответствующего типы данных. Для базовых типов данных (double, int и т.д.) Конструктор не обязательно назначает начальное значение, в результате чего будет неопределенное значение. Пользователю рекомендуется установить значения исходных образцов в случае использования задержки порта.

На рисунке 2.5 показан пример модели TDF, содержащей цикл. Это довольно распространенная ситуация при с обработкой сигнала с обратной связью. Обязательное назначение задержки порта со значением задержки 1 (D:

1) выполняется на выходном порту модуля C TDF. Присвоение задержки выходному порту модуля C позволяет запустить модуль B, когда первая выборка модуля A станет доступной на входе in0 модуля B.

A возможное расписание для этой модели TDF: {A→B→C}.

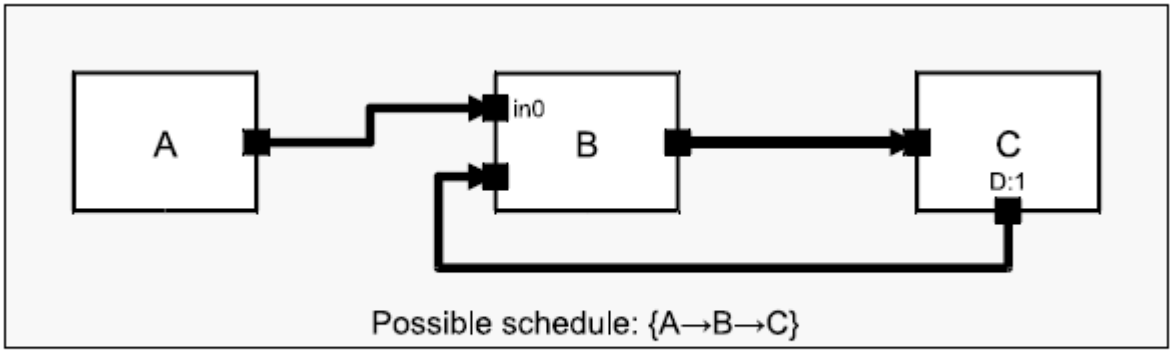


Figure 2.5. TDF model with loop, and port delay assignment

Figure 2.5. TDF model with loop, and port delay assignment

Рисунок 2.6 показывает более сложный пример смешивания мульти скорости и задержки. Возможный график кластера {A→B→B→C→D}. Модуль B выполняется дважды из-за назначений скорости порта (R: 2), выполненных на два соединенных порта (выходной порт модуля A и входной порт модуля C). Задержка порта на выходном порту модуля D (D: 1) требуется для правильного расчета графика (плана, расписания, режима).

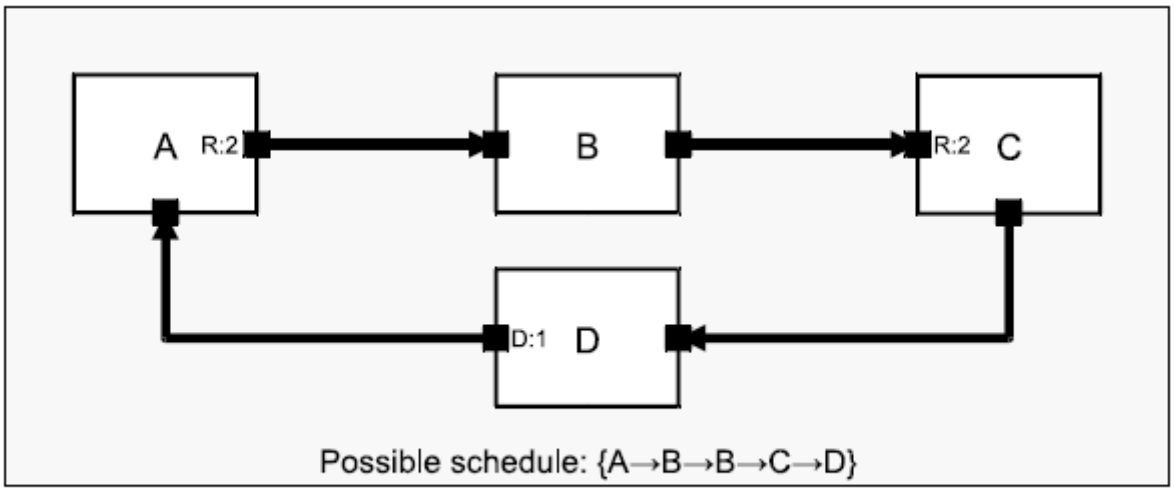


Figure 2.6. Multirate TDF model with loop

Figure 2.6. Multirate TDF model with loop

Рисунок 2.6. Многоскоростная модель TDF с петлей

Еще одним условием для правильного расписания является то, что сумма выборок, произведенных на выходных портах в пределах цикла должен быть равена сумме выборок, потребляемых входными портами в цикле. В противном случае, любой ограниченный график будет накапливать избыточные выборки где-то в кластере при его повторном выполнении.

Например, в случае, если скорость входного порта модуля C на рисунке 2.6 была изменена с 2 на 1, график {A→B→C→D→B→C→D} приведет к одному дополнительному образцу на выходе модуля D после выполнения графика один раз (см. рисунок 2.7)

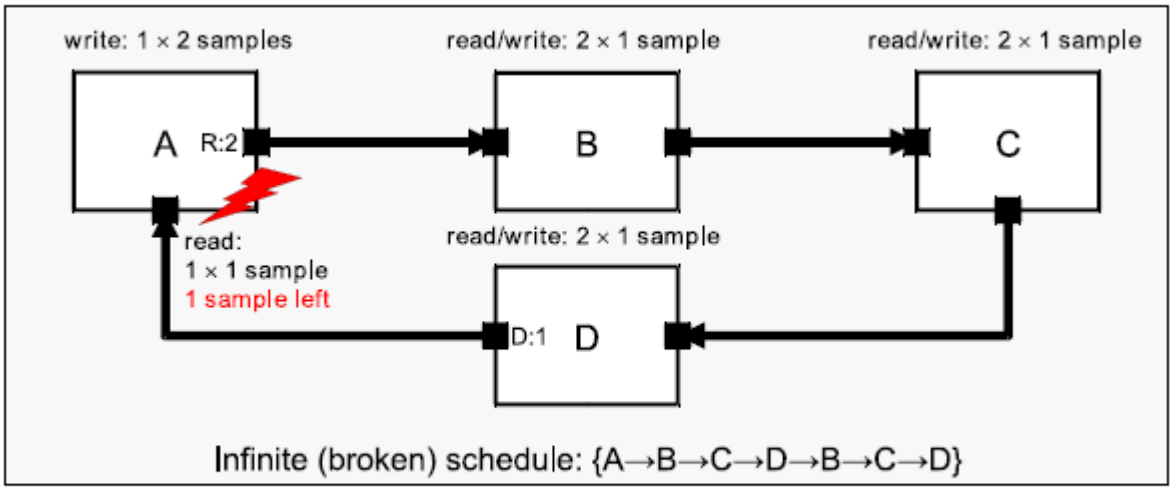


Figure 2.7. Multirate TDF model containing a loop with incompatible rates, resulting in accumulation of samples in the cluster yielding to an infinite (broken) schedule

Рисунок 2.7. Многоскоростная модель TDF, содержащая цикл с несовместимыми скоростями, в результате накоплении выборок в кластере, дают к бесконечный (сломанному) графику

На рисунке 2.8 показано, как можно связать модель TDF с областью дискретных событий с помощью портов конвертера TDF (обозначено). Например, сигнал дискретного события доступен на порту преобразователя TDF модуля A. TDF Модуль D имеет входной порт преобразователя TDF, считывающий дискретное событие - контрольный сигнал. Особое внимание следует уделять взаимодействию между TDF и областью дискретных событий. Это описано в разделе 2.4.

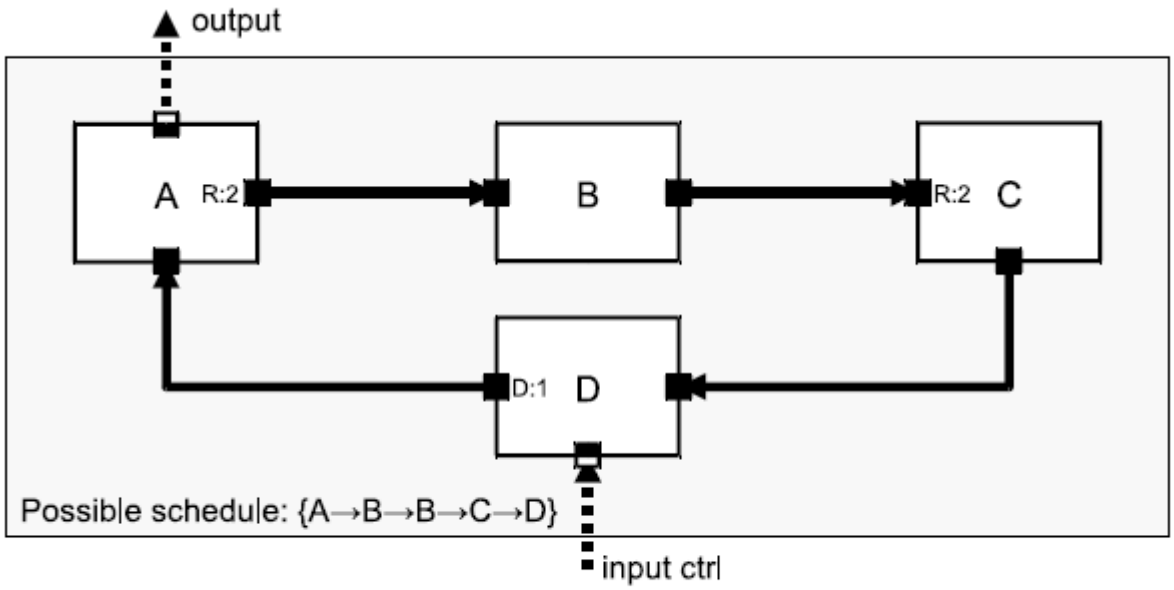


Figure 2.8. TDF model interfacing with discrete-event domain

Рисунок 2.8. Взаимодействие модели TDF с областью дискретных событий

Другой особый случай, когда модель TDF становится частью замкнутого контура, который включает в себя путь через область дискретных событий, как показано на рисунке 2.9. Сам кластер TDF не содержит петли, поэтому нет задержки порта, необходимой для расчета правильного расписания. Модуль A считывает образец с дискретного домена в первом дельта-цикле момента времени, связанного с выборкой с использованием входного сигнала преобразователя TDF порта. Модуль C записывает пример в область дискретных событий в том же дельта-цикле, используя преобразователь TDF выходного порта. Обратите внимание, что образцы TDF считываются из модуля C и проходят через модуль дискретных событий D на вход модуля A с задержкой на один шаг по времени TDF из-за механизма оценки / обновления ядра SystemC.

Более подробная информация о взаимодействии между TDF и областью дискретных событий описана в разделе 2.1.4. и 2.4.

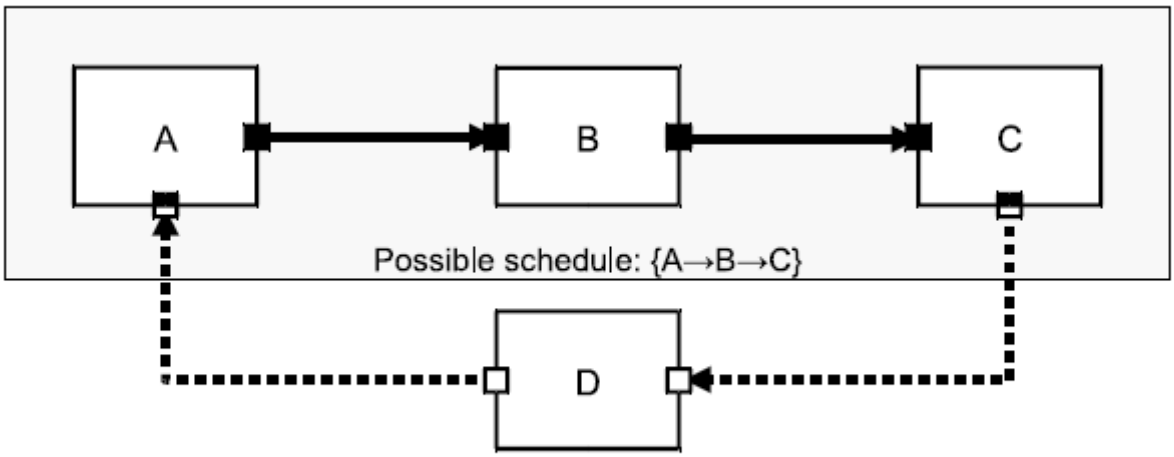


Figure 2.9. TDF model with loop via the discrete-event domain

Рисунок 2.9. Модель TDF с циклом через область дискретных событий

2.1.3. Назначение и распространение временных шагов

Определение скорости порта и задержек очень полезно для обработки различных частотных областей в пределах той же модель TDF, а также для создания сложных структур модулей TDF, включающих вложенные циклы. Главный пункт в том, что согласованность кластера зависит исключительно от совместимости скорости порта и задержки значения и, таким образом, по сути не зависит от выбранного временного шага (периода выборки) для его запуска. Если однажды эта проверка согласованности была подтверждена для конкретного кластера, она может работать на любой частоте посредством назначение временного шага порта или временного назначение модуля.

Рисунок 2.10 иллюстрирует простейший случай, когда все скорости установлены в 1 (графически не представлен), начало с шагом времени порта 10 мкс, назначенным входному порту модуля C (обозначается как Tp: 10 мкс), эта цифра показывает, как это значение временного шага используется для транзитивного расчета временных шагов других портов и модулей (обозначается курсивом Tp и Tm). Когда конкретная скорость (R) и задержка (D) не назначены порту, скорость 1 и задержка 0 отсчетов принимаются по умолчанию.

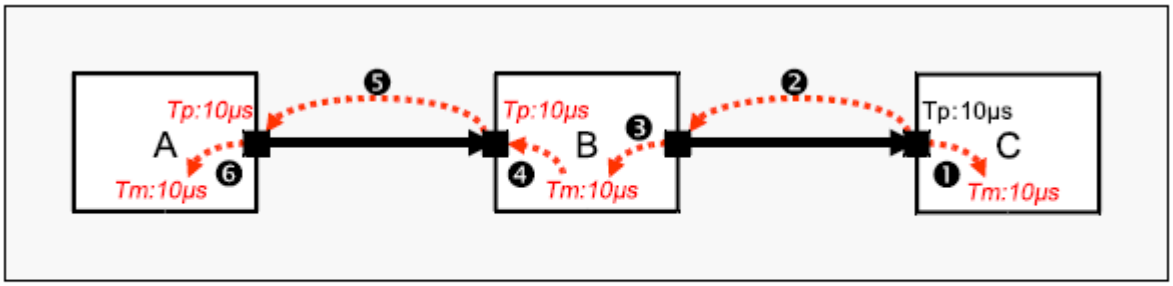


Figure 2.10. Propagation of the time step Tp:10μs set on the input port of module C

Рисунок 2.10. Распространение временного шага Tp: 10 мкс на входном порте модуля C

Шаг распространения по времени выполняется вверх и вниз по течению от целевого элемента с выполненным назначением временного шага (порт или модуль) в модели TDF. Этот процесс иллюстрируется пунктирными стрелками на Рисунок 2.10. Например, назначение временного шага порта на входе модуля C распространяется вниз по течению, установив временной шаг модуля C на 10 мкс (Tm: 10 мкс, пунктирная стрелка 1). Аналогично, временной шаг, назначенный на входном порте модуля C (Tp: 10 мкс), распространяется вверх по течению к выходному порту модуля B (пунктирная стрелка 2). Затем временному шагу модуля B назначается тот же временной шаг (Tm: 10 мкс, пунктирная стрелка 3), который, в свою очередь, перенаправляется на входной порт модуля B (Tp: 10 мкс, пунктирная стрелка 4) на выходной порт модуля A (Tp: 10 мкс, пунктирная стрелка 5) и, наконец, шаг по времени модуля A (Tm: 10 мкс, пунктирная стрелка 6).

Согласованность назначения и распространения временных шагов

Пример на рисунке 2.10 иллюстрирует пример распространения только с одним временным шагом порта (вход порт модуля TDF C). Если модель TDF не содержит петель, представленная схема распространения всегда генерирует правильное назначение временного шага, независимо от того, был ли один временной шаг назначен порту или к модулю. После того, как два или более временных шага порта и / или модуля были назначены в кластере TDF, необходимо проверить согласованность, чтобы обеспечить их совместимость с распространяющимися временными шагами, в зависимости от скорости портов.

На рисунке 2.11 ниже показан модуль, в котором шаг времени входного порта установлен на 10 мкс (Tp: 10 мкс) со скоростью 2 (R: 2), и шаг времени модуля установлен на 20 мкс (Tm: 20 мкс). Поскольку скорость выходного порта не установлена, он будет использовать скорость по умолчанию 1, что приводит к временному шагу выходного порта 20 мкс.

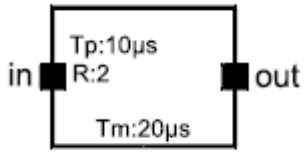


Рисунок 2.11. Шаг по времени порта, скорость порта и шаг по времени модуля должны быть согласованы

Шаг по времени модуля должен соответствовать скорости и временному шагу любого порта в модуле.

Соотношение между этими временными шагами и ставками становится:

module time step = input port time step ·X input port rate = output port time stepX output port rate

In the example of Figure 2.11, the following relation is checked: 20 µs = 10 µs · 2 = 20 µs · 1.

шаг по времени модуля = шаг по времени входного порта X скорость входного порта = шаг по времени выходного порта X· скорость выходного порта

В примере на рисунке 2.11 проверяется следующее соотношение: 20 мкс = 10 мкс ·x 2 = 20 мкс ·x 1.

В примере на рисунке 2.12 несколько модулей образуют кластер, где пользователь устанавливает два временных шага:

временной шаг модуля A установлен на 20 мкс (Tm: 20 мкс), а временной шаг входного порта модуля C установлен на 10 мкс (Tp: 10 мкс). Кроме того, пользователь установил скорость выходного порта модуля A на 2 (R: 2). Следовательно, модуль A активируется в два раза реже, чем модули B и C, так как модуль A записывает 2 выборки в активации, см. рисунок 2.3.

Указанный временной шаг порта на входе модуля C (Tp: 10 мкс - 1) распространяется вниз по потоку к модулю C, таким образом установка шага по времени на 10 мкс (Tm: 10 мкс, пунктирная стрелка 2). Аналогично, шаг по времени назначается входному порту модуля C (Tp: 10 мкс, 1) распространяется вверх по течению к выходному порту модуля B (пунктирная стрелка 3). Потом, временному шагу модуля B назначается тот же временной шаг (Tm: 10 мкс, пунктирная стрелка 4), который в свою очередь перенаправляется на входной порт модуля B (Tp: 10 мкс, пунктирная стрелка 5) и распространяется вверх по течению к выходному порту модуля A (Tp: 10 мкс, пунктирная стрелка 6). Поскольку скорость порта вывода модуля A равна 2, распространяемый шаг модуля по времени должен составлять 20 мкс (Tm: 20 мкс, пунктирная стрелка 7), что соответствует шагу времени, указанному пользователем модуля A (Tm: 20 мкс - 1).

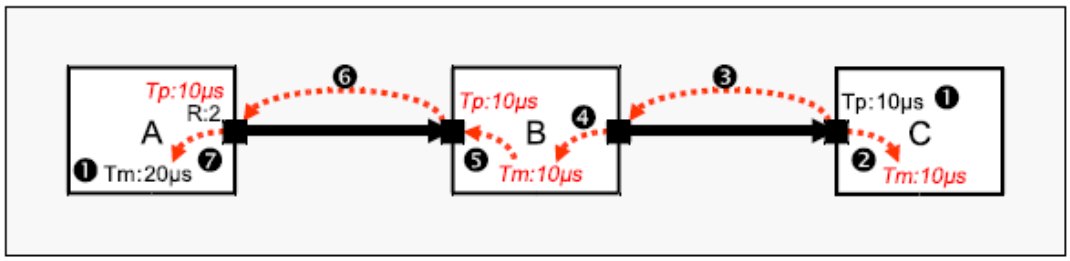


Figure 2.12. Time step propagation for a multirate TDF model with consistent time step assignments done by the user

Рисунок 2.12. Распространение временного шага для многоскоростной модели TDF с последовательными назначениями временных шагов, выполненными пользователем

На рисунке 2.13 показана та же модель TDF с несовместимым распространением шага по времени, что приводит к непланируемому кластеру.

Ожидаемый шаг модуля по времени в результате распространения составляет 20 мкс (Tm: 20 мкс, пунктирная стрелка 8), который отличается от назначенного временного шага модуля модуля A (Tm: 10 мкс). Поэтому не может быть выведен согласованный график.

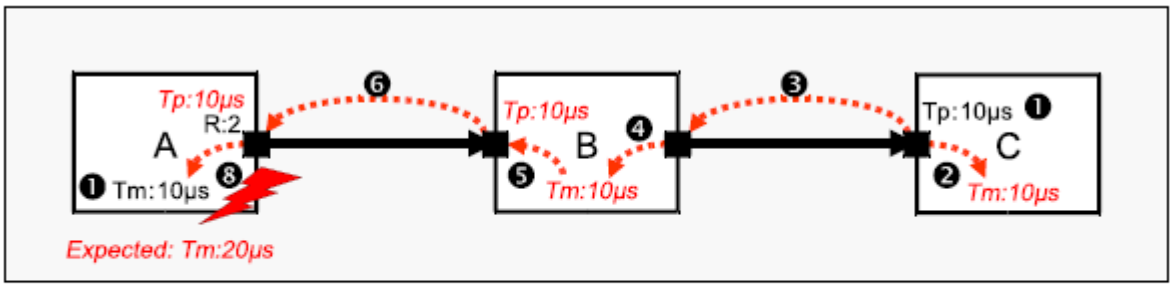


Figure 2.13. Time step propagation for a multirate TDF model with inconsistent time step assignments done by the user

Рисунок 2.13. Распространение шага по времени для многоскоростной модели TDF

с непоследовательными временными шагами, сделанными пользователем

В случае, если модель TDF содержит петли, определенные скорости портов, задержки и временные шаги должны быть согласованы с временными шагами, распространяющимися по петле вверх и вниз по течению, чтобы сделать модель TDF планируемый.

2.1.4. Несколько расписаний или кластеров

В одном приложении может быть несколько кластеров TDF. В этом случае каждый кластер TDF имеет свои собственные характеристики потока данных (частота выборки, период выборки и т. д.), планирование и порядок выполнения.

Основным элементом, который косвенно изменяет структуру кластера, является использование портов преобразователя TDF. Как объясняет рисунок 2.8 эти порты облегчают интерфейс с областью дискретных событий и таким образом определяют, где статическое расписание начнется или остановится. На рисунке 2.14 показан пример, в котором порты конвертера TDF используются для того, чтобы намеренно разделить кластер. Обратите внимание, что пунктирный сигнал указывает на использование сигнала дискретного события между модулем B и модулем C.

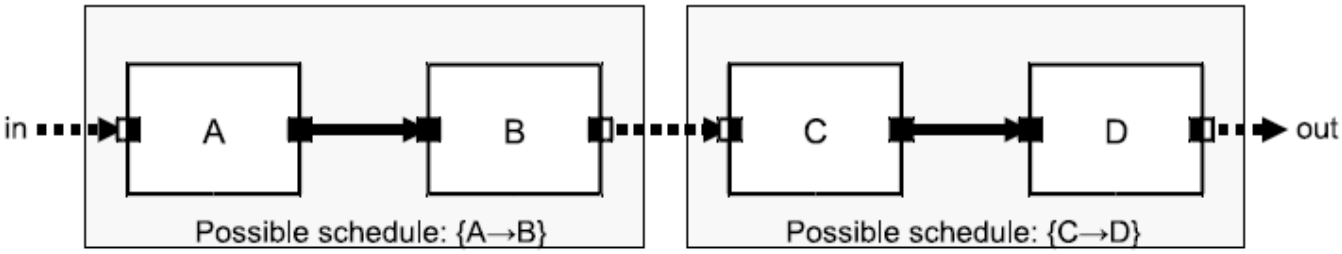


Figure 2.14. Use of TDF converter ports to deliberately split a cluster in two independent ones

Рисунок 2.14. Использование портов конвертера TDF для преднамеренного разделения кластера на два независимых

В связи с введением сигнала дискретного события в цепочку модулей, выполнение графика для каждого кластера становится независимым. Порт преобразователя модуля B запишет значение выборки в фазе оценки ядра SystemC, в первом дельта-цикле соответствующей временной точки выборки.

Порт преобразователя модуля C будет считывать выборку для соответствующего момента времени во время той же фазы оценки в том же дельта-цикле. Это означает, что модуль C будет читать предыдущее значение из модуля B, поскольку значение, записанное модулем B, будет изменено только на этапе обновления ядра SystemC, которое следует за завершением фазы оценки дельта-цикла на определенный момент времени. Это приводит к эффективной задержке одного временного шага TDF для выборок, считываемых модулем C.

Более подробная информация о взаимодействии между TDF и областью дискретных событий описана в разделе 2.4

2.1.5. Поведение обработки сигналов в моделях TDF

На рисунке 2.15 показано, как кластер модулей TDF обрабатывает сигналы, периодически активируя обработку функций содержащихся модулей в порядке производного графика. Он генерирует образцы для каждого модуля как функция времени. Поскольку все скорости установлены на 1, обработка очевидна: Модуль A записывает выборку в момент времени 0 мкс, которая считывается модулем B в момент времени 0 мкс, а модуль B записывает выборку в момент времени 0 мкс, который считывается модулем C в момент времени 0 мкс. С точки зрения сгенерированных образцов важно заметить, что именно операция записи образца, созданного модулем A, фактически включает модуль B. Соответственно, генерация выборки модулем B запускает модуль C.

Выход модуля A выдает непрерывный сигнал (Vin), значения которого доступны только при дискретных моменах времени. Шаг по времени между этими выборками равноудален и определяется шагом по времени на выходе порта модуля A (Tp: 10 мкс). Сигнал Vin подается в модуль B. В этом примере предполагается простой усилитель, с постоянным усилением. Образцы усиленного выходного сигнала (Vout) становятся доступными на выходе

модуля B. Это происходит одновременно с модулем A.

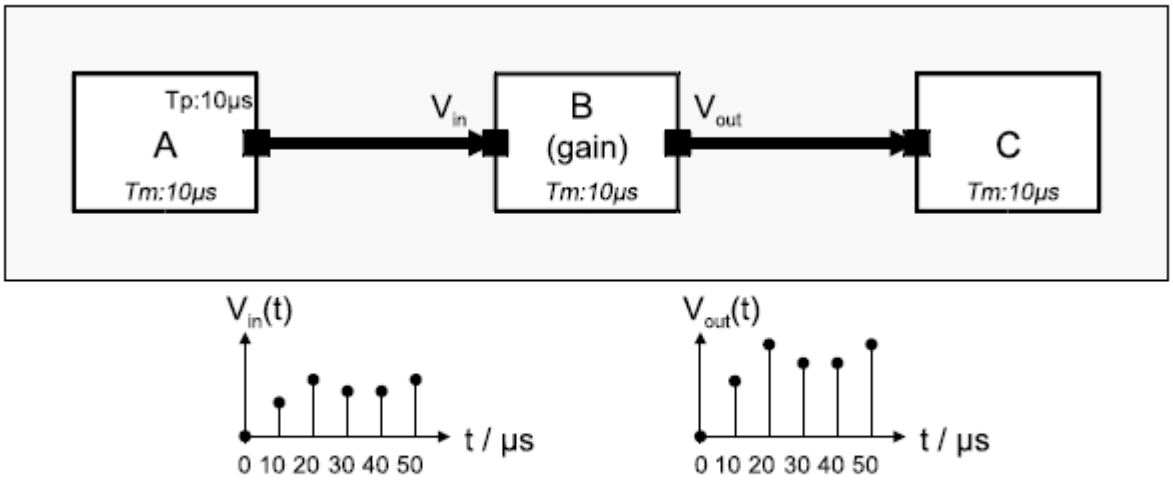


Figure 2.15. TDF module activation (processing) with read and written samples

Рисунок 2.15. Активация (обработка) модуля TDF с прочитанными и записанными образцами

Помимо использования модулей TDF для описания поведения в дискретном времени, модуль TDF может использоваться для инкапсуляции непрерывного поведения. Раздел 2.3 объяснит использование TDF для моделирования поведения дискретного и непрерывного времени.

2.2. Языковые конструкции

2.2.1. Модули TDF

Модуль TDF - это определяемый пользователем примитивный модуль для определения дискретного времени или встраивания поведение непрерывного времени.

Пример ниже показывает типичную структуру модуля TDF.

```
SCA_TDF_MODULE(my_tdf_module) ❶
{
  // port declarations
  sca_tdf::sca_in<double>  in; ❷
  sca_tdf::sca_out<double> out;

  SCA_CTOR(my_tdf_module) {} ❸

  void set_attributes() ❹
  {
    // module and port attributes
  }

  void initialize() ❺
  {
    // initial values of ports with a delay
  }

  void processing() ❻
  {
    // time-domain signal processing behavior or algorithm
  }

  void ac_processing() ❼
  {
    // small-signal frequency-domain behavior
  }
};

class my_second_module : public sca_tdf::sca_module ❽
{
 public:
  // port declarations
  // ...

  my_second_module( sc_core::sc_module_name ) {} ❾

  // definition of the TDF member functions as done above
  // ...
```

```
SCA_TDF_MODULE(my_tdf_module)
{
// port declarations
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
SCA_CTOR(my_tdf_module) {}
void set_attributes()
{
// module and port attributes
}
void initialize()
{
// initial values of ports with a delay
}
void processing()
{
```

```
    // time-domain signal processing behavior or
algorithm
    }
    void ac_processing()
    {
    // small-signal frequency-domain behavior
    }
    };
    class my_second_module : public sca_tdf::sca_module
    {
    public:
    // port declarations
    // ...
    my_second_module( sc_core::sc_module_name ) {}
    // definition of the TDF member functions as done
above
    // ...
```

1. Объявление примитивного модуля с использованием макроса SCA_TDF_MODULE для определения нового производного класса из класса sca_tdf :: sca_module.

2. Модуль TDF может иметь несколько входных и выходных портов. Только TDF порты должны быть созданы, см. Раздел 2.2.2.

3. Обязательный конструктор, использующий предопределенный макрос SCA_CTOR, который должен иметь имя модуля в качестве аргумента.

4. Необязательная функция-член set_attributes, в которой могут быть определены атрибуты модуля TDF и порта. Пользователь не имеет права вызывать эту функцию-член напрямую. Вызывается ядром симуляции во время разработки.

5. Необязательная функция-член initialize для инициализации данных-членов, представляющих состояние модуля и особенно начальные образцы портов с назначенными задержками. Пользователь не имеет права вызывать этого участника для функционирования напрямую. Он вызывается ядром моделирования в конце разработки, перед началом симуляции переходного процесса.

6. Обязательная функция-член обработки, которая инкапсулирует фактическую функцию обработки сигнала. Пользователь не имеет права вызывать эту функцию-член напрямую. Он вызывается ядром симуляции как часть моделирования во временной области (переходного процесса), когда активация каждого модуля продвигает локальный модуль на время по назначенному или производному временному шагу модуля.

7. Необязательная функция-член ac_processing, которая инкапсулирует частотную область слабого сигнала (AC) и шумовое поведение в частотной области слабого сигнала. Пользователь не имеет права вызывать этого участника, чтобы функционировать напрямую. Он вызывается ядром

моделирования при выполнении анализа частотной области слабого сигнала (см. главу 5).

8. Объявление модуля TDF путем создания нового класса, публично полученного из класса sca_tdf :: sca_module.

9 Конструктор, который всегда должен иметь параметр класса sc_core :: sc_module_name для назначения имени для модуля.

Модуль TDF содержит такие элементы, как порты, сигналы, параметры и функции-члены для домена времени (переходный) и анализа частотной области (AC) слабого сигнала. Вместе эти элементы реализуют поведение модуля.

Атрибуты модуля

Атрибуты модуля и порта, такие как частота дискретизации, задержка и шаг по времени, могут быть определены в функции-члене set_attributes. Функция-член может использовать любой допустимый оператор C ++ в дополнение к определению атрибутов модуля или порта. Эта функция-член вызывается во время разработки. Пример ниже показывает назначение временного шага модуля 10 мс и задержки одной выборки TDF на выход порта.

```
void set_attributes()
{
  set_timestep(10.0, sc_core::SC_MS); // module time step assignment of a of 10 ms

  out.set_delay(1); // set delay of port out to 2 samples
}
```

How to define port attributes inside this member function is explained in Section 2.2.2.

```
    void set_attributes()
    {
    set_timestep(10.0, sc_core::SC_MS); // module time
step assignment of a of 10 ms
    out.set_delay(1); // set delay of port out to 2
samples
```

}

Как определить атрибуты порта внутри этой функции-члена объясняется в разделе 2.2.2.

Инициализация модуля

Функция-член initialize может использоваться для установки локальных переменных, используемых в качестве переменных состояния, для чтения порта или атрибутов модуля, таких как временные шаги или скорости портов, или инициализация портов с задержкой. Эта функция-член выполняется только один раз, непосредственно перед началом фактической активации модуля (см. следующий раздел). Пример ниже показывает инициализацию внутренней переменной состояния s и использование функции-члена порта get_timestep и initialize. Доступные функции-члены порта описаны в разделе 2.2.2.

```
void initialize()
{
  s = 4.56; ❶

  std::cout << out.name() << ": Time step = " << out.get_timestep() << std::endl; ❷

  out.initialize(1.23); ❸
}
```

```
void initialize()
{
s = 4.56;
std::cout << out.name() << ": Time step = " <<
out.get_timestep() << std::endl;
out.initialize(1.23);
}
```

1. Установить локальную переменную состояния "s" (закрытый элемент данных типа double)

2. Получить шаг по времени выходного порта.

3. Инициализируйте первый образец выходного порта со значением 1.23.

Как использовать инициализацию порта внутри этой функции-члена, описано в разделе 2.2.2.

Модуль активации (обработки)

Обработка функции-члена является единственной обязательной функцией, которая должна быть перегружена в любом модуле TDF, поскольку она фактически определяет поведение модуля TDF в дискретном или непрерывном времени.

Эта функция-член выполняется при каждой активации модуля (см. Раздел 2.3). Пример ниже показывает очень простой случай, когда значение внутреннего элемента данных val записывается в выходной порт.

```
void processing()
{
  out.write(val); // writes value to output port out
}
```

```
void processing()
{
out.write(val); // writes value to output port out
}
```

Модуль местного времени

Функция-член get_time может использоваться в функции обработки для получения фактического, локального модульного времени. Возвращает время первой входной выборки текущей активации модуля как тип класса sca_core ::

sca_time. При разработке и инициализации фактическое время модуля, возвращаемое этой функцией, равно нолю (sc_core :: SC_ZERO_TIME), так как модуль еще не активирован. Пример ниже показывает как локальное время модуля может быть получено.

```
void processing()
{
  sca_core::sca_time local_time;
  local_time = get_time(); // get actual, local module time
}
```

```
    void processing()
    {
    sca_core::sca_time local_time;
    local_time = get_time(); // get actual, local module
time
```

}

Для многоскоростных моделей TDF местное время отдельных модулей TDF может отличаться. Кроме того, может быть смещение времени между временем локального модуля TDF и временем ядра SystemC. Поэтому функцию get_time следует использовать внутри модуля TDF в качестве замены sc_core :: sc_time_stamp.

Модуль конструктор

Макрос SCA_CTOR помогает определить стандартный конструктор модуля класса sca_tdf :: sca_module.

У него есть только один обязательный аргумент - имя модуля. В случаях, когда необходимо передать параметры с помощью конструктора, пользователь может определить обычный конструктор с произвольным числом параметров.

Данные члена должны быть инициализированы в списке инициализации конструктора, чтобы все члены были правильно инициализированы до вызова конструктора my_tdf_module.

```
my_tdf_module( sc_core::sc_module_name nm, double param_ )
: param(param_) {}
```

```
    my_tdf_module( sc_core::sc_module_name nm, double
param_ )
    : param(param_) {}
```

Ограничения на использование

Модуль TDF является примитивом модели вычисления TDF. Поэтому он не может создавать экземпляры субмодулей.

Структурный состав модулей TDF возможен путем определения классов, полученных из регулярных классов SystemC sc_core :: sc_module или использованием эквивалентного макроса SC_MODULE. Это обсуждается в Раздел 2.3.3.

Функции-члены set_attributes, initialize, processing и ac_processing не должны вызываться непосредственно пользователем. Эти функции-члены вызываются как часть семантики выполнения для временной области моделирования (раздел 2.5) или анализа частотной области слабого сигнала (раздел 5.1.2).

Функции SystemC для описания поведения дискретных событий, таких как создание методов и потоков, указание чувствительности, ожидание событий и т. д. не разрешается вызывать в модуле TDF. В противном случае, семантика выполнения для обработки дискретных событий SystemC может помешать выполнению модулей TDF.

Это означает, что функции-члены и макросы, такие как SC_HAS_PROCESS, SC_METHOD, SC_THREAD, wait, next_trigger, чувствительный, не должны использоваться в модуле TDF.

Поскольку местное время модуля TDF рассчитывается независимо от времени в области дискретных событий (Время ядра SystemC), функция sc_core :: sc_time_stamp не должна использоваться внутри модуля TDF.

Вместо этого следует использовать функцию-член get_time.

В случае, если сигналы SystemC необходимы для обработки в модуле TDF, специализированные порты преобразователя должны использоваться, как описано в следующем разделе.

2.2.2. TDF порты

Порт TDF - это объект, который предоставляет модулю TDF средства для связи с другими подключенными модулями. Из-за природы формализма моделирования TDF порт TDF может быть либо входным портом, либо выходным портом, но не inout (который доступен в SystemC). Порты TDF могут быть объявлены для любого типа данных определенных в C ++, SystemC, расширениями SystemC AMS, сторонней библиотекой или пользователем.

В настоящее время существует четыре класса портов TDF:

• TDF-порты класса sca_tdf :: sca_in <T> (входной порт) или sca_tdf :: sca_out <T> (выходной порт).

• Порты конвертера TDF класса sca_tdf :: sca_de :: sca_in <T> (входной порт конвертера) или sca_tdf :: sca_de :: sca_out <T> (порт выходного преобразователя).

Порты TDF используются для подключения модулей TDF с использованием сигналов класса sca_tdf :: sca_signal <T>. Конвертерные TDF порты позволяют модулям TDF взаимодействовать с сигналами дискретных событий класса sc_core :: sc_signal <T> или sc_core :: sc_buffer. Это объясняется в разделе 2.4.

Классы шаблонов портов позволяют использовать разные типы данных, например, double, int или bool. Тип данных double часто используется для представления амплитуды непрерывного сигнала. Пример ниже показывает создание четырех доступных классов портов TDF.

```
SCA_TDF_MODULE(my_tdf_module)
{
  sca_tdf::sca_in<double> in;    ❶
  sca_tdf::sca_out<double> out; ❷

  sca_tdf::sca_de::sca_in<bool> inp;  ❸
  sca_tdf::sca_de::sca_out< sc_dt::sc_logic > outp; ❹

  // rest of module not shown
};
```

```
SCA_TDF_MODULE(my_tdf_module)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
sca_tdf::sca_de::sca_in<bool> inp;
sca_tdf::sca_de::sca_out< sc_dt::sc_logic > outp;
// rest of module not shown
};
```

TDF input port that carries a continuous-value (real) signal.

TDF output port that carries a continuous-value (real) signal.

TDF input converter port from the discrete-event domain, using a boolean signal.

TDF output converter port to the discrete-event domain, using a SystemC logic signal.

1. Входной порт TDF, который несет непрерывный (реальный) сигнал.
2. Выходной порт TDF, который несет непрерывный (реальный) сигнал.
3. Порт входного преобразователя TDF из области дискретных событий, использующий логический сигнал.
4. Порт выходного преобразователя TDF в область дискретных событий, используя логический сигнал SystemC.

Атрибуты порта

Ряд атрибутов может быть назначен портам TDF. Они используются для контроля оценки и исполнения кластера TDF, к которому принадлежит модуль TDF. Атрибуты порта TDF должны быть установлены в функции – члене set_attributes модуля TDF, в котором объявлен порт (см. 2.2.1). Следующие функции – члены доступны для портов TDF для установки или получения атрибутов:

• Функции-члены set_timestep и get_timestep будут устанавливать и возвращать, соответственно, шаг по времени (период выборки) между двумя последовательными выборками.

• Функции-члены set_rate и get_rate будут устанавливать и возвращать, соответственно, количество выборок которые должны быть прочитаны или записаны в порт при выполнении каждого модуля. Скорость по умолчанию - 1 (односкоростной порт).

• Функции-члены set_delay и get_delay будут устанавливать и возвращать,

соответственно, количество выборок, которые вставляются перед чтением или записью в первый раз в порт. Значение по умолчанию зависит от конструктора по умолчанию для типа данных. В случае базового типа C ++, такого как bool, int, long, float и double, начальное значение может быть неопределенным. Поэтому рекомендуется инициализировать порт с начальным значением, если для порта была указана задержка (см. раздел «Инициализация порта»).

• Функция-член set_timeoffset и get_timeoffset установят или вернут фактическое время первой выборки порта. Эта функция доступна только для портов преобразователя.

В приведенном ниже примере показано использование этих функций-членов:

```
void set_attributes()
{
  out.set_timestep(0.01, sc_core::SC_US);   // set time step of port out
  out.set_rate(1);                          // set rate of port out to 1
  out.set_delay(2);                         // set delay of port out to 2 samples
  outp.set_timeoffset(0.2, sc_core::SC_US); // set absolute time of first sample of converter port
}

void initialize()
{
  out.get_rate();        // return the rate of port out
  out.get_delay();       // return the delay of port out
  out.get_timestep();    // return actual timestep of port out
  outp.get_timestep();   // return actual timestep of converter port outp
  outp.get_timeoffset(); // return absolute time of first sample of converter port outp
}
```

```
    void set_attributes()
    {
    out.set_timestep(0.01, sc_core::SC_US); // set time
step of port out
    out.set_rate(1); // set rate of port out to 1
    out.set_delay(2); // set delay of port out to 2
samples
    outp.set_timeoffset(0.2, sc_core::SC_US); // set
absolute time of first sample of converter port
    }
    void initialize()
    {
    out.get_rate(); // return the rate of port out
    out.get_delay(); // return the delay of port out
    out.get_timestep(); // return actual timestep of port
out
    outp.get_timestep(); // return actual timestep of
converter port outp
    outp.get_timeoffset(); // return absolute time of
first sample of converter port outp
    }
```

Инициализация порта

Начальные значения портов TDF с указанной задержкой должны быть указаны в функции-члене initialize соответствующего модуля TDF. В приведенном ниже примере показана инициализация порта, у которого задержка была установлена для 2 образцов.

```
void initialize() // use initialize method of TDM module to initialize ports
{
  // initialize port out (which has a delay attribute of 2)
  out.initialize(1.23);    // initialize first  sample with value 1.23 or
  out.initialize(1.23,0);  // initialize first  sample with value 1.23
  out.initialize(4.56,1);  // initialize second sample with value 4.56
}
```

```
    void initialize() // use initialize method of TDM
module to initialize ports
    {
    // initialize port out (which has a delay attribute
of 2)
    out.initialize(1.23); // initialize first sample with
value 1.23 or
    out.initialize(1.23,0); // initialize first sample
with value 1.23
    out.initialize(4.56,1); // initialize second sample
with value 4.56
    }
```

Порт чтения и записи доступ

Образцы (примеры) можно прочитать из входного порта TDF, вызвав функцию-член, прочитанную изнутри члена - функция обработки соответствующего модуля TDF. В случае многоскоростного порта образец индекса может быть передан в качестве аргумента для чтения.

В случае односкоростного входного порта TDF чтение с этого порта выполняется следующим образом:

```
SCA_TDF_MODULE(my_tdf_sink)
{
  sca_tdf::sca_in<double> in;

  SCA_CTOR(my_tdf_sink) : in("in") {}

  void processing()
  {
    // local variable
    double val; // variable to store value read from port in

    val = in.read(); // reading first sample from the input port
  }
};
```

```
    SCA_TDF_MODULE(my_tdf_sink)
    {
    sca_tdf::sca_in<double> in;
    SCA_CTOR(my_tdf_sink) : in("in") {}
```

```
    void processing()
    {
    // local variable
    double val; // variable to store value read from port
in
    val = in.read(); // reading first sample from the
input port
    }
    };
```

Последовательный доступ к чтению во время активации одного и того же модуля возвращает одно и то же значение, то есть образец ввода не используется доступом для чтения.

В случае многоскоростного входного порта TDF чтение с этого порта выполняется следующим образом:

```
SCA_TDF_MODULE(my_multi_rate_sink)
{
  sca_tdf::sca_in<double> in;

  SCA_CTOR(my_multi_rate_sink) : in("in") {}

  void set_attributes()
  {
    in.set_rate(2); // 2 samples read per module activation
  }

  void processing()
  {
    // local variable
    double val; // variable to store values read from port in

    val = in.read();  // read first sample
    val = in.read(0); // same method with index for first sample
    val = in.read(1); // same method with index for second sample
  }
};
```

```
    SCA_TDF_MODULE(my_multi_rate_sink)
    {
    sca_tdf::sca_in<double> in;
    SCA_CTOR(my_multi_rate_sink) : in("in") {}
    void set_attributes()
    {
    in.set_rate(2); // 2 samples read per module
activation
    }
    void processing()
    {
    // local variable
    double val; // variable to store values read from
port in
    val = in.read(); // read first sample
```

```
    val = in.read(0); // same method with index for first
sample
    val = in.read(1); // same method with index for
second sample
    }
    };
```

Атрибут скорости входного порта определяет количество выборок, доступных для активации модуля.

В приведенном выше примере скорость порта 2 дает доступ к двум выборкам с соответствующими индексами 0 и 1. Что касается портов с единой скоростью, последовательные обращения к чтению во время одной и той же активации модуля, возвращают одинаковое значение.

Образцы могут быть записаны в выходной порт TDF, передав значение образца в качестве аргумента функции - члена записи изнутри обработки функции-члена соответствующего модуля TDF. В случае многоскоростного порта образец индекса может быть передан вместе со значением образца в качестве аргумента для записи.

В случае односкоростного выходного порта TDF запись в этот порт выполняется следующим образом.

```
SCA_TDF_MODULE(my_const_source)
{
  sca_tdf::sca_out<double> out;

  my_const_source( sc_core::sc_module_name, double val_ = 1.0 )
  : out("out"), val( val_ ) {}

  void processing()
  {
    out.write( val ); // writes val as a new sample to the port out
  }

 private:
  double val; // value to be written to the port out
};
```

```
    SCA_TDF_MODULE(my_const_source)
    {
    sca_tdf::sca_out<double> out;
    my_const_source( sc_core::sc_module_name, double val_
= 1.0 )
    : out("out"), val( val_ ) {}
    void processing()
    {
    out.write( val ); // writes val as a new sample to
the port out
    }
    private:
    double val; // value to be written to the port out
    };
```

Последовательные обращения к записи во время одной и той же оценки

модуля перезаписывают значение выборки, т. е. только последний записанный выходной образец выпускается.

В случае многоскоростного выходного порта TDF запись в этот порт выполняется следующим образом:

```
SCA_TDF_MODULE(my_multi_rate_const_source)
{
  sca_tdf::sca_out<double> out;

  my_multi_rate_const_source(sc_core::sc_module_name, double val_ = 1.0 )
  : out("out"), val( val_ ) {}

  void set_attributes()
  {
    out.set_rate(2);  // 2 samples written per module activation
  }

  void processing()
  {
    out.write(val);   // writes val as the first sample to the port out
    out.write(val,0); // writes val as the first sample to the port out by specifying the index 0
    out.write(val,1); // writes val as the second sample to the port out by specifying the index 1
  }

 private:
  double val; // value to be written to the port out
};
```

```
SCA_TDF_MODULE(my_multi_rate_const_source)
{
sca_tdf::sca_out<double> out;
my_multi_rate_const_source(sc_core::sc_module_name,
double val_ = 1.0 )
: out("out"), val( val_ ) {}
void set_attributes()
{
out.set_rate(2); // 2 samples written per module
activation
}
void processing()
{
out.write(val); // writes val as the first sample to
the port out
out.write(val,0); // writes val as the first sample
to the port out by specifying the index 0
out.write(val,1); // writes val as the second sample
to the port out by specifying the index 1
}
private:
double val; // value to be written to the port out
};
```

Атрибут скорости выходного порта определяет количество выборок, которые могут быть записаны в порт за время активации модуля. В приведенном выше примере скорость порта 2 дает доступ на запись к 2

выборкам с соответствующими индекс 0 и 1. Что касается односкоростных портов, последовательный доступ к записи во время активации одного и того же модуля переписывает предыдущее значение образца.

Доступ на чтение и запись к сигналам дискретных событий SystemC осуществляется с помощью так называемых портов преобразователя класса sca_tdf :: sca_de :: sca_in <T> или sca_tdf :: sca_de :: sca_out <T>. Использование этих портов конвертера обсуждается в разделе 2.4.

Порт и время выборки

Функция-член get_time может использоваться только после завершения разработки, то есть в модуле TDF функции-члены инициализируют и обрабатывают, чтобы получить фактическое время запрошенной выборки на входе или выходном порту. Если аргумент не используется, он возвращает время первого образца, который был прочитан или записан в порт. В эту функцию может быть передан аргумент для указания индекса образца, где 0 указывает на первый образец.

```
void processing()
{
  sca_core::sca_time t;

  t = out.get_time();  // return time of the first sample of port out
  t = out.get_time(0); // same method, the first sample has index 0

  t = in.get_time(1);  // return time of second sample of port in, with index 1
}
```


```
    void processing()
    {
    sca_core::sca_time t;
    t = out.get_time(); // return time of the first
sample of port out
    t = out.get_time(0); // same method, the first sample
has index 0
    t = in.get_time(1); // return time of second sample
of port in, with index 1
    }
```


Ограничения на использование

Функции-члены порта TDF set_timestep, set_delay, set_rate и set_timeoffset для преобразователя TDF портов могут вызываться только в функции-члене TDF-модуля set_attributes, так как эта информация требуется для этапа разработки.

Функции-члены порта TDF get_timestep, get_delay, get_rate, get_time и get_timeoffset для портов преобразователя TDF могут быть вызваны только после завершения разработки, то есть в модуля TDF функция-член **initialize** or **processing** (инициализация или обработка).

2.2.3. TDF сигналы

Сигналы TDF используются для соединения портов TDF различных примитивных модулей TDF вместе. TDF сигналы несут выборки сигнала, в то время как порты TDF определяют направление сигналов от одного модуля TDF к другому. Сигналы TDF объявляются с использованием шаблона класса sca_tdf :: sca_signal <T>. Тип данных сигнала передается в качестве аргумента шаблона этому классу. Например, непрерывный сигнал может быть представлены с использованием типа данных double:

```
// signal declarations
sca_tdf::sca_signal<double> sig; // continuous-value signal
```


```
// signal declarations
sca_tdf::sca_signal<double> sig; // continuous-value signal
```


В отличие от сигналов SystemC, сигналы TDF расширений AMS не предоставляют функциям-членам возможность прямо читать или писать с канала. Вместо этого функции-члены read и write определены для входных и выходных TDF-портов соответственно, как уже описано в разделе 2.2.2.

Как и в SystemC, инициализация конструктора родительского модуля может использоваться для назначения определенного пользователем имени сигналу:

```
// assign the name "sig" to a TDF signal instance
called sig in the constructor initialization list
SC_CTOR(my_module) : sig("sig") {}
```

Раздел 2.3.3 опишет структурный состав модулей TDF более подробно и покажет примеры назначения пользовательских имен для портов и сигналов.

2.3. Моделирование дискретного и непрерывного поведения

Модуль TDF является основным структурным структурным блоком для описания поведения дискретного и непрерывного времени.

Это класс, который реализует описание поведения TDF и не может создавать экземпляры других модулей.

Модули TDF действуют как примитивные модули.

2.3.1. Дискретное моделирование

Поведение с дискретным временем может быть определено при обработке функции-члена. В этой функции-члене может быть дано чистое алгоритмическое или процедурное описание в C ++, которое выполняется при активации каждого модуля.

Активация модуля определяется временным шагом модуля, который может быть задан пользователем вместе с функцией - членом set_timestep или полученная путем распространения по времени (см. раздел 2.1.3).

На рисунке 2.16 приведен пример для синусоидального источника 1 кГц.

Определяя временной шаг модуля 0,125 мс, фактический выходной сигнал будет передискретизирован с коэффициентом 8.

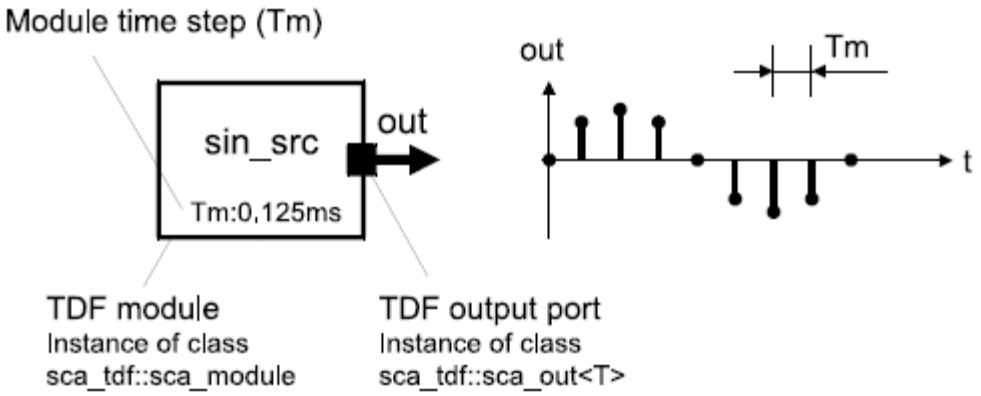


Figure 2.16. TDF primitive module implementing a sinusoidal source

Рисунок 2.16. Примитивный модуль TDF, реализующий синусоидальный источник

Соответствующий исходный код C ++ приведен ниже. Конструктор имеет параметры со значениями по умолчанию, которые определяют амплитуду, частоту и период выборки (в этом случае равны временному шагу модуля) синусоиды, которая будет генерироваться источником. Шаг по времени модуля обычно задается в функции-члене set_attributes. Функция sinus sin, которая является частью математической библиотеки C ++, используется в функции-члене **processing**. (обработка). Для записи примеров в выходной порт используется функция-член порта write.

```
    SCA_TDF_MODULE(sin_src)
    {
    sca_tdf::sca_out<double> out; // output port
    sin_src( sc_core::sc_module_name nm, double ampl_ =
1.0, double freq_ = 1.0e3,
    sca_core::sca_time Tm_ = sca_core::sca_time(0.125,
sc_core::SC_MS) )
    : out("out"), ampl(ampl_), freq(freq_), Tm(Tm_)
    {}
    void set_attributes()
    {
    set_timestep(Tm);
    }
    void processing()
    {
    double t = get_time().to_seconds(); // actual time
```

```
    out.write( ampl * std::sin( 2.0 * M_PI * freq * t )
);
    }
    private:
    double ampl; // amplitude
    double freq; // frequency
    sca_core::sca_time Tm; // module time step
    };
```

Пример 2.3.1

Заголовочный файл

```
    #ifndef SIN_SOURCE_H
    #define SIN_SOURCE_H

    #include <systemc-ams.h>                    // SystemC
AMS header

    SCA_TDF_MODULE(sin_source)                  // Declare
a TDF module
    {
      sca_tdf::sca_out<double> out;             // TDF
output port

      //parameter
      double ampl;                              //
amplitude
      double freq;                              //
frequency

      void set_attributes();                    // Set TDF
attributes

      void processing();                        //
Describe time-domain behaviour

      SCA_CTOR(sin_source)                      //
Constructor of the TDF module
      : out("out"),                             // Name
the port(s)
        ampl(1.0), freq(1e3) {}                 // Initial
values for ampl and freq

    };
```

```
#endif /* SIN_SOURCE_H */
```

Исполняемый файл

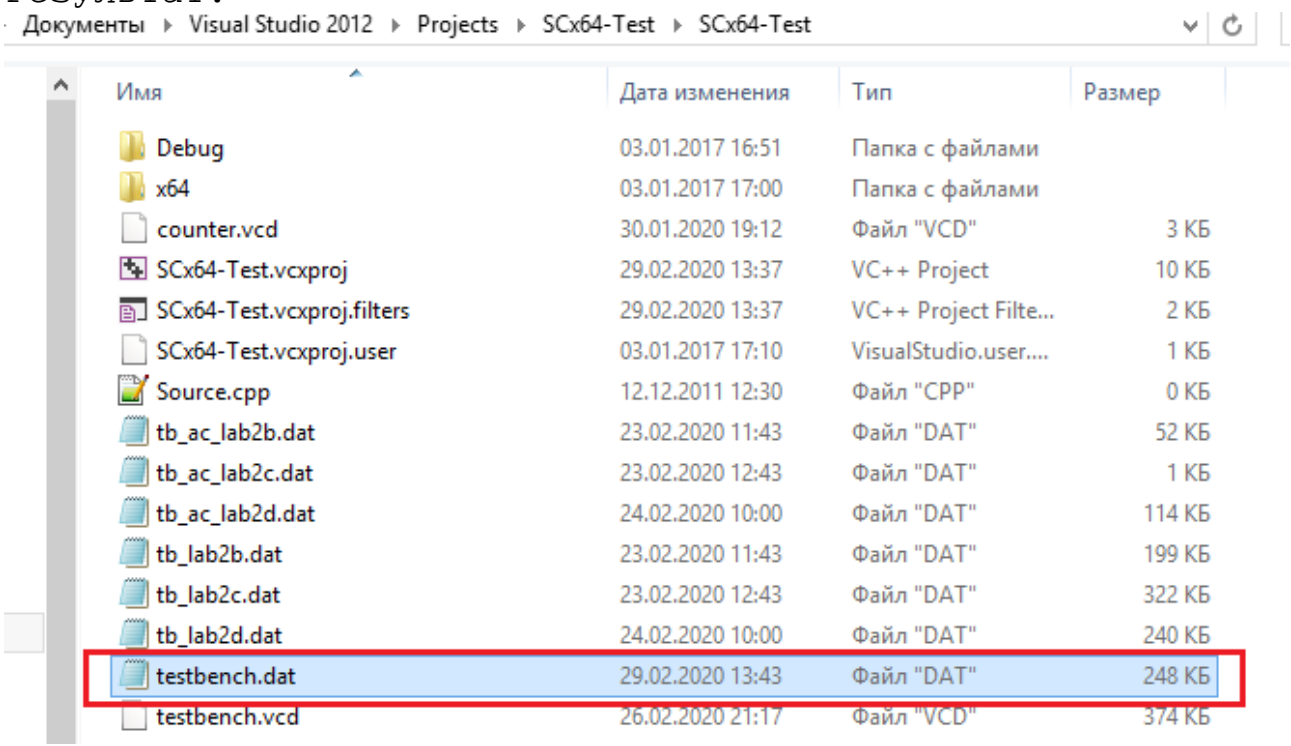
```
#include "sin_source.h"
#include <cmath>


SCA_TDF_MODULE(sin_src)
{
sca_tdf::sca_out<double> out; // output port
sin_src( sc_core::sc_module_name nm, double ampl_=
1.0, double freq_ = 1.0e3,
sca_core::sca_time Tm_ = sca_core::sca_time(0.125,
sc_core::SC_MS) )
: out("out"), ampl(ampl_), freq(freq_), Tm(Tm_)
{}
void set_attributes()
{
set_timestep(Tm);
}
void processing()
{
double t = get_time().to_seconds(); // actual time
out.write( ampl * std::sin( 2.0 * 3.1414 * freq * t )
);
}
private:
double ampl; // amplitude
double freq; // frequency
sca_core::sca_time Tm; // module time step

};
```

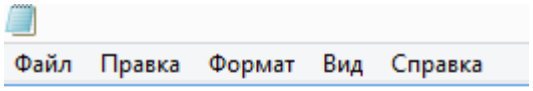
Testbench из Lab1A

```
#include "systemc-ams.h"
#include "sin_source.h"
#include "sin_source.cpp"

int sc_main(int argn, char* argc[])          //
SystemC main program
{
```

```
      sca_tdf::sca_signal<double> sig_1;          //
Signal to connect source w sink

      sin_source src_1("src_1");                  //
Instantiate source
        src_1.out(sig_1);                         //
Connect (bind) with signal

      sca_trace_file* tfp =                       // Open
trace file
        sca_create_tabular_trace_file("testbench");

      sca_trace(tfp, sig_1, "sig_1");             //
Define which signal to trace

      sc_start(10.0, SC_MS);                      // Start
simulation for 10 ms

      sca_close_tabular_trace_file(tfp);          // Close
trace file

      return 0;                                   // Exit
with return code 0
    }
```

Результат:



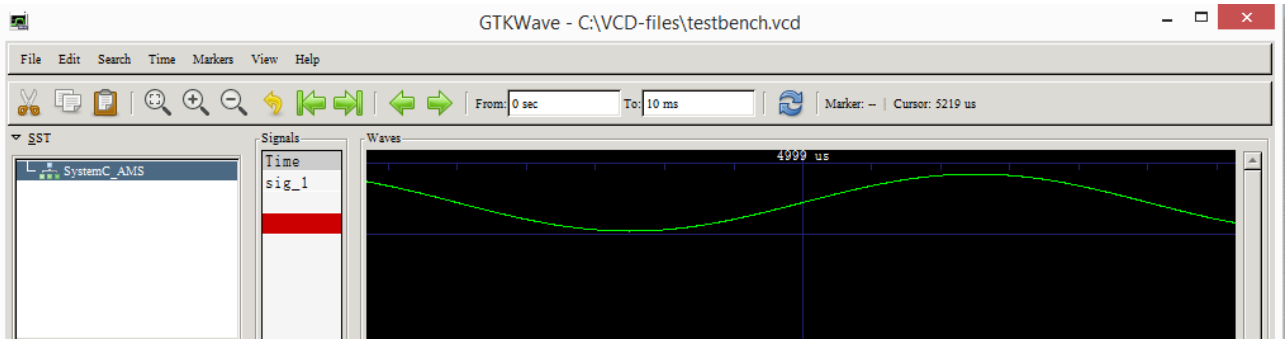


Файл Правка Формат Вид Справка

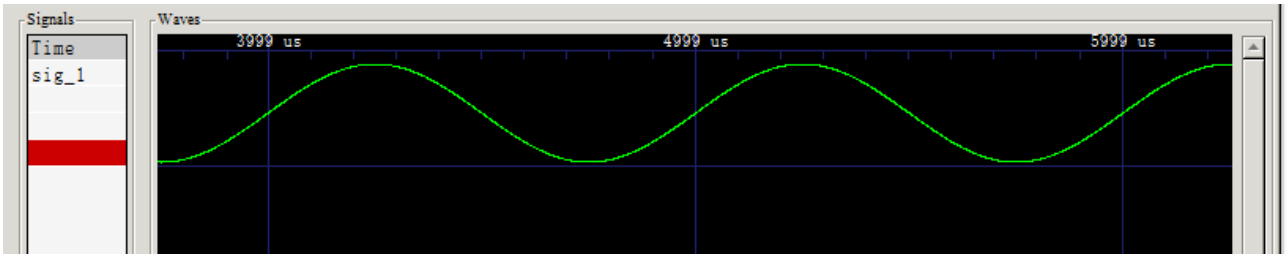
```
%time sig_1
0 0
1e-006 0.00628295866204
2e-006 0.0125656692983
3e-006 0.0188478838926
4e-006 0.0251293544487
5e-006 0.0314098329994
6e-006 0.0376890716168
7e-006 0.043966822422
8e-006 0.0502428375947
9e-006 0.0565168693832
1e-005 0.062788670114
1.1e-005 0.0690579922019
1.2e-005 0.0753245881593
1.3e-005 0.0815882106063
1.4e-005 0.0878486122805
1.5e-005 0.0941055460465
```

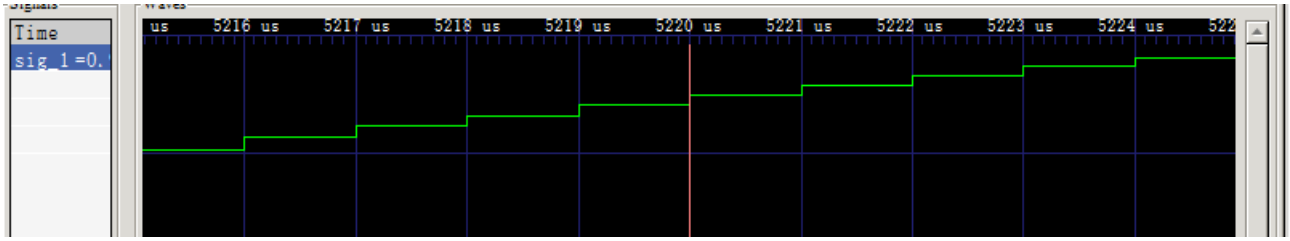
```
0.000236 0.996129766239
0.000237 0.996662343793
0.000238 0.997155577145
0.000239 0.997609446824
0.00024 0.998023934913
0.000241 0.99839902505
0.000242 0.998734702429
0.000243 0.999030953797
0.000244 0.999287767459
0.000245 0.999505133279
0.000246 0.999683042675
0.000247 0.999821488624
0.000248 0.999920465661
0.000249 0.999979969878
0.00025 0.999999998927
0.000251 0.999980552017
0.000252 0.999921629915
0.000253 0.999823234948
0.000254 0.999685371
0.000255 0.999508043513
0.000256 0.999291259487
0.000257 0.99903502748
0.000258 0.998739357608
0.000259 0.998404261541
0.00026 0.998029752509
0.000261 0.997615845295
```

Заменили табуляцию на VCD
T=0.125 ms

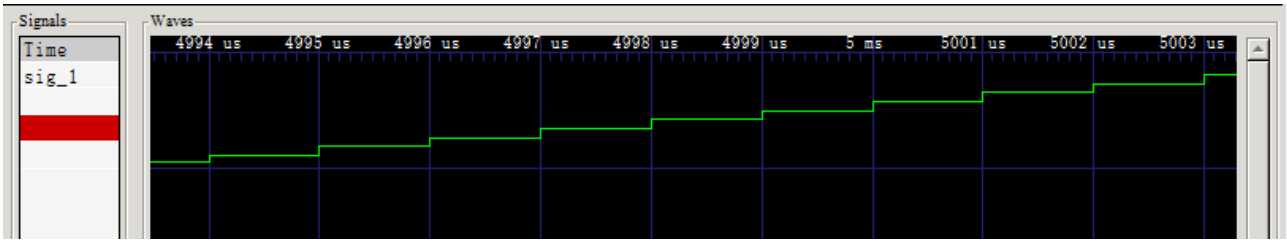


T=1 ms



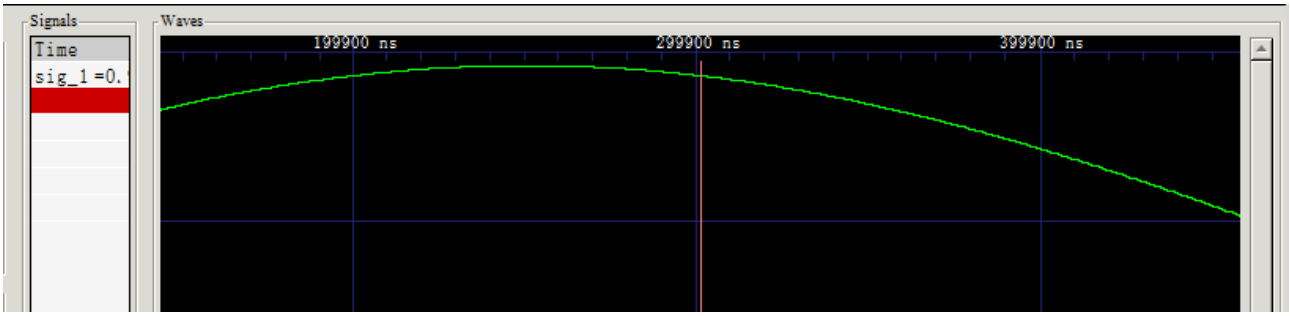


Шаг 1 мкс



1 мкс задано 10 мкс ???

Проверить изменение шага по времени !!!



Формат Step

2.3.2. Непрерывное моделирование

Модуль TDF можно использовать для встраивания линейных динамических уравнений в виде линейных передаточных функций в уравнения области Лапласа или пространство состояний. Хотя модель вычисления TDF обрабатывает образцы на дискретных временных шагах, уравнения этих встроенных функций будут решаться, считая сэмплы (оцифрованные фрагменты данных) входных данных как непрерывные сигналы. Результат встроенной системы линейных динамических уравнений, которая также непрерывна по времени и значению, дискретизируется в сигнал с использованием временного шага, который соответствует времени шага порта, в котором записаны образцы.

В приведенном ниже примере показан соответствующий поток сигналов при встраивании передаточной функции Лапласа (LTF) в модуле TDF. Входной сигнал представляет собой пошаговую функцию сэмплирования. Этот сигнал с дискретным временем интерпретируется функцией LTF как непрерывный сигнал. Отфильтрованный непрерывный сигнал записывается на выходной порт. Во время этой операции записи сигнал непрерывного времени дискретизируется в сигнал дискретного времени с использованием атрибутов выходного порта.

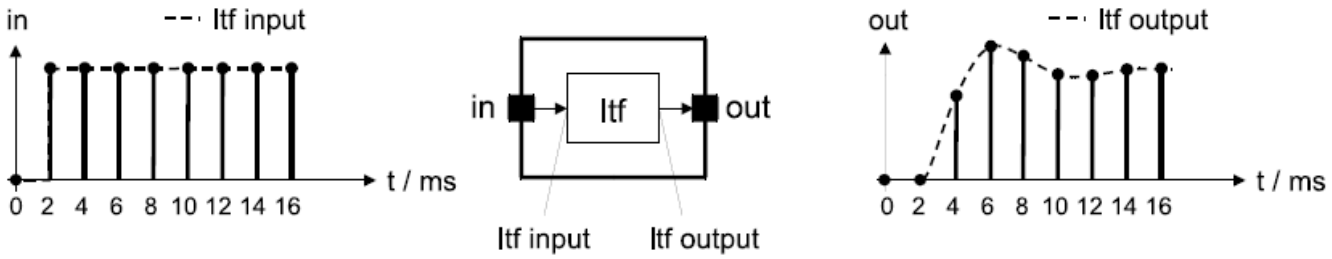


Figure 2.17. TDF primitive module embedding a continuous-time Laplace transfer function (LTF)

Рисунок 2.17. Примитивный модуль TDF, встраивающий непрерывную передаточную функцию Лапласа (LTF)

Передаточные функции Лапласа

Передаточная функция Лапласа (LTF) может использоваться в числителе-знаменателе или в форме нулей - полюсов.

Класс sca_tdf :: sca_ltf_nd реализует масштабированную линейную передаточную функцию непрерывного времени с переменной s Лапласа в форме дроби (числитель-знаменатель):

$$H(s) = k \cdot \frac{\sum_{i=0}^{M-1} num_i \cdot s^i}{\sum_{i=0}^{N-1} den_i \cdot s^i} \cdot e^{(-s \cdot delay)}$$

где k - постоянное усиление передаточной функции, M и N – количество коэффициентов числителя и знаменателя, соответственно, и num_i и den_i являются действительными коэффициентами числителя и знаменателя, соответственно. Коэффициенты должны быть объявлены как объекты класса sca_util :: sca_vector с типом данных *double*. Параметр *double* - это длительная задержка, применяемая к значениям, доступным на входе.

В приведенном ниже примере показан фильтр нижних частот первого порядка, использующий следующую передаточную функцию Лапласа:

$$H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c} s}$$

где H_0 - коэффициент усиления по постоянному току, а fc - частота среза фильтра в Гц.

Следующий код реализует такое поведение в модуле TDF с использованием класса sca_tdf :: sca_ltf_nd, который создает соответствующую систему уравнений. Коэффициенты числителя и знаменателя рассчитывается исходя из заданного пользователем коэффициента усиления и частоты среза.

```
    SCA_TDF_MODULE(ltf_nd_filter)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    ltf_nd_filter( sc_core::sc_module_name nm, double
fc_, double h0_ = 1.0 )
    : in("in"), out("out"), fc(fc_), h0(h0_) {}
    void initialize()
    {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0 /( 2.0 * M_PI * fc );
    }
    void processing()
```

```
    {
    out.write( ltf_nd( num, den, in.read(), h0 ) );
    }
    private:
    sca_tdf::sca_ltf_nd ltf_nd; // Laplace transfer
function
    sca_util::sca_vector<double> num, den; // numerator
and denominator coefficients
    double fc; // 3dB cut-off frequency in Hz
    double h0; // DC gain
    };
```

В следующем примере показан тот же фильтр, но теперь он реализован как описание нулей - полюсов с использованием класса sca_tdf :: sca_ltf_zp.

Класс sca_tdf :: sca_ltf_zp реализует масштабированную линейную передаточную функцию непрерывного времени в области Лапласа с переменной s в форме нулевого полюса:

$$H(s) = k \cdot \frac{\prod_{i=0}^{M-1}(s - zeros_i)}{\prod_{i=0}^{N-1}(s - poles_i)} \cdot e^{(-s \cdot delay)}$$

где k - постоянное усиление передаточной функции, M и N - число нулей и полюсов соответственно, и $zeros_i$ и $poles_i$ являются комплексными нулями и полюсами соответственно. Если M или N равно нулю, соответствующий член числителя или знаменателя должен быть константой 1. Параметр delay - это постоянная по времени задержка применяется к значениям, доступным на входе.

Нули и полюсы должны быть объявлены как объекты класса sca_util :: sca_vector со *complex* типом данных класса sca_util :: sca_complex.

Для фильтра нижних частот первого порядка, представление нулевого полюса становится:

$$H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c}s} = \frac{H_0 2\pi f_c}{s + 2\pi f_c}$$

Этот фильтр не требует определения нулей. Полюса и значение k фильтра рассчитываются из определяемого пользователем усиления постоянного тока H_0 и частоты среза fc.

```
    SCA_TDF_MODULE(ltf_zp_filter)
    {
    sca_tdf::sca_in<double> in;
```

```
    sca_tdf::sca_out<double> out;
    ltf_zp_filter( sc_core::sc_module_name nm, double
fc_, double h0_ = 1.0 )
    : in("in"), out("out"), fc(fc_), h0(h0_) {}
    void initialize()
    {
    // filter requires no zeros to be defined
    poles(0) = sca_util::sca_complex( -2.0 * M_PI * fc,
0.0 );
    k = h0 * 2.0 * M_PI * fc;
    }
    void processing()
    {
    out.write( ltf_zp( zeros, poles, in.read(), k ) );
    }
    private:
    double k; // filter gain
    sca_tdf::sca_ltf_zp ltf_zp; // Laplace transfer
function
    sca_util::sca_vector<sca_util::sca_complex > poles,
zeros; // poles and zeros as complex values
    double fc; // 3dB cut-off frequency in Hz
    double h0; // DC gain
    };
```

Коэффициенты числителя и знаменателя или значения нулевого полюса не должны быть статическими. Их значения могут измениться во время моделирования.

Уравнения пространства состояний

Класс sca_tdf :: sca_ss реализует систему с непрерывным временем, поведение которой определяется следующими уравнениями пространства состояния:

$$\frac{ds(t)}{dt} = \mathbf{A} \cdot s(t) + \mathbf{B} \cdot x(t - delay)$$
$$y(t) = \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay)$$

где s (t) - вектор состояния, x (t) - входной вектор, а y (t) - выходной вектор. Задержка параметра - постоянная по времени задержка применяется к значениям, доступным на входе. A, B, C и D являются матрицами, имеющими следующие характеристики:

• A - матрица размером n x n, где n - количество состояний.

• B - матрица размером n × m, где m - количество входов.

• C - матрица r-x-n, где r - количество выходов.
• D - матрица размером r x m.

Матрицы A, B, C и D должны быть объявлены как объекты класса sca_util :: sca_matrix с типом данных double.

В следующем примере показан тот же фильтр нижних частот, но теперь он реализован как уравнение пространства состояний с использованием класса sca_tdf :: sca_ss.

```
    SCA_TDF_MODULE(statespace_eqn)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    statespace_eqn( sc_core::sc_module_name nm, double
fc_, double h0_ = 1.0 )
    : in("in"), out("out"), fc(fc_), h0(h0_) {}
    void initialize()
    {
    double r_val = 1e3;
    double c_val = 1.0 / ( 2.0 * M_PI * fc * r_val);
    a(0,0) = -1.0 / ( c_val * r_val );
    b(0,0) = 1.0 / r_val;
    c(0,0) = h0 / c_val;
    d(0,0) = 0.0;
    }
    void processing()
    {
    sca_util::sca_vector<double> x;
    x(0) = in.read();
    sca_util::sca_vector<double> y = state_space1( a, b,
c, d, s, x );
    out.write(y(0));
    }
    private:
    sca_tdf::sca_ss state_space1; // state-space equation
    sca_util::sca_matrix<double> a, b, c, d; // state-
space matrices
    sca_util::sca_vector<double> s; // state vector
    double fc; // 3dB cut-off frequency in Hz
    double h0; // DC gain
    };
```

Использование вектора состояния

Если коэффициент (то есть параметр) в передаточной функции Лапласа или в уравнении пространства состояний имеет изменения, соответствующая

система уравнений будет переинициализирована. Пользовательский вектор класса sca_util :: sca_vector <double> может использоваться для хранения состояния системы уравнений. Если не указано, используется внутренний вектор состояния, который недоступен для пользователя. Пользовательский вектор состояния не изменяется во время повторной инициализации, но только внутреннее состояние по умолчанию сбрасывается в ноль. Это позволяет создавать фильтры с разными параметрами, например, для реализации переключателя с разными частотами отсечки путем определения нескольких экземпляров LTF, использующих один и тот же вектор состояния. Пример ниже показывает, как смоделировать такой переключатель.

```
    SCA_TDF_MODULE(ltf_switch)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    sca_tdf::sca_de::sca_in<bool> fc_high; // control
signal from the discrete-event domain
    ltf_switch( sc_core::sc_module_name nm, double fc0_,
double fc1_, double h0_ = 1.0 )
    : in("in"), out("out"), fc_high("fc_high"),
fc0(fc0_), fc1(fc1_), h0(h0_) {}
    void initialize()
    {
    num(0) = 1.0;
    den0(0) = den1(0) = 1.0;
    den0(1) = 1.0 /( 2.0 * M_PI * fc0 );
    den1(1) = 1.0 /( 2.0 * M_PI * fc1 );
    }
    void processing()  //1
    {
    if ( fc_high.read() )
    out.write( ltf1( num, den1, state, in.read(), h0 ) );
    else
    out.write( ltf0( num, den0, state, in.read(), h0 ) );
    }
    private:
    sca_tdf::sca_ltf_nd ltf0, ltf1;
    sca_util::sca_vector<double> num, den0, den1;
    sca_util::sca_vector<double> state;  //2
    double fc0, fc1;
    double h0;
    };
```

//1. Определяемый пользователем вектор состояния сохраняется постоянным во время повторной инициализации функции LTF.

//2. Объявление определяемого пользователем вектора состояния для хранения состояния системы во время повторной инициализации функции LTF.

Использование передаточных функций Лапласа или уравнений пространства состояний в многоскоростных приложениях

Передаточные функции Лапласа или примеры уравнений пространства состояний, показанные до сих пор, используют метод чтения из входного порта для извлечения одного значения и использования метода записи для записи одного значения в выходной порт.

Передаточная функция Лапласа или уравнения пространства состояний также могут быть встроены в многоскоростные приложения, где, например, входной сигнал имеет более высокую скорость, чем выходной сигнал, как показано на рисунке 2.18.

В этом примере модуль TDF должен прочитать два входных значения при активации каждого модуля, которые затем должны быть переданы к встроенной функции.

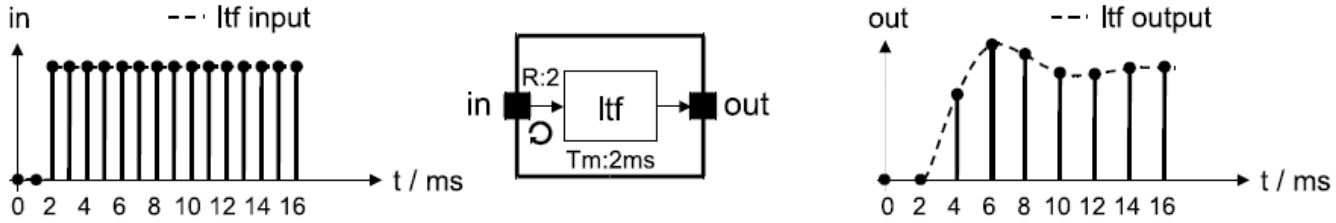


Figure 2.18. Laplace transfer function used for combined filtering and decimation

Рисунок 2.18. Передаточная функция Лапласа, используемая для комбинированной фильтрации и прореживания

Чтобы передать все доступные сэмплы на входном порте непосредственно в функцию LTF, не значения, а ссылка на сам порт передается в качестве аргумента функции LTF, как показано в примере ниже.

```
SCA_TDF_MODULE(ltf_multirate_filter)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
tf_multirate_filter( sc_core::sc_module_name nm,
double fc_, double h0_ = 1.0 )
: in("in"), out("out"), fc(fc_), h0(h0_) {}
set_attributes()
{
in.set_rate(2);
}
```

```
void initialize()
{
num(0) = 1.0;
den(0) = 1.0;
den(1) = 1.0 /( 2.0 * M_PI * fc );
}
void processing()
{
out.write( filter( num, den, in, h0 ) ); //1
}
private:
sca_tdf::sca_ltf_nd filter;
sca_util::sca_vector<double> num, den;
double fc;
double h0;
};
```

//1. Аргумент in напрямую передает ссылку на входной порт функции LTF. Обратите внимание, что в предыдущих случаях использовался элемент чтения входного порта, который возвращает значение типа double, которое переходит в функцию LTF.

Аналогичным образом модули TDF со встроенными передаточными функциями Лапласа или уравнениями в пространстве состояний могут быть разработаны с использованием выходных портов со скоростью выше 1. Запись нескольких выборок в выходной порт облегчается методом записи порта, который может получить доступ к значениям непрерывного времени из преобразования Лапласа или пространства состояний и записать полный набор выходных выборок в выходной порт.

Там нет другого языка конструкция, необходимого для использования этой функции.

Особая осторожность должна быть предпринята в случае, если количество выходных выборок превышает количество входных выборок.

Например, в модуле TDF с частотой выходного порта 3 и скоростью входного порта 2 существует то, что 1 выборка отсутствует при первой активации модуля для записи требуемых сэмплов (3) на выход. Чтобы решить это, *time continuous* задержка для входного сигнала должна быть указана как дополнительный параметр *delay*, который является одним из параметры функции.

2.3.3. Структурная композиция модулей TDF

То, как модули TDF создаются и взаимосвязаны для формирования кластера TDF, не отличаются от обычных модулей SystemC. Они могут быть созданы как дочерние модули внутри обычного родительского модуль SystemC, созданного с помощью макроса SC_MODULE или путем публичного

(открытого) извлечения из sc_core :: sc_module. Этот родительский модуль также создает все необходимые порты для связи с внешним миром и внутренние сигналы для взаимосвязи дочерними модулями. Параметризация инстанцированных модулей, а также взаимосвязь модулей должны быть выполнены в конструкторе (например, созданном с помощью макроса SC_CTOR) родительского модуля SystemC. Инстанцирование и соединение модулей TDF на верхнем уровне внутри sc_main выполняется аналогичным образом.

Привязка порта

Для правильного подключения модулей TDF к другим модулям и сигналам TDF, или даже с обычными для SystemC модулями и сигналами возможны следующие конкретные привязки, как показано на рисунке 2.19 и рисунке 2.20. Правила привязки портов совместимы и дополняют правила SystemC.

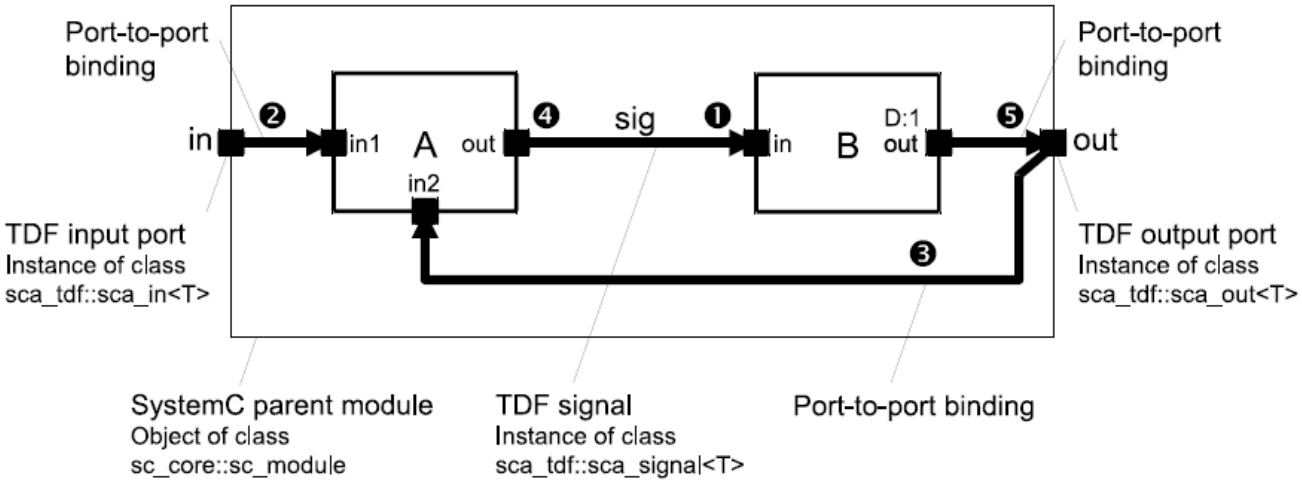


Figure 2.19. Port binding rules for TDF input and output ports

Рисунок 2.19. Правила привязки портов для портов ввода и вывода TDF

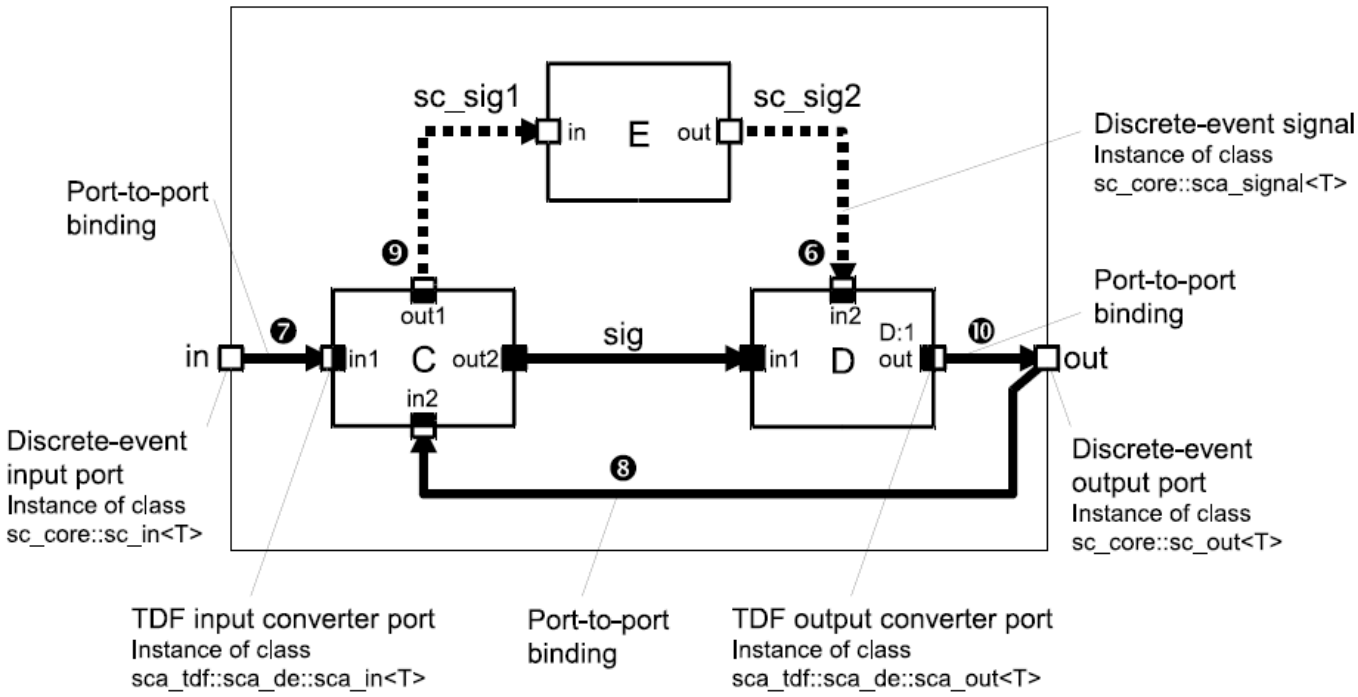


Figure 2.20. Port binding for TDF input and output converter ports

Рисунок 2.20. Связывание портов для входных и выходных портов преобразователя TDF

1. Связывание входного порта TDF с сигналом TDF.
2. Связывание входного порта TDF с входным портом TDF родительского модуля (привязка порт-порт).
3. Привязка входного порта TDF к выходному порту TDF родительского модуля (привязка порт-порт).
4. Связывание выходного порта TDF с сигналом TDF.
5. Привязка выходного порта TDF к выходному порту TDF родительского модуля (привязка порт-порт).
6. Привязка порта входного преобразователя TDF к входному сигналу дискретного события.
7. Привязка порта преобразователя ввода TDF к порту ввода дискретного события родительского модуля (порт-порт связывание).
8. Привязка входного порта преобразователя TDF к выходному порту дискретного события родительского модуля (порт-порт связывание).
9. Связывание порта выходного преобразователя TDF с выходным сигналом дискретного события.
10. Привязка порта выходного преобразователя TDF к порту вывода дискретного события родительского модуля (порт-порт связывание).

Кроме того, входной порт TDF или выходной порт TDF должен быть привязан только к одному сигналу TDF на всем протяжении всей иерархии.

Сигнал TDF должен быть привязан точно к одному выходному TDF порту примитивного модуля TDF, и могут быть связаны с TDF входными портами примитивных модулей по всей иерархии.

Пример ниже показывает реализацию структурного состава рисунка 2.19.

```
SC_MODULE(my_structural_module)
{
sca_tdf::sca_in<double> in; //1
sca_tdf::sca_out<double> out;
mod_a a; //2
mod_b b;
SC_CTOR(my_structural_module)
: in("in"), out("out"), a("a"), b("b"), sig("sig")//3
{
a.in1(in);//4
a.in2(out);
a.out(sig);
b.in(sig);
b.out(out);
}
private:
sca_tdf::sca_signal<double> sig;//5
};
```

1. Входные и выходные порты TDF, объявленные внутри этого модуля класса sc_core :: sc_module, становятся частью структурного состава.

2. Дочерние TDF модули объявляются в родительском модуле. Объявление этих дочерних модулей должно быть известно до объявления в этом контексте, например, путем включения их через заголовочные файлы.

3. Список инициализации в конструкторе родительского модуля распространяет необходимую конфигурацию параметры для портов TDF, сигналов TDF и дочерних модулей.

4. Привязка порта выполняется внутри конструктора.

5. Внутренние сигналы TDF используются для подключения TDF портов и дочерних модулей. Эти сигналы объявлены быть приватным, так как они не должны быть доступны извне модуля.

Пример ниже показывает реализацию структурного состава рисунка 2.20.

```
SC_MODULE(my_mixed_module)
{
sc_core::sc_in<double> in;
sc_core::sc_out<double> out;
mod_c c; // TDF primitive module
mod_d d; // TDF primitive module
mod_e e; // SystemC module
```

```
SC_CTOR(my_mixed_module)
: in("in"), out("out"), c("c"), d("d"), e("e"),
sig("sig"), sc_sig1("sc_sig1"), sc_sig2("sc_sig2")
{
c.in1(in);
c.in2(out);
c.out1(sc_sig1);
c.out2(sig);
d.in1(sig);
d.in2(sc_sig2);
d.out(out);
e.in(sc_sig1);
e.out(sc_sig2);
}
private:
sca_tdf::sca_signal<double> sig;
sc_core::sc_signal<bool> sc_sig1;
sc_core::sc_signal<bool> sc_sig2;
};
```

2.3.4. Многоскоростное поведение

Для реализации многоскоростного поведения в модуле TDF может использоваться функция-член TDF-порта set_rate.

На рисунке 2.21 ниже показан пример, в котором скорость выходного порта установлена на 2. Для каждой активации модуля один образец считывается из входного порта, а два образца записываются в выходной порт. Это приводит к тому, что сигнал передискретизации на выходе со скоростью, равной скорости выходного порта.

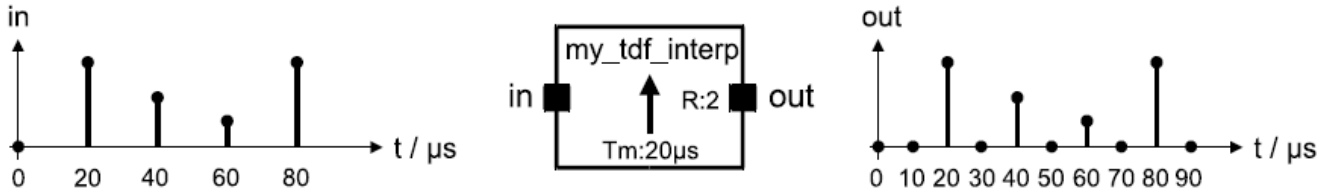


Figure 2.21. Multirate example: 2 times oversampling by inserting zeros

Рисунок 2.21. Пример с несколькими скоростями: 2-кратная передискретизация с добавлением нулей

Как уже обсуждалось в разделе 2.1.3, шаг по времени входного порта, выходного порта и TDF модуля должен быть последовательным. Поскольку шаг по времени модуля установлен на 20 мкс (Tm: 20 мкс), при частоте входного порта 1, выборки на входном порте считываются каждые 20 мкс. Сэмплы на выходном порте записываются с временным шагом 10 мкс.

Этот пример вставляет нули для дополнительных выборок, но другие

методы, такие как линейная интерполяция или выборка и хранение также могут быть реализованы.

```
SCA_TDF_MODULE(my_tdf_interp) {
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
SCA_CTOR(my_tdf_interp) : in("in"), out("out") {}
void set_attributes()
{
out.set_rate(2);
}
void processing()
{
out.write( in.read() ); // input sample directly fed
to the output
out.write( 0.0, 1 ); // insert zero as 2nd sample
}
};
```

На рисунке 2.22 показан пример, который выполняет прореживание входного сигнала так как скорость входного порта выше, чем скорость выходного порта.

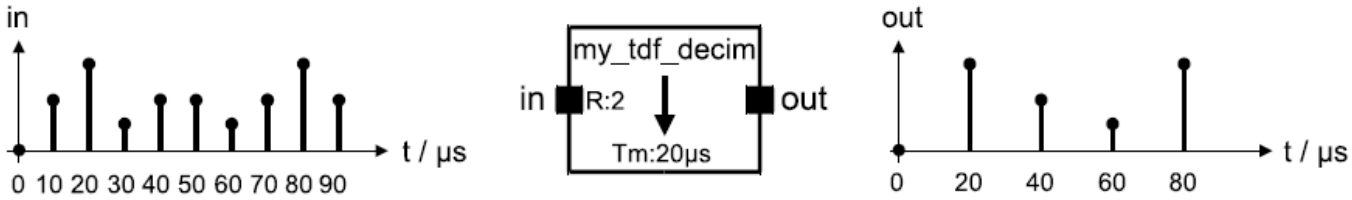


Figure 2.22. Multirate example: Downsampling by a factor of 2

Рисунок 2.22. Пример с несколькими скоростями: понижающая дискретизация в 2 раза

```
SCA_TDF_MODULE(my_tdf_decim)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
SCA_CTOR(my_tdf_decim) : in("in"), out("out") {}
void set_attributes()
{
in.set_rate(2);
}
void processing()
{
```

```
    out.write( in.read() ); // only write the first
sample and neglect the second one
    }
    };
```

2.3.5. Введение задержки

В разделе 2.1.2 объясняются случаи, когда задержки являются существенными в TDF модели. Введение задержек в кластере TDF приведет к вставке отсчетов в начале отсчетов сигналов TDF. Вставленные выборки имеют тот же тип значения, который используется портом TDF и сигналом. Так как начальное значение для обычного типа данных C ++ не определено, и, следовательно, значение вставленного образца не определено, рекомендуется инициализировать эти отсчеты задержки.

На рисунке 2.23 показан базовый TDF модуль, в котором на выходной порт вводится задержка в одну выборку.

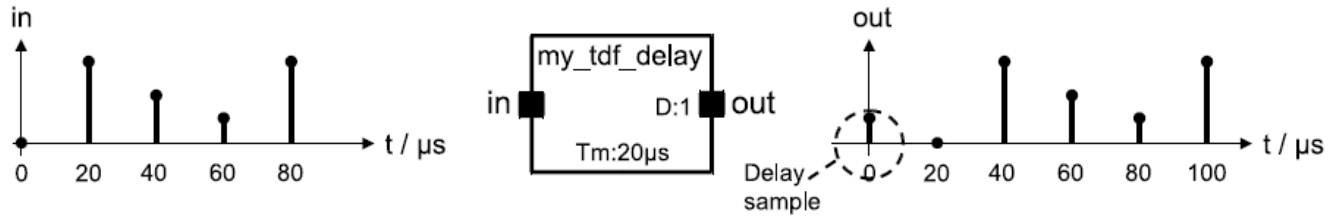


Figure 2.23. TDF module introducing a delay of one sample

Рисунок 2.23. TDF модуль, вводящий задержку одного образца

Реализация этой задержки приведена в следующем примере. В коде видно, что значение задержки также инициализируется значением по умолчанию 1.1.

```
    SCA_TDF_MODULE(my_tdf_delay) {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    SCA_CTOR(my_tdf_delay) : in("in"), out("out") {}
    void set_attributes()
    {
    out.set_delay(1);
    }
    void initialize()
    {
    out.initialize(1.1);
    }
    void processing()
    {
    out.write( in.read() ); // directly write the input
sample to the output (incl the delay)
    }
```

```
};
```

2.4. Взаимодействие между TDF и областью дискретных событий

Как объяснено в разделе 2.1, модель вычисления TDF имеет свои собственные механизмы для аннотации времени, что может привести к разнице во времени между местным временем каждого модуля TDF и временем в домене дискретных событий (время ядра SystemC). Поэтому особое внимание следует уделять синхронизации TDF-сигналов с областью дискретных событий SystemC в обоих направлениях (то есть чтение из и запись в дискретные сигналы событий).

Поддерживать высокую эффективность моделирования, несмотря на наличие взаимодействия TDF и области дискретных событий, используется слабосвязанный механизм синхронизации, который называется синхронизацией данных.

Для моделирования TDF это означает, что дискретные события не будут влиять на активацию и выполнение TDF модулей.

2.4.1. Чтение из области дискретных событий

Для чтения из канала, поступающего из области дискретных событий, входной порт преобразователя TDF должен использоваться класс sca_tdf :: sca_de :: sca_in <T>, см. рисунок 2.24. Для удобства можно использовать короткое название sca_tdf :: sc_in <T>, имя которого sc_in указывает интерфейс для дискретного события SystemC домена. В отличие от обычных входных портов TDF класса sca_tdf :: sca_in <T>, наличие сигнала дискретного события на входных портах преобразователя TDF не активирует («включает») выполнение модуля. Вместо этого порядок активации модуля TDF (расписание) определяется независимо на его индивидуальном временном шаге порта в соответствие со скоростью порта преобразователя и временному шагу TDF модуля.

Условием правильной синхронизации данных является то, что значение, считываемое с порта преобразователя, должно быть доступно в первом дельта-цикле соответствующего момента времени в области дискретных событий. Так как TDF кластер работает независимо от области дискретных событий, может случиться так, что считывается предыдущее значение дискретного события. Это указывает на то, что процесс дискретного события не записал значение в канал до первого дельта - цикл. Это приведет к задержке сигнала. Чтобы преодолеть это, небольшой временной сдвиг может быть введен использованием функции-члена порта set_timeoffset (см. раздел «Атрибуты порта»).

В приведенном ниже примере показано использование модуля TDF, который считывает значения из дискретного события для дальнейшей обработки сигналов TDF и записи их в выходной порт TDF каждую миллисекунду.

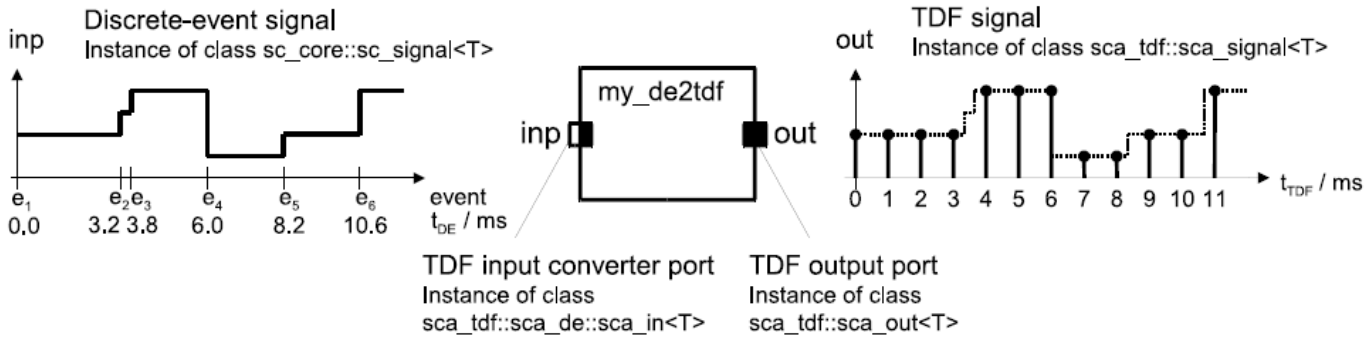


Figure 2.24. TDF module with converter port as input

Рисунок 2.24. Модуль TDF с портом преобразователя в качестве входа

```
    SCA_TDF_MODULE(my_de2tdf)
    {
    sca_tdf::sca_de::sca_in<double> inp; // TDF input
converter port
    sca_tdf::sca_out<double> out; // TDF output port
    SCA_CTOR(my_de2tdf) : inp("inp"), out("out") {}
    void set_attributes()
    {
    set_timestep(1.0, sc_core::SC_MS);
    }
    void processing()
    {
    out.write( inp.read() );
    }
    };
```

2.4.2. Запись в область дискретных событий

Для записи в канал в области дискретных событий следует использовать выходной порт преобразователя TDF класса sca_tdf :: sca_de :: sca_out <T>, см. рис. 2.25. Для удобства можно использовать короткое название sca_tdf :: sc_out <T>, имя которого sc_out напрямую указывает интерфейс к SystemC в область дискретных событий. Смещение по времени и шаг по времени, назначенные порту выходного преобразователя, определяют, при котором моменте времени и интервале времени значение записывается в область дискретных событий.

Условием правильной синхронизации данных является возможность записи образца, записанного в порт преобразователя, к связанному каналу в первом дельта-цикле соответствующей временной точки дискретного события. В случае, если канал класса sc_core :: sc_signal <T> подключен к порту преобразователя, генерируется только дискретные события в случае изменения сигнала, как указано событиями e1, e2 и e3. В случае, если канал класса sc_core :: sc_buffer <T> подключен к порту конвертера, все сэмплы, записанные в порт,

сгенерируют событие, которое указывается с помощью дополнительных отсчетов e11, e12, e13 и т. д.

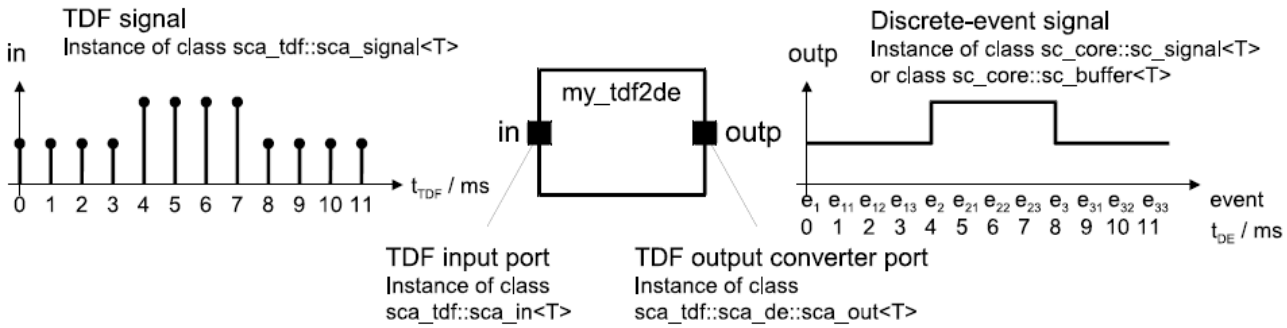


Figure 2.25. TDF module with a converter port as output

Рисунок 2.25. Модуль TDF с выходным портом преобразователя

В приведенном ниже примере показана реализация модуля TDF, который записывает сэмплы в область дискретных событий.

```
SCA_TDF_MODULE(my_tdf2de)
{
sca_tdf::sca_in<double> in; // TDF input port
sca_tdf::sca_de::sca_out<double> outp; // TDF output
converter port
SCA_CTOR(my_tdf2de) : in("in"), outp("outp") {}
void set_attributes()
{
set_timestep(1.0, sc_core::SC_MS);
}
void processing()
{
outp.write( in.read() );
}
};
```

2.4.3. Использование дискретных управляющих сигналов

В приведенном ниже примере показан простой усилитель с цифровым управлением, в котором коэффициент усиления определяется внешним управляющим сигналом из области дискретных событий. Частота выполнения функции-члена **processing** определяется временным шагом модуля, равным 1 мс. Каждый раз, когда функция **processing** (обработки) вызывается, данные из области дискретных событий считываются.

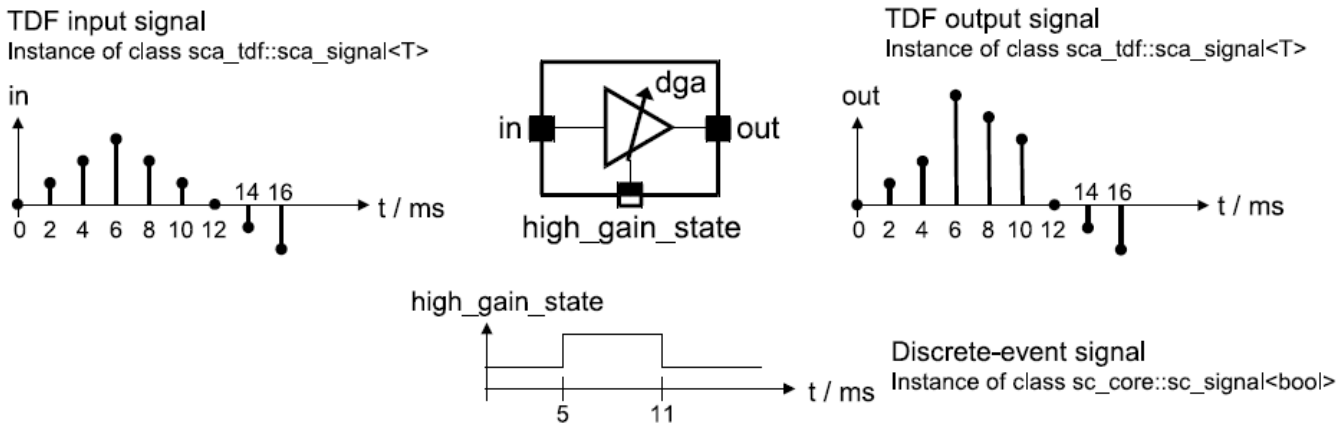


Рисунок 2.26. TDF модуль с портом преобразователя, используемым в качестве управляющего входа

```
    SCA_TDF_MODULE(my_dga)
    {
    sca_tdf::sca_in<double> in; // input port
    sca_tdf::sca_out<double> out; // output port
    // control signal from the discrete-event domain
    sca_tdf::sca_de::sca_in<bool> high_gain_state; //
input converter port
    SCA_CTOR(my_dga)
    : in("in"), out("out"),
high_gain_state("high_gain_state"),
    high_gain(100.0), low_gain(1.0) {}
    void set_attributes()
    {
    set_timestep(1.0, sc_core::SC_MS);
    }
    void processing()
    {
    double gain = high_gain_state.read() ? high_gain :
low_gain;
    out.write( gain * in.read() );
    }
    private:
    double high_gain, low_gain;
    };
```

2.5. Семантика исполнения TDF

В дополнение к этапам разработки и моделирования, как это определено в стандарте языка SystemC IEEE 1666-2005, специфическая функциональность

реализована для разработки и исполнения моделей TDF.

Основные функции-члены модуля TDF для моделирования во временной области: set_attributes, initialize и processing. Пользователь должен перегрузить эти функции-члены для осуществления инициализации и инициализации поведения обработки сигнала, определенного пользователем модуля TDF. Не разрешается вызывать функции – члены непосредственно.

Как показано на рисунке 2.27, этап разработки включает в себя следующие этапы:

• Настройки атрибутов модуля TDF: выполнить функцию-член set_attributes всех модулей TDF.

• Расчет и распространение временного шага TDF: распространение и вычисление неназначенного времени порта и шагов модуля на основе назначенных временных шагов и скорости портов. (см. раздел 2.1.3).

• Проверка совместимости кластера TDF: определите и проверьте расписание кластера.

Шаги для этапа моделирования:

• Инициализация модуля TDF: выполнить (необязательно) функцию-член инициализации всех модулей TDF.

• Активация и обработка модуля TDF: непрерывно выполнять обработку функций-членов каждого модуль TDF, пока все образцы не будут обработаны.

• Постобработка модуля TDF: выполнение (необязательной) функции-члена end_of_simulation всех модули TDF. Обратите внимание, что эта функция-член не является специфичной для AMS, но наследуется от SystemC модуля базового класса.

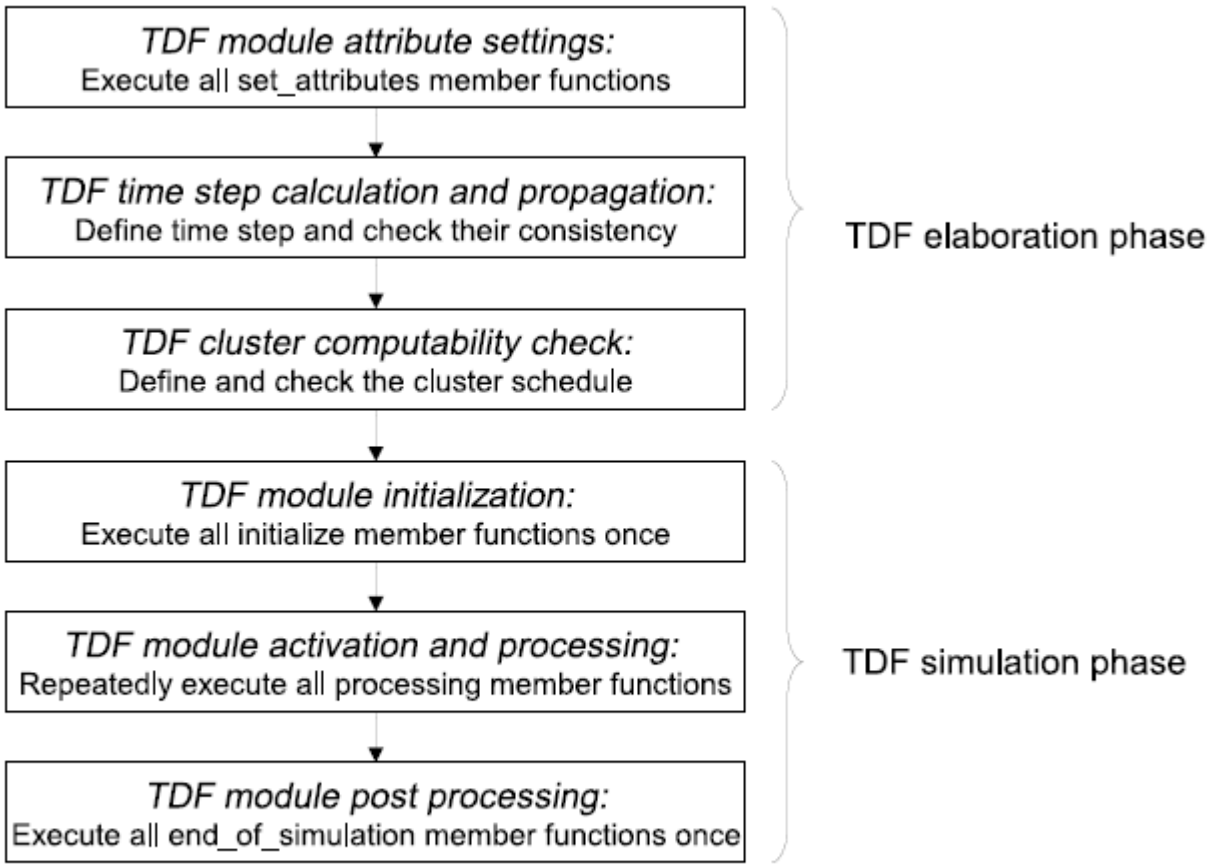


Figure 2.27. TDF elaboration and simulation phases

Рисунок 2.27. TDF этапы разработки и моделирования

Этапы разработки и моделирования выполняются путем запуска моделирования во временной области с использованием функции sc_core :: sc_start. Это объясняется в разделе 6.1.1.

2.6. Примеры моделирования временных потоков данных TDF

2.6.1. Формирование непрерывного синусоидального сигнала

Исследуем аналоговую и цифровую обработки сигналов с использованием SystemC AMS 2.0.

В Lab1 есть 3 неполных модуля AMS, которые вы должны заполнить и исправить (подход «заполнить пробелы», мини-задачи).

Лабораторная работа 1a: создание синусоидального источника (используйте sin_source_with_noise.h и sin_source_with_noise.cpp).

Лабораторная работа 2b: предварительный фильтр (prefilter.h и prefilter.cpp).

Лабораторная работа 2c: adc_sd (adc_sd.h и adc_sd.cpp).

Лабораторная работа 2d: comb_filter (comb_filter.h и comb_filter.cpp).

Программа для формирования синусоидального сигнала показана на листингах 2.1 – 2.3.

Листинг 2.1

Заголовочный файл sin_sousce_h

```
    Original Author: Karsten Einwich Fraunhofer IIS/EAS
Dresden
    //
    //   Created on: 16.02.2010
    //
//-------------------------------------------------------
                    --------------


    #ifndef SIN_SOURCE_H
    #define SIN_SOURCE_H

    #include <systemc-ams.h>                    // SystemC
AMS header

    SCA_TDF_MODULE(sin_source)    // Declare a TDF module
    {
      sca_tdf::sca_out<double> out;             // TDF
output port

      //parameter
      double ampl;       //   amplitude
      double freq;       //   frequency

      void set_attributes();    // Set TDF attributes

      void processing();                        //
Describe time-domain behaviour

      SCA_CTOR(sin_source)                      //
Constructor of the TDF module
      : out("out"),                             // Name
the port(s)
        ampl(1.0), freq(1e3) {}                 // Initial
values for ampl and freq

    };

    #endif /* SIN_SOURCE_H */
```

Листинг 2.2

Исполняемый файл `sin_source.cpp`

```
    #include "sin_source.h"
    #include <cmath>
// for std::sin

    void sin_source::set_attributes()
// Set TDF attributes
    {
      out.set_timestep(1.0, SC_US);
// Set time step of output port
    }

    void sin_source::processing()
// Describe time-domain behaviour
    {
      double t = out.get_time().to_seconds();
// Get current time of the sample
      double x = ampl * std::sin(2.0 * 3.1415 * freq *
t);  // Calculate sine wave
      out.write(x);
// Write sample to the output
                              }
```

Листинг 2.3

Испытательный стенд `Testbench.cpp`

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
------------------

    #include "sin_source.h"

    int sc_main(int argn, char* argc[])          //
SystemC main program
    {
      sca_tdf::sca_signal<double> sig_1;         //
Signal to connect source w sink
```

```
      sin_source src_1("src_1");                  //
Instantiate source
        src_1.out(sig_1);                         //
Connect (bind) with signal

      sca_trace_file* tfp =           // Open trace file
        sca_create_vcd_trace_file("testbench");

      sca_trace(tfp, sig_1, "sig_1");             //
Define which signal to trace

      sc_start(10.0, SC_MS);      // Start simulation for
100 ms

      sca_close_vcd_trace_file(tfp);  // Close trace file

      return 0;             // Exit with return code 0
    }
```

Примечания:

1. В исходных файлах выполнялась трассировка `tabular`. Однако, открытие файлов с расширением SDF затруднительно и требует установки дополнительных программ. Поэтому мы заменили в трассировке формат `tabular` на формат графиков `VCD`.

2. В исходном файле постоянная M_PI заменена значением 3.1415.

Результаты компиляции проекта показаны на рис. 2.28.

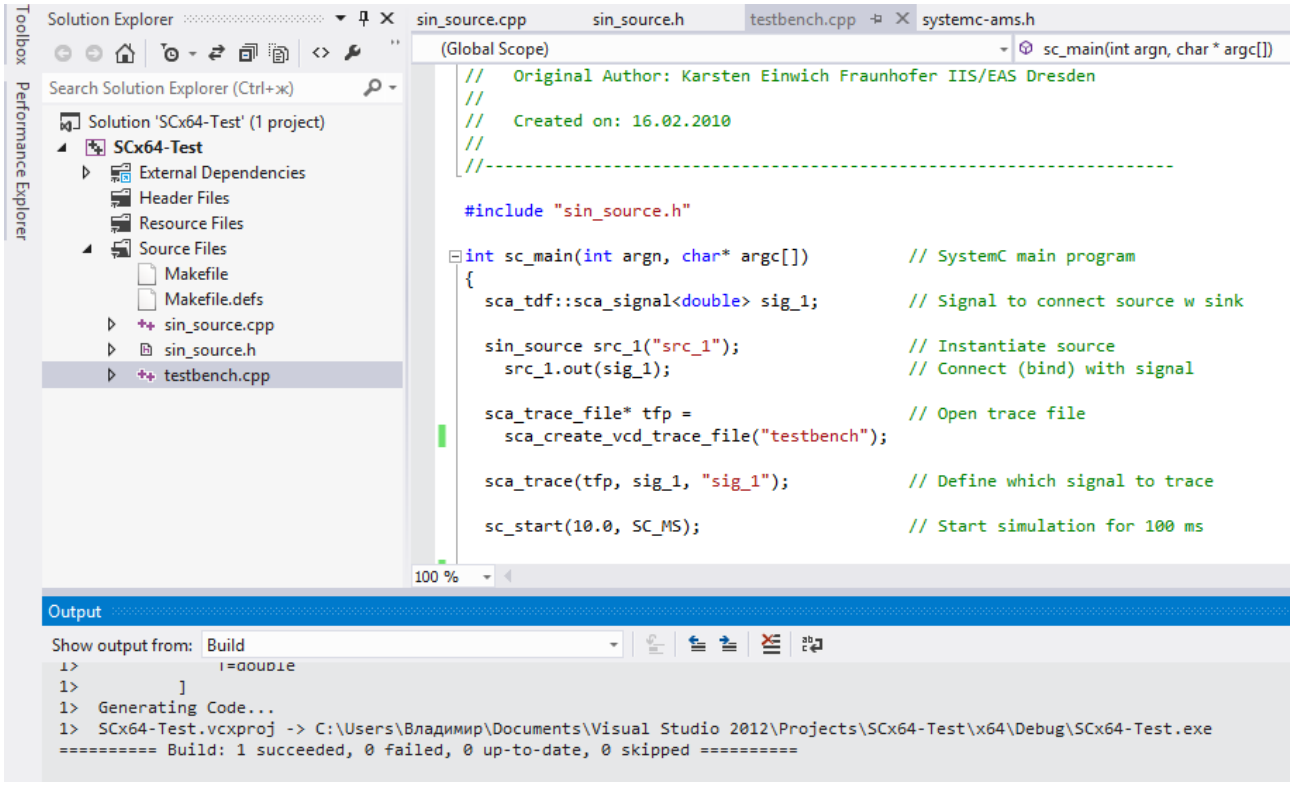


Рис. 2.28. Результаты компиляции проекта

Выполним решение, выбрав Debug – Start Debugging. Результатом решения график в виде файла с расширением VCD. На рис. 2.29 показано расположение этого файла в папке проекта. Так VCD – файлы сохраняются следует идентифицировать их по времени создания.

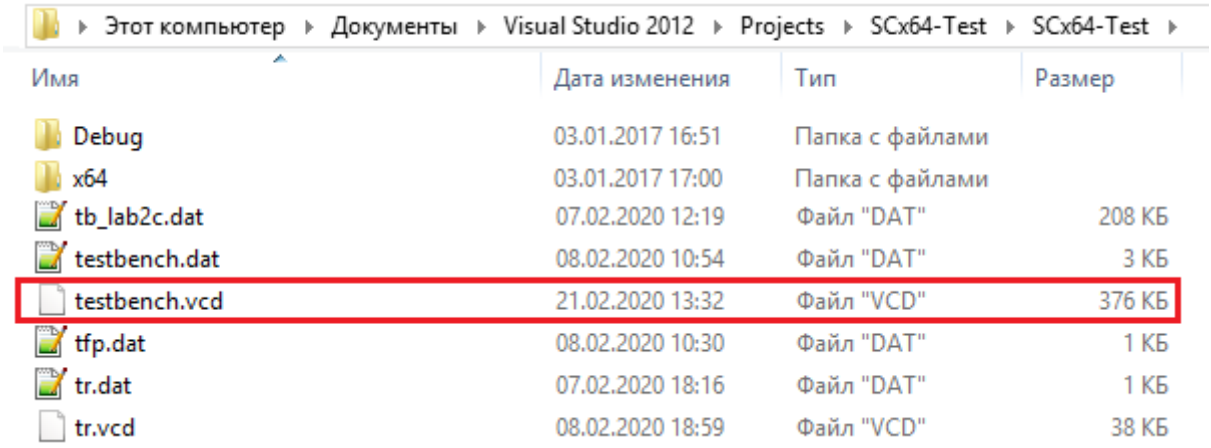


Рис. 2.29. Расположение VCD – файла в проекте

Для удобства открытия VCD файлов в программе GTKWave рекомендуем создать для их хранения дополнительную папку C:/VCD-files.

Далее открываем программу GTKWave. В меню File выбираем Open New Tab. На диске C находим папку VCD-files и нажимаем Insert (Рис. 2.30).

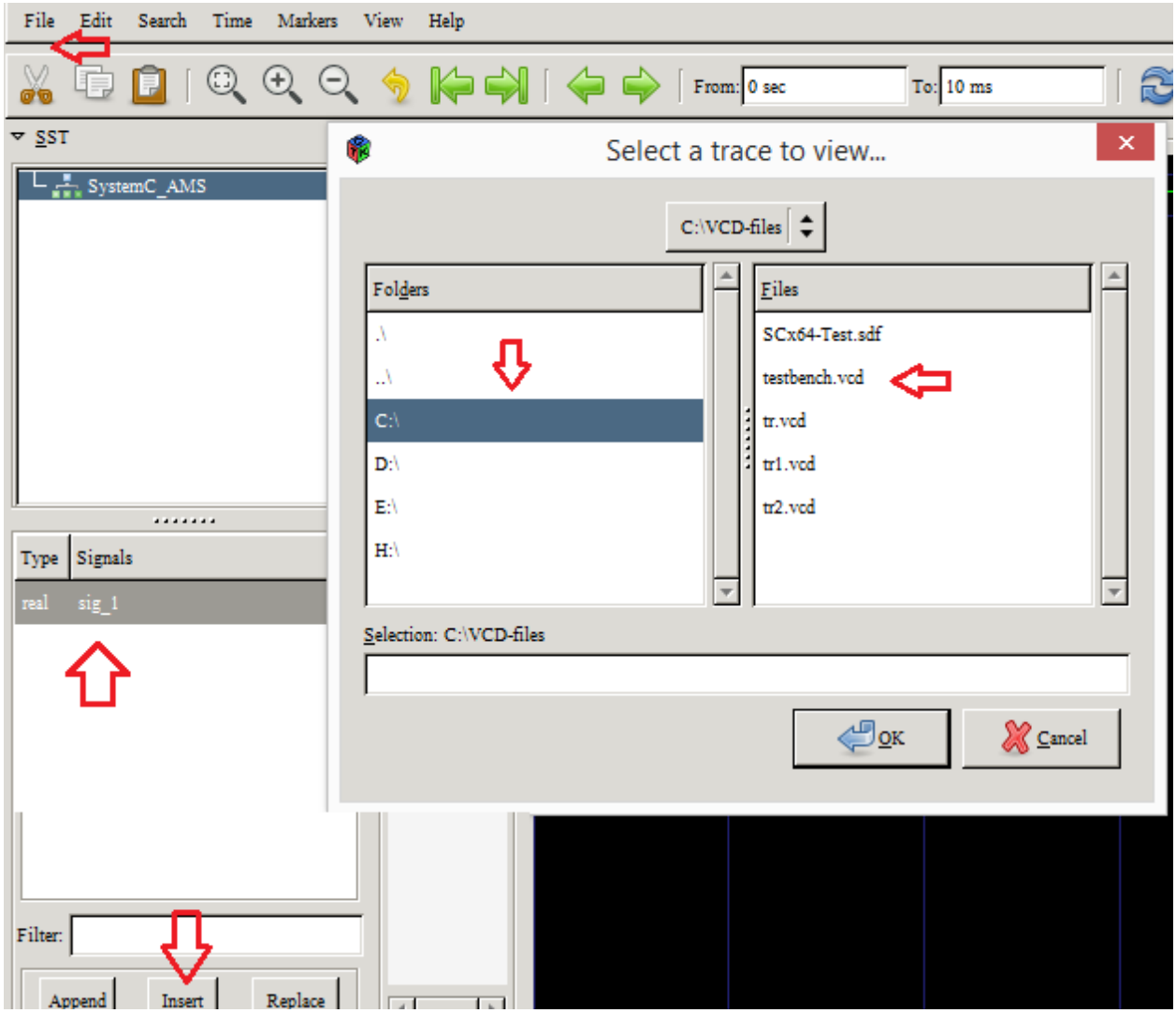


Рис. 2.30. Открытие VCD – файла.

Далее устанавливаем оптимальный масштаб времени. Выполняем: Time – Zoom – Zoom Best Fit (рис. 2.31)

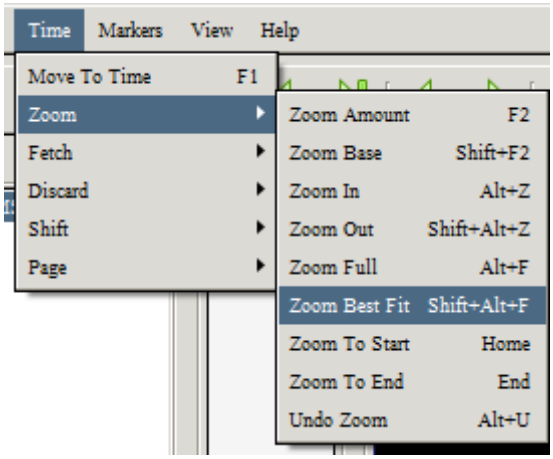


Рис. 2.31. Установка оптимального масштаба времени

Далее устанавливаем формат сигнала. Для аналогового сигнала выбираем Data Format – Analog – Interpolated (рис. 2.32).

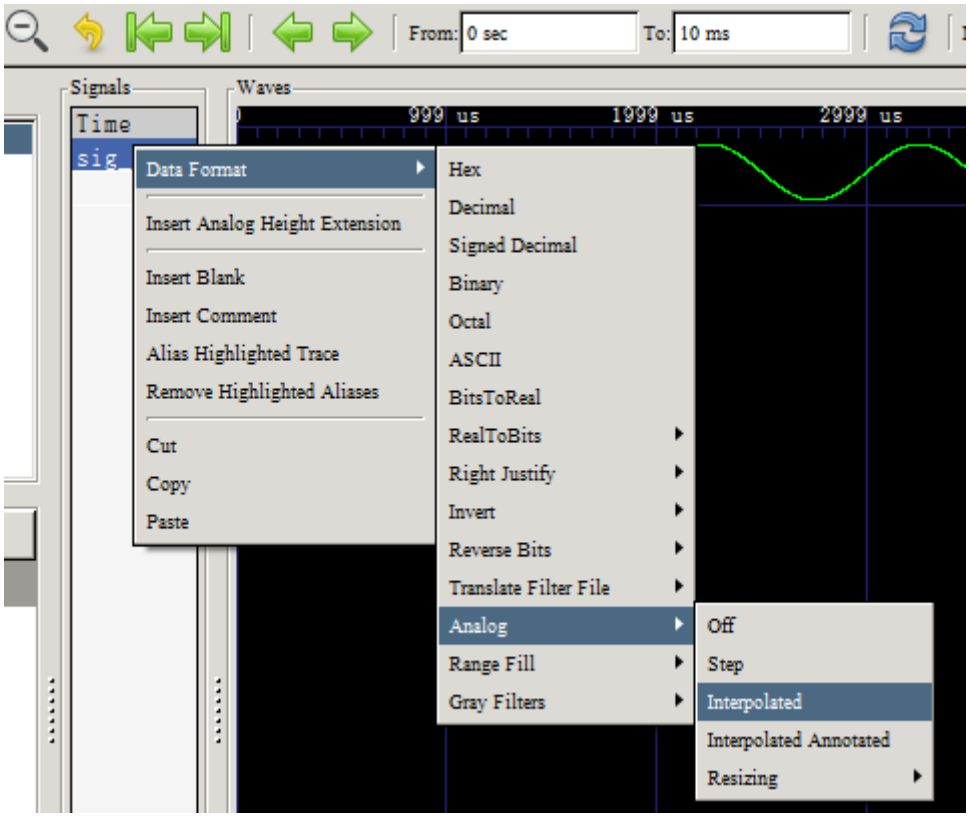


Рис. 2.32. Изменение формата сигнала

Увеличить амплитуду сигнала можно последовательно выполняя Insert Analog Height Extension (рис. 2.33).

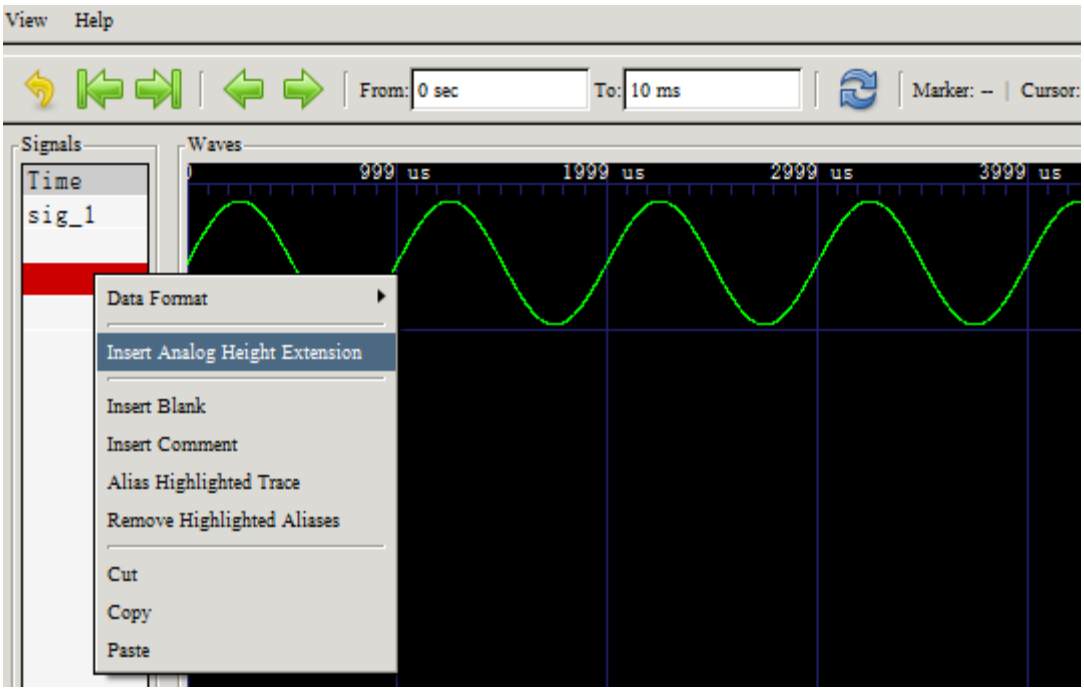


Рис. 2.33. Увеличение амплитуды аналогового сигнала

Изменим шаг временной шаг выходного порта. Для этого в исполняемом файле `sin_source.cpp` установим временной шаг 100 мкс:

```
// Set TDF attributes
```

```
    {
      out.set_timestep(100.0, SC_US);
// Set time step of output port
    }
```

Выполним моделирование. Вид графика показан на рис. 2.37.

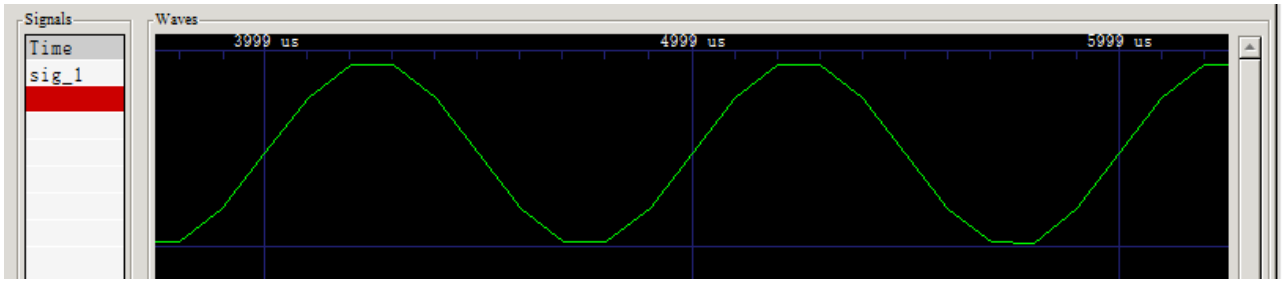


Рис. 2.37. График функции с частотой 1 кГц и с выходным шагом 100 мкс

Заметна более грубая временная дискретизация функции.

2.6.2. Вывод на печать и трассировка синусоидального сигнала

Рассмотрим более общий пример моделирования синусоидального сигнала. Схема модели показана на рис. 2.38.

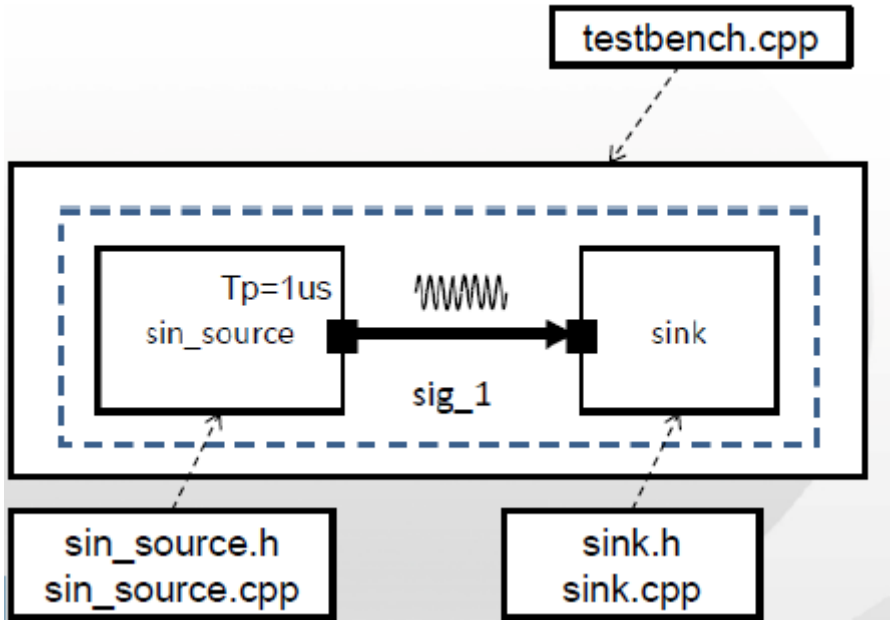


Рис. 2.38. Схема генератора синусоидального сигнала с стоком

Листинги программных файлов представлены в 2.4 – 2.8.

Листинг 2.4

Заголовочный файл SIN_SOURCE_H

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-------------------------------------------------
------------------

    #ifndef SIN_SOURCE_H
    #define SIN_SOURCE_H

    #include <systemc-ams.h>    // SystemC AMS header

    SCA_TDF_MODULE(sin_source)  // Declare a TDF module
    {
      sca_tdf::sca_out<double> out;  // TDF output port

      //parameter
      double ampl;       //   amplitude
      double freq;      //   frequency

      void set_attributes();                    // Set TDF
attributes

      void processing();   // Describe time-domain
behaviour

      SCA_CTOR(sin_source)  // Constructor of the TDF
module
      : out("out"),   // Name the port(s)
        ampl(1.0), freq(1e3) {}    // Initial values for
ampl and freq

    };

    #endif /* SIN_SOURCE_H */
```

Листинг 2.5

Исполняемый файл `sin_source.cpp`

```
// Original Author: Karsten Einwich Fraunhofer IIS/EAS Dresden
//
// Created on: 16.02.2010
```

```
                                //
    #include "sin_source.h"
    #include <cmath>        // for std::sin

    void sin_source::set_attributes()  // Set TDF
attributes
    {
      out.set_timestep(1.0, SC_US); // Set time step of
output port
    }

    void sin_source::processing()  // Describe time-
domain behaviour
    {
      double t = out.get_time().to_seconds(); // Get
current time of the sample
      double x = ampl * std::sin(2.0 * 3.1415 * freq *
t);  // Calculate sine wave
      out.write(x);   // Write sample to the output
    }
```

Листинг 2.6

Заголовочный файл SINK_H

```
    Original Author: Karsten Einwich Fraunhofer IIS/EAS
Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
-------------------

    #ifndef SINK_H
    #define SINK_H

    #include <systemc-ams.h>     // SystemC AMS header

    SCA_TDF_MODULE(sink)
    {
      sca_tdf::sca_in<double> in;  // TDF input port

      SCA_CTOR(sink)     // Constructor of the TDF module
      : in("in")               // Name the port(s)
      {}

      void processing(); // Describe time-domain
behaviour
```

```
};

#endif // SINK_H
```

Листинг 2.7

Исполняемый файл sink.cpp

```
#include "sink.h"
#include <iostream>   // for std::cout and std::endl

void sink::processing()  // Describe time-domain
behaviour
{
  std::cout << this->name()  // Display name of
module
            << " @ "
            << this->get_time() // Show module time
            << ": "
            << in.read()   // Read value from input
port
            << std::endl;    // and display it
}
```

Листинг 2.8

Исполняемый файл testbench.cpp

```
#include "sin_source.h"
#include "sink.h"
#include "conio.h"


int sc_main(int argn, char* argc[])      // SystemC
main program
{
  sca_tdf::sca_signal<double> sig_1;         //
Signal to connect source w sink

  sin_source src_1("src_1");                 //
Instantiate source
    src_1.out(sig_1);                        //
Connect (bind) with signal

  sink sink_1("sink_1");                     //
Instantiate sink
```

```
        sink_1.in(sig_1);                          //
Connect (bind) with signal

      sca_trace_file* tfp =       // Open trace file
        sca_create_vcd_trace_file("testbench");
        sca_trace(tfp, sig_1, "sig_1");            //
Define which signal to trace

      sc_start(10.0, SC_MS);                       // Start
simulation for 10 ms

      sca_close_vcd_trace_file(tfp);          // Close
trace file
    _getch();
      return 0;                                    // Exit
with return code 0
    }
```

Как и в предыдущем примере мы заменили `tabular` на `vcd`, и вместо постоянной M_PI ввели значение 3.1415. Временной шаг установлен 10 мкс.

Кроме того, в программе испытательного стенда для вывода результатов в консоль добавлено `#include "conio.h"` и `_getch();`.

В таблице (рис. 2.39) показан фрагмент выходных данных с шагом 10 мкс.

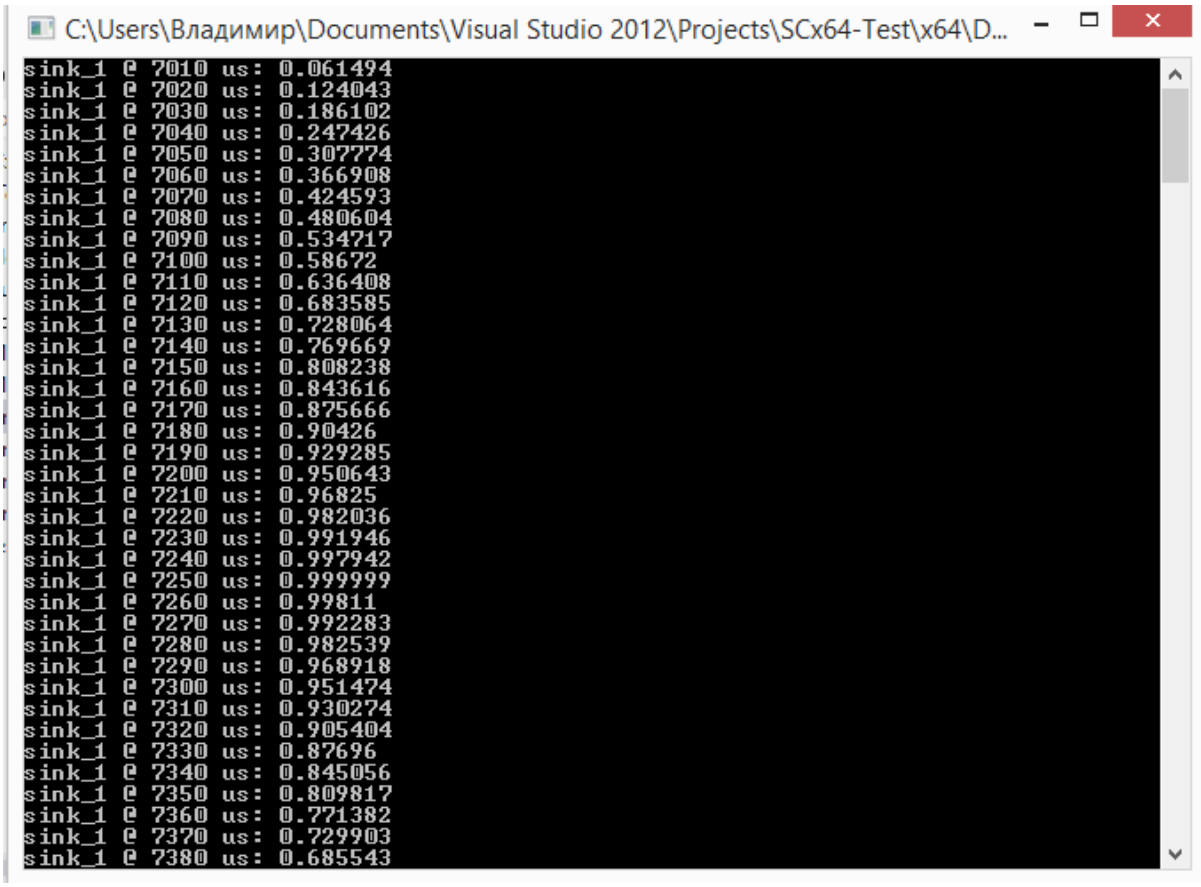


Рис. 2.39. Фрагмент выходных данных

На рис. 2.40 показана трассировка функции.

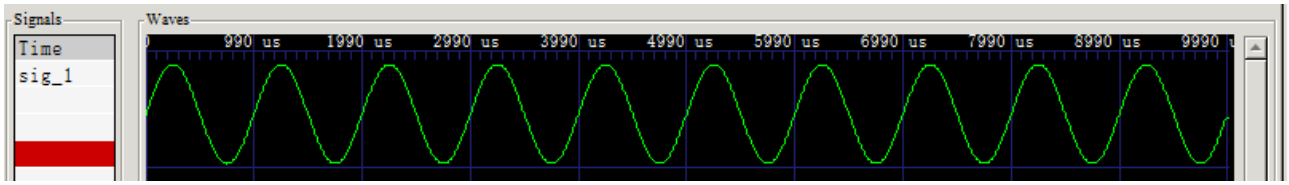


Рис. 2.40. Трассировка синусоидальной функции

В атрибутах синусоидального генератора установим шаг выходного порта 100 мкс (out.set_timestep(100.0, SC_US). В GTKWave установим Data Format > Analoge-Step . Результат моделирования показан на рис. 2.41.

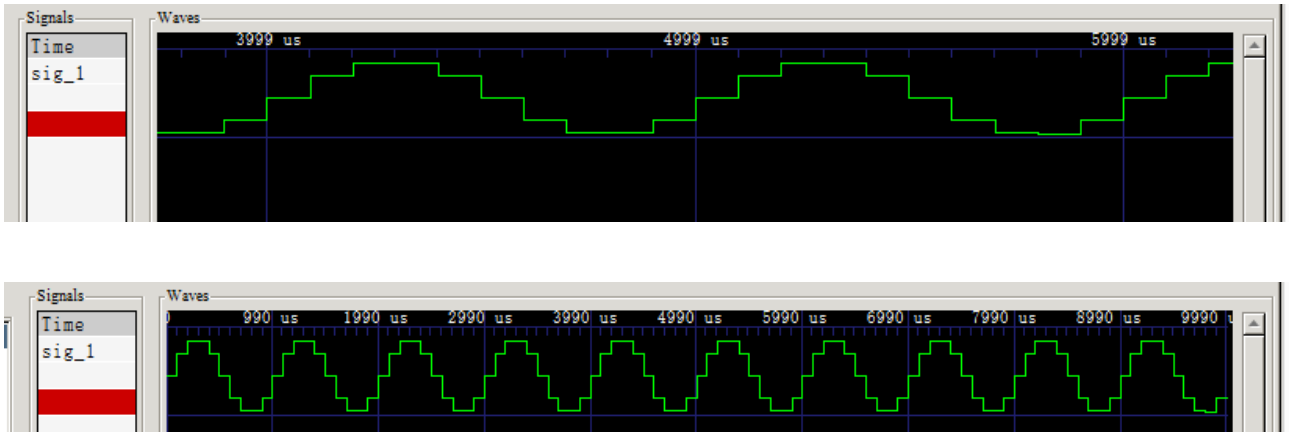


Рис. 2.41. Результаты моделирования с шагом 100 мкс

2.6.3. Изменение атрибутов порта

Измененим параметры в атрибутах выходного порта источника гармонического сигнала. Установим скорость rate =2, задержку delay = 2:

```
    void set_attributes()
    {
    out.set_timestep(0.01, sc_core::SC_US); // set time
step of port out
    out.set_rate(2); // set rate of port out to 2
    out.set_delay(2); // set delay of port out to 2
samples
    }
```

Результаты моделирования в течение 10 мс для разных значений скорости порта, задержки, шага по времени показаны на рис. 2.42 – 2.44.

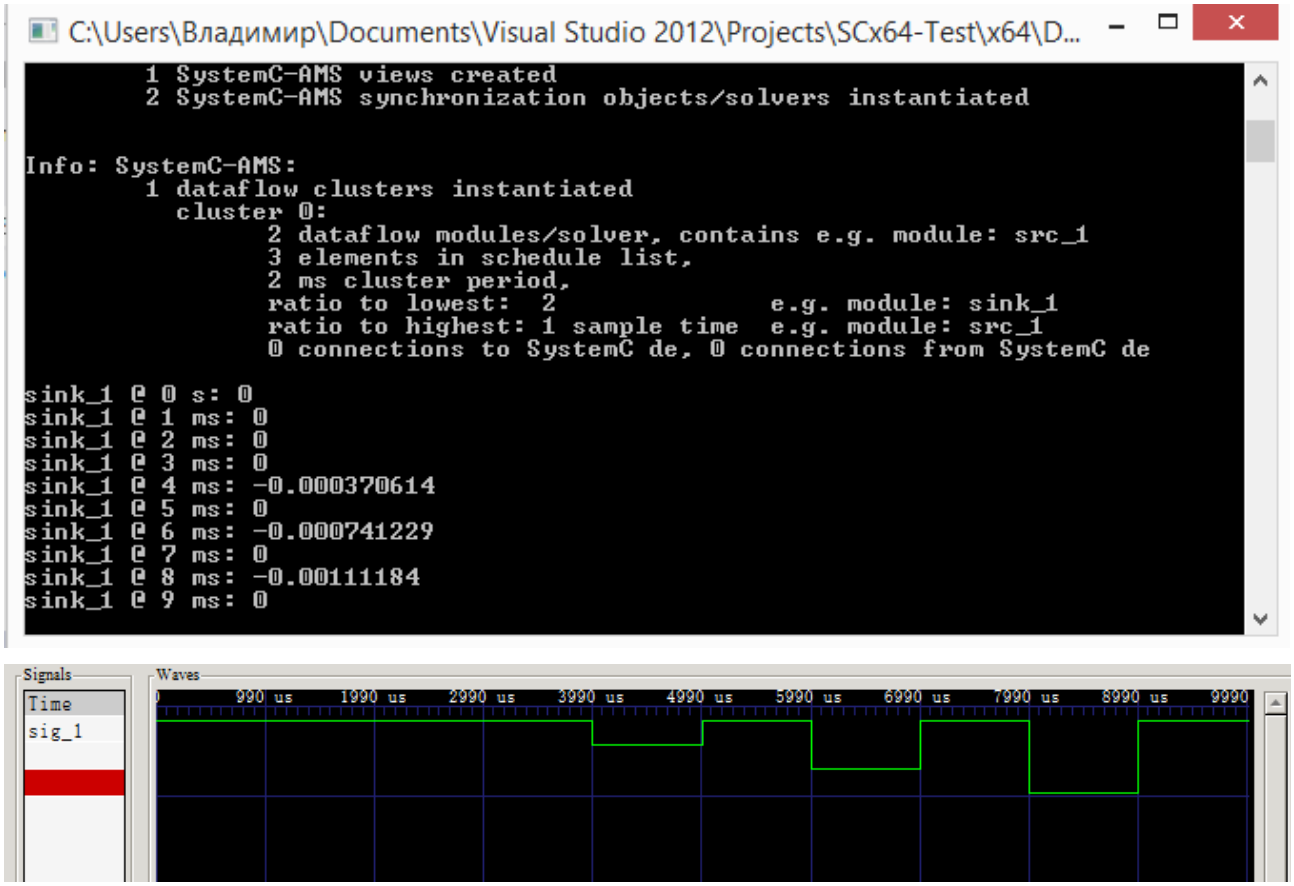


Рис. 2.42. Rate 2, delay 2, timeset = 1ms, start 10 ms

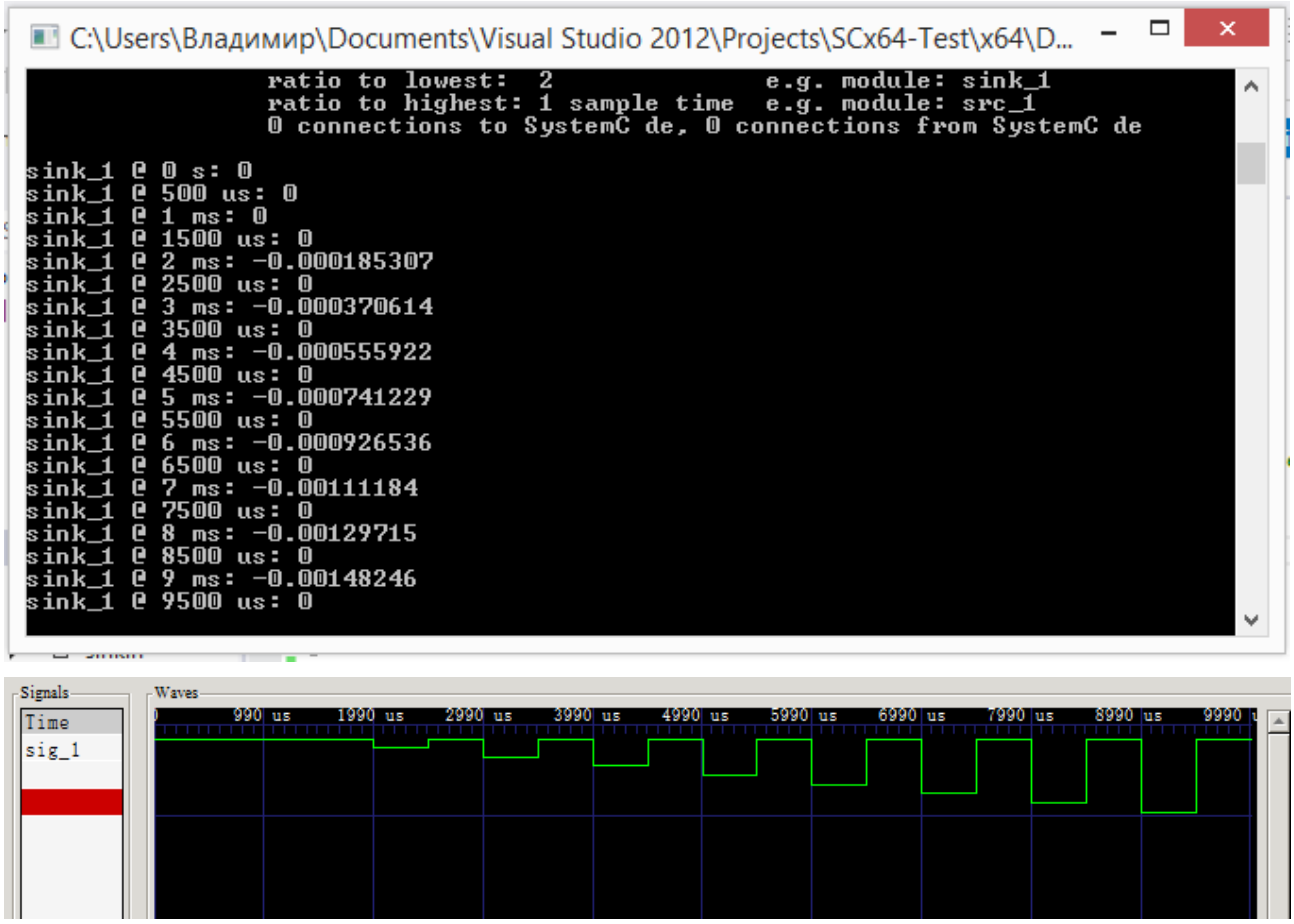


Рис. 2.43. Rate 2, delay 2, timeset = 500 US, start 10 ms

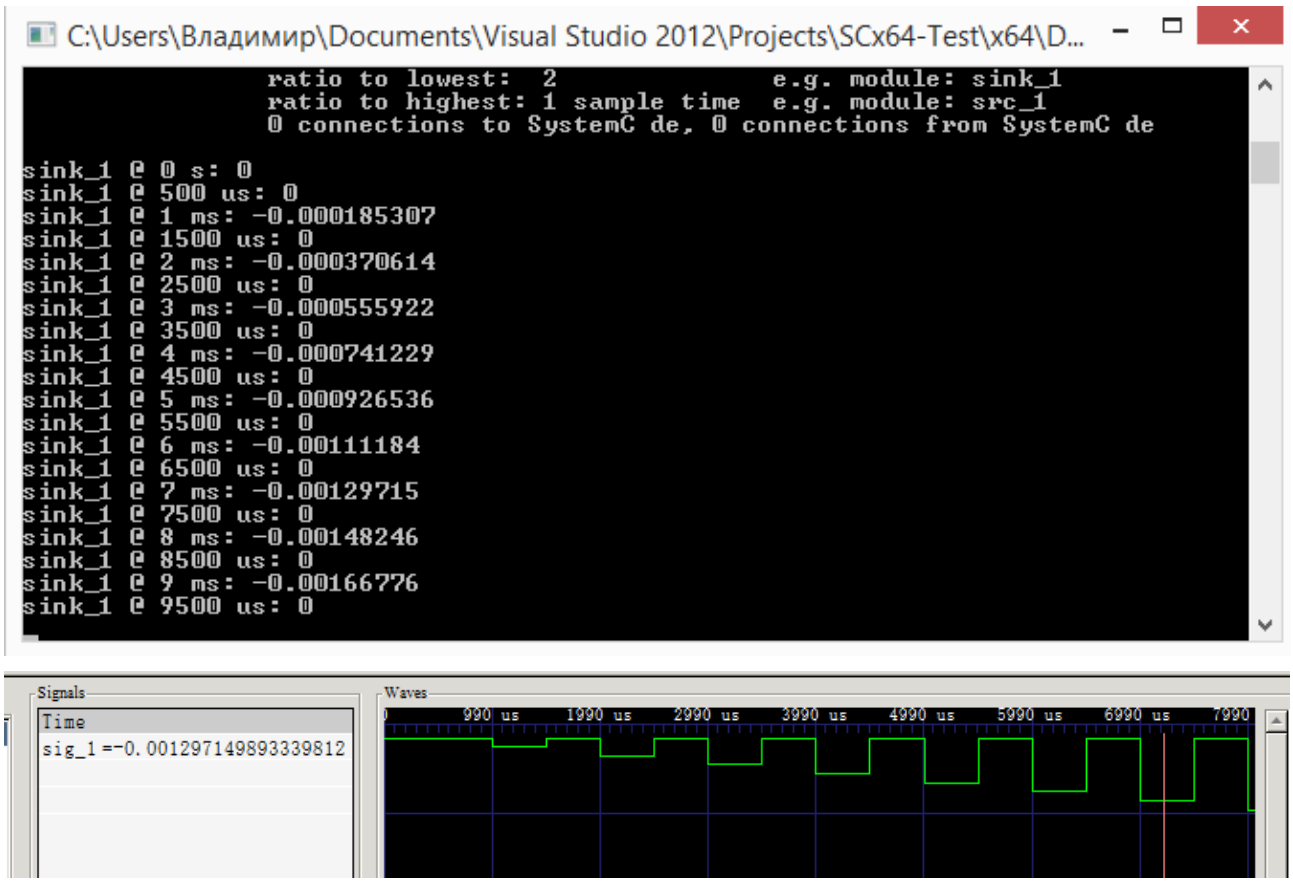
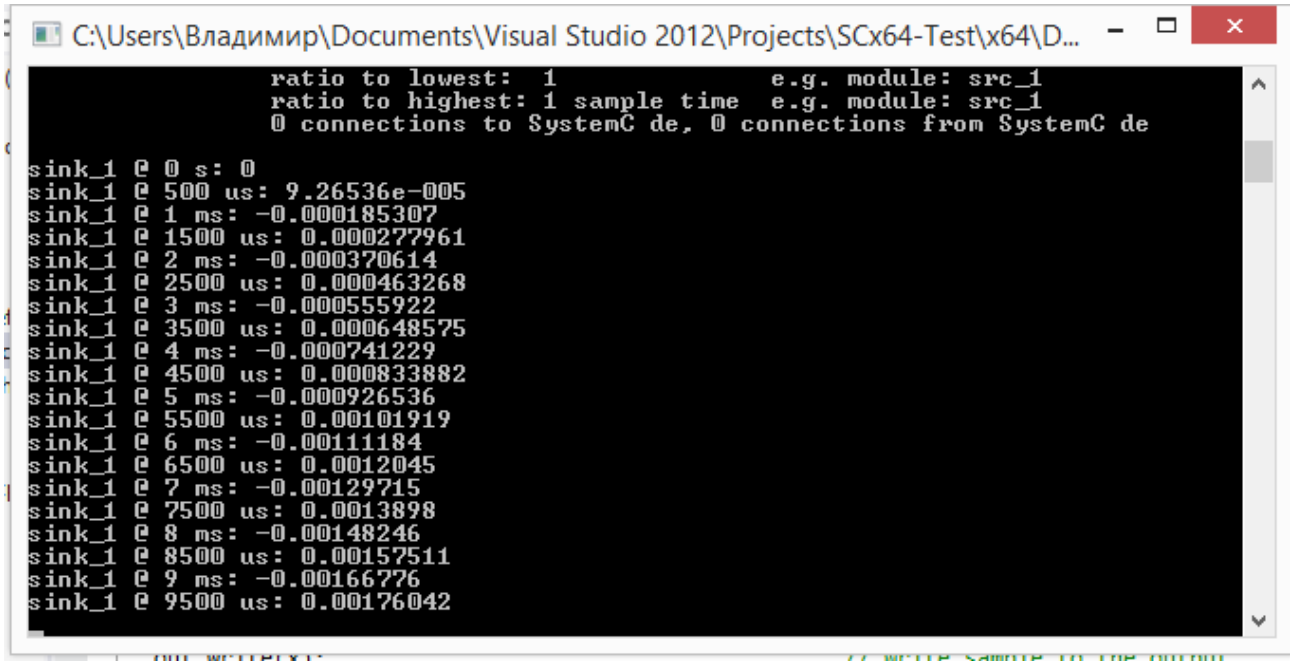


Рис. 2.44. Rate 2, delay 0, timeset = 500 US, start 10 ms



Выполним моделирование с шагом 250 мкс = T/4 для скорости порта Rate 1 и delay 0. Результаты показаны на рис. 2.45.

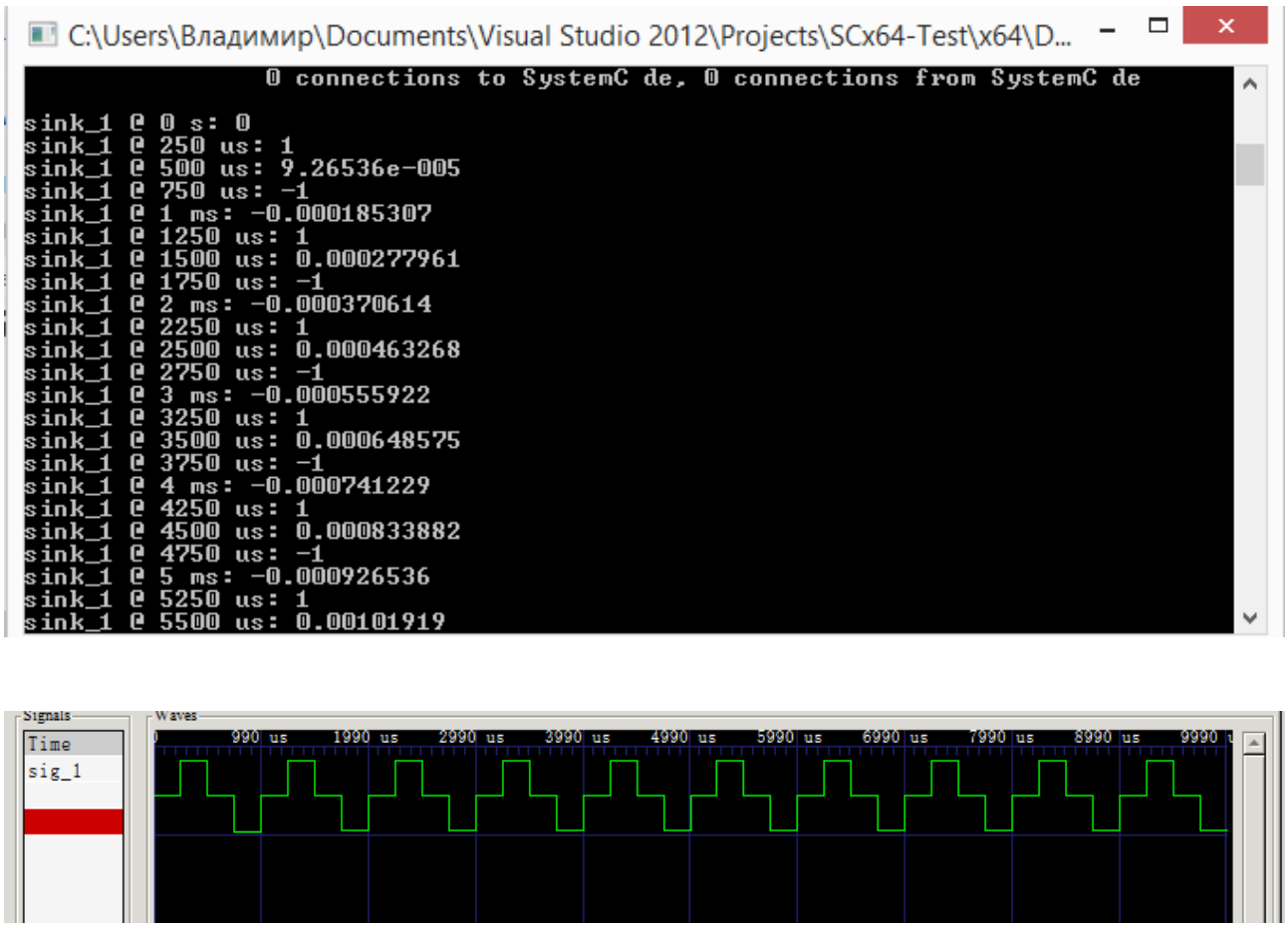


Рис. 2.45. Rate 1, delay 0, timeset = 250 US, start 10 ms

2.7. Формирование последовательности импульсов

В исполняемом файле `sin_source.cpp` (Листинг 2.5) изменим формируемую функцию так:

```
    void sin_source::processing()
// Describe time-domain behaviour
    {
      {
          out.write( (bool)(true) );
    }
    }
```

Установим шаг 250 мкс, rate =2. При моделировании получим последовательности импульсов (рис. 2.46):

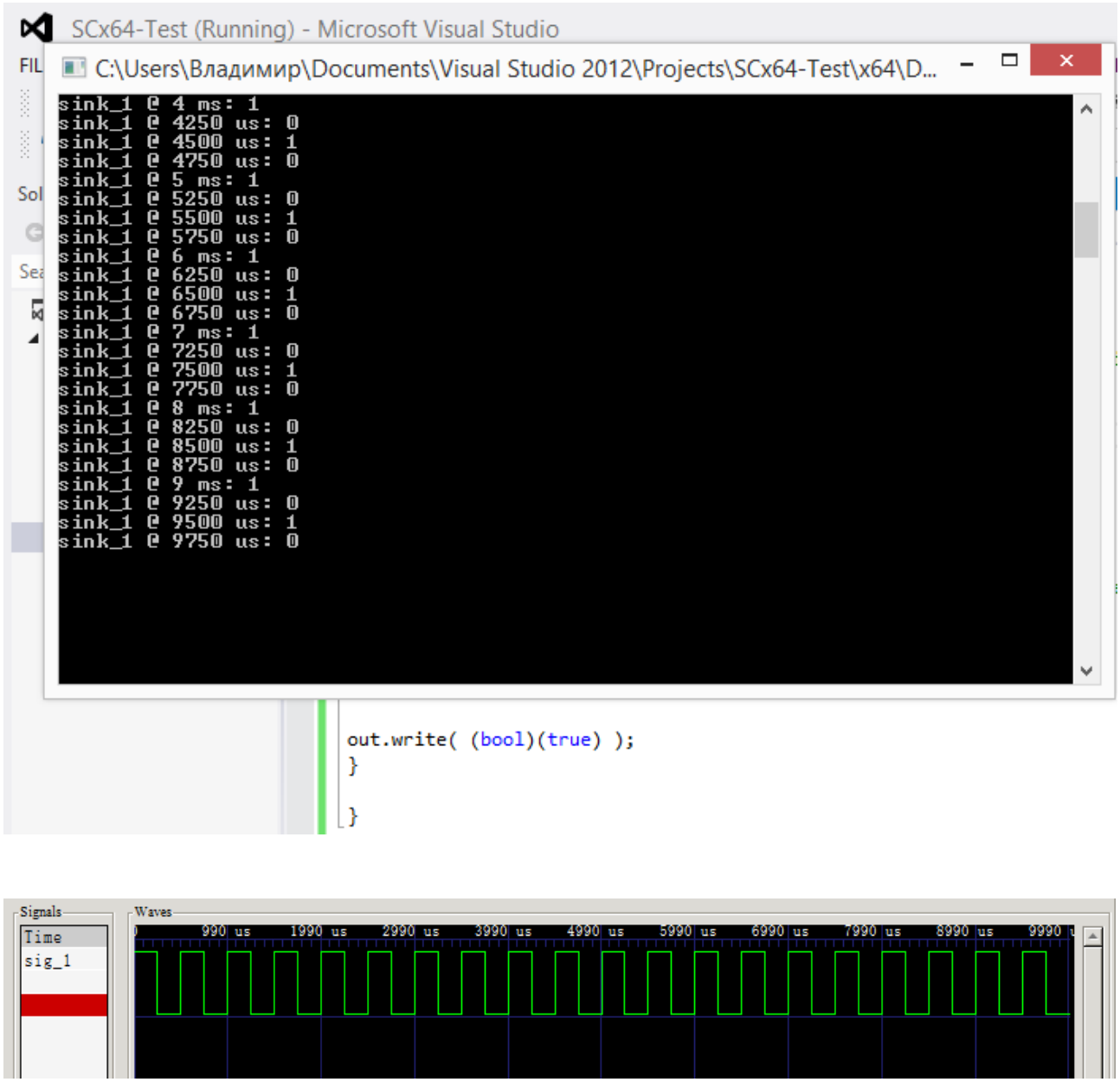


Рис. 2.46. Последовательность импульсов для шага 250 мкс, rate =2.

На рис. 2.47 показаны импульсы для атрибутов : шаг 250 мкс, rate = 4.

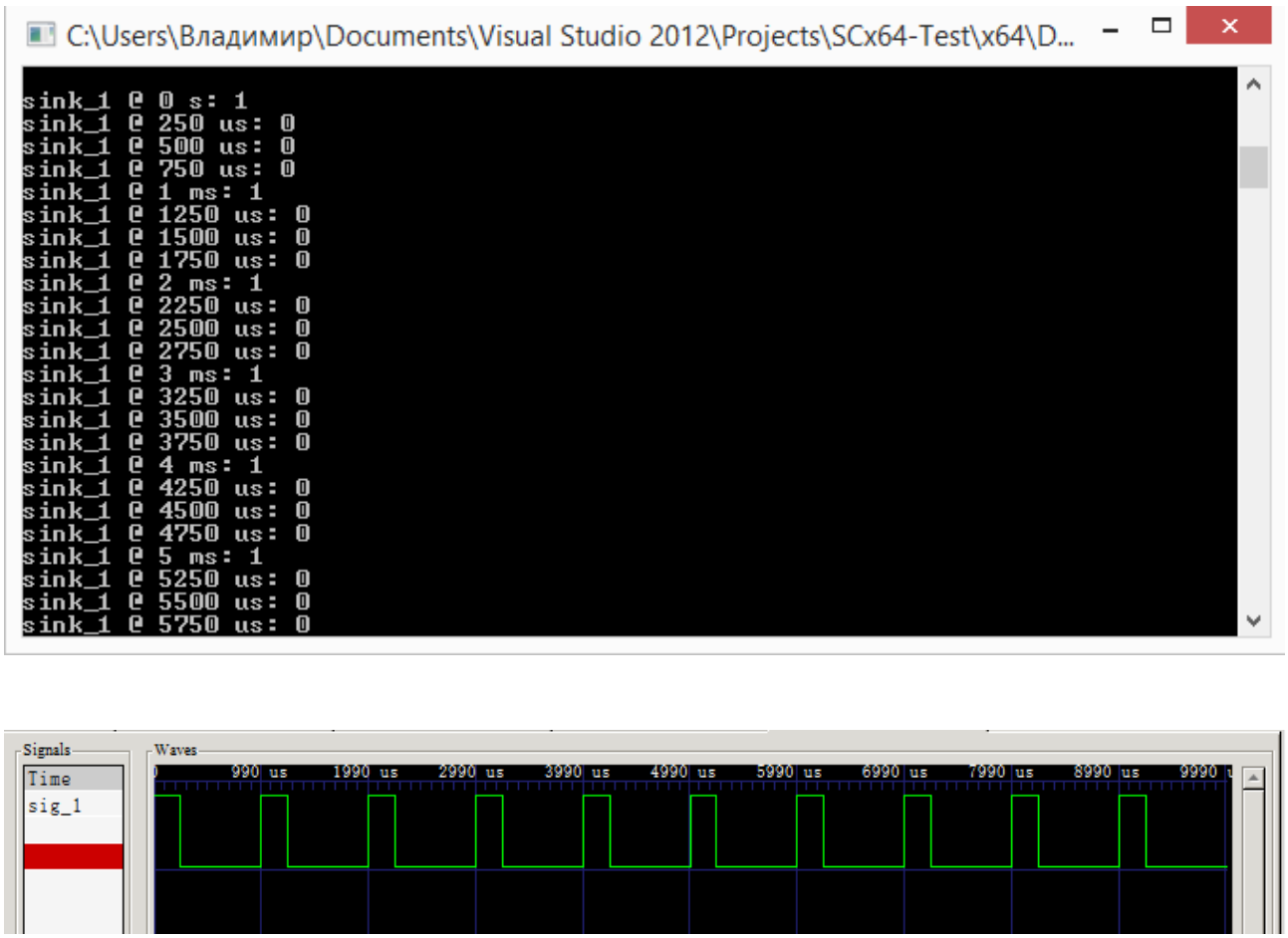
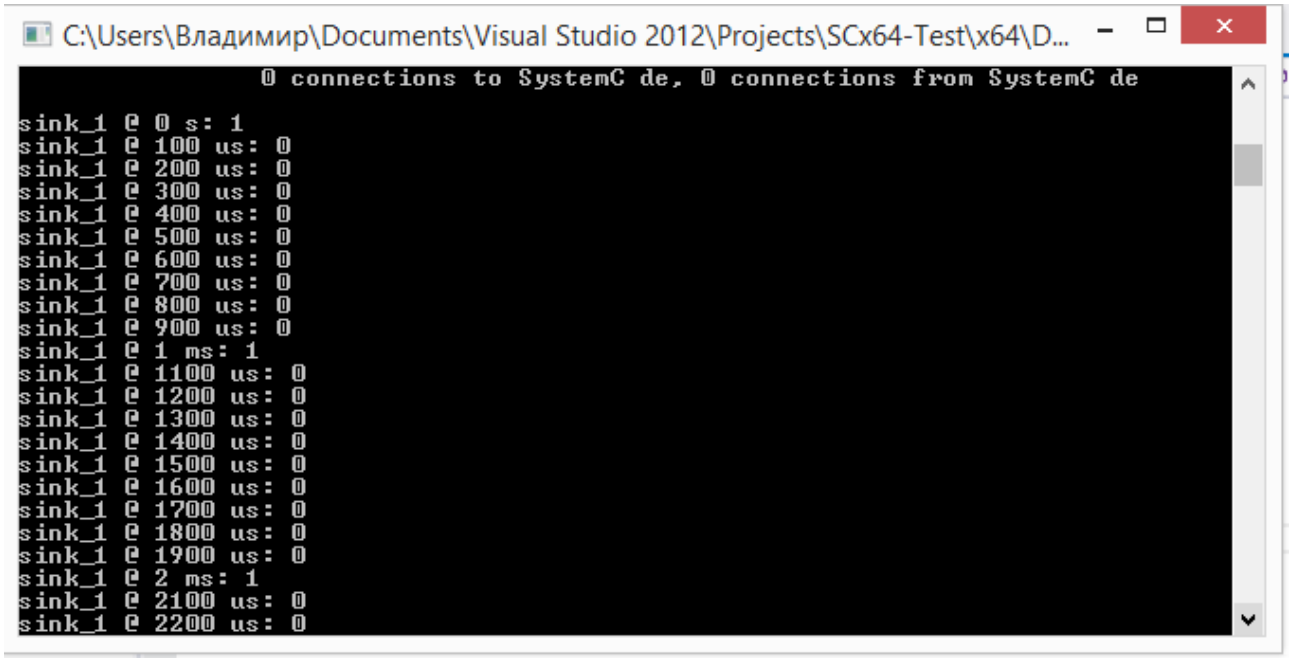


Рис. 2.47. Импульсы с атрибутами: шаг 250 мкс, rate = 4

На рис. 2.48 показаны импульсы с атрибутами шаг 100 мкс, rate = 10.



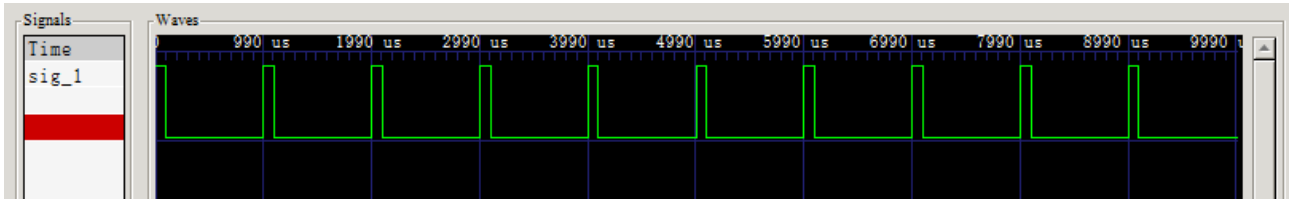


Рис. 2.48. Импульсы с атрибутами шаг 100 мкс, rate = 10

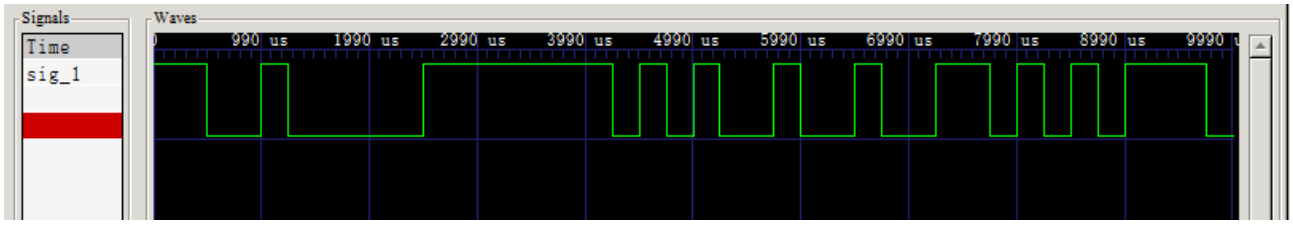
2.9. Формирование случайной последовательности импульсов

В исполняемом файле `sin_source.cpp` (Листинг 2.5) изменим формируемую функцию так:

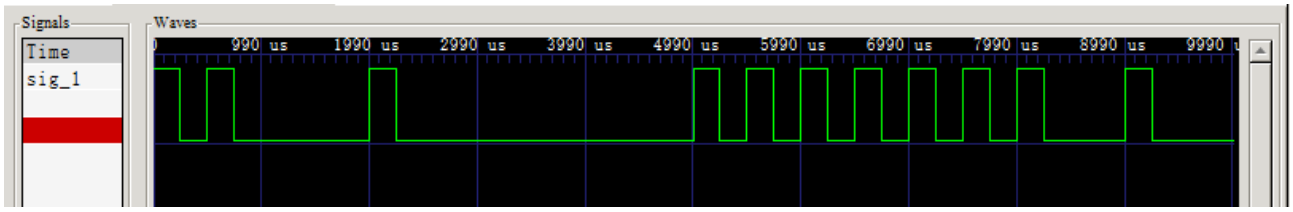
```
    void sin_source::processing()
// Describe time-domain behaviour
    {
      {
    out.write( (bool)(std::rand()%2) );
    }

    }
```

Установим шаг выходного порта 250 мкс. На рис. 2.49 показаны случайные последовательности импульсов для скорости порта 1, 2 и 4.

Rate = 1



Rate = 2



Rate = 4

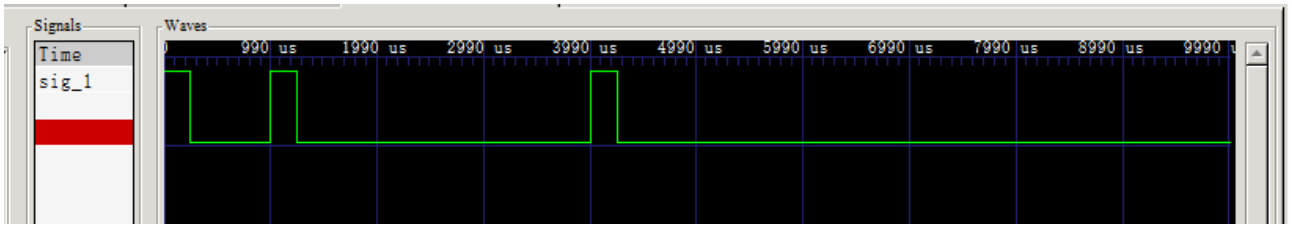


Рис. 2.49. Случайные последовательности импульсов для разных скоростей выходного порта

2.10. Моделирование гармонической функции с шумом

В данном примере формируется шумовой сигнал с нормальным законом распределения (Гауссов шум), который суммируется с гармоническим сигналом.

Листинги программ приведены на 2.9 – 2.14.

Листинг 2.9

Заголовочный файл `sin_source_with_noise.h_`

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
------------------

    #include "sin_source.h"
    #include <cmath>          // for std::sin

    void sin_source::set_attributes()  // Set TDF
attributes
    {
      out.set_timestep(1.0, SC_US);  // Set time step of
output port
    }

    void sin_source::processing()  // Describe time-
domain behaviour
    {
      double t = out.get_time().to_seconds();  // Get
current time of the sample
      double x = ampl * std::sin(2.0 * 3.1415 * freq *
t);  // Calculate sine wave
      out.write(x);       // Write sample to the output
    }
```

Листинг 2.10

Исполняемый файл `sin_source_with_noise.cpp`

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
```

```
    //--------------------------------------------------
------------------

    #include "sin_source_with_noise.h"

    #include <cstdlib> // for std::rand
    #include <cmath>   // for M_PI, std::sin, std::sqrt,
and std::log

    double gauss_rand(double variance)
    {
      double rnd1, rnd2, Q, Q1, Q2;

      do
      {
        rnd1 = static_cast<double>(std::rand()) /
RAND_MAX;
        rnd2 = static_cast<double>(std::rand()) /
RAND_MAX;

        Q1 = 2.0 * rnd1 - 1.0;
        Q2 = 2.0 * rnd2 - 1.0;
        Q = Q1 * Q1 + Q2 * Q2;
      }
      while (Q > 1.0);

      return ( std::sqrt(variance) * ( std::sqrt( - 2.0 *
std::log(Q) / Q) * Q1) );
    }

    void sin_source_with_noise::set_attributes()  // Set
TDF attributes
    {
      out.set_timestep(1.0, SC_US);   // Set time step of
output port
    }

    void sin_source_with_noise::processing()
    {
      double t = out.get_time().to_seconds(); // Get
current time of the sample
      double n = gauss_rand(variance);
      double x = ampl * sin(2.0 * 3.1415 * freq * t) + n;
// Calculate sine wave
```

```
      out.write(x);              // Write sample to the
output
    }
```

Листинг 2.11

Заголовочный файл `sink.h`

```
    #ifndef SINK_H
    #define SINK_H

    #include <systemc-ams.h>   // SystemC AMS header

    SCA_TDF_MODULE(sink)
    {
      sca_tdf::sca_in<double> in;  // TDF input port

      SCA_CTOR(sink)   // Constructor of the TDF module
      : in("in")     // Name the port(s)
      {}

      void processing();   // Describe time-domain
behaviour
    };

    #endif // SINK_H
```

Листинг 2.12

Исполняемый файл `sink.cpp`

```
    #include "sink.h"
    #include <iostream>   // for std::cout and std::endl

    void sink::processing()  // Describe time-domain
behaviour
    {
      std::cout << this->name()  // Display name of
module
                << " @ "
                << this->get_time() // Show module time
                << ": "
                << in.read()  // Read value from input
port
                << std::endl; // and display it
    }
```

Листинг 2.13

Исполняемый файл testbench.cpp

```
#include "sin_source_with_noise.h"
#include "sink.h"

int sc_main(int argn, char* argc[])  // SystemC main
program
{
  sca_tdf::sca_signal<double> sig_1;   // Signal to
connect source w sink

  sin_source_with_noise src_1("src_1");   //
Instantiate source
    src_1.out(sig_1);  // Connect (bind) with signal

  sink sink_1("sink_1");  // Instantiate sink
    sink_1.in(sig_1);    // Connect (bind) with
signal

  sca_trace_file* tfp =      // Open trace file
    sca_create_vcd_trace_file("testbench");
    sca_trace(tfp, sig_1, "sig_1");  // Define which
signal to trace

  sc_start(10.0, SC_MS);   // Start simulation for 10
ms

  sca_close_vcd_trace_file(tfp);  // Close trace file

  return 0;      // Exit with return code 0
}
```

На листинге 2.14 представлена программа формирования шума

Листинг 2.14

Файл gauss.cpp

```
#include <cstdlib> // for std::rand
#include <cmath> // for std::sqrt and std::log
double gauss_rand(double variance)
{
double rnd1, rnd2, Q, Q1, Q2;
do
{
rnd1 = static_cast<double>(std::rand()) / RAND_MAX;
rnd2 = static_cast<double>(std::rand()) / RAND_MAX;
Q1 = 2.0 * rnd1 - 1.0;
```

```
    Q2 = 2.0 * rnd2 - 1.0;
    Q = Q1 * Q1 + Q2 * Q2;
    }
    while (Q > 1.0);
    return std::sqrt(variance) * ( std::sqrt( -2.0 *
std::log(Q) / Q ) * Q1 );
    }
```

На рис. 2.40 показана трассировка гармонического сигнала с шумом.

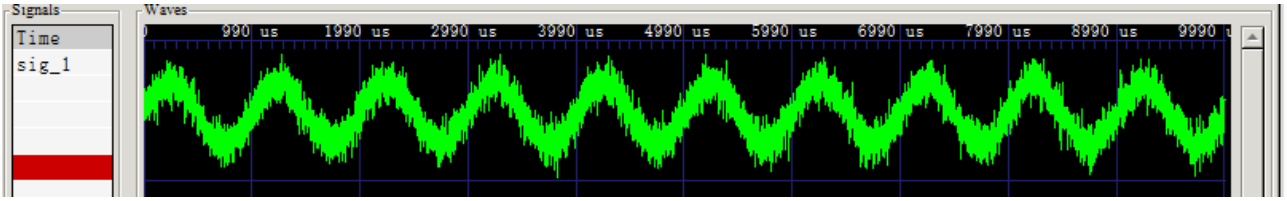


Рис. 2.50. Гармонический сигнал с шумом

2.11. Примеры применения

В этом разделе показаны конкретные примеры применения модели вычислений Timed Data Flow и ее многократные (многоскоростные) возможности. В частности, взаимодействие временных шагов и скорости передачи данных будет играть здесь важную роль. Читателю рекомендуется воспроизвести вычисления, касающиеся скорости передачи данных и временных шагов из примеров в этом разделе, чтобы понять концепции моделирования потока данных по времени.

2.11.1. BASK модулятор

В данном разделе описываются схемы двоичной фазовой манипуляции (BPSK) и двоичной амплитудно-импульсной манипуляции (BASK).

В этом примере рассматривается модуляция с двоичной амплитудой (BASK), где синусоидальный носитель модулируется двоичным сигналом. Модулятор BASK состоит из источника несущего сигнала (sin_src) и микшера (микшер), который в основном умножает двоичный сигнал основной полосы (bit_src) на сегменты сигнала несущей.

На рисунке 2.28 показан структурный состав модулятора BASK. Сигналы на этом рисунке иллюстрируют концепцию двоичной амплитудно-импульсной манипуляции

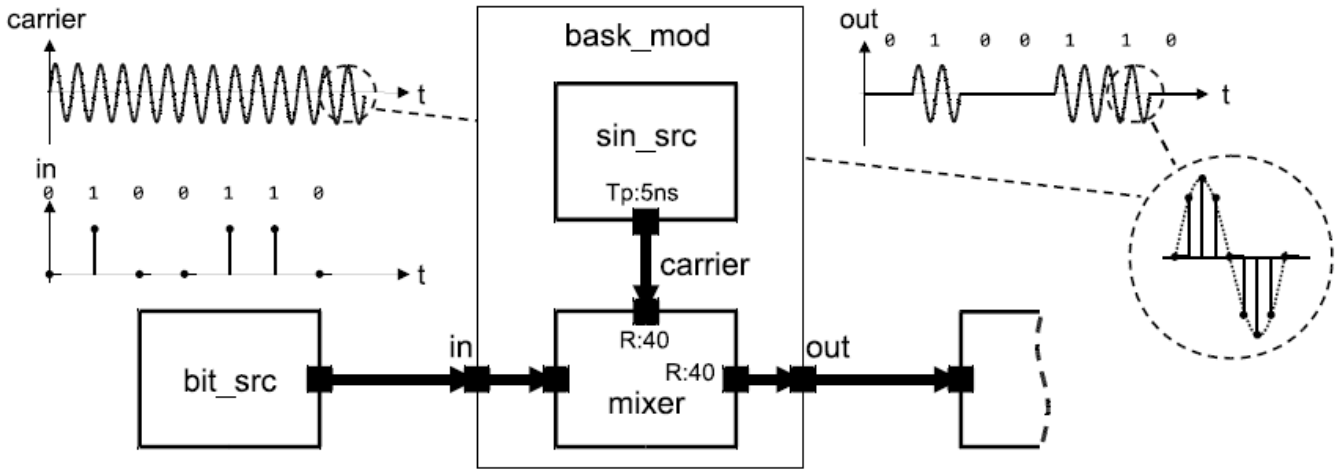


Figure 2.28. BASK modulator

Рисунок 2.28. BASK модулятор

Модуль sin_src уже описан в разделе 2.3.1. Микшер считывает 40 выборок несущей на каждый модулирующий образец. Это может быть реализовано следующим образом:

```
    SCA_TDF_MODULE(mixer)
    {
    sca_tdf::sca_in<bool> in_bin; // input port baseband
signal
    sca_tdf::sca_in<double> in_wav; // input port carrier
signal
    sca_tdf::sca_out<double> out; // output port
modulated signal
    SCA_CTOR(mixer)
    : in_bin("in_bin"), in_wav("in_wav"), out("out"),
rate(40) {} // use a carrier data rate of 40
    void set_attributes()
    {
    in_wav.set_rate(rate);
    out.set_rate(rate);
    }
    void processing()
    {
    for(unsigned long i = 0; i < rate; i++)
    {
    if ( in_bin.read() )
    out.write( in_wav.read(i), i );
    else
    out.write( 0.0, i );
    }
    }
    private:
```

```
unsigned long rate;
};
```

Это, очевидно, более разумно, чем сначала повышать частоту дискретизации двоичного сигнала до скорости передачи данных в 40 раз, так чтобы оба сигнал несущей и сигнал модулирующей частот подавылись на микшер с обоими входными портами, настроенными на скорость передачи данных 1.

Следующий фрагмент кода показывает, как эти два модуля могут быть объединены в модуль модулятора BASK.

Отметим, что в этом случае используется обычный SC_MODULE, в котором создаются два примитивных модуля TDF.

```
SC_MODULE(bask_mod)
{
sca_tdf::sca_in<bool> in;
sca_tdf::sca_out<double> out;
sin_src sine;
mixer mix;
SC_CTOR(bask_mod)
: in("in"), out("out"),
sine("sine", 1.0, 1.0e7, sca_core::sca_time( 5.0,
sc_core::SC_NS ) ),
mix("mix")
{
sine.out(carrier);
mix.in_wav(carrier);
mix.in_bin(in);
mix.out(out);
}
private:
sca_tdf::sca_signal<double> carrier;
};
```

Обратите внимание, что несущая частота 10 МГц устанавливается путем передачи параметра в модуль sin_src, тогда как модулирующая частота определяется косвенно скоростью передачи данных модуля микшера, и шаг по времени устанавливается на выводе модуля sin_src. Порт in_wav микшера модуля имеет тот же временной шаг, что и выход модуля sin_src (а именно 5 нс), но скорость передачи данных 40. Следовательно, порт in_bin модуля микшера, который имеет скорость передачи данных 1, получает шаг по времени 200 нс. Это приводит к частоте основной полосы частот 5 МГц, для которой именно такая ситуация изображена на рисунке 2.28.

Для полноты, код двоичного источника основной полосы, который производит случайный двоичный сигнал приведен ниже.

```
SCA_TDF_MODULE(bit_src)
```

```
{
sca_tdf::sca_out<bool> out; // output port
SCA_CTOR(bit_src) : out("out") {}
void processing()
{
out.write( (bool)(std::rand()%2) );
}
};
```

2.11.2. БАСК демодулятор

Демодуляция модулированного сигнала BASK выполняется сначала с использованием выпрямителя (который формирует абсолютное значение значение сигнала), за которым следует фильтр нижних частот, который можно реализовать, как описано в разделе 2.3.2. с модулем ltf_nd_filter. Выпрямитель может быть реализован следующим образом:

```
SCA_TDF_MODULE(rectifier)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
SCA_CTOR(rectifier) : in("in"), out("out") {}
void processing()
{
out.write( std::abs(in.read()) );
}
};
```

Выходной сигнал фильтра нижних частот представляет собой сигнал типа double, который содержит 40 выборок за 200 нс, и необходимо получить одну выборку за 200 нс (см. рисунок 2.29).

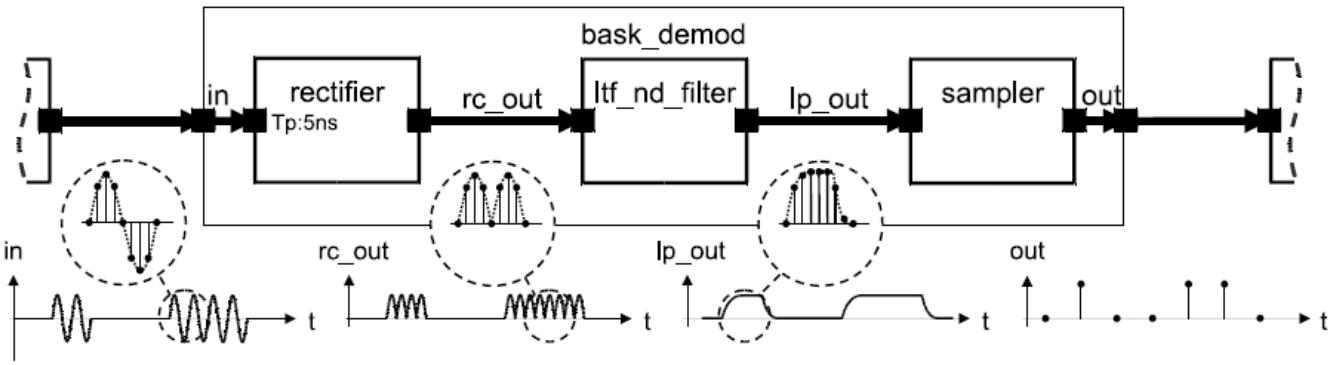


Figure 2.29. BASK demodulator

Рисунок 2.29. БАСК демодулятор

Следующий листинг показывает реализацию сэмплера. Он имеет скорость ввода данных 40. Поэтому он читает точное количество выборок,

которые связаны с одним конкретным битом в сигнале модулирующей частоты. Он использует только одну выборку в фиксированной позиции выборки без второй (нижней) половины потока выборки для каждого выполняемого модуля. Идея этого заключается в том, что выходной сигнал фильтра нижних частот можно рассчитывать по времени. Этот образец сравнивается с пороговым значением: если оно больше, выход сэмплера имеет значение true, и ложь в противном случае. Это эффективно моделирует 1-битный аналого-цифровой преобразователь, который дискретизирует входные данные каждые 200 нс.

```
    SCA_TDF_MODULE(sampler)
    {
    sca_tdf::sca_in<double> in; // input port
    sca_tdf::sca_out<bool> out; // output port
    SCA_CTOR(sampler) : in("in"), out("out"), rate(40),
threshold(0.2) {}
    void set_attributes()
    {
    in.set_rate(rate);
    sample_pos = (unsigned long)std::ceil( 2.0 *
(double)rate/3.0 );
    }
    void processing()
    {
    if( in.read(sample_pos) > threshold )
    out.write(true);
    else
    out.write(false);
    }
    private:
    unsigned long rate;
    double threshold;
    unsigned long sample_pos;
    };
```

Обратите внимание, что приведенный выше код несет определенную причинный недостаток, который может возникнуть, если скорость входного порта больше 1: значение выходной выборки вычисляется на основе входной выборки, которая имеет отметку времени больше, чем выходное значение. Следовательно, что касается времени моделирования TDF-модели вычисления, следствие предшествует причине. Эта нерегулярность может быть легко устранена путем введения задержки, например, с помощью set_delay (1) на выходной порт. Однако это не является действительно необходимым, так как серьезные проблемы (то есть парадоксы) могут произойти, только если полученное выходное значение будет подано в контур обратной связи. Но в этом случае задержка есть в любом случае (см. раздел 2.1.2), что решает

проблему автоматически.

Следующий листинг показывает, как эти три модуля объединяются для всего модуля демодулятора BASK.

Обратите внимание, что шаг времени здесь явно не установлен, так как мы ожидаем, что он будет установлен в той части модели, которая обеспечивает модулированный сигнал.

```
SC_MODULE(bask_demod)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<bool> out;
rectifier rc;
ltf_nd_filter lp;
sampler sp;
SC_CTOR(bask_demod)
: in("in"), out("out"), rc("rc"), lp("lp", 3.3e6),
sp("sp"), rc_out("rc_out"), lp_out("lp_out")
{
rc.in(in);
rc.out(rc_out);
lp.in(rc_out);
lp.out(lp_out);
sp.in(lp_out);
sp.out(out);
}
private:
sca_tdf::sca_signal<double> rc_out, lp_out;
};
```

2.11.3. TDF-симуляция примера BASK

Реализация полного приложения BASK выполняется в программе sc_main. В рамках программы создаются тело, модуль источника битов bit_src, модуль модулятора BASK bask_mod и модуль демодулятора BASK bask_demod. Эти модули TDF связаны между собой с использованием сигналов TDF.

```
int sc_main(int argc, char* argv[])
{
sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);
sca_tdf::sca_signal<bool> in_bits, out_bits;
sca_tdf::sca_signal<double> wave;
bit_src bs("bs"); // random bit source
bs.out(in_bits);
bask_mod mod("mod"); // modulator
mod.in(in_bits);
mod.out(wave);
```

```
    bask_demod demod("demod"); // demodulator
    demod.in(wave);
    demod.out(out_bits);
    sca_util::sca_trace_file* atf =
sca_util::sca_create_vcd_trace_file( "tr.vcd" );
    sca_util::sca_trace( atf, in_bits, "in_bits" );
    sca_util::sca_trace( atf, wave, "wave" );
    sca_util::sca_trace( atf, out_bits, "out_bits" );
    sc_core::sc_start(1, sc_core::SC_US);
    sca_util::sca_close_vcd_trace_file( atf );
    return 0;
    }
```

Более подробную информацию о возможностях управления и отслеживания симуляции можно найти в главе 6.

2.11.4. Взаимодействие примера BASK с SystemC

Как показано на рисунке 2.28, компоненты, созданные в примере BASK, являются TDF модулями, которые принадлежат к тому же кластеру TDF. В частности, случайный двоичный сигнал на входе данных смесителя имеет вид, который генерируется чистым модулем TDF bit_src.

На практике этот двоичный сигнал с большей вероятностью будет генерироваться цифровым компонентом, который следует правилам области дискретных событий, в результате чего получается настоящая гетерогенная система, состоящая из двух цифровых частей (генератор случайных данных и прием данных) и одна часть AMS TDF (модулятор и демодулятор BASK).

На рисунке 2.30 показаны основные изменения, вызванные этой конструкцией: ввод данных модулятора BASK (соответственно вывод данных демодулятора BASK) теперь должен быть портом SystemC sc_core :: sc_in <T> (соотв. порт sc_core :: sc_out <T>), содержащий значения bool. Таким образом, с точки зрения TDF требуется порт преобразователя для чтение из канала (или, соответственно, для записи в канал), соответствующего порту домена дискретных событий.

Такие порты обозначены символом  на этом рисунке.

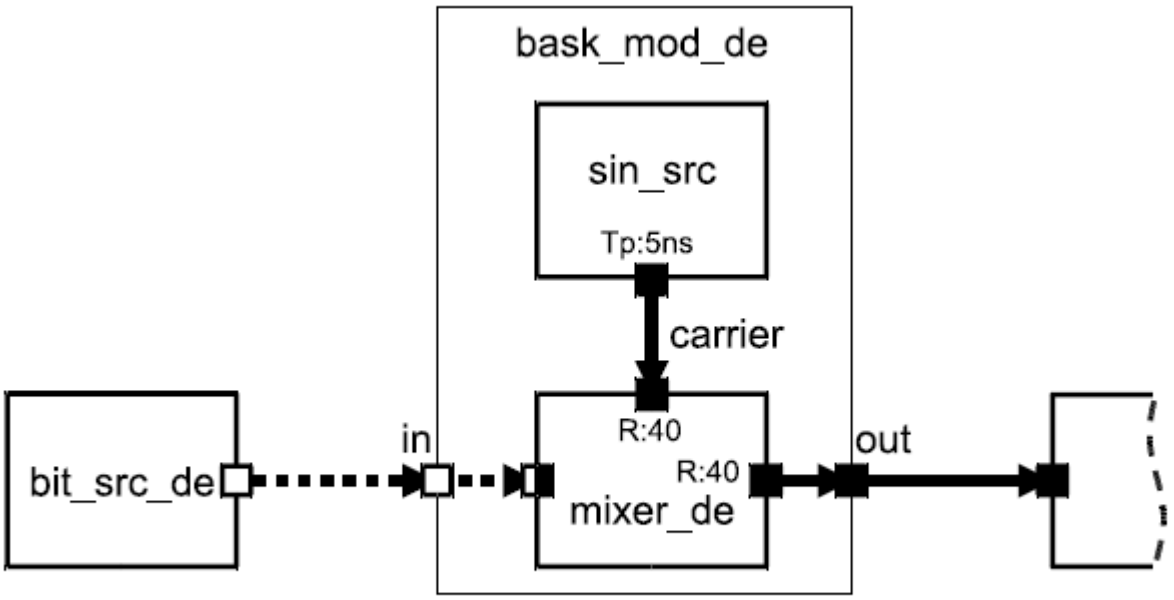


Figure 2.30. BASK modulator, mixing discrete-event and TDF domain

Рисунок 2.30. Модулятор BASK, смешивающий область дискретных событий и TDF

Следующий код - чистый SystemC. Благодаря бесконечному циклу в конструкции SystemC SC_THREAD, эта новая версия источника битов, теперь называемая bit_src_de, генерирует новое случайное значение bool на своем выходном порту каждые 200 нс.

```
SC_MODULE(bit_src_de)
{
sc_core::sc_out<bool> out;
SC_CTOR(bit_src_de): out("out")
{
SC_THREAD(bit_gen_thread);
}
void bit_gen_thread()
{
while(true)
{
bool var = (bool)(std::rand()%2);
out.write(var);
sc_core::wait( 200, sc_core::SC_NS );
}
}
};
```

Модуль микшера TDF теперь имеет цифровой вход in_bin, подключенный к выходу bit_src_de моделя SystemC. Исходный код микшера не сильно

отличается от предыдущей основной модификация, будучи введением порта преобразователя дискретных событий:

```
    SCA_TDF_MODULE(mixer_de)
    {
    sca_tdf::sca_de::sca_in<bool> in_bin; // TDF
converter input port from discrete-event domain
    sca_tdf::sca_in<double> in_wav;
    sca_tdf::sca_out<double> out;
    SCA_CTOR(mixer_de)
    : in_bin("in_bin"), in_wav("in_wav"), out("out"),
rate(40) {}
    void set_attributes()
    {
    in_wav.set_rate(rate);
    out.set_rate(rate);
    }
    void processing()
    {
    for(unsigned long i = 0; i < rate; i++)
    {
    if(in_bin.read())
    out.write( in_wav.read(i), i );
    else
    out.write( 0.0, i );
    }
    }
    private:
    unsigned long rate;
    };
```

Соответственно, исходный код модулятора BASK, показанный ниже, подробно описывает небольшое изменение: ввод данных теперь является портом ввода дискретного события:

```
    SC_MODULE(bask_mod_de)
    {
    sc_core::sc_in<bool> in; // data input is now digital
    sca_tdf::sca_out<double> out;
    sin_src sine;
    mixer_de mix; // use mixer with discrete-event input
    SC_CTOR(bask_mod_de)
    : in("in"), out("out"),
    sine("sine", 1.0, 1.0e7, sca_core::sca_time( 5.0,
sc_core::SC_NS ) ),
    mix("mix"), carrier("carrier")
    {
    sine.out(carrier);
```

```
    mix.in_wav(carrier);
    mix.in_bin(in);
    mix.out(out);
    }
    private:
    sca_tdf::sca_signal<double> carrier;
    };
```

Для полноты исходный код сэмплера BASK в демодуляторе приведен ниже. Выходные данные out теперь является выходным портом преобразователя. Соответствующий порт в демодуляторе, который создает сэмплер объявлен как традиционный выходной порт SystemC.

```
    SCA_TDF_MODULE(sampler_de)
    {
    sca_tdf::sca_in<double> in; // input port
    sca_tdf::sca_de::sca_out<bool> out; // TDF converter
output port to discrete-event domain
    SCA_CTOR(sampler_de) : in("in"), out("out"),
rate(40), threshold(0.2) {}
    void set_attributes()
    {
    in.set_rate(rate);
    sample_pos = (unsigned long)std::ceil( 2.0 *
(double)rate/3.0 );
    }
    void processing()
    {
    if( in.read(sample_pos) > threshold )
    out.write(true);
    else
    out.write(false);
    }
    private:
    unsigned long rate;
    double threshold;
    unsigned long sample_pos;
    };
```

Глава 3. Моделирование линейного потока сигналов

3.1. Основы моделирования

Модель вычисления Linear Signal Flow (LSF) позволяет моделировать поведение AMS, определяемое как отношения между переменными набора линейных алгебраических уравнений. LSF - это стиль моделирования с непрерывным временем, использующий направленные действительные сигналы, приводящие к неконсервативному описанию системы. Нет зависимости между потоком и потенциальными величинами. Вместо этого для представления каждого сигнала используется только одно действительное значение.

Модели потока сигналов могут быть описаны в обозначениях блок-схем. Элементарные части или функции представлены блоками. Сигналы используются для соединения этих блоков. В результате отношения между блоки определяют эквивалентные математические уравнения. На рисунке 3.1 показан пример диаграммы такого блока потока сигналов, составленной из четырех модулей LSF, которые связаны между собой с помощью сигналов LSF. Обратите внимание, что дополнение «Оператор», хотя и имеет другое графическое представление, также является модулем LSF. Модель LSF состоит из набора связанных модулей LSF, которые вместе образуют систему уравнений LSF или LSF кластер. Получающаяся модель LSF имеет входные и выходные порты LSF для соединения с другими модулями.

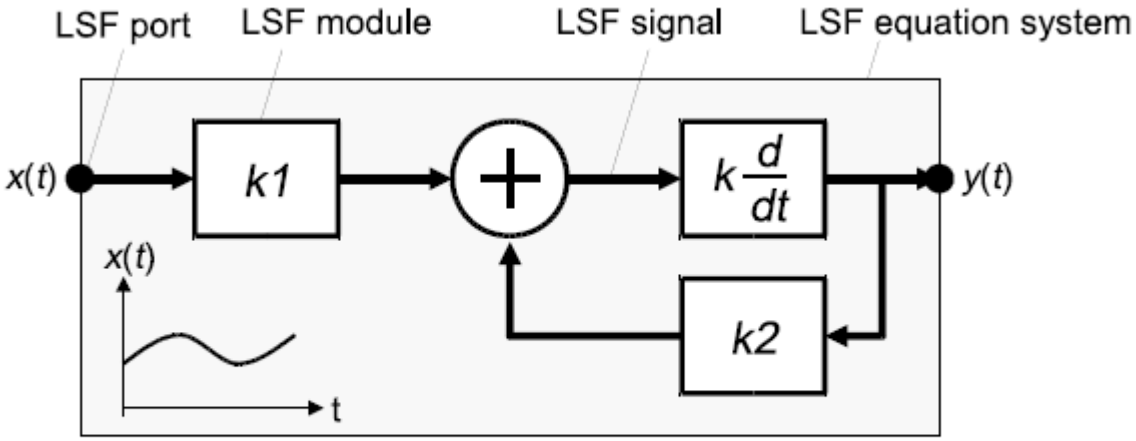


Figure 3.1. Example of a basic LSF model composed of 4 LSF modules

Рисунок 3.1. Пример базовой модели LSF, состоящей из 4 модулей LSF

3.1.1. Настройка системы уравнений LSF

Расширения SystemC AMS предлагают конечный набор предопределенных примитивных модулей LSF, реализующих такие функции, как сложение, умножение, интеграция и т. д. В отличие от стиля моделирования

TDF, модели LSF могут быть составленным только из этих примитивов. Расширения AMS не предоставляют возможности для реализации пользовательских примитивов LSF. Вместо этого математические уравнения, описывающие предполагаемую функциональность, должны быть созданы путем составления предопределенного набора примитивных модулей LSF. Рисунок 3.2 показывает несколько основных примеров примитивов LSF и соответствующих им математических уравнений.

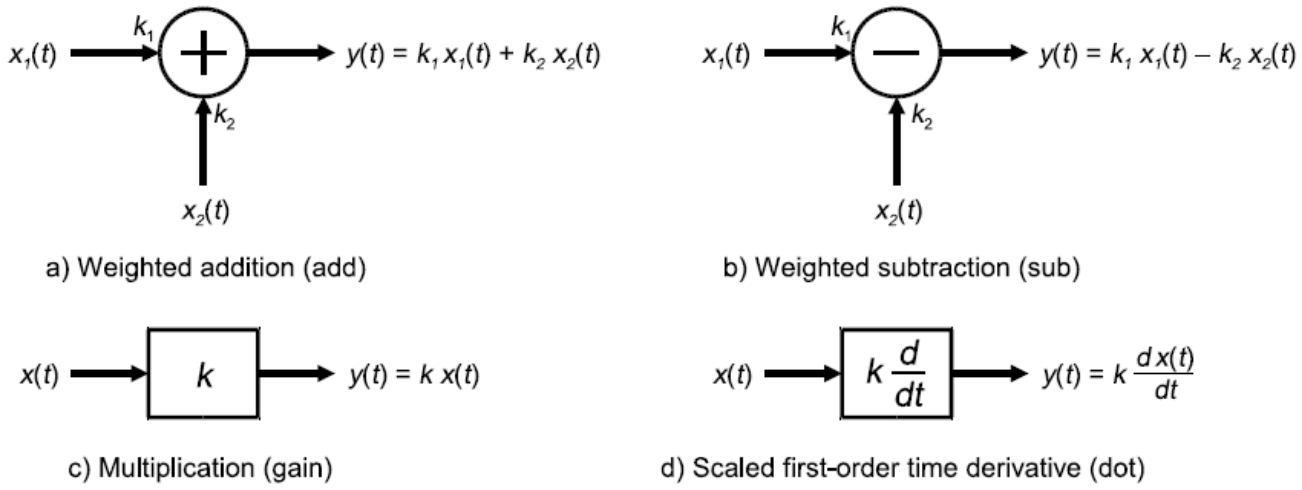


Figure 3.2. Examples of some basic LSF primitives and their corresponding mathematical equations

Рисунок 3.2. Примеры некоторых основных примитивов LSF и соответствующих им математических уравнений

При создании модели LSF (блок-схемы), математические уравнения для каждого блока и их взаимосвязь будет использоваться для составления общей системы уравнений. Например, для модели LSF, представленной на рисунке 3.1, будет приведена следующая система уравнений, основанная на представленных уравнениях каждого примитив, как показано на рисунке 3.2:

$$y(t) = k_1 \cdot \frac{dx(t)}{dt} + k_2 \cdot \frac{dy(t)}{dt}$$

$$y(t) = \mathrm{k}\left(k_1 \cdot \frac{dx(t)}{dt} + k_2 \cdot \frac{dy(t)}{dt}\right)$$

Обратите внимание, что масштабные коэффициенты сложения и блока производной по времени первого порядка установлены в 1. Вместо этого дополнительные блоки умножения k1 и k2 используются для этого примера.

3.1.2. Назначение и распространение временных шагов

Как и для модуля TDF, шаг времени может быть назначен непосредственно модулю LSF или может быть назначен автоматически используя механизм распространения временного шага в кластере LSF. В случае, если LSF модель связана с моделью TDF, шаг по времени от подключенного порта (портов) TDF распространяется на LSF модель. Согласованность между локально заданным временным шагом модуля LSF и распространяющимся временным шагом имеет важное значение.

В противном случае моменты времени для решения системы уравнений LSF или связи с подключенным TDF моделью не может быть определена должным образом (см. Также раздел 2.1.3). Шаг по времени должен быть определен как минимум в одном месте во всей системе.

Во время моделирования система уравнений LSF решается численно с соответствующими временными шагами, которые могут быть меньше назначенного временного шага. Решатель по крайней мере предоставит результаты в рассчитанные моменты времени от назначенных временных шагов.

3.2. Языковые конструкции

3.2.1. LSF модули

Модуль Linear Signal Flow - это предопределенный примитивный модуль для представления конкретной функции или математическое соотношение, которое станет частью общей системы уравнений. Доступные предопределенные LSF примитивные модули перечислены в Таблице 3.1 ниже. Приложение A дает подробную информацию для каждого модуля LSF.

LSF module name Description
sca_lsf::sca_add Weighted addition of two LSF signals. Взвешенное сложение двух сигналов LSF.
sca_lsf::sca_sub Weighted subtraction of two LSF signals. Взвешенное вычитание двух сигналов LSF.
sca_lsf::sca_gain Multiplication of an LSF signal by a constant gain. Умножение LSF-сигнала на постоянное усиление.
sca_lsf::sca_dot Scaled first-order time derivative of an LSF signal. Масштабированная производная по времени первого порядка сигнала LSF.
sca_lsf::sca_integ Scaled time-domain integration of an LSF signal. Масштабированное интегрирование во временной области сигнала LSF.
sca_lsf::sca_delay Scaled time-delayed version of an LSF signal. Масштабированная версия LSF с задержкой по времени.
sca_lsf::sca_source LSF source. Источник LSF.
sca_lsf::sca_ltf_nd Scaled Laplace transfer function in the time-domain in the
numerator-denominator form. Масштабная передаточная функция Лапласа во временной области в форме числитель-знаменатель.
sca_lsf::sca_ltf_zp Scaled Laplace transfer function in the time-domain in the zero-

pole
form. Масштабированная передаточная функция Лапласа во временной области в форме нуль - полюс.
sca_lsf::sca_ss Single-input single-output state-space equation. Уравнение пространства состояний с одним входом и одним выходом
sca_lsf::sca_tdf::sca_gain,
sca_lsf::sca_tdf_gain
Scaled multiplication of a TDF input signal with an LSF input signal. Масштабное умножение входного сигнала TDF на входной сигнал LSF.
sca_lsf::sca_tdf::sca_source,
sca_lsf::sca_tdf_source
Scaled conversion of a TDF input signal to an LSF output signal. Масштабное преобразование входного сигнала TDF в выходной сигнал LSF.
sca_lsf::sca_tdf::sca_sink,
sca_lsf::sca_tdf_sink
Scaled conversion from an LSF input signal to a TDF output signal. Масштабное преобразование из входного сигнала LSF в выходной сигнал TDF.
sca_lsf::sca_tdf::sca_mux,
sca_lsf::sca_tdf_mux
Selection of one of two LSF input signals by a TDF control signal
(multiplexer). Выбор одного из двух входных сигналов LSF управляющим сигналом TDF (Мультиплексор).
sca_lsf::sca_tdf::sca_demux,
sca_lsf::sca_tdf_demux
Routing of an LSF input signal to either one of two LSF output
signals controlled by a TDF signal (demultiplexer). Маршрутизация входного сигнала LSF на один из двух выходов LSF сигнала, управляемая сигналом TDF (демультиплексор).
sca_lsf::sca_de::sca_gain,
sca_lsf::sca_de_gain
Scaled multiplication of a discrete-event input signal by an LSF input
signal. Масштабное умножение входного сигнала дискретного события на входной LSF сигнал.
sca_lsf::sca_de::sca_source,
sca_lsf::sca_de_source
Scaled conversion of a discrete-event input signal to an LSF output
signal. Масштабное преобразование входного сигнала дискретного события в выходной LSF сигнал.
sca_lsf::sca_de::sca_sink,
sca_lsf::sca_de_sink
Scaled conversion from an LSF input signal to a discrete-event
output signal. Масштабное преобразование из входного сигнала LSF в дискретное событие выходного сигнала.
sca_lsf::sca_de::sca_mux,

sca_lsf::sca_de_mux
Selection of one of two LSF input signals by a discrete-event control
signal (multiplexer). Выбор одного из двух входных сигналов LSF с помощью управления дискретным событием сигнал (мультиплексор).
sca_lsf::sca_de::sca_demux,
sca_lsf::sca_de_demux
Routing of an LSF input signal to either one of two LSF output signals controlled by a discrete-event signal (demultiplexer). Маршрутизация входного сигнала LSF на один из двух выходных сигналов LSF, управляемая сигналом дискретного события (демультиплексор)

Таблица 3.1. LSF примитивные модули

Шаг по времени модуля

Чтобы решить систему уравнений LSF, шаг времени должен быть связан с набором подключенных модулей LSF как часть этапа разработки. Это можно сделать с помощью функции-члена модуля LSF set_timestep. В качестве альтернативы модель LSF может опираться на механизм распространения шага по времени, который передает шаг по времени от модуля к модулю через его порты в моделях вычислений TDF, LSF и ELN.

Таким образом, в случаях, когда модель LSF подключена к модели TDF, временной шаг от подключенного порта, если это доступно, распространяется на модель LSF. В случае, если распространяются временные шаги и пользовательские временные шаги, то при этом соответствие между этими временными шагами является обязательным, как описано в разделе 2.1.3.

Шаг по времени модуля может быть назначен путем вызова функции-члена set_timestep экземпляра объекта в конструкторе родительского модуля и передачи значение *double* и единицы времени или объекта типа sca_core :: sca_time, как показано в следующем примере:

```
SC_MODULE(my_lsf_source)
{
// port declaration
sca_lsf::sca_out y;
// child module declaration
sca_lsf::sca_source src;
SC_CTOR(my_lsf_source)
: y("y"),
src("src", 0.0, 0.0, 1.0e-3, 1.0e3) // 1 kHz
sinusoidal source with an amplitude of 1e-3
{
src.set_timestep(0.5, sc_core::SC_MS); // set module
timestep of source to 0.5 ms
```

```
src.y(y);
}
};
```

3.2.2. LSF порты

Порт LSF - это объект, который можно использовать для соединения нескольких моделей LSF с использованием сигналов LSF, которые привязаны к этому порту. Из-за природы формализма моделирования LSF порт может быть порт или входным или выходной порт, но не *inout*. Порты LSF используются для подключения модулей LSF с использованием сигналов класса sca_lsf :: sca_signal. Поскольку порты LSF всегда являются иерархическими портами внутри родительского модуля, их можно использовать для подключения к дочерним модулям LSF напрямую, следуя правилу привязки порт-порт (см. раздел 3.3.1).

Порты LSF имеют предопределенный тип данных, также называемый природой потока сигналов, что предотвращает использование пользовательских типы данных.

В настоящее время существует два класса портов LSF:
• LSF входные порты класса sca_lsf :: sca_in.
• LSF выходные порты класса sca_lsf :: sca_out.

В приведенном ниже примере показано, как порты LSF используются в структурной модели LSF.

```
SC_MODULE(my_lsf_model)
{
// port declarations
sca_lsf::sca_in x;
sca_lsf::sca_out y;
SC_CTOR(my_lsf_model) : x("x"), y("y")
{
// model implementation here
}
};
```

1. Входной порт LSF, который несет сигнал непрерывного времени и непрерывного значения x (t).

2. Выходной порт LSF, который передает сигнал непрерывного времени и непрерывного значения y (t).

3. Использование списка инициализации конструктора для присвоения имен «x» и «y» входным и выходным портам, соответственно.

Для LSF нет доступных портов конвертера. Вместо этого предоставляются специализированные модули преобразователя для подключиться к TDF или домену дискретных событий. Это объясняется в разделе 3.4. В отличие от портов TDF, LSF порты не предоставляют функции-

члены для прямого чтения или записи из канала.

3.2.3. LSF сигналы

Сигналы LSF используются для соединения примитивных модулей LSF. Сигналы LSF несут непрерывное время и непрерывное значение сигнала, в то время как порты LSF определяют направление сигналов от одного модуля LSF другому. Как и для портов LSF, сигналы LSF используют внутреннюю структуру данных для хранения непрерывного времени и непрерывный сигнал. Поэтому сигналы LSF не определены как класс шаблона и должны использоваться в соответствии с примером ниже:

```
// signal declaration
sca_lsf::sca_signal sig; // LSF signal
```

Как и в SystemC, список инициализации конструктора родительского модуля может использоваться для назначения определенного пользователем имени сигналу:

```
// assign the names of LSF signal instance in the
constructor initialization-list
SC_CTOR(my_module) : sig("sig") {}
```

Раздел 3.3 опишет создание структурных моделей LSF и покажет примеры назначения пользовательских имен для портов и сигналов.

3.3. Моделирование непрерывного поведения

Модели LSF могут использоваться для реализации линейного динамического поведения с непрерывным временем. Модели LSF могут быть составлены только с использованием примитивных модулей LSF. Поэтому модель LSF всегда является структурной моделью.

3.3.1. Структурный состав модулей LSF

Модули LSF должны создаваться как дочерние модули внутри созданного обычного родительского модуля SystemC с помощью макроса SC_MODULE или путем публичного получения из sc_core :: sc_module. Этот родительский модуль также создает все необходимые порты для связи с внешним миром и внутренних сигналов для взаимосвязи дочерних модулей. Параметризация созданных экземпляров модулей, а также соединение модулей должно быть сделано в конструкторе (например, создано с помощью макроса SC_CTOR) родительского модуля SystemC.

Привязка порта

Для правильного подключения модулей LSF к другим модулям и сигналам LSF необходимо соблюдать возможны привязки, показанные на рисунке 3.3. Правила привязки порта совместимы и дополняют правилам

SystemC и TDF (см. также раздел 2.3.3).

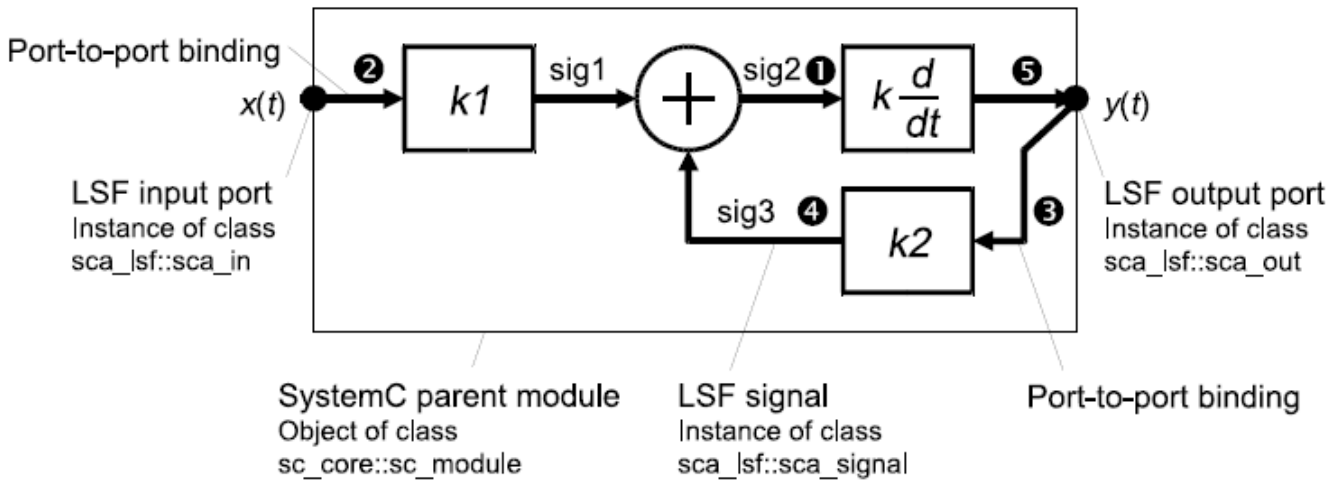


Figure 3.3. Port binding rules for LSF input and output ports

Рисунок 3.3. Правила привязки портов для портов ввода и вывода LSF

1. Связывание входного порта LSF с сигналом LSF.
2. Привязка входного порта LSF к входному порту LSF родительского модуля (привязка порт-порт).
3. Привязка входного порта LSF к выходному порту LSF родительского модуля (привязка порт-порт).
4. Связывание выходного порта LSF с сигналом LSF.
5. Привязка выходного порта LSF к выходному порту LSF родительского модуля (привязка порт-порт).

Кроме того, каждый сигнал LSF должен быть привязан точно к одному выходному порту LSF примитивного модуля LSF, и может быть привязан к любому количеству входных портов LSF примитивных модулей LSF на всем протяжении иерархии.

Для примитивных модулей LSF, порты которых подключены к TDF или сигналам или портам дискретных событий, следует следуйте правилам привязки портов соответствующих моделей вычислений.

Пример ниже показывает реализацию структурного состава на рисунке 3.3.

```
SC_MODULE(my_structural_lsf_model)
{
sca_lsf::sca_in x;
sca_lsf::sca_out y;
sca_lsf::sca_gain gain1, gain2;
```

```
    sca_lsf::sca_dot dot1;
    sca_lsf::sca_add add1;
    my_structural_lsf_model( sc_core::sc_module_name,
double k1, double k2 )
    : x("x"), y("y"), gain1("gain1", k1), gain2("gain2",
k2), dot1("dot1"), add1("add1"),
    sig1("sig1"), sig2("sig2"), sig3("sig3")
    {
    gain1.x(x);
    gain1.y(sig1);
    gain1.set_timestep(1,sc_core::SC_MS);
    add1.x1(sig1);
    add1.x2(sig3);
    add1.y(sig2);
    dot1.x(sig2);
    dot1.y(y);
    gain2.x(y);
    gain2.y(sig3);
    }
    private:
    sca_lsf::sca_signal sig1, sig2, sig3;
    };
```

Входные и выходные порты LSF, объявленные внутри этого модуля класса sc_core :: sc_module, становятся частью структурного состава.

Примитивные модули LSF объявлены в родительском модуле как дочерние модули.

Список инициализации в конструкторе родительского модуля распространяет необходимую конфигурацию параметров для портов LSF, сигналов LSF и дочерних модулей.

Привязка порта выполняется внутри конструктора родительского модуля.

Шаг по времени для примитивных модулей LSF выполняется внутри конструктора родительского модуля. LSF модуль также может получить свой временной шаг посредством распространения временного шага своих подключенных модулей.

Внутренние сигналы LSF используются для подключения портов LSF и дочерних модулей. Эти сигналы должны быть объявлены приватным (private), так как они не должны быть доступны извне модуля.

3.3.2. Непрерывное моделирование

В приведенном ниже примере показан фильтр нижних частот первого порядка, основанный на той же передаточной функции Лапласа, что и описано в разделе 2.3.2:

$$H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c} s}$$

где H_0 - коэффициент усиления по постоянному току, а fc - частота среза фильтра в Гц. Передаточная функция Лапласа может быть переписан для реализации LSF в виде:

$$y(t) = H_0 x(t) - \frac{1}{2\pi f_c} \frac{dy(t)}{dt}$$

Соответствующие обозначения блок-схем и код реализации приведены ниже, где масштабированные коэффициенты примитивных модулей LSF используются для реализации усиления постоянного тока H_0 и отсечки фильтра на частота fc:

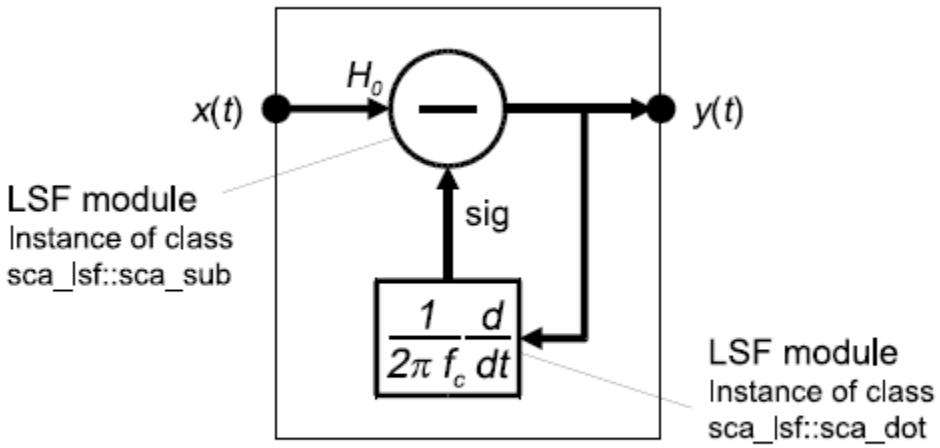


Figure 3.4. Example of an LSF model implementing a first-order low-pass filter

Рисунок 3.4. Пример модели LSF, реализующей фильтр нижних частот первого порядка

```
    SC_MODULE(my_lsf_filter)
    {
    sca_lsf::sca_in x;
    sca_lsf::sca_out y;
    sca_lsf::sca_sub sub1;
    sca_lsf::sca_dot dot1;
    my_lsf_filter( sc_core::sc_module_name, double h0 =
1.0, double fc = 1.0e3 )
```

```
    : x("x"), y("y"), sub1("sub1", h0), dot1("dot1",
1.0/(2.0*M_PI*fc) ), sig("sig")
    {
    sub1.x1(x);
    sub1.x2(sig);
    sub1.y(y);
    dot1.x(y);
    dot1.y(sig);
    }
    private:
    sca_lsf::sca_signal sig;
    };
```

Коэффициент усиления H_0 для входного сигнала передается через конструктор в экземпляр sub1 и частота fc передается через конструктор экземпляру dot1.

3.4. Взаимодействие между LSF и моделями дискретных событий или TDF

Модель вычисления LSF установит и решит систему уравнений для симуляции поведения модели непрерывного времени, основанную на базовом наборе примитивных модулей LSF, описанных в разделе 3.2.1. Любое «внешнее» входное значение, например, из сигнала дискретного события или выборки TDF, должно быть введено в систему уравнений через один из этих примитивных модулей LSF. Поэтому доступны специализированные примитивные модули LSF с портами для моделей вычислений в области дискретных событий и TDF, которые называются модулями преобразования.

Основное назначение этих модулей - создать интерфейс для преобразования и передачи данных из одной модели вычисление к другой.

3.4.1. Чтение и запись в модели дискретных событий

Для соединения моделей LSF с моделями с дискретными событиями должны использоваться модули преобразователей LSF с внутренним портом класса sc_core :: sc_in или sc_core :: sc_out.

На рисунке 3.5 показан примитивный модуль LSF sca_lsf :: sca_de :: sca_source, считывающий сигнал дискретного события и записывающий в сигнал LSF. В этом примере временному шагу модуля 1 мс назначен LSF модуль преобразователя. Модель LSF непрерывно считывает значения с входа в моменты времени, которые рассчитывается по назначенным временным шагам. Входное значение считается постоянным, пока не будет прочитано следующее значение.

Входные значения интерпретируются для формирования непрерывного сигнала, который становится доступным на выходе модуля преобразователя (считывают входные выборки, показанные в виде точечного сигнала).

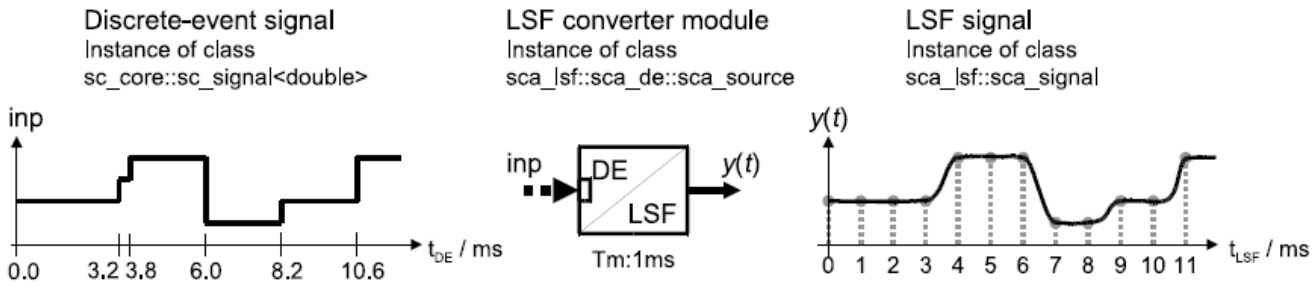


Figure 3.5. LSF converter module reading from a discrete-event input signal and writing to an LSF output signal

Рисунок 3.5. Модуль преобразования LSF, считывающий из входного сигнала дискретного события и записывающий в выходной сигнал LSF

На рисунке 3.6 показан примитивный модуль LSF sca_lsf :: sca_de :: sca_sink, который считывает сигнал LSF и записывает эквивалентное значение в сигнал дискретного события. Значения на выходном порте записываются во временных точках, которые рассчитываются из заданного временного шага модуля 1 мс.

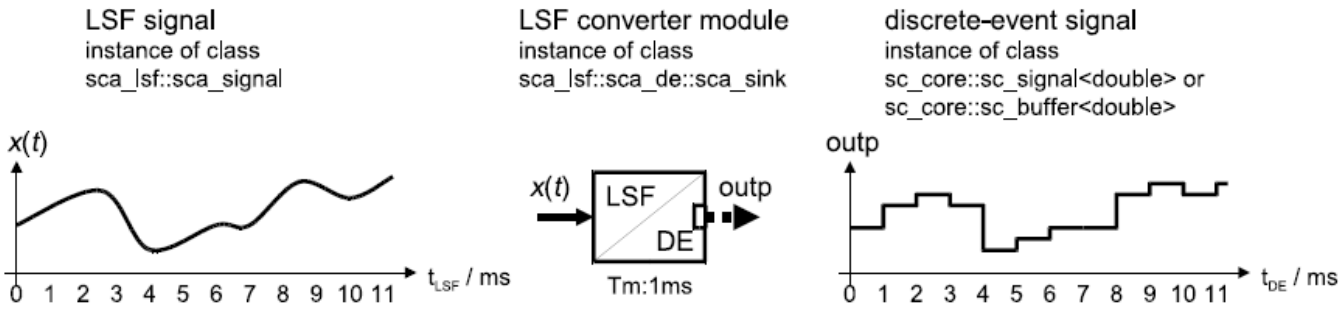


Figure 3.6. LSF converter module reading from an LSF input signal and writing to a discrete-event output signal

Рисунок 3.6. Модуль чтения LSF считывает из входного сигнала LSF и записывает в выходной сигнал дискретного события

3.4.2. Чтение и запись в модели TDF

Аналогичным образом, модели LSF можно подключать к моделям TDF с помощью модулей преобразователя с внутренним портом класса sca_tdf :: sca_in или sca_tdf :: sca_out.

На рисунке 3.7 показан примитивный модуль LSF sca_lsf :: sca_tdf :: sca_source, считывающий сигнал TDF и записывающий в сигнал LSF. В этом примере временной шаг модуля 1 мс назначается модулю преобразователя LSF.

Модель LSF непрерывно считывает сэмплы с входа TDF. Входные образцы интерпретируются, чтобы сформировать непрерывный сигнал, доступный на выходе модуля преобразователя (входные выборки показаны в виде пунктирный сигнал).

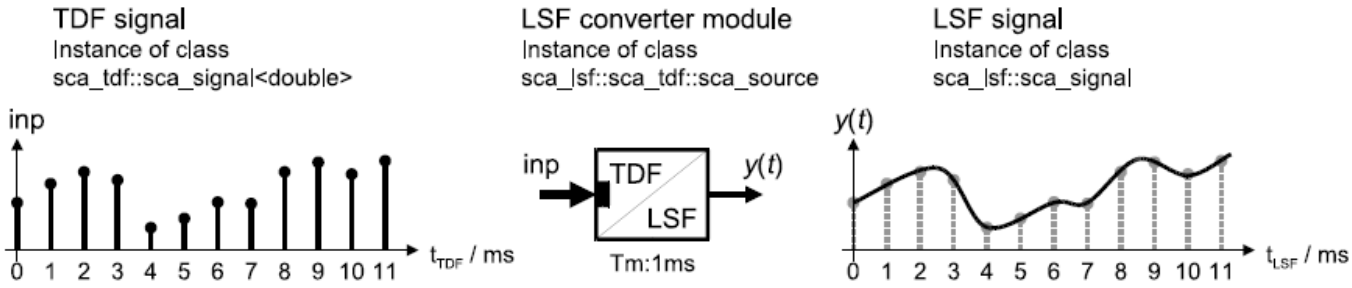


Figure 3.7. LSF converter module reading from a TDF input signal and writing to an LSF output signal

Рисунок 3.7. Модуль чтения LSF считывает из входного сигнала TDF и записывает в выходной сигнал LSF

На рисунке 3.8 показан модуль примитива LSF sca_lsf :: sca_tdf :: sca_sink, считывающий сигнал LSF и записывающий эквивалентные значения для сигнала TDF. Образцы на выходном порту записываются в моменты времени, которые рассчитываются от заданного временного шага модуля 1 мс.

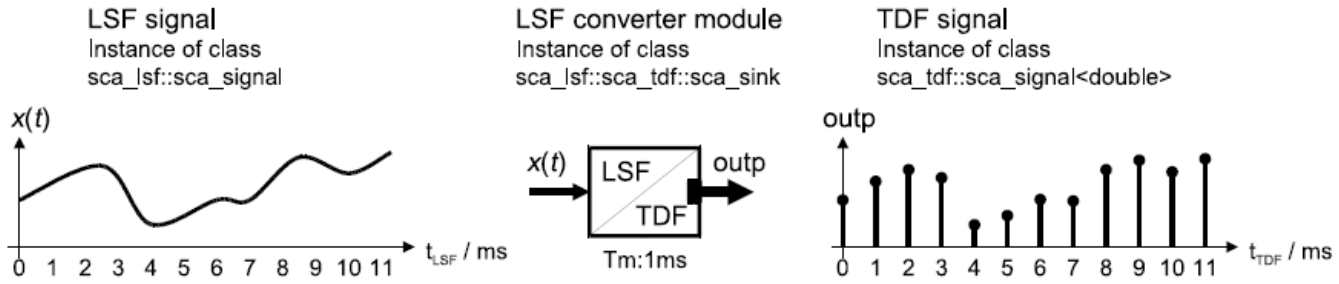


Figure 3.8. LSF converter module reading from an LSF input signal and writing to a TDF output signal

Рисунок 3.8. Модуль чтения LSF считывает из входного сигнала LSF и записывает в выходной сигнал TDF

3.4.3. Использование дискретных событий или сигналов управления TDF

Хотя нет принципиальных отличий от модулей преобразователя LSF, описанных в предыдущих две разделах, дополнительные примитивы LSF доступны для управления или масштабирования переменных или сигналов в системе уравнений LSF. Примитивы LSF, используемые для управления, могут быть идентифицированы с помощью имеющего входной порт класса sc_core ::

sc_in или sca_tdf :: sca_in для типа данных bool. Примеры являются мультиплексорами (sca_lsf :: sca_de :: sca_mux и sca_lsf :: sca_tdf :: sca_mux) и демультиплексорами (sca_lsf :: sca_de :: sca_demux и sca_lsf :: sca_tdf :: sca_demux). Примитивы, которые могут масштабировать переменные или сигналы, используют одни и те же порты, но используют тип данных double. Примерами являются примитивы умножения (sca_lsf :: sca_de :: sca_gain и sca_lsf :: sca_tdf :: sca_gain). Обратите внимание, что если параметр LSF модуля изменился, соответствующая система уравнений LSF будет переинициализирована.

На рисунке 3.9 показан пример использования примитивов LSF в структурной модели для управления или масштабирования сигналов.

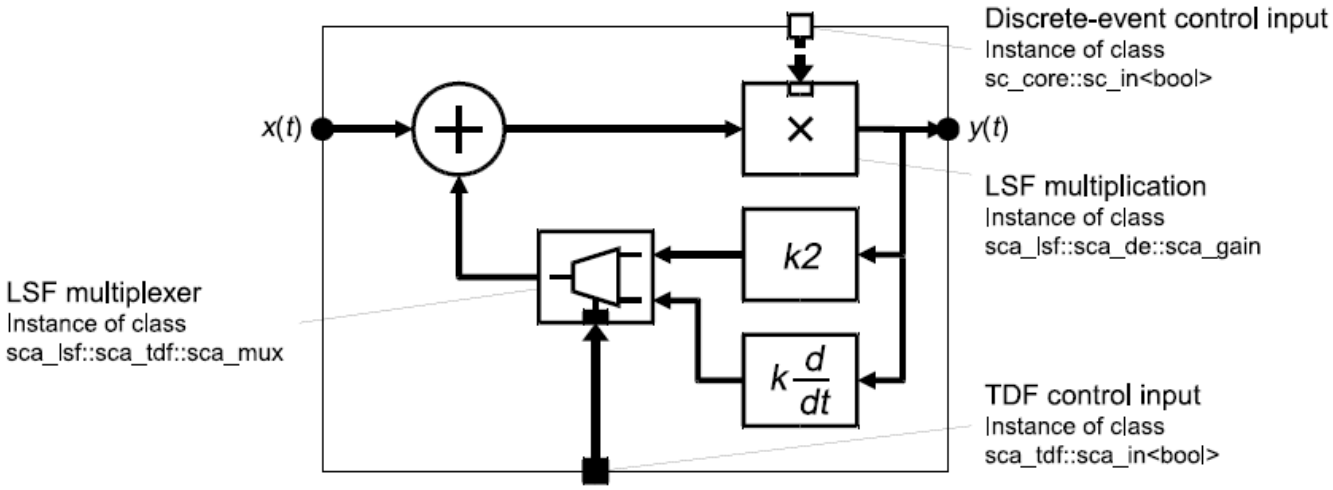


Figure 3.9. LSF model using discrete-event and TDF control signals

Рисунок 3.9. Модель LSF с использованием сигналов дискретного события и TDF контрольных сигналов

Аналогично модулям преобразователя LSF, описанным в разделе 3.4, управляющие сигналы дискретного события или TDF читаются с фиксированным временным шагом, который соответствует временному шагу модуля. Только тогда система уравнений LSF будет обновлена.

3.4.4. Инкапсуляция модели LSF

Модули преобразователя, описанные в предыдущих разделах, могут использоваться для инкапсуляции модели LSF в другую модель вычисления. На рис. 3.10 показан пример использования модулей преобразователя для из модели вычислений TDF для инкапсуляции поведения LSF. В этом случае доступ к системе уравнений LSF использует сигналы с дискретным временем в соответствии с семантикой TDF, тогда как внутренние LSF сигналы и вычисления непрерывны. Этот подход дает еще одну возможность встраивать поведение непрерывного времени в модели вычисления TDF, кроме встроенных линейных динамических уравнений для TDF модулей, описанных в разделе 2.3.2.

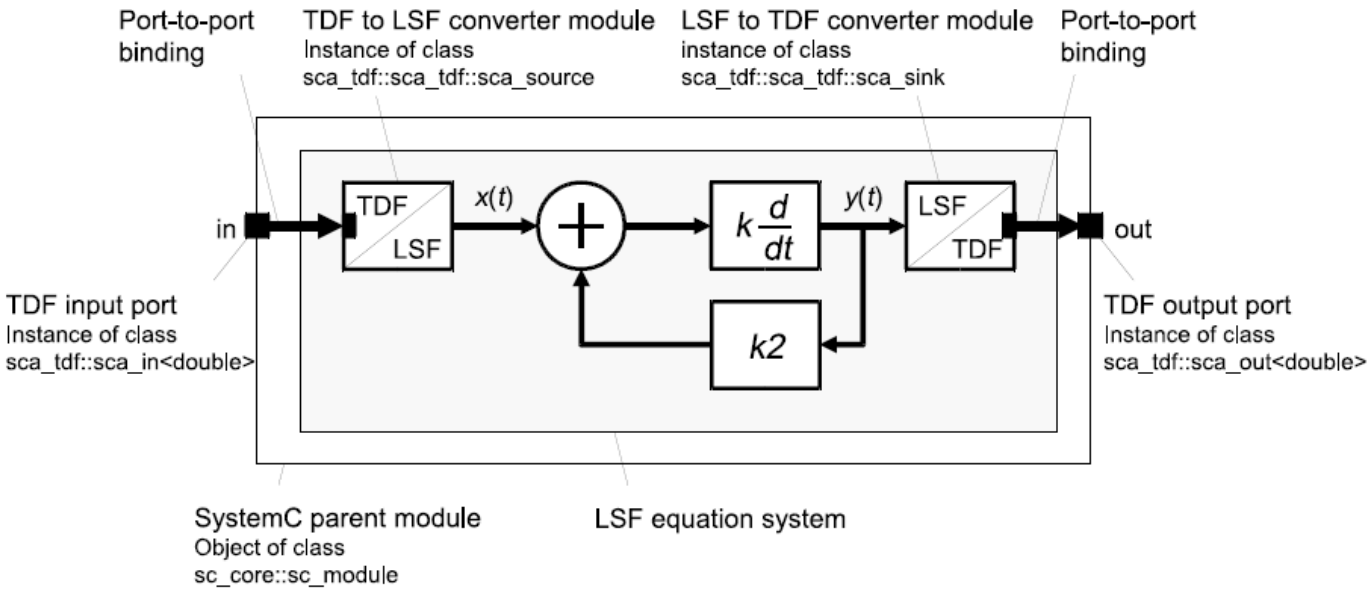


Figure 3.10. LSF equation system encapsulated for inclusion into a structural TDF model description by using converter modules

Рисунок 3.10. Система уравнений LSF, инкапсулированная для включения в описание структурной модели TDF с использованием конвертерных модулей

В приведенном ниже примере показана реализация рисунка 3.10.

```
SC_MODULE(lsf_in_tdf)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
sca_lsf::sca_add add1;
sca_lsf::sca_dot dot1;
sca_lsf::sca_gain gain1;
sca_lsf::sca_tdf::sca_source tdf2lsf;
sca_lsf::sca_tdf::sca_sink lsf2tdf;
lsf_in_tdf( sc_core::sc_module_name, double k, double
k2 )
: in("in"), out("out"), add1("add1"), dot1("dot1",
k), gain1("gain1", k2), tdf2lsf("tdf2lsf"),
lsf2tdf("lsf2tdf"), sig1("sig1"), sig2("sig2"),
sig3("sig3"), sig4("sig4")
{
tdf2lsf.inp(in);
tdf2lsf.y(sig1);
add1.x1(sig1);
add1.x2(sig3);
```

```
add1.y(sig2);
dot1.x(sig2);
dot1.y(sig4);
gain1.x(sig4);
gain1.y(sig3);
lsf2tdf.x(sig4);
lsf2tdf.outp(out);
}
private:
sca_lsf::sca_signal sig1, sig2, sig3, sig4;
};
```

Аналогичный подход можно использовать для инкапсуляции модели LSF для включения в структурное описание модели дискретного события с использованием модулей преобразования в и из области дискретных событий, как описано в Раздел 3.4.1.

3.5. Семантика исполнения LSF

В дополнение к этапам разработки и моделирования, как это определено в стандарте языка SystemC IEEE 1666-2005, специфическая функциональность реализована для разработки и исполнения моделей LSF.

Как показано на рисунке 3.11, этап разработки включает в себя следующие этапы:

• Расчет и распространение временного шага LSF: определите временной шаг и проверьте согласованность внутри каждого LSF модель (см. также раздел 3.1.2).

• Настройка уравнения LSF и проверка разрешимости: составьте систему уравнений LSF из уравнения, представленных предопределенными примитивными модулями LSF, и их взаимосвязь, определяемую композицией. Проверьте, может ли полученная система уравнений быть решена.

Шаги для этапа моделирования:

• Инициализация LSF: сначала установите все сигналы LSF на ноль, а затем установите начальные условия системы на основе на потенциально определенных начальных условиях примитивов LSF.

• LSF моделирование во временной области: система уравнений LSF решается численно с использованием подходящих шагов времени, которые могут быть меньше назначенного временного шага. Решатель, по крайней мере, даст результаты для временных точек, рассчитанных по заданному временному шагу.

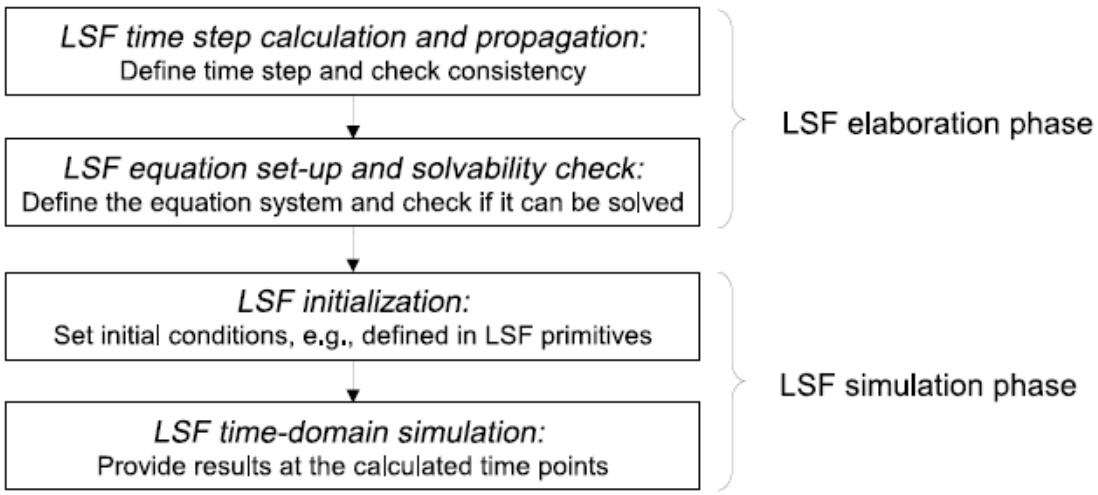


Figure 3.11. LSF elaboration and simulation phases

Рисунок 3.11. Фазы разработки и моделирования LSF

Этап разработки и моделирования выполняется путем запуска моделирования во временной области с использованием функции sc_core :: sc_start.

3.6. Примеры применения

В этом разделе показаны некоторые основные примеры применения с использованием моделирования линейного потока сигналов.

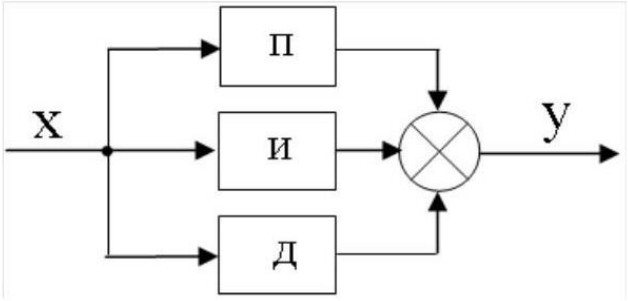
3.6.1. ПИД-регулятор

Формализм LSF моделирования очень подходит для моделирования систем управления. Пример такого системы контроля показана на рисунке 3.12. Этот пример показывает использование Пропорционального - Интегро – Дифференциального регулятора (PID).

Дифференциальный пропорционально-интегральный регулятор -- устройство, которое устанавливают в автоматизированных системах для поддержания заданного параметра, способного к изменениям.

ПИД регулятор -- прибор, встроенный в управляющий контур, с обязательной обратной связью. Он предназначен для поддержания установленных уровней задаваемых величин, например, температуры воздуха. Устройство подаёт управляющий или выходной сигнал на устройство регулирования, на основании полученных данных от датчиков или сенсоров. Контроллеры обладают высокими показателями точности переходных процессов и качеством выполнения поставленной задачи.

ПИД-регулятор (пропорционально-интегро-дифференциальный регулятор)



$$W(s) = K_1 + K_2 s + \frac{K_3}{s}$$

Контроллер ПИД является частью петли обратной связи управления. На вход ПИД-регулятора подается сигнал ошибки e (t), который является разницей между измеренным выходным значением y (t) определенного устройства и требуемым опорным входом y_0.

Управляющий выход u(t), генерируемый ПИД-регулятором, который регулирует поведение устройства под управления, будет таким, что сигнал ошибки будет минимизирован. Отклик и поведение ПИД и ошибка контроллера, вызванная (внезапным) изменением эталонного входного или выходного значения, зависит от характеристики ПИД-регулятора и определяются параметрами Kp, Ki и Kd.

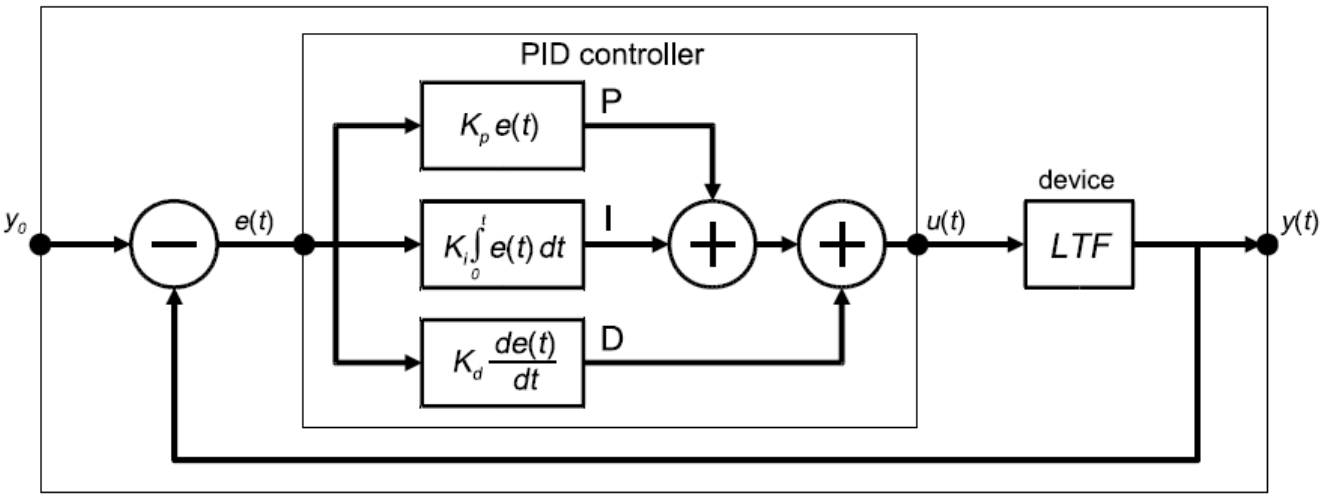


Figure 3.12. Block diagram of a PID controller within a control loop

Рисунок 3.12. Блок-схема ПИД-регулятора в контуре управления

Параметры Kp, Ki и Kd используются в ПИД-регуляторе для установки пропорционального, интегрального и производныго слагаемых, которые затем суммируются для расчета контрольного выхода. Система уравнений ПИД контроллера, в котором e (t) является входным сигналом ошибки, а u(t) является

выходом контроллера, затем становится такой:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_o^t e(t)dt + K_d \cdot \frac{de(t)}{dt}$$

Контроллер PID может быть реализован с использованием примитивных модулей LSF в родительском модуле, как показано ниже:

```
SC_MODULE(pid_controller)
{
sca_lsf::sca_in e;
sca_lsf::sca_out u;
sca_lsf::sca_gain gain1;
sca_lsf::sca_integ integ1;
sca_lsf::sca_dot dot1;
sca_lsf::sca_add add1, add2;
pid_controller( sc_core::sc_module_name, double kp,
double ki, double kd )
: e("e"), u("u"), gain1("gain1", kp),
integ1("integ1", ki), dot1("dot1", kd), add1("add1"),
add2("add2"), sig_p("sig_p"), sig_i("sig_i"),
sig_d("sig_d"), sig_pi("sig_pi")
{
gain1.x(e);
gain1.y(sig_p);
integ1.x(e);
integ1.y(sig_i);
dot1.x(e);
dot1.y(sig_d);
add1.x1(sig_p);
add1.x2(sig_i);
add1.y(sig_pi);
add2.x1(sig_pi);
add2.x2(sig_d);
add2.y(u);
}
private:
sca_lsf::sca_signal sig_p, sig_i, sig_d, sig_pi;
};
```

1. Для суммирования пропорциональных, интегральных и производных членов в качестве примитива используются два сумматора, так как модуль сумматора имеет только два входа.

2. Параметры для ПИД - регулятора могут быть назначены через конструктор, который позволяет их настройку из родительского модуля (или

функции sc_main), в котором создается экземпляр ПИД-регулятора.

3.6.2. Непрерывный сигма-дельта модулятор

На рисунке 3.13 показано применение сигма-дельта-модулятора с непрерывным временем (CTSD), содержащий контурный фильтр H (s) (фильтр в цепи обратной связи), квантователь и цифроаналоговый преобразователь (ЦАП) в тракте обратной связи.

Циклический фильтр реализован с использованием примитивов LSF. Квантизатор и ЦАП реализованы в виде модулей TDF.

Модули преобразователя LSF *в* и *из* модели вычисления TDF используются, чтобы иметь возможность преобразовывать образцы выходного сигнала фильтра непрерывного времени U (s) в сигнал области дискретного времени V (z) и для преобразования дискретного времени выходного сигнала ЦАП W (z) к непрерывному сигналу обратной связи T (s).

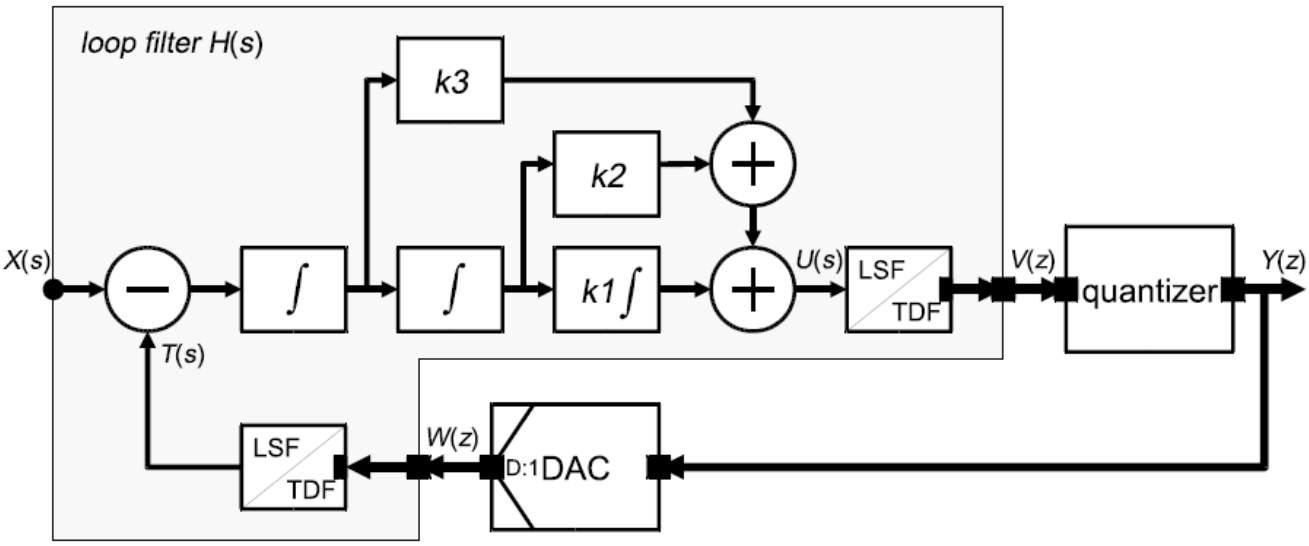


Figure 3.13. Block diagram of a continuous-time sigma-delta (CTSD) modulator

Рисунок 3.13. Структурная схема непрерывного сигма-дельта модулятора (CTSD)

Циклический фильтр 3-го порядка реализован с использованием трех интеграторов, которые каскадируются и суммируются с весовыми коэффициентами k1, k2 и k3. Соответствующая передаточная функция H (s) для этого фильтра контура становится такой:

$$H(s) = \frac{k_1 s^2 + k_2 s + k_3}{s^3}$$

Циклический фильтр может быть реализован с использованием примитивных модулей LSF в родительском модуле, как показано ниже:

```
    SC_MODULE(ctsd_loop_filter)
    {
    sca_lsf::sca_in x;
    sca_tdf::sca_out<double> v;
    sca_tdf::sca_in<double> w;
    sca_lsf::sca_tdf::sca_source tdf2lsf;
    sca_lsf::sca_sub sub1;
    sca_lsf::sca_integ integ1, integ2, integ3;
    sca_lsf::sca_gain gain2, gain3;
    sca_lsf::sca_add add1, add2;
    sca_lsf::sca_tdf::sca_sink lsf2tdf;
    ctsd_loop_filter( sc_core::sc_module_name, double k1,
double k2, double k3 )
    : x("x"), v("v"), w("w"), tdf2lsf("tdf2lsf"),
sub1("sub1"), integ1("integ1", k1), integ2("integ2"),
    integ3("integ3"), gain2("gain2", k2), gain3("gain3",
k3), add1("add1"), add2("add2"),
    lsf2tdf("lsf2tdf"), sig_t("sig_t"), sig_i("sig_1"),
sig_i1("sig_i1"), sig_i2("sig_i2"),
    sig_i3("sig_i3"), sig_a1("sig_a1"), sig_a2("sig_a2"),
sig_a3("sig_a3"), sig_u("sig_u")
    {
    tdf2lsf.inp(w);
    tdf2lsf.y(sig_t);
    sub1.x1(x);
    sub1.x2(sig_t);
    sub1.y(sig_i);
    integ3.x(sig_i);
    integ3.y(sig_i3);
    integ2.x(sig_i3);
    integ2.y(sig_i2);
    integ1.x(sig_i2);
    integ1.y(sig_i1);
    gain3.x(sig_i3);
    gain3.y(sig_a1);
    gain2.x(sig_i2);
    gain2.y(sig_a2);
    add1.x1(sig_a1);
    add1.x2(sig_a2);
    add1.y(sig_a3);
    add2.x1(sig_a3);
    add2.x2(sig_i1);
    add2.y(sig_u);
    lsf2tdf.x(sig_u);
    lsf2tdf.outp(v);
```

```
    }
    private:
    sca_lsf::sca_signal sig_t, sig_i, sig_i1, sig_i2,
sig_i3;
    sca_lsf::sca_signal sig_a1, sig_a2, sig_a3, sig_u;
    };
```

3.7. Фильтрация шума и АЦП

3.7.1. Применение фильтра нижних частот для сигнала с шумом

На схеме (рис. 3.14) показан испытательный стенд, в котором к генератору синусоидального сигнала с шумом добавлен предварительный фильтр нижних частот.

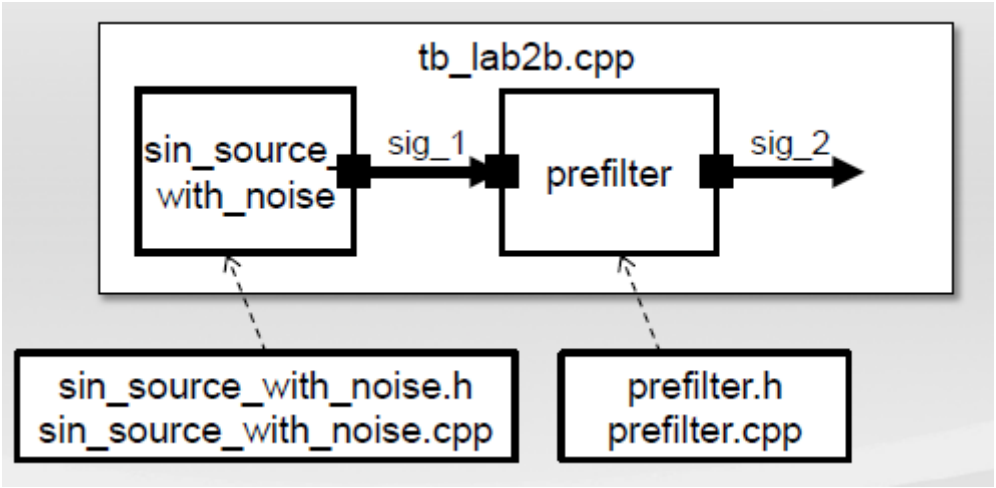


Рис. 3.14. Генератор синусоиды с шумом и предварительный фильтр

В файле prefilter.cpp надо назначить 1 кГц переменной – члену fc частоты среза в конструкторе.

Передаточная функция фильтра показана на рис. 3.15.

$$H(s) = \frac{b_n \cdot s^n + b_{n-1} \cdot s^{n-1} + \ldots + b_0}{a_m \cdot s^m + a_{m-1} \cdot s^{m-1} + \ldots + a_0}$$

$$H(s) = \frac{1}{1 + \frac{1}{2\pi f_c} s}$$

Рис. 3.15. Передаточная функция предварительного фильтра

Листинг 3.1

Заголовочный файл SIN_SOURCE_WITH_NOISE_H

```
    #ifndef SIN_SOURCE_WITH_NOISE_H
    #define SIN_SOURCE_WITH_NOISE_H

    #include <systemc-ams.h>        // SystemC AMS header

    SCA_TDF_MODULE(sin_source_with_noise)  // Declare a
TDF module
    {
      sca_tdf::sca_out<double> out;    // TDF output port

      void set_attributes();      // Set TDF attributes

      void processing();     // Describe time-domain
behaviour

      void ac_processing();  // Describe freq-domain
behaviour

      SCA_CTOR(sin_source_with_noise)    // Constructor
of the TDF module
      : out("out"),             // Name the port(s)
        ampl(1.0), freq(1e3),variance(0.1) {}  // Initial
values for ampl and freq

     private:            // Private variables
      double ampl;       //   amplitude
      double freq;        //   frequency
      double variance;   //   variance for noise
    };

    #endif /* SIN_SOURCE_WITH_NOISE_H */
```

Листинг 3.2

Исполняемый файл `sin_source_with_noise`

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
------------------
```

```
    #include "sin_source_with_noise.h"
    #include "systemc-ams.h"

    #include <cstdlib> // for std::rand
    #include <cmath>   // for M_PI, std::sin, std::sqrt,
and std::log

    double gauss_rand(double variance)
    {
      double rnd1, rnd2, Q, Q1, Q2;

      do
      {
        rnd1 = static_cast<double>(std::rand()) /
RAND_MAX;
        rnd2 = static_cast<double>(std::rand()) /
RAND_MAX;

        Q1 = 2.0 * rnd1 - 1.0;
        Q2 = 2.0 * rnd2 - 1.0;
        Q = Q1 * Q1 + Q2 * Q2;
      }
      while (Q > 1.0);

      return ( std::sqrt(variance) * ( std::sqrt( - 2.0 *
std::log(Q) / Q) * Q1) );
    }

    void sin_source_with_noise::set_attributes()    //
Set TDF attributes
    {
      out.set_timestep(1.0, SC_US);                 //
Set time step of output port
    }

    void sin_source_with_noise::processing()
    {
      double t = out.get_time().to_seconds();       //
Get current time of the sample
      double n = gauss_rand(variance);
      double x = ampl * sin(2.0 * 3.1415 * freq * t) + n;
// Calculate sine wave
      out.write(x);
// Write sample to the output
    }
```

```
    void sin_source_with_noise::ac_processing()
    {
      sca_ac(out) = 1.0;
    }
```

Листинг 3.3

Заголовочный файл `PREFILTER_H`

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
------------------

    #ifndef PREFILTER_H
    #define PREFILTER_H

    #include <systemc-ams.h>

    SCA_TDF_MODULE(prefilter)
    {
      sca_tdf::sca_in<double> in;
      sca_tdf::sca_out<double> out;

      void initialize();
      void processing();
      void ac_processing();

      SCA_CTOR(prefilter);

    private:
      sca_vector<double> num, den;
      sca_tdf::sca_ltf_nd ltf1;
      double fc;
    };

    #endif /* PREFILTER_H */
```

Листинг 3.4

Исполняемый файл `prefilter.cpp`

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
------------------

    #include "prefilter.h"
    #include "systemc-ams.h"

    prefilter::prefilter(sc_core::sc_module_name nm)
    {
      fc = 1e3;
    }

    void prefilter::initialize()
    {
      num(0) = 1.0;
      den(0) = 1.0;
      den(1) = 1.0 / (2.0 * 3.1415 * fc);
    }

    void prefilter::processing()
    {
      out = ltf1(num, den, in);
    }

    void prefilter::ac_processing()
    {
      sca_ac(out) = sca_ac_ltf_nd(num, den, sca_ac(in));
    }
```

Листинг 3.5

Исполняемый файл `testbench.cpp`

```
    include "sin_source_with_noise.h"
    #include "prefilter.h"
    #include "systemc-ams.h"

    int sc_main(int argn,char* argc[])
    {
      sca_tdf::sca_signal<double> sig_1, sig_2;
```

```
      sin_source_with_noise sin1("sin1");
      sin1.out(sig_1);

      prefilter prefi1("lp1");
      prefi1.in(sig_1);
      prefi1.out(sig_2);

      sca_trace_file* tfp =
        sca_create_tabular_trace_file("tb_lab2b.dat");

      sca_trace(tfp, sig_1, "sig_1");
      sca_trace(tfp, sig_2, "sig_2");

      sc_start(5.0, SC_MS);

      tfp->reopen("tb_ac_lab2b.dat");
tfp->set_mode(sca_ac_format(sca_util::SCA_AC_DB_DEG));

      sca_ac_start(1.0, 1e6, 1000, SCA_LOG);

      return 0;
```

Выполняя моделирование этого примера, сделайте следующее:

1. Проверьте, правильно ли компилируется тестовый стенд.

2. Выполните 10 мс моделирования во временной области.

3. Создайте табличный файл трассировки и отследите глобальный сигнал sig_1. Запустите симулятор и просмотрите результаты в программе просмотра формы волны.

4. Создайте новый табличный файл трассировки, выполните анализ переменного тока (от 1 Гц до 1 МГц, с логарифмическим интервалом 1000 точек) и просмотрите результаты в средстве просмотра формы сигнала.

5. В файле prefilter.cpp назначьте 1 кГц переменной-члену fc частоты среза в конструкторе.

6. На бумаге определите соответствующие коэффициенты числителя и знаменателя для представления фильтра, используя передаточную функцию Лапласа (дана подсказка). После этого откройте файл prefilter.h и завершите MINITASK 2.3.

7. В файле prefilter.h добавьте двойные векторы с именами num и den в качестве переменных-членов. Также добавьте переменную-член ltf1 типа sca_tdf :: sca_ltf_nd. Затем перейдите к функции initialize () в prefilter.cpp и, соответственно, установите коэффициенты num и den.

8. Затем перейдите к функции initialize () в prefilter.cpp и соответственно установите коэффициенты num и den.

9. В файле prefilter.cpp в функции processing () запишите соответствующий вызов функции LTF, используя ltf1, num, den, входную

выборку и выходную выборку.

10. В файле tb_lab2b.cpp отследите сигналы sig_1 и sig_2, запустите симуляцию во временной области 10 мс и просмотрите результаты в средстве просмотра формы сигнала.

11. В файле prefilter.cpp измените функцию ac_processing () для выполнения вызова функции sca_ac_ltf_nd ().

12. В файле tb_lab2b.cpp выполните анализ AC (используйте предыдущие параметры).

}

В результате решения мы получили численные значения сигналов sig_1 и sig_2 в папке tb_lab2b.dat (рис. 3.16).

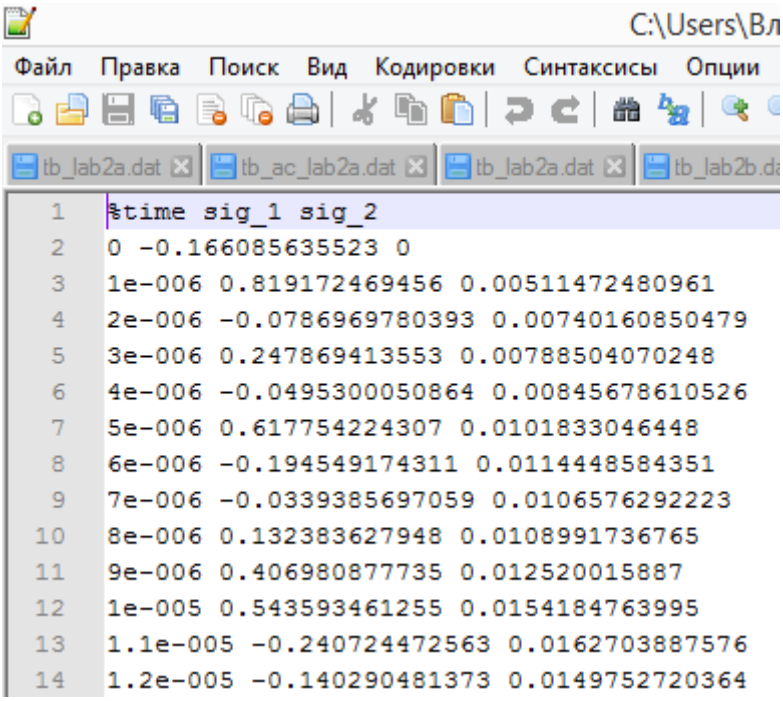


Рис. 3.16. Численные значения сигналов в папке tb_lab2b.dat

В папке tb_ac_lab2b.dat (рис. 3.17) данные предствлены в другом логарифмическом формате с указание текущих фаз.

tb_lab2a.dat | tb_ac_lab2a.dat | tb_lab2a.dat | tb_lab2b.dat | tb_ac_lab2b.dat

```
%frequency sig_1.db sig_1.deg sig_2.db sig_2.deg
1  0 0  -4.34319882738e-006 -0.0572974502615
1.01392540756  0 0  -4.46500261314e-006 -0.0580953400653
1.02804473209  0 0  -4.59022234923e-006 -0.0589043407869
1.04236067398  0 0  -4.71895383408e-006 -0.0597246071492
1.05687597118  0 0  -4.85129555756e-006 -0.06055629603
1.07159339982  0 0  -4.98734876464e-006 -0.0613995664913
1.08651577465  0 0  -5.12721754119e-006 -0.0622545798103
1.10164594963  0 0  -5.27100889599e-006 -0.0631214995099
1.11698681847  0 0  -5.41883283497e-006 -0.0640004913902
1.13254131515  0 0  -5.57080245375e-006 -0.0648917235598
1.14831241454  0 0  -5.72703401196e-006 -0.0657953664686
1.16430313292  0 0  -5.88764703733e-006 -0.0667115929398
1.18051652857  0 0  -6.05276440673e-006 -0.0676405782032
1.19695570236  0 0  -6.2225124416e-006 -0.0685824999286
1.21362379834  0 0  -6.39702100735e-006 -0.0695375382601
1.23052400436  0 0  -6.57642361066e-006 -0.0705058758502
1.24765955263  0 0  -6.76085750661e-006 -0.0714876978947
1.2650337204  0 0  -6.95046379121e-006 -0.0724831921685
1.28264983053  0 0  -7.14538752579e-006 -0.0734925490611
```

Рис. 3.17. Численные значения сигналов в папке tb_ac_lab2b.dat

Существуют программы для отображения вида функций по данным в формате .dat.

Мы воспользуемся программой GTKWave. Для этого в исполняемом файле `testbench.cpp` заменим `tabular` на `vcd`. При этом в результате моделирования в папке проекта будут файлы с расширением .dat (рис. 3.18).

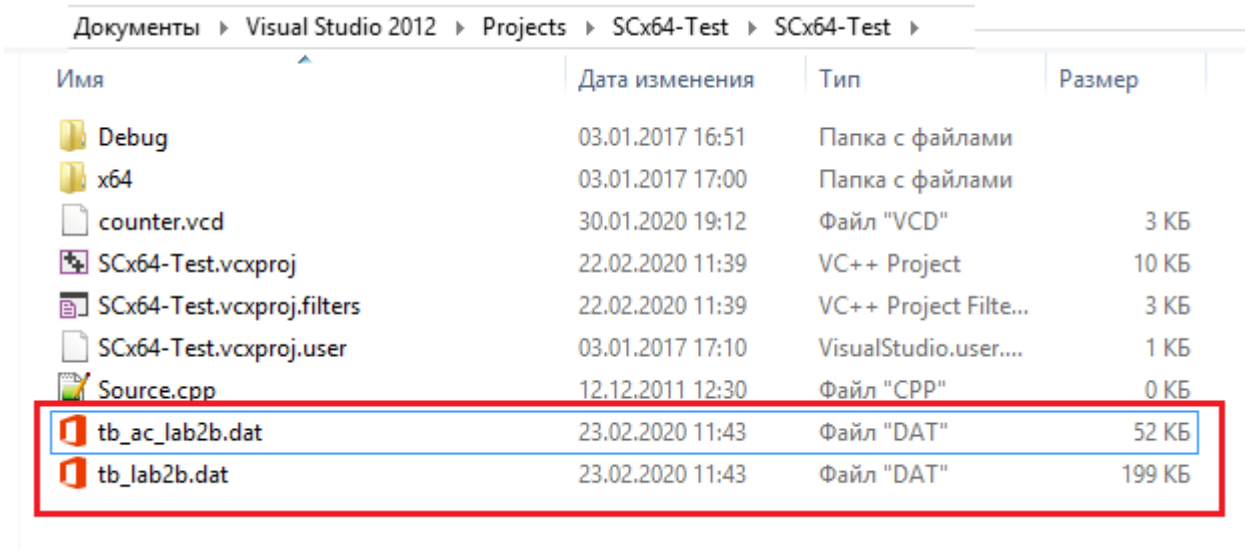


Рис. 3.18. Файлы решения в папке проекта

Переместим эти два файла в папку vcd-files и откроем в GTKWave.

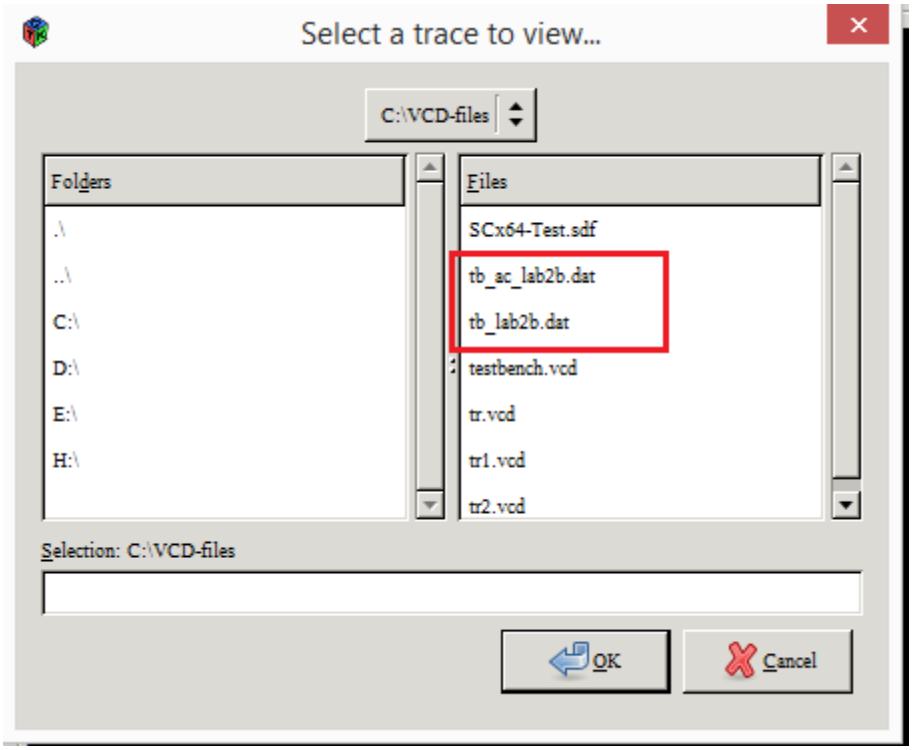


Рис. 3.19. Выбор файлов в GTKWave

Откроем сначала два файла sig_1 и sig_2 из папки tb_lab2b.dat, а затем из папки tb_ac_lab2b.dat. Результаты показаны на рис. 3.20. Как видно, результаты практически совпадают, причем фильтрация устраняет шум в sig_2

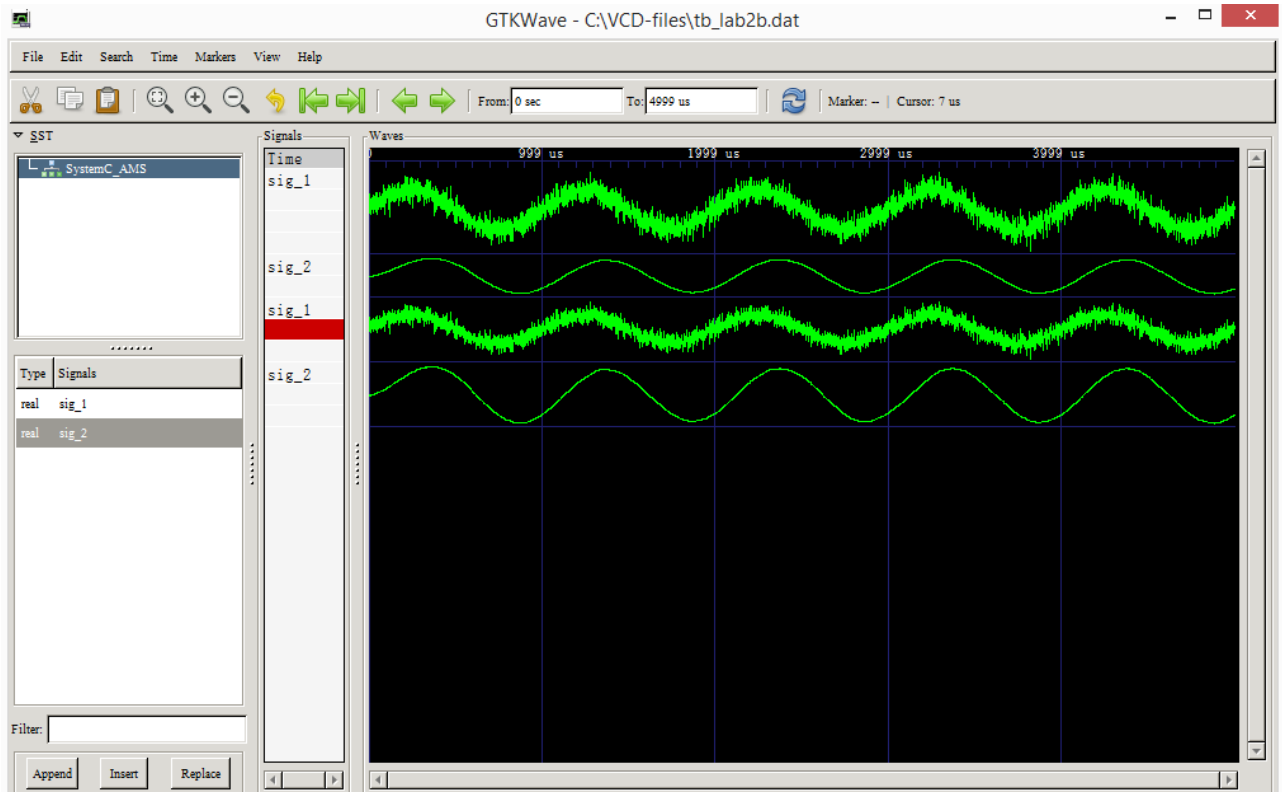


Рис. 3.20. Результаты моделирования фильтрации зашумленного сигнала

3.7.2. Моделирование фильтрации и создание преобразователя ΣΔ АЦП

На рис. 3.21 показана схема фильтрации синусоидального сигнала с шумом с последующим преобразованием в цифровую форму с использованием сигма-дельта АЦП.

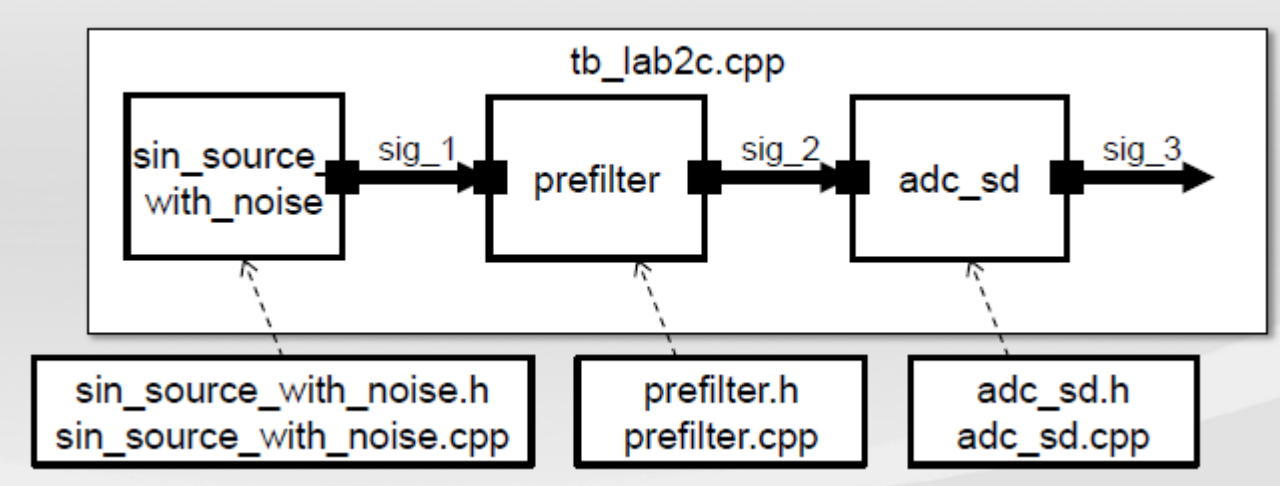


Рис. 3.21. Схема фильтрации и АЦП

На листингах 3.6 – 3.12 приведены программы этой модели.

Листинг 3.6

Заголовочный файл SIN_SOURCE_WITH_NOISE_H

```
    //   Original Author: Karsten Einwich Fraunhofer
IIS/EAS Dresden
    //
    //   Created on: 16.02.2010
    //
    //-----------------------------------------------
------------------

    #ifndef SIN_SOURCE_WITH_NOISE_H
    #define SIN_SOURCE_WITH_NOISE_H

    #include <systemc-ams.h>        // SystemC AMS header

    SCA_TDF_MODULE(sin_source_with_noise) // Declare a
TDF module
    {
      sca_tdf::sca_out<double> out;   // TDF output port

      void set_attributes();   // Set TDF attributes

      void processing();     // Describe time-domain
behaviour
```

```
      void ac_processing();    // Describe freq-domain
behaviour

      SCA_CTOR(sin_source_with_noise) // Constructor of
the TDF module
      : out("out"),    // Name the port(s)
        ampl(1.0), freq(1e3),variance(0.1) {}  // Initial
values for ampl and freq

     private: // Private variables
      double ampl;      //   amplitude
      double freq; //   frequency
      double variance;  //   variance for noise
    };

    #endif /* SIN_SOURCE_WITH_NOISE_H */
```

Листнинг 3.7

Исполняемый файл `sin_source_with_noise.cpp`

```
    #include "sin_source_with_noise.h"

    #include <cstdlib> // for std::rand
    #include <cmath>   // for M_PI, std::sin, std::sqrt,
and std::log

    double gauss_rand(double variance)
    {
      double rnd1, rnd2, Q, Q1, Q2;

      do
      {
        rnd1 = static_cast<double>(std::rand()) /
RAND_MAX;
        rnd2 = static_cast<double>(std::rand()) /
RAND_MAX;

        Q1 = 2.0 * rnd1 - 1.0;
        Q2 = 2.0 * rnd2 - 1.0;
        Q = Q1 * Q1 + Q2 * Q2;
      }
      while (Q > 1.0);
```

```
      return ( std::sqrt(variance) * ( std::sqrt( - 2.0 *
std::log(Q) / Q) * Q1) );
    }

    void sin_source_with_noise::set_attributes()    //
Set TDF attributes
    {
      out.set_timestep(1.0, SC_US);                 //
Set time step of output port
    }

    void sin_source_with_noise::processing()
    {
      double t = out.get_time().to_seconds();       //
Get current time of the sample
      double n = gauss_rand(variance);
      double x = ampl * sin(2.0 * 3.1415 * freq * t) + n;
// Calculate sine wave
      out.write(x);
// Write sample to the output
    }

    void sin_source_with_noise::ac_processing()
    {
      sca_ac(out) = 1.0;
    }
```

Листнинг 3.8

Заголовочный файл `PREFILTER_H`

```
    #ifndef PREFILTER_H
    #define PREFILTER_H

    #include <systemc-ams.h>

    SCA_TDF_MODULE(prefilter)
    {
      sca_tdf::sca_in<double> in;
      sca_tdf::sca_out<double> out;

      void initialize();
      void processing();
      void ac_processing();

      SCA_CTOR(prefilter);
```

```
 private:
  sca_vector<double> num, den;
  sca_tdf::sca_ltf_nd ltf1;
  double fc;
};

#endif /* PREFILTER_H */
```

Листинг 3.9

Исполняемый файл `prefilter.cpp`

```
#include "prefilter.h"

prefilter::prefilter(sc_core::sc_module_name nm)
{
  fc = 1e3;
}

void prefilter::initialize()
{
  num(0) = 1.0;
  den(0) = 1.0;
  den(1) = 1.0 / (2.0 * 3.1415 * fc);
}

void prefilter::processing()
{
  out = ltf1(num, den, in);
}

void prefilter::ac_processing()
{
  sca_ac(out) = sca_ac_ltf_nd(num, den, sca_ac(in));
}
```

Листинг 3.10

Заголовочный файл `ADC_SD_H`

```
#ifndef ADC_SD_H
#define ADC_SD_H

#include <systemc-ams.h>

SCA_TDF_MODULE(adc_sd)
{
```

```
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<bool>  out;

  void processing();
  void ac_processing();

  SCA_CTOR(adc_sd);

private:
  double integ1, integ2;
};


#endif /* ADC_SD_H */
```

Листинг 3.11

Исполняемый файл ADC_SD

```
#include "adc_sd.h"

adc_sd::adc_sd(sc_core::sc_module_name nm)
{
  integ1 = 0.0;
  integ2 = 0.0;
}

void adc_sd::processing()
{
  double a = 0.5;
  double b = 0.5;
  double c = 1.0;

  double feedback = (integ2 >= 0.0) ? 1.0 : -1.0;

  bool output = (integ2 >= 0.0) ? true : false;
  out.write(output);

  double s1 = in.read() - feedback;

  double k1 = a * s1;
  double s2 = k1 + integ1;

  double k3 = c * feedback;
  double s3 = integ1 - k3;

  double k2 = b * s3;
```

```
    double s4 = k2 + integ2;

    integ2 = s2;
    integ1 = s4;
  }

  void adc_sd::ac_processing()
  {
    sca_ac(out) = 0.25 / (sca_ac_z(2) - 1.5 *
sca_ac_z() + 0.75) * sca_ac(in);
  }
```

Листинг 3.12

Исполняемый файл `tb_lab2c.cpp`

```
  #include "sin_source_with_noise.h"
  #include "prefilter.h"
  #include "adc_sd.h"

  int sc_main(int argn,char* argc[])
  {
    sca_tdf::sca_signal<double> sig_1, sig_2;
    sca_tdf::sca_signal<bool> sig_3;

    sin_source_with_noise sin1("sin1");
    sin1.out(sig_1);

    prefilter prefi1("lp1");
    prefi1.in(sig_1);
    prefi1.out(sig_2);

    adc_sd adc1("adc1");
    adc1.in(sig_2);
    adc1.out(sig_3);

    sca_trace_file* tfp =
      sca_create_vcd_trace_file("tb_lab2c.dat");

    sca_trace(tfp, sig_1, "sig_1");
    sca_trace(tfp, sig_2, "sig_2");
    sca_trace(tfp, sig_3, "sig_3");

    sc_start(5.0, SC_MS);
```

```
        tfp->reopen("tb_ac_lab2c.dat");
        tfp-
>set_mode(sca_ac_format(sca_util::SCA_AC_DB_DEG));

        sca_ac_start(1.0, 1e6, 1000, SCA_LOG);
        sca_close_vcd_trace_file(tfp);          // Close
trace file

        return 0;
                                    }
```

В последнем файле мы заменили `tabular` на `vcd` в командах трассировки.

На рис. 2.44 показаны результаты компиляции программы. Результаты решения формируются в папках tb_lab2c.dat и tb_ac_lab2c.dat.

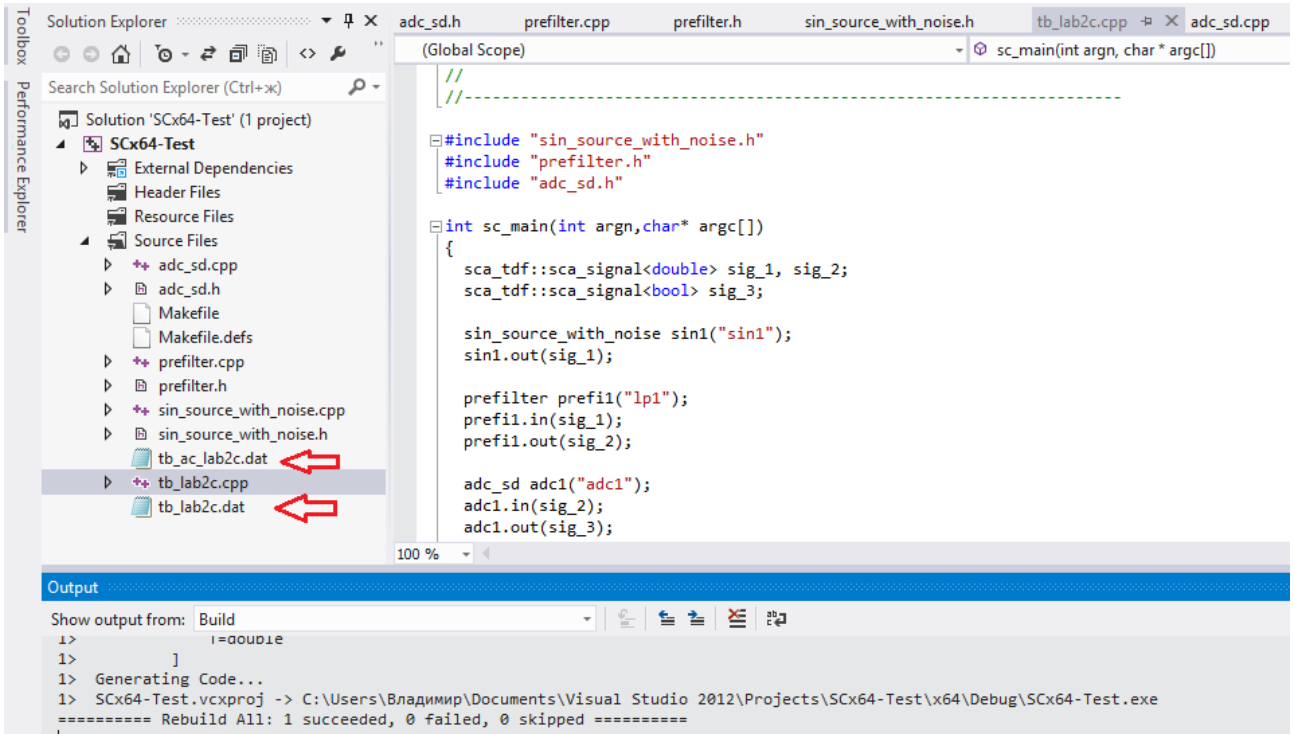


Рис. 3.22. Результаты компиляции программы

Сигналы из папок tb_lab2c.dat и tb_ac_lab2c.dat откроем в GTKWave. На рис. 3.23 показаны зашумленный sig_1 и отфильтрованный sig_2 сигналы.

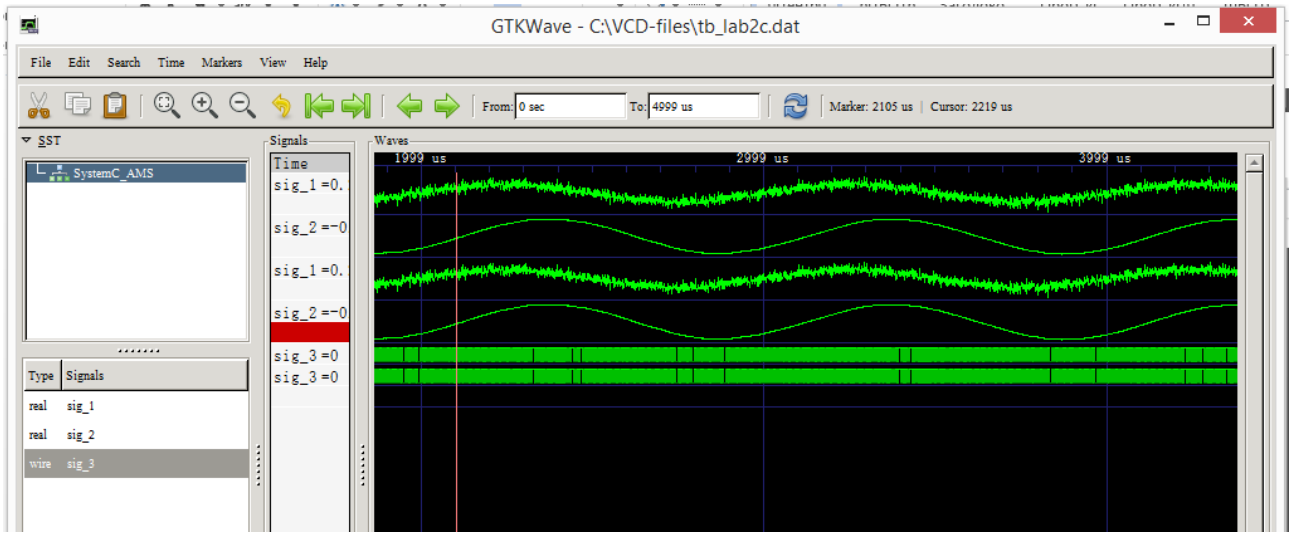


Рис. 3.23. Зашумленные и отфильтрованные сигналы

Сигнал сигма-дельта АЦП можно наблюдать, увеличив масштаб по времени (рис. 3.24). Длительность одного имрульса составляет 1 мкс.

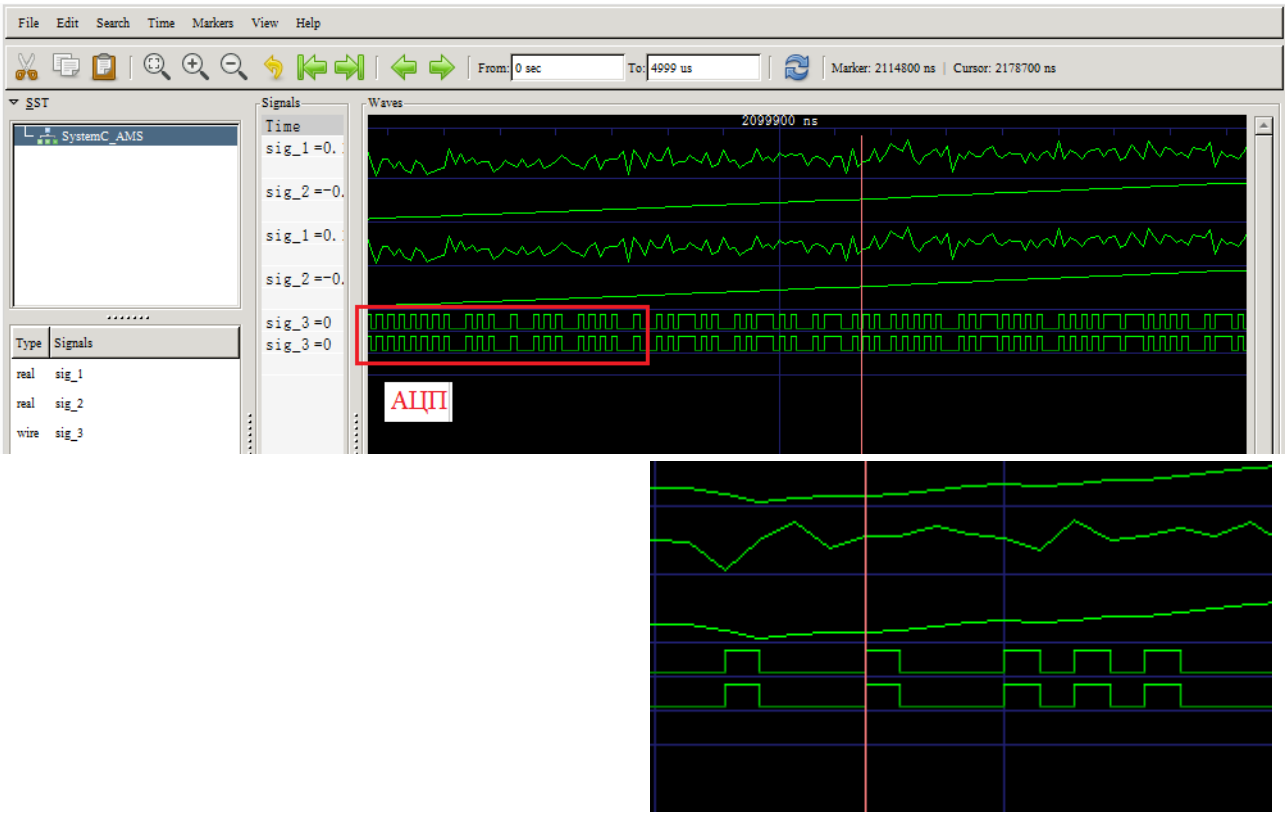


Рис. 3.24. Сигналы сигма-дельта АЦП

Посмотрим как можно реализовать поведение ADC_SD в модуле TDF adc_sd.h, adc_sd.cpp как C ++ - код. На рис. 3.25 ниже показана реализация (с a = 0.5, b = 0.5, c = 1.0). Входной порт TDF типа double, выходной порт TDF типа bool.

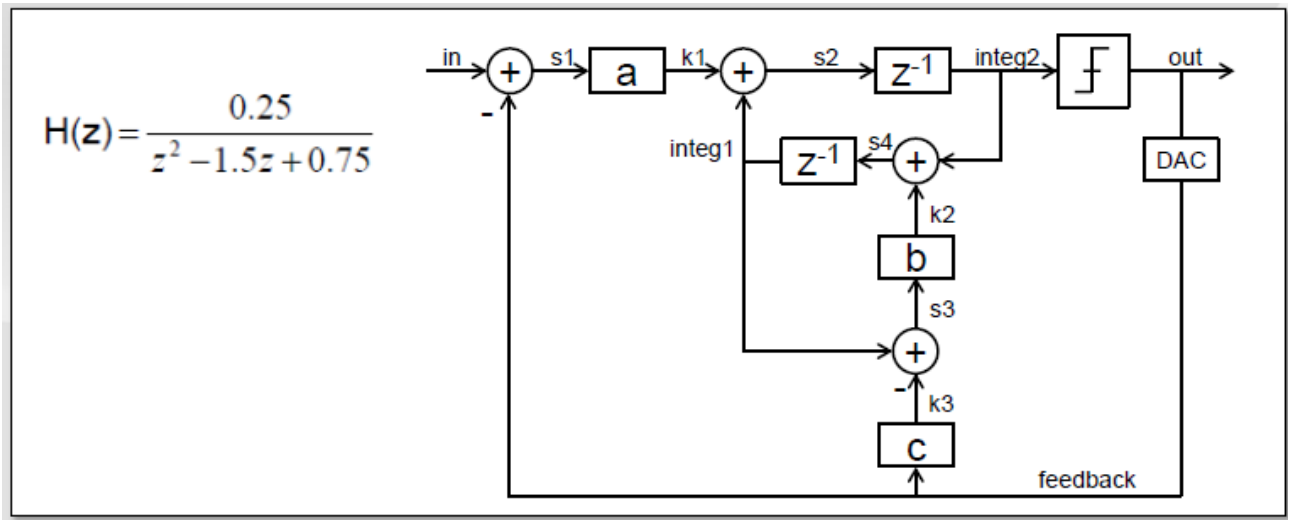


Рис. 3.25. Схема реализации сигма-дельта АЦП

1. В файле adc_sd.h определите две частные переменные с двойным членом, названные integ1 и integ2. Эти переменные содержат предыдущие / интегрированные значения (см. Рис. 3.25). Они читаются в начале функции processing () и записываются в конце.

2. В файле adc_sd.cpp в функции processing () объявите вывод локальной переменной bool. Присвойте ему значение true (если integ2> = 0.0) или false (если integ2 <0.0). Также объявите локальную двойную переменную обратную связь. Присвойте ему 1,0 (если целое число> = 0,0) или -1,0 (если целое число <0,0). Запишите выходное значение в выходной порт TDF.

3. В файле adc_sd.cpp в функции processing () с помощью рисунка объявите и присвойте s1, k1, s2, k2, k3, s3, s4 в правильном порядке. Некоторые присвоения (то есть s2 или s3) зависят от предыдущих / интегрированных значений, Integ1 и Integ2, которые просто используются в качестве RHS на данном этапе функции processing ().

4. В файле adc_sd.cpp в функции processing () заполните / разрешите циклы обратной связи с соответствующими назначениями переменных-членов Integ1 и Integ2 с только что вычисленными значениями s4 и s2.

5. В файле тестового стенда tb_lab2c.cpp выполните моделирование во временной области в течение 10 мс и отследите сигнал sig_3, поступающий от adc_sd.

3.7.3. Моделирование и создание гребенчатого фильтра

Дополним предыдущую схему гребенчатым фильтром (рис. 3.26)

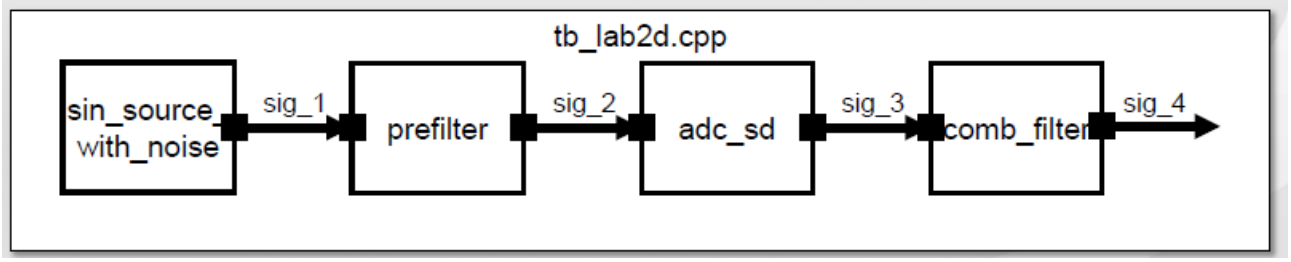


Рис. 3.26. Схема фильтрации и преобразования с гребенчатым фильтром

Принцип работы гребенчатого фильтра показан на рис. 3.27.

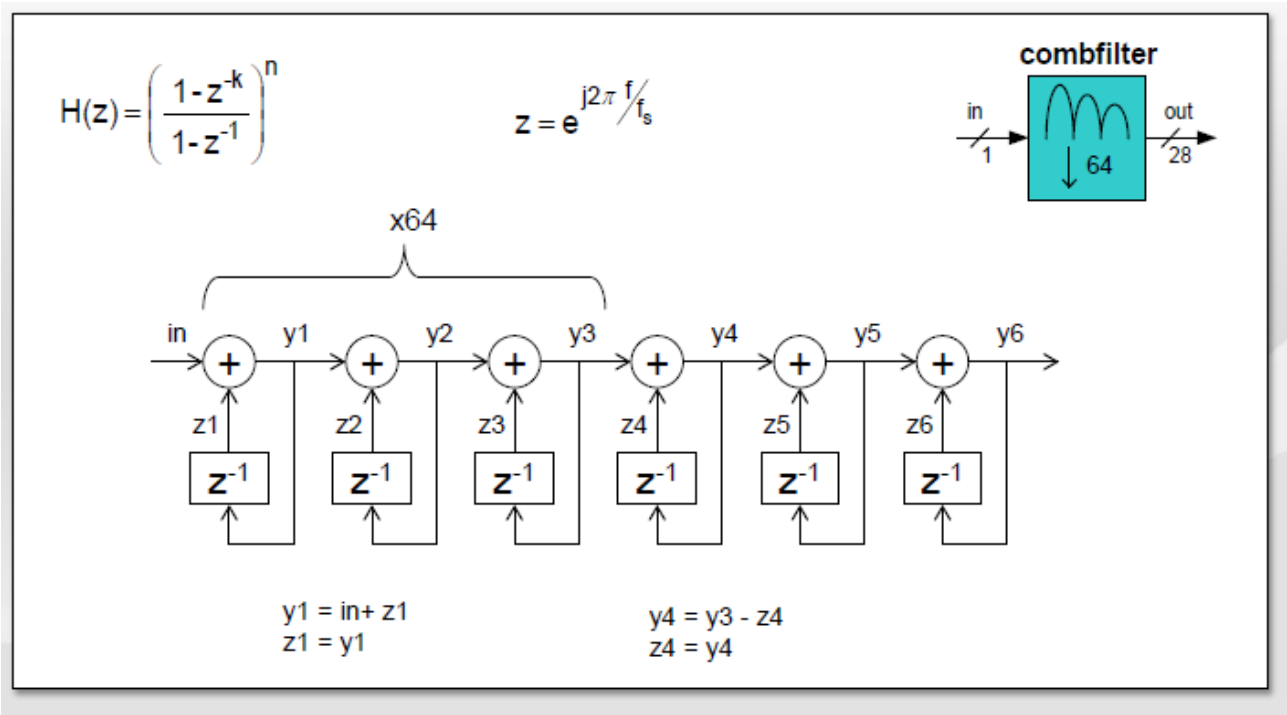


Рис. 3.27. Принцип работы гребенчатого фильтра

1. Изучите файл comb_filter.h и определите размер (в битах) данных, переносимых выходным портом.

2. В файле comb_filter.h объявите 6 переменных-членов от z1 до z6 (19-разрядные длинные целые числа).

3. В файле comb_filter.h выполните функцию set_attributes (), чтобы скорость входного входа составляла 64, а скорость выходного выхода равнялась 1. Таким образом, comb_filter потребляет 64 исходящих выборки для получения одной выходной выборки.

4. В файле comb_filter.cpp, используя принцип принципа гребенчатого фильтра, завершите функцию processing () для выполнения гребенчатой фильтрации.

5. Выполните моделирование во временной области за 10 мс и отследите сигнал sig_4.

На листингах 3.13 – 3.21 показаны файлы программы.

Листинг 3.13

Заголовочный файл SIN_SOURCE_WITH_NOISE_H

```
#ifndef SIN_SOURCE_WITH_NOISE_H
#define SIN_SOURCE_WITH_NOISE_H
```

```
    #include <systemc-ams.h>       // SystemC AMS header

    SCA_TDF_MODULE(sin_source_with_noise) // Declare a
TDF module
    {
      sca_tdf::sca_out<double> out;   // TDF output port

      void set_attributes();  // Set TDF attributes

      void processing();     // Describe time-domain
behaviour

      void ac_processing();  // Describe freq-domain
behaviour

      SCA_CTOR(sin_source_with_noise)  // Constructor of
the TDF module
      : out("out"),     // Name the port(s)
        ampl(1.0), freq(1e3),variance(0.1) {}  // Initial
values for ampl and freq

     private:      // Private variables
      double ampl;       //   amplitude
      double freq;  //   frequency
      double variance;     //   variance for noise
    };

    #endif /* SIN_SOURCE_WITH_NOISE_H */
```

Листинг 3.14

Исполняемый файл `sin_source_with_noise.cpp`

```
    #include "sin_source_with_noise.h"

    #include <cstdlib> // for std::rand
    #include <cmath>   // for  std::sin, std::sqrt, and
std::log

    double gauss_rand(double variance)
    {
      double rnd1, rnd2, Q, Q1, Q2;

      do
      {
```

```
        rnd1 = static_cast<double>(std::rand()) /
RAND_MAX;
        rnd2 = static_cast<double>(std::rand()) /
RAND_MAX;

        Q1 = 2.0 * rnd1 - 1.0;
        Q2 = 2.0 * rnd2 - 1.0;
        Q = Q1 * Q1 + Q2 * Q2;
      }
      while (Q > 1.0);

      return ( std::sqrt(variance) * ( std::sqrt( - 2.0 *
std::log(Q) / Q) * Q1) );
    }

    void sin_source_with_noise::set_attributes()    //
Set TDF attributes
    {
      out.set_timestep(1.0, SC_US);                 //
Set time step of output port
    }

    void sin_source_with_noise::processing()
    {
      double t = out.get_time().to_seconds();       //
Get current time of the sample
      double n = gauss_rand(variance);
      double x = ampl * sin(2.0 * 3.1415 * freq * t) + n;
// Calculate sine wave
      out.write(x);
// Write sample to the output
    }

    void sin_source_with_noise::ac_processing()
    {
      sca_ac(out) = 1.0;
    }
```

Листинг 3.15

Заголовочный файл `ADC_SD_H`

```
    #ifndef ADC_SD_H
    #define ADC_SD_H

    #include <systemc-ams.h>
```

```
SCA_TDF_MODULE(adc_sd)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<bool>  out;

  void processing();
  void ac_processing();

  SCA_CTOR(adc_sd);

private:
  double integ1, integ2;
};


#endif /* ADC_SD_H */
```

Листинг 3.16

Исполняемый файл ADC_SD.cpp

```
#include "adc_sd.h"

adc_sd::adc_sd(sc_core::sc_module_name nm)
{
  integ1 = 0.0;
  integ2 = 0.0;
}

void adc_sd::processing()
{
  double a = 0.5;
  double b = 0.5;
  double c = 1.0;

  double feedback = (integ2 >= 0.0) ? 1.0 : -1.0;

  bool output = (integ2 >= 0.0) ? true : false;
  out.write(output);

  double s1 = in.read() - feedback;

  double k1 = a * s1;
  double s2 = k1 + integ1;
```

```
    double k3 = c * feedback;
    double s3 = integ1 - k3;

    double k2 = b * s3;

    double s4 = k2 + integ2;

    integ2 = s2;
    integ1 = s4;
  }

  void adc_sd::ac_processing()
  {
    sca_ac(out) = 0.25 / (sca_ac_z(2) - 1.5 *
sca_ac_z() + 0.75) * sca_ac(in);
  }
```

Листинг 3.17

Заголовочный файл PREFILTER.H

```
#ifndef _PREFILTER_H
#define _PREFILTER_H

#include <systemc-ams.h>

SCA_TDF_MODULE(prefilter)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;

  void initialize();
  void processing();
  void ac_processing();

  SCA_CTOR(prefilter);

 private:
  sca_vector<double> num, den;
  sca_tdf::sca_ltf_nd ltf1;
  double fc;
};

#endif /* _PREFILTER_H */
```

Листинг 3.18

Исполняемый файл prefilter.cpp

```
#include "prefilter.h"

prefilter::prefilter(sc_core::sc_module_name nm)
{
  fc = 1e3;
}

void prefilter::initialize()
{
  num(0) = 1.0;
  den(0) = 1.0;
  den(1) = 1.0 / (2.0 * 3.1415 * fc);
}

void prefilter::processing()
{
  out = ltf1(num, den, in);
}

void prefilter::ac_processing()
{
  sca_ac(out) = sca_ac_ltf_nd(num, den, sca_ac(in));
}
```

Листинг 3.19

Заголовочный файл `comb_filter.h`

```
#ifndef COMB_FILTER_H
#define COMB_FILTER_H

#include<systemc-ams>

SCA_TDF_MODULE(comb_filter)
{
  static const int WL=19;

  sca_tdf::sca_in<bool>                 in;
  sca_tdf::sca_out<sc_dt::sc_int<WL> > out;

  void set_attributes()
  {
    in.set_rate(64);
```

```
    out.set_rate(1);
  }

  void processing();
  void ac_processing();

  SCA_CTOR(comb_filter);

private:
  sc_dt::sc_int<WL>  z1;
  sc_dt::sc_int<WL>  z2;
  sc_dt::sc_int<WL>  z3;
  sc_dt::sc_int<WL>  z4;
  sc_dt::sc_int<WL>  z5;
  sc_dt::sc_int<WL>  z6;
};

#endif /* COMB_FILTER_H */
```

Листинг 3.20

Исполняемый файл `comb_filter.cpp`

```
#include <systemc-ams.h>
#include "comb_filter.h"

comb_filter::comb_filter(sc_core::sc_module_name nm)
{
  z1 = 0;
  z2 = 0;
  z3 = 0;
  z4 = 0;
  z5 = 0;
  z6 = 0;
}


void comb_filter::processing()
{
  sc_int<WL> y, y1, y2, y3, y4, y5;

  // Integrators
  for (int i = 0; i < 64; i++)
  {
    y1 = in.read(i) ? z1 + 1 : z1 - 1;
    z1 = y1;
```

```
        y2 = y1 + z2;
        z2 = y2;

        y3 = y2 + z3;
        z3 = y3;
      }

      //Differentiators
      y4 = y3 - z4;
      z4 = y3;

      y5 = y4 - z5;
      z5 = y4;

      y = y5 - z6;
      z6 = y5;

      out.write(y);
    }


    void comb_filter::ac_processing()
    {
        double k = 64.0;
        double n = 3.0;

        sca_complex z = sca_ac_z(1, in.get_timestep());
        sca_complex h;

        h = pow((1.0 - pow(z, -k)) / (1.0 - 1.0 / z),
n); // comb

        sca_ac(out) = h * sca_ac(in);

    }
```

Листинг 3.21

Исполняемый файл `tb_lab2d.cpp`

```
    #include "sin_source_with_noise.h"
    #include "prefilter.h"
    #include "adc_sd.h"
```

```
#include "comb_filter.h"
#include "systemc-ams.h"

int sc_main(int argn,char* argc[])
{
  sca_tdf::sca_signal<double> sig_1, sig_2;
  sca_tdf::sca_signal<bool> sig_3;
  sca_tdf::sca_signal<sc_int<19> > sig_4;

  sin_source_with_noise sin1("sin1");
  sin1.out(sig_1);

  prefilter prefi1("lp1");
  prefi1.in(sig_1);
  prefi1.out(sig_2);

  adc_sd adc1("adc1");
  adc1.in(sig_2);
  adc1.out(sig_3);

  comb_filter cbf1("cb1");
  cbf1.in(sig_3);
  cbf1.out(sig_4);

  sca_trace_file* tfp =
    sca_create_vcd_trace_file("tb_lab2d.dat");

  sca_trace(tfp, sig_1, "sig_1");
  sca_trace(tfp, sig_2, "sig_2");
  sca_trace(tfp, sig_3, "sig_3");
  sca_trace(tfp, sig_4, "sig_4");

  sc_start(5.0, SC_MS);

  tfp->reopen("tb_ac_lab2d.dat");
  tfp-
>set_mode(sca_ac_format(sca_util::SCA_AC_DB_DEG));

  sca_ac_start(1.0, 1e6, 1000, SCA_LOG);
  sca_close_vcd_trace_file(tfp);         // Close
trace file

  return 0;
}
```

Результат компиляции программы показан на рис. 3.28.

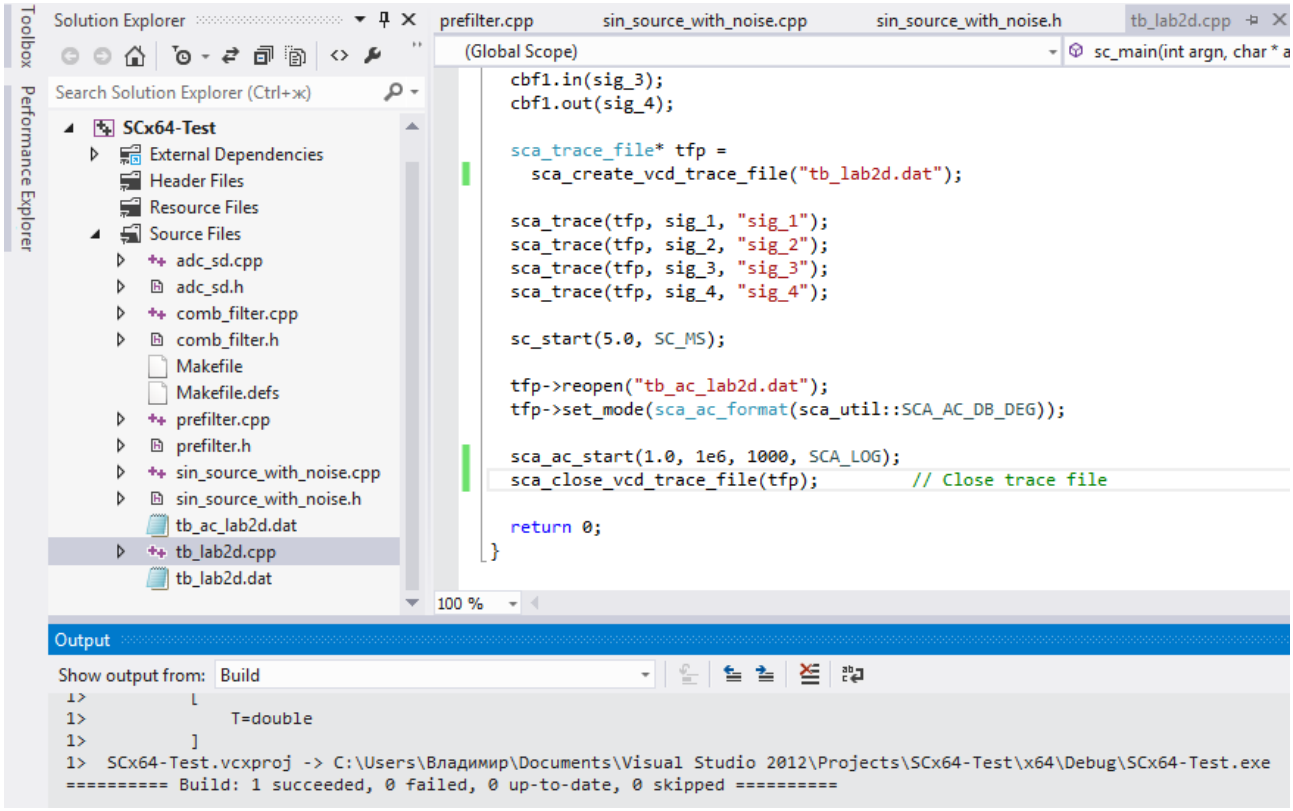
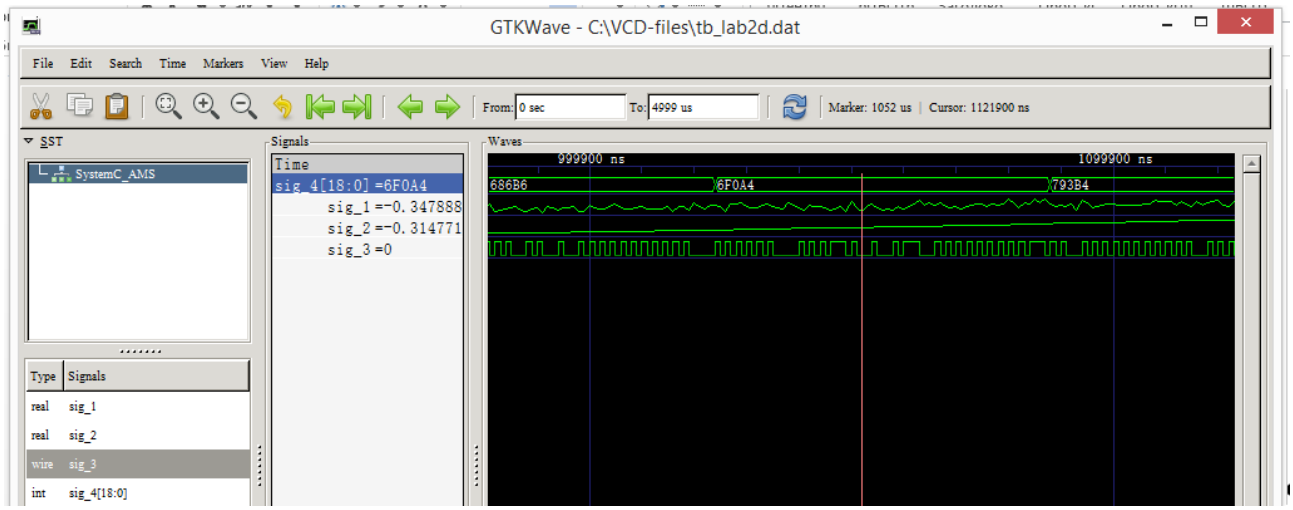


Рис. 3.28. Результат компиляции программы

Результаты моделирования для сигналов sig_4 показаны на рис. 3.29 в формате HEX.



HEX

Рис. 3.29. Сигналы в контрольных точках стенда.

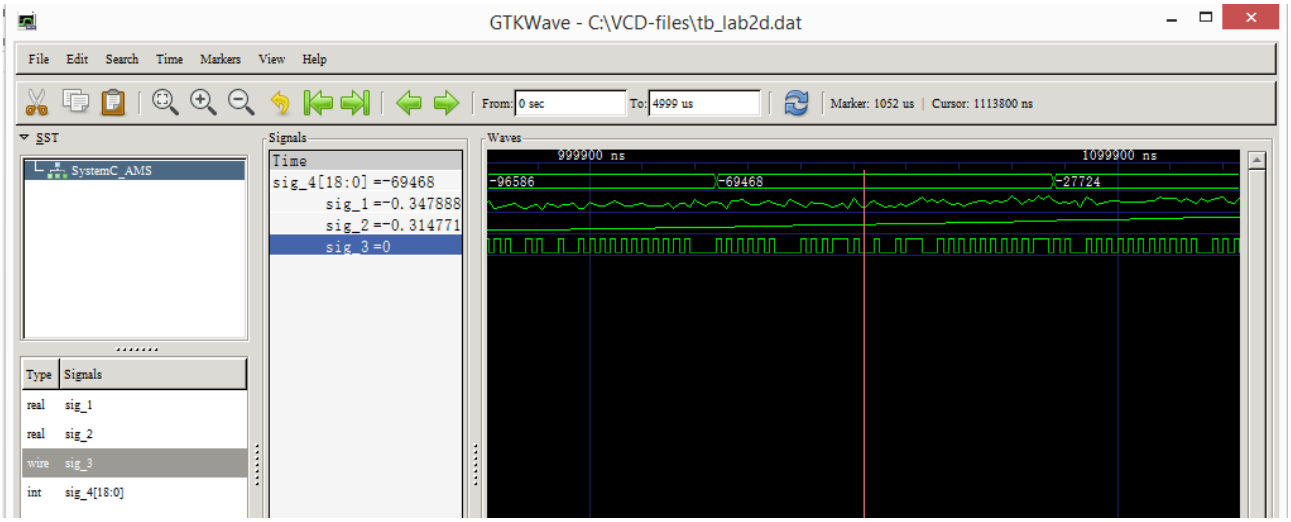


Рис. 3.30. Сигналы в контрольных точках стенда.

Для сигнала sig_4 использован децимальный знаковый формат.

Повторим моделирование для трассировки в формате tabular. Фрагмент результатов показан на рис. 3.31.

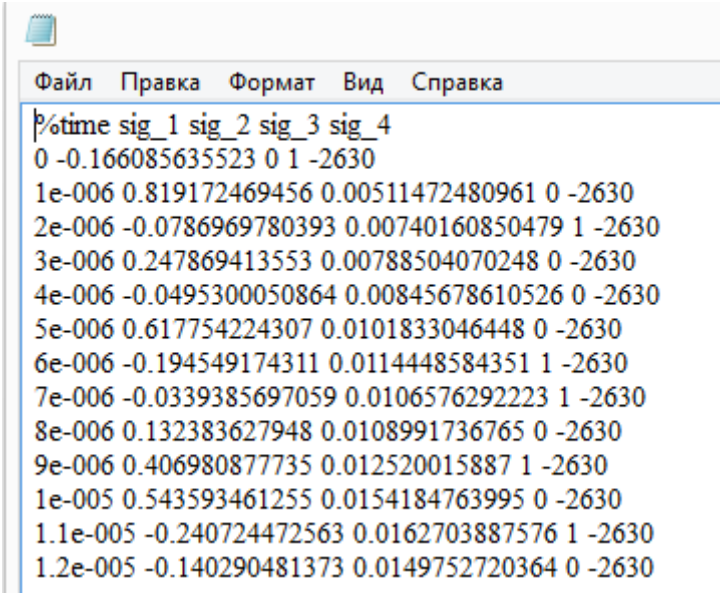
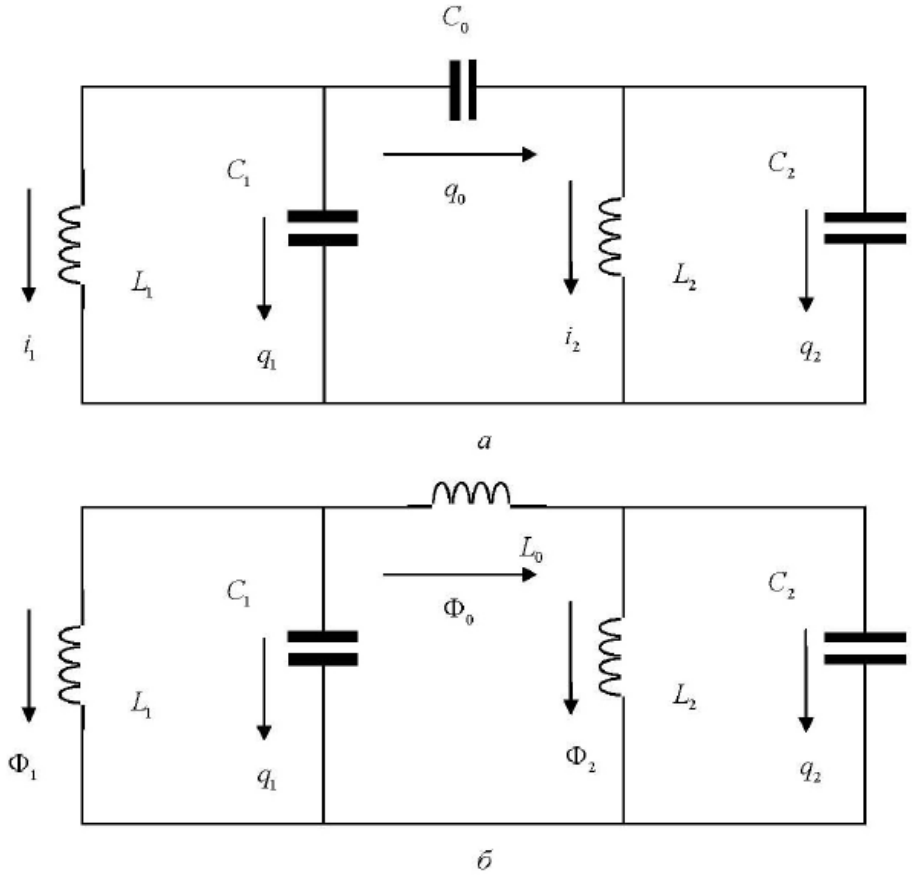


Рис. 3.31. Фрагмент результатов моделирования из папки tb_lab2d.dat.

4. Моделирование электрических линейных сетей

4.1. Основы моделирования

Модель вычисления Electric Linear Networks вводит использование электрических примитивов и их взаимосвязи, чтобы моделировать консервативное, непрерывное поведение. Стиль моделирования ELN позволяет создание электрических примитивов, которые могут быть соединены вместе с помощью электрических узлов, чтобы сформировать электрическая сеть. Математические отношения между электрическими примитивами определены в каждом узле сети, где используются как потенциальные (напряжение) и потоковые (ток) величины в соответствии с законами напряжения Кирхгофа (KVL) и закону Кирхгофа для токов (KCL). Таким образом, электрическая сеть представляет собой набор дифференциальных алгебраических уравнений, которые будут решены во время моделирования в определить фактическое поведение цепи.



Схемы трехконтурной консервативной ЭЦ1 (*а*) и ЭЦ2 (*б*), образованных двумя последовательно связанными через емкость C_0 (*а*) и индуктивность L_0 (*б*) параллельными контурами

На рисунке 4.1 показан пример электрической сети с двумя резисторами, конденсатором и источником тока.

Такая сеть называется моделью ELN и состоит из набора связанных примитивных модулей ELN, которые образуют вместе систему уравнений ELN или кластер. Каждый примитивный модуль ELN может иметь один или несколько ELN терминалов. Примитивные модули ELN соединяются через

свои терминалы с использованием узлов ELN.

Опорный или заземляющий узел, который всегда имеет нулевое напряжение, называется опорным узлом ELN (*ELN reference node*). ELN терминалы также используются в качестве интерфейса для соединения модели ELN с другими моделями ELN.

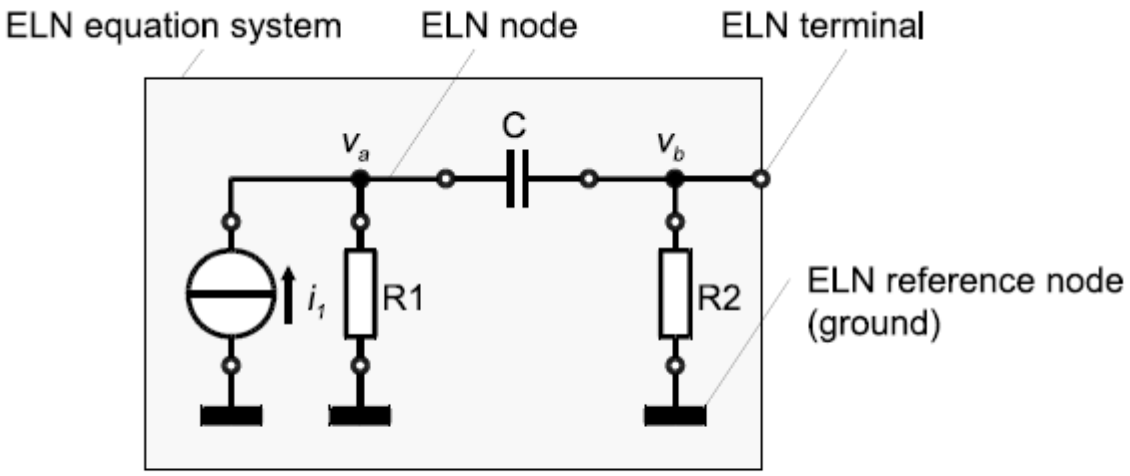


Figure 4.1. Example of a basic ELN model representing an electrical network

Рисунок 4.1. Пример базовой модели ELN, представляющей электрическую сеть

4.1.1. Настройка системы уравнений

Расширения SystemC AMS предлагают конечный набор примитивных модулей ELN, таких как источники (напряжение или тока), линейные сосредоточенные элементы (резисторы, конденсаторы, катушки индуктивности), линейные распределенные элементы (линии передачи), идеальный усилитель (nullor), идеальный трансформатор, линейный гиратор и идеальные переключатели. Также как в стиле моделирования LSF, модели ELN могут быть составлены только из этих примитивов, так как нет возможности реализовать определяемые пользователем электрические примитивы. На Рисунке 4.2 показаны некоторые элементы ELN с сосредоточенными параметрами и их соответствующие математические уравнения.

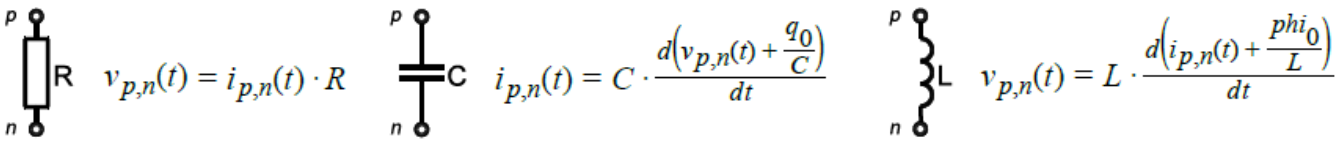


Figure 4.2. Examples of the basic ELN lumped elements: resistor (R), capacitor (C), and inductor (L) with their corresponding mathematical equations

Рисунок 4.2. Примеры основных ELN сосредоточенных элементов: резистор

(R), конденсатор (C) и индуктор (L) с соответствующими им математическими уравнениями

При создании модели ELN (электрической сети), математические уравнения для каждого примитива и их отношения, определённые в каждом узле, будут использованы для составления общей системы уравнений. Например, модель ELN, представленная на рисунке 4.1, приведет к созданию системы уравнений ELN для узлов A и B следующим образом по законам тока и напряжения Кирхгофа, а также с использованием заданных уравнений каждого примитива, как показано на рисунке 4.2.

$$-i_1+\frac{v_a}{R1}+C\cdot\frac{d\left(v_{a,b}+\frac{q_0}{C}\right)}{dt}=0 \qquad \frac{v_b}{R2}-C\cdot\frac{d\left(v_{a,b}+\frac{q_0}{C}\right)}{dt}=0$$

Обратите внимание, что ток через примитивы ELN с двумя терминалами определяется как ток, протекающий от терминал *p* до терминала *n*. Это также относится к источникам тока.

4.1.2. Назначение и распространение временных шагов

Как и для модуля TDF, временной шаг может быть назначен непосредственно модулю ELN или может быть назначен автоматически используя механизм распространения временного шага в системе уравнений ELN. В случае, если модель ELN подключена к модели TDF, временной шаг от подключенного порта (портов) TDF распространяется на модель ELN. Согласованность между локально определёнными временными шагами модуля ELN и распространенными временными шагами является существенной. В противном случае моменты времени для решения системы уравнений ELN или связи с моделью подключённого TDF не может быть определена должным образом (см. также раздел 2.1.3). Шаг по времени должен быть определён по крайней мере, в одном месте во всей системе.

Во время моделирования эта система уравнений ELN решается численно с соответствующими временными шагами, которые могут быть меньше назначенного временного шага. Решатель по крайней мере предоставит результаты в рассчитанные моменты времени от назначенных временных шагов.

4.2. Языковые конструкции

4.2.1. Модули ELN

Модуль ELN - это предопределённый электрический примитив, который можно использовать для построения электрической сети.

Доступные предопределённые примитивные модули ELN перечислены в таблице 4.1 ниже. Приложение A даёт подробную информацию для каждого модуля ELN.

ELN module name	Description
sca_eln::sca_r	Resistor Резистор
sca_eln::sca_c	Capacitor Емкость
sca_eln::sca_l	Inductor Индуктивность
sca_eln::sca_vcvs	Voltage controlled voltage source Источник напряжения, управляемый напряжением
sca_eln::sca_vccs	Voltage controlled current source Источник тока, управляемый напряжением
sca_eln::sca_ccvs	Current controlled voltage source Источник напряжения, управляемый током
>sca_eln::sca_cccs	Current controlled current source Источник тока, управляемый током
sca_eln::sca_nullor	Nullor (nullator - norator pair), ideal op-amp Идеальный операционный усилитель, нуллатор
sca_eln::sca_ideal_transformer	Ideal transformer Идеальный трансформатор
sca_eln::sca_transmission_line	Transmission line Линия передачи
sca_eln::sca_vsource	Independent voltage source - Независимый источник напряжения
sca_eln::sca_isource	Independent current source Независимый источник тока
sca_eln::sca_tdf::sca_r, **sca_eln::sca_tdf_r**	Variable resistor controlled by a TDF input signal - Переменный резистор, управляемый входным TDF сигналом
sca_eln::sca_tdf::sca_c, **sca_eln::sca_tdf_c**	Variable capacitor controlled by a TDF input signal – Переменная емкость, управляемая входным TDF сигналом
sca_eln::sca_tdf::sca_l, **sca_eln::sca_tdf_l**	Variable inductor controlled by a TDF input signal – Переменная индуктивность, управляемая входным TDF сигналом.
sca_eln::sca_tdf::sca_rswitch, **sca_eln::sca_tdf_rswitch**	Switch controlled by a TDF input signal – Переключатель, управляемый входным TDF сигналом.
sca_eln::sca_tdf::sca_vsource, **sca_eln::sca_tdf_vsource**	Voltage source driven by a TDF input signal – Источник напряжения, управляемый входным TDF сигналом.
sca_eln::sca_tdf::sca_isource, **sca_eln::sca_tdf_isource**	Current source driven by a TDF input signal - Источник тока, управляемый входным TDF сигналом.
sca_eln::sca_tdf::sca_vsink, **sca_eln::sca_tdf_vsink**	Converts voltage to a TDF output signal - Преобразует напряжение в выходной сигнал TDF
sca_eln::sca_tdf::sca_isink, **sca_eln::sca_tdf_isink**	Converts current to a TDF output signal - Преобразует ток в выходной сигнал TDF
sca_eln::sca_de::sca_r, **sca_eln::sca_de_r**	Variable resistor controlled by a discrete-event input signal - Переменный резистор, управляемый входным сигналом дискретного события
sca_eln::sca_de::sca_c, **sca_eln::sca_de_c**	Variable capacitor controlled by a discrete-event input signal - Переменная емкость, управляемая входным сигналом дискретного события
sca_eln::sca_de::sca_l, **sca_eln::sca_de_l**	Variable inductor controlled by a discrete-event input signal - Переменная индуктивность, управляемая входным сигналом дискретного события
sca_eln::sca_de::sca_rswitch, **sca_eln::sca_de_rswitch**	Switch controlled by a discrete-event input signal – Переключатель, управляемый входным сигналом дискретного события
sca_eln::sca_de::sca_vsource, **sca_eln::sca_de_vsource**	Voltage source driven by a discrete-event input signal – Источник напряжения, управляемый входным сигналом дискретного события

sca_eln::sca_de::sca_isource, **sca_eln::sca_de_isource**	Current source driven by a discrete-event input signal – Источник тока, управляемый входным сигналом дискретного события
sca_eln::sca_de::sca_vsink, **sca_eln::sca_de_vsink**	Converts voltage to a discrete-event output signal - Преобразует напряжение в выходной сигнал дискретного события
sca_eln::sca_de::sca_isink, **sca_eln::sca_de_isink**	Converts current to a discrete-event output signal - Преобразует ток в выходной сигнал дискретного события

Шаг по времени модуля

Чтобы решить систему уравнений ELN, шаг времени должен быть связан с набором подключенных модулей ELN как часть этапа разработки. Это можно сделать с помощью функции-члена модуля ELN set_timestep. В качестве альтернативы модель ELN может опираться на механизм распространения шага по времени, который проходит как шаг по времени от модуля к модулю через его порты в моделях вычислений TDF, LSF и ELN.

Таким образом, в случаях, когда модель ELN подключена к модели TDF, временной шаг от подключенного порта, если это доступно, распространяется на модель ELN. В случае, если распространяются временные шаги и пользовательские временные шаги, то при этом соответствие между этими временными шагами является обязательным, как описано в разделе 2.1.3.

Шаг по времени модуля может быть назначен путём вызова функции-члена set_timestep экземпляра объекта в конструкторе родительского модуля и передачей *double* (двойное значение) и единицы времени или объекта типа sca_core :: sca_time, как показано в следующем примере:

```
SC_MODULE(my_eln_source)
{
// terminal declaration
sca_eln::sca_terminal p;
// child module declaration
sca_eln::sca_vsource v_src;
SC_CTOR(my_eln_source)
: p("p"),
v_src("v_src", 0.0, 0.0, 1.0e-3, 1.0e3), // 1 kHz
sinusoidal source with an amplitude of 1 mV
gnd("gnd")
{
v_src.set_timestep(0.25, sc_core::SC_MS); // set
module timestep to 0.25 ms
v_src.p(p);
v_src.n(gnd);
}
private:
sca_eln::sca_node_ref gnd;
};
```

4.2.2. ELN терминалы

Терминал ELN - это объект, который можно использовать для соединения нескольких моделей ELN с использованием узлов ELN, которые связаны с этим терминалом. Из-за консервативного характера формализма моделирования ELN, терминал ELN не определён как порт ввода или вывода. Вместо этого эти терминалы используются для соединения с узлами класса sca_eln :: sca_node или sca_eln :: sca_node_ref (см. раздел 4.2.3). Так как ELN клеммы всегда используются в структурном (родительском) модуле, их также можно использовать для подключения к дочернему элементу ELN модуля напрямую, следуя правилу привязки порт-порт (см. раздел 4.3.1). Терминалы ELN используют внутренний тип данных, также называемый электрической природой, который предотвращает использование пользовательских типов данных.

В приведённом ниже примере показано, как терминалы ELN используются в структурной модели ELN.

```
SC_MODULE(my_eln_model)
{
// terminal declarations
sca_eln::sca_terminal p;   //1
sca_eln::sca_terminal n;
SC_CTOR(my_eln_model) : p("p"), n("n")  //2
{
// model implementation here
}
};
```

1. ELN положительный (p) и отрицательный (n) терминал, который несёт сигнал непрерывного времени и - значения.

2. Использование списка инициализации конструктора для присвоения имён «p» и «n» терминалам p и n, соответственно.

Для подключения модулей ELN к TDF или области дискретных событий доступны специализированные модули преобразователей.

Это объясняется в разделе 4.4. Терминалы ELN не предоставляют методы доступа для чтения или записи.

4.2.3. Узлы ELN

Узлы ELN используются для соединения примитивных модулей ELN. В этом случае несколько общих примитивов ELN делят тот же узел (также называемый net). Существует два класса узлов ELN:

• Узел ELN класса sca_eln :: sca_node.

• Опорный узел ELN (земля) класса sca_eln :: sca_node_ref.

Узлы ELN и опорные узлы используются для настройки всей системы уравнений. Пример ниже показывает, как использовать узлы ELN и эталонные

узлы ELN.

```
    // node declarations
    sca_eln::sca_node net1; // ELN node (called "net1")
    sca_eln::sca_node_ref gnd; // ELN reference node
(called ground, "gnd")
```

Как и в SystemC, список инициализации конструктора родительского модуля может использоваться для назначения определённого пользователем имени узла:

```
    // using the constructor initialization-list to
assign the names to the declared ELN nodes
    SC_CTOR(my_eln_module) : net1("net1"), gnd("gnd") {}
```

Раздел 4.3 будет описывать создание структурных моделей ELN и покажет примеры назначения пользовательских имена терминалам и узлам.

4.3. Моделирование непрерывного поведения

Модели ELN могут использоваться для реализации линейного динамического, непрерывного, консервативного поведения. ELN модели могут быть составлены только с использованием примитивных модулей ELN. Поэтому модель ELN всегда является структурной моделью.

4.3.1. Структурная композиция модулей ELN

Модули ELN должны создаваться как дочерние модули внутри созданного обычного родительского модуля SystemC с помощью макроса SC_MODULE или путём публичного получения из sc_core :: sc_module. Этот родительский модуль также создаёт все необходимые терминалы для связи с внешним миром и внутренними узлами для взаимосвязи дочерних модулей. Параметризация созданных экземпляров модулей, а также соединение модулей должно быть сделано в конструкторе (например, создан с помощью макроса SC_CTOR) родительского модуля SystemC.

Порт (терминал) привязки

Для правильного подключения модулей ELN к другим модулям и узлам ELN необходимо выполнить возможные привязки, как показано на рисунке 4.3. Правила привязки порта совместимы и дополняются к правилам SystemC.

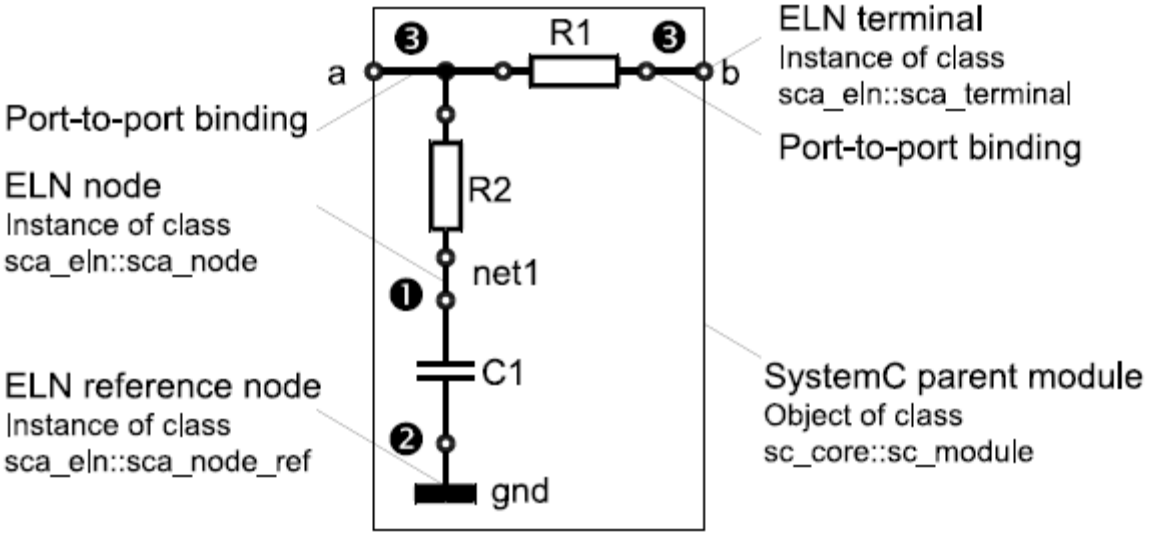


Figure 4.3. Port binding rules for ELN terminals

Рисунок 4.3. Правила привязки портов для терминалов ELN

1. Связывание терминала ELN с узлом ELN.
2. Связывание терминала ELN с эталонным узлом ELN.
3. Связывание терминала ELN с терминалом ELN родительского модуля (привязка порт-порт).

Кроме того, ELN терминал должен быть связан с точно одним узлом или опорным узлом ELN на протяжении всей иерархия. Узел ELN или эталонный узел ELN должны быть связаны с одним или несколькими терминалами ELN по всей иерархии.

Примитивные модули ELN, которые имеют порты для подключения к TDF или сигналам или портам дискретных событий, должны следовать правилам привязки портов соответствующих моделей вычислений.

Пример ниже показывает реализацию структурного состава рисунка 4.3.

```
    SC_MODULE(my_structural_eln_model)
    {
    sca_eln::sca_terminal a;    //1
    sca_eln::sca_terminal b;
    sca_eln::sca_r r1, r2;    //2
    sca_eln::sca_c c1;
    SC_CTOR(my_structural_eln_model)
    : a("a"), b("b"), r1("r1", 10e3), r2("r2", 100.0),
c1("c1", 100e-6), net1("net1"), gnd("gnd")      //3
    {
    r1.p(a);    //4
```

```
r1.n(b);
r2.p(a);
r2.n(net1);
c1.p(net1);
c1.n(gnd);
}
private:
sca_eln::sca_node net1;//5
sca_eln::sca_node_ref gnd;
};
```

1. Терминалы ELN, объявленные внутри этого модуля класса sc_core :: sc_module, становятся частью структурного состава.

2. Примитивные модули ELN объявляются в родительском модуле как дочерние модули.

3. Список инициализации в конструкторе родительского модуля распространяет необходимую конфигурацию на параметры для терминалов ELN, узлов ELN и дочерних модулей.

4. Привязка порта (терминала) осуществляется внутри конструктора родительского модуля.

5. Внутренние узлы ELN используются для подключения терминалов ELN и дочерних модулей. Эти узлы объявлены в приватном пространстве, так как они не должны быть доступны извне модуля.

4.3.2. Непрерывное моделирование

В приведенном ниже примере показан фильтр нижних частот первого порядка, основанный на той же передаточной функции Лапласа, что и описано в разделе 2.3.2. Частота среза фильтра определяется постоянной времени τ фильтра, которая является произведением значения сопротивления и емкости:

$$f_c = \frac{1}{2\pi\tau} = \frac{1}{2\pi RC}$$

Схема реализации этого фильтра довольно проста, как показано на рисунке 4.4.

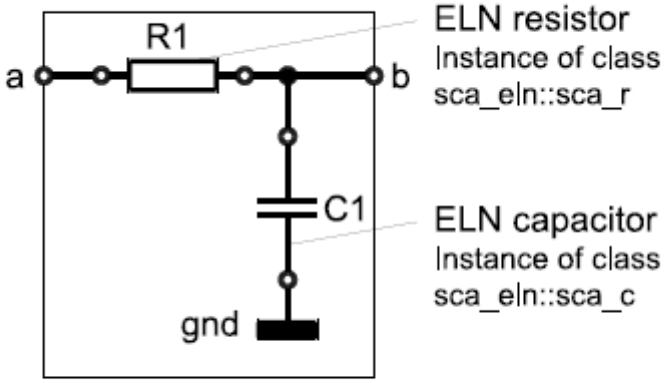


Figure 4.4. ELN circuit implementation of a first-order low-pass filter

Рисунок 4.4. Реализация схемы ELN фильтра нижних частот первого порядка

Реализация кода для фильтра нижних частот первого порядка, реализованного в виде RC-сети, приведена ниже:

```
    SC_MODULE(my_eln_filter)
    {
    sca_eln::sca_terminal a;
    sca_eln::sca_terminal b;
    sca_eln::sca_r r1;
    sca_eln::sca_c c1;
    my_eln_filter( sc_core::sc_module_name, double
r1_value, double c1_value )
    : a("a"), b("b"), r1("r1", r1_value), c1("c1",
c1_value), gnd("gnd"),
    {
    r1.n(a);
    r1.p(b);
    c1.n(b);
    c1.p(gnd);
    }
    private:
    sca_eln::sca_node_ref gnd;
    };
```

Обратите внимание, что временной шаг для этой сети не был определён в этом модуле ELN. Это означает, что это модель основана на механизме распространения шага по времени.

4.4. Взаимодействие между ELN и моделями дискретных событий или TDF

Модель вычисления ELN установит и решит систему уравнений для симуляции смоделированного поведения непрерывного времени, основанного на базовом наборе примитивных модулей ELN, описанном в разделе 4.2.1. Любое «внешнее» входное значение, например, из сигнала дискретного события или выборки TDF, должно быть введено в систему уравнений через один из этих примитивных модулей ELN. Поэтому используют специализированные примитивные модули ELN с портами для доступа к моделям вычислений в области дискретных событий и TDF, которые называются модулями преобразования.

Основное назначение этих модулей - создать интерфейс для преобразования и передачи данных из одной модели вычислений к другой.

4.4.1. Чтение и запись в модели дискретных событий

Для соединения моделей ELN с моделями с дискретными событиями модули преобразователя ELN с внутренним портом должены использоваться порт класса sc_core :: sc_in или sc_core :: sc_out.

На рисунке 4.5 показаны примитивные модули ELN sca_eln :: sca_de :: sca_vsource и sca_eln :: sca_de :: sca_isource, которые считывают сигнал дискретного события, представляющий реальное значение и преобразующий это значение для электрического напряжения или тока соответственно. В этом примере шаг модуля в 1 мс равен назначенному для модуля преобразователя ELN. Модель ELN непрерывно считывает значения с входа на временных точках, которые рассчитываются по назначенным временным шагам. Входное значение считается постоянным до следующего прочитанного значения. Входные значения интерпретируются для формирования непрерывного сигнала, который сделан доступном на выходе модуля преобразователя (считывает входные отсчёты, показанные в виде точечного сигнала). Рисунок 4.5.

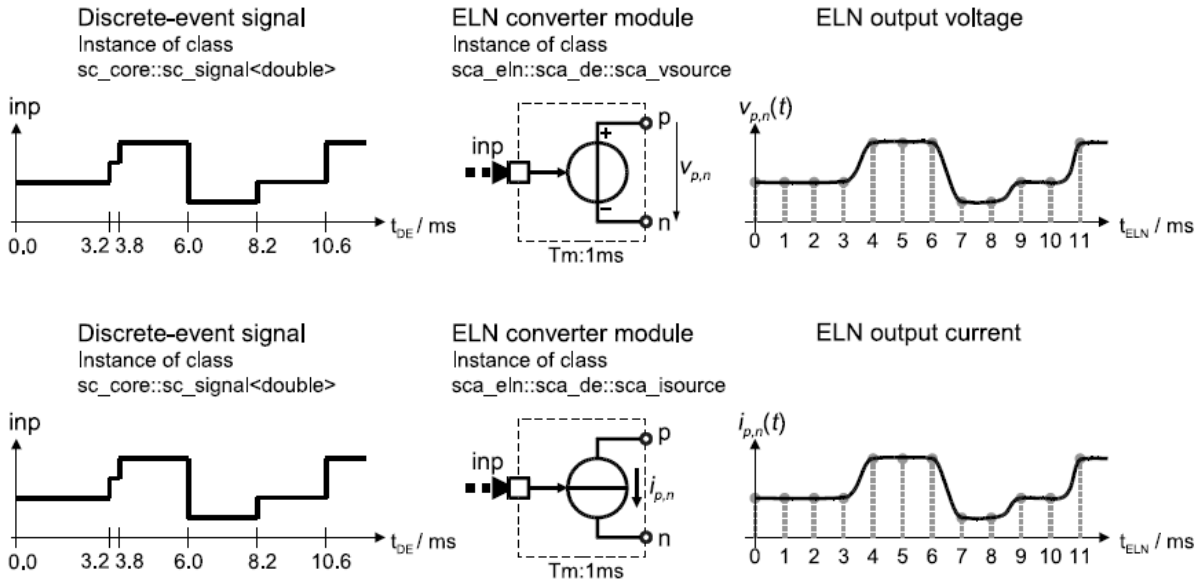


Figure 4.5. ELN converter modules reading double values from a discrete-event input signal and converting them to a continuous-time electrical voltage or current

Рисунок 4.5. Преобразовательные модули ELN, считывающие двойные значения (double) из дискретного события входного сигнала и преобразующие их в постоянное электрическое напряжение или ток

На рисунке 4.6 показаны примитивные модули ELN sca_eln :: sca_de :: sca_vsink и sca_eln :: sca_de :: sca_isink, которые преобразуют электрическое напряжение или ток в реальное значение, сигнал дискретного события. Значения на выходе порта записывается в моменты времени, рассчитанные от назначенного модулю временного шага в 1 мс.

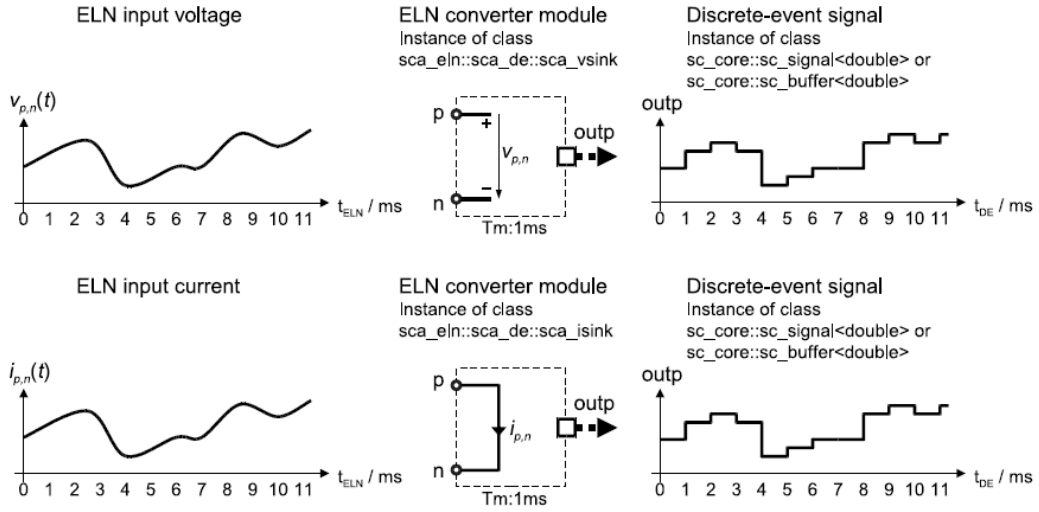


Figure 4.6. ELN converter modules to convert an electrical voltage or current to a real value, discrete-event output signal

Рисунок 4.6. Модули ELN преобразователей для преобразования электрического напряжение или тока в действительные значения, выходного сигнала дискретного события

4.4.2. Чтение и запись в модели TDF

Аналогичным образом, модели ELN могут быть подключены к моделям TDF с помощью модулей преобразователя с внутренним портом класса sca_tdf :: sca_in или sca_tdf :: sca_out.

На рисунке 4.7 показаны примитивные модули ELN sca_eln :: sca_tdf :: sca_vsource и sca_eln :: sca_tdf :: sca_isource, которые считывают значение из сигнала TDF и преобразует это значение в электрические напряжение или ток соответственно. В этом примере шаг времени модуля 1 мс назначен преобразователю ELN модуля. Модель ELN непрерывно считывает сэмплы с входа TDF. Входные образцы интерпретируется для формирования непрерывного сигнала, который становится доступным на выходе модуля преобразователя (входные выборки показаны в виде точечного сигнала).

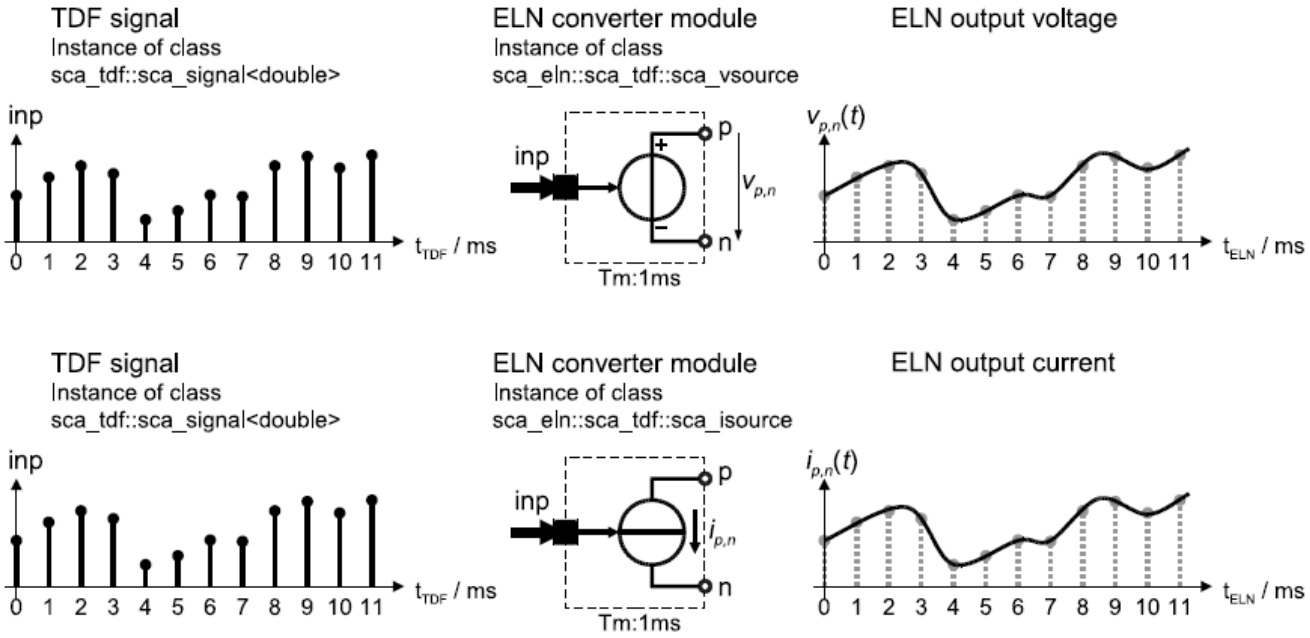


Figure 4.7. ELN converter modules reading double values from a TDF input signal and converting them to a continuous-time electrical voltage or current

Рисунок 4.7. Преобразовательные модули ELN, считывающие двойные значения со входа TDF сигнал и преобразующие их в постоянное электрическое напряжение или ток

На рисунке 4.8 показаны примитивные модули ELN sca_eln :: sca_tdf :: sca_vsink и sca_eln :: sca_tdf :: sca_isink, которые преобразует электрическое напряжение или ток в сигнал TDF выборок на выходном порту и записывают их в рассчитанные моменты времени, которые соответствуют назначенному модулю шагу по времени 1 мс

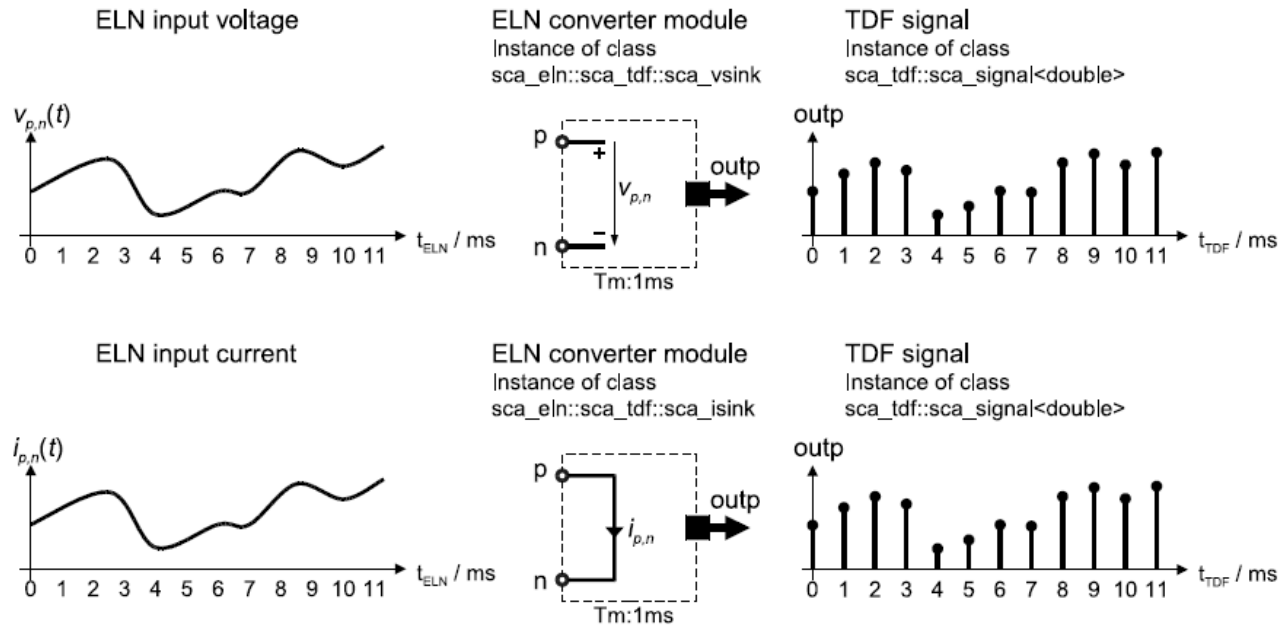


Figure 4.8. ELN converter modules convert an electrical voltage or current to a TDF output signal

Рисунок 4.8. Преобразовательные модули ELN преобразуют электрическое напряжение или ток в выходной сигнал TDF

4.4.3. Инкапсуляция модели ELN

Модули преобразователя, описанные в предыдущих разделах, могут использоваться для инкапсуляции модели ELN в другой модели вычисления. На рисунке 4.9 показан пример использования модулей преобразователя для из модели вычислений TDF для инкапсуляции поведения ELN. В этом случае доступ к и от ELN система уравнений использует сигналы с дискретным временем в соответствии с семантикой TDF, тогда как внутренние сигналы ELN и вычисления непрерывны.

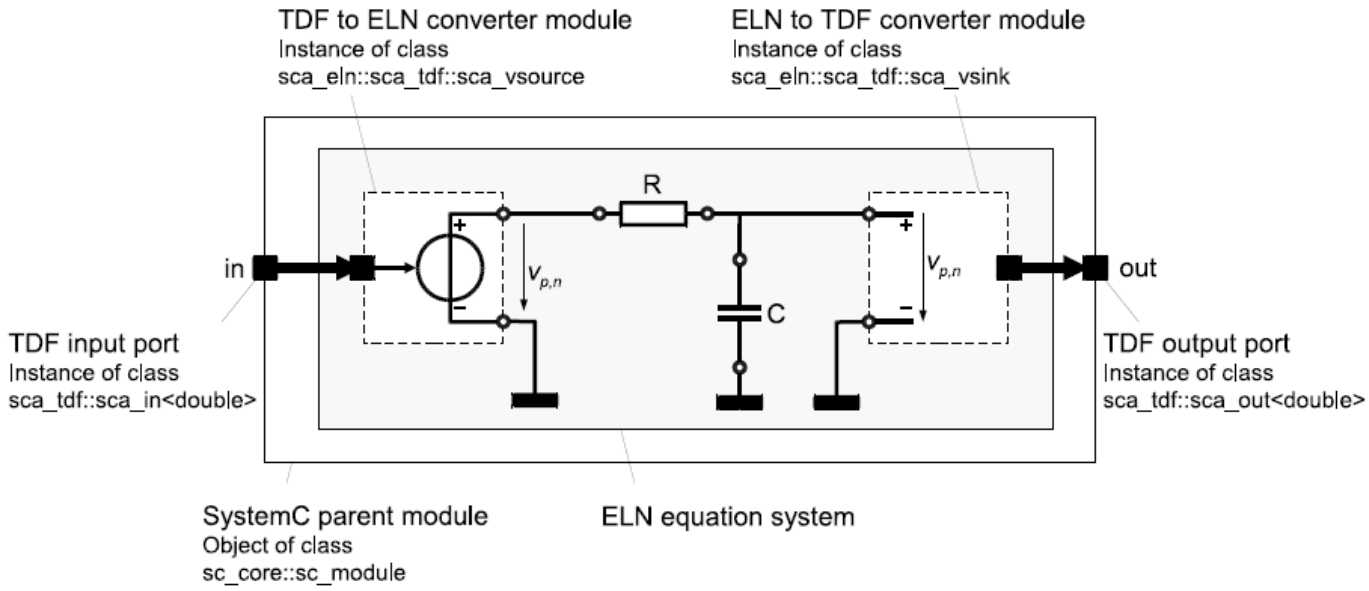


Figure 4.9. ELN equation system encapsulated for inclusion into a structural TDF model description by using converter modules

Рисунок 4.9. Система уравнений ELN, инкапсулированная для включения в описание структурной модели TDF с использованием конвертерных модулей

Пример ниже показывает реализацию рисунка 4.9

```
    SC_MODULE(eln_in_tdf)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    sca_eln::sca_tdf::sca_vsource vin;
    sca_eln::sca_tdf::sca_vsink vout;
    sca_eln::sca_r r;
    sca_eln::sca_c c;
    eln_in_tdf( sc_core::sc_module_name, double r_val,
double c_val )
    : in("in"), out("out"), vin("vin"), vout("vout"),
r("r", r_val), c("c", c_val),
    n1("n1"), n2("n2"), gnd("gnd")
    {
    vin.inp(in);
    vin.p(n1);
    vin.n(gnd);
    r.p(n1);
    r.n(n2);
```

```
c.p(n2);
c.n(gnd);
vout.p(n2);
vout.n(gnd);
vout.outp(out);
}
private:
sca_eln::sca_node n1, n2;
sca_eln::sca_node_ref gnd;
};
```

Аналогичный подход можно использовать для инкапсуляции модели ELN для включения в структурное описание модели дискретного события с использованием модулей преобразования в и из области дискретных событий, как описано в разделе 4.4.1.

4.5. Семантика исполнения ELN

В дополнение к этапам разработки и моделирования, как это определено в стандарте языка SystemC IEEE 1666-2005, специфическая функциональность реализована для разработки и исполнения моделей ELN. Эти дополнения аналогичны тем, которые есть в LSF.

Как показано на рисунке 4.10, этап разработки включает в себя следующие этапы:

• Расчет и распространение временного шага ELN: определите временной шаг и проверьте согласованность внутри каждой ELN модели (см. также раздел 4.1.2).

• Настройка уравнения ELN и проверка разрешимости: составьте систему уравнений ELN из уравнений, предоставляемых предопределенными примитивными модулями ELN, и их взаимосвязь, определяемая композицией. Проверьте, может ли полученная система уравнений быть решена.

Шаги для этапа моделирования:

• Инициализация ELN: сначала установите все сигналы ELN на ноль, а затем установите начальные условия системы на основе на потенциально определенных начальных условиях примитивов ELN.

• Моделирование во временной области ELN: система уравнений ELN решается численно с использованием подходящих шагов времени, которые могут быть меньше назначенного временного шага. Решатель, по крайней мере, даст результаты на временные точки, рассчитанные по заданному временному шагу.

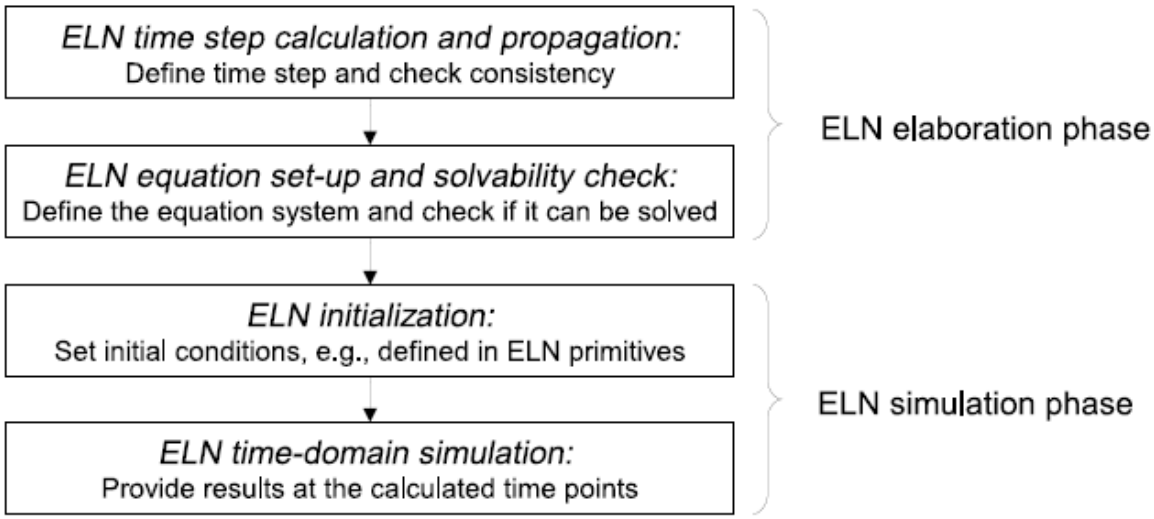


Figure 4.10. ELN elaboration and simulation phases

Рисунок 4.10. Этапы разработки и моделирования ELN

Этап разработки и моделирования выполняется путём запуска моделирования во временной области с использованием функции sc_core :: sc_start. Это объясняется в разделе 6.1.1.

4.6. Примеры применения

В этом разделе показаны некоторые основные примеры применения с использованием моделирования ELN.

4.6.1. POTS-интерфейс

Внешний вид простой старой телефонной системы (POTS) показан на рисунке 4.11. Он состоит из телефона, линия передачи, схема защиты и схема интерфейса абонентской линии (SLIC), которая может быть смоделирована естественно, используя ELN примитивы. Интерфейс от и до внешнего интерфейса POTS основан на TDF или дискретном сигнале.

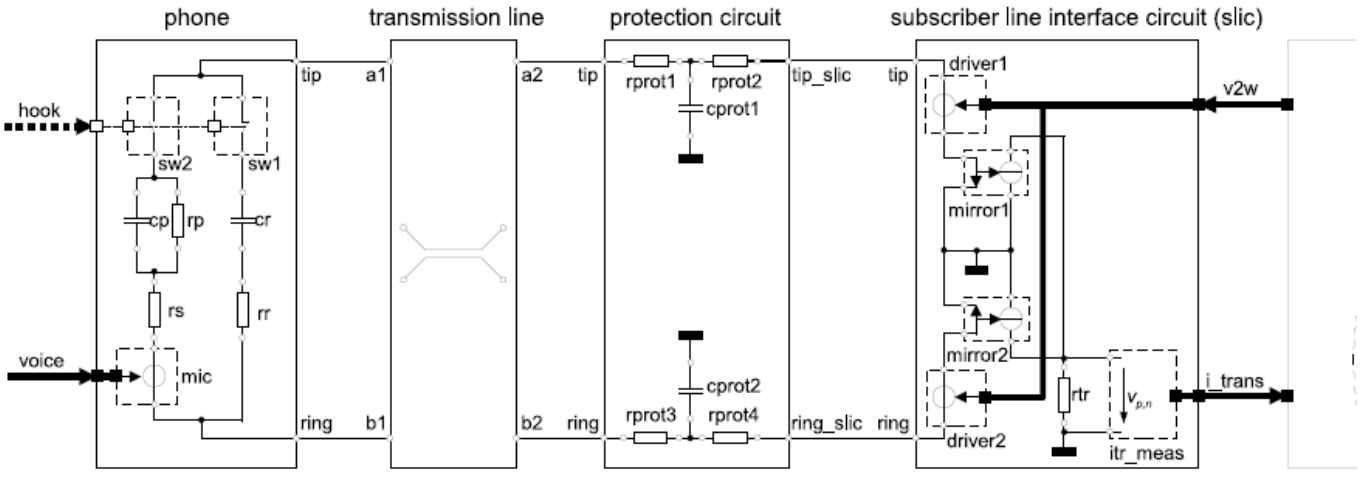


Figure 4.11. The Plain Old Telephone System (POTS) front-end

Рисунок 4.11. Внешний интерфейс простой телефонной системы (POTS)

Реализация телефона, схемы защиты и SLIC приведены ниже.

```
    SC_MODULE(phone)
    {
    // terminals and ports
    sca_eln::sca_terminal tip;
    sca_eln::sca_terminal ring;
    sca_tdf::sca_in<double> voice;
    sc_core::sc_in<bool> hook;
    // electrical primitives
    sca_eln::sca_de::sca_rswitch sw1;
    sca_eln::sca_de::sca_rswitch sw2;
    sca_eln::sca_c cr, cp;
    sca_eln::sca_r rr, rs, rp;
    sca_eln::sca_tdf::sca_vsource mic;
    phone( sc_core::sc_module_name nm, double cr_val =
1.0e-6, double rr_val = 1.0e3,
    double rs_val = 220.0, double cp_val = 115.0e-9,
    double rp_val = 820.0 )
    : tip("tip"), ring("ring"), voice("voice"),
hook("hook"),
    sw1("sw1"), sw2("sw2"), cr("cr", cr_val), cp("cp",
cp_val),
    rr("rr", rr_val), rs("rs", rs_val), rp("rp", rp_val),
mic("mic"),
    w_offhook("w_offhook"), w_onhook("w_onhook"),
w1("w1"), w2("w2"), wring("wring")
    {
    // architecture
    sw1.p(tip);
    sw1.n(w_onhook);
    sw1.ctrl(hook);
    sw1.off_state = true;
    sw2.p(tip);
    sw2.n(w_offhook);
    sw2.ctrl(hook);
    cr.p(wring);
    cr.n(w_onhook);
    rr.p(wring);
    rr.n(ring);
    rs.p(w1);
    rs.n(w2);
    cp.p(w1);
    cp.n(w_offhook);
    rp.p(w_offhook);
```

```
    rp.n(w1);
    mic.p(w2);
    mic.n(ring);
    mic.inp(voice);
    }
    private:
    // nodes
    sca_eln::sca_node w_offhook, w_onhook, w1, w2, wring;
    };


    SC_MODULE(protection_circuit)
    {
    // terminals
    sca_eln::sca_terminal tip_slic;
    sca_eln::sca_terminal ring_slic;
    sca_eln::sca_terminal tip;
    sca_eln::sca_terminal ring;
    // electrical primitives
    sca_eln::sca_r rprot1, rprot2, rprot3, rprot4;
    sca_eln::sca_c cprot1, cprot2;
    protection_circuit( sc_core::sc_module_name, double
rprot1_val = 20.0, double rprot2_val = 20.0,
    double rprot3_val = 20.0, double rprot4_val = 20.0,
    double cprot1_val = 18.0e-9,
    double cprot2_val = 18.0e-9 )
    : tip_slic("tip_slic"), ring_slic("ring_slic"),
tip("tip"), ring("ring"),
    rprot1("rprot1", rprot1_val), rprot2("rprot2",
rprot2_val),
    rprot3("rprot3", rprot3_val), rprot4("rprot4",
rprot4_val),
    cprot1("cprot1", cprot1_val), cprot2("cprot2",
cprot2_val),
    n_tip("n_tip"), n_ring("n_ring"), gnd("gnd")
    {
    // architecture
    rprot1.p(tip);
    rprot1.n(n_tip);
    rprot2.p(tip_slic);
    rprot2.n(n_tip);
    cprot1.p(n_tip);
    cprot1.n(gnd);
    rprot3.p(ring);
    rprot3.n(n_ring);
```

```
    rprot4.p(ring_slic);
    rprot4.n(n_ring);
    cprot2.p(n_ring);
    cprot2.n(gnd);
    }
    private:
    // nodes
    sca_eln::sca_node n_tip, n_ring;
    sca_eln::sca_node_ref gnd;
    };



    SC_MODULE(slic)
    {
    // terminals and ports
    sca_eln::sca_terminal tip;
    sca_eln::sca_terminal ring;
    sca_tdf::sca_in<double> v2w;
    sca_tdf::sca_out<double> i_trans;
    // electrical primitives
    sca_eln::sca_tdf::sca_vsource driver1, driver2;
    sca_eln::sca_tdf::sca_vsink itr_meas;
    sca_eln::sca_cccs mirror1, mirror2;
    sca_eln::sca_r rtr;
    slic( sc_core::sc_module_name, double scale_v_tr =
1.0, double scale_i_tr = 1.0 )
    : tip("tip"), ring("ring"), v2w("v2w"),
i_trans("i_trans"),
    driver1("driver1", scale_v_tr/2.0),
driver2("driver2", scale_v_tr/2.0),
    itr_meas("itr_meas", scale_i_tr),
    mirror1("mirror1", 0.5), mirror2("mirror2", -0.5),
rtr("rtr", 1.0),
    n_tr_i("n_tr_i"), n_tip_gnd("n_tip_gnd"),
n_ring_gnd("n_ring_gnd"),
    gnd("gnd")
    {
    // architecture
    driver1.inp(v2w);
    driver1.p(tip);
    driver1.n(n_tip_gnd);
    driver2.inp(v2w);
    driver2.p(ring);
    driver2.n(n_ring_gnd);
```

```
    mirror1.ncp(n_tip_gnd);
    mirror1.ncn(gnd);
    mirror1.np(n_tr_i);
    mirror1.nn(gnd);
    mirror2.ncp(n_ring_gnd);
    mirror2.ncn(gnd);
    mirror2.np(n_tr_i);
    mirror2.nn(gnd);
    rtr.p(n_tr_i);
    rtr.n(gnd);
    itr_meas.p(n_tr_i);
    itr_meas.n(gnd);
    itr_meas.outp(i_trans);
    }
    private:
    // nodes
    sca_eln::sca_node n_tr_i, n_tip_gnd, n_ring_gnd;
    sca_eln::sca_node_ref gnd;
    };
```

Реализация внешнего интерфейса POTS выполняется в функции sc_main, которая является основной программой.

Здесь показан только конкретизация и структурный состав.

```
    int sc_main(int argc,char* argv[])
    {
    sca_eln::sca_node n_slic_tip, n_slic_ring;
    sca_eln::sca_node n_tip_a1, n_tip_a2, n_ring_b1,
n_ring_b2;
    transmission_line;
    sca_tdf::sca_signal<double> s_v_in;
    sca_tdf::sca_signal<double> s_i_trans;
    sca_tdf::sca_signal<double> s_voice;
    sc_core::sc_signal<bool> s_hook;
    // testbench modules
    ...
    slic i_slic("i_slic");
    i_slic.tip(n_slic_tip);
    i_slic.ring(n_slic_ring);
    i_slic.v2w(s_v_in);
    i_slic.i_trans(s_i_trans);
    protection_circuit
i_protection_circuit("i_protection_circuit");
    i_protection_circuit.tip_slic(n_slic_tip);
    i_protection_circuit.ring_slic(n_slic_ring);
    i_protection_circuit.tip(n_tip_a2);
```

```
    i_protection_circuit.ring(n_ring_b2);
    sca_eln::sca_transmission_line
i_transmission_line("i_transmission_line",
    50.0, sc_core::SC_ZERO_TIME, 0.0);
    i_transmission_line.a1(n_tip_a1);
    i_transmission_line.b1(n_ring_b1);
    i_transmission_line.a2(n_tip_a2);
    i_transmission_line.b2(n_ring_b2);
    phone i_phone("i_phone");
    i_phone.tip(n_tip_a1);
    i_phone.ring(n_ring_b1);
    i_phone.voice(s_voice);
    i_phone.hook(s_hook);
    ...
    };
```

Глава 5. Анализ слабых сигналов в частотной области

5.1. Основы моделирования

Для анализа поведения в частотной области системы аналоговых / смешанных сигналов, меняющихся малых сигналов, называется сигналы переменного тока (AC) на разных частотах используются симуляции и анализ отклика цепи в установившемся состоянии. Используются либо синусоидальные источники слабого сигнала, либо источники шума, и применяются к схеме, которая линеаризуется вокруг заданной рабочей точки постоянного тока (DC). Это означает, что поведение для сильных сигналов, такое как нелинейности, вызывающие искажения, не фиксируется во время малосигнального частотного анализа.

Эти методы анализа в области переменного тока могут вычислять поведение в частотной области для слабого сигнала всей аналоговая / смешанная сигнальная система, которая может состоять из модулей, доступных для моделей вычислений. Модули TDF могут встраивать пользовательское описание частотной области малого сигнала. Для LSF и примитивных модулей ELN, поведение в частотной области малого сигнала неявно является частью описания примитива. На рисунке 5.1 показан пример системы со смешанными сигналами, содержащей модели TDF, LSF и ELN.

Модули, помеченные «AC», помимо описания во временной области, имеют связанный с ним представление частотного домена малого сигнала.

На основе структурного состава, составляются линейные комплексные уравнения.

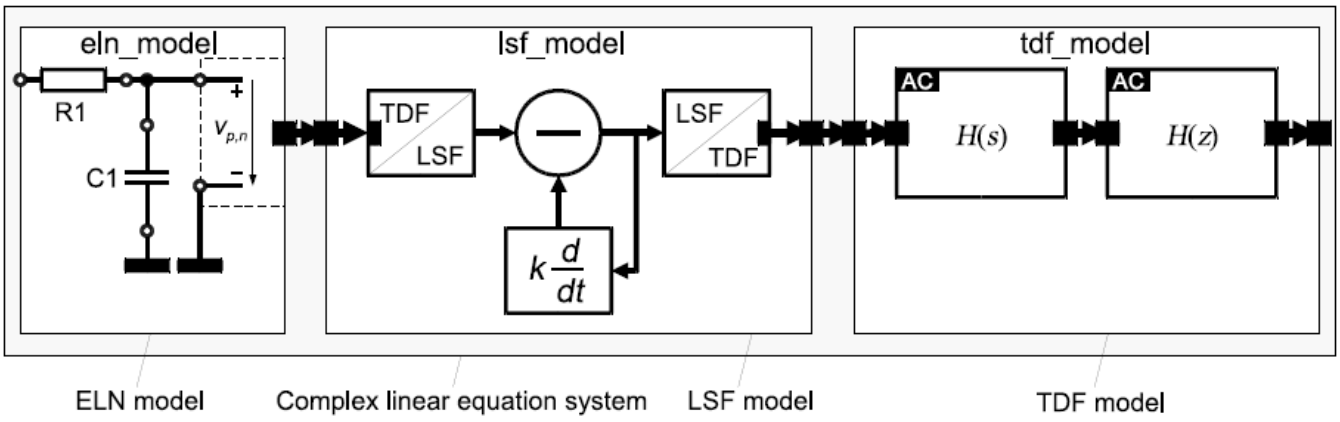


Figure 5.1. Small-signal frequency-domain description using TDF, LSF and ELN modules

Рисунок 5.1. Описание частотной области слабого сигнала с использованием модулей TDF, LSF и ELN

5.1.1. Настройка системы уравнений

Система линейных комплексных уравнений будет использовать кластер TDF, а также уравнения LSF и ELN системы, которые изначально определены для моделирования во временной области. Преобразование уравнения LSF и ELN системы из представления во временной области в представление частотной области слабого сигнала сделано, используя правила преобразования

Лапласа. Как правило, для данной функции f (t) следующие замены будут применяется к системам уравнений ELN и LSF, составленным во временной области:

- Дифференцирование d / dt заменяется на jw.
- Интегрирование заменяется на 1 / jw.
- Задержка f (t-delay) заменяется на $e^{-jw\ delay}$

Замена приведет к функции F (jw) в частотной области для составляющих LSF и ELN.

Модули TDF позволяют определять пользовательское поведение в частотной области малого сигнала как часть определения примитива. Нет доступного механизма для получения «AC representation - представления AC». Это полностью ответственность пользователя за обеспечение согласованности определённой частотной области и временной области представления. Как реализовать поведение частотной области слабого сигнала в модулях TDF обсуждается в разделе 5.2.1.

5.1.2. Методы анализа

Поддерживаются два типа анализа:

1. Анализ в частотной области слабого сигнала: Решает для каждой точки частоты линейное комплексное уравнение системы, включая все составляющие источника малого сигнала в частотной области.

2. Анализ шума в малой области частотной области: решает систему линейных комплексных уравнений для каждой частотной точки и каждой составляющей источника шума в частотной области малого сигнала. Для этого все составляющие источников частотной области слабого сигнала и источников шума частотной области слабого сигнала, кроме активированного в настоящее время источника шума, установлены на ноль.

Результатом анализа частотной области или шума слабого сигнала является установившийся отклик или функция передачи схемы, описанная от входного порта до выходного порта всей системы. Во время анализа полученная система линейных комплексных уравнений решается для заданных частотных точек.

5.2. Языковые конструкции

5.2.1. Описание частотной области слабого сигнала в модулях TDF

Поведение в слабой области частотной области модуля TDF может быть определено в функции-члене ac_processing. Описание должно быть написано в виде линейной комплексной передаточной функции, захватывая поведение от входного порта TDF до выходного порта TDF. Различные функции, доступные для определения линейной комплексной передаточной функции, будут представлены в следующих разделах. Для этих расчетов должен быть использован контейнер данных типа sca_util :: sca_complex.

В приведенном ниже примере показана реализация передаточной функции H(s)=1. Промежуточный результат хранится в переменной res типа

sca_util :: sca_complex, которая назначается выходному порту TDF. Дополнительные подробности о методах доступа к портам приведены в следующем разделе.

```
SCA_TDF_MODULE(flat_response)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
SCA_CTOR(flat_response) {}
void processing()
{
out.write( in.read() );
}
void ac_processing()
{
double h = 1.0; // flat frequency response H(s) = 1
sca_util::sca_complex res;
res = h * sca_ac_analysis::sca_ac(in);
sca_ac_analysis::sca_ac(out) = res;
}
};
```

В случае, если выполняется анализ частотной области слабого сигнала, но нет определенных функций-членов ac_processing или, если комплексное значение не назначено одному или нескольким выходным портам TDF, все связанные значения портов устанавливаются в ноль, независимо от доступных значений на входных портах.

Обратите внимание, что нет автоматической проверки согласованности между описаниями во временной и частотной областях, поскольку эти определения используются заданными.

5.2.2. Доступ к порту

Для анализа частотной области слабого сигнала - комплексное значение всех портов TDF, кроме портов преобразователя, могут быть доступны с помощью функции sca_ac_analysis :: sca_ac с аргументами экземпляров портов, как показано в предыдущем примере. Этот метод доступа не зависит от типа порта, требуемого в моделировании домена времени.

Для входных портов функция sca_ac_analysis :: sca_ac возвращает постоянную ссылку на значение типа sca_util :: sca_complex, что означает, что никакое значение не может быть назначено входному порту TDF.

Для выходных портов, функция возвращает ссылку на значение типа sca_util :: sca_complex, позволяя присваивать вклад для анализа частотной области слабого сигнала.

Для анализа шума в частотной области малого сигнала источник шума, независимый от значений входного порта, может быть назначен на выходной порт TDF с помощью функции sca_ac_analysis :: sca_ac_noise. В этом случае

значение, присвоенное с помощью функции sca_ac_analysis :: sca_ac, будет игнорироваться.

Обратите внимание, что значения возвращаются из функций sca_ac_analysis :: sca_ac и sca_ac_analysis :: sca_ac_noise определяются реализацией и не имеют физической интерпретации. Эти значения могут использоваться только для описания сложной линейной зависимости между входным и выходным портами, к которым осуществляется доступ, используя эти функции доступа к порту.

5.3. Сервисные функции

Расширения SystemC AMS предлагают набор служебных функций, которые могут использоваться внутри функции-члена ac_processing для определения поведения частотной области слабого сигнала. Обратите внимание, что эти функции не могут использоваться при обработке метода **processing** во временной области.

5.3.1. Задержка в частотной области

Функция sca_ac_analysis :: sca_ac_delay может использоваться для реализации непрерывной задержки, определяемой как $e^{-jw\ delay}$. Следующий пример является расширением примера задержки TDF, представленного в разделе 2.3.5. Задержка теперь является параметром модуля и используется для инициализации задержки отсчётов до 0,0 для моделирования во временной области.

Обратите внимание, что параметр delay является целочисленным значением, отражающим количество выборок, которые будут вставлены для моделирования во временной области, используя дискретный шаг по времени. Функция-член ac_processing реализует поведение этой задержки в частотной области. Во-первых, задержка переводится в непрерывный вариант времени, используя функцию-член get_timestep, умноженную на количество отсроченных выборок. Это значение типа sca_core :: sca_time передается в качестве аргумента функции sca_ac_analysis :: sca_ac_delay, которая определяет задержку в частотной области.

```
    SCA_TDF_MODULE(my_tdf_ac_delay)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    my_tdf_ac_delay( sc_core::sc_module_name, unsigned
long delay_ )
    : in("in"), out("out"), delay(delay_) {}
    void set_attributes()
    {
    out.set_delay(delay);
    }
    void initialize() // time-domain initialization
```

```
    {
    for( unsigned long i = 0; i < delay; i++ )
    out.initialize( 0.0, i );
    }
    void processing() // time-domain implementation
    {
    out.write( in.read() );
    }
    void ac_processing() // frequency-domain
implementation
    {
    sca_core::sca_time ct_delay = out.get_timestep() *
delay; // calculate continuous-time delay
    sca_ac_analysis::sca_ac(out) =
sca_ac_analysis::sca_ac(in) *
    sca_ac_analysis::sca_ac_delay( ct_delay );
    }
    private:
    unsigned long delay;
    };
```

5.3.2. Передаточные функции Лапласа

Описания в частотной области передаточных функций Лапласа в форме числитель-знаменатель и нуль - полюс доступны с использованием служебных функций sca_ac_analysis :: sca_ac_ltf_nd илиса_ac_analysis :: sca_ac_ltf_zp, соответственно. Их можно использовать в сочетании с представлением временной областью, как показано в примере ниже.

```
    SCA_TDF_MODULE(ltf_filter_ac)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    ltf_filter_ac( sc_core::sc_module_name nm, double
fc_, double h0_ = 1.0 )
    : in("in"), out("out"), fc(fc_), h0(h0_) {}
    void initialize()
    {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0 / ( 2.0 * M_PI * fc );
    }
    void processing() // time-domain implementation
    {
    out.write( ltf_nd( num, den, in.read(), h0 ) );
    }
```

```
    void ac_processing() // frequency-domain
implementation
    {
    sca_ac_analysis::sca_ac(out) =
sca_ac_analysis::sca_ac_ltf_nd(
    num, den, sca_ac_analysis::sca_ac(in), h0 );
    }
    private:
    sca_tdf::sca_ltf_nd ltf_nd; // Laplace transfer
function
    sca_util::sca_vector<double> num, den; // numerator
and denominator coefficients
    double fc; // 3dB cutoff frequency in Hz
    double h0; // DC gain
    };
```

5.3.3. Определения S-домена

Функция sca_ac_analysis :: sca_ac_s поддерживает представления в частотной области, определенные в s-домене, используя оператор Лапласа $S^n = (j\omega)^n$.

На рисунке 5.2 показано определение, частотная характеристика H (s) и реализация фильтра второго порядка низкочастотного сигнала, реализованный во временной и частотной области.

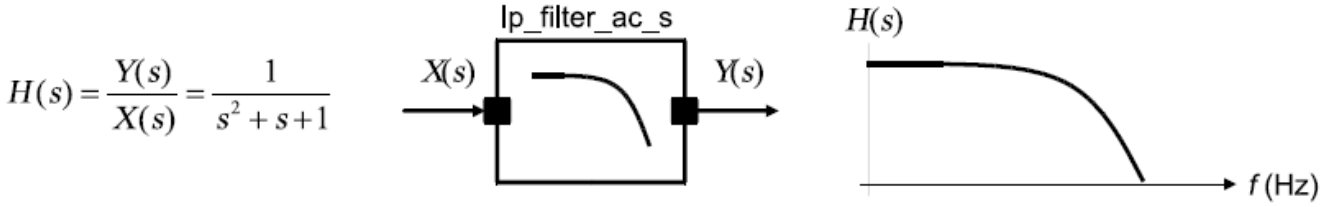


Figure 5.2. Frequency response of second order low-pass filter implemented in the s-domain

Рисунок 5.2. Частотная характеристика фильтра нижних частот второго порядка, реализованного в s-области

```
    SCA_TDF_MODULE(lp_filter_ac_s)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    SCA_CTOR(lp_filter_ac_s) : in("in"), out("out") {}
    void initialize()
    {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0;
```

```
    den(2) = 1.0;
    }
    void processing() // time-domain implementation
    {
    out.write( ltf_nd( num, den, in.read(), 1.0 ) );
    }
    void ac_processing() // frequency-domain
implementation
    { sca_util::sca_complex h = 1.0 / (
sca_ac_analysis::sca_ac_s(2) +
    sca_ac_analysis::sca_ac_s(1) + 1.0 );
    sca_ac_analysis::sca_ac(out) = h *
sca_ac_analysis::sca_ac(in);
    }
    private:
    sca_tdf::sca_ltf_nd ltf_nd;
    sca_util::sca_vector<double> num, den;
    };
```

Альтернативно, поведение в частотной области может быть реализовано с использованием отношения $s = j\omega$. Функция-член ac_processing из предыдущего примера может быть заменена реализацией, которая использует функция sca_ac_analysis :: sca_ac_w, которая возвращает текущую угловую частоту в радианах / секундах:

```
    void ac_processing() // frequency-domain
implementation using s = j*w
    {
    sca_util::sca_complex s = sca_util::SCA_COMPLEX_J *
sca_ac_analysis::sca_ac_w();
    sca_util::sca_complex h = 1.0 / ( s * s + s + 1.0 );
    sca_ac_analysis::sca_ac(out) = h *
sca_ac_analysis::sca_ac(in);
    }
```

Согласно соотношению $\omega = 2\pi f$, частотный член также может быть использован.

Реализация с использованием функции sca_ac_analysis :: sca_ac_f, которая возвращает текущую частоту в герцах, становится:

```
    void ac_processing() // frequency-domain
implementation using s = j*2*PI*f
    {
    sca_util::sca_complex s = sca_util::SCA_COMPLEX_J *
2.0 * M_PI * sca_ac_analysis::sca_ac_f();
    sca_util::sca_complex h = 1.0 / ( s * s + s + 1.0 );
```

```
    sca_ac_analysis::sca_ac(out) = h *
sca_ac_analysis::sca_ac(in);
    }
```

5.3.4. Определения Z-домена

Функция sca_ac_analysis :: sca_ac_z поддерживает представления в частотной области, определенные в z-домене, используя оператор $z^n (= e^{j\omega \cdot n \cdot nstep})$,где n - целое число, определяющее задержку, и tstep является временным шагом между задержками. Если этот аргумент не используется, tstep будет определён как временной шаг, возвращаемый функцией-членом get_timestep.

На рисунке 5.3 показано определение и частотная характеристика H (z) гребенчатого фильтра.

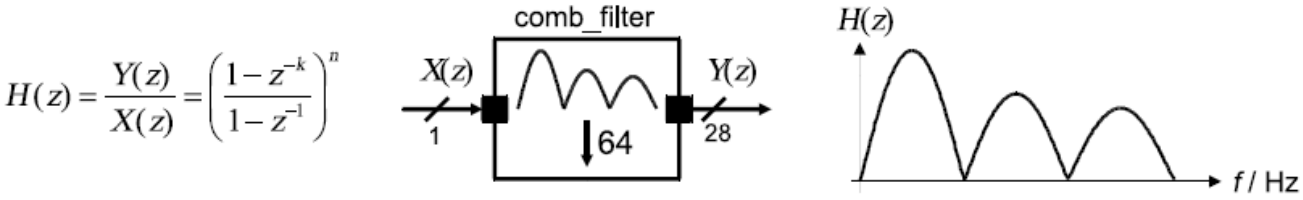


Figure 5.3. Frequency response of a comb-filter implemented in the z-domain

Рисунок 5.3. АЧХ гребенчатого фильтра, реализованного в z-области

Для реализации в частотной области используется функция sca_ac_analysis :: sca_ac_z, как показано в пример ниже.

```
    SCA_TDF_MODULE(comb_filter)
    {
    sca_tdf::sca_in<bool> in;
    sca_tdf::sca_out<sc_dt::sc_int<28> > out;
    comb_filter( sc_core::sc_module_name, int k_ = 64,
int n_ = 3 )
    : in("in"), out("out"), k(k_), n(n_) {}
    void set_attributes()
    {
    in.set_rate(k);
    out.set_rate(1);
    }
    void ac_processing() // frequency-domain
implementation
    {
    // complex transfer function
```

```
    sca_util::sca_complex h = pow( ( 1.0 -
sca_ac_analysis::sca_ac_z(-k) ) /
    ( 1.0 - sca_ac_analysis::sca_ac_z(-1) ), n );
    sca_ac_analysis::sca_ac(out) = h *
sca_ac_analysis::sca_ac(in) ;
    }
    void processing() // time-domain implementation
    {
    int x, y, i;
    for( i = 0; i < k; i++) {
    x = in.read(i);
    ...
    }
    out.write(y);
    }
    private:
    int k; // decimation factor
    int n; // order of filter
    };
```

5.3.5. Обнаружение слабосигнального анализа в частотной области

Функции утилиты sca_ac_analysis :: sca_ac_is_running и sca_ac_analysis :: sca_ac_noise_is_running можно использовать при обработке функции-члена или ac_processing модуля TDF для реализации специфического поведения, которое зависит от типа выполнения (run) анализа.

Функция sca_ac_analysis :: sca_ac_is_running возвращает true, когда выполняется частотный анализ слабого сигнала или анализ шума. Функция sca_ac_analysis :: sca_ac_noise_is_running возвращает true только, если проводится анализ шума в частотной области слабого сигнала.

Пример ниже показывает реализацию синусоидального источника, который может использоваться для моделирования во временной области и в частотной области.

```
    SCA_TDF_MODULE(sin_src)
    {
    sca_tdf::sca_out<double> out;
    sin_src( sc_core::sc_module_name nm, double offset_=
0.0, double ampl_= 1.0,
    double noise_ampl_= 0.1, double freq_ = 1.0e3,
    sca_core::sca_time Tm_ = sca_core::sca_time(0.125,
sc_core::SC_MS) )
    : out("out"), offset(offset_), ampl(ampl_),
noise_ampl(noise_ampl_), freq(freq_), Tm(Tm_)
    {}
    void set_attributes()
```

```
    {
    set_timestep(Tm);
    }
    void processing()
    {
    double t = get_time().to_seconds(); // actual time
    out.write( offset + ampl * std::sin( 2.0*M_PI*freq*t
) );
    }
    void ac_processing()
    {
    if( sca_ac_analysis::sca_ac_noise_is_running() ) //1
    sca_ac_analysis::sca_ac_noise(out) = noise_ampl;
    else
    sca_ac_analysis::sca_ac(out) = ampl;
    }
    private:
    double offset, ampl, noise_ampl, freq;
    sca_core::sca_time Tm;
    };
```

1. Только для анализа слабого шума в частотной области, функция sca_ac_analysis :: sca_ac_noise_is_running возвращает true. В этом случае амплитуда шума установлена в источнике.

5.4. Анализ частотной области слабого сигнала с комбинированным TDF, LSF и ELN моделями

Как уже говорилось во введении к этой главе, анализ частотной области слабого сигнала способен определить частотное поведение всей системы аналоговых / смешанных сигналов. Частотная характеристика всей системы может быть проанализирована с использованием модулей TDF, у которых определено поведение в частотной области в их функции-члене ac_processing, плюс описание частотной области примитивных модулей LSF и ELN, которые извлекаются из системы уравнений LSF и ELN во время разработки.

Реализация, показанная ниже, основана на составе модуля, как показано на рисунке 5.1.

Пример показывает время, частотную область и моделирование шума. Результаты записываются в разные файлы трассировки.

```
    int sc_main(int argc, char* argv[])
    {
    sca_eln::sca_node net1;
    sca_tdf::sca_signal<double> sig1, sig2, sig3;
    ... // source and sink
    eln_model a("a");
```

```
    a.p(net1);
    a.outp(sig1);
    lsf_model b("b");
    b.in(sig1);
    b.out(sig2);
    tdf_model c("c");
    c.in(sig2);
    c.out(sig3);
    // tracing
    sca_util::sca_trace_file* tf =
sca_util::sca_create_tabular_trace_file("trace.dat");
    sca_util::sca_trace(tf, net1, "net1");
    sca_util::sca_trace(tf, sig1, "sig1");
    sca_util::sca_trace(tf, sig2, "sig2");
    sca_util::sca_trace(tf, sig3, "sig3");
    // start time-domain simulation
    sc_core::sc_start(10, sc_core::SC_MS);
    tf->reopen("ac_trace.dat");
    tf-
>set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RA
D));
    // start frequency-domain simulation
    sca_ac_analysis::sca_ac_start(1.0e3, 100.0e4, 4,
sca_ac_analysis::SCA_LOG);
    tf->reopen("ac_noise_trace.dat");
    tf-
>set_mode(sca_util::sca_noise_format(sca_util::SCA_NOISE_
ALL));
    // start frequency-domain noise simulation
    sca_ac_analysis::sca_ac_noise_start(1.0e3, 100.0e4,
4, sca_ac_analysis::SCA_LOG);
    sca_util::sca_close_tabular_trace_file(tf);
    return 0;
    }
```

Более подробную информацию о возможностях управления и отслеживания симуляции можно найти в главе 6.

Глава 6. Моделирование и трассировка

Расширения AMS используют функции SystemC для запуска и остановки моделирования во временной области. Новые функции доступны для моделирования в частотной области. Расширенный механизм отслеживания доступен для включения или отключения трассировки во временной или частотной областях при выполнении моделирования.

6.1. Симуляция управления

6.1.1. Моделирование во временной области

Моделирование во временной области (переходный процесс) запускается вызовом sc_core :: sc_start из функции sc_main, как показано в примере ниже.

```
#include <systemc-ams>
#include "my_source.h"
#include "my_control.h"
#include "my_dut.h"
#include "my_sink.h"
int sc_main(int argc, char* argv[])
{
sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);
sca_tdf::sca_signal<double> sig1, sig2;
sc_core::sc_signal<bool> sc_sig;
my_source i_my_source("i_my_source");
i_my_source.out(sig1);
my_control i_my_ctrl("i_my_ctrl");
i_my_ctrl.out(sc_sig);
my_dut i_my_dut("i_my_dut");
i_my_dut.in(sig1);
i_my_dut.ctrl(sc_sig);
i_my_dut.out(sig2);
my_sink i_my_sink("i_my_sink");
i_my_sink.in(sig2);
sc_core::sc_start(10.0, sc_core::SC_MS);
return 0;
}
```

Программные аргументы

Функция sc_main действует как основная программа и имеет ту же сигнатуру аргументов и возвращаемое значение, что и обычная в C ++ функция ввода программы int main (int argc, char * argv []). Аргумент argc указывает количество аргументов, передаваемых в основную процедуру. Аргумент argv [] является полем указателей на различных строковых аргументов.

Обратите внимание, что реализации или симуляторы, которые поддерживают SystemC и расширения AMS, могут использовать разные механизмы для определения основного тела программы или даже использовать

альтернативный подход к sc_main.

Разрешение по времени

Для моделирования AMS рекомендуется использовать наименьшее возможное временное разрешение, охватывающее время симуляции с использованием функции sc_core :: sc_set_time_resolution. Рекомендуется добавить эту функцию как первый оператор в функции sc_main. Для моделирования во временной области, временное разрешение 1 фемтосекунда (fs) рекомендуется, что является наименьшим возможным временным разрешением, допускающим максимальное время моделирования 2^{64} фс, что составляет примерно 5 часов. В случае, если требуется более длительное время моделирования, разрешение по времени должно быть увеличено, что приведёт к более грубой временной сетке и возможным ошибкам округления.

Моделирование аргументов

Функция sc_core :: sc_start без аргументов приведёт к симуляции, которая выполняется до последнего события было обработано, что может быть бесконечно долго. Для симуляции в течение ограниченного периода времени, время для симуляции можно указывать как двойное значение double вместе с единицей времени или как объект класса sc_core :: sc_time.

Функция sc_core :: sc_start может вызываться несколько раз, как показано в примере ниже:

```
int sc_main(int argc, char* argv[])
{
// instantiate design and testbench, setup tracing,
...
...
sc_core::sc_start(10.0, sc_core::SC_MS);  //1
...
sc_core::sc_time sim_time(10.0, sc_core::SC_MS);
sc_core::sc_start(sim_time); //2
...
sc_core::sc_start(); //3
return 0;
}
```

1. Начать анализ переходных процессов, где время моделирования задается двумя аргументами. Первый аргумент время типа double. Второй аргумент - это единица времени, которая является объектом класса sc_core :: sc_time_unit.

2. Начать анализ переходных процессов, где время моделирования задается одним аргументом, который является объектом класса sc_core :: sc_time.

3. В этом случае время моделирования не указывается. Анализ переходных процессов будет выполняться до тех пор, пока очередь событий не

станет пустой.

6.1.2. Моделирование в слабой области частотной области

Моделирование в частотной области также запускается из функции sc_main, используя sca_ac_analysis :: sca_ac_start для моделирования слабого сигнала (AC) и sca_ac_analysis :: sca_ac_noise_start для моделирования шума в слабой области частотной области. В случае, когда описание модели не было разработано, потому что sc_core :: sc_start ещё не был вызван, это будет выполнено автоматически до начала первого моделирования в частотной области.

Можно последовательно вызвать функции запуска анализа частотной и временной областей в любом порядоке внутри функции sc_main, чтобы проанализировать описание системы в различных рабочих точках или цифровые состояния.

В приведенном ниже примере показано использование функций, которые принимают в качестве аргументов начальную частоту, частоту остановки, количество частотных точек и линейную (sca_ac_analysis :: SCA_LIN) или логарифмическую (sca_ac_analysis :: SCA_LOG) шкалу частот следует использовать.

```
    // frequency-domain simulations from 1kHz to 10kHz
with 100 points on a linear scale:
    sca_ac_analysis::sca_ac_start(1.0e3, 10.0e3, 100,
sca_ac_analysis::SCA_LIN);
    sca_ac_analysis::sca_ac_noise_start(1.0e3, 10.0e3,
100, sca_ac_analysis::SCA_LIN);
    // frequency-domain simulations from 1Hz to 1MHz with
1001 points on a logarithmic scale:
    sca_ac_analysis::sca_ac_start(1.0, 1.0e6, 1001,
sca_ac_analysis::SCA_LOG);
    sca_ac_analysis::sca_ac_noise_start(1.0, 1.0e6, 1001,
sca_ac_analysis::SCA_LOG);
```

6.2. Трассировка

Расширения SystemC AMS предоставляют служебные функции для записи результатов моделирования (сигналов) в файлы трассировки, используя формат Value Change Dump (VCD) или табличный формат. Формат VCD имеет ограниченные возможности для отслеживания сигналов, узлов, портов, терминалов или переменных AMS. Помимо отслеживания регулярных переменные и сигналов SystemC, он поддерживает трассировку только для моделирования во временной области. Табличный формат может использоваться для записи трассировок как во временной, так и в частотной областях.

Файл трассировки обычно создаётся на верхнем уровне (например, внутри sc_main) после того, как все модули и сигналы были созданы, и

непосредственно перед началом фактического моделирования с помощью sc_core :: sc_start, sca_ac_analysis :: sca_ac_start или sca_ac_analysis :: sca_ac_noise_start.

6.2.1. Файлы трассировки и форматы

Трассировка в файл VCD

Для трассировки сигналов с использованием формата VCD файл трассировки создается путем вызова функции sca_util :: sca_create_vcd_trace_file с именем файла в качестве аргумента. Эта функция возвращает указатель на структуру данных, которая используется для трассировки. Закрытие файла трассировки выполняется с помощью функции sca_util :: sca_close_vcd_trace_file с аргументом указатель на ту же структуру данных.

```
    // open trace file in VCD format
    sca_util::sca_trace_file* atf =
sca_util::sca_create_vcd_trace_file( "my_trace.vcd" );
    ...
    // close the trace file
    sca_util::sca_close_vcd_trace_file( atf );
```

Примечание: обычная трассировка VCD SystemC может использоваться для отслеживания сигналов AMS с использованием функций sc_core :: sc_create_vcd_trace_file, sc_core :: sc_trace и sc_core :: sc_close_vcd_trace_file, но в этом случае сигналы AMS транслируются и отслеживаются как сигналы дискретных событий с использованием выходных портов преобразователя TDF.

Таким образом, аспекты синхронизации между TDF и моделями вычисления дискретных событий могут играют роль в точности синхронизации отдельных отсчетов этих сигналов (см. раздел 2.4).

Трассировка в табличный файл

Для трассировки сигналов с использованием табличного формата файл трассировки создается путем вызова функции sca_util :: sca_create_tabular_trace_file с именем файла в качестве аргумента. Функция возвращает указатель на структуру данных, которая используется для трассировки. Закрытие файла трассировки выполняется с помощью функции sca_util :: sca_close_tabular_trace_file с аргументом указателя на ту же структуру данных, как показано в примере ниже.

```
    // open trace file in tabular format
    sca_util::sca_trace_file* atf =
sca_util::sca_create_tabular_trace_file( "my_trace.dat"
);
    ...
    // close the trace file
```

```
sca_util::sca_close_tabular_trace_file( atf );
```

Трассировка в табличный поток

Поскольку трассировка аналоговых сигналов может привести к очень большим файлам трассировки, функция трассировки AMS была расширена для трассировки в выходной поток, поэтому файл трассировки не создается. Это позволяет выполняит прямую обработку сигналов AMS, доступных в выходном потоке, полученном из std :: ostream, например, для немедленного отображения результатов или сжатия результаты в файл архива.

Для трассировки сигналов в выходной поток файл трассировки создается путем вызова функции sca_util :: sca_create_tabular_trace_file с объектом выходного потока в качестве аргумента. Функция возвращает указатель на объект класса sca_util :: sca_trace_file, который ссылается на поток и используется для управления трассировкой сигнала к нему (объекту). Закрытие файла трассировки выполняется с помощью функции sca_util :: sca_close_tabular_trace_file с аргументом указателя на тот же поток вывода, как показано в примере ниже.

```
    // trace in tabular format to the shell
    sca_util::sca_trace_file* atfs =
sca_util::sca_create_tabular_trace_file(std::cout);
    ...
    // close the trace file handle, the stream is
automatically closed once the scope of os is left.
    sca_util::sca_close_tabular_trace_file(atfs);
```

Контроль файла трассировки

Так как трассировка сигналов AMS может привести к очень большим и неуправляемым файлам сигналов, дополнительная функциональность доступна для контроля записи файлов трассировки. Следующие методы контроля файла трассировки доступны для класса sca_util :: sca_trace_file:

1. Функция-член enable начнет трассировку во время симуляции, где вызывается этот метод.

2. Функция-член **disable** остановит трассировку во время симуляции, где вызывается этот метод.

3. Функция-член **reopen** закроет существующий файл трассировки (если он был открыт) и продолжит трассировку в новом файле трассировки во время моделирования, где вызывается этот метод.

4. Функция-член **set_mode** изменит режим файла трассировки во время симуляции, где метод вызывается.

Доступны следующие моды:

- Шаг по времени (выборка) между выборками может быть установлен с помощью функции sca_util :: sca_sampling, где первый аргумент - шаг по времени, а второй аргумент - смещение по времени. Оба аргумента должен быть объектом класса sca_core ::

sca_time.

- Функция sca_util :: sca_decimation с аргументом n записывает в трассу только n-й пример файл.
- Функция sca_util :: sca_multirate определяет, какое значение сигнала следует записать в трассу файла, если фактическое значение не доступно. Это может происходить при отслеживании сигналов с разными скоростями и временными шагами. Доступные аргументы для интерполяции (sca_util :: SCA_INTERPOLATE), чтобы использовать последнее доступное значение (sca_util :: SCA_HOLD_SAMPLE) или не записывать значение вообще (Sca_util :: SCA_DONT_INTERPOLATE). В последнем случае символ «*» записывается в файл трассы.
- Для трассировки частотной области слабого сигнала функция sca_util :: sca_ac_format определяет формат, в котором записаны сигналы. Доступные аргументы функции: real /мнимый (sca_util :: SCA_AC_REAL_IMAG) и амплитуда / фаза по величине / радианы (sca_util :: SCA_AC_MAG_RAD) или дБ / градусы (sca_util :: SCA_AC_DB_DEG).
- Для трассировки частотной области слабого сигнала функция sca_noise_format определяет, как вклад шума записывается в файл трассировки. Если передано sca_util :: SCA_NOISE_ALL, каждый отдельный вклад шума записывается в файл трассировки. Если sca_util :: SCA_NOISE_SUM пройден, сумма всех вкладов шума записываются в файл трассировки.

В следующих разделах приводятся примеры использования управления файлами трассировки в сочетании с моделированием контроль.

6.2.2. Трассировка сигналов и комментариев

Поддерживаемые сигналы AMS

Функция sca_util :: sca_trace используется для отслеживания реальных сигналов AMS. Следующие элементы могут быть прослежены:

• Для моделей TDF, трассировка возможна для сигналов TDF, портов TDF и переменных, полученных из класса sca_tdf :: sca_trace_variable.

• Для моделей LSF отслеживание возможно для сигналов LSF и портов LSF.

• Для моделей ELN отслеживание напряжения поддерживается для узлов и терминалов. В текущем отслеживании через ELN поддерживаются примитивные модули с двумя терминалами. Некоторые симуляторы также поддерживают текущую трассировку через примитивные модули ELN с более чем двумя терминалами.

• SystemC (дискретное событие) сигналы и порты.

В приведенном ниже примере показано, как использовать функцию sca_util :: sca_trace для отслеживания сигналов AMS в моделях TDF, LSF или ELN.

```
    sca_util::sca_trace( atf, sig1, "sig1" ); // trace
TDF signal sig1
    sca_util::sca_trace( atf, sig_de, "sig_de" ); //
trace SystemC signal sig_de
    sca_util::sca_trace( atf, my_source.out, "out1" ); //
trace output of module my_source
    sca_util::sca_trace( atf, my_source.i_sin_src->out,
"out2" ); // trace output of nested module
    sca_util::sca_trace( atf, my_sink.trv, "trv" ); //
trace variable in module my_sink
```

Запись комментариев в файл трассировки

Чтобы записать некоторые пользовательские комментарии или замечания в табличный файл трассировки, функция можно использовать sca_util :: sca_write_comment, где первый аргумент - указатель на структуру данных файл трассировки и второй аргумент - это строка, содержащая комментарий. Комментарий, в том числе предыдущий символ «%» добавляется в файл трассировки во время симуляции, где вызывается эта функция.

```
    // open trace file in tabular format
    sca_util::sca_trace_file* atf =
sca_util::sca_create_tabular_trace_file( "my_trace.dat"
);
    ...
    // add comment to trace file
    sca_util::sca_write_comment( atf, "user-defined
comments" );
    ...
    // close the trace file
    sca_util::sca_close_tabular_trace_file( atf );
```

Обратите внимание, что добавление пользовательских комментариев может привести к несовместимости при использовании определённого сигнала наблюдения, в зависимости от поддерживаемых форматов файлов. Рекомендуется проверить, является ли конкретный вид сигнала поддерживаемым форматом, который позволяет включать пользовательские комментарии.

Пример файла трассировки

В этом разделе показаны некоторые результаты трассировки сигналов времени и частоты, основанные на следующей трассировке определение в программе sc_main:

```
    ...
    sca_util::sca_trace_file* tf =
sca_util::sca_create_tabular_trace_file("trace.dat");//1
    sca_util::sca_trace(tf, sig1, "sig1"); //2
```

```
    sca_util::sca_trace(tf, sig2, "sig2");
    sc_core::sc_start(2.0, sc_core::SC_MS);  //3
    tf->reopen("ac_trace.dat"); //4
    tf-
>set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RA
D)); //5
    sca_ac_analysis::sca_ac_start(1.0e3, 1.0e6, 4,
sca_ac_analysis::SCA_LOG);  //6
    sca_util::sca_close_tabular_trace_file(tf); //7
    …
```

1. Трассировка сигналов AMS в файл в табличном формате с использованием функции трассировки расширений AMS.
2. Определите, какие сигналы отслеживать.
3. Начать моделирование во временной области. Сигналы «sig1» и «sig2» будут отслежены.
4. Закройте текущий файл трассировки и начните трассировку в новый файл для анализа в частотной области.
5. Определение для отслеживания амплитуды и фазы сигналов по амплитуде и радианам.
6. Начните моделирование в частотной области от 1 кГц до 1 МГц с 4 точками в логарифмическом масштабе.
7. Закройте файл трассировки.

Файл trace.dat показан ниже. % time в первой строке указывает, что этот файл был создан во время моделирование во временной области, и показывает имена сигналов, которые отслеживаются. Каждая строка показывает время в секундах и значения сигнала в этот момент времени. Значения разделены одним или несколькими пробелами.

```
%time sig1 sig2
0 0 0
0.0005 1 1e-6
0.001 2 1.5e-6
0.0015 3 2e-6
0.002 4 2.5e-5
```

В следующем примере показан результат трассировки частотной области слабого сигнала в ac_trace.dat. Файл начинается с *%frequency* в заголовке. Формат сигналов переменного тока устанавливается на амплитуду (величину) и фазу (в радианах), обозначенная суффиксами *.mag* и *.rad* к именам сигналов, соответственно.

```
%frequency sig1.mag sig1.rad sig2.mag sig2.rad
1000 1 0 2.53302962314e-08 -3.14143349864
10000 1 0 2.53302959138e-10 -3.1415767381
100000 1 0 2.53302959106e-12 -3.14159106204
1000000 1 0 2.53302959106e-14 -3.14159249443
```

6.3. Testbenches

Испытательные стенды используются для обеспечения сигналов стимулов для тестируемого устройства (DUT) и проверки результатов или реакции DUT. Очень часто DUT переводится в определённое состояние с использованием внешнего управления. Структура типичного стенд приведена на рисунке 6.1.

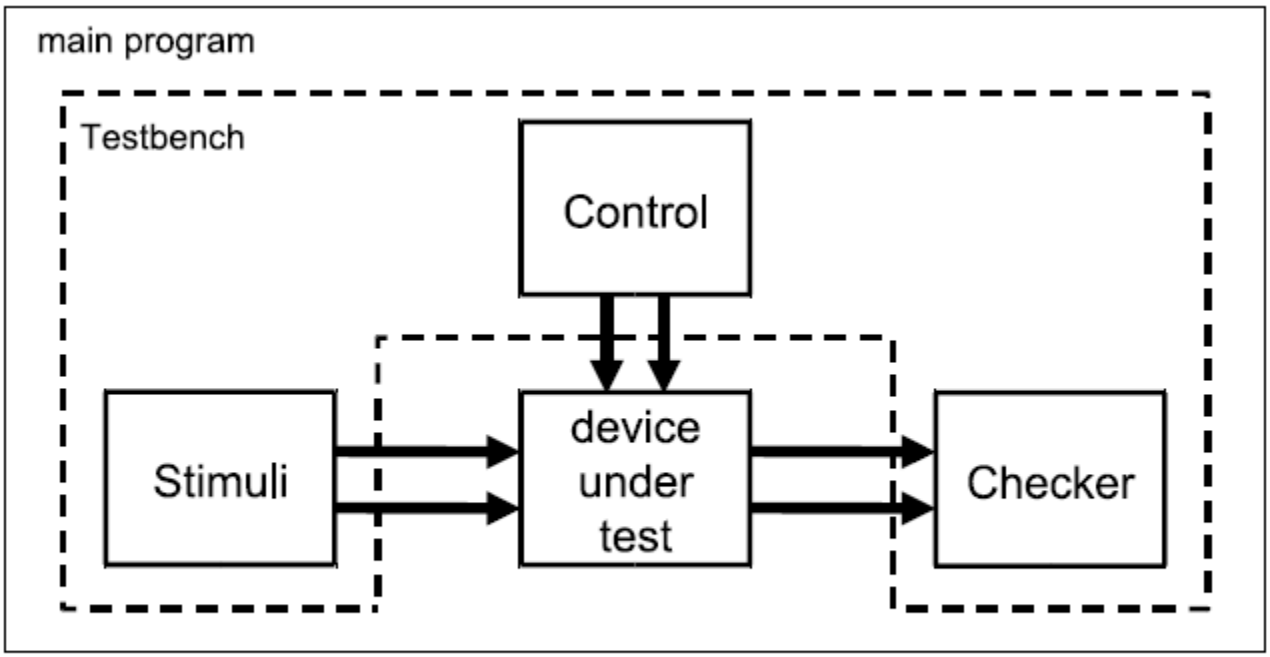


Figure 6.1. Testbench containing stimulus, control, checker, and device under test

Рисунок 6.1. Испытательный стенд, содержащий стимул, контроль, контроллер и тестируемое устройство

Тестовый стенд может быть реализован различными способами:

• Стимул и контроллер могут быть встроены в основную программу, а результаты проверены в другом модуле. Таким образом, основная программа выступает в качестве тестового стенда.

• Стимул, контроллер и контроллер являются частью специального модуля, который создаётся в главной программе. Такой модуль часто называют компонентом проверки, который в основном действует в качестве тестового стенда.

• Стимул и контроллер - это отдельные модули, оба экземпляра которых созданы в основной программе. Проверка встроена в основную программу, которая выступает в качестве тестового стенда.

Помимо приведённых выше примеров, есть и другие возможности для создания тестового стенда. Очевидно, нет единого «правильного» способа

создания тестового стенда; это зависит от приложения.

В приведённом ниже примере показана основная программа, в которой используются стимулы my_source, элемент управления my_control и сток my_sink, созданные как объекты. Вместе с трассировкой, реализованной в виде встроенного кода, они сформировали стенд Тестируемое устройство my_dut создается как модуль и подключается к модулям испытательного стенда.

```
    #include <systemc-ams>
    #include "my_source.h"
    #include "my_control.h"
    #include "my_dut.h"
    #include "my_sink.h"
    int sc_main(int argc, char* argv[])
    {
    sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);
    sca_tdf::sca_signal<double> sig1, sig2;
    sc_core::sc_signal<bool> sc_sig;
    my_source i_my_source("i_my_source");
    i_my_source.out(sig1);
    my_control i_my_ctrl("i_my_ctrl");
    i_my_ctrl.out(sc_sig);
    my_dut i_my_dut("i_my_dut");
    i_my_dut.in(sig1);
    i_my_dut.ctrl(sc_sig);
    i_my_dut.out(sig2);
    my_sink i_my_sink("i_my_sink");
    i_my_sink.in(sig2);
    sc_core::sc_trace_file* tf =
sc_core::sc_create_vcd_trace_file("my_sc_trace"); //1
    sc_core::sc_trace(tf, sc_sig ,"sc_sig");
    sca_util::sca_trace_file* atf1 =
sca_util::sca_create_vcd_trace_file("ams_vcd_trace.vcd");
//2
    sca_util::sca_trace(atf1, sig1 ,"sig1");
    sca_util::sca_trace_file* atf2 =
sca_util::sca_create_tabular_trace_file("ams_trace.dat");
//3
    sca_util::sca_trace(atf2, sig2 ,"sig2");
    sc_core::sc_start(2.0, sc_core::SC_MS); //4
    atf2->reopen("ams_trace.dat"); //5
    sc_core::sc_start(2.0, sc_core::SC_MS);
    atf2->disable();//6
    sc_core::sc_start(2.0, sc_core::SC_MS);
    atf2->enable(); //7
    atf2->set_mode( sca_util::sca_decimation(2) );
```

```
    sc_core::sc_start(2.0, sc_core::SC_MS);
    atf2->reopen("ams_trace3.dat"); //8
    sca_core::sca_time tstep(1.0, sc_core::SC_MS);//9
    atf2->set_mode( sca_util::sca_sampling( tstep,
sc_core::SC_ZERO_TIME ) );
    sc_core::sc_start(10.0, sc_core::SC_MS);
    sc_core::sc_close_vcd_trace_file(tf); //10
    sca_util::sca_close_vcd_trace_file(atf1);
    sca_util::sca_close_tabular_trace_file(atf2);
    return 0;
    }
```

1. Сигналы трассировки с использованием стандартного средства трассировки SystemC. Имейте в виду, что в случае AMS (например, TDF) сигналы отслеживаются, они автоматически преобразуются в дискретные сигналы событий с использованием преобразователя TDF портов, что влияет на точность синхронизации записанных образцов.
2. Трассировка сигналов AMS в файл в формате VCD с использованием функции трассировки расширений AMS.
3. Трассировка сигналов AMS в файл в табличном формате с использованием функции трассировки расширений AMS.
4. Начать моделирование во временной области. Сигналы «sig1» и «sig2» будут отслежены.
5. Закройте текущий файл трассировки и начните трассировку для нового файла (с тем же именем).
6. Отключите трассировку для atf2, чтобы не записывать следующие 2 мс.
7. Снова включите трассировку для atf2, но с другим периодом выборки, определяемым коэффициентом прореживания 2 (пропустить один образец).
8. Закройте текущий файл трассировки atf2 и начните трассировку до нового файла, используя другой временной шаг.
9. Определите, как образцы записываются в файл трассировки. Выборка каждые 1 мс, начиная с 0 мс.
10. Закройте все файлы трассировки.

7. Моделирование стратегий

Расширения SystemC AMS предоставляют разработчикам мощные инструменты для моделирования систем с аналоговым и смешанным сигналом. Расширения охватывают случаи использования, описанные в главе 1, предоставляя модели вычислений: временной поток данных (TDF), линейный поток сигналов (LSF) и электрические линейные сети (ELN), в дополнение к подходам SystemC к дискретным событиям и к моделированию уровня транзакций. Эта глава даёт дополнительные советы о том, как использовать и комбинировать эти модели вычислений эффективным способом. Представленные стратегии не являются обязательными, иногда могут быть другие или лучшие подходы. Они предоставляются для того, чтобы направлять неопытного пользователя и помочь ему создать свои первые модели, достичь высоких результатов моделирования, и повысить производительность при проектировании смешанных аналоговых / цифровых систем.

7.1. Поведенческое моделирование с использованием доступных моделей вычислений

Модели вычислений, предоставляемые расширениями SystemC AMS, в первую очередь предназначены для облегчения поведенческого моделирования аналоговых цепей, а также функций обработки сигналов (независимо от того, будут они реализованы в аналоговой или цифровой области). В зависимости от требуемой абстракции, подходящая модель расчёта для поведенческого моделирования должна быть выбрана. Рисунок 7.1 даёт обзор доступных моделей вычислений и обусловленные ими абстракции с учётом аспектов поведения, структуры, коммуникаций и соотношения время / частота.

Model of Computation	Imposed abstractions			
	Behavior	Structure	Communication	Time/Frequency
Timed Data Flow (TDF)	Algorithmic descriptions, transfer functions	Functional blocks (non-conservative system)	Sequence of samples of arbitrary type	(Over)sampling, baseband modeling
Linear Signal Flow (LSF)	Linear functional descriptions	Structural representation of linear equations (non-conservative system)	Directed signals, continuous value	No abstraction (continuous time)
Electrical Linear Networks (ELN)	Macro modeling with linear primitives and ideal switches	Simplified network (conservative system)	No abstraction (physical quantities)	No abstraction (continuous time)

Figure 7.1. Abstractions imposed by the AMS models of computation

Рисунок 7.1. Абстракции, навязанные AMS моделями вычислений

Наиболее важные абстракции, налагаемые моделями вычислений:

• Линеаризация нелинейного поведения из-за акцента на линейное поведение в моделях вычислений, которые требуют решения систем уравнений

(LSF, ELN).

• Абстракция к функциональным блокам (неконсервативным системам) с направленными сигналами в моделях вычисление LSF и TDF. Эта абстракция заменяет физические величины (i (t), u (t)) абстрактными количество x (т).

• Выборка сигналов непрерывного времени в сигналы с дискретным временем для модели вычисления TDF.

На рисунке 7.2 показано влияние абстракции и выборки на неконсервативное поведение сигнала в электрической сети.

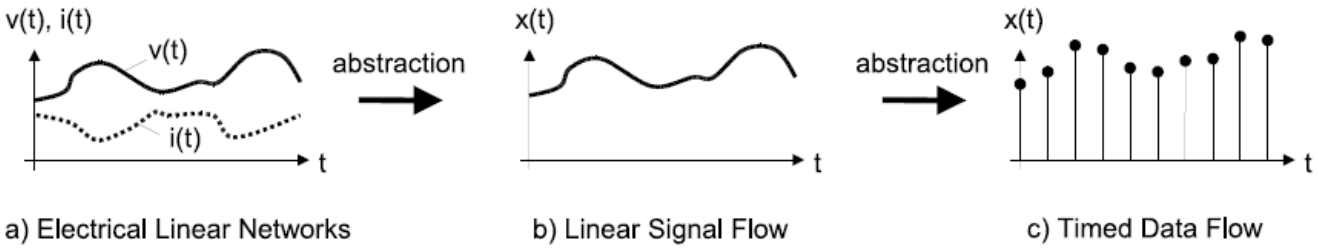


Figure 7.2. Abstraction of communication and time using the AMS models of computation

Рисунок 7.2. Абстракция коммуникаций и времени с использованием моделей вычислений AMS

В следующих подразделах дано краткое и общее описание возможностей каждой модели вычислений. При использовании нескольких моделей вычислений важно правильно их комбинировать. Обязательное разделение функциональности на различные модели вычислений обсуждается в разделе 7.2.1.

7.1.1. Макромоделирование с помощью электрических линейных сетей

Модель вычислений ELN допускает макромоделирование: точные физические устройства, такие как транзисторы, моделируется простыми электрическими примитивами, такими как (идеальные) переключатели, источники напряжения и другие электрические линейные примитивы. Цель состоит в том, чтобы определить абстрактную модель с упрощённым поведением. Для учета сигналов и интерфейсов, абстракция не требуется. Модель вычисления ELN должна использоваться в следующих случаях:

1. Описание систем, в которых нелегко или неестественно дать уравнения, например, модели линий электропередачи, или почти линейные нагрузки, которые переключаются в пределах рабочего цикла.

2. Аналоговые интерфейсы и компоненты, которые имеют отношение к размерам системы или ее общему поведению. Поэтому они должны отображаться на системном уровне.

Для настройки макромодели ELN поведение электрической цепи должно быть линеаризовано. Доступность переключателей в дополнение к линейным компонентам позволяет осуществлять переключение между различными

операциями режимов или включение / выключение нагрузки. Следующая стратегия может быть полезна для получения модели ELN от данной схемы:

1. Определите участки схемы с почти линейным поведением и смоделируйте их, используя компоненты ELN. Компоненты, которые не требуются для общей функциональности (например, фиксирующие диоды), могут быть опущены.

2. Определите переключающие компоненты и замените их идеальными переключателями.

3. В зависимости от предполагаемой среды модели:

• Если применяется как часть ELN, моделируйте входной и выходной импедансы.

• Если применяется как часть TDF или дискретного события, используйте соответствующие элементы преобразователя.

Обратите внимание, что модель вычисления ELN не поддерживает моделирование нелинейного ограничения или последствия насыщения. Рекомендуется разделить модель так, чтобы нелинейные эффекты моделировались с использованием модели TDF вычисления.

На рисунке 7.3 показан пример драйвера питания с использованием широтно-импульсной модуляции (ШИМ). Оригинальная схема показана на рисунке 7.3a. Для применения макромоделирования ELN фиксирующие диоды опущены в предположении, что сама схема была проверена с использованием симулятора цепи. КМОП транзисторы, которые переключают нагрузку (катушку с сопротивлением 10 Ом), заменяются идеальными выключателями. Полученная макромодель ELN показана на рисунке 7.3b.

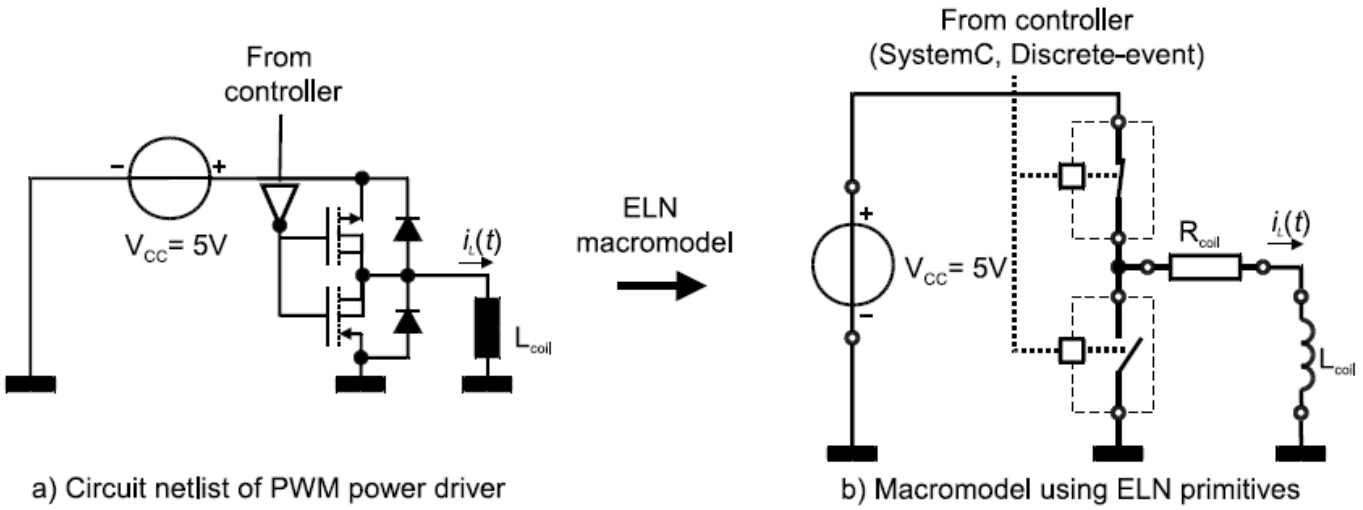


Figure 7.3. Abstraction of PWM power driver into an ELN macromodel

Рисунок 7.3. Абстракция драйвера питания ШИМ в макромодель ELN

Драйвер ШИМ вместе со своей нагрузкой работает как фильтр нижних частот для тока нагрузки I_L (t). Как таковой, он также может быть смоделирован как функциональный блок. Однако сама нагрузка обычно является внешней частью, и, таким образом, может быть изменена пользователем. Поэтому имеет

смысл предоставить электрический терминал и (линейную) макромодель нагрузки. В следующем примере кода показана модель ELN драйвера ШИМ.

```
    SC_MODULE(pwm_driver)
    {
    sc_core::sc_in<bool> in;
    sca_eln::sca_terminal out;
    sca_eln::sca_vsource vcc; // voltage source
    sca_eln::sca_de::sca_rswitch highside, lowside; //
two switches
    pwm_driver( sc_core::sc_module_name nm, double vcc_ =
5.0)
    : in("in"), out("out"),
    vcc("vcc"), highside("highside"), lowside("lowside"),
node("node"), gnd("gnd")
    {
    vcc.offset = vcc_; // usage as constant voltage
source
    vcc.p(node);
    vcc.n(gnd);
    highside.ctrl(in); // 1st switch
    highside.p(node);
    highside.n(out);
    lowside.ctrl(in); // 2nd switch...
    lowside.p(out);
    lowside.n(gnd);
    lowside.off_state = true; // ...is inverted
    }
    private:
    sca_eln::sca_node node;
    sca_eln::sca_node_ref gnd;
    };
```

Нагрузка также может быть легко описана с использованием линейных примитивов, в самом простом случае - катушки с некоторым сопротивлением может быть достаточно:

```
    SC_MODULE(load)
    {
    sca_eln::sca_terminal p, n;
    sca_eln::sca_r r;
    sca_eln::sca_l l;
    load( sc_core::sc_module_name nm,
    double res_ = 500.0, double ind_ = 0.000001 )
    : p("p"), n("n"), r("r", res_ ), l("l", ind_ ),
node("node")
```

```
{
r.p(p);
r.n(node);
l.p(node);
l.n(n);
}
private:
sca_eln::sca_node node;
};
```

7.1.2. Поведенческое моделирование с помощью линейного потока сигналов

Модель вычисления LSF позволяет получить описание блок-схем для вычисления линейных дифференциальных уравнений. По сравнению с передаточными функциями LSF позволяет указать порядок вычислений и получить доступ к промежуточным результатам или коэффициентам. В частности, LSF полезен для того, чтобы:

1. Моделировать фильтры с заданной структурой, которая, например, оказывает влияние на шум.

2. Моделировать системы непрерывного контроля, в частности те, которые требуют доступа к коэффициентам от других моделей вычислений.

Для LSF требуется абстракция физических сигналов, как описано в главе 3. В частности, это также требует абстрагирования коммуникаций с направленными сигналами. Учитывая структуру и поведение, функциональные блоки должны быть идентифицированы, а их поведение должно быть описано путём создания предопределённых функциональных примитивов. Рассматривая время, абстракция не требуется.

Обратите внимание, что LSF не предоставляет средства для указания нелинейного ограничения или насыщения. Рекомендуется разделить модель таким образом, чтобы нелинейные эффекты, если необходимо, определялись с использованием модели TDF вычислений. Типичный пример применения, где LSF полезен, показан на рисунке 7.4. Этот ПИД-регулятор может быть частью модели системы управления с обратной связью. Его коэффициенты могут быть скорректированы с помощью модели TDF.

Чтобы без задержки смоделировать систему управления с обратной связью, само устройство также должно быть смоделировано с использованием модель вычисления LSF. Использование любой другой модели вычислений (ELN, TDF) приведёт к задержке в контуре управления.

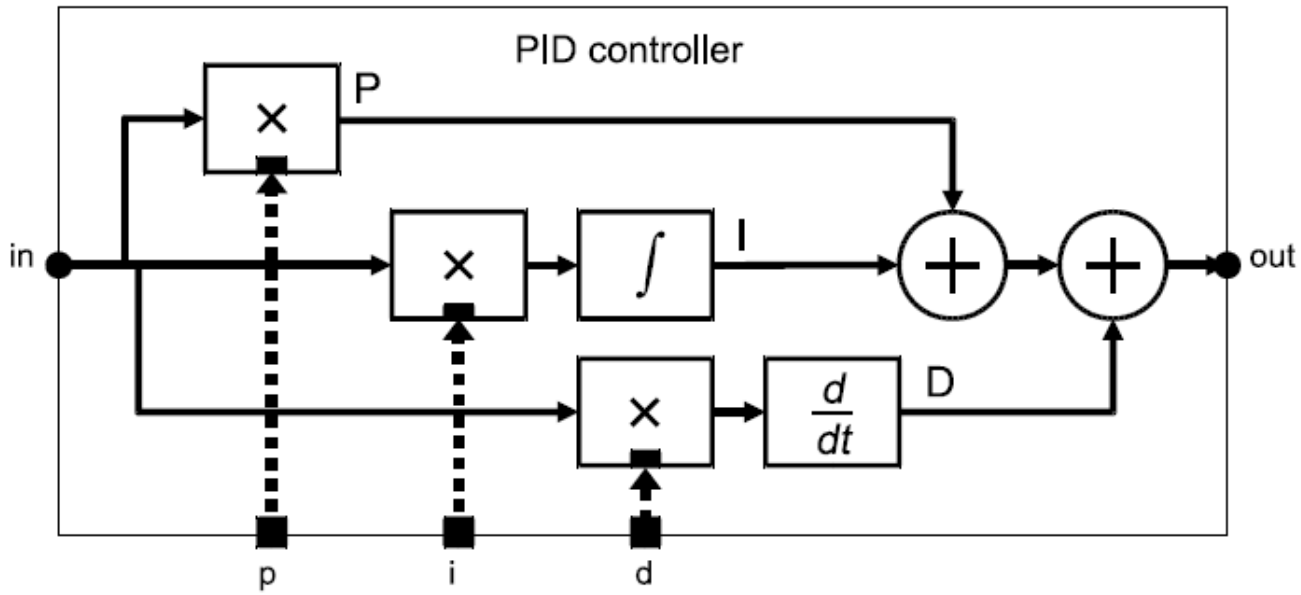


Figure 7.4. LSF model of a PID controller with adjustable coefficients

Рисунок 7.4. LSF модель ПИД-регулятора с регулируемыми коэффициентами

```
    SC_MODULE(lsf_pid_external_control)
    {
    sca_lsf::sca_in in;
    sca_lsf::sca_out out;
    sca_tdf::sca_in<double> p, i, d; // adjustable
coefficients
    sca_lsf::sca_tdf::sca_gain gain_p, gain_i, gain_d; //
coefficients used to scale the gain
    sca_lsf::sca_integ integ;
    sca_lsf::sca_dot dot;
    sca_lsf::sca_add add1, add2;
    lsf_pid_external_control( sc_core::sc_module_name
name )
    : in("in"), out("out"), p("p"), i("i"), d("d"),
    gain_p("gain_p"), gain_i("gain_i"), gain_d("gain_d"),
    integ("integ"), dot("dot"), add1("add1"),
add2("add2"),
    sig_gain("sig_gain"), sig_integ1("sig_integ1"),
sig_integ2("sig_integ2"),
    sig_dot1("sig_dot1"), sig_dot2("sig_dot2"),
    sig_add("sig_add")
    {
    gain_p.x(in);
    gain_p.y(sig_gain);
```

```
    gain_p.inp(p);
    gain_i.x(in);
    gain_i.y(sig_integ1);
    gain_i.inp(i);
    gain_d.x(in);
    gain_d.y(sig_dot1);
    gain_d.inp(d);
    integ.x(sig_integ1);
    integ.y(sig_integ2);
    dot.x(sig_dot1);
    dot.y(sig_dot2);
    add1.x1(sig_gain);
    add1.x2(sig_integ2);
    add1.y(sig_add);
    add2.x1(sig_add);
    add2.x2(sig_dot2);
    add2.y(out);
    }

    private:
    sca_lsf::sca_signal sig_gain, sig_integ1, sig_integ2,
sig_dot1, sig_dot2, sig_add;
    };
```

7.1.3. Поведенческое и базовое моделирование с синхронизированным потоком данных

Модель вычисления TDF позволяет моделировать аналоговые системы на высоком уровне абстракции, а также моделирование функций обработки сигналов.

Для моделирования аналогового поведения модель вычисления TDF требует аппроксимации в дискретном времени аналоговых сигналов непрерывного времени. Тем не менее, TDF позволяет, в отличие от LSF и ELN, проводить моделирование нелинейного поведения. Аппроксимация в дискретном времени уменьшает непрерывный сигнал до последовательности дискретных образцов. Эта абстракция устраняет необходимость решения (нелинейных) уравнений и, таким образом, улучшает производительность моделирования. Помимо дискретизации, модель вычисления TDF также требует нарушения циклических зависимостей (также известных как алгебраические циклы) путём вставки задержек (см. раздел 2.1.2).

В обмен на эти абстракции модели TDF позволяют описывать обработку потоков выборок произвольным алгоритмическим способом с помощью функции-члена **processing**. В частности, также нелинейные передаточные функции (т.е. для ограничения моделирования) или справочные таблицы могут быть легко реализованы. Более того, спецификация методов обработки

сигналов в терминах передаточных функций H (s), H (z) или представления пространства состояний поддерживаются в TDF (см. раздел 2.3.2).

Следующие абстракции представлены в TDF:

1. Как и в LSF, необходимо определить структуру блок-схемы. В отличие от LSF, практически нет ограничений к поведению отдельных блоков.
2. Частота дискретизации должна быть определена.
3. Модель TDF требует ациклических структур для обеспечения планируемости. Ациклическая структура может быть достигнута путём введения задержки в цикл (раздел 2.1.2). Обратите внимание, что большинство контрольных циклов используют в настоящее время цифровые контроллеры, которые так или иначе вводят задержки. Расположение цифрового контроллера может быть хорошим местом для введения такой задержки.

Определение и распространение временных шагов и скоростей

Временные шаги и скорости в TDF должны выбираться тщательно, чтобы соответствовать задаче моделирования. Также рекомендуется тщательно выбирать места, где определены временные шаги и скорости.

Для моделирования аналогового поведения рекомендуется обеспечить достаточно высокую частоту дискретизации. Частота дискретизации должна быть значительно выше, чем удвоенная частота, определяемая наименьшей постоянной временем в системе. В сомнительных случаях, фактор 10 рекомендуется. Выбор более высокой скорости или меньших временных шагов приводит к более высокой точности на высоких частотах за счёт производительности моделирования. Подходящим местом для определения временного шага может быть испытательный стенд.

Системы с постоянными времени, которые отличаются на порядки величин (жёсткие системы), представляют собой особую проблему.

Мы рекомендуем разбивать такие системы на части с низкими постоянными времени и части с более высокими временными константами. Затем разные скорости вычисления TDF-модели могут использоваться для определения различных выборок частоты в каждом разделе.

Моделирования методов цифровой обработки сигналов (DSP) (например, с использованием H (z) или представлений цифровых фильтров в пространстве состояний) приводит к зависимости между функциональностью и выбранным временным шагом. Для методов DSP, которые предназначены для использования на определённой частоте выборки, рекомендуется определить временной шаг в самом модуль (или в его портах соответственно). Обратите внимание, что тестовый стенд все ещё может определять временные шаги. Тем не менее, об ошибке будет сообщено, если проверка согласованности после распространения временных шагов не удалась (см. раздел 2.5).

Поведенческое моделирование с TDF

В разделе 2.6 приводятся два примера применения, представляющих

поведенческое моделирование с использованием модели TDF вычисления. Обратите внимание, что расширения SystemC AMS позволяют записывать произвольный код C ++ в функции-члене **processing** модуля TDF. Это позволяет комбинировать идеальные функции обработки сигналов (обычно можно найти в библиотеках блоков), таких как функции усиления, умножения или передачи очень эффективным способом с неидеальным поведением. Например, усилитель можно моделировать, комбинируя следующие функции:

1. Его поведение в частотной области может быть смоделировано с использованием передаточной функции Лапласа, как обсуждалось в разделе 2.3.2. Полясы и нули можно легко идентифицировать с помощью схемотехнического моделирования или графика Боде.

2. Поведение большого сигнала (например, ограничение, нелинейность) может быть смоделировано с использованием кода C ++.

```
    SCA_TDF_MODULE(amplifier)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    amplifier( sc_core::sc_module_name, double gain_ =
100.0,
    double dom_pole_ = 5.0e8,
    double limit_ = 5.0 )
    : in("in"), out("out"), gain(gain_),
dom_pole(dom_pole_), limit(limit_) {}
    void initialize()
    {
    // filter requires no zeros to be defined
    poles(0) = sca_util::sca_complex( -2.0 * M_PI *
dom_pole, 0.0 );
    k = gain * 2.0 * M_PI * dom_pole;
    }
    void processing()
    {
    // time-domain implementation of amplifier behavior
as function of frequency
    double internal = ltf_zp( zeros, poles, state,
in.read(), k );
    // limiting the signal
    if (internal > limit) internal = limit;
    else if (internal < -limit) internal = -limit;
    out.write(internal);
    }
    private:
    double gain; // DC gain
    double dom_pole; // 3dB cutoff frequency in Hz
```

```
    double limit; // limiter value
    double k; // filter gain
    sca_tdf::sca_ltf_zp ltf_zp; // Laplace transfer
function
    sca_util::sca_vector<sca_util::sca_complex > poles,
zeros; // poles and zeros as complex values
    sca_util::sca_vector<double> state; // state vector
    };
```

Моделирование модулирующей полосы с TDF

При моделировании радиочастотных (РЧ) систем с высокими несущими частотами значительное ускорение моделирования может быть достигнуто путём применения *модулирующей полосы* (полосы модулирующих частот). Эта стратегия моделирования основана на том факте, что методы цифровой модуляции используют амплитуду r и фазу f для передачи информации. Информация сама по себе тогда не зависит от (обычно высокой) несущей частоты. Идея моделирования модулирующей полосы состоит в том, чтобы отобразить несущую РЧ на нулевую частоту, как показано на рисунке 7.5. Требуемая частота дискретизации зависит только от полосы пропускания модулированного сигнала.

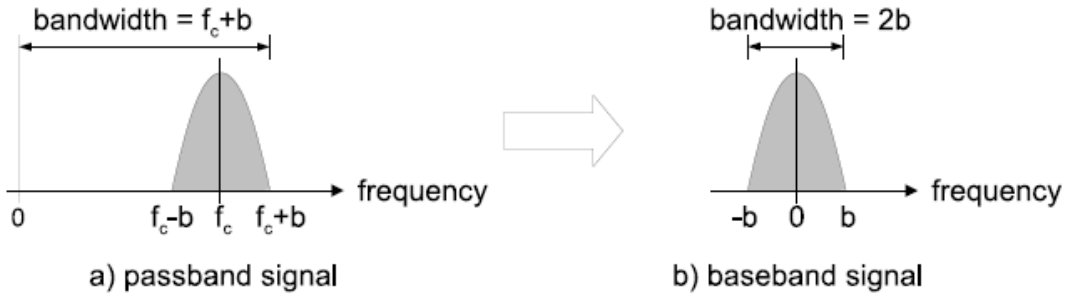


Figure 7.5. Passband (a) and baseband (b) representation of signals in the frequency domain

Рисунок 7.5. Полоса пропускания (a) и модулирующая полоса (b) представления сигналов в частотной области

Формально модулированный сигнал несущей x (т) может быть описан как:

$$\begin{aligned} x(t) &= r(t)\cos\left(2\pi f_c t + \varphi(t)\right) \\ &= \mathrm{Re}\left\{r(t)\, e^{j\left(2\pi f_c t + \varphi(t)\right)}\right\} \\ &= \mathrm{Re}\left\{r(t)\, e^{j\,\varphi(t)} e^{j\,2\pi f_c t}\right\} \end{aligned}$$

где $r(t)$ является модулирующим сигналом, $\varphi(t)$ модулированной фазой и f_c несущей частоты. Термин, который включает в себя несущую частоту f_c, может быть отделен от части сигнала, которая содержит передаваемую информацию. Сигнал $v(t)$, который содержит информацию, не зависит от

несущей частоты f_c:

$$v(t) = r(t)e^{j\varphi(t)}$$

$$v(t) = r(t)\, e^{j\,\varphi(t)}$$

Этот сигнал называется *комплексным эквивалентом нижних частот* или *комплексной огибающей*. Для сигнала модулирующей полосы несущая частота f_c установлена на ноль. При $s_i = r\cos\phi$ и $s_q = r\sin\phi$ результирующий сигнал модулирующей полосы становится:

$$v(t) = s_i(t) + js_q(t)$$

$$v(t) = s_i(t) + j\, s_q(t)$$

где $s_i(t)$ представляет синфазный член сигнала основной полосы частот, а $s_q(t)$ представляет квадратурный член.

Амплитуда и фаза сигнала несущей могут быть вычислены из этих сигналов в каждый момент времени.

Чтобы использовать эти сигналы модулирующей полосы частот, необходим специальный тип данных, который поддерживает определение комплексных значений. Расширения SystemC AMS предлагают класс sca_util :: sca_complex, который может быть использован для этой цели. Эти сигналы могут использоваться в модулях TDF, в которых типы значения портов TDF изменены со скалярных значений (типа double) на комплексные значения, как показано в примере ниже. Используется функция std :: pow (c, y) из стандартного библиотечного комплекса C ++, которая вычисляет с возводением в степень y, где c является комплексным значением.

```
    #include <complex>
    SCA_TDF_MODULE(baseband_amplifier)
    {
    sca_tdf::sca_in< sca_util::sca_complex > in;
    sca_tdf::sca_out< sca_util::sca_complex > out;
    baseband_amplifier( sc_core::sc_module_name, double
gain = 1.0, double iip3 = 1e-3 )
    : in("in"), out("out"), a1( gain ), a3( -4/3 * ( gain
/ std::pow(iip3,2)) ) {}
    void processing()
    {
    out.write( a1 * in.read() + a3 * std::pow(
in.read(),3 ) );
    }
    private:
    double a1, a3;
```

```
};
```

Ограничение использования sca_util :: sca_complex в качестве типа данных заключается в том, что оно описывает только сложную оболочку модулированного сигнала, и что информация о несущей частоте теряется. Благодаря этому, такие эффекты, как гармоники несущей или продуктов интермодуляции не представлены, так как они выходят за пределы полосы пропускания сигнала.

Решением этой проблемы является создание пользовательского типа данных, аналогичного sca_util :: sca_complex, который поддерживает расчеты в основной полосе частот с несколькими несущими.

7.2. Моделирование встроенных аналоговых / смешанных сигнальных систем

Поведенческое моделирование с использованием единой модели вычислений накладывает ряд ограничений, как показано на рисунке 7.1. Их можно преодолеть, комбинируя (сильные стороны) различных моделей вычислений.

В следующих подразделах описывается, как разделить функциональное поведение на различные модели вычислений. Затем даётся ряд простых рекомендаций по моделированию, как моделировать свойства уровень архитектуры аналоговых цепей.

7.2.1. Поведение разбиения на разные модели вычислений

Простая, но общая стратегия, которая позволяет новичкам распространять структурную схему, такую как спецификация, на различные модели вычислений, предоставляемые расширениями SystemC AMS, показаны на рисунке 7.6. Она может применяться для каждого блока последовательно.

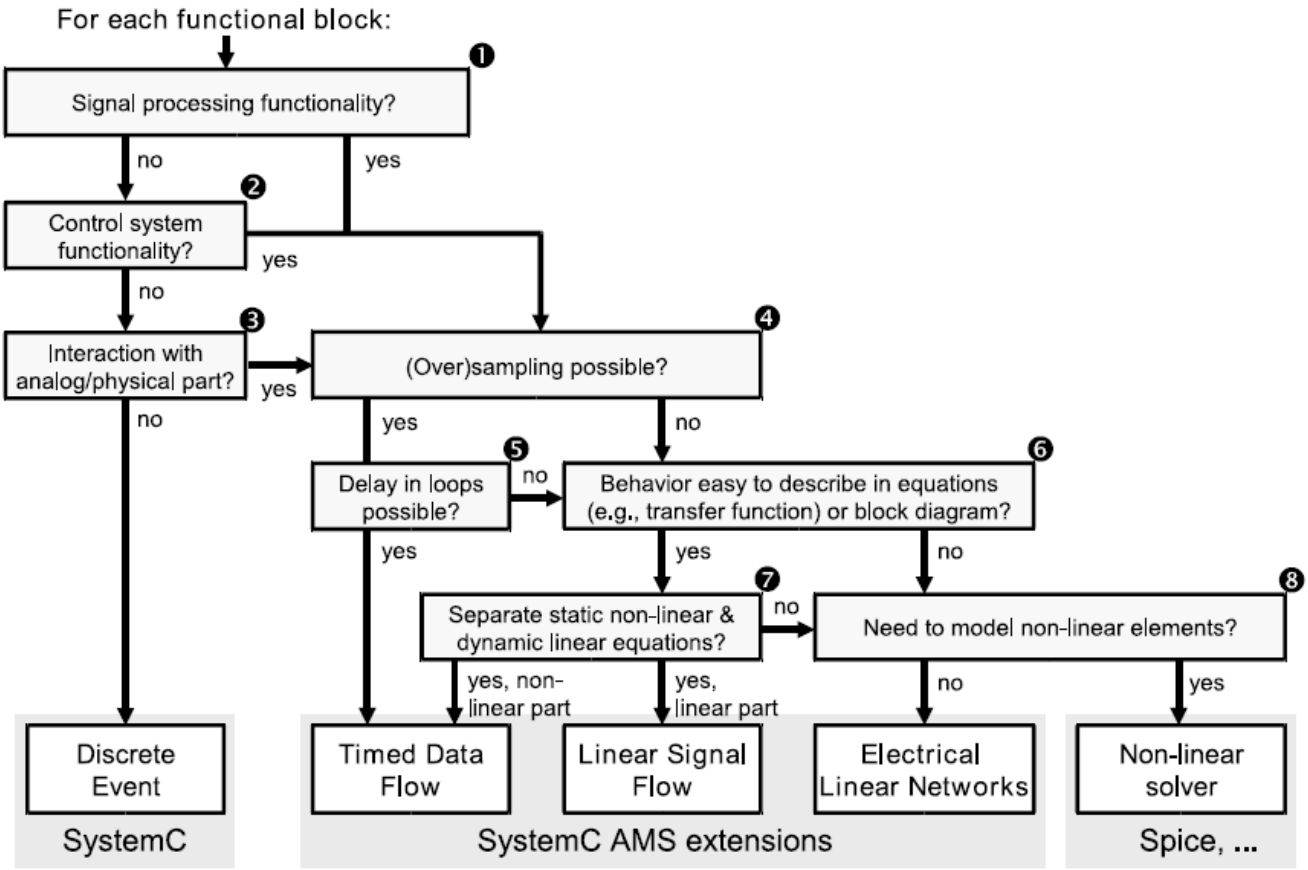


Figure 7.6. Partitioning of behavior to different models of computation

Рисунок 7.6. Разделение поведения на различные модели вычислений

На первом этапе (метки 1, 2 и 3) следует выяснить, является ли модель вычисления дискретного события подходящей или расширения AMS являются лучшим выбором. Расширения SystemC AMS это - лучший выбор для моделирования функций обработки сигналов. Обратите внимание, что функции обработки сигналов, которые реализованные в цифровом или программном обеспечении, также могут быть эффективно смоделированы с использованием расширений SystemC AMS на функциональном уровне. Если должно быть смоделировано конкретное сопоставление с оборудованием на уровне архитектуры или ниже, SystemC более уместно. Ещё одна веская причина для использования расширений AMS - необходимость наличия аналоговых терминалов и / или физические величины, такие как доступный ток, например, для моделирования внешних нагрузок или аналогового поведения линий связи.

На втором шаге (метки 4 и 5) следует рассмотреть модель вычислений с временным потоком данных, чтобы моделировать подсистему AMS. Это требует дискретного моделирования аналоговых сигналов и (в случае циклического зависимости) вставки дополнительных задержек. Этот вариант предлагает большинство вариантов для спецификации поведения аналоговой и сигнальной обработки.

Если дискретное приближение не подходит (метки 6, 7 и 8), необходимо рассмотреть модели вычислений LSF и ELN. Оба полагаются на линейный решатель. Следовательно, поведение должно быть разбито на линейную и нелинейную функциональность, где последняя может быть реализована с использованием TDF. Если требуется точное моделирование нелинейного консервативного поведения или электрической сети, следует рассмотреть возможность использования соответствующего нелинейного решателя или симулятора схемы, возможно, в сочетании с SystemC.

7.2.2. Моделирование свойств уровня архитектуры

Для оценки осуществимости и производительности различных архитектур функциональная модель может быть используется и уточняется путём добавления определённых свойств. Эти свойства включают в себя: шум, затухание, искажения, ограничение, дрожание, задержки, квантование, частоты дискретизации и многое другое. Далее даны некоторые простые рекомендации по обработке этих эффектов во время исследования архитектуры.

Моделирование искажений, ограничений и квантования

Для изучения влияния искажений и ограничений на общую функциональность системы, аналоговые модули следует разделить на линейное динамическое поведение и нелинейное статическое поведение. Линейное динамическое поведение может быть описано, например, с использованием передаточных функций в TDF (см. раздел 2.3.2). Нелинейное поведение, такое как искажения и ограничение может быть легко смоделировано с использованием функций C ++ в модуля TDF функцией-членом **processing** (см. раздел 7.1.3).

Моделирование шума во временной области

Шум в модели вычисления TDF может быть смоделирован путём добавления гауссовских распределенных случайных чисел к сигналу TDF. В следующем примере демонстрируется простая модель (белого) шума и затухания в беспроводной линии связи. Для этого используется функция gauss_rand, которая генерирует распределённые по Гауссу случайные числа.

```
    // the gauss_rand() function returns a gaussian
distributed
    // random number with variance "variance", centered
around 0, using the Marsaglia polar method
    #include <cstdlib> // for std::rand
    #include <cmath> // for std::sqrt and std::log
    double gauss_rand(double variance)
    {
    double rnd1, rnd2, Q, Q1, Q2;
    do
```

```
    {
    rnd1 = ((double)std::rand()) / ((double)RAND_MAX) ;
    rnd2 = ((double)std::rand()) / ((double)RAND_MAX) ;
    Q1 = 2.0 * rnd1 - 1.0 ;
    Q2 = 2.0 * rnd2 - 1.0 ;
    Q = Q1 * Q1 + Q2 * Q2 ;
    } while (Q > 1.0) ;
    return ( std::sqrt(variance) *( std::sqrt( - 2.0 *
std::log(Q) / Q) * Q1) );
    }
    SCA_TDF_MODULE(air_channel_with_noise)
    {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;
    void processing()
    {
    out.write( in.read() * attenuation +
gauss_rand(variance) );
    }
    air_channel_with_noise( sc_core::sc_module_name nm,
    double attenuation_,
    double variance_ )
    : in("in"), out("out"), attenuation(attenuation_),
variance(variance_) {}
    private:
    double attenuation;
    double variance;
    };
```

Для получения цветного шума выходные данные функции gauss_rand могут быть отфильтрованы с использованием соответствующей функции передачи.

7.3. Уточнение дизайна и смешанное моделирование

7.3.1. Смешанный сигнал, смешанный уровень моделирования

Проектирование встраиваемых аналоговых / цифровых систем требует комбинации различных моделей вычислений и разных уровней абстракции. Это требует преобразования коммуникации / синхронизации на границах между различными моделями вычислений. Расширения SystemC AMS предоставляют базовый набор языковых примитивов, которые обеспечивают преобразование между SystemC (дискретное событие), TDF, ELN и LSF. В ELN и LSF, предоставляются модули преобразователя; в TDF доступны порты конвертера. Обратите внимание, что ELN и LSF может связываться с дискретным событием и TDF, но не друг с другом напрямую.

Рекомендуется моделировать общий поток сигналов системы, используя

модель вычисления TDF, если это возможно. Это имеет следующие преимущества:

1. Модель вычислений TDF обеспечивает преобразование во все другие модели вычислений.

2. Модель вычисления TDF необходима для предоставления временных шагов для подключённых компонентов ELN и LSF.

На рисунке 7.7 в качестве примера показана часть цепочки обработки сигналов: контроллер LSF (показан слева) передает свой выход через управляемый источник напряжения в фильтр нижних частот ELN. Для того, чтобы соединить ELN и LSF, сигнал LSF преобразуется в сигнал TDF, для которого должен быть задан временной шаг. Сигнал TDF контролирует (TDF) управляемый источник напряжения, который является частью модели ELN.

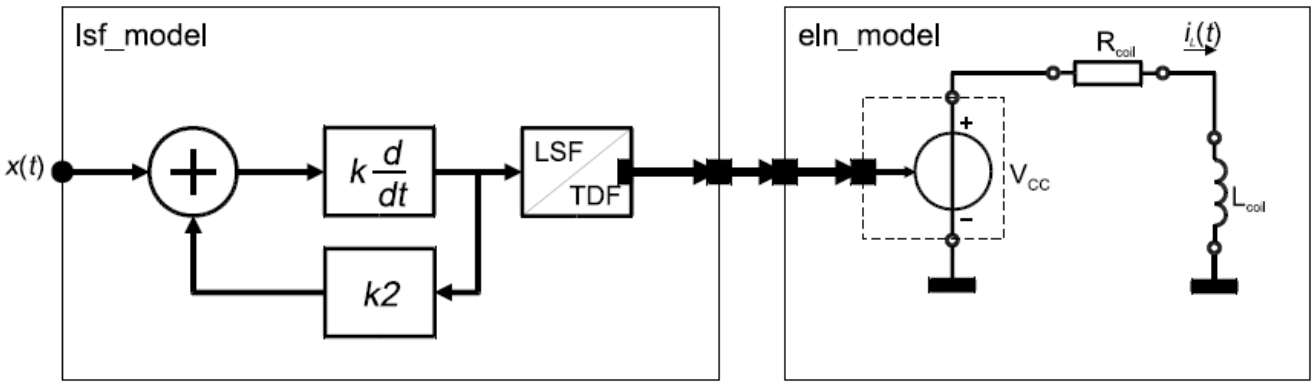


Figure 7.7. Coupling of LSF and ELN via an LSF/TDF converter module

Рисунок 7.7. Соединение LSF и ELN через модуль преобразования LSF / TDF

Имейте в виду, что преобразование из LSF (или ELN) в TDF, а затем в ELN (или LSF) приводит к задержке одного временного шага.

7.3.2. Уточнение дизайна и варианты использования

Для проектирования цифровых систем проектирование сверху вниз является современным. Интеграция аналоговых / смешанных сигнальных подсистем, которые в основном разработаны снизу вверх, в нисходящий потоке с цифровым доминированием по-прежнему проблемно. В Разделе 1.2.1 были представлены предполагаемые варианты использования расширений AMS. Этот раздел описывает, как применять расширения SystemC AMS для повышения эффективности и производительности в процессе проектирования встраиваемых аналоговых / цифровых систем. Это дополняет известный подход к уточнению из SystemC. На рис. 7.8 представлен обзор применения расширений SystemC AMS.

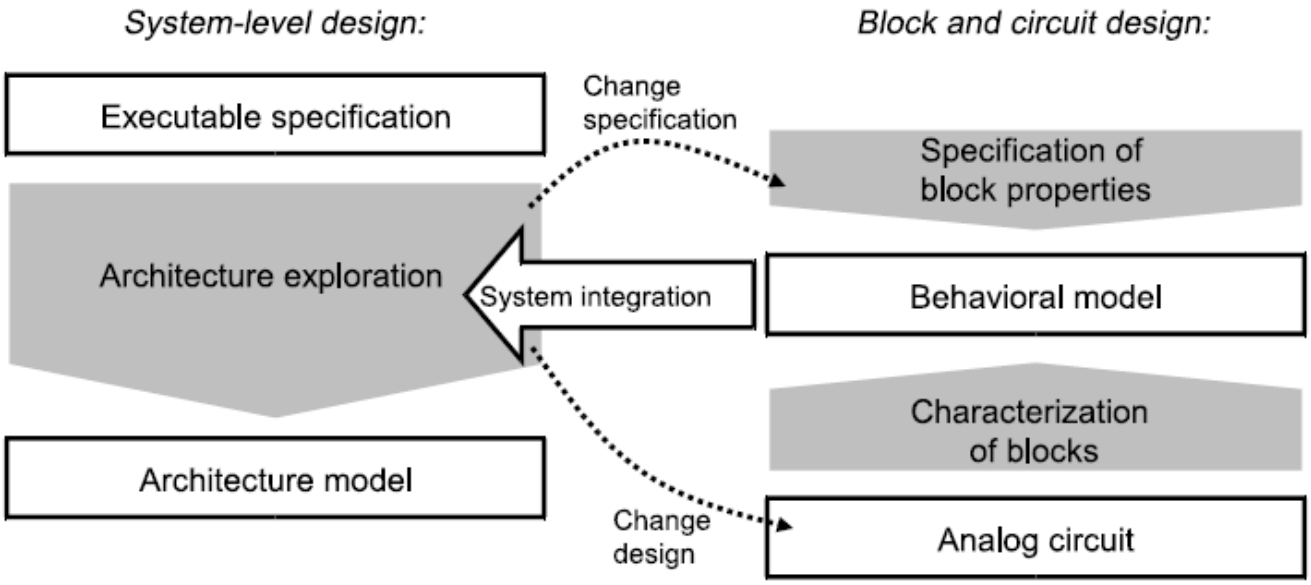


Figure 7.8. Use cases for the SystemC AMS extensions within top-down refinement

Рисунок 7.8. Варианты использования для расширений SystemC AMS при уточнении сверху вниз

В идеальном случае уточнение сверху вниз начинается с исполняемой спецификации предполагаемого поведения на системном уровне. Обычно модель вычислений TDF подходит для разработки для этой цели функциональной модели. Уточнение исполняемой спецификации является частью варианта использования для исследования архитектуры.

Процесс уточнения состоит из поэтапного подхода к замене блоков в системе на более точные (менее абстрактные) модели.

Архитектурное исследование различает три отдельных аспекта, каждый из которых является противоположностью одной из абстракции на рисунке 7.1:

- уточнение поведения
- уточнение структуры
- уточнение связи / интерфейсов

Поведенческое уточнение дополняет функциональную модель, используемую для исполняемой спецификации, конкретными свойствами архитектуры (реализация). Это позволяет оценить целесообразность и выполнение разных архитектур (реализаций). Свойства, которые могут быть легко включены в функциональную модель включает в себя: шум, затухание, искажения, ограничение, дрожание, задержки, квантование, выборку частоты и многое другое.

В качестве примера на рисунке 7.9 показан идеальный (линейный) и нелинейный усилитель, где линейное усиление (a_1) и нелинейный член (a_3) добавляются с полиномиальным представлением.

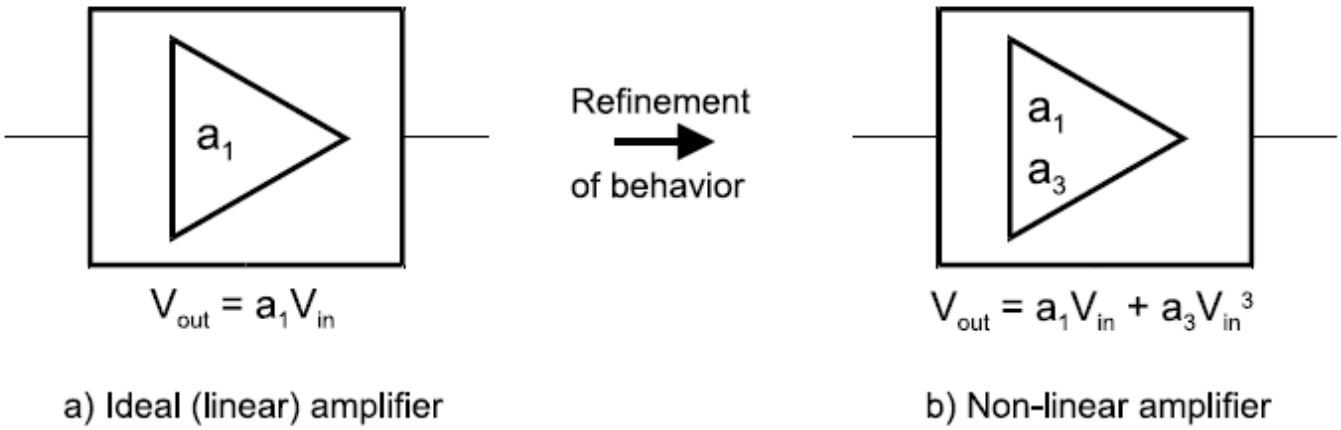


Figure 7.9. Refinement of behavior of an amplifier

Рисунок 7.9. Уточнение поведения усилителя

Пример кода ниже показывает, как может быть реализовано нелинейное поведение. Используется функция std :: pow (x, y) из стандартной библиотеки C ++ cmath, которая вычисляет x, возведенное в степень y.

```
#include <cmath>
SCA_TDF_MODULE(non_linear_amplifier)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<double> out;
non_linear_amplifier( sc_core::sc_module_name, double
gain = 1.0, double iip3 = 1e-3 )
: in("in"), out("out"), a1( gain ), a3( -4/3 * ( gain
/ std::pow(iip3,2)) ) {}
void processing()
{
out.write( a1 * in.read() + a3 *
std::pow(in.read(),3) );
}
private:
double a1, a3;
};
```

Уточнение структуры перераспределяет систему (обычно похожую на блок-схему), используемую для исполняемого файла спецификации со структурой функциональных блоков, каждый из которых представляет схему или процессор, который должен быть спроектирован.

Обратите внимание, что модель расчёта также изменяется в зависимости от предполагаемой области реализации.

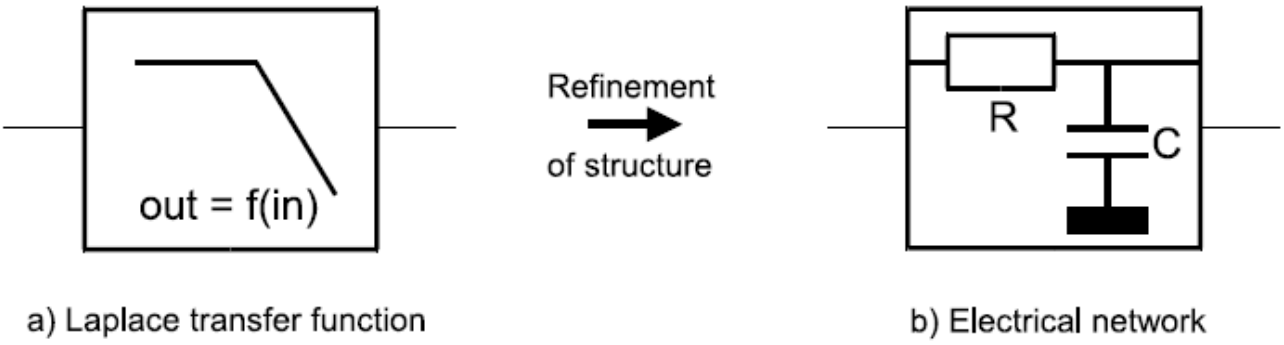


Figure 7.10. Structural refinement of a filter

Рисунок 7.10. Структурная обработка фильтра

Чтобы упростить уточнение модели, концепция пространства имён позволяет повторно использовать большую часть существующей инфраструктуры моделирования, такую как объявления модулей и портов. Тем не менее, поведение и (уточнённая) структура должна быть написана с нуля.

Уточнение коммуникаций/ интерфейсов заменяет абстрактную связь, используемую в модели TDF вычисления с конкретными сигналами, например, электрическими напряжениями и токами или цифровыми (дискретное событие) SystemC сигналами. Для этого также необходимо добавить порты преобразователя или модули в модели. Конверсия между моделями вычисления обсуждаются в разделе 7.3.1. Для поддержки улучшения коммуникации/ интерфейсов, рекомендуется создавать классы адаптера / конвертера, как это известно из расширений SystemC TLM.

7.4. Стиль моделирования и кодирования

7.4.1. Пространства имен

Расширения SystemC AMS широко используют пространства имен C ++, чтобы иметь возможность четко идентифицировать доступные модели вычислений и использовать доступные примитивные модули в нужном контексте. Пространства имен sca_tdf, sca_lsf и sca_eln являются зарезервированными именами для языковых конструкций, используемых для TDF, LSF и ELN модели расчёта соответственно. Другие зарезервированные пространства имён: sca_util для служебных классов и функции, и sca_ac_analysis для анализа частотной области слабого сигнала. Пользователь не должен добавлять новые определения в этих пространствах имён. Вместо этого рекомендуется объявить пользовательские модули, принадлежащие к той же модели вычисления для уникального пользовательского пространства имён, как показано в примере ниже.

```
namespace my_tdf {
SCA_TDF_MODULE(my_source)
{
...
```

```
}
}; // namespace my_tdf
```

Реализация этого объекта будет выглядеть так:

```
SC_MODULE(analog_top)
{
...
my_tdf::my_source i_my_source("i_my_source");
...
}
```

Заголовочные файлы и соглашения об именах

Заголовочный файл <systemc-ams> не импортирует зарезервированные пространства имен sca_tdf, sca_lsf, sca_eln, sca_util и sca_ac_analysis в рамках программы. Это означает, что пользователь должен явно добавить идентификатор пространства имён для каждого элемента при создании или объявлении такого объекта. Хотя имена немного дольше писать, это приведёт к чёткому соглашению об именах, где пользователь может сразу же распознать принадлежит ли объект к определённой библиотеке классов расширений SystemC AMS, или объект является частью пользовательской библиотеки. Пример ниже и предыдущие примеры, приведённые в этом руководстве следует этому соглашению об именах.

```
#include <systemc-ams>
#include "my_source.h"
int sc_main(int argc, char* argv[])
{
sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);
sca_tdf::sca_signal<double> sig1;
// instantiate user-defined module from user-defined
'my_tdf' namespace
my_tdf::my_source i_my_source("i_my_source");
i_my_source.out(sig1);
// instantiate other modules
...
// tracing AMS signals
sca_util::sca_trace_file* tf =
sca_util::sca_create_tabular_trace_file("trace.dat");
sca_util::sca_trace(tf, sig1 ,"sig1");
sc_core::sc_start(10.0, sc_core::SC_MS);
tf-
>set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RA
D));
sca_ac_analysis::sca_ac_start(1.0e3, 1.0e6, 4,
sca_ac_analysis::SCA_LOG);
sca_util::sca_close_tabular_trace_file(tf);
```

```
    return 0;
    }
```

При использовании заголовочного файла <systemc-ams.h> все элементы, которые принадлежат пространству имен sca_core, sca_util и sca_ac_analysis, импортируются в область действия программы. Это означает, что пользователь может опустить префикс элементов в этих пространствах имён. Обратите внимание, что пространство имён для различных моделей вычислений не объявляются, поэтому даже в этом случае пользователь должен явно использовать пространство имён для создания TDF, LSF и модели ELN. Программа ниже показывает тот же пример, что и приведённый выше, но теперь с использованием файла заголовка

```
    <SystemC-ams.h>.
    #include <systemc-ams.h>
    #include "my_source.h"
    int sc_main(int argc, char* argv[])
    {
    sc_set_time_resolution(1.0, sc_core::SC_FS);
    sca_tdf::sca_signal<double> sig1;
    // instantiate user-defined module from user-defined
'my_tdf' namespace
    my_tdf::my_source i_my_source("i_my_source");
    i_my_source.out(sig1);
    // instantiate other modules
    ...
    // tracing AMS signals
    sca_trace_file* tf =
sca_create_tabular_trace_file("trace.dat");
    sca_trace(tf, sig1 ,"sig1");
    sc_start(10.0, SC_MS);
    tf->reopen("ac_trace.dat");
    tf->set_mode( sca_ac_format(SCA_AC_MAG_RAD) );
    sca_ac_start(1.0e3, 1.0e6, 4, SCA_LOG);
    sca_close_tabular_trace_file(tf);
    return 0;
    }
```

Рекомендуется использовать заголовочный файл <systemc-ams>, в результате чего соглашение об именах отражает полные имена классов и функций.

Использование директивы

Директива using в C ++ позволяет использовать элементы в пространстве имён без явного добавления идентификатора пространства имён для каждого

элемента. Он должен использоваться только в реализации модуля, а не в объявлении модуля (например, определение в заголовочном файле). Рекомендуется применять директиву using только в пределах локальная область действия, например, как часть реализации функции-члена класса. Пример ниже показывает, как эта концепция может применяться для описания частотной области, как описано в разделе 5.3.3.

```
void ac_processing()
{
using namespace sca_util;
using namespace sca_ac_analysis;
sca_complex s = SCA_COMPLEX_J * sca_ac_w();
sca_complex h = 1.0 / ( s * s + s + 1.0 );
sca_ac(out) = h * sca_ac(in);
}
```

7.4.2. Динамическое распределение памяти

В большинстве примеров, показанных в этом руководстве пользователя, используются объекты (например, примитивные модули), которые непосредственно создаются в теле функции и, таким образом, автоматически выделяется в стеке. В случае больших конструкций при использовании многих модулей в сложной иерархии этот подход не является наиболее эффективным, поскольку он может привести к переполнение стека для автоматических переменных. Динамическое распределение памяти имеет то преимущество, что даёт пользователю более прямой контроль, в каком порядке построены модули. На экземпляры объектов ссылаются указателями, чтобы им больше не приходилось постоянно находиться в области памяти, что может привести к проблеме распределения ресурсов. Кроме того, это позволяет создавать экземпляры произвольного числа модулей, которые определяются во время выполнения, на которые ссылаются из динамически создаваемого массива указателей модулей, и конструкторы которых могут вызываться индивидуально для изменения параметризации каждого объекта.

Оператор C ++ *new* используется для динамического выделения памяти в массиве для хранения объектов. Так как распределение возвращает адрес во вновь выделенную память, доступ к функциям-членам объекта осуществляется с помощью указателя. Любая память, динамически выделяемая оператором new, должна быть освобождена (освобождена) с помощью оператора удаления. Этот оператор обычно вызывается для каждого динамически создаваемого объекта-члена в деструкторе класса.

В приведённом ниже примере показано использование динамического выделения и освобождения памяти для BASK демодулятор, аналогичного описанному в разделе 2.6.2.

```
SC_MODULE(bask_demod)
{
sca_tdf::sca_in<double> in;
```

```
    sca_tdf::sca_out<bool> out;
    rectifier* rc;
    ltf_nd_filter* lp;
    sampler* sp;
    SC_CTOR(bask_demod) : in("in"), out("out"),
rc_out("rc_out"), lp_out("lp_out")
    {
    rc = new rectifier("rc");
    rc->in(in);
    rc->out(rc_out);
    lp = new ltf_nd_filter("lp", 3.3e6);
    lp->in(rc_out);
    lp->out(lp_out);
    sp = new sampler("sp");
    sp->in(lp_out);
    sp->out(out);
    }
    ~bask_demod()
    {
    delete(rc);
    delete(lp);
    delete(sp);
    }
    private:
    sca_tdf::sca_signal<double> rc_out, lp_out;
    };
```

7.4.3. Параметры модуля

Модули должны быть гибкими, чтобы их можно было повторно использовать, то есть их поведение и внутренняя структура должны быть параметризованы в разумной степени, чтобы позволить им принятие различных спецификаций. Это особенно интересно для ранних этапов проектирования архитектуры и последующего уточнение структуры системы.

В разделе 2.6.1 была представлена модель BASK модулятора с жестко закодированными проектными параметрами, такими как несущая частота 70 МГц. Что касается этой несущей частоты, значения временного шага и скорости передачи данных, которые жёстко запрограммированы, так что результирующий сигнал был достаточно дискретизирован. Такие «магические числа», порт с жёстким кодом, скорости, задержки и временные шаги являются типичными признаками негибкой реализации. Если, например, несущая частота будет увеличена без изменения временного шага, модель может работать некорректно, в связи с недостаточной выборкой.

Более гибкий подход заключается в получении значений шага по времени и скорости передачи данных из параметров функционального модуля.

В этом разделе показано, как сделать параметризованную версию BASK-модулятора из Раздела 2.6.1, с регулируемой несущей частотой и модулирующей полосы частот, а также способ автоматического определения из этого скорости передачи данных и временных интервалов. Во-первых, необходим микшер с параметризованной скоростью передачи данных:

```
SCA_TDF_MODULE(mixer)
{
sca_tdf::sca_in<bool> in_bin; // input port baseband signal
sca_tdf::sca_in<double> in_wav; // input port carrier signal
sca_tdf::sca_out<double> out; // output port modulated signal
mixer( sc_core::sc_module_name nm, unsigned long rate_ )
: in_bin("in_bin"), in_wav("in_wav"), out("out"), rate(rate_)
{
using namespace sc_core; // essential for sc_assert to work, when using OSCI systemc-2.2.0
sc_assert(rate_ > 0);
}
void set_attributes()
{
in_wav.set_rate(rate);
out.set_rate(rate);
}
void processing()
{
for(unsigned long i = 0; i < rate; i++)
{
if( in_bin.read() )
out.write( in_wav.read(i), i );
else out.write( 0.0 , i );
}
}
private:
unsigned long rate;
};
```

Если используются параметры, которые вычисляются в другом месте, всегда полезно проверять достоверность. Следовательно, конструктор микшеров содержит строку sc_assert (rate_> 0), чтобы проверить, равен ли параметр rate минимум 1. Обратите внимание, что реализация sc_assert в ссылочной версии реализации OSCI SystemC 2.2 (systemc-2.2.0) не

соответствует стандарту IEEE 1666-2005 и, следовательно, *using namespace sc_core* должен быть добавлен перед вызовом sc_assert.

Используя этот микшер и параметризованный синусоидальный источник, уже использованный в Разделе 2.6.1, параметризованный модулятор BASK может быть реализован следующим образом:

```
SC_MODULE(bask_mod)
{
sca_tdf::sca_in<bool> in;
sca_tdf::sca_out<double> out;
sin_src sine;
mixer mix;
bask_mod( sc_core::sc_module_name nm,
double baseband_freq,
double carrier_freq,
double carrier_ampl = 1.0,
unsigned long samples_per_period = 20 )
: in("in"), out("out"),
sine("sine",
carrier_ampl,
carrier_freq,
sca_core::sca_time( (1.0 / (samples_per_period *
carrier_freq) ), sc_core::SC_SEC) ),
mix("mix", (int)ceil(
static_cast<double>(samples_per_period) * carrier_freq /
baseband_freq ) ),
carrier("carrier")
{
using namespace sc_core; // essential for sc_assert
to work, when using OSCI systemc-2.2.0
// Plausibility checks
sc_assert(carrier_freq > baseband_freq); // wouldn't
make sense otherwise!
sc_assert(samples_per_period > 2); // Nyquist
criterion satisfied?
sc_assert(carrier_ampl > 0.0); // Otherwise the
output is 0 all the way!
sine.out(carrier);
mix.in_wav(carrier);
mix.in_bin(in);
mix.out(out);
}
private:
sca_tdf::sca_signal<double> carrier;
};
```

Приведенный выше модулятор BASK можно настроить со следующими параметрами:

• baseband_freq - частота двоичного сигнала.

• carrier_freq - частота несущего сигнала.

• carrier_ampl - это амплитуда несущего сигнала, которая по умолчанию равна 1.

• samples_per_period - количество выборок, используемых для одного периода несущей синусоидального сигнала. По умолчанию 20 обеспечивает достаточную выборку.

Из этих параметров вычисляются соответствующие параметры для конструкторов sin_src и mixer.

Опять же, конструктор содержит некоторые проверки правдоподобия с использованием sc_assert. Временной шаг sin_src является обратном произведению несущей частоты и выборок на использованный синусовый период. Например, если несущая частота составляет 10 МГц и используется 20 выборок за период, общая частота дискретизации становится 200 МГц, что даёт шаг по времени 5 нс. Скорость порта in_wav микшера должна быть соотношением произведения выборок за период и несущей частоты на частоту основной полосы частот. Предполагая последнее равным 2 МГц и снова для несущей частоты 10 МГц с 20 выборками за период, это приведёт к получению скорость данных 100. Обратите внимание, что операция максимума в коде модулятора может привести к несколько более высоким выборкам за скоростной период, чем предполагалось.

7.4.4. Разделение определения модуля и реализации

Приведенные выше сжатые примеры реализовали поведение или структурную композицию непосредственно внутри определения класса. Рекомендуется отделить определение модуля от фактической реализации, в файл заголовка (с расширением .h или .hpp) и файл реализации (с расширением .cpp), так как это обычная практика программирования на C ++. Таким образом, только информация, необходимая для использования модуля выставляется другим файлам, включая заголовок, а не детали его реализации. Генерацию дублированного кода следует избегать для сокращения общего времени компиляции. Только для объявления шаблонных классов и реализации необходимо, чтобы оба хранились в заголовочных файлах, так как компилятор C ++ должен уметь специализировать реализацию переданным параметрам шаблона.

В приведённом ниже примере показан пример демодулятора BASK из раздела 2.6.2, где композиция реализована в отдельном файле реализации, как часть конструктора модуля. Определение класса помещается в заголовочный файл, что позволяет включать его в другие файлы. Обратите внимание, что это разделение не может применяться в случае, если модуль создан с использованием шаблона класса.

// bask_demod.h

```
#ifndef BASK_DEMOD_H_
#define BASK_DEMOD_H_
#include <systemc-ams>
#include "rectifier.h"
#include "ltf_nd_filter.h"
#include "sampler.h"
SC_MODULE(bask_demod)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<bool> out;
rectifier* rc;
ltf_nd_filter* lp;
sampler* sp;
bask_demod( sc_core::sc_module_name nm );
private:
sca_tdf::sca_signal<double> rc_out, lp_out;
};
#endif // BASK_DEMOD_H_
```

Реализация класса, содержащая фактическую структурную композицию, хранится в отдельном файле:

```
#include "bask_demod.h"
bask_demod::bask_demod(sc_core::sc_module_name nm)
: in("in"), out("out"), rc_out("rc_out"),
lp_out("lp_out")
{
rc = new rectifier("rc");
rc->in(in);
rc->out(rc_out);
lp = new ltf_nd_filter("lp", 3.3e6);
lp->in(rc_out);
lp->out(lp_out);
sp = new sampler("sp");
sp->in(lp_out);
sp->out(out);
}
```

7.4.5. Шаблоны классов

Шаблоны классов C ++ могут использоваться в случае, если требуется несколько экземпляров, использующих разные типы данных или размеров в проекте. Например, если параллельный поток данных шириной N должен быть сериализован, это может быть смоделировано очень естественно, с модулем TDF, имеющим входную скорость передачи данных 1 и выходную скорость передачи данных N. Рисунок 7.11 показывает определение сериализатора, реализованного в виде шаблона класса с параметром N. Для сериализации 3

битного вектора, параметр шаблона N установлен в 3.

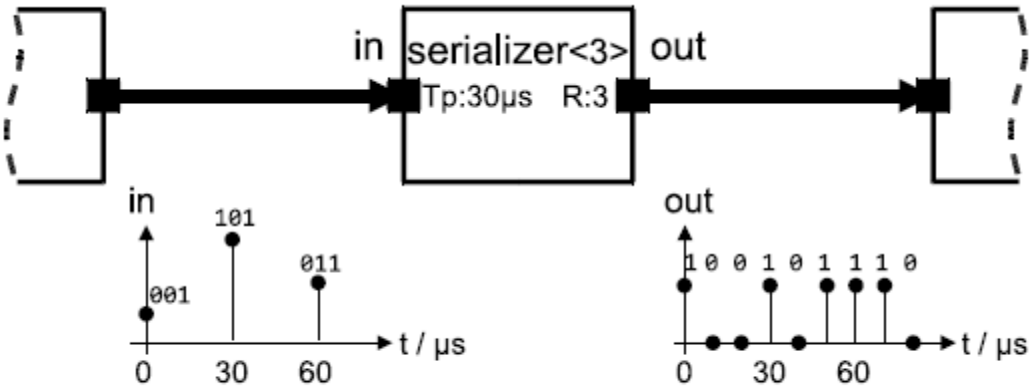


Figure 7.11. Serialization of a 3-bit vector

Рисунок 7.11. Сериализация 3-битного вектора

```
template <int N>
SCA_TDF_MODULE(serializer)
{
sca_tdf::sca_in<sc_dt::sc_bv<N> > in; // input port
sca_tdf::sca_out<bool> out; // output port
SCA_CTOR(serializer) : in("in"), out("out") {}
void set_attributes()
{
out.set_rate(N);
}
void processing()
{
for(int i = 0; i < N; i++)
{
out.write( in.read().get_bit(i), i );
}
}
};
```

В приведенном ниже примере показано, как такой класс шаблона можно использовать в структурном модуле.

```
SC_MODULE(modulator)
{
sca_tdf::sca_in<sc_dt::sc_bv<3> > in;
sca_tdf::sca_out<double> out;
serializer<3> ser;
bask_mod mod;
SC_CTOR(modulator) : in("in"), out("out"),
ser("ser"), mod("mod"), bits("bits")
```

```
    {
    ser.in(in);
    ser.out(bits);
    mod.in(bits);
    mod.out(out);
    }
    private:
    sca_tdf::sca_signal<bool> bits;
    };
```

Шаблоны классов также облегчают уточнение коммуникаций, как описано в разделе 7.3. В примере ниже показан модуль усилителя раздела 7.2, реализованный в виде шаблона класса. В зависимости от шаблона параметра type, модуль может использоваться либо в качестве модели полосы пропускания, при использовании типа double, либо в качестве модели модулирующей полосы с использованием типа данных sca_util :: sca_complex.

```
    #include <cmath>
    #include <complex>
    template <class T>
    SCA_TDF_MODULE(amplifier)
    {
    sca_tdf::sca_in<T> in;
    sca_tdf::sca_out<T> out;
    amplifier( sc_core::sc_module_name, double gain =
1.0, double iip3 = 1e-3 )
    : in("in"), out("out"), a1( gain ), a3( -4/3 * (gain
/ std::pow(iip3,2)) ) {}
    void processing()
    {
    out.write( a1 * in.read() + a3 *
std::pow(in.read(),3) );
    }
    private:
    double a1, a3;
    };
```

7.4.6. Публичные и частные члены класса

При создании модуля с использованием макроса SC_MODULE или SCA_TDF_MODULE класс определяется, используя структуру ключевых слов C ++. В этом случае все члены класса, такие как функции и переменные данных, общедоступный по умолчанию. Эти члены могут быть доступны извне класса, например, из функции и т.п., основная программа sc_main или другого класса, например, родительского модуля. Модули, которые определены с ключевого слова class, есть закрытые члены по умолчанию.

Чтобы иметь возможность создать экземпляр модуля и связать его с другими модулями, конструктором и портами они должны быть объявлены как публичные (public). Рекомендуется объявлять внутренние сигналы, узлы, переменные, функции и примитивные модули, как частные (private), если только нет веских причин для доступа к ним вне области действия *класса*. Например, сигналы и узлы могут быть объявлены public для облегчения отладки.

Чтобы облегчить отслеживание сигналов или узлов, которые объявлены private, вспомогательная функция trace_internals может быть определяется как public member, открытый член (public member), который будет записывать сигналы в файл трассировки, определённый аргументом. Пример ниже расширяет демодулятор BASK из раздела 2.6.2 с отслеживанием частных (private) членов. В таком случае, нет необходимости объявлять сами сигналы как общедоступные (public).

```
SC_MODULE(bask_demod)
{
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<bool> out;
rectifier rc;
ltf_nd_filter lp;
sampler sp;
SC_CTOR(bask_demod)
: in("in"), out("out"), rc("rc"), lp("lp", 3.3e6),
sp("sp"), rc_out("rc_out"), lp_out("lp_out")
{
rc.in(in);
rc.out(rc_out);
lp.in(rc_out);
lp.out(lp_out);
sp.in(lp_out);
sp.out(out);
}
void trace_internals( sca_util::sca_trace_file* tf )
{
sca_util::sca_trace(tf, rc_out, rc_out.name() );
sca_util::sca_trace(tf, lp_out, lp_out.name() );
}
private:
sca_tdf::sca_signal<double> rc_out, lp_out;
};
```

Приложение А. Справочник по языку

Примечание. В этом приложении приведен только список основных определений языка для примитивов TDF, LSF или ELN модули. Полный список определений можно найти в Справочном руководстве по языкам SystemC Расширения AMS.

Если значение по умолчанию для параметра не указано в таблицах ниже, это значение должно быть предоставлено пользователем и не может быть опущено во время разработки.

A.1. TDF modules

Имя	Тип	Описание
	T	Произвольный тип данных (e.g double, sca_util::sca_vector, ...
tstep	sca_core::sca_time	Шаг по времени как объект
abstime	sca_core::sca_time	Шаг по времени как объект
tstepd	double	Шаг по времени в секундах
tunit	sc_core::sc_time_unit	Единица времени (e.g., sc_core::SC_US, sc_core::SC_MS, ...)
name	const char*	Имя модуля в виде строки
modname	sc_core::sc_module_name	Имя модуля как объекта

```
     SCA_TDF_MODULE( name )
     {
     // port declarations
     sca_tdf::sca_in<T> in; // input port
     sca_tdf::sca_out<T> out; // output port
     // Converter ports
     sca_tdf::sca_de::sca_in<T> inp; // converter port from
discrete-event domain
     sca_tdf::sca_de::sca_out<T> outp; // converter port to
discrete-event domain
     // TDF methods, called automatically by the scheduler
     void set_attributes()
     {
     // module and port attributes (optional)
     }
     void initialize()
     {
     // initial values of ports with a delay (optional)
     }
     void processing()
     {
     // time-domain signal processing behavior or algorithm
(mandatory)
     }
     void ac_processing()
     {
     // small-signal frequency-domain behavior (optional)
     }
     // module constructor
```

```
    SCA_CTOR( name ) {} // macro, or
    name( modname ) {} // full constructor, can also be used to
pass parameters
    };
```

A.2. TDF ports

Имя	Тип	Описание
value	T	Значение с произвольным типом (double, sca_util::sca_vector, ...)
sample_id	unsigned long	Образец ID: 0 for single-rate, 0...(rate-1) for multirate
nsamples	unsigned long	Количество образцов
rate	unsigned long	Скорость порта
tstep	sca_core::sca_time	Шаг по времени как объект
tstepd	double	Шаг по времени в секундах
tunit	sc_core::sc_time_unit	Единица времени (e.g., sc_core::SC_US, sc_core::SC_MS, ...)
toffset	sca_core::sca_time	Смещение времени как объекта
toffsetd	double	Смещение времени в секундах

```
sca_tdf::sca_in<T> in;
sca_tdf::sca_out<T> out;
sca_tdf::sca_de::sca_in<T> inp;
sca_tdf::sca_de::sca_out<T> outp;
out.set_delay( nsamples );
out.set_rate( rate );
out.set_timestep( tstep );
out.set_timestep( tstepd, tunit );
outp.set_timeoffset( toffset );
outp.set_timeoffset( toffsetd, tunit );
nsamples = out.get_delay();
rate = out.get_rate();
abstime = out.get_time();
abstime = out.get_time( sample_id );
tstep = out.get_timestep();
tstepd = out.get_timestep().to_seconds();
toffset = outp.get_timeoffset();
out.initialize( value, sample_id );
value = in.read();
value = in.read( sample_id );
out.write( value );
out.write( value, sample_id );
```

A.3. TDF сигналы

```
// type T
sca_tdf::sca_signal<T> // TDF сигнал
```

A.4. Встроенные функции передачи Лапласа

A.4.1. sca_tdf::sca_ltf_nd

Описание

Масштабированная передаточная функция Лапласа во временной области в форме числитель-знаменатель.

Определение

```
sca_tdf::sca_ltf_nd( num, den, delay, state, input,
k, tstep );
```

Уравнение

$$H(s) = k \cdot \frac{\sum_{i=0}^{M-1} num_i \cdot s^i}{\sum_{i=0}^{N-1} den_i \cdot s^i} \cdot e^{(-s \cdot delay)}$$

Параметры

Имя	Тип	По умолчанию	Описание
num	**sca_util::sca_vector**<double>		Числовые коэффициенты
den	den **sca_util::sca_vector**<double>		Коэффициенты денумератора
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки (опционально)
state	**sca_util::sca_vector**<double>		Вектор состояния (необязательно)
input	double, **sca_tdf::sca_in**<double>, **sca_tdf::sca_de::sca_in**<double>, **sca_util::sca_vector**<double>		Входное значение или сигнал от порта
k	double	1.0	Коэффициент усиления (необязательно)
tstep	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Шаг времени

Ограничение использования:

Задержка должна быть больше или равна нулю.

A.4.2. sca_tdf::sca_ltf_zp

Описание

Масштабная передаточная функция Лапласа во временной области в форме нуль - полюс.

Определение

```
sca_tdf::sca_ltf_zp( zeros, poles, delay, state,
input, k, tstep );
```

Уравнение

$$H(s) = k \cdot \frac{\prod_{i=0}^{M-1}(s - zeros_i)}{\prod_{i=0}^{N-1}(s - poles_i)} \cdot e^{(-s \cdot delay)}$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
zeros	**sca_util::sca_vector**< **sca_util::sca_complex** >		Числовые коэффициенты
poles	**sca_util::sca_vector**< **sca_util::sca_complex** >		Коэффициенты денумератора
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки (опционально)
state	**sca_util::sca_vector**<double>		Вектор состояния (необязательно)
input	double, **sca_tdf::sca_in**<double>, **sca_tdf::sca_de::sca_in**<double>, **sca_util::sca_vector**<double>		Входное значение или сигнал от порт
k	double	1.0	Коэффициент усиления (необязательно)
tstep	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Шаг времени

Ограничение использования

Задержка должна быть больше или равна нулю.

A.4.3. sca_tdf::sca_ss

Описание

Уравнение пространства состояний с одним входом и одним выходом.

Определение

```
sca_tdf::ss_eqn( a, b, c, d, delay, s, x, tstep );
```

Уравнение

$$\frac{ds(t)}{dt} = \mathbf{A} \cdot s(t) + \mathbf{B} \cdot x(t - delay)$$
$$y(t) = \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay)$$

Параметры

Имя	Тип	По умолчанию	Описание
a	**sca_util::sca_matrix**<double>		Матрица A размером n-на-n (n= количество состояний)
b	**sca_util::sca_matrix**<double>		Матрица B размером n-на-m (м = количество входов)
c	**sca_util::sca_matrix**<double>		Матрица C размером r-на-n (r = количество выходов)
d	**sca_util::sca_matrix**<double>		Матрица D размером r на m
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки (необязательный)
state	double	1.0	Вектор состояния (необязательно)
x	**sca_util::sca_vector**<double>, **sca_util::sca_matrix**<double>, **sca_tdf::sca_in**<double>, **sca_tdf::sca_in**< **sca_util::sca_vector**<double>, **sca_tdf::sca_de::sca_in**< **sca_util::sca_vector**<double>	**sc_core::SC_ZERO_TIME**	Входной вектор, матрица или сигнал из порта
tstep	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Шаг времени

Ограничение использования

Задержка должна быть больше или равна нулю.

A.5. LSF примитивные модули

A.5.1. sca_lsf::sca_add

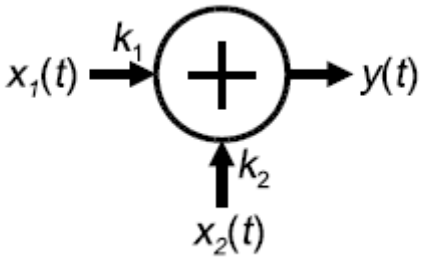
Описание

Взвешенное сложение двух сигналов LSF.

Определение

```
sca_lsf::sca_add( nm, k1, k2 );
```

Символ



Уравнение

$$y(t) = k_1 \cdot x_1(t) + k_2 \cdot x_2(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
k1	double	1,0	Весовой коэффициент для сигнала LSF в порту x1
k2	double	1,0	Весовой коэффициент для сигнала LSF в порту x2

Порты

Имя	Интерфейс	Тип/Характер	Описание
x1	**sca_lsf::sca_in**	Поток сигналов	LSF вход 1
x2	**sca_lsf::sca_in**	Поток сигналов	LSF вход 2
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.2. sca_lsf::sca_sub

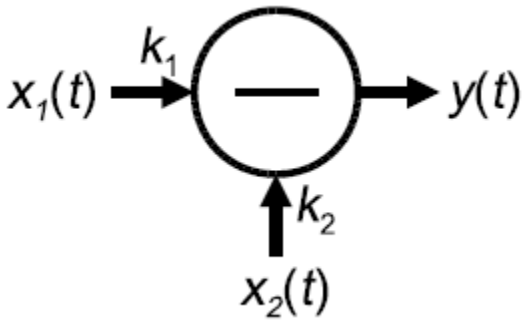
Описание

Взвешенное вычитание двух сигналов LSF.

Определение

```
sca_lsf::sca_sub( nm, k1, k2 );
```

Символ



Уравнение

$$y(t) = k_1 \cdot x_1(t) - k_2 \cdot x_2(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
k1	double	1,0	Весовой коэффициент для сигнала LSF в порту x1
k2	double	1,0	Весовой коэффициент для сигнала LSF в порту x2

Порты

Имя	Интерфейс	Тип/Характер	Описание
x1	**sca_lsf::sca_in**	Поток сигналов	LSF вход 1
x2	**sca_lsf::sca_in**	Поток сигналов	LSF вход 2
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.3. sca_lsf::sca_gain

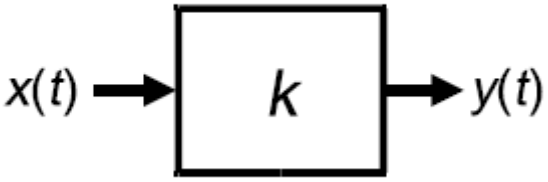
Описание

Умножение LSF-сигнала на постоянное усиление.

Определение

```
sca_lsf::sca_gain( nm, k );
```

Символ



Уравнение

$$y(t) = k \cdot x(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
k	double	1,0	Коэффициент усиления

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.4. sca_lsf::sca_dot

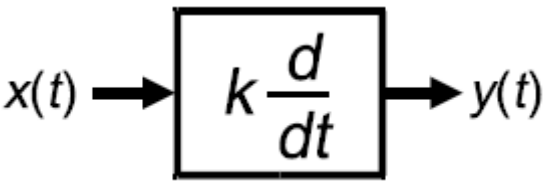
Описание

Масштабированная производная по времени первого порядка сигнала LSF.

Определение

```
sca_lsf::sca_dot( nm, k );
```

Символ



Уравнение

$$y(t) = k \cdot \frac{dx(t)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
k	double	1,0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.5. sca_lsf::sca_integ

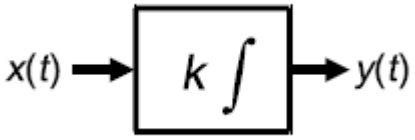
Описание

Масштабированное интегрирование во временной области сигнала LSF.

Определение

```
sca_lsf::sca_integ( nm, k, y0 );
```

Символ



Уравнение

$$y(t) = k \cdot \int_o^t x(t)dt + y_0$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
k	double	1.0	Коэффициент шкалы
y0	double	0.0	Начальное состояние при t=0

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.6. sca_lsf::sca_delay

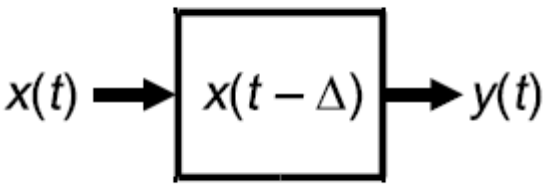
Описание

Масштабированная версия LSF с задержкой по времени.

Определение

```
sca_lsf::sca_delay( nm, delay, k, y0 );
```

Символ



Уравнение

$$y(t) = \begin{cases} y_0 & t \le delay \\ k \cdot x(t - delay) & t > delay \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки
k	double	1.0	Коэффициент шкалы
y0	double	0.0	Выходное значение до задержки как результат

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

Ограничение использования

Задержка должна быть больше или равна нулю.

A.5.7. sca_lsf::sca_source

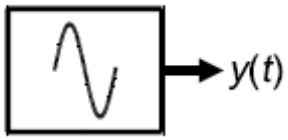
Описание

Источник LSF.

Определение

```
    sca_lsf::sca_source( nm, init_value, offset,
amplitude, frequency, phase, delay,
    ac_amplitude, ac_phase, ac_noise_amplitude );
```

Символ



Уравнения

For time-domain simulation:

$$y(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

For small-signal frequency-domain simulation:

$$y(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

For small-signal frequency-domain noise simulation:

$$y(f) = ac\_noise\_amplitude$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
init_value	double	0.0	Начальное значение
offset	double	0.0	Значение смещения
amplitude	double	0.0	Амплитуда источника
frequency	double	0.0	Частота источника в герцах

phase	double	0.0	Фаза источника в радианах
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки
ac_amplitude	double	0.0	Амплитуда слабого сигнала *)
ac_phase	double	0.0	Фаза слабого сигнала в радианах*)
ac_noise_ amplitude	double	0.0	Амплитуда шума слабого сигнала **)

*) только для моделирования слабых сигналов в частотной области.
**) только для моделирования слабых сигналов и шума в частотной области.

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

Ограничение использования

Задержка должна быть больше или равна нулю.

A.5.8. sca_lsf::sca_ltf_nd

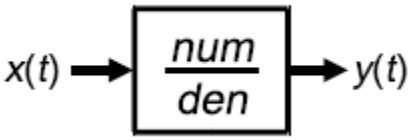
Описание

Масштабированная передаточная функция Лапласа во временной области в форме числитель-знаменатель.

Определение

```
sca_lsf::sca_ltf_nd( nm, num, den, delay, k );
```

Символ



Уравнения

$$den_{N-1}\frac{d^{N-1}y(t)}{dt}+den_{N-2}\frac{d^{N-2}y(t)}{dt}+\cdots+den_1\frac{dy(t)}{dt}+den_0\cdot y(t)$$

$$=k\cdot\left(num_{M-1}\frac{d^{M-1}x(t-delay)}{dt}+num_{M-2}\frac{d^{M-2}x(t-delay)}{dt}+\cdots+num_1\frac{dx(t-delay)}{dt}+num_0\cdot x(t-delay)\right)$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
num	**sca_util::sca_vector**<double>		Numerator coefficients
den	**sca_util::sca_vector**<double>		Denumerator coefficients
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay
k	double	1.0	Gain coefficient

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

Ограничение использования

Задержка должна быть больше или равна нулю.

A.5.9. sca_lsf::sca_ltf_zp

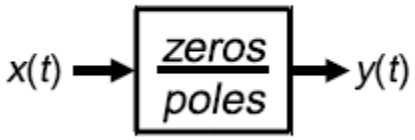
Описание

Масштабная передаточная функция Лапласа во временной области в форме нуль - полюс.

Определение

```
sca_lsf::sca_ltf_zp( nm, zeros, poles, delay, k );
```

Символ



Уравнения

$$\left(\frac{d}{dt} - poles_{N-1}\right)\left(\frac{d}{dt} - poles_{N-2}\right)\cdots\left(\frac{d}{dt} - poles_{1}\right)\left(\frac{d}{dt} - poles_{0}\right)y(t)$$

$$= k \cdot \left\{\left(\frac{d}{dt} - zeros_{M-1}\right)\left(\frac{d}{dt} - zeros_{M-2}\right)\cdots\left(\frac{d}{dt} - zeros_{1}\right)\left(\frac{d}{dt} - zeros_{0}\right)x(t - delay)\right\}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
zeros	**sca_util::sca_vector**< **sca_util::sca_complex**>		Нули
poles	**sca_util::sca_vector**< **sca_util::sca_complex**>		Полюсы
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки

k	double	1.0	Коэффициент усиления

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

Ограничения на использование

Расширение числителя и знаменателя должно привести к реальной стоимости, соответственно. Задержка должна быть больше или равно нулю.

A.5.10. sca_lsf::sca_ss

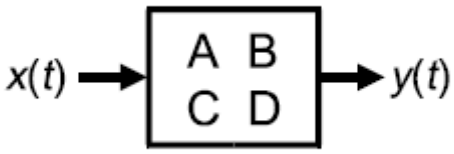
Описание

Уравнение пространства состояний с одним входом и одним выходом.

Определение

```
sca_lsf::sca_ss( nm, a, b, c, d, delay );
```

Символ



Уравнения

$$\frac{ds(t)}{dt} = \mathbf{A}\cdot s(t) + \mathbf{B}\cdot x(t-delay)$$
$$y(t) = \mathbf{C}\cdot s(t) + \mathbf{D}\cdot x(t-delay)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля

a	**sca_util::sca_matrix<double>**		Матрица A размером n-на-n
b	**sca_util::sca_matrix<double>**		Матрица B с одним столбцом размером n
c	**sca_util::sca_matrix<double>**		Матрица C с одним рядом размер n
d	**sca_util::sca_matrix<double>**		Матрица D размера 1
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

Ограничение использования

Задержка должна быть больше или равна нулю.

A.5.11. sca_lsf::sca_tdf::sca_gain, sca_lsf::sca_tdf_gain

Описание

Масштабное умножение входного сигнала TDF на входной сигнал LSF.

Определение

Класс sca_lsf :: sca_tdf :: sca_gain должен реализовывать примитивный модуль для LSF MoC, который реализует масштабированное умножение входного сигнала TDF на входной сигнал LSF. Примитив должен внести следующее уравнение к системе уравнений:

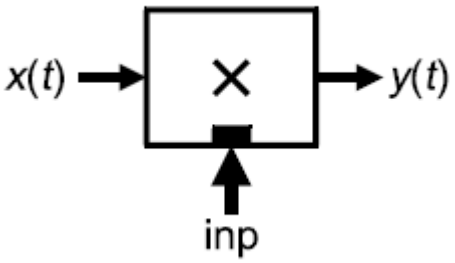
y (т) = масштаб · дюйм · x (т) (4,27),

где масштаб - постоянный масштабный коэффициент, inp - входной

сигнал TDF, который должен интерпретироваться как сигнал непрерывного времени, x (t) - входной сигнал LSF, а y (t) - выходной сигнал LSF.

```
sca_lsf::sca_tdf::sca_gain( nm, scale );
sca_lsf::sca_tdf_gain( nm, scale );
```

Символ



Уравнения

$$y(t) = scale \cdot inp \cdot x(t)$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы
b	**sca_util::sca_matrix<double>**		Матрица B с одним столбцом размером n

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
inp	**sca_tdf::sca_in<T>**	double	TDF вход
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.12. sca_lsf::sca_tdf::sca_source, sca_lsf::sca_tdf_source

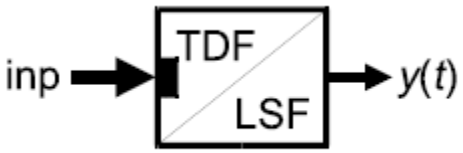
Описание

Масштабное преобразование входного сигнала TDF в выходной сигнал LSF.

Определение

```
sca_lsf::sca_tdf::sca_source( nm, scale );
sca_lsf::sca_tdf_source( nm, scale );
```

Символ



Уравнение

$$y(t) = scale \cdot inp$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
inp	**sca_tdf::sca_in<T>**	double	TDF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.13. sca_lsf::sca_tdf::sca_sink, sca_lsf::sca_tdf_sink

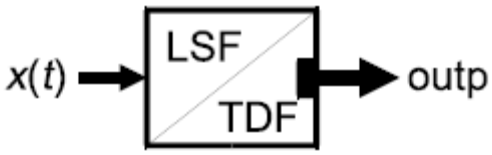
Описание

Масштабное преобразование из входного сигнала LSF в выходной сигнал TDF.

Определение

```
sca_lsf::sca_tdf::sca_sink( nm, scale );
sca_lsf::sca_tdf_sink( nm, scale );
```

Символ



Уравнение

Нет никакого уравнения, внесенного в общую систему уравнений для этого модуля.

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
outp	**sca_tdf::sca_out<T>**	double	TDF выход

A.5.14. sca_lsf::sca_tdf::sca_mux, sca_lsf::sca_tdf_mux

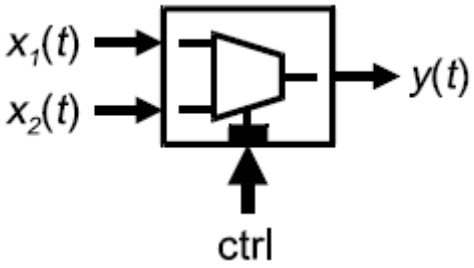
Описание

Выбор одного из двух входных сигналов LSF с помощью управляющего сигнала TDF (мультиплексор).

Определение

```
sca_lsf::sca_tdf::sca_mux( nm );
sca_lsf::sca_tdf_mux( nm );
```

Символ



Уравнение

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля

Порты

Имя	Интерфейс	Тип/Характер	Описание
x1	**sca_lsf::sca_in**	Поток сигналов	LSF вход 1
x2	**sca_lsf::sca_in**	Поток сигналов	LSF вход 2
ctrl	**sca_tdf::sca_in<T>**	bool	TDF control input
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.15. sca_lsf::sca_tdf::sca_demux, sca_lsf::sca_tdf_demux

Описание

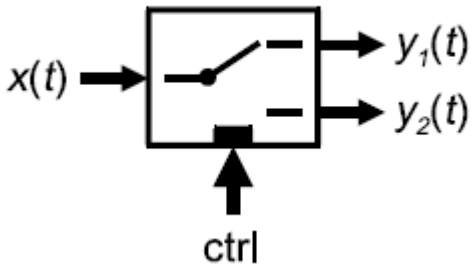
Маршрутизация входного сигнала LSF на один из двух выходных сигналов LSF, управляемых сигналом TDF

(Демультиплексор).

Определение

```
sca_lsf::sca_tdf::sca_demux( nm );
sca_lsf::sca_tdf_demux( nm );
```

Символ



Уравнения

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход 1
ctrl	**sca_tdf::sca_in<T>**	bool	Вход управления TDF
y1	**sca_lsf::sca_out**	Поток сигналов	LSF выход 1
y2	**sca_lsf::sca_out**	Поток сигналов	LSF выход 1

A.5.16. sca_lsf::sca_de::sca_gain, sca_lsf::sca_de_gain

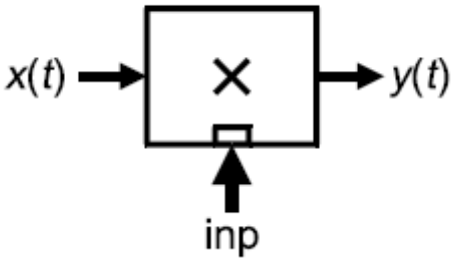
Описание

Масштабное умножение входного сигнала дискретного события на входной сигнал LSF.

Определение

```
sca_lsf::sca_de::sca_gain( nm, scale );
sca_lsf::sca_de_gain( nm, scale );
```

Символ



Уравнения

$$y(t) = scale \cdot inp \cdot x(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Scale coefficient

Порты

Имя	Интерфейс	Тип/Характер	Описание
inp	**sc_core::sc_in<T>**	double	Ввод дискретных событий
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.17. sca_lsf::sca_de::sca_source, sca_lsf::sca_de_source

Описание

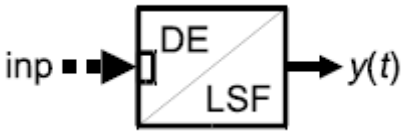
Масштабное преобразование входного сигнала дискретного события в выходной сигнал LSF.

Определение

```
sca_lsf::sca_de::sca_source( nm, scale );
```

```
sca_lsf::sca_de_source( nm, scale );
```

Символ



Уравнение

$$y(t) = scale \cdot inp$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалыt

Порты

Имя	Интерфейс	Тип/Характер	Описание
inp	**sc_core::sc_in<T>**	double	Ввод дискретных событий
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.18. sca_lsf::sca_de::sca_sink, sca_lsf::sca_de_sink

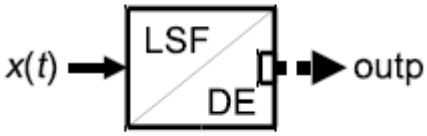
Описание

Масштабное преобразование из входного сигнала LSF в выходной сигнал дискретного события.

Определение

```
sca_lsf::sca_de::sca_sink( nm, scale );
sca_lsf::sca_de_sink( nm, scale );
```

Символ



Уравнение

Нет никакого уравнения, внесенного в общую систему уравнений для

этого модуля.

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалыt

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
outp	**sc_core::sc_out<T>**	double	Вывод дискретного события

A.5.19. sca_lsf::sca_de::sca_mux, sca_lsf::sca_de_mux

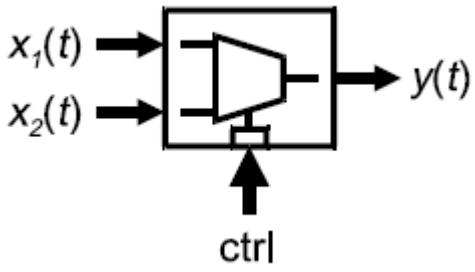
Описание

Выбор одного из двух входных сигналов LSF с помощью сигнала управления дискретным событием (мультиплексор).

Определение

```
sca_lsf::sca_de::sca_mux( nm );
sca_lsf::sca_de_mux( nm );
```

Символ



Уравнения

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание

nm	**sc_core::sc_module_name**		Имя модуля

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
x1	**sca_lsf::sca_in**	Поток сигналов	LSF вход 1
x2	**sca_lsf::sca_in**	Поток сигналов	LSF вход 2
ctrl	**sc_core::sc_in<T>**	bool	Вход управления дискретным событием
y	**sca_lsf::sca_out**	Поток сигналов	LSF выход

A.5.20. sca_lsf::sca_de::sca_demux, sca_lsf::sca_de_demux

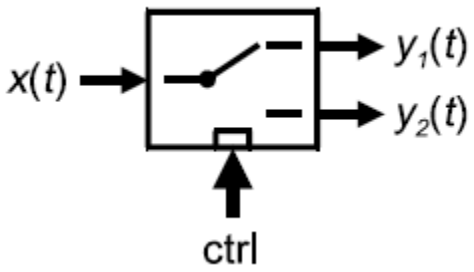
Описание

Маршрутизация входного сигнала LSF на один из двух выходных сигналов LSF, управляемых сигналом дискретного события (демультиплексор).

Определение

```
sca_lsf::sca_de::sca_demux( nm );
sca_lsf::sca_de_demux( nm );
```

Символ



Уравнения

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля

Порты

Имя	Интерфейс	Тип/Характер	Описание
x	**sca_lsf::sca_in**	Поток сигналов	LSF вход
ctrl	**sc_core::sc_in<T>**	bool	Вход управления дискретным событием
y1	**sca_lsf::sca_out**	Поток сигналов	LSF выход 1
y2	**sca_lsf::sca_out**	Поток сигналов	LSF выход 2

A.6. ELN primitive modules

A.6.1. sca_eln::sca_r

Описание

Резистор.

Определение

```
sca_eln::sca_r( nm, value );
```

Символ

Уравнение

$$v_{p,n}(t) = i_{p,n}(t) \cdot value$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Сопротивление в Омах

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал

A.6.2. sca_eln::sca_c

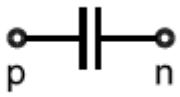
Описание

Конденсатор.

Определение

```
sca_eln::sca_c( nm, value, q0 );
```

Символ



Уравнение

$$i_{p,n}(t) = \frac{d(value \cdot v_{p,n}(t) + q_0)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Емкость в Фараде
q0	double	0.0	Первоначальный заряд в кулоне

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал

Ограничение использования

Значение параметра value не должно быть численно нулевым.

A.6.3. sca_eln::sca_l

Описание

Катушка индуктивности.

Определение

```
sca_eln::sca_l( nm, value, phi0 );
```

Символ

Уравнение

$$v_{p,n}(t) = \frac{d(value \cdot i_{p,n}(t) + phi_0)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Индуктивность в Генри
phi0	double	0.0	Начальный магнитный поток в Вебере

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал

n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал

Ограничение использования

Значение параметра *value* не должно быть численно нулевым.

A.6.4. sca_eln::sca_vcvs

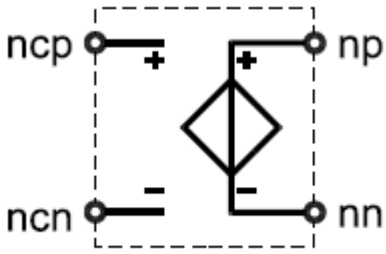
Описание

Источник напряжения, управляемый напряжением.

Определение

```
sca_eln::sca_vcvs( nm, value );
```

Символ



Уравнение

$$v_{np,nn}(t) = value \cdot v_{ncp,ncn}(t)$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Шкала коэффициента контроля напряжения

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
ncp	**sca_eln::sca_terminal**	Электрический	Положительный терминал контроля
ncn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал контроля

np	**sca_eln::sca_terminal**	Электрический	Положительный терминал источника
nn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал источника

A.6.5. sca_eln::sca_vccs

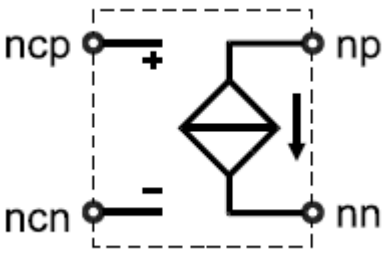
Описание

Источника тока, контролируемый напряжением

Определение

```
sca_eln::sca_vccs( nm, value );
```

Символ



Уравнение

$$i_{np,nn}(t) = value \cdot v_{ncp,ncn}(t)$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Масштабный коэффициент в сименсах контрольного напряжения

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
ncp	**sca_eln::sca_terminal**	Электрический	Положительный терминал контроля
ncn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал контроля

np	**sca_eln::sca_terminal**	Электрический	Положительный терминал источника
nn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал источника

A.6.6. sca_eln::sca_ccvs

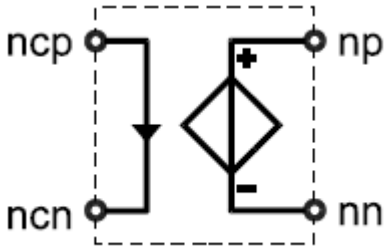
Описание

Управляемый током источник напряжения.

Определение

```
sca_eln::sca_ccvs( nm, value );
```

Символ



Уравнения

$$v_{np,nn}(t) = value \cdot i_{ncp,ncn}(t)$$

$$v_{ncp,ncn}(t) = 0$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Scale coefficient in Ohm of the control current

Порты

Имя	Интерфейс	Тип/Характер	Описание

ncp	**sca_eln::sca_terminal**	Электрический	Положительный терминал контроля
ncn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал контроля
np	**sca_eln::sca_terminal**	Электрический	Положительный терминал источника
nn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал источника

A.6.7. sca_eln::sca_cccs

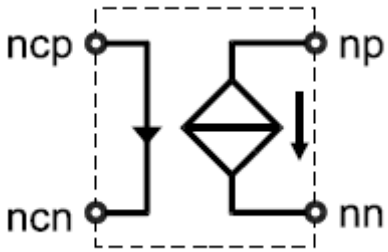
Описание

Источник тока, контролируемый током.

Определение

```
sca_eln::sca_cccs( nm, value );
```

Символ



Уравнения

$$i_{np,nn}(t) = value \cdot i_{ncp,ncn}(t)$$

$$v_{ncp,ncn}(t) = 0$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
value	double	1.0	Шкала коэффициента контроля тока

Порты

Имя	Интерфейс	Тип/Характер	Описание
ncp	**sca_eln::sca_terminal**	Электрический	Положительный терминал контроля
ncn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал контроля
np	**sca_eln::sca_terminal**	Электрический	Положительный терминал источника
nn	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал источника

A.6.8. sca_eln::sca_nullor

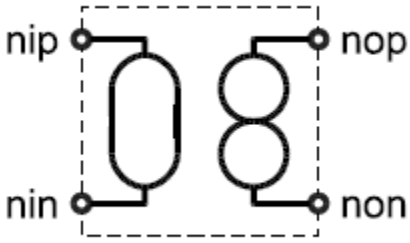
Описание

Nullor (пара нуллятор - норатор), идеальный Opamp.

Определение

```
sca_eln::sca_nullor( nm );
```

Символ



Уравнения

$$v_{nip,nin}(t) = 0$$

$$i_{nip,nin}(t) = 0$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля

Порты

Имя	Интерфейс	Тип/Характер	Описание
nip	**sca_eln::sca_terminal**	Электрический	Положительный терминал нулатора
nin	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал нулатора
nop	**sca_eln::sca_terminal**	Электрический	Положительный терминал норатора
non	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал норатора

A.6.9. sca_eln::sca_gyrator

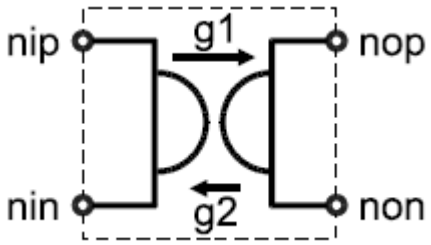
Описание

Гиратор.

Определение

```
sca_eln::sca_gyrator( nm, g1, g2 );
```

Символ



Уравнения

$$i_{p_1,n_1}(t) = g_2 \cdot v_{p_2,n_2}(t)$$

$$i_{p_2,n_2}(t) = -g_1 \cdot v_{p_1,n_1}(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
g1	double	1.0	Gyration conductance in Siemens
g2	double	1.0	Gyration conductance in Siemens

Порты

Имя	Интерфейс	Тип/Характер	Описание
p1	**sca_eln::sca_terminal**	Электрический	Положительный терминал первичного порта
n1	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал первичного порта
p2	**sca_eln::sca_terminal**	Электрический	Положительный терминал вторичного порта
n2	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал вторичного порта

A.6.10. sca_eln::sca_ideal_transformer

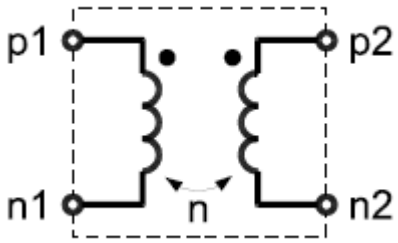
Описание

Идеальный трансформатор.

Определение

```
sca_eln::sca_ideal_transformer( nm, ratio );
```

Символ



Уравнения

$$v_{p1,n1}(t) = ratio \cdot v_{p2,n2}(t)$$

$$i_{p2,n2}(t) = ratio \cdot i_{p1,n1}(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
ratio	double	1.0	Transformation ratio

Порты

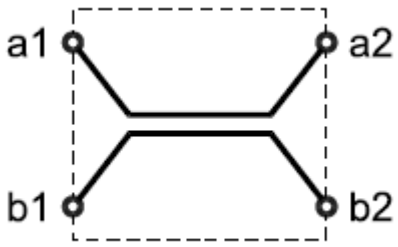
Имя	Интерфейс	Тип/Характер	Описание
p1	**sca_eln::sca_terminal**	Электрический	Положительный терминал первичного порта
n1	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал первичного порта
p2	**sca_eln::sca_terminal**	Электрический	Положительный терминал вторичного порта
n2	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал вторичного порта

A.6.11. sca_eln::sca_transmission_line

Описание

Линия передачи.

Символ



Определение

```
sca_eln::sca_transmission_line( nm, z0, delay, delta0 );
```

Уравнения

$$v_{a_1,b_1}(t) = \begin{cases} z_0 \cdot i_{a_1,b_1}(t) & t < delay \\ e^{-delta_0 \cdot delay}\left(v_{a_2,b_2}(t-delay) + z_0 \cdot i_{a_2,b_2}(t-delay)\right) + z_0 \cdot i_{a_1,b_1}(t) & t \geq delay \end{cases}$$

$$v_{a_2,b_2}(t) = \begin{cases} z_0 \cdot i_{a_2,b_2}(t) & t < delay \\ e^{-delta_0 \cdot delay}\left(v_{a_1,b_1}(t-delay) + z_0 \cdot i_{a_1,b_1}(t-delay)\right) + z_0 \cdot i_{a_2,b_2}(t) & t \geq delay \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание

nm	**sc_core::sc_module_name**		Имя модуля
z0	double	100.0	Characteristic impedance of the transmission line in Ohm
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Transmission delay
delta0	double	0.0	Dissipation factor in 1/ seconds

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
a1	**sca_eln::sca_terminal**	Электрический	Провод A на первичной стороне
b1	**sca_eln::sca_terminal**	Электрический	Провод B на первичной стороне
a2	**sca_eln::sca_terminal**	Электрический	Провод A на вторичной стороне
b2	**sca_eln::sca_terminal**	Электрический	Провод B на вторичной стороне

Ограничение использования

Задержка должна быть больше или равна нулю.

A.6.12. sca_eln::sca_vsource

Описание

Независимый источник напряжения.

Определение

```
    sca_eln::sca_vsource( nm, init_value, offset,
amplitude, frequency, phase, delay,
    ac_amplitude, ac_phase, ac_noise_amplitude );
```

Символ

Уравнения

Для моделирования во временной области:

$$v_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

Для моделирования слабого сигнала в частотной области:

$$v_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

Для моделирования слабого шума в области частот:

$$v_{p,n}(f) = ac\_noise\_amplitude$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
init_value	double	0.0	Начальное значение
offset	double	**0.0**	Значение смещения
amplitude	double	0.0	Амплитуда источника
frequency	double	0.0	Частота источника в Герцах
phase	double	0.0	Фаза источника в радианах
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки
ac_amplitude	double	**0.0**	Амплитуда слабого сигнала *)
ac_phase	double	0.0	Фаза слабого сигнала в радианах*)
ac_noise_ amplitude	double	**0.0**	Амплитуда слабого шума**)

*) только для моделирования слабых сигналов в частотной области.

**) только для моделирования слабого шума в области частот.

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	Электрический	Положительный терминал
n	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал

Ограничение использования

Задержка должна быть больше или равна нулю.

A.6.13. sca_eln::sca_isource

Описание

Независимый источник тока.

Определение

```
    sca_eln::sca_isource( nm, init_value, offset,
amplitude, frequency, phase, delay,
    ac_amplitude, ac_phase, ac_noise_amplitude );
```

Символ

Для моделирования во временной области:

$$i_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

Для моделирования слабого сигнала в частотной области:

$$i_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

Для моделирования слабого шума в области частот:

$$i_{p,n}(f) = ac\_noise\_amplitude$$

Параметры

Имя	Тип	По умолчанию	Описание

nm	**sc_core::sc_module_name**		Имя модуля
init_value	double	0.0	Начальное значение
offset	double	**0.0**	Значение смещения
amplitude	double	0.0	Амплитуда источника
frequency	double	0.0	Частота источника в Герцах
phase	double	0.0	Фаза источника в радианах
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Время непрерывной задержки
ac_amplitude	double	**0.0**	Амплитуда слабого сигнала *)
ac_phase	double	0.0	Фаза слабого сигнала в радианах*)
ac_noise_ amplitude	double	**0.0**	Амплитуда слабого шума**)

*) только для моделирования слабых сигналов в частотной области.

**) только для моделирования слабого шума в области частот.

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
p	**sca_eln::sca_terminal**	Электрический	Положительный терминал
n	**sca_eln::sca_terminal**	Электрический	Отрицательный терминал

Ограничение использования

Задержка должна быть больше или равна нулю.

A.6.14. sca_eln::sca_tdf::sca_r, sca_eln::sca_tdf_r

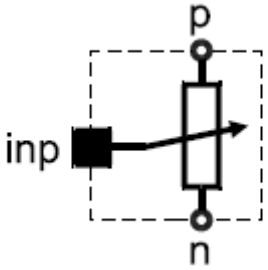
Описание

Переменный резистор управляется входным сигналом TDF.

Определение

```
sca_eln::sca_tdf::sca_r( nm, scale );
sca_eln::sca_tdf_r( nm, scale );
```

Символ



Уравнение

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления TDF

A.6.15. sca_eln::sca_tdf::sca_c, sca_eln::sca_tdf_c

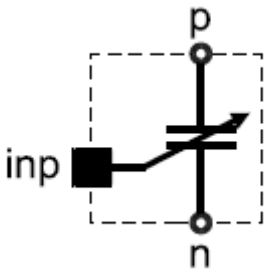
Описание

Переменный конденсатор управляется входным сигналом TDF.

Определение

```
sca_eln::sca_tdf::sca_c( nm, scale, q0 );
sca_eln::sca_tdf_c( nm, scale, q0 );
```

Символ



Уравнения

$$i_{p,n}(t) = scale \cdot \frac{d(inp \cdot v_{p,n}(t) + q_0)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы
q0	double	0.0	Первоначальный заряд в кулоне

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления TDF

A.6.16. sca_eln::sca_tdf::sca_l, sca_eln::sca_tdf_l

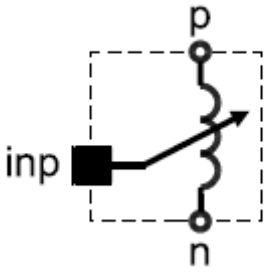
Описание

Переменный индуктор, управляемый входным сигналом TDF.

Определение

```
sca_eln::sca_tdf::sca_l( nm, scale, phi0 );
sca_eln::sca_tdf_l( nm, scale, phi0 );
```

Символ



Уравнения

$$v_{p,n}(t) = scale \cdot \frac{d(inp \cdot i_{p,n}(t) + phi_0)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы
phi0	double	0.0	Начальный магнитный поток в Вебере

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления TDF

A.6.17. sca_eln::sca_tdf::sca_rswitch, sca_eln::sca_tdf_rswitch

Описание

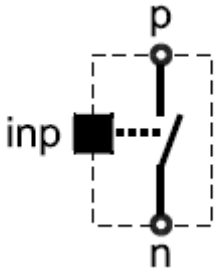
Переключатель управляется входным сигналом TDF.

Определение

```
    sca_eln::sca_tdf::sca_rswitch( nm, ron, roff,
off_state );
```

sca_eln::sca_tdf_rswitch(nm, ron, roff, off_state);

Символ



Уравнения

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
ron	double	0.0	Сопротивление включенного состояния в Ом
roff	double	**sca_util::SCA_INFINITY**	Сопротивление выключенного состояния в Ом
off_state	bool	false	Определите, какая позиция является нерабочим положением

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
ctrl	**sca_tdf::sca_in<T>**	bool	Вход управления TDF

A.6.18. sca_eln::sca_tdf::sca_vsource, sca_eln::sca_tdf_vsource

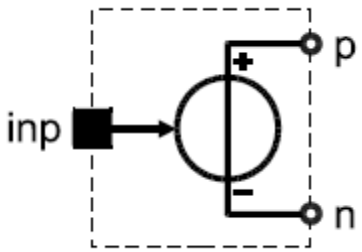
Описание

Источник напряжения, управляемый входным сигналом TDF.

Определение

```
sca_eln::sca_tdf::sca_vsource( nm, scale );
sca_eln::sca_tdf_vsource( nm, scale );
```

Символ



Уравнение

$$v_{p,n}(t) = scale \cdot inp$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления TDF

A.6.19. sca_eln::sca_tdf::sca_isource, sca_eln::sca_tdf_isource

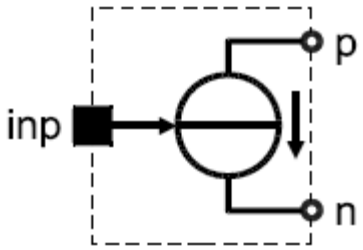
Описание

Источник тока, управляемый входным сигналом TDF.

Определение

```
sca_eln::sca_tdf::sca_isource( nm, scale );
sca_eln::sca_tdf_isource( nm, scale );
```

Символ



Уравнение

$$i_{p,n}(t) = scale \cdot inp$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления TDF

A.6.20. sca_eln::sca_tdf::sca_vsink, sca_eln::sca_tdf_vsink

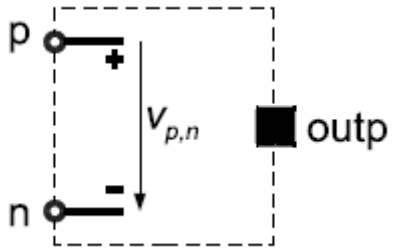
Описание

Преобразует напряжение в выходной сигнал TDF.

Определение

```
sca_eln::sca_tdf::sca_vsink( nm, scale );
sca_eln::sca_tdf_vsink( nm, scale );
```

Символ



Уравнение

Ни одно уравнение не добавлено в систему уравнений.

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
outp	**sca_tdf::sca_in<T>**	double	TDF выход

A.6.21. sca_eln::sca_tdf::sca_isink, sca_eln::sca_tdf_isink

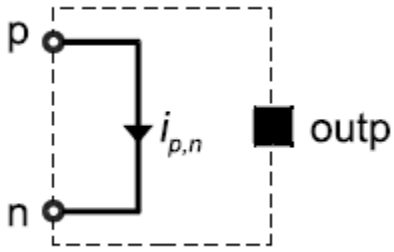
Описание

Преобразует ток в выходной сигнал TDF.

Определение

```
sca_eln::sca_tdf::sca_isink( nm, scale );
sca_eln::sca_tdf_isink( nm, scale );
```

Символ



Уравнение

$$v_{p,n}(t) = 0$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
outp	**sca_tdf::sca_in<T>**	double	TDF выход

A.6.22. sca_eln::sca_de::sca_r, sca_eln::sca_de_r

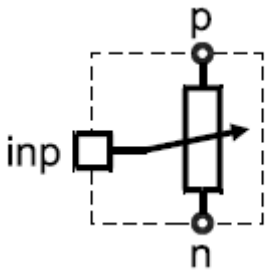
Описание

Переменный резистор управляется входным сигналом дискретного события.

Определение

```
sca_eln::sca_de::sca_r( nm, scale );
sca_eln::sca_de_r( nm, scale );
```

Символ



Уравнение

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t)$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления дискретным событием

A.6.23. sca_eln::sca_de::sca_c, sca_eln::sca_de_c

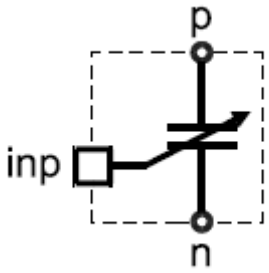
Описание

Переменный конденсатор, управляемый входным сигналом дискретного события.

Определение

```
sca_eln::sca_de::sca_c( nm, scale, q0 );
sca_eln::sca_de_c( nm, scale, q0);
```

Символ



$$i_{p,n}(t) = scale \cdot \frac{d(inp \cdot v_{p,n}(t) + q_0)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы
q0	double	0.0	Начальный заряд в Кулоне

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления дискретным событием

A.6.24. sca_eln::sca_de::sca_l, sca_eln::sca_de_l

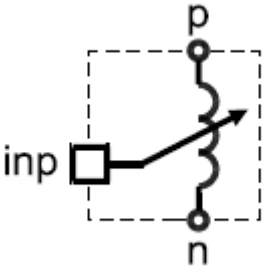
Описание

Переменный индуктор, управляемый входным сигналом дискретного события.

Определение

```
sca_eln::sca_de::sca_l( nm, scale, phi0 );
sca_eln::sca_de_l( nm, scale, phi0 );
```

Символ



Уравнение

$$v_{p,n}(t) = scale \cdot \frac{d\left(inp \cdot i_{p,n}(t) + phi_0\right)}{dt}$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы
phi0	double	0.0	Начальный магнитный поток в Вебере

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления дискретным событием

A.6.25. sca_eln::sca_de::sca_rswitch, sca_eln::sca_de_rswitch

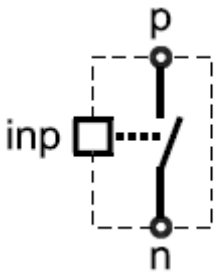
Описание

Переключатель, управляется входным сигналом дискретного события.

Определение

```
    sca_eln::sca_de::sca_rswitch( nm, ron, roff,
off_state );
    sca_eln::sca_de_rswitch( nm, ron, roff, off_state );
```

Символ



Уравнение

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases}$$

Параметры

Имя	**Тип**	**По умолчанию**	**Описание**
nm	**sc_core::sc_module_name**		Имя модуля
ron	double	0.0	Сопротивление включённого состояния в Ом
roff	double	**sca_util::SCA_INFINITY**	Сопротивление выключенного состояния в Ом
off_state	bool	false	Определите, какая позиция является нерабочим положением

Порты

Имя	**Интерфейс**	**Тип/Характер**	**Описание**
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
ctrl	**sca_tdf::sca_in<T>**	bool	Вход управления дискретным событием

A.6.26. sca_eln::sca_de::sca_vsource, sca_eln::sca_de_vsource

Описание

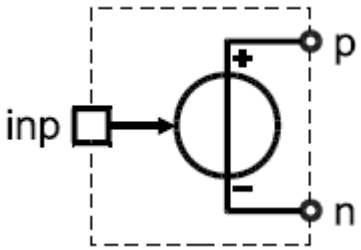
Источник напряжения, управляемый входным сигналом дискретного события.

Определение

```
sca_eln::sca_de::sca_vsource( nm, scale );
```

```
sca_eln::sca_de_vsource( nm, scale );
```

Символ



Уравнение

$$v_{p,n}(t) = scale \cdot inp$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления дискретным событием

A.6.27. sca_eln::sca_de::sca_isource, sca_eln::sca_de_isource

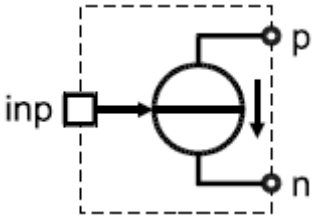
Описание

Источник тока, управляемый входным сигналом дискретного события.

Определение

```
sca_eln::sca_de::sca_isource( nm, scale );
sca_eln::sca_de_isource( nm, scale );
```

Символ



Уравнение

$$i_{p,n}(t) = scale \cdot inp$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
inp	**sca_tdf::sca_in<T>**	double	Вход управления дискретным событием

A.6.28. sca_eln::sca_de::sca_vsink, sca_eln::sca_de_vsink

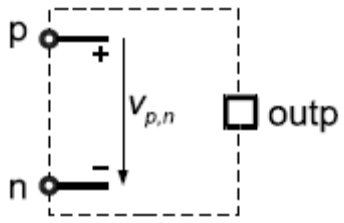
Описание

Преобразует напряжение в выходной сигнал дискретного события.

Определение

```
sca_eln::sca_de::sca_vsink( nm, scale );
sca_eln::sca_de_vsink( nm, scale );
```

Символ



Уравнение

Ни одно уравнение не добавлено в систему уравнений.

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
outp	**sca_tdf::sca_ out<T>**	double	Вывод дискретного события

A.6.29. sca_eln::sca_de::sca_isink, sca_eln::sca_de_isink

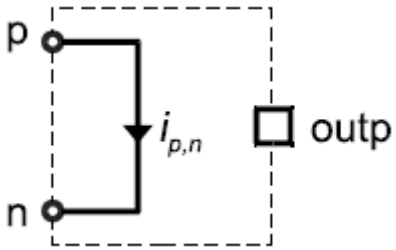
Описание

Преобразует ток в выходной сигнал дискретного события

Определение

```
sca_eln::sca_de::sca_isink( nm, scale );
sca_eln::sca_de_isink( nm, scale );
```

Символ



Уравнение

$$v_{p,n}(t) = 0$$

Параметры

Имя	Тип	По умолчанию	Описание
nm	**sc_core::sc_module_name**		Имя модуля
scale	double	1.0	Коэффициент шкалы

Порты

Имя	Интерфейс	Тип/Характер	Описание
p	**sca_eln::sca_terminal**	электрический	Положительный терминал
n	**sca_eln::sca_terminal**	электрический	Отрицательный терминал
outp	**sca_tdf::sca_ out<T>**	double	Вывод дискретного события

Приложение B. Символы и графические изображения

В этом приложении даётся обзор символов и графических изображений, используемых в данном руководстве пользователя.

В случае, если производные блок-схемы или электрические сети извлечены из этого руководства пользователя, настоятельно рекомендуется использовать эти символы и графические изображения в согласованном порядке.

Символы для отдельных примитивов LSF и ELN приведены в Приложении A.

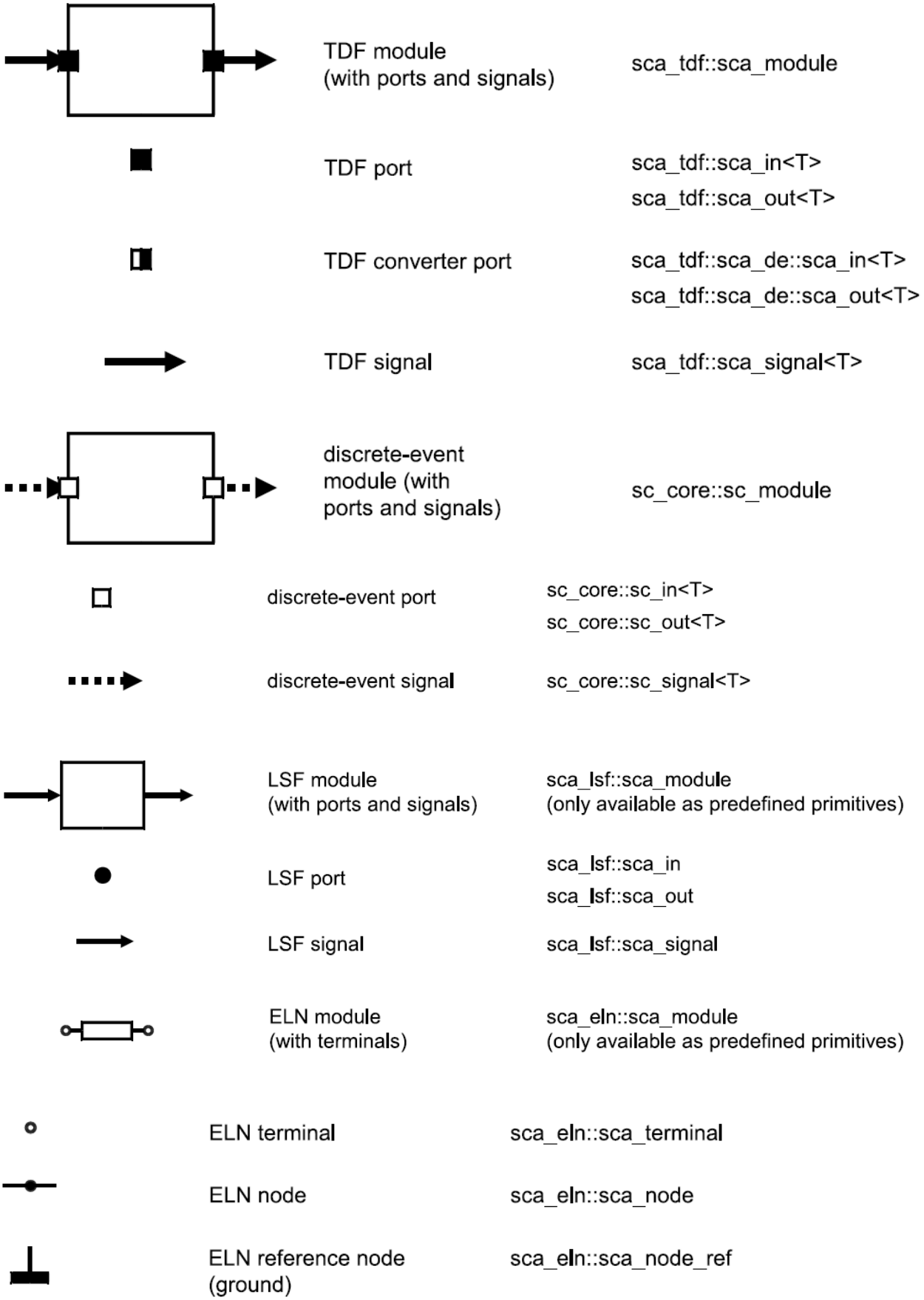


Рисунок Б.1. Символы и графические изображения TDF, LSF, ELN и элементов дискретных событий.

Библиография

1. SystemC – AMS extensions User's Guide. Open SystemC Initiative (OSCI). 2010. p. 166. [Электронный ресурс]. http://www.systemc.org

2. Banerjee A., Sur B. SystemC and SystemC – AMS in Practice. SystemC 2.3, 2.2 and SystemC – AMS 1.0. New York, Dordrecht, London: Springer Cham Heidelberg, 2014. p. 462.

3. Grimm Ch., Barnasconi M., Vachoux A., Einwich K. An Introduction to Modeling Embedded Analog/Mixed-Signal Systems using SystemC – AMS Extensions. Open SystemC Initiative (OSCI). 2008. p. 12. [Электронный ресурс]. https://publik.tuwien.ac.at/files/PubDat_171466.pdf

4. Standard SystemC – AMS extensions Language Reference Manual. Open SystemC Initiative (OSCI). 2010. p. 152. [Электронный ресурс]. http://www.systemc.org

5. Requirements specification for SystemC Analog Mixed Signal (AMS) extensions Version 2.1. Open SystemC Initiative (OSCI). March 8, 2010. p. 27. [Электронный ресурс]. http://www.systemc.org

6. Functional specification for SystemC 2.0. April 5, 2002. [Электронный ресурс]. http://www.systemc.org

7. Алехин В.А. Проектирование электронных систем с использованием SystemC и SystemC-AMS. //Российский технологический журнал, 2020;8(4). С. 81-97.

8. Алехин В.А. SystemC. Моделирование электронных систем. Учебное пособие для вузов. М.: Горячая линия – Телеком, 2018. 320 с.

9. Алехин В.А., Быков И.А. Применение SystemC для проектирования электронных систем. Сборник трудов Международной научно-практической конференции «Наука, образование, общество». Тамбов, 31 марта 2018. с. 7–11.