УДК 681.3.06 (07)
У 912

КАФЕДРА МАТЕМАТИЧЕСКОГО ОБЕСПЕЧЕНИЯ И ПРИМЕНЕНИЯ ЭВМ

ФАВТ

№ 3451

МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Государственное образовательное учреждение
высшего профессионального образования

ТАГАНРОГСКИЙ ГОСУДАРСТВЕННЫЙ
РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Учебно-методическое пособие
по курсу
АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ СИГНАЛЬНЫХ ПРОЦЕССОРОВ

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРОЦЕССОРОВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

Часть 1

Для студентов специальностей 220400, 351500

Таганрог 2003

УДК 681.3.06 (076.5) + 681.325.5. (076.5)

Составитель Н.Ш. Хусаинов

Учебно-методическое пособие по курсу “Архитектура и программирование сигнальных процессоров". Разработка программного обеспечения процессоров цифровой обработки сигналов. Часть 1. Таганрог: Изд-во ТРТУ, 2003. 80с.

Пособие посвящено вопросам разработки программного обеспечения процессоров цифровой обработки сигналов (ЦОС) и представляет интерес для инженеров, разработчиков и программистов в области ЦОС, систем реального времени, параллельного программирования, машинно-ориентированного программирования.

В первой части рассматриваются принципы проектирования системы цифровой обработки сигналов на основе сигнальных процессоров. Основное внимание уделяется этапам и средствам разработки программного обеспечения процессоров в системе ЦОС. На примере интегрированной среды VisualDSP++ для процессоров фирмы Analog Devices Inc. проанализированы возможности современных сред разработки по созданию, трансляции и отладке программного кода, в том числе для многопроцессорных систем. Значительное внимание уделено фундаментальным различиям в построении и функционировании программ для DSP-процессоров и для универсальных компьютеров. Приведены примеры организации обработки сигналов одновременно с организацией ввода/вывода данных в реальном масштабе времени.

Для студентов специальностей 220400, 351500, изучающих курс "Архитектура и программирование сигнальных процессоров". Может быть полезна для студентов ряда специальностей и инженеров, занимающихся проектированием систем ЦОС и разработкой программного обеспечения для них.

Ил.40. Библиогр.: 16 назв.

Рецензент В.Е.Золотовский, д-р техн. наук, профессор кафедры ВТ ТРТУ.

Хусаинов Наиль Шавкятович

Учебно-методическое пособие
по курсу
АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ
СИГНАЛЬНЫХ ПРОЦЕССОРОВ

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРОЦЕССОРОВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

Часть 1

Для студентов специальностей 220400, 351500

Ответственный за выпуск Хусаинов Н.Ш.
Редактор Лунева Н.И.
Корректор Селезнева Л.И., Чиканенко Л.В.

ЛР № 020565 от 23.06.1997г. Подписано к печати . .03г.
Формат 60x84 1/16 Бумага офсетная.
Офсетная печать. Усл.п.л. – 5,0. Уч.-изд. л. – 4,8.
Заказ № Тираж 200 экз.

"С"

Издательство Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Некрасовский, 44
Типография Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Энгельса, 1

УДК 681.3.06 (076.5) + 681.325.5. (076.5)

Составитель Н.Ш. Хусаинов

Учебно-методическое пособие по курсу “Архитектура и программирование сигнальных процессоров". Разработка программного обеспечения процессоров цифровой обработки сигналов. Часть 1. Таганрог: Изд-во ТРТУ, 2003. 80с.

Пособие посвящено вопросам разработки программного обеспечения процессоров цифровой обработки сигналов (ЦОС) и представляет интерес для инженеров, разработчиков и программистов в области ЦОС, систем реального времени, параллельного программирования, машинно-ориентированного программирования.

В первой части рассматриваются принципы проектирования системы цифровой обработки сигналов на основе сигнальных процессоров. Основное внимание уделяется этапам и средствам разработки программного обеспечения процессоров в системе ЦОС. На примере интегрированной среды VisualDSP++ для процессоров фирмы Analog Devices Inc. проанализированы возможности современных сред разработки по созданию, трансляции и отладке программного кода, в том числе для многопроцессорных систем. Значительное внимание уделено фундаментальным различиям в построении и функционировании программ для DSP-процессоров и для универсальных компьютеров. Приведены примеры организации обработки сигналов одновременно с организацией ввода/вывода данных в реальном масштабе времени.

Для студентов специальностей 220400, 351500, изучающих курс "Архитектура и программирование сигнальных процессоров". Может быть полезна для студентов ряда специальностей и инженеров, занимающихся проектированием систем ЦОС и разработкой программного обеспечения для них.

Ил.40. Библиогр.: 16 назв.

Рецензент В.Е.Золотовский, д-р техн. наук, профессор кафедры ВТ ТРТУ.

1. ЭТАПЫ И СРЕДСТВА РАЗРАБОТКИ ПО ДЛЯ СИСТЕМ ЦОС

1.1. Обобщенная структура системы ЦОС и принципы ее функционирования

Хотя процессор цифровой обработки сигналов (Digital Signal Processor, DSP) предназначен в первую очередь для высокоэффективного выполнения специализированных вычислительных операций над сигналами, он должен содержать средства для взаимодействия как с периферийными устройствами, обеспечивающими ввод/вывод и преобразование этих сигналов (АЦП и ЦАП), так и, возможно, с другими процессорами. При этом важно, чтобы операции ввода и вывода данных также выполнялись с высокой скоростью с минимальным задействованием процессорного ядра.

Структура типичной системы ЦОС приведена на рис. 1. Как видим, в дополнение к функциям обработки потоков данных, поступающих с АЦП или аппаратных кодеков, процессор должен "уметь" обрабатывать управляющие и информационные сигналы, поступающие от хост-процессора или средств контроля пользователя. Обычно процессор функционирует синхронно с АЦП или кодеком в целях согласования регулярных временных интервалов дискретизации.

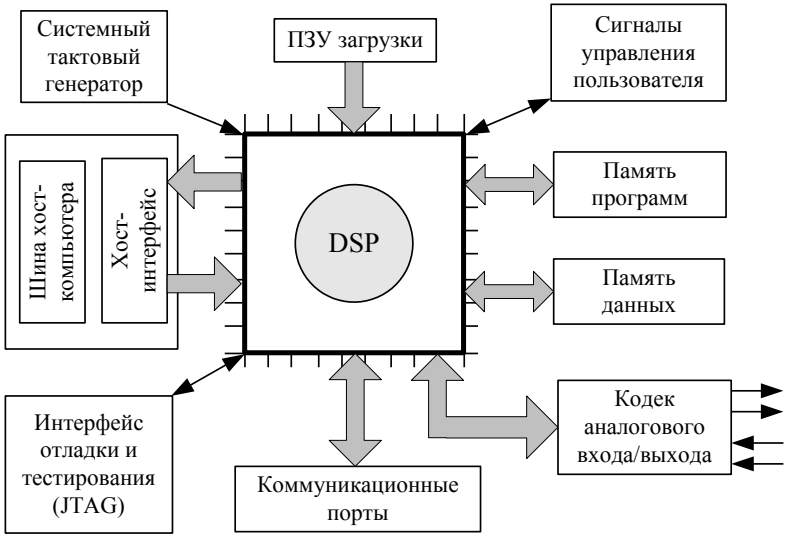


Рис.1. Обобщенная структура системы ЦОС

При получении очередного входного отчета процессор должен выполнить необходимую обработку и выдать очередное выходное значение на ЦАП или выход кодека. В то же время на процессор поступают прерывания, например, от других процессоров или хост-процессора, которые могут изменять логику выполнения алгоритма и его параметры. Подобные события отличаются от

информационных сигналов (потоков) тем, что они происходят нерегулярно (асинхронно). Таким образом, DSP-процессор должен уметь обрабатывать как синхронные события (например, поступление отсчетов с постоянным временным интервалом), так и асинхронные ("случайные" управляющие сигналы от внешних устройств), причем последние должны обрабатываться не оказывая влияния на скорость обработки регулярных событий. В противном случае часть информации может быть потеряна.

Для реализации такого способа взаимодействий могут использоваться следующие подходы.

1. Ввод/вывод с последовательным опросом. DSP-процессор опрашивает внешние устройства путем анализа состояния своих сигнальных выводов или определенных ячеек памяти. Если считается, что событие произошло (например, получено очередное слово данных), то вызывается соответствующая процедура обработки. Данный подход очень прост, однако его главным недостатком является затрата больших ресурсов процессора на постоянную регулярную проверку каких-либо условий (вне зависимости от того, произошли они в действительности или еще нет), что, естественно, снижает его возможности по выполнению "полезной" работы.

2. Управление по прерываниям. Данный режим возможен, если сигнальным ножкам процессора поставить в соответствие определенные вектора прерывания (и соответствующие обработчики прерываний). В этом случае процессор "отвлекается" от вычислительной обработки только в том случае, когда определенное событие (внешнее прерывание или сигнал от таймера) действительно произошло. Такой подход требует более тщательного программирования, но существенно повышает количество "полезного" времени, идущего на обработку данных.

3. При взаимодействии с внешними устройствами, особенно в современных процессорах, могут широко использоваться механизмы прямого доступа к памяти, которые позволяют организовать обмен большими объемами (блоками) данных между коммуникационными портами процессора и внутренней памятью без использования ресурсов процессорного ядра параллельно с ходом выполнения основной программы. Это наиболее эффективный способ ввода и вывода информации, однако его применение обычно целесообразно при вводе/выводе потоков данных, а не сигнальной информации.

1.2. Этапы проектирования системы цифровой обработки сигналов

Процесс разработки системы цифровой обработки сигналов обычно представляет собой последовательность этапов, начинающихся с постановки задачи и анализа требований к системе и завершающихся разработкой аппаратной схемы и соответствующего программного обеспечения к ней, которые отвечают всем начальным требованиям (рис.2).

Рис.2. Упрощенная схема процесса проектирования системы ЦОС

Основной задачей на первых шагах является формирование требований к аппаратной платформе, периферийным устройствам и способам обмена данными между устройствами внутри системы, а также между устройствами системы и "внешним миром". Эти требования определяются в первую очередь поставленной перед разработчиками задачей и областью применения проектируемой системы.

Наличие такого "заказа" позволяет существенно ограничить перечень аппаратных средств (DSP-процессоров и периферии), которые могут быть использованы для решения поставленной задачи, и исключить из дальнейшего рассмотрения аппаратные компоненты, не удовлетворяющие требованиям, например, по точности вычислений, производительности, стоимости, потребляемой мощности. Окончательный выбор аппаратной платформы (главным образом – DSP-процессора) осуществляется, принимая во внимание следующие объективные и субъективные факторы:

- базовые требования: полная и доступная документация по всем стадиям цикла разработки программного обеспечения, средства разработки ПО на языке низкого уровня, средства проверки работоспособности и отладки системы;

- дополнительные требования: компилятор с языка высокого уровня (C/C++, ADA, FORTRAN) для разработки больших и сложных проектов, подключаемые

библиотеки стандартных процедур, отладка системы в режиме реального времени, длительность цикла создания готового коммерческого продукта, низкая стоимость аппаратных стендов-прототипов, возможность использования рабочих станций стандартной конфигурации (например, на базе персонального компьютера), консультационная поддержка производителя чипа (курсы, семинары), личные предпочтения (опыт работы).

В конечном счете должно быть выбрано устройство, которое позволяет получить наиболее экономически привлекательное решение в отведенные для реализации проекта сроки. Спроектированное на его основе аппаратное решение и разработанное и отлаженное программное обеспечение в конечном счете проверяются на работоспособность и правильность выполнения поставленной задачи.

Вполне реальна ситуация, когда выбор DSP-процессора для проектируемой системы превращается в итерационный процесс: не всегда получается сделать правильный выбор аппаратной платформы с первого раза. Зачастую уже на этапе тестирования производительности системы и разработанного ПО в режиме реального времени оказывается, например, что можно использовать менее производительный (более дешевый) процессор, что имеет особенно большое значение при разработке систем массового назначения (бытовая техника, персональные мобильные средства связи и т.п.). Чтобы избежать лишних затрат временных, человеческих и материальных ресурсов на реализацию всех этапов цикла разработки системы ЦОС и, как следствие, сократить стоимость процесса разработки и время выпуска готового продукта на рынок (time-to-market) разработчики DSP-чипов и их партнеры (сторонние фирмы-разработчики, third party developers) предоставляют широкий набор вспомогательных программных и аппаратных средств – интегрированных систем разработки и отладки, оценочных плат и интегрированных модулей, внутрисхемных эмуляторов и т.д.

1.3. Средства разработки ПО для систем ЦОС

1.3.1. Набор базовых средств разработки

Разработка программного обеспе ьчения для DSP-процессора требует использования целого набора базовых средств – утилит, отвечающих за написание программного кода, его перевод в машинную форму, понятную для процессора, а также загрузку полученного машинного кода в процессор или преобразование в формат для последующей записи в ППЗУ (рис.3).

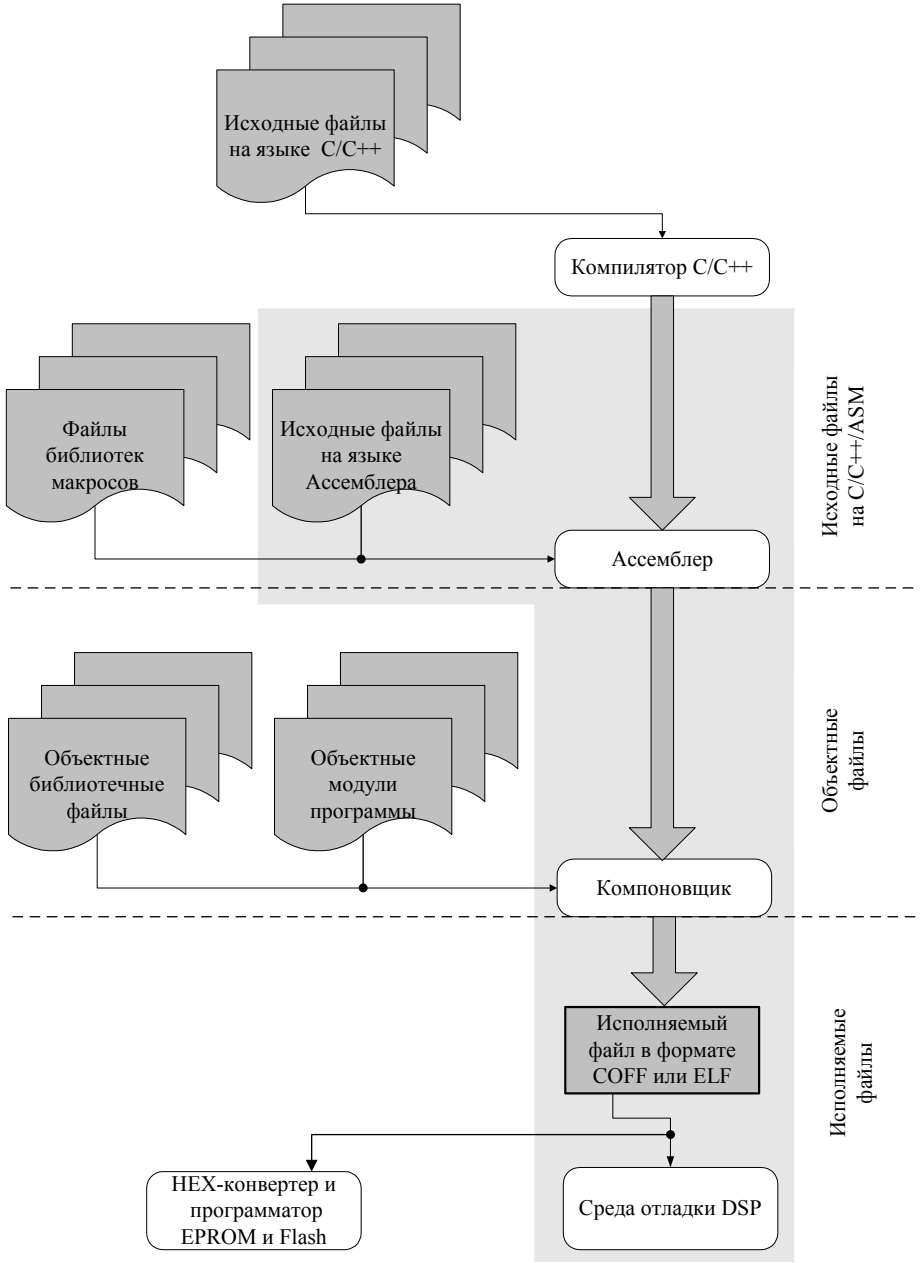


Рис.3. Этапы разработки программного обеспечения

Во многом этапы разработки программного обеспечения системы ЦОС напоминают процесс разработки ПО для современных персональных компьютеров, работающих, например, под операционной системой Windows.

Из текста исходных модулей программы на языке высокого и/или низкого уровня сначала формируются один или несколько объектных файлов, которые содержат коды исполняемых инструкций, но еще не "привязаны" к конкретным адресам в пространстве памяти DSP-процессора. Основными функциями компоновщика являются установление соответствия между логическими сегментами программы и физическими адресами памяти, а также генерация исполняемого файла для процессора. Эту операцию линкер выполняет на основе задаваемого описания архитектуры целевой системы. Выходом компоновщика является бинарный файл(ы) обычно в COFF- или ELF-формате. Полученный таким образом и отлаженный с использованием программного симулятора исполняемый файл должен быть загружен для выполнения в DSP-процессор. Это может быть сделано одним из двух способов: либо в режиме отладки с использованием специальной программы-загрузчика (на промежуточных этапах отладки), либо путем прошивки исполняемого файла в ППЗУ или записи во Flash-память с использованием специального программатора ППЗУ или Flash-программатора (используется на конечном этапе отладки для завершающего тестирования). Основным отличием использования Flash-памяти от обычного ППЗУ является то, что для записи в Flash-память не нужен специальный программатор – в DSP-процессор загружается специальная стандартная утилита работы с Flash-памятью, посредством которой сгенерированный программой-загрузчиком файл может быть записан во Flash-память.

Рассмотрим более подробно назначение каждой системной утилиты:

1. Компиляторы языка C/C++ для систем ЦОС. Как в любой среде разработки компилятор C/C++ предназначен для решения следующих основных задач:

- обработки исходных файлов с программой на языке C/C++ и генерации машинно-ориентированной версии программного кода (на языке ассемблера) и файлов с объектным кодом;
- генерации и помещения перемещаемого кода и отладочной информации в объектные файлы;
- генерации перемещаемых сегментов памяти данных и памяти программ для их размещения компоновщиком в памяти процессора.

Язык C, который "понимают" компиляторы фирм Analog Devices и Texas Instruments, представляет собой стандартизованный диалект языка C "ANSI/ISO C". Версия языка C++ за некоторыми отличиями также близка к стандарту ISO/IEC 14882-1998. Характер этих отличий определяется в первую очередь особенностями использования кода во встроенных системах. К ним, например, относятся отсутствие обработки исключений, отсутствие идентификации типа объекта во время выполнения (ограничения в реализации динамического связывания), трудности использования шаблонов и др.

2. Ассемблер. Основная функция ассемблера – на основе получаемого на входе файла с текстом программы на языке ассемблера генерация двоичного перемещаемого объектного кода в одном из стандартных форматов: COFF (Common Object File Format, для средств разработки фирмы Texas Instruments) или ELF (Executable and Linkable Format, для средств разработки фирмы Analog Devices). "Побочным" продуктом функционирования ассемблера является файл листинга (необязательно).

Как правило, перед ассемблированием программы выполняется ее предварительная обработка препроцессором (самостоятельной или интегрированной в ассемблер утилитой), что позволяет использовать в тексте программы на ассемблере макрокоманды и директивы и иногда существенно облегчает написание кода и возможность его повторного использования.

В состав некоторых средств разработки, например, Code Composer Studio фирмы Texas Instruments, входит еще одна специальная утилита – оптимизирующий ассемблер. Ее использование позволяет программисту писать программу на "линейном ассемблере", уделяя гораздо меньше внимания возможным потерям производительности вследствие простоя процессора при выполнении условных переходов, попытках распараллеливания команд и т.п. Оптимизирующий ассемблер преобразует исходный "линейный" код с учетом архитектуры процессора с целью повышения производительности программы.

В отличие от языков высокого уровня (обычно – языка C/C++), сходных в различных средах разработки для различных процессоров, синтаксисы языков ассемблера разных производителей существенно различаются. Доступность языка низкого уровня и понятность программного кода, написанного на нем, является одним из важных критериев выбора того или иного производителя или семейства процессоров.

3. Архиватор (библиотекарь). Утилита поддержки библиотек позволяет создавать собственные библиотеки ресурсов (функций или модулей) для последующего повторного использования в различных проектах. Стандартными функциями архиватора являются создание библиотечного файла на основе одного или нескольких программных файлов, добавление/удаление/замену программных файлов в библиотеке, доступ к программному файлу в библиотеке.

Архиваторы обычно обязательно поддерживают библиотеки перемещаемых объектных файлов (формата ELF или COFF) и иногда библиотеки макроопределений.

Чтобы программный код (например, какая-либо функция) мог быть помещен в библиотеку, он должен отвечать некоторым несложным требованиям, описанным в руководстве разработчика и касающимся, в основном, правильного определения области видимости функции и ее доступности. Для доступа к библиотечной функции необходимо подключить соответствующий библиотечный файл и должным образом объявить эту функцию в теле программы.

4. Компоновщик. Компоновщик (линкер) генерирует исполняемые двоичные

модули в формате COFF или ELF[1] путем компоновки нескольких объектных модулей и подключаемых библиотечных файлов. При этом размещение отдельных участков кода и данных в памяти системы обязательно задается специальным файлом описания архитектуры. Иногда этот файл описания для компоновщика в явном виде не создается. В этом случае используется файл описания архитектуры по умолчанию.

Несколько исполняемых модулей после компоновки получаются в случае использования в качестве целевой архитектуры многопроцессорной системы: каждому процессору обычно соответствует свой исполняемый модуль.

Возможна также генерация компоновщиком дополнительных файлов, например файлов оверлеев, файлов с содержимым разделяемой между несколькими процессорами памяти и т.п.

5. Загрузчик. Утилита "загрузчик" (loader) используется на завершающей стадии разработки и отладки программного обеспечения для подготовки его к загрузке в целевую систему. Загрузчик выполняет обработку исполняемых файлов, генерируя специальный исполняемый загрузочный файл (boot-loadable file). Этот файл содержит основные параметры приложения (размеры сегментов, их размещение и т.п.), и используется для автоматической загрузки исполняемой программы во внутреннюю память процессора после включения питания или программного сброса. Обычно сигнальный процессор поддерживает несколько различных способов загрузки: из внешнего ПЗУ, через хост-компьютер, через линк-порты и т.п. В упрощенном виде процесс загрузки выглядит следующим образом (на примере загрузки из внешнего ПЗУ процессоров семейства SHARC ADSP):

- на основе анализа исполняемых файлов утилита типа loader формирует небольшой фрагмент программного кода фиксированного размера (256 48-битовых инструкций) – ядро загрузки;
- размещение ядра загрузки производится путем прошивки внешнего ПЗУ (для загрузки из ПЗУ) по жестко фиксированному адресу. На этот адрес по умолчанию "настраиваются" внутренние регистры процессора при включении питания;
- при включении питания на один из выводов процессора, определяющий источник загрузки, подается соответствующая комбинация сигналов, определяющая источник загрузки (ПЗУ, хост-компьютер, линк-порт). На основании этого сигнала процессор автоматически инициирует DMA-пересылку 256 48-разрядных слов (ядра загрузки) из жестко фиксированного адреса внешнего ПЗУ в заданную область внутренней памяти;
- осуществляется передача управления на первую инструкцию ядра загрузки;
- поскольку программный код ядра загрузки был сгенерирован (на основе шаблона) под конкретное приложение и архитектуру системы, то он выполняет

[1] Обратите внимание, что файлы в форматах COFF и ELF могут содержать как перемещаемый объектный код, так и исполняемый программный код.

загрузку разработанного приложения из указанного источника, причем обычно инструкции приложения записываются поверх загрузчика и по завершении этого процесса получают управление.

У большинства DSP-процессоров имеется возможность выполнения программы из внешней памяти (например, если память на кристалле отсутствует). В этом случае использование утилиты Loader не требуется – программа будет выполняться непосредственно из внешней памяти, что может привести к снижению производительности (при медленной внешней памяти), но совершенно необходимо при значительном объеме программного кода и/или небольшой внутренней памяти процессора. Во всех остальных случаях предпочтительнее использовать загрузку исполняемого кода во внутреннюю память процессора и выполнение программы из внутренней памяти.

6. Сплиттер. Утилита сплиттер (splitter, hex conversion utility) используется на последнем этапе разработки и отладки программного обеспечения для подготовки к прошивке его в ППЗУ. В зависимости от типа микросхемы ППЗУ и программатора ему на вход должен быть подан файл в одном из стандартных форматов программаторов ППЗУ, например Motorola-S, Intel-Hex и т.п. Получая бинарный исполняемый файл, сплиттер генерирует двоичный файл в требуемом формате.

7. Программатор Flash. Как известно, flash-память представляет собой энергонезависимую память, доступную для чтения, записи и стирания. Обычно flash-память используется для хранения программного кода, который загружается в DSP-процессор при включении питания, и данных, требующих длительного хранения. Если flash-память соединена соответствующим образом с DSP-процессором, то последний может быть использован для ее пере-программирования.

1.3.2. Критерии выбора языка программирования

Программы для систем ЦОС разрабатываются на тех же языках программирования, что и большинство других инженерных и научных приложений: обычно это C и ассемблер. Программы, написанные на ассемблере, меньше по объему и быстрее, тогда как C-программы проще в разработке и модификации.

В традиционных программах, разрабатываемых, например, для персональных компьютеров, в первую очередь используется язык высокого уровня и только в некоторых случаях для разработки некоторых критичных ко времени участков кода обращаются к ассемблеру. Программное обеспечение для систем ЦОС отличается от традиционного программного обеспечения тем, что, во-первых, критическим фактором всегда является производительность, во-вторых, объем ЦОС-программ сравнительно невелик (как правило, не более нескольких сотен строк кода) и ограничивается объемом внутренней или внешней памяти системы.

Пример повышения производительности при написании программного кода на ассемблере по сравнению с кодом на С при вычислении скалярного произведения двух векторов *x* и *y* на процессоре ADSP SHARC приведен ниже. Как видно, оптимизированный код в несколько раз быстрее, чем код, полученный путем непосредственного "перевода" программы на С в программу на ассемблере.

Исходный текст на С:

```
#define LEN 20
float dm x[LEN];
float pm y[LEN];
float result;
main()
{
int n;
float s;
for (n=0;n<LEN;n++)
    s += x[n]*y[n];
result = s;
}
```

*Неоптимизированный ассемблерный код (время выполнения 4*LEN + 4):*

```
i12 = _y;     i4 = _x;
lcntr = LEN, do (pc,4) until lce;
 f2 = dm(i4,m6);
 f4 = pm(i12,m14);
 f8 = f2*f4;
 f12 = f8 + f12;
dm(_result) = f12;
```

Оптимизированный ассемблерный код (время выполнения ((LEN-2)+4)+4):

```
i12 = _y;     i4 = _x;
f2 = dm(i4,m6), f4 = pm(i12,m14);
f8 = f2*f4, f2 = dm(i4,m6), f4 = pm(i12,m14);
lcntr = LEN-2, do (pc,1) until lce;
  f12 = f8 + f12, f8 = f2*f4, f2 = dm(i4,m6), f4 = pm(i12,m14);
f12 = f8 + f12, f8 = f2*f4;
f12 = f8 + f12;
```

Одним из способов повышения эффективности программ, написанных на языках С/С++, является использование оптимизированных компиляторов С и С++. Однако следует иметь в виду, что оптимальный код при их использовании получается только для тех фрагментов программы, распараллеливание которых очевидно. Как заявляют сами разработчики сигнальных процессоров и С-компиляторов к ним, производительность кода, сгенерированного компилятором, в среднем от 1,5 до 3 раз уступает производительности кода, написанного "вручную" на ассемблере и оптимизированного под архитектуру конкретного DSP-процессора.

Именно поэтому традиционно значительная часть инженеров и программистов ЦОС-систем отдают предпочтение ассемблеру. Особенно это проявляется при разработке систем для последующего массового коммерческого

производства: более оптимальный программный код позволяет добавить новые функциональные возможности или применить менее производительный (и, соответственно, менее дорогостоящий) процессор. Как результат – при бо́льших затратах на разработку получается более дешевая система, имеющая лучшие перспективы на рынке.

Однако условия работы меняются, и на современном этапе надо учитывать целый ряд изменившихся факторов. С одной стороны, усложнение алгоритмов делает их реализацию на ассемблере все более трудоемкой. С другой стороны, все более жесткие требования ставятся к срокам проведения разработки. Одновременно, усложнение архитектуры самих DSP-процессоров и внедрение параллельных решений требуют от разработчика держать в голове все больший объём информации, отвлекая его от основной задачи – разработки собственно алгоритма.

Ещё одной особенностью является ускорение развития архитектур DSP и более быстрая смена семейств, что требует обеспечить большую степень “платформенной” независимости и лучшую переносимость как между DSP-процессорами одного семейства, так и между семействами сигнальных процессоров. Да и поставляемые производителями DSP-процессоров и средств разработки наборы библиотечных функций, которые написаны на ассемблере, оптимизированы под конкретную архитектуру, имеют стандартный интерфейс для вызова из С-программ, также подталкивают к выводу о том, что на сегодняшний день более эффективным для практического применения подходом является разработка программного кода на языке С с использованием оптимизированных библиотечных функций и ассемблерных вставок в наиболее критичных ко времени выполнения фрагментах программы.

Предположим, что разработан алгоритм "на бумаге" и мы хотим проверить его эффективность. Что делать? Можно, конечно, написать программу на С/С++. Однако гораздо более простой и быстрый путь заключается в использовании для моделирования системы типа Matlab. Во всем мире Matlab является стандартом де-факто при экспериментах с алгоритмами ЦОС. Она включает в себя богатейшие библиотеки готовых функций и процедур, в том числе и для обработки сигналов, изображений. Имеются готовые библиотека блоков для разработки в Matlab программ для процессоров Texas Instruments, Analog Devices, Motorola, причем некоторые из этих библиотек входят в комплект поставки Matlab. Входной язык программирования этой среды во многом похож на С. Кроме того, в состав Matlab входит Simulink, в котором реализована концепция графического программирования, позволяющая создавать программы путем рисования блок-схем алгоритмов (рис.4). Поэтому классический путь создания новых ЦОС-систем выглядит так: Matlab -> С -> ассемблер.

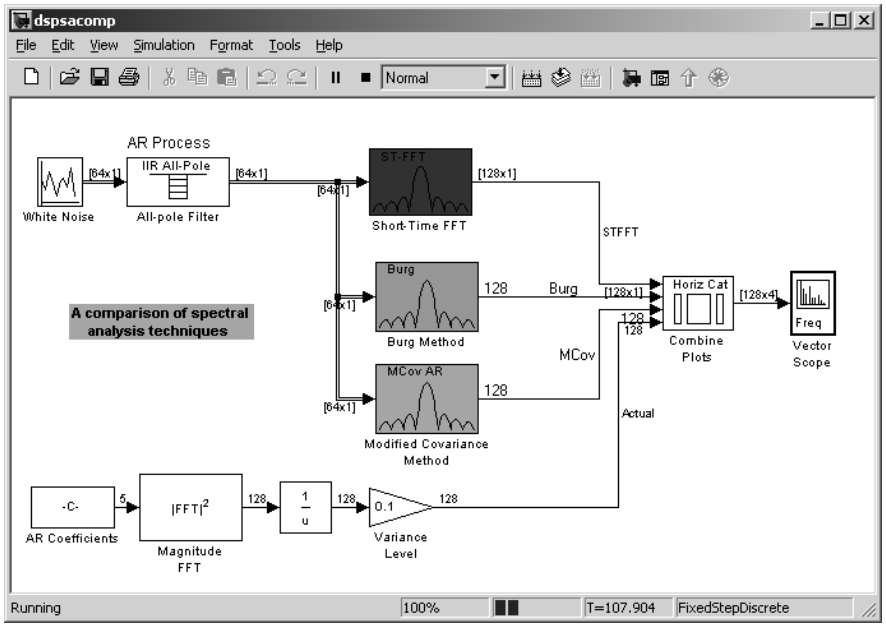


Рис.4. Рабочий экран Matlab Simulink с блоками модели анализа спектральных характеристик

Среди разработчиков систем ЦОС широко распространены и другие программные средства, позволяющие выполнять визуальное проектирование и имитацию работы системы ЦОС, такие, как HyperSignal (фирмы Hypercection Inc.), LabView (фирмы National Industrial) и другие. Особенно широкое применение они находят при проектировании многопроцессорных систем.

1.4. Отладка и тестирование ПО для DSP-процессоров

Тестирование и отладка проекта для системы ЦОС обычно выполняется последовательно с использованием программной и аппаратной отладки (эмуляции). Современные системы разработки ПО для DSP (VisualDSP++ для процессоров фирмы Analog Devices, CodeComposerStudio для процессоров фирмы Texas Instruments, CodeWarrior для процессоров фирмы Motorola) обычно интегрируют в себе эти возможности. Разработка и отладка программного кода для системы ЦОС – циклический процесс с большим количеством итераций.

1. На этапе программной эмуляции (Software Simulation) программный код выполняется под управлением специальной программы – симулятора, имитирующего выполнение DSP-программы на обычном персональном компьютере и позволяющего проверить логику организации программы, правильность выполняемых вычислений, оценить время выполнения отдельных участков программы путем профилирования. Для этого этапа не требуется наличие процессора или каких-либо специальных аппаратных средств. Предоставляя программисту практически неограниченные возможности по анализу хода выполнения программы (просмотр/модификация содержимого памяти и регистров, управление выполнением программы, отображение данных в графическом виде, загрузка/выгрузка содержимого памяти и т.д.), программная эмуляция имеет два основных недостатка:

- выполнение не в реальном времени, что в большинстве случаев не позволяет оценить реальную производительность системы;
- трудности моделирования взаимодействия между процессором и периферийными устройствами.

Поэтому обязательным этапом разработки и тестирования любого алгоритма ЦОС является его загрузка и выполнение в режиме реального времени на реальном DSP-процессоре.

2. На этапе аппаратной эмуляции (Hardware Emulation) в режиме реального времени выполнение программы осуществляется на аппаратной платформе, содержащей DSP-процессор(ы). Основное назначение этого этапа – проверка правильности функционирования программы в режиме реального времени, устранение возможных рассогласований временных интервалов выполнения различных участков программного кода, а также ошибок, связанных с обменами данными между процессором и внешними устройствами, окончательный выбор модели процессора, например, с точки зрения необходимой тактовой частоты. При этом у программиста обычно сохраняются некоторые возможности контроля за ходом выполнения программы, такие как пошаговое выполнение, установка точек останова, просмотр содержимого памяти и регистров. Широко распространены следующие подходы к аппаратной эмуляции:

а) наиболее удобный (и дорогостоящий) способ аппаратной эмуляции – применение внутрисхемного эмулятора, который подключается к плате с установленным процессором через специальный интерфейс, находящийся внутри кристалла процессора и отслеживает выполнение всех действий процессором в реальном времени, не оказывая влияния на ход выполнения программы.

В процессорах фирм Analog Devices и Texas Instruments используется стандартный "внутрикристальный" отладочный порт, совместимый со стандартом JTAG (IEEE 1149.1), который подключается через USB, PCI или Ethernet-интерфейс к хост-процессору (персональному компьютеру) посредством внутрисхемных эмуляторов типа EZ-ICE (для процессоров ADSP) или XDS, SDSP (для процессоров TI), как показано на рис.5. Отличием внутрикристального порта OnCE фирмы Motorola является то, что для аппаратной отладки не требуется дополнительного внешнего эмулятора – внутрикристальный эмулятор OnCE поддерживает интерфейс RS-232 и напрямую соединяется с ПК.

Управление ходом выполнения программы на DSP-процессоре выполняется из интегрированной среды разработки (Integrated Development Environment, IDE)

в реальном времени (отображение/модификация значений регистров и ячеек памяти, точки останова, пошаговый режим выполнения).

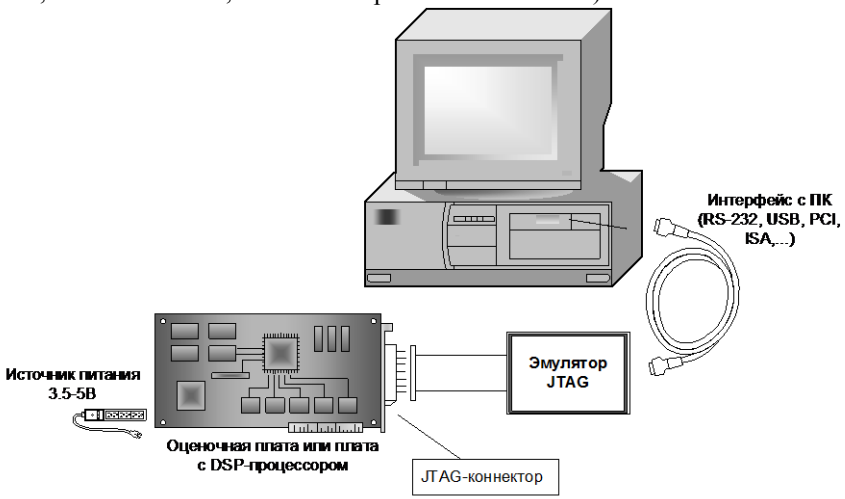


Рис.5. Схема подключения внутрисхемного JTAG-эмулятора при отладке системы

Кроме разработки новой системы ЦОС, такой подход может быть полезен при поиске неисправностей в уже функционирующей системе. Его единственным недостатком является высокая стоимость: цена внутрисхемного эмулятора составляет от 1 до 6 тыс. долл.;

б) более дешевым способом аппаратной эмуляции является использование так называемой оценочной платы (Evaluation Board). Такие средства (EZ-LAB, EZLITE для процессоров фирмы Analog Devices, DSK и DSP Starter Kit для процессоров фирмы Texas Instruments, EVM для процессоров фирмы Motorola) обычно имеют стандартную конфигурацию (обычно это DSP определенной модели, звуковой порт, внешняя память, ПЗУ, интерфейс для подключения к ПК и блок питания). Особым элементом такой системы является небольшая программа-монитор, обычно размещаемая в ПЗУ на плате и загружаемая в DSP-процессор при включении питания. Она представляет собой упрощенный аналог операционной системы, отвечающей главным образом за организацию взаимодействия между процессором и ПК: загрузка/выгрузка содержимого памяти и регистров DSP-процессора, передача управления загруженному приложению, управление ходом выполнения приложения по сигналам от специализированного программного обеспечения, запущенного на компьютере.

Основными недостатками такого подхода являются:

- анализ выполнения программы на "лабораторном" стенде со стандартным набором периферийных аппаратных средств, зачастую не отвечающим требованиям конкретного приложения;

- хотя программный код выполняется на процессоре в реальном времени,

контроль содержимого памяти и регистров не всегда может осуществляться в реальном времени (обычно для этого требуется остановить выполнение приложения; передать управление программе-монитору, которая и выгрузит требуемые данные в специальную программную среду, запущенную на ПК; вернуть управление приложению);

- возможность потери контроля над выполнением приложения на DSP-процессоре (например, если пользовательская программа глобально запрещает прерывания, то монитор не может вернуть себе управление).

Главное достоинство этих средств – низкая стоимость и доступность: в зависимости от комплектации цена на такие платы варьируется в среднем от 100 до 1000 долл. Поэтому они широко используются при разработке несложных приложений и в процессе обучения;

в) и, наконец, еще одним способом программной и аппаратной отладки разработанного проекта является использование интегрированных оценочных модулей (Integrated Evaluation Module), поставляемых разработчиками DSP-процессоров. Эти модули обычно являются платами, содержащими DSP-процессор, определенный набор аппаратных средств ввода/вывода, ОЗУ и допускающими подключение по PCI, EISA или USB-интерфейсу к персональному компьютеру. Комплект включает также программные средства разработки и отладки приложений. Разработка приложения в этом случае выполняется поэтапно: начиная с имеющейся аппаратной платформы, по мере необходимости добавляются новые периферийные устройства и соответствующий программный код[1]. В конечном итоге получается новая аппаратная платформа (и программное обеспечение), базовыми элементами которой являются предустановленные разработчиком интегрированного модуля схемы и устройства. Внесение изменений в аппаратную часть модуля облегчается тем, что большинство разработчиков DSP-процессоров в комплекте с платами поставляют CAD-файлы для их дальнейшего автоматизированного проектирования.

Контрольные вопросы

1. В чем преимущества использования схем эмуляции при разработке ПО?
2. Какие дополнительные возможности предоставляют программисту интегрированная среда разработки ПО?
3. В чем основные отличительные особенности разработки ПО для систем ЦОС?

[1] Большая часть оценочных плат также позволяет в определенной степени изменять конфигурацию системы, например, увеличивать объем внешней памяти.

2. ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ УСТРОЙСТВ ДЛЯ ОБРАБОТКИ ЦИФРОВЫХ СИГНАЛОВ В РЕАЛЬНОМ МАСШТАБЕ ВРЕМЕНИ

2.1. Особенности разработки и отладки ПО для систем реального времени

При традиционном программировании участки выполняемого кода организовываются в виде отдельных подпрограмм, вызываемых последовательно в соответствии с логикой программы (рис.6). Такой подход прекрасно подходит для большинства приложений, функционирующих на ПК, так как позволяет структурировать программу, сформировать отдельные, пригодные для повторного использования блоки программного кода. Однако он неприемлем при разработке приложений реального времени, поскольку не содержит средств синхронизации с событиями реального времени.

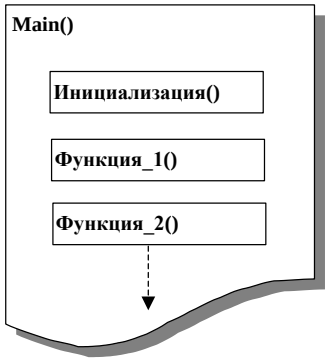


Рис.6. Принцип традиционного структурного подхода к разработке программ

Наиболее целесообразным для приложений реального времени считается такая структура программы, при которой фрагменты кода выполняются при вызове процедур обработки прерываний (Interrupt Service Routine, ISR). До разрешения прерываний система должна быть "инициализирована" – приведена в определенное начальное состояние (сконфигурированы управляющие биты и регистры, порты ввода/вывода, таймеры). После разрешения прерываний процессор обычно переводится в состояние простоя и ожидает, когда произойдет очередное разрешенное системное событие (прерывание). После обработки прерывания процессор возвращается в состояние простоя (рис.7).

Одной из проблем разработки ПО для систем реального времени является трудность отслеживания происходящих событий. Программа может быть разработана и отлажена с использованием симулятора (что необходимо для проверки работоспособности алгоритма), однако, в конечном счете, быстродействие написанного программного кода может оцениваться только при функционировании системы "на полной скорости", возможно, во взаимодействии с другими программными и/или аппаратными компонентами. Например, процедура фильтрации сигнала даже с высокой частотой дискретизации сама по себе не является требовательной к ресурсам ЦОС-системы, особенно при небольшом числе коэффициентов фильтра. Но для того чтобы она работала, необходим еще целый ряд подпрограмм, обеспечивающих загрузку ПО из микросхемы внешней памяти при включении, организацию ввода/вывода данных через порты ввода/вывода, реакцию на управляющие сигналы от внешних устройств и т.п. В конечном итоге отладка даже такой "элементарной" системы может быть сопряжена с проблемами. Так, например, некоторые процедуры могут кратковременно запрещать внешние прерывания на время выполнения определенных операций. Это приводит к недоступности процессора для отладочного программного обеспечения на этот период. Определенную трудность представляет также набор статистики и профилирование кода, поскольку обмен этими данными с хост-компьютером обычно выполняется со сравнительно невысокой скоростью и может ухудшать показатели быстродействия системы (а при статистическом профилировании – снизить его достоверность).

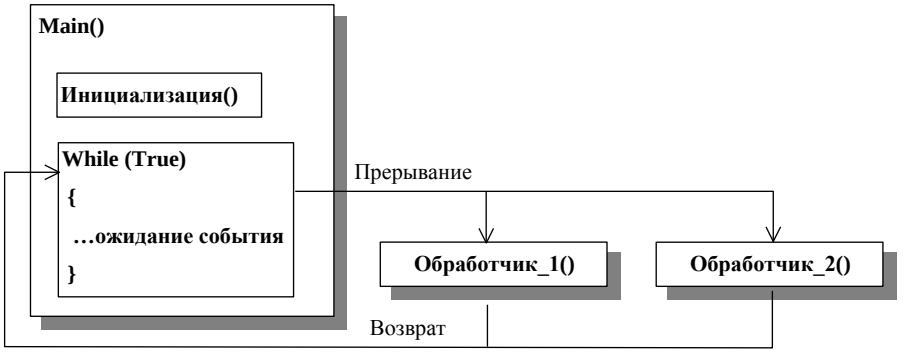


Рис.7. Принцип функционирования приложения реального времени

Повышение производительности процессоров, использование их для одновременного решения нескольких задач еще более усугубляет ситуацию. В конечном итоге разработка и отладка программного кода для системы, в которой множество различных событий могут происходить в произвольные моменты времени, требует (дополнительно к алгоритмам обработки сигналов) новых программ-сервисов, реализующих возможности планировщика задач, а также поддержки коммуникаций, управления ресурсами и анализа. Весь этот набор сервисов обеспечивается либо ОС реального времени, либо написанным самими разработчиками ядром. Операционные системы реального времени, как правило, дороги и требуют для своей работы большое количество ресурсов. Специально написанные ядра являют собой другую крайность – они малы по размерам, но их,

как правило, очень сложно конфигурировать, компоновать и переносить на другие платформы и приложения.

В качестве решения данной проблемы ведущие производители DSP и средств разработки для них предлагают унифицированные программные ядра реального времени Real-Time Operating Kernel (иногда сами разработчики их все же называют операционными системами реального времени Real-Time Operating System, RTOS), которые представляют собой прослойку между уровнем реальных прерываний и пользовательскими программами для их обработки и обычно предоставляют разработчику примерно следующий перечень сервисов (рис.8):

- гибкий планировщик задач с возможностью мультизадачной работы;
- уровень аппаратной абстракции для работы с периферийными устройствами процессора;
- независимый ввода/вывод для передачи потоков данных в реальном времени;
- средства анализа поведения ЦОС-приложения и обмена с ним данными в реальном времени.

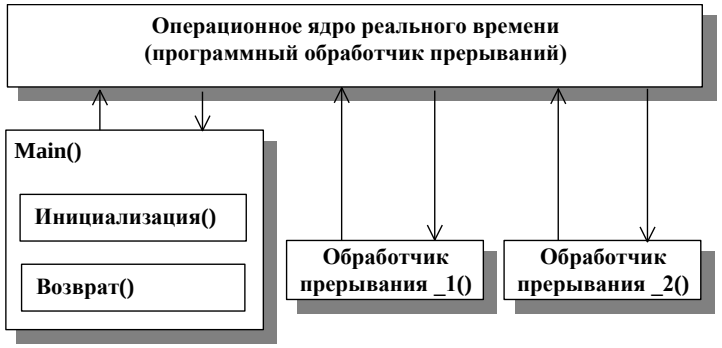


Рис.8. Принцип функционирования программного ядра реального времени

Практически это оптимизированная для ЦОС-приложений мультизадачная операционная система реального времени, которая входит в состав интегрированной среды разработки DSP/BIOS для Code Composer Studio (для TMS-процессоров), VDK для VisualDSP++ (для процессоров фирмы Analog Devices), OSE_{ck} или $RTXC^{TM}$ для CodeWarrior (для процессоров фирмы Motorola).

Рассматривая применение операционных систем в ЦОС-приложениях, необходимо помнить три очень важных момента. Первый – ограниченность ресурсов встроенной системы, которая заставляет очень жестко подходить к дополнительным расходам, например, памяти. Второй момент – необходимость минимизации временных накладных расходов, поскольку работа идет в реальном масштабе времени и желательно максимально использовать всю производительность DSP для решения основной задачи. И, наконец, чем сложнее операционная среда, тем сложнее предсказать её поведение, что никак не допустимо в алгоритмах ЦОС. Следовательно, надо иметь средства однозначного

задания поведения системы и средства контроля ее поведения.

Для реализации этих достаточно противоречивых требований при построении систем типа DSP/BIOS и VDK используются следующие положения: система является статически конфигурируемым ядром реального времени со статически определяемой приоритетной моделью исполнения процессов. Этим достигается, во-первых, минимизация дополнительных расходов памяти, поскольку в исполняемый код включаются только модули, необходимые при реализации данной задачи, а не вся операционная среда. Во-вторых, обеспечивается оптимизация производительности процессора, поскольку большинство статических вызовов после компиляции упаковываются в несколько команд. Далее, поскольку время выполнения команд DSP известно, конфигурация системы задается статически и тоже известна, модель исполнения и система приоритетов также задается статически, то имеем полностью предсказуемое и однозначно определенное поведение системы.

Одной из задач, решаемых DSP/BIOS и VDK, является предоставление разработчику уровня аппаратной абстракции, то есть единого логического интерфейса работы с аппаратной частью системы ЦОС, при этом учет особенностей работы аппаратных узлов того или иного DSP возлагается на инструментальные средства. Неотъемлемой частью RTOS обычно имеется стандартный интерфейс для организации каналов ввода/вывода, включающий в себя драйверы периферийных устройств, драйверы портов, к которым они подключаются, и средства организации стандартных потоков ввода/вывода. При этом вполне достижим уровень аппаратной абстракции, при котором получается полная межплатформенная переносимость приложения. Фактически сбывается очень давняя мечта разработчиков – если на конечной стадии разработки выяснится, что ресурсов выбранного ЦСП не хватает или, что ещё более болезненно, законченная и с таким трудом упакованная задача требует расширения, то можно просто перейти на более мощный ЦСП (при этом даже поменяв платформу), ничего не меняя в написанном, проверенном и отлаженном коде. Ещё одно преимущество переносимости – обратный процесс оптимизации. Можно взять мощный ЦСП, спокойно написать и отладить задачу, имея запас производительности для сервиса и отладки, для обкатки вариантов реализации и для оптимизации участков кода, а затем, отладив и оптимизировав программу, перенести её на оптимальный по параметрам и цене ЦСП для серийного выпуска устройства. С учётом наличия ЦСП различной мощности и объёма памяти в совместимых корпусах, возможности практически безболезненного переноса позволяют производить как функционально-стоимостную оптимизацию, так и модернизацию устройства без дорогостоящих затрат на модернизацию аппаратных составляющих устройства, например, на переразводку печатных плат.

Использование библиотек, предоставляющих оптимизированные для каждой платформы основные ЦОС-алгоритмы с единым интерфейсом, а также оптимизирующих компиляторов с языка С для написания алгоритмов, позволяет создать реально абстрагированное, "отвязанное" от конкретной платформы ЦОС-

приложение. Наличие операционной среды, кроме удобства разработки, даёт ещё один плюс – все компоненты работы с аппаратным обеспечением уже отлажены и работают – не надо тратить время на их написание и отладку.

Ещё одной важной особенностью ядра реального времени является наличие визуальных средств контроля и протоколирования порядка и последовательности исполнения нитей (процессов), а также средств отслеживания критических мест, что позволяет легко оценить ход исполнения приложения и быстро отследить и устранить критические участки.

2.2. Организация хранения фрагментов сигнала и доступа к данным. Кольцевые буферы

При написании большинства программ для приложений, не подпадающих под категории приложений реального времени (например, обычных баз данных, вычислительных программ Web-сайтов, программ для персонального компьютера), предполагается выполнение одной или нескольких предпосылок:

- процесс вычислений одноразовый;
- вся входная информация уже имеется и хранится, например, в виде файла;
- практически отсутствуют требования на "время отклика системы", т.е. время выполнения того или иного действия программой может меняться от раза к разу и зависит от таких внешних для нее факторов, как, например, количество выполняемых в системе задач, загруженность процессора, объем доступной оперативной памяти и частота обращения к странице виртуальной памяти, которой нет в данный момент в физическом ОЗУ и др.

В системе ЦОС обработка очередного отсчета (группы отсчетов) сигнала и переход к обработке следующего отсчета (группе отсчетов) должны происходить строго в соответствии с частотой дискретизации сигнала (периодом между поступлением отсчетов на входной порт процессора). Это значит, что несмотря на то, что какой-то отсчет или группа отсчетов в силу своих параметров и особенностей алгоритма может быть обработана быстрее (например, после проверки какого-либо условия выполнение алгоритма идет по "сокращенной" ветке вычислений), следующий отсчет должен быть "затребован" (прочитан из входного порта) не раньше ожидаемого времени его поступления. То же самое можно сказать и о выводе сформированных/обработанных значений в выходной порт. Таким образом, систему ЦОС можно охарактеризовать как систему, выполняющую однотипные вычислительные операции над данными в "потоковом режиме", причем обработка каждого отсчета или группы отсчетов выполняется

практически за один и то же интервал времени[1].

В большинстве приложений ЦОС обрабатываемый сигнал рассматривается как бесконечно продолжающийся во времени. Поэтому хранение такого сигнала в вычислительном устройстве, имеющем ограниченный размер памяти в принципе невозможно. К счастью, практически всегда для реализации алгоритма обработки сигнала достаточно иметь какое-то определенное количество последних входных отсчетов (например, нерекурсивный фильтр), а иногда и выходных отсчетов (рекурсивный фильтр).

Как нельзя лучше для хранения последних отсчетов сигнала подходит такая структура данных, как кольцевой буфер, поддерживаемый аппаратными возможностями всех современных DSP-процессоров. Он позволяет эффективно организовать (рис.9):

- обновление буфера, при этом последний записываемый в буфер отсчет X(i) "затирает" самый "старый" отсчет X(i-N) (для буфера длиной *N*) и в буфере остаются *N* последних отсчетов;
- последовательное считывание последних *N* отсчетов в случае, если указатель на начало буфера (для чтения) Index_Read имеет значение на 1 больше, чем указатель на последний помещенный в буфер элемент Index_New. За счет кругового перемещения указателя Index_Read в цикле чтения буфера первым прочитанным элементом будет самый "старый" из последних *N* отсчетов, а последним - только поступивший отсчет. Такой подход используется, если чтение элементов буфера необходимо выполнить в порядке их поступления, т.е. X(i-N+1), X(i-N+2),...,X(i). Реализовать вариант, когда чтение должно выполняться в порядке, обратном поступлению отсчетов, т.е. от последнего поступившего элемента к самому старому также не составляет труда.

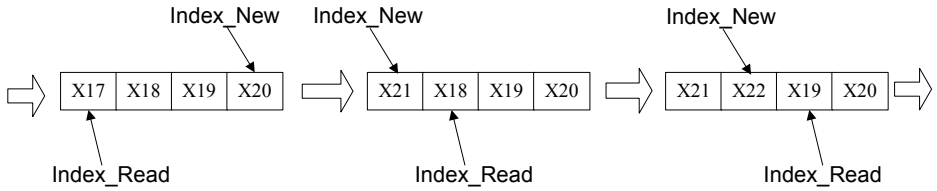


Рис.9. Пример взаимного положения указателей Index_New и Index_Read

Как было отмечено выше, время обработки одного отсчета сигнала реализуемым в системе ЦОС алгоритмом должно находиться в строгом соответствии с частотой поступления отсчетов сигнала. Рассмотрим два варианта

[1] Естественно, что это справедливо только в том случае, если система имеет дело с цифровыми сигналами, полученными из реальных аналоговых сигналов с использованием АЦП. Если же речь идет, например, об обмене промежуточными результатами вычислений, управляющей или служебной информацией между процессорами (т.е. приеме/передаче информации, для которой постоянство временного интервала между последовательными значениями не имеет значения), то тогда могут быть применимы способы обмена данными, используемые в традиционном программировании.

организации ввода/вывода данных, отвечающих этому требованию. Обратите внимание на количество буферов, которые необходимо поддерживать одновременно для правильного и эффективного функционирования алгоритмов.

Пример 1. Организация буферов при поэлементном формировании результатов обработки. В первом примере рассмотрим простейший нерекурсивный фильтр, реализуемый как сумма поэлементных произведений последних N отсчетов входного сигнала на N коэффициентов фильтра. Результатом выполнения операции является отсчет выходного (отфильтрованного) сигнала. Каждый раз при поступлении нового отсчета сигнала можно выполнять обработку N последних отсчетов и формировать очередной отсчет выходного сигнала. Таким образом, алгоритм вычисления формулы фильтра должен укладываться в интервал между поступлениями входных отсчетов (мы рассматриваем простейший вариант, когда интервалы между отсчетами (частоты дискретизации) входного и выходного сигнала совпадают).

Этапы выполнения итерационной процедуры обработки данных можно записать следующим образом (рис.10):

1. По таймеру или сигналу от внешнего устройства вызывается обработчик прерывания, который выводит в выходной порт сформированный на предыдущей итерации выходной отсчет и переносит из входного порта в приемный буфер новый отсчет входного сигнала.

2. Обработчик вызывает процедуру вычисления, в которой выполняется обход N элементов буфера входного сигнала (и соответствующий обход буфера с коэффициентами фильтра) и формируется очередной отсчет выходной сигнала.

3. Работа обработчика завершается, процессор переходит в цикл ожидания следующего прерывания.

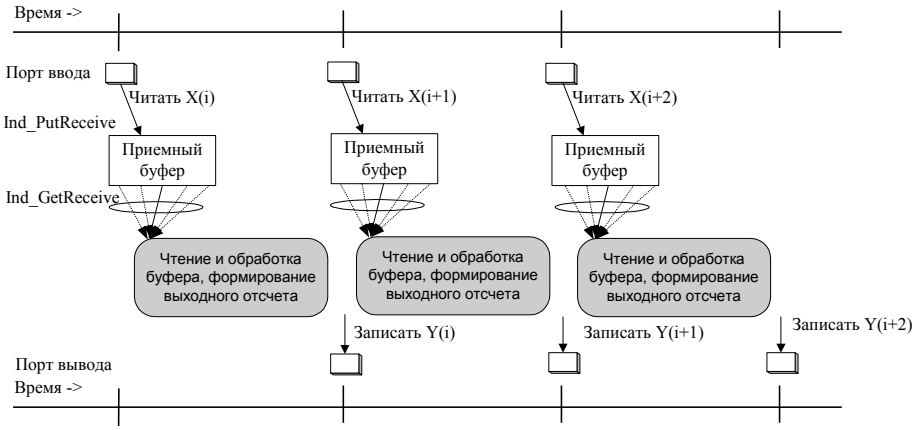
Следует обратить внимание на то, что вывод сформированных отсчетов всегда отстает от ввода отсчетов исходного сигнала. Обозначения типа X(i) и Y(i) на рисунке используются для того, чтобы проиллюстрировать соответствие между номерами входного отсчета и получаемого с его использованием выходного отсчета.

При программной реализации алгоритма одним из ключевых моментов является верхняя оценка частоты дискретизации входного сигнала, для которой выполняется корректная обработка отсчетов на данном процессоре. Сравнение этой граничной величины с действительной частотой дискретизации входного сигнала позволяет сделать вывод о возможности обработки исходного сигнала с использованием разработанного аппаратно-программного обеспечения.

Рассмотрим пример вычисления оценки предельной частоты дискретизации входного сигнала для приведенной в данном разделе схемы поэлементной обработки.

Для реализации системы ЦОС выбран процессор с тактовой частотой 100МГц (длительность одного процессорного такта – 10нс). Пусть длительность выполнения собственно программной реализации алгоритма обработки (включая

переход на обработчик прерывания, выполнения вычислений и возврат из обработчика с учетом некоторых потерь на переходы и т.п.) составляет, например 400 тактов. Тогда интервал между поступлениями последовательных отсчетов входного сигнала не может быть меньше 400 тактов = 4 мкс. Находя обратную величину от этого интервала, получаем предельную скорость поступления входных отсчетов – 250000 отсчетов/с = 250 КГц – это и есть предельно допустимая частота дискретизации входного сигнала.



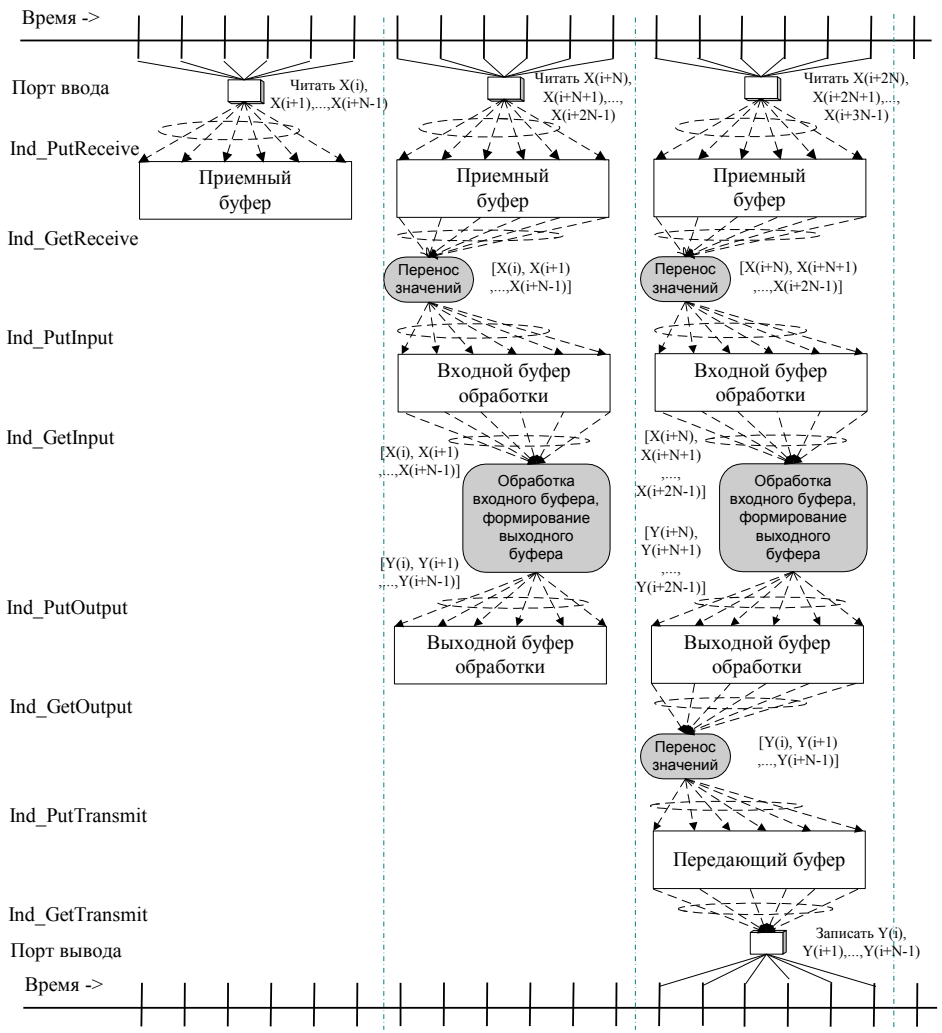
Пример 2. Организация буферов при поблочной обработке входных данных. Во втором примере рассмотрим организацию буферов ввода/вывода для реализации алгоритма, который последовательно обрабатывает непересекающиеся фрагменты из N последовательных отсчетов сигнала, например, при выполнении преобразования Фурье.

Этапы выполнения итерационной процедуры обработки данных можно записать следующим образом (рис. 11):

1. По таймеру или сигналу от внешнего устройства вызывается обработчик прерывания, который выводит в выходной порт очередной выходной отсчет из передающего буфера, а также переносит из входного порта в приемный буфер новый отсчет входного сигнала. Если приемный буфер полон (произошел круговой "заворот" указателя), генерируется специальное прерывание переполнения буфера. Работа обработчика прерывания таймера завершается.

2. Если произошло прерывание по переполнению приемного буфера, вызывается соответствующий обработчик, который переносит данные их приемного буфера во входной буфер и из выходного буфера в передающий буфер.

3. Выполняется обработка данных. Чтение данных в процедуре обработки осуществляется из входного буфера, а запись результатов – в выходной буфер.



4. По завершении обработки фрагмента работа обработчика прерывания завершается, процессор переходит в цикл ожидания следующего прерывания.

Для обеспечения работоспособности схемы приоритет обработчика прерывания от таймера должен быть выше приоритета обработчика прерывания по переполнению кругового буфера, поскольку во время обработки фрагмента сигнала (и даже, возможно, во время переноса данных из буфера ввода во внутренний приемный буфер) прием/выдача очередных отсчетов должны

продолжаться. Это означает, что время работы процедуры обработки следует увеличить на время выполнения обработчика прерывания от таймера, умноженного на количество отсчетов в одном фрагменте данных.

Сигнал генерации прерывания по переполнению кругового буфера обычно можно "повесить" на один из указателей, например, на указатель записи данных во входной буфер. Также можно отметить, что копирование между буферами не обязательно должно завершиться до поступления первого отсчета следующего фрагмента. Главное – чтобы приемный буфер освобождался быстрее, чем будет заполняться новым фрагментом.

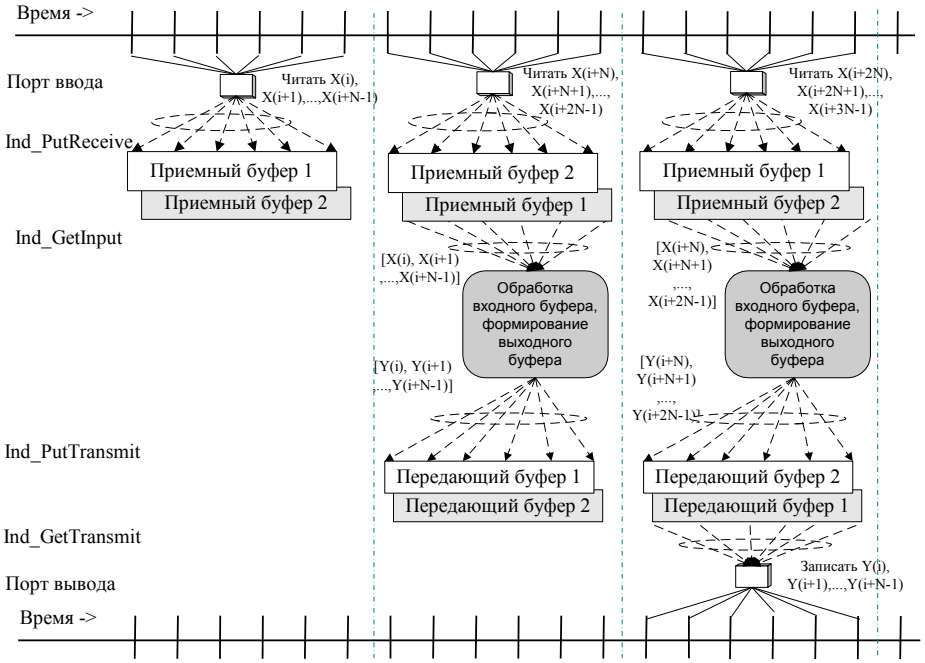
Следует обратить внимание на то, что вывод сформированных отсчетов "отстает" от ввода отсчетов исходного сигнала на время, равное длительности 2-х фрагментов сигнала, т.е. задержка будет равна длительности 2*N* отсчетов.

Рассмотрим пример вычисления оценки предельной частоты дискретизации входного сигнала для приведенной в данном разделе схемы поблочной обработки.

Для реализации системы ЦОС выбран процессор с тактовой частотой 100МГц (длительность одного процессорного такта – 10нс. Размер буфера – 35 отсчетов сигнала). Пусть длительность обработчика прерывания для помещения нового отсчета в приемный буфер и вывода из передающего буфера составляет 6 тактов. Пусть также длительность выполнения программной реализации алгоритма обработки целого буфера (включая переход на обработчик прерывания по заполненности приемного буфера, переноса данных между буферами, выполнения собственно вычислений и возврат из обработчика с учетом некоторых потерь на переходы и т.п.) составляет, например 16000 тактов. Однако, поскольку процедура обработки должна прерываться столько раз, сколько отсчетов в буфере (для накопления следующего фрагмента сигнала и вывода предыдущего), то ее длительность необходимо увеличить на 35*6 – именно столько тактов будет занимать "чистый" прием фрагмента сигнала в буфер. Таким образом, полное время обработки буфера составляет 16000+35*6=16210 тактов. Разделив это время на количество отсчетов, которые надо принять для формирования следующего фрагмента сигнала, получаем 16210/35≈464 такта, или 4,64 мкс. Находя обратную величину, как и в предыдущем примере, получаем предельную частоту дискретизации входного сигнала 215517 Гц ≈ 215,5 КГц.

Упростить рассмотренную процедуру в большинстве случаев можно путем использования обмена указателей на приемный и входной буферы, как показано на рис.12. Повышение производительности соответствует разнице в трудоемкости переноса данных из одного буфера в другой и перестановкой указателей (особенно актуально для буферов больших размеров).

Справедливости ради следует заметить, что большинство современных процессоров обладают возможностями обмена блоками данных между портами ввода/вывода и внутренней памятью процессора посредством DMA-контроллера, т.е. без "отвлечения" процессора от выполнения "основных" вычислений.



Контрольные вопросы

1. Приведите вариант организации буферов для реализации рекурсивного фильтра.
2. Как можно еще более повысить эффективность схемы организации буферов за счет использования теневых (альтернативных) регистров указателей?
3. Какие конкретные упрощения в организации ввода/вывода данных возможны при использовании DMA-пересылок при записи/чтении значений из внутренней памяти?
4. Вспомните механизм аппаратной поддержки прерываний в современных процессорах. Какие действия выполняет процессор при вызове прерывания и при возврате из него?

3. ФОРМАТЫ ПРЕДСТАВЛЕНИЯ ДАННЫХ В ЦИФРОВЫХ СИГНАЛЬНЫХ ПРОЦЕССОРАХ

В зависимости от конкретного приложения разработчик цифровой системы

должен определить требуемый уровень точности вычислений и динамический диапазон системы. Для этого необходимо выбрать соответствующий формат данных для обработки в DSP-процессорах. Обычно 16- и 24-разрядные DSP-процессоры используются для обработки данных в формате с фиксированной точкой (фиксированной запятой, ФЗ), а 32-разрядные DSP-процессоры – для обработки данных в формате с плавающей точкой (плавающей запятой, ПЗ).

3.1. Представление данных с фиксированной запятой

Представление данных в ФЗ-формате означает, что для всех чисел заданного формата логически фиксируется одинаковое местоположение точки, разделяющей целую и дробную части числа. Разряды слева от точки представляют целую часть числа и его знак (если число знаковое), а справа – дробную часть числа. Общим для целых и дробных ФЗ-чисел является возможность представления как знаковых, так и беззнаковых чисел.

Арифметические операции над ФЗ-числами выполняются практически одинаково независимо от фактического содержимого регистров. Поэтому задача правильной трактовки результатов целиком ложится на программиста.

Дробные числа с фиксированной точкой. Наиболее естественным с точки зрения соответствия реальным сигналам является использование в DSP-процессоре знаковых дробных (fractional) ФЗ-чисел, поскольку дробное представление четко показывает, какое отношение амплитуда отсчета, полученного с АЦП, имеет к полному размаху сигнала (обычно +/- 5В). Операции с дробными ФЗ-числами более устойчивы к переполнению, поскольку перемножение дробных чисел дает в результате значение меньшее (по модулю), чем каждый из операндов.

Операции, выполняемые DSP-процессором над ФЗ-числами, приводят к получению результатов, которые могут потребовать для своего хранения больше битов, чем отводится разрядной сеткой процессора. Для решения этой проблемы для дробных ФЗ-чисел применяют операции усечения и округления результатов до заданной разрядности. Это, а также возможные переполнения в процессе реализации алгоритмов обработки сигналов, являются одними из источников ошибок в системах ЦОС, приводящих к снижению динамического диапазона сигнала.

В дробном ФЗ-формате предполагается, что логическая (двоичная) точка расположена либо справа от самого старшего бита (для знаковых чисел), либо слева от самого старшего бита (для беззнаковых чисел).

Бит	15	14	13		2	1	0
Вес	-2^0	2^{-1}	2^{-2}	…	2^{-13}	2^{-14}	2^{-15}

Знаковый бит

Знаковое дробное

Бит	15	14	13		2	1	0

Вес	2^{-1}	2^{-2}	2^{-3}	…	2^{-14}	2^{-15}	2^{-16}

Беззнаковое дробное

Арифметические инструкции большинства DSP-процессоров оптимизированы для операций со знаковыми дробными числами в формате *Qn*, где (*n*+1) – разрядность слова процессора. Например, запись формата Q7 обозначает, что в числе содержится 1 знаковый бит (самый левый, старший бит) и 7 значащих бит. Предполагается, что фиксированная точка расположена между знаковым и старшим значащим битами.

Целые числа с фиксированной точкой. В целочисленном ФЗ-формате предполагается, что логическая (двоичная) точка расположена справа от самого младшего бита.

Целые ФЗ-числа могут быть знаковыми и беззнаковыми, причем знаковые числа представляются в дополнительном коде. Беззнаковые числа могут принимать только положительные значения.

Бит	15	14	13		2	1	0
Вес	-2^{15}	2^{14}	2^{13}	…	2^{2}	2^{1}	2^{0}
	Знаковый бит						

Знаковое целое

Бит	15	14	13		2	1	0
Вес	2^{15}	2^{14}	2^{13}	…	2^{2}	2^{1}	2^{0}

Беззнаковое целое

Целочисленный формат иногда обозначается в виде *b*.0, т.е. *b* разрядов отводятся на представление целой части числа и его знака.

Если процессор поддерживает обработку только целочисленных или только дробных ФЗ-данных, что перевод между двумя типами форматов можно осуществить, исходя из соответствия значения 1 в дробном формате целому числу 2^n (и то и другое число не может быть представлено в ФЗ-формате разрядности *n*). Например, для знаковых и беззнаковых 16-разрядных чисел можно воспользоваться следующими соотношениями (a-дробное число, A-целое число).

Беззнаковые: $1 - 65536$, $a - A$ Знаковые: $1 - 32768$, $a - A$

3.2. Формат чисел с плавающей запятой

Число с плавающей запятой представляется в виде

$$x = f * 2^e,$$

где f – мантисса, а e – порядок (экспонента). Обычно мантисса представляется в формате с фиксированной запятой, а порядок является целым числом.

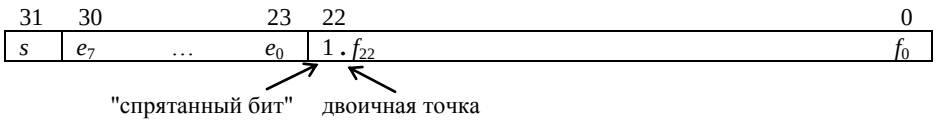
Поскольку в экспоненциальной форме записи числа присутствуют два параметра, такое представление оказывается неоднозначным, например:

$$2 = 2 * 2^0 = 1* 2^1 = ½ * 2^2 = \dots$$

Чтобы устранить эту неоднозначность, принято ограничивать диапазон допустимых значений мантиссы $1 \le f \le 2$. Процедура приведения мантиссы к допустимому диапазону называется нормализацией, а экспоненциальная запись числа, удовлетворяющая указанным ограничениям, – нормализованной экспоненциальной формой.

Нормализация мантиссы позволяет сэкономить один разряд в ее двоичном представлении. Действительно, если для нормализации используется диапазон [1,2[, двоичная запись мантиссы всегда будет выглядеть, как 1,… То есть заранее известно, что первый значащий разряд равен единице, поэтому его можно не хранить. Это называется использованием неявного (спрятанного) старшего бита.

Согласно стандарту IEEE-754/854, 32-разрядное число с плавающей точкой представляется в следующем формате:



Число одинарной точности в формате с плавающей запятой состоит из знакового бита *s*, 24-битового поля мантиссы (f_{22}-f_0) и 8-битовой беззнаковой смещенной[1] двоичной экспоненты (e_7-e_0). Предполагается, что мантисса состоит из 23-битовой части и "спрятанного" бита, предшествующего биту f_{22} и всегда равного единице. Значение беззнаковой двоичной экспоненты e лежит в диапазоне от 1 до 254 и получается путем прибавления к несмещенной двоичной экспоненте числа +127 (при вычислении несмещенной двоичной экспоненты, естественно, необходимо вычесть это же число из смещенной экспоненты).

32-разрядное число с плавающей точкой можно представить в виде

$$n = m * 2^{e-128},$$

а число в формате IEEE-754/854 можно записать в виде

$$n = (-1)^s * 2^{e-128}(1.b_0b_1b_2\dots b_{23}).$$

Для повышения точности представления мантиссы числа в DSP фирмы Analog Devices реализована поддержка 40-разрядных чисел с плавающей точкой расширенной точности, в которых для хранения мантиссы используется не 23, а 31 бит.

Использование строго нормализованной мантиссы делает невозможным

[1] Смещенная экспонента получается в результате вычитания из двоичной экспоненты значения 127.

представление значения "0" в ПЗ-формате. Поэтому используется специальное соглашение о том, что число, содержащее нули во всех разрядах мантиссы и порядка, считается нулем. Стандарт IEEE определяет еще некоторые "особые" типы данных, используемые для сигнализации о переполнении, потере значимости и т.п. (см. табл.1).

Таблица 1

Тип	Экспонента	Мантисса	Значение	Примечание
NAN	255	$\neq 0$	"неопределенность"	недопустимый ПЗ-формат
Infinity	255	= 0	$(-1)^s * \infty$	бесконечность
Normal	$1 \le e \le 254$	любая	$(-1)^s * (1.f_{22\text{-}0})2^{e-127}$	нормальное число
Zero	0	0	$(-1)^s * 0$	нуль

Чтобы использование соглашения о представлении нуля не привело к возникновению "дырок" в наборе представимых чисел, нулевое представление порядка должно соответствовать не нулевому, а минимально возможному (т.е. отрицательному) его значению. Поэтому для представления порядка ПЗ-числа используется смещенная экспонента.

Итак, для представления нулевого значения в ПЗ-формате мантисса должна быть денормализованной. Однако, если использовать денормализованную мантиссу только для представления нуля, малые по модулю числа, представимые в данном формате, оказываются расположенными неравномерно – вокруг нуля появляется "мертвая" зона, размер которой существенно превышает расстояние между ближайшими к нулю представимыми числами. Чтобы решить эту проблему и сделать расположение представимых чисел вблизи нуля равномерным, применяют следующее соглашение о денормализации: если все разряды порядка имеют нулевое значение, то величина порядка увеличивается на единицу, а мантисса считается денормализованной, то есть содержащей в старшем спрятанном разряде не единицу, а нуль. Это позволяет расширить диапазон представления малых по модулю чисел. Фактически в данном случае диапазон возможных отрицательных порядков увеличивается за счет сокращения числа значащих цифр мантиссы.

3.3. Переполнение разрядной сетки и округление промежуточных результатов вычислений

В процессе обработки любого сигнала производится множество операций сложения и умножения. При этом промежуточные результаты вычислений могут существенно (иногда на несколько порядков) превосходить исходные и окончательные значения и вызывать переполнение разрядной сетки вычислительного устройства. Одним из основных источников возникновения ошибок при выполнении вычислений является возможная необходимость

округления/усечения промежуточных результатов вычислений[1]. Для иллюстрации достаточно вспомнить, что, например, при выполнении умножения двух дробных 32-разрядных ФЗ-чисел результат может иметь до 64 значащих цифр, тогда как для сохранения его в регистрах процессора или памяти он должен быть округлен до 32 бит. Аналогичные проблемы, хотя и в меньшей степени, могут проявляться и при операциях с ПЗ-числами.

При обучении программированию обычно даются рекомендации, как организовывать вычисления таким образом, чтобы уменьшить накопление ошибок вследствие операций округления, например, складывать числа, начиная с наименьших (по модулю) и заканчивая самыми большими (см., например, [7]). Однако при реализации алгоритмов ЦОС обычно отсутствует возможность менять последовательность выполнения операций, так как выполнение алгоритма ведется в реальном масштабе времени, а изменение порядка выполнения арифметических инструкций потребует существенных дополнительных издержек на сортировку данных.

Уменьшить вероятность переполнений можно, используя для хранения чисел формат с плавающей запятой, обладающий большим динамическим диапазоном. В то же время следует помнить, что разрядность дробного ФЗ-числа составляет 32 бита, а мантиссы ПЗ-числа – только 23. Это значит, что точность представления ПЗ-данных ниже, чем ФЗ-данных и они должны использоваться только если диапазон чисел, охватывающий возможные значения исходных, промежуточных и результирующих данных велик.

Современные DSP-процессоры содержат возможности по отслеживанию операций переполнения или потери значимости (например, генерация аппаратных прерываний), позволяющих во время выполнения программы принимать меры по уменьшению влияния таких ситуаций на результаты обработки. В любом случае разработанная система ЦОС должна тщательно тестироваться для обнаружения потенциальных проблем, связанных с конечной точностью вычислений.

3.4. Динамический диапазон чисел в разных форматах и диапазон значений

Диапазон представления чисел устанавливает границы между минимально и максимально допустимыми значениями, представляемыми в заданном формате и коде.

Динамический диапазон (ДД) определяется, как

$$ДД\ (дБ) = 20 \lg \frac{|макс.значение|}{|мин.значение \neq 0|}.$$

Максимально допустимая ошибка при представлении дробных чисел равна:

[1] Более подробное рассмотрение причин возникновения и особенностей распространения ошибки в результате выполнения арифметических операций приведено в работе [6].

- при округлении $2^{-(b+1)}=2^{-m}$ – половине младшего значащего бита (половине шага квантования);

- при усечении 2^{-b} – младшему значащему биту (шагу квантования).

Точность представления чисел определяет максимально допустимую точность представления дробной части вещественных чисел (в двоичном формате дробные числа представляются приближенно):

$$Точность = \log_2\left(\frac{|макс.значение|}{|макс.ошибка\ при\ округлении|}\right),$$

где *макс.значение* соответствует:

- для дробных чисел – максимальному (по модулю) значению, представленному в заданном формате и коде;

- для смешанных чисел (содержащих и целую и дробную части, например, регистров-аккумуляторов операций умножения с накоплением) – максимальному (по модулю) значению дробной части числа, представленной в заданном формате и коде.

Наличие расширений во внутренних регистрах вычислительных блоков большинства современных DSP-процессоров позволяет хранить при внутренних вычислениях смешанные числа. Диапазон их представления в двоичном коде определяется, как

$$-2^{EXT} \le C \le 2^{EXT} - 2^{-(k-EXT-1)},$$

где C – значение смешанного числа; k – длина расширенного слова, равная длине смешанного числа со знаком; EXT – длина расширения, равная длине целой части числа; (k-EXT-1) – длина дробной части числа.

Динамический диапазон равен

$$ДД(дБ) = 20\lg\left(\frac{|-2^{EXT}|}{|2^{-(k-EXT-1)}|}\right) = 20\lg\left(2^{k-1}\right).$$

Точность представления дробной части смешанного числа в соответствии с определением

$$Точность = \log_2\left(\frac{|-1|}{|2^{-(k-EXT)}|}\right) = (k - EXT)\ \text{бит},$$

где $2^{-(k\text{-EXT})}=2^{-(k\text{-EXT-1})-1}$ – максимально допустимая ошибка при округлении дробной части числа, равная половине младшего значащего бита.

Примеры динамических диапазонов знаковых смешанных слов в зависимости от длины смешанного слова и количества разрядов, приходящихся на целую и дробную части приведены в табл.2.

Таблица 2

Разрядность k (битов)	Длина расширения EXT (битов)	Точность представления дробной части (k-EXT)	Диапазон представления	ДД (дБ)
40	8	32	$-2^8 \le C \le 2^8 - 2^{-31}$ -256…256-(4,6e-10)	2^{39} 234,8
56	8	48	$-2^8 \le C \le 2^8 - 2^{-47}$ -256…256-(7,1e-15)	2^{55} 331,1
80	16	64	$-2^{16} \le C \le 2^{16} - 2^{-63}$ -65536…65536-(1,1e-19)	2^{79} 475,6

Каждый дополнительный бит, используемый в DSP для представления или обработки чисел, повышает отношение сигнал/шум примерно на 6 дБ. Диапазоны значений и динамические диапазоны при представлении чисел в различных форматах приведены в табл.3.

Иногда для повышения динамического диапазона обрабатываемых значений используется моделирование ПЗ-арифметики на ФЗ-процессорах. При таком моделировании для хранения ПЗ-числа требуются две ячейки: одна для мантиссы, другая – для экспоненты.

В архитектуре ФЗ-процессоров предусмотрены специальные операции, позволяющие моделировать ПЗ-представление чисел:

- нормализация числа (осуществление сдвига числа влево до тех пор, пока значения знакового и старшего значащего битов перестанут совпадать, с сохранением в специальном регистре числа сдвигов – двоичного порядка числа);

- выделение порядка для блока чисел, когда внутри блока выделяется самое большое по модулю число, оно нормализуется, в специальном регистре сохраняется его порядок, а все остальные числа блока представляются в формате с тем же порядком, что у наибольшего числа.

Блочная ПЗ-арифметика эффективна для представления малых (по модулю) чисел с небольшим разбросом значений. Моделировать такие вычисления целесообразно, только если их объем невелик, в противном случае имеет смысл использовать ПЗ-процессор.

Таблица 3

Формат	Разрядность	Значащих битов	Вес младшего разряда	Диапазон значений (представления)	ДД, дБ
Целочисленный беззнаковый	16	16	1	$0 \le C \le (2^{16}-1)$ 0…65535	96,3
	24	24	1	$0 \le C \le (2^{24}-1)$ 0…16777215	144,5
	32	32	1	$0 \le C \le (2^{32}-1)$ 0…4294967295	192,7
Целочисленный знаковый	16	15	1	$-2^{15} \le C \le (2^{15}-1)$ -32768…32767	90,3
	24	23	1	$-2^{23} \le C \le (2^{23}-1)$ -8388608…8388607	138,5
	32	31	1	$-2^{31} \le C \le (2^{31}-1)$ -2147483648…2147483647	186,6
Дробный беззнаковый	16	16	2^{-16} 0,000015	$0 \le C \le (1-2^{-16})$ 0,0…0,999985	96,3
	24	24	2^{-24} 0,0000000596	$0 \le C \le (1-2^{-24})$ 0,0…0,9999999404	144,5
	32	32	2^{-32} 0,00000000023	$0 \le C \le (1-2^{-32})$ 0,0…0,99999999977	192,7
Дробный знаковый	16	15	2^{-15} 0,000031	$-1 \le C \le (1-2^{-15})$ -1,0…0,999969	90,3
	24	23	2^{-23} 0,00000012	$-1 \le C \le (1-2^{-23})$ -1,0…0,99999988	138,5
	32	31	2^{-31} 0,00000000047	$-1 \le C \le (1-2^{-31})$ -1,0…0,99999999953	186,6
С плавающей точкой	32	смещ. порядок-8 мантисса-24, дробн.часть-23, неявная-1 $-126 \le e \le 127$	min (при e = -126, m=1,00…***0***) – $1{,}4 * 10^{-45}$ max (при e = 127, m=1,00…***0***) – $2 * 10^{31}$	(+/-) $1 * 2^{-126} \le C \le (2-2^{-23}) * 2^{127}$ C>0: $-3{,}4 * 10^{38}$… $-1{,}18 * 10^{-38}$ C<0: $1{,}18 * 10^{38}$…$3{,}4 * 10^{38}$	1530

3.5. Оценка необходимой разрядности процессора для приложений ЦОС

В системе ЦОС, основанной на использовании DSP-процессора, каждое из звеньев в цепи прохождения и обработки сигнала оказывает влияние на точность его представления и обработки и определяет общие качественные характеристики системы в целом (на примере обработки аудиосигналов):

- характеристики устройства ввода "реального" аналогового сигнала (обычно микрофона или аналогового входа);
- размер слова и ошибка аналого-цифрового преобразования АЦП;
- эффекты "конечной длины слова" в DSP-процессоре, возникающие в результате округления и усечения результатов вычислений;
- размер слова ЦАП;
- характеристики устройств воспроизведения (например, акустических

систем);

- характеристики других аналоговых или цифровых устройств, участвующих в процессе обработки сигнала.

Общая оценка качественных характеристик системы (динамический диапазон) определяется самым "слабым" с точки зрения точности представления и обработки данных звеном цепи. Например, если в сочетании с 16-разрядным АЦП, обеспечивающим качество преобразования (динамический диапазон) примерно 96 дБ, мы будем использовать 24-разрядный DSP-процессор, способный обеспечить динамический диапазон до 144 дБ, то "полный" динамический диапазон системы составит только 96 дБ.

Предположим, что с выхода АЦП поступает 16-разрядный цифровой сигнал с динамическим диапазоном 96 дБ. При использовании для его обработки 16-разрядного ЦСП и наличия в алгоритме обработки нескольких операций умножения, умножения с накоплением и т.п. в результате накопления ошибок округления и усечения ошибка распространится на младшие биты значений сигнала. Происходит сокращение динамического диапазона сигнала и, следовательно, ухудшается качество сигнала. Распространение ошибки на каждый следующий разряд приводит к снижению *ОСШК* на 6 дБ (рис.13). При использовании 16-разрядного DSP-процессора искажения, внесенные в процессе цифровой обработки, попадут в выходной сигнал и приведут к снижению динамического диапазона до 96 – 18 = 78 дБ. При использовании 24-разрядного DSP-процессора ошибки квантования и усечения не превысят пороговый уровень шума для АЦП и не приведут к дальнейшему ухудшению качественных характеристик сигнала.

Одним из основных правил при использовании систем ЦОС является выбор DSP-процессора и реализация алгоритмов обработки, обеспечивающих "непревышение" накопленных в результате промежуточных вычислений ошибок выше порога шума АЦП. С этой целью разрядность ЦСП обычно выбирается "с запасом" в зависимости от сложности алгоритмов обработки.

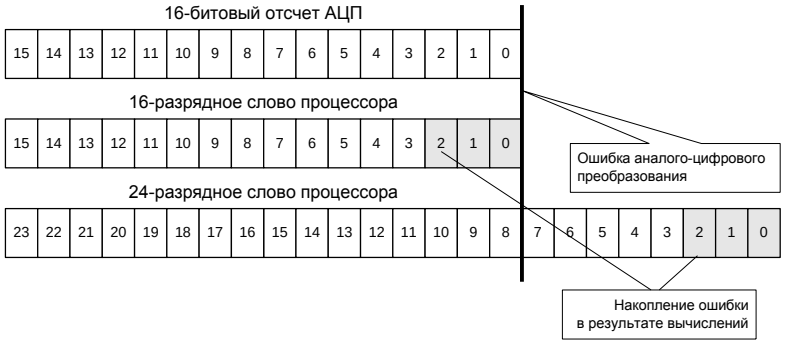


Рис. 13. Накопление ошибки при обработке данных

В табл. 4 для справки приведены типовые требования к обеспечению динамического диапазона в системах обработки аудиосигналов для различных областей применения, которые необходимо учитывать при выборе DSP-процессора и реализации алгоритмов обработки сигнала.

Таблица 4

Устройство/приложение обработки звука	Динамический диапазон, дБ
AM-радио	48
Аналоговое вещательное телевидение	60
FM-радио	70
Аналоговый видеоплейер	73
Видеокамера	75
16-битовый аудиоконвертер	от 90 до 95
Цифровое вещательное телевидение	85
Проигрыватель мини-дисков	90
Проигрыватель компакт-дисков	от 92 до 96
DAT-устройство	110
Аналоговый микрофон	120

Контрольные вопросы

1. Почему использование нормализованной мантиссы приводит к снижению точности представления чисел вблизи нуля?

2. Сколько операций умножения/накопления необходимо выполнить над 16-разрядными дробными числами (с усечением или округлением результата каждой операции), чтобы динамический диапазон поступившего на вход процессора сигнала уменьшился на 20 дБ?

3. Что дает расширение разрядности внутренних регистров умножителя для уменьшения накопления ошибки при выполнении операций умножения с накоплением?

4. ИНТЕГРИРОВАННАЯ СРЕДА VISUALDSP++

4.1. Состав и основные функциональные возможности VisualDSP++

Интегрированная среда VisualDSP++ представляет собой набор средств для разработки и отладки программного обеспечения для систем цифровой обработки сигналов. Она включает:

- интегрированную оболочку для разработки и отладки приложений (Integrated Development and Debugging Environment, IDDE);
- поддержку ядра VisualDSP++ Kernel (VDK), которое представляет набор базовых блоков для построения приложения. При этом обеспечивается автоматическая генерация шаблонного программного кода на языках C/C++ или ассемблер, поддержка многозадачности, модульность, переносимость, масштабируемость;
- оптимизирующий компилятор языков C и C++ с библиотеками;
- средства генерации исполняемого кода и карты памяти системы для загрузки при включении питания (и запуска во внутренней памяти процессора(ов) или выполнения из внешней памяти системы;
- программное обеспечение для программной и аппаратной эмуляции;
- возможность разработки ПО для многопроцессорных систем.

Интерфейс среды VisualDSP++ приведен на рис.13.

VisualDSP++ предоставляет пользователю широкий набор средств и возможностей для эффективной и быстрой разработки ПО.

1. Набор средств генерации кода включает компилятор языков C и C++ с возможностью оптимизации кода, препроцессор, ассемблер, компоновщик, библиотекарь, загрузчик и сплиттер. VisualDSP++ позволяет использовать средства генерации кода третьих фирм, ориентированные на конкретные аппаратные средства.

2. VisualDSP++ предоставляет возможности по работе с проектом, позволяющие эффективно изменять состав файлов проекта, модифицировать его опции и установки.

3. Работу в режиме редактирования и отладки облегчает использование пиктограмм статуса, которые отмечают точки останова, закладки, этапы выполнения инструкций в конвейере программного секвенсора. Содержимое регистров, ячеек внутренней и внешней памяти может отображаться в различных числовых форматах.

3. При отладке программ в среде VisualDSP++ могут быть использованы точки останова (breakpoints) и останова по событию (watchpoints), Tcl-сценарии для автоматизации выполнения ключевых операций, трассировка программы, статистическое и линейное профилирование, имитация внешних прерываний и потоков ввода/вывода данных через порты и отображаемые в память внешние устройства, графическое представление массивов данных, отладка в режиме многопроцессорной системы.

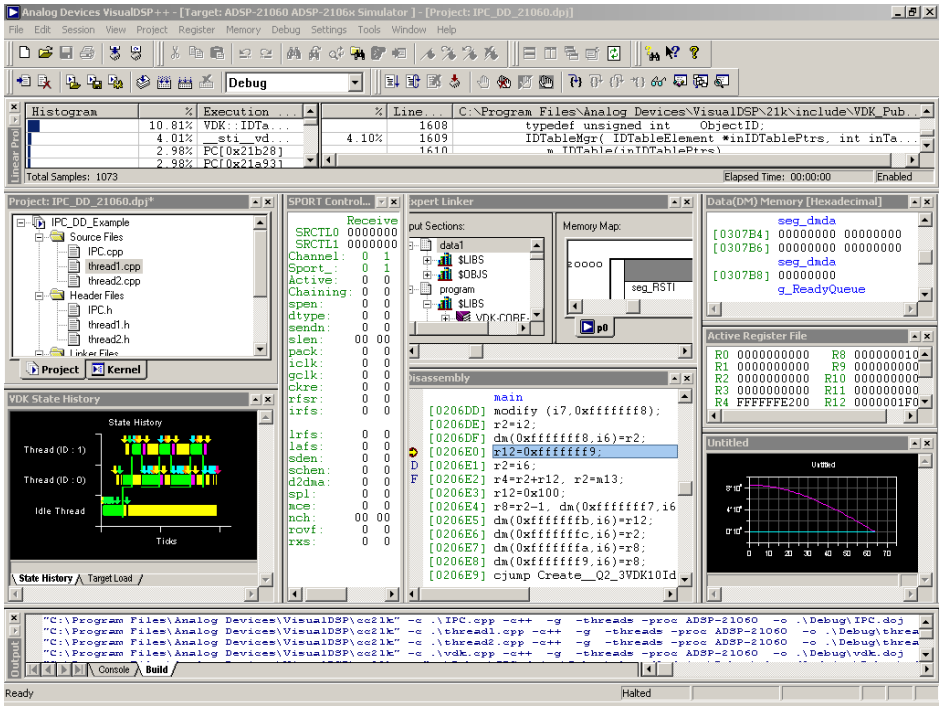


Рис.13. Интерфейс среды разработки VisualDSP++

4. Ядро VisualDSP++ (VDK) предоставляет пользователю базовые блоки для построения приложения и автоматической генерации отдельных элементов программного кода. Использование VDK нацелено на поддержку многозадачности с приоритетами, модульности, детерминированного принципа выполнения отдельных программных блоков, переносимости и независимости от типа архитектуры DSP-процессора.

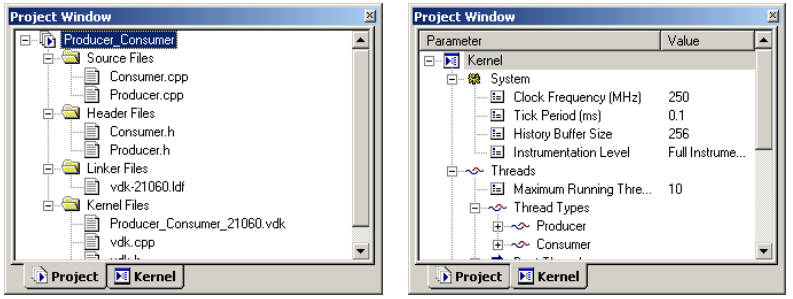
5. Возможности VisualDSP++ Component Software Engineering (VCSE) по созданию программных объектов и интерфейса для их использования. VCSE позволяет организовать повторное многократное использование однажды разработанных компонентов, облегчает использование компонентов из других проектов третьих разработчиков, написанных на языках C/C++ или ассемблере.

4.2. Этапы разработки приложения в среде VisualDSP++

Процесс разработки приложения в среде VisualDSP++ состоит из следующих основных шагов.

1. Создание проекта. Любое приложение в VisualDSP++ представляет собой проект, содержащий один или несколько файлов с программным кодом, файл

описания компоновщика и имеющий собственные опции для проекта и отдельных утилит – ассемблера, компоновщика и т.д. Файл проекта имеет расширение ".dpj" и содержит список файлов с их расположением, а также опции проекта (рис.14,а). Проект может содержать не только программные модули, но и другие файлы, например текстовые, файлы данных и т.п. Если проект содержит VDK, то окно проекта содержит вкладку Kernel (рис. 14,б), позволяющую определять специфические компоненты VDK – потоки, циклические приоритеты, семафоры, флаги событий, прерывания и драйверы устройств.



а б

Рис.14. Окно проекта

2. Установка общих опций проекта и выбор средств генерации исполняемого кода. Этот шаг предполагает задание процессора, для которого разрабатывается приложение, выбор утилит генерации кода и установку опций для них.

3. Работа с файлами проекта. Проект может содержать один или несколько программных файлов на языках C, C++ и ассемблере (с расширениями, соответственно .c, .cpp, .asm), файлы заголовков (с расширением .h), файл описания линкера (файл с расширением .ldf) и другие файлы. Добавление и удаление файлов может выполняться через окно проекта. Использование в каком-либо модуле данных из другого файла (например, инициализация массива значениями из текстового файла) оказывает влияние на порядок компиляции файлов проекта и указывается в генерируемом самой средой VisualDSP++ make-файле (с расширением .mak). Изменение взаимозависимостей между файлами проекта (Project Dependencies) может происходить при изменении опций проекта, добавлении новых файлов.

4. Установка опций для средств генерации кода. После создания проекта, выбора целевой архитектуры, разработки исходных файлов программы необходимо установить опции компилятора, ассемблера, компоновщика и других используемых утилит генерации кода (сплиттер, загрузчик). Следует отметить, что для каждого файла проекта могут быть использованы не только общие опции компиляции, установленные в окне Project Options для всех файлов данного типа, но указаны специфичные ключи компиляции (вплоть до исключения файла из процесса построения проекта).

5. Генерации отладочной версии кода. Генерация отладочной версии программы заключается в запуске компилятора, ассемблера и компоновщика с целью получения исполняемого файла с расширением .dxe. Ход процесса построения проекта, ошибки компиляции и компоновки отображаются в окне Output.

6. Установка отладочной сессии и загрузка кода проекта. Чтобы заставить VisualDSP++ имитировать выполнение инструкций или взаимодействовать с платами расширения, необходимо выбрать "отладочную сессию" (рис.15). Под сессией в VisualDSP++ понимается целевая конфигурация архитектуры системы, на которой будет запущена программа (модель DSP-процессора, программный симулятор или аппаратная плата).

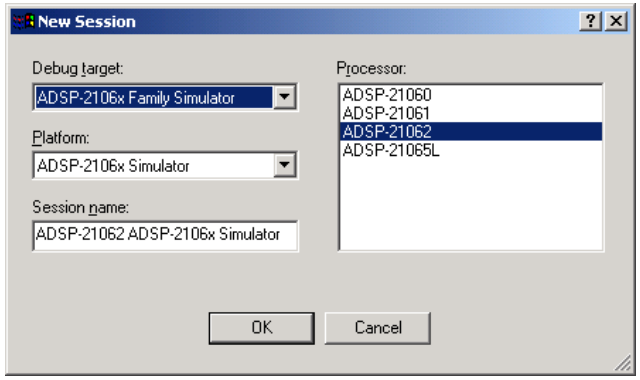


Рис.15. Окно выбора отладочной сессии

7. Выполнение и отладка программы. После успешной генерации исполняемого dxe-файла он может быть загружен в выбранную целевую конфигурацию и запущен в режиме выполнения или отладки. Если с момента последней генерации проекта произошли какие-либо изменения в составе или содержании файлов, опциях проекта, то VisualDSP++ может автоматически сгенерировать новую версию программного кода и загрузить ее в целевую конфигурацию.

8. Генерация конечного варианта проекта (Release). После отладки программы генерируется версия проекта для прошивки ПЗУ или загрузки во внутреннюю память DSP-процессора в реальной системе.

4.3. Работа с проектом и генерация исполняемого кода в среде VisualDSP++

4.3.1. Установка опций проекта

Разработка проекта в VisualDSP++ объединяет:

- выбор средств генерации кода, в качестве которых могут использоваться компилятор, ассемблер, компоновщик, сплиттер и загрузчик, как входящие в состав пакета VisualDSP++, так и поставляемые вместе со специфическим аппаратным обеспечением третьих разработчиков;
- установку опций построения проекта (опций компилятора, ассемблера и т.д.), причем для каждого файла проекта могут быть установлены собственные опции генерации кода, отличные от опций проекта;
- создание и модификацию исходных файлов программы.

Общие установки проекта (вкладка Project, рис.16) требуют указать архитектуру DSP-процессора, для которого разрабатывается программный код, и требуемый тип (формат) генерируемого выходного файла:

DSP executable file	- самостоятельный исполняемый файл, который может быть загружен в целевую систему;
DSP library file	- статическая библиотека объектных модулей;
DSP loader file	- файл для загрузчика, представляющий собой бинарный "образ" для начальной загрузки (boot) через ППЗУ, хост-компьютер, линк-порт;
DSP object file	- объектный файл машинных команд, не готовый для исполнения;
DSP splitter file	- бинарный "файл-образ" для прошивки ППЗУ, которые должен выполняться, размещаясь во внешней памяти;
VCSE component library	- библиотечный компонент VCSE.

На этой же вкладке указываются средства генерации кода, которые должны быть использованы на различных этапах построения проекта.

На следующих вкладках устанавливаются опции для каждого из выбранных средств генерации кода. Совокупность общих установок для всех средств генерации кода в VisualDSP++ называется конфигурацией. По умолчанию в проект включаются две конфигурации: Debug (отладка) и Release (готовый продукт). Отличия между ними заключаются главным образом в опциях компилятора языка C/C++ и компоновщика: конфигурация Debug ориентирована на получение отладочной версии кода и поэтому при ее использовании по умолчанию отсутствует какая-либо оптимизация кода, и отладочная информация включается в генерируемый объектный и исполняемый модули. Переход к конфигурации Release обычно осуществляется после отладки программы для получения варианта окончательного оптимизированного по быстродействию программного кода для загрузки целевую аппаратуру. Пользователь может изменять установки для обеих конфигураций и создавать собственные конфигурации с необходимыми установками на основе уже имеющихся.

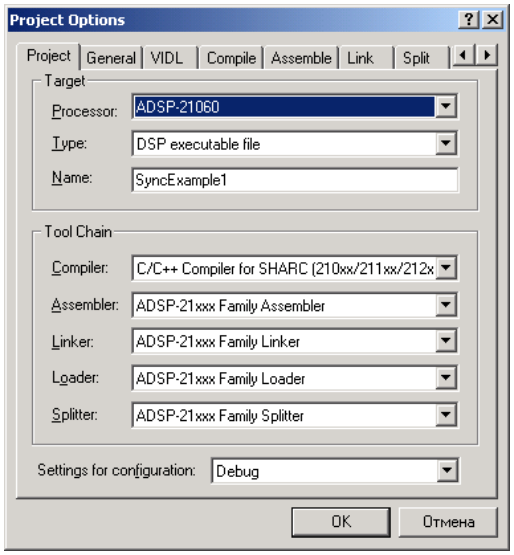


Рис.16. Окно опций проекта. Вкладка "Project"

Вызывая контекстное меню для файлов проекта (с расширениями .c, .cpp и .asm), пользователь может установить либо специфические опции генерации объектного кода при использовании стандартных средств пакета VisualDSP++ (отличающиеся от общих для всех файлов проекта опций), либо указать командную строку для вызова компилятора или ассемблера стороннего разработчика, либо вообще исключить данный файл из процесса построения программного кода. Файлы с собственными опциями компиляции и файлы, исключенные из процесса компиляции, помечаются в окне проекта специальными пиктограммами.

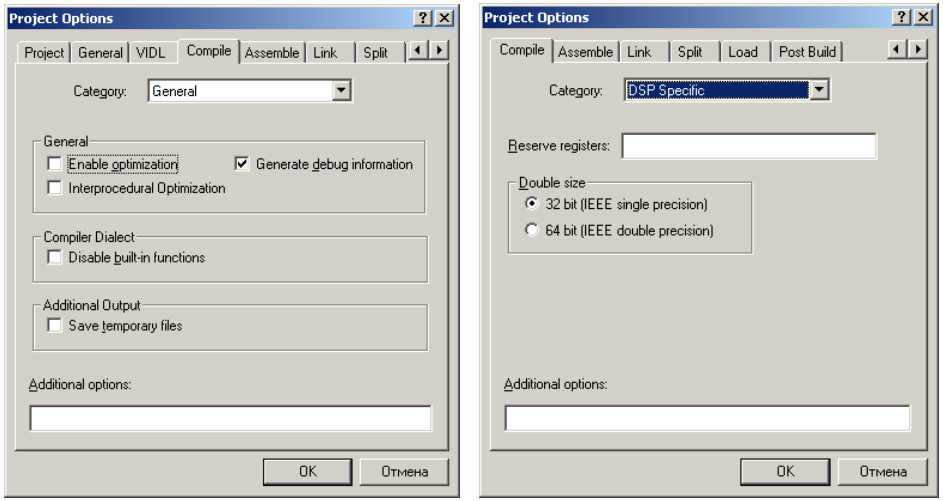
Кратко рассмотрим остальные вкладки окна Project Options:

1. Вкладка General позволяет задавать пути размещения временных и результирующих файлов, генерируемых средствами разработки.

2. На вкладке VIDL указаны опции для VIDL-компилятора. Язык описания прикладного интерфейса среды VisualDSP++ (VisualDSP++ Interface Definition Language, VIDL) представляет собой описательное средство для задания компонентов VCSE и интерфейса взаимодействия с ними. Компилятор с языка VIDL обрабатывает VIDL-спецификации и преобразует их во фрагменты программы на языках C, C++ или ассемблер. Эти фрагменты являются "скелетом" реализации компонента и его программного интерфейса, реализованном на соответствующем языке программирования.

3. Вкладка Compile служит для задания опций компилятора языка C/C++ и состоит из нескольких разделов. В частности, раздел General позволяет задавать режимы оптимизации, флаг использования встроенных функций (рис. 17,а). В разделе DSP Specific указываются регистры, которые компилятор не должен

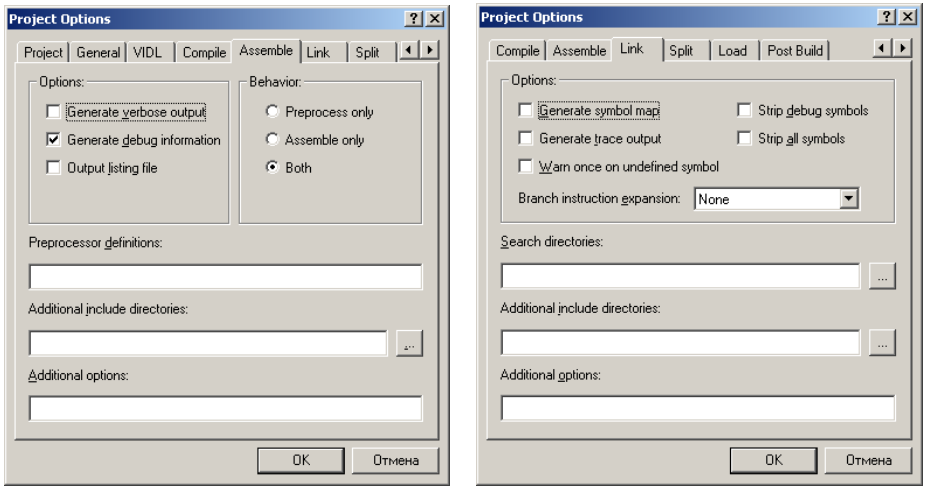
использовать при генерации кода (рис. 17,б). Остальные разделы содержат определения директив препроцессора языка C/C++, опций генерации предупреждений при компиляции и т.п.



а б

Рис.17. Вкладки опций компилятора и ассемблера

4. На вкладке Assemble (рис. 18,а) указываются режимы трансляции программного кода на языке ассемблера в объектный модуль, в том числе необходимость генерации информации для отладчика и дополнительные опции-ключи для ассемблера.



а б

Рис.18. Вкладки опций компилятора и ассемблера

5. Вкладка Link позволяет задавать опции компоновщика, сходные с опциями для разработки приложения для MS-DOS и Windows: ключи генерации таблицы символов, компоновки с трассировкой, игнорирования символьной информации для отладчика при компоновке и др. (рис. 18,б).

6. На вкладке Split размещены элементы управления для правильной генерации карты памяти при использовании режима работы процессора с выполнением исполняемого кода из внешней памяти (без загрузки во внутреннюю память).

7. Вкладка Load используется для выбора средства инициализации памяти целевой системы. Использование утилиты Mem21k для инициализации пространства памяти системы скомпилированным программным кодом необходимо только если программа должна выполняться во внешней памяти без загрузки во внутреннюю память процессора. Во всех остальных случаях следует выбирать утилиту Loader, которая преобразует сгенерированный исполняемый файл в файл образа памяти, подлежащий загрузке в процессор при включении питания (boot-loadable file). В зависимости от сигналов на ножках процессора загрузка может выполняться из внешнего ППЗУ, через линк-порт, хост-интерфейс, SPI-интерфейс (для ADSP-2116x и выше) или вообще отсутствовать. Загрузка программы во внутреннюю память процессора осуществляется следующим образом. При включении питания процессор загружает через заданный канал небольшую подпрограмму из 256 инструкций – загрузочное ядро (Boot Kernel) и затем передает управление этой подпрограмме. Утилита Loader конфигурирует библиотечный загрузочное ядро (Boot Kernel) таким образом, чтобы эта подпрограмма смогла загрузить программные сегменты по соответствующим адресам внутренней памяти (в конечном итоге перезаписав их на самое себя) и передать управление на первую команду пользовательского приложения.

8. И, наконец, вкладка Post Build позволяет после успешного построения проекта выполнять дополнительные команды или программы.

4.3.2. Структура проекта

При создании проекта необходимо руководствоваться следующими правилами:

- в проект могут быть включены файлы с любым расширением, но каждый файл должен иметь уникальное имя;
- в проекте может быть только один LDF-файл компоновщика;
- однотипные файлы могут быть помещены в одну папку. При создании в проекте новой папки указывается, файлы с каким расширением она должна хранить. Если в проект добавляется файл с расширением, которое не поддерживает ни одна из имеющихся папок, то он помещается в корень проекта. Папки по умолчанию содержат файлы со следующими расширениями:

Папка	Ассоциированные файлы по умолчанию
Source Files	.c, .cpp, .cxx, .asm, .dsp, .s
Header Files	.h, .hpp, .hxx
Linker Files	.ldf, .dlb, .doj
Kernel Files	.vdk
Component Files	.idl, .xml
Documentation	.hhc, .hhk, .html, .css, .js

- в ходе построения проекта могут использоваться файлы следующих основных типов:

Тип файла	Категория	Назначение / содержание
.asm, .dsp	Исходный файл	Файлы с исходными текстами на языке ассемблера
.c	Исходный файл	Файл с исходной программой на языке ANSI C и расширениями, поддерживаемыми компилятором фирмы Analog Devices
.cpp, .cxx, .hpp, .hxx	Исходный файл	Файлы с исходной программой на языке ANSI C++ и заголовочные файлы
.dpj	Файл проекта	Описание опций проекта для построения исполняемой программы
.ldf	Файл описания компоновщика	Текстовый файл, содержащий команды для компоновщика на специальном языке описания сценария компоновки (проще говоря – карта памяти системы)
.s, .pp, .is	Промежуточные файлы	Ассемблерные файлы, сгенерированные препроцессором
.doj	Объектный файл	Выходной бинарный файл ассемблера
.dlb	Библиотечный файл	Бинарный библиотечный файл в формате ELF
.h	Файл заголовка	Файл описания заголовков, используемый препроцессором, исходный файл для компилятора и ассемблера
.dat	Файл данных	Файл, используемый ассемблером для инициализации данных (обычно массивов)
.dxe, .sm, .ovl, .dlo	Файлы отладчика	Выходные бинарные файлы компоновщика в формате ELF/DWARF
.map	Карта памяти компоновщика	Выходной (необязательный) текстовый файл компоновщика, содержащий информацию о структуре памяти
.tcl	Файл описания сценария	Файл на языке Tcl (Tool Command Language) для задания сценария (команд пакетного режима) работы со средой VisualDSP++
.lst	Файл листинга	Выходной (необязательный) текстовый файл компоновщика с листингом программы
.ldr, .bnm	Файлы загрузчика	Результат работы утилиты Loader в ASCII-формате. Используется для создания загрузочной ППЗУ
.s_#, .h_#, .stk	Файлы ППЗУ	Результат работы утилиты Splitter в формате Motorola
.vdk	Файл поддержки VDK	Обеспечивает поддержку функций VDK в проекте
.mak, .mk	Make-файл	Автоматически генерируемый make-файл, содержащий опции и установки, необходимые для построения проекта

4.3.3. Построение проекта (генерация исполняемого кода)

Построение проекта (Build) заключается в выполнении последовательности операций (таких, как вызов препроцессора, компиляция, ассемблирование и компоновка) над входящими в состав проекта файлами. В процессе построения проекта VisualDSP++ обрабатывает файлы, которые были модифицированы с момента последней генерации кода (out of date) и файлы, которые включают модифицированные файлы (например, если был изменен заголовочный файл с расширением .h, то будет обработан файл на языке C/C++, который включает этот заголовочный файл). Для выбора таких файлов VisualDSP++ использует информацию о зависимостях (dependencies) между файлами.

Выбор опции Rebuild All заставляет VisualDSP++ выполнить обработку всех файлов проекта независимо от того, были они модифицированы или нет.

Дополнительно к стандартной последовательности генерации программного кода вызовов утилит могут быть вызваны дополнительные программные модули, имена и размещение которых, а также порядок их вызовов через командную строку указывается в окне опций проекта (вкладка Post Build).

4.4. Базовые элементы оконного интерфейса VisualDSP++

Интерфейс VisualDSP++ содержит стандартные элементы управления в виде меню, контекстных меню, панелей инструментов. К основным окнам интегрированной оболочки VisualDSP++, используемым при создании проекта, можно отнести следующие окна:

1. Окно проекта (Project Window), содержащее вкладку Project с информацией о структуре проекта и (если используется VDK) вкладку Kernel с основными параметрами ядра приложения и средств поддержки многозадачности (параметров нитей, семафоров и т.п.);

2. Окно редактора (Editor Window) с поддержкой работы с несколькими файлами, функциями поиска, установки закладок (bookmarks), работы с буфером обмена, перемещением к очередной синтаксической ошибке и другими возможностями;

3. Окно вывода (Output Window), используемое для вывода системной информации и организации ввода/вывода текстовых сообщений. Окно вывода содержит две вкладки:

- вкладка Console позволяет просматривать системные сообщения об ошибках VisualDSP++ или подключенной платы отладки, выполнять вывод из программ на языке C/C++ в стандартный поток STDIO, набирать и запускать на выполнение Tcl-сценарии (рис.19);

- вкладка Build отображает информацию о ходе построения проекта и наличии ошибок при компиляции и компоновке программы (рис. 20).

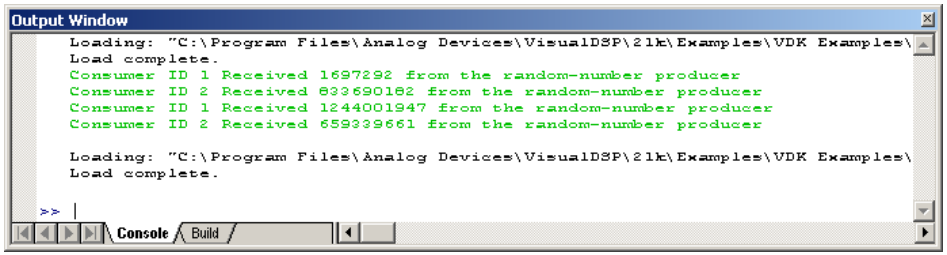


Рис.19. Вкладка Console окна Output Window

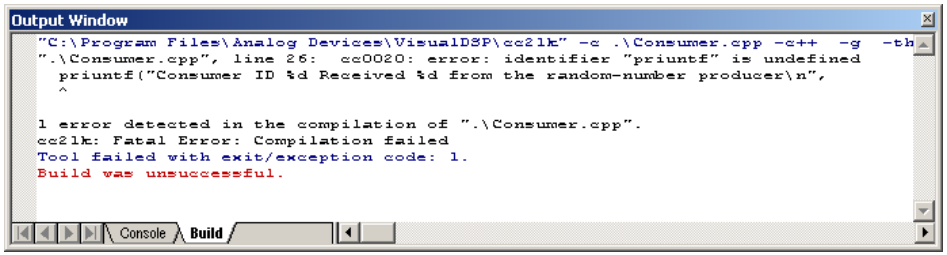


Рис.20. Вкладка Build окна Output Window

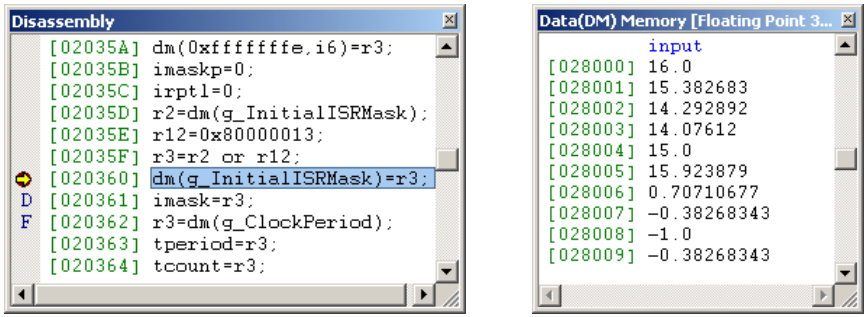
Сообщения об ошибках и предупреждениях делятся на следующие категории:

Категория	Описание
Fatal error	Характеризует фатальную ошибку, вследствие которой дальнейшая обработка (компиляция) программы невозможна
Error	Характеризует ошибку, которая позволяет продолжить процесс компиляции для выявления других ошибок
Warning	Характеризует ситуацию, которая не влияет на процесс компиляции, но потенциально может вызвать ошибку при выполнении
Remark	Примерно соответствует Warning, но имеет еще меньшую важность с точки зрения компилятора

4.5. Средства отладки проекта

К основным возможностям отладки программного обеспечения в среде VisualDSP++ относятся:

1. Просмотр исполняемого кода в дизассемблированном формате (окно Disassembly). Основные функции этого окна – просмотр и анализ кода, сгенерированного компилятором C/C++, на предмет его оптимальности, оперативная модификация отдельных инструкций (путем непосредственного ввода новых инструкций), установка/сброс точек останова (breakpoints) и анализ содержимого трехступенчатого конвейера во время пошагового выполнения программы (рис. 21,а).



а б

Рис.21. Окна Disassembly и Memory

Если при компиляции проекта был установлен флажок включения в исполняемый файл информации для отладчика, то при пошаговой отладке программы можно наблюдать синхронное перемещение подсвеченной строки как в окне с дизассемблированным (исполняемым) кодом, так и в окне редактора с исходным текстом проекта.

2. Просмотр содержимого памяти (окна Memory Map и Memory). Окно карты памяти Memory Map содержит справочные сведения об адресах размещения в памяти целевой системы сегментов, описанных в LDF-файле. Окно Memory позволяет получить детальную информацию о содержимом каждой ячейки внутренней или внешней памяти (рис. 21,б).

Удобство работы с этим окном обеспечивается благодаря ряду возможностей, реализуемых при помощи контекстного меню:

- переход к требуемому адресу (команда Go To) выполняется особенно просто, если необходимо перейти к адресу, помеченному меткой (естественно, если в исполняемый модуль включена информация для отладчика);

- редактирование (команда Edit) содержимого ячейки памяти по выбранному адресу;

- выгрузка содержимого памяти в файл в требуемом формате (команда Dump) позволяет сохранить образ памяти для последующего использования или анализа;

- загрузка содержимого памяти из внешнего файла или заполнения диапазона адресов памяти константой (команда Fill). Удобна при инициализации массивов начальным значением;

- функция Tracking для оперативного обновления окна содержимого памяти в соответствии с введенным выражением. Особенно удобно использование этой возможности при необходимости отслеживания положения указателя в памяти. В этом случае в качестве выражения может быть введено имя регистра (например, $I4 – знак "$" означает, что используется регистр, I4 – имя регистра). При каждом изменении значения регистра окно Memory будет позиционироваться в памяти в соответствии с его новым значением;

- представление данных памяти в формате, наиболее удобном для отладки

(двоичном, восьмеричном, шестнадцатеричном, знаковом/беззнаковом целом, знаковом/беззнаковом дробном, с плавающей запятой).

3. Окна просмотра содержимого регистров процессорного ядра (Core) и процессора ввода/вывода (IOP). Более 20 окон для регистров процессорного ядра и 10 окон для IOP-процессора позволяют не только отслеживать содержимое регистров регистрового файла, но и детально анализировать ход выполнения программы (рис. 22).

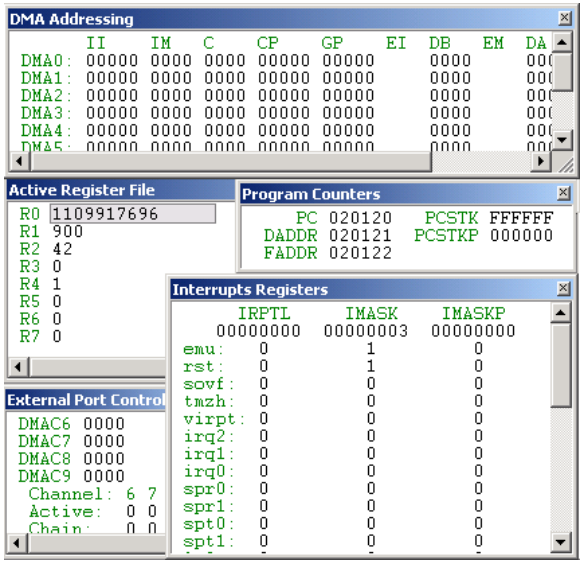


Рис.22. Просмотр содержимого регистров процессорного ядра и IOP-процессора

Кроме предопределенных окон, пользователь может формировать окна с необходимыми наборами регистров по собственном желанию (Custom), а также выгружать содержимое всех регистров процессора в виде текстового файла.

4. Поддержка вложенных вызовов функций на языках высокого уровня (окна Locals и Call Stack). Окно Locals показывает значения локальных переменных той функции, телу которой принадлежит исполняемая инструкция. Окно Call Stack отражает вложенность вызовов функций.

5. Анализ хода выполнения программы (окна Trace, Linear/Statistical Profiling). Окно Trace позволяет анализировать длительность исполнения каждой инструкции в тактах, адреса, по которым выполняются обращения к памяти, тип обращения (чтение/запись) и прочитанное/записанное значение (рис.23). Для включения/выключения режима трассировки необходимо использовать пункт меню Tools->Trace.

Одним из наиболее полезных при анализе производительности программного кода является окно профилирования (рис.24). Основное назначение профилирования – сбор статистической информации о том, какой процент

времени занимает выполнение той или иной функции в общем времени работы программы. Следует различать линейное (Linear) и статистическое (Statistical) профилирование. Линейное профилирование имеет место при отладке программы в симуляторе. В этом случае учитывается каждое изменение счетчика команд. Статистическое профилирование используется при анализе работы программного обеспечения на целевой аппаратуре, подключенной к компьютеру через интерфейс JTAG. В этом случае значения счетчика команд процессора анализируются не каждый такт, а через определенные временные интервалы.

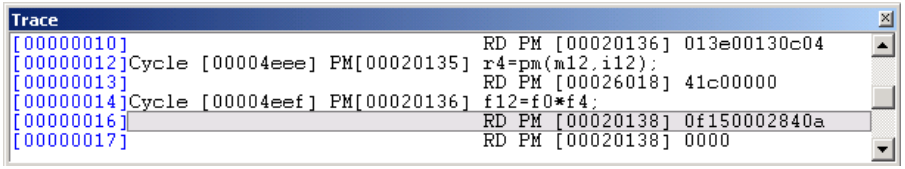


Рис.23. Окно отладки Trace

Левая часть окна содержит информацию о проценте времени выполнения каждой функции (если отладочная информация включена в исполняемый файл). Щелчок на имени функции приводит к отображению в правой части окна подробной статистики по данной функции.

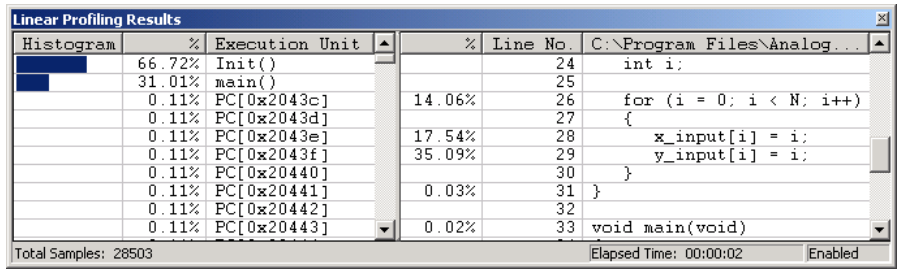


Рис.24. Окно результатов линейного профилирования

6. Графическое отображение и аудиовоспроизведение содержимого памяти (окна Plot, ImageViewer). Окно Plot предоставляет широкие возможности для графического отображения содержимого памяти в виде графиков различных типов (рис. 25). При этом возможны автоматическая обработка указанного диапазона адресов и вывода на график результата обработки (например, не значений буфера, а спектрограммы значений). Графики могут быть экспортированы в виде графического файла, распечатаны и даже воспроизведены через звуковую плату компьютера.

Для повышения эффективности создания программного обеспечения систем обработки изображений в состав среды VisualDSP++ входит утилита – просмотрщик изображений, которая позволяет загрузить/выгрузить изображение в одном из стандартных форматов (bmp, jpg, png) из файла в память системы.

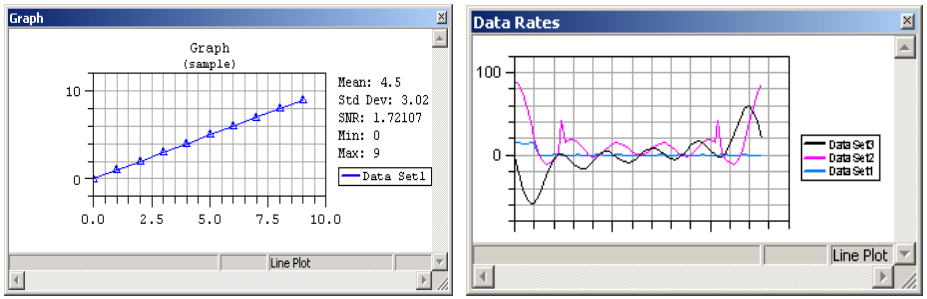


Рис.25. Окно построения графиков

7. Поддержка возможностей VDK (окна VDK Status и VDK History). Окно статуса VDK отображает информацию о параметрах, заданных в окне ядра VDK проекта. Окно истории графически иллюстрирует передачу управления между различными нитями в многозадачной системе и позволяет оценить общую загрузку процессора.

4.6. Выполнение программы

Перед загрузкой исполняемой dxe-программы необходимо убедиться в правильности выбора сессии: тип выбранного процессора должен совпадать с типом, указанным в команде PROCESSOR LDF-файла описания компоновщика.

Выполнение программы контролируется с использованием стандартных команд:

- Run (F5) – запуск на выполнение загруженного файла. Выполнение продолжается до точки останова (breakpoint), останова по условию (watchpoint) или вмешательства пользователя (команда Halt);
- Halt (Shift+F5) останавливает выполнение программы и обновляет содержимое всех окон;
- Run to Cursor (Ctrl+F10) – выполнение до строки, в которой находится курсор. Курсор может быть помещен как в окно с дизассемблированным кодом, так и в окно редактора с исходным модулем на C/C++ или ассемблере (если в исполняемый модуль при построении проекта была включена отладочная информация);
- Step Over (F10) – выполнение шага без захода в функцию. "Работает" в окне редактора с исходным модулем на языке C/C++;
- Step Into (F11) – выполнение шага с максимальной детализацией;
- Step Out Of (Alt+F11) – завершение выполнения функции и возврат на команду, следующей за командой вызова для программ на языке C/C++. Режим доступен в окне редактора с исходным модулем на языке C/C++.

При повторном запуске программы командой Restart сначала выполняется сброс процессора (аналог – команда Reset). При отладке в симуляторе

сбрасываются значения всех регистров, а значения ячеек памяти не изменяются (для сброса памяти необходимо загрузить исполняемую программу вновь). При отладке в режиме аппаратной эмуляции сброс при выполнении команд Restart совершенно идентичен выполнению команды Reset: выполняется сброс только тех регистров, для которых определены значения "по умолчанию" (при сбросе). Значения остальных регистров не изменяются.

Для приостановки выполнения программы могут использоваться точки останова (breakpoints) и останова по событию (watchpoints). Если точки условного и безусловного останова, позволяющие прерывать выполнение программы перед выполнением инструкции по указанному адресу, являются "штатным" средством практически всех современных средств отладки, то точки watchpoints поддерживаются не всеми отладчиками. Назначением "останова по событию" является прерывание выполнения программы не на заданной инструкции, а при выполнении того или иного события (точка прерывания заранее неизвестна). Такими событиями в VisualDSP++ являются, например, запись или чтение значения из указанного регистра (любая операция записи/чтения над регистром или запись/чтение только определенного значения), обращение к диапазону ячеек памяти, изменение содержимого аппаратных стеков циклов, программного секвенсора и статуса.

Остановы по событию могут быть использованы только в режиме программной симуляции, тогда как точки останова breakpoints – и при симуляции, и при эмуляции.

4.7. Моделирование ввода/вывода данных через порты процессора

VisualDSP++ имеет возможности имитации ввода/вывода данных через последовательные порты, линк-порты и отображаемые во внешнюю память порты ввода/вывода. При операции чтения из порта отладчик выполняет чтения из указанного текстового файла, при записи значения в порт – соответственно операцию записи. Все установленные соответствия между потоками ввода/вывода и файлами данных отображаются в окне потоков (пункт меню Settings->Streams…). Установка соответствия между потоком и файлом, например, при вводе данных осуществляется путем указания файла, откуда выполняется чтение (Source) и порта процессора, в который помещается прочитанное значение (Destination). При моделировании вывода данных в качестве источника (Source) указывается порт процессора, а в качестве приемника (Destination) – имя файла (рис. 26).

Следует иметь в виду, что имитация ввода/вывода отсчетов сигнала посредством использования файлов данных позволяет лишь отладить логику работы программы, однако не позволяет протестировать временные соответствия между поступлениями отсчетов, поскольку чтение из файла (при вводе данных) реально выполняется только тогда, когда процессор запросит очередное значение из порта.

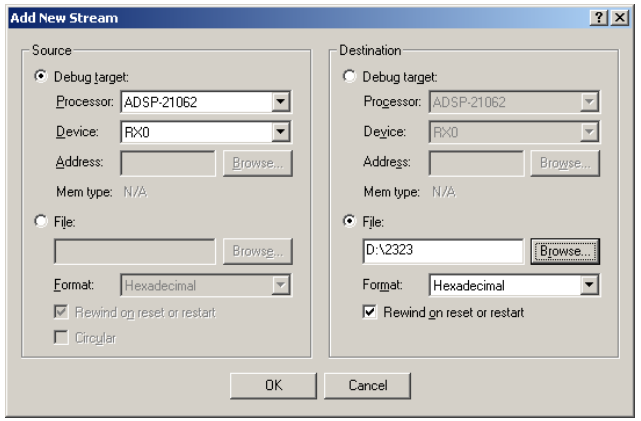


Рис.26. Установка соответствия потока ввода/вывода и файла

4.8. Моделирование внешних событий

Архитектура системы ЦОС практически всегда предполагает возможность взаимодействия с внешними устройствами, причем генерация управляющих и информационных сигналов от сопряженной аппаратуры имеет асинхронный характер. VisualDSP++ позволяет имитировать генерацию подобных сигналов с учетом их относительной "непредсказуемости" (рис.27).

Внешние события по линиям, выбранным в окне Interrupt Timing, генерируются через случайные интервалы времени (количество процессорных циклов) в пределах от Min до Max.

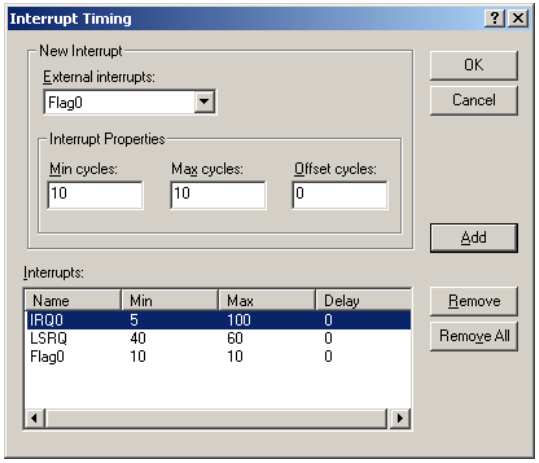
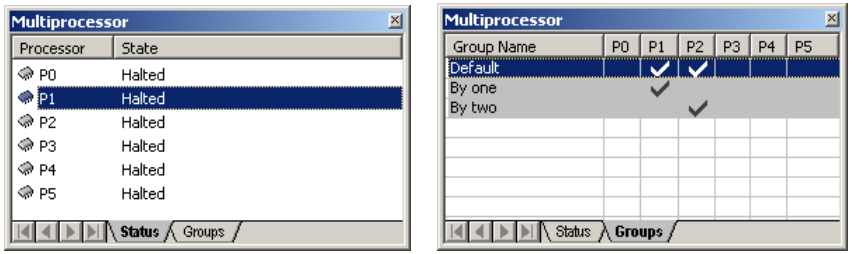


Рис.27. Установки для имитации внешних событий

4.9. Отладка программного обеспечения для многопроцессорной системы

Разработка и отладка ПО для многопроцессорной системы в VisualDSP++ требует соответствующего описания архитектуры системы в LDF-файле. При генерации исполняемого программного кода для каждого процессора генерируется свой образ памяти.

При запуске программы на экране появляется окно Multiprocessor, имеющее вкладки Status и Groups (рис. 28). На вкладке Status путем выбора соответствующего процессора можно загружать в окна отладчика содержимое памяти, регистров, текст программы и т.п., относящееся к выбранному процессору. Имеется также возможность закрепить связь между содержимым конкретного окна отладчика и номером процессора (команда Pin to Processor в контекстном меню окна).



а б

Рис.28. Вкладки окна Multiprocessor при отладке многопроцессорной системы

Вкладка Group позволяет группировать процессоры с тем, чтобы выполнять стандартные команды выполнения при отладке (Multiprocessor Run, Multprocessor Step и т.п.) для активной группы процессоров. Остальные процессоры будут в это время простаивать.

Контрольные вопросы

1. Какие возможности по работе с мультимедийными данными предоставляется данная интегрированная среда?
2. В чем отличия имитации ввода/вывода данных в VisualDSP++ от реального ввода/вывода?
3. Для чего используется режим Tracking при просмотре содержимого памяти в режиме отладки?

5. РАЗРАБОТКА ФАЙЛА ОПИСАНИЯ АРХИТЕКТУРЫ ДЛЯ КОМПОНОВЩИКА

5.1. Назначение файла описания для компоновщика

Для указания компоновщику, в какие участки памяти следует отображать программный код или данные, используется LDF-файл (Linker Description File). LDF-файл состоит из команд, описывающих используемые компоненты программного кода и целевой системы. Суть процесса компоновки с LDF-файлом заключается в следующем:

- в исходной программе описывается одна или более входных секций (Input Sections) как место размещения программных объектов, например:

```
.section /DM asmdata       // секция asmdata содержит массив my_buffer
.var my_buffer[3];

.section /PM asmcode       // секция asmcode содержит инструкции
r0 = 0x1234;
r1 = 0x5678;
r2 = r0 + r1;
```

- компилятор и ассемблер генерируют объектный код с сохранением секционной структуры. Каждая входная секция может содержать несколько элементов программы, но каждый программный элемент может входить только в одну входную секцию;

- компоновщик отображает входную секцию из объектного файла в выходную секцию (Output Section) по правилам, указанным в LDF-файле. В одну выходную секцию могут быть помещены несколько входных секций;

- компоновщик отображает каждую выходную секцию в сегмент памяти (Memory Segment), который представляет собой непрерывный участок памяти системы с DSP. Элементы, размещаемые в одном сегменте, должны иметь одинаковую разрядность, в противном случае они должны размещаться в разных сегментах. В один сегмент памяти могут быть отображены несколько выходных секций.

Пример LDF-файла приведен на листинге.

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )
$OBJECT1 = $COMMAND_LINE_OBJECTS;
$LIBRARIES = lib060.dlb;
MEMORY  {
 mem_rsti { TYPE(PM RAM) START(0x00008000) END(0x000080ff) WIDTH(48) }
 mem_pmco { TYPE(PM RAM) START(0x00008100) END(0x000087ff) WIDTH(48) }
 mem_pmda { TYPE(PM RAM) START(0x00009000) END(0x00009fff) WIDTH(32) }
 mem_dmda { TYPE(DM RAM) START(0x0000c000) END(0x0000dfff) WIDTH(32) }
}
PROCESSOR p0 {
OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
SECTIONS       {
 dxe_rsti { INPUT_SECTIONS ($OBJECT1 (seg_rth) $LIBRARIES(seg_rth)) } > mem_rsti
 dxe_pmco { INPUT_SECTIONS ($OBJECT1 (seg_pmco)) } > mem_pmco
 dxe_pmda { INPUT_SECTIONS ($OBJECT1 (seg_pmda)) } > mem_pmda
 dxe_dmda { INPUT_SECTIONS ($OBJECT1 (seg_dmda)) } > mem_dmda
    }   }
```

Здесь:

1. Команда ARCHITECTURE определяет тип DSP в системе ЦОС и, следовательно, конфигурацию и размер памяти, возможные разрядности ячеек памяти, распределение памяти по блокам и т.п. В общем, ту информацию, которая используется отладчиком, компоновщиком, загрузчиком, сплиттером и другими утилитами в среде разработки для ADSP. Необходимо, чтобы указанный тип процессора поддерживался имеющейся версией среды разработчика. В имени архитектуры различаются малые и большие буквы.

2. Команда SEARCH_DIR добавляет пути поиска библиотек и объектных файлов к директории по умолчанию (обычно директория по умолчанию – это тот каталог, в котором находятся файлы проекта). Если добавляемых путей несколько, то они разделяются точкой с запятой. Длинные пути записываются в двойных кавычках (например: "C:\Program Files\Analog Devices"). Часто при указании пути к файлам стандартных библиотек и заголовков используется стандартное макроопределение $ADI_DSP, вместо которого подставляется имя каталога, в который установлена среда VisualDSP++.

3. Макроопределение $OBJECT1 получает текстовую строку параметров, передаваемую компоновщику при вызове. Стандартное макроопределение $COMMAND_LINE_OBJECTS возвращает список объектных (.doj) и библиотечных (.dlb) файлов, указанных при вызове компоновщика в командной строке.

4. Макроопределение $LIBRARIES задает библиотечный файл lib060.dlb, содержащий библиотеку подпрограмм и секций. Если описание секции или подпрограммы с указанным именем не может быть найдено в объектных файлах, то в соответствие с командой INPUT_SECTIONS поиск может быть продолжен в библиотеке. Это позволяет использовать LDF-файл "по умолчанию", например, при разработке программ на языках высокого уровня C/C++.

5. Команда MEMORY определяет физическую память системы ЦОС. Список ее аргументов делит память на сегменты памяти (Memory Segments), задавая для каждого из них имя, адреса начала и окончания, разрядность слов сегмента и тип памяти (DM, PM, RAM, ROM). Каждый сегмент памяти должен иметь уникальное имя.

6. Команда OUTPUT указывает компоновщику на необходимость генерации выходного исполняемого файла (.DXE) с заданным именем. Имя файла передается компоновщику через командную строку и может быть доступно через стандартную макрокоманду $COMMAND_LINE_OUTPUT_FILE. Этот макрос может быть использован в LDF-файле только один раз. Поэтому при генерации исполняемых файлов для многопроцессорной системы (когда для каждого процессора должен быть сгенерирован свой DXE-файл), необходимо явно указывать имена выходных файлов.

7. Команда SECTIONS определяет местоположение кода или данных в физической памяти. Используя информацию, указываемую для каждого правила отображения (выходной секции), компоновщик "берет" с указанным именем из

входной секции (объектного файла или библиотеки), размещает его в выходной секции и отображает выходную секцию в сегмент памяти, заданный командой MEMORY. При этом оператор INPUT_SECTIONS определяет объектный файл(ы) и при необходимости файл(ы) библиотеки, используемые компоновщиком для поиска входного объекта.

5.2. Структура LDF-файла

Для получения простого и понятного LDF-файла следует разрабатывать и модифицировать его параллельно с проектированием структуры системы ЦОС с DSP. При этом следует принимать во внимание следующие рекомендации:

- каждому процессору системы должна соответствовать отдельная секция, описываемая командой PROCESSOR;
- размещение каждой команды MEMORY в структуре LDF-файла должно производиться с учетом ее области видимости: память, используемая данным процессором, должна быть описана внутри соответствующей секции PROCESSOR; описания разделяемой памяти (SHARED_MEMORY) и памяти многопроцессорной (MPMEMORY) должны быть размещены в области глобальной видимости до первой команды PROCESSOR;
- при использовании команд MPMEMORY или SHARED_MEMORY они должны быть помещены в область глобальной видимости, т.к. они описывают системные ресурсы многопроцессорной системы.

5.2.1. Области видимости в LDF-файле

В LDF-файле используются две области видимости: *глобальная* и *внутрикомандная*.

Внутрикомандная область видимости определяется содержанием команды OUTPUT, используемой внутри команд PROCESSOR и SHARED_MEMORY. Действие команд и выражений, размещенных в пределах области видимости команды, ограничивается только этой командой.

Глобальная область видимости определяется вне каких-либо команд. Команды и выражения, размещенные в глобальной области видимости, доступны в глобальной области и видны во всех вложенных областях.

Примечание. Макросы в LDF имеют глобальную область видимости независимо от размещения их описания.

На рис.29 приведен пример, поясняющий действие областей видимости. Так, команда MEMORY, размещенная в области глобальной видимости, доступна во всех командных областях видимости, а действие команд MEMORY, описанных внутри команд, ограничено этими командами.

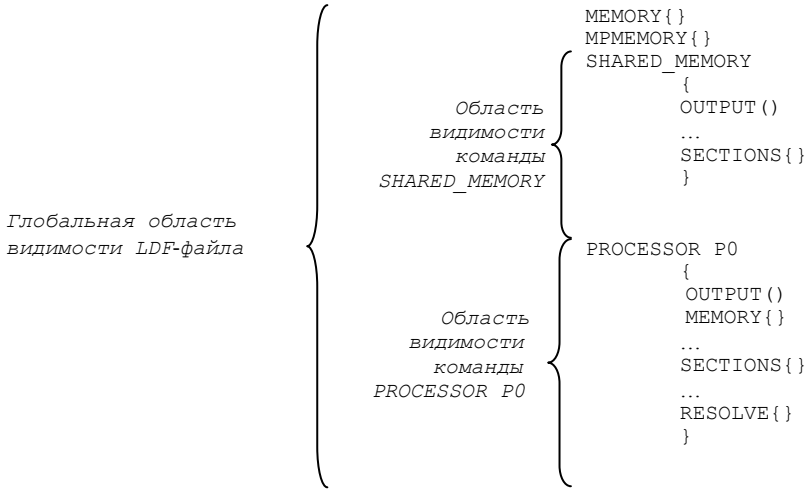


Рис.29. Области видимости различных команд

5.2.2. Операторы и макросы LDF-файла

Операторы в LDF-файле обычно используются для работы с адресами памяти. Выражение, содержащее оператор, завершается точкой с запятой, за исключением тех случаев, когда оператор используется как переменная.

LDF-макрос представляет собой текстовую строку с присвоенным именем. Когда компоновщик встречает имя макроса, он заменяет имя макроса его телом, т.е. выполняет простую текстовую подстановку. Компоновщик поддерживает ряд предопределенных макросов ($COMMAND_LINE_LINK_AGAINST, $ADI_DSP, $COMMAND_LINE_OBJECTS и др.) и позволяет пользователю создавать собственные макросы. При вызове макроса перед его именем должен быть указан знак доллара.

Более подробную информацию об операторах и макросах LDF-файла можно найти в документации по утилитам VisualDSP++.

5.3. Основные команды LDF-файла

Команды LDF-файла описывают моделируемую систему и действуют (т.е. оказывают влияние на процесс компоновки и другие команды) в той области видимости, в которой они описаны. Рассмотрим некоторые наиболее часто используемые команды LDF-файла.

Команда *ARCHITECTURE*(<*processor*>) задает процессор в моделируемой системе. LDF-файл должен содержать единственную команду ARCHITECTURE(), причем она должна быть описана глобально и действовать во всем LDF-файле. В

качестве <processor> может быть указано одно из следующих значений: ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, ADSP-21160, ADSP-21161 и др., например:

```
ARCHITECTURE(ADSP-21060)
```

Команда *INCLUDE*(<*filename*>) определяет дополнительный LDF-файл, который будет обработан прежде, чем продолжится обработка текущего LDF-файла. При этом типы процессора в команде ARCHITECTURE() во всех соединяемых LDF-файлах должны совпадать (обычно тип процессора задается в одном, самом "старшем", LDF-файле). Имя файла берется в кавычки.

Команда *LINK_AGAINST*(<*executable file names*>) указывает линкеру на необходимость проверить один или несколько указанных исполняемых файлов (разделенных пробелами) с расширением .dxe или .sm для разрешения переменных и меток, которые не были разрешены локально. Эта инструкция обычно присутствует в LDF-файле при компоновке программы для многопроцессорной системы. Появление команды LINK_AGAINST() возможно в командах PROCESSOR(), SHARED_MEMORY(), OVERLAY_INPUT(). Синтаксис команды (на примере использования внутри команды PROCESSOR) выглядит следующим образом:

```
PROCESSOR processor_name
  { …
    LINK_AGAINST (executable_file_names)
    …    }
```

Команда *MAP*(<*project_file_name.map*>) формирует map-файл. Включение этой команды в LDF-файл аналогично вызову линкера с ключом -Map.

Команда *MEMORY {}* описывает сегменты физической памяти моделируемой системы. После объявления в этой команде компоновщик может использовать имена сегментов при описании секций программы командой SECTIONS. Длина имени сегмента ограничена 8-ю символами.

LDF-файл должен содержать глобальную команду MEMORY для описания любой глобальной памяти в системе и может содержать несколько команд MEMORY для описания памяти каждого процессора в системе.

Примечание. В одном LDF-файле предпочтительнее использовать или только глобальное описание памяти процессоров (когда конфигурация памяти одинакова), или только локальные команды MEMORY для каждого процессора.

Количество сегментов, которые могут быть описаны в команде MEMORY, не ограничено. За командой MEMORY, независимо от того, в какой области видимости она записана, должна следовать команда SECTIONS.

Если карта памяти моделируемой системы не описана с использованием команды MEMORY, то программа не будет скомпонована. Если в описании соответствующей секции (в команде SECTIONS) для данного сегмента потребуется объем памяти больший, чем выделено, то линкер остановит компоновку и выдаст сообщение об ошибке.

Синтаксис команды MEMORY:

```
MEMORY {    <описание сегмента>
            <описание сегмента>
            <описание сегмента>
          …}
```

где <описание сегмента> представляет собой структуру, состоящую из следующих полей:

```
<имя сегмента>  {
                TYPE (PM | DM   RAM | ROM | PORT)
                START (адресное выражение)
                LENGTH (длина в словах) | END (адресное выражение)
                WIDTH (размер в битах)
                }
```

Имя сегмента может начинаться с буквы, символа подчеркивания и включать любые буквы, цифры и символы подчеркивания. Тип сегмента TYPE заносится линкером в исполняемый файл для использования другими средствами разработки ПО, например, сплиттером. Адреса начала (START) и конца (END) сегмента должны представлять собой абсолютный адрес или константное выражение, определяющее адрес. Размер сегмента LENGTH задается в виде абсолютного значения или константного выражения и указывается в словах. При указании размера сегмента необходимо использовать только одно из ключевых слов LENGTH или END (понятно, что START + LENTGH = END). Поле размерности слов сегмента WIDTH задает размер каждого слова сегмента в битах.

Пример использования команды MEMORY:

```
MEMORY {
 mem_rth  { TYPE(PM RAM) START(0x20000) END(0x200ff) WIDTH(48) }
 mem_init { TYPE(PM RAM) START(0x20100) END(0x201ff) WIDTH(48) }
 mem_pmda { TYPE(PM RAM) START(0x26000) END(0x27fff) WIDTH(32) }
 mem_dmda { TYPE(DM RAM) START(0x2a000) END(0x2bfff) WIDTH(32) }
}
```

Команда *MPMEMORY* определяет смещение физической памяти каждого процессора в памяти моделируемой многопроцессорной системы. После определения символьных имен процессоров и смещений пространств памяти для каждого процессора линкер может использовать эти адреса для многопроцессорной компоновки.

LDF-файл может содержать только одну команду MPMEMORY. Максимально возможное количество процессоров определяется особенностями архитектуры системы. За командой MPMEMORY должны следовать команды PROCESSOR <имя процессора>, содержащие описания команд MEMORY и SECTIONS для каждого процессора.

Синтаксис команды MPMEMORY:

```
MPMEMORY {<описание разделяемого сегмента>
                <описание разделяемого сегмента>
                … }
```

где <описание разделяемого сегмента> представляет собой структуру вида

```
<имя процессора> { START (начальный адрес) }
```

Пример использования команды MPMEMORY:


```
MPMEMORY {
    PSH0 { START (0x80000) }
    PSH1 { START (0x1000000) }
  }
```

Команда *PLIT* позволяет добавлять в LDF-файл таблицы компоновки процедур (Procedure Linkage Table, PLT). Эти команды представляют собой шаблон, на основе которого компоновщик генерирует ассемблерный код всякий раз, когда встречает обращение к функции, размещенной в оверлейной памяти. Эти инструкции стандартным образом обрабатывают вызов функции путем вызова менеджера оверлеев. Команда PLIT имеет глобальную область видимости и размещается в LDF-файле в внутри команд PROCESSOR или SECTIONS.

Если нет необходимости в использовании стандартного обработчика вызовов функций (например, если в программе нет оверлеев или при разработке собственного менеджера оверлеев), не следует вообще включать команду PLIT в LDF-файл. В противном случае, для каждой оверлейной подпрограммы компоновщик построит и сохранит список PLIT-таблиц, согласно шаблону. Более подробное описание команды PLIT и принципов генерации и функционирования менеджера оверлеев приведено в документации по утилитам VisualDSP++.

Команда *PROCESSOR* определяет процессор и связанные с ним данные для компоновщика. Команда PROCESSOR содержит команды MEMORY, SECTIONS, RESOLVE и другие команды, описывающие данный процессор. Количество команд PROCESSOR в LDF-файле ограничено только особенностями архитектуры используемых процессоров.

Синтаксис команды PROCESSOR:

```
PROCESSOR <имя процессора> {
        OUTPUT (<file_name>.dxe)
        [MEMORY {команды описания сегментов}]
        [PLIT {команды описания plit-кода}]
        SECTIONS {команды описания секций}
        RESOLVE {symbol_name, project_file}
    }
```

Здесь символьное <имя процессора> задает условное имя процессора в системе, команда OUTPUT определяет имя исполняемого файла с расширением .dxe, в который будет помещен исполняемый код для данного процессора (команда OUTPUT должна быть записана до команды SECTIONS для той же области видимости). Команда MEMORY описывает сегменты памяти, относящиеся к данному процессору. Команда PLIT содержит описание таблицы компоновки процедур вызова оверлеев для данного процессора. Команда SECTIONS используется для задания секций в указанном исполняемом файле. Использование команды RESOLVE позволяет переопределить порядок поиска определенных переменных и меток. Эта команда заставляет компоновщик игнорировать команду LINK_AGAINST для указанного символа *symbol_name* (для всех остальных символов LINK_AGAINST по-прежнему будет выполнена). RESOLVE манипулирует символами и указывает компоновщику на необходимость разрешения данного символа по адресу, задаваемому вторым аргументом *project_file*, который может быть абсолютным адресом или

исполнительным файлом (.dxe или .sm), содержащим определение данного символа. Если компоновщик не находит заданный символ в исполняемом файле, то он сигнализирует об ошибке. Пример использования команды PROCESSOR для описания многопроцессорной системы:

```
PROCESSOR P0{
 MEMORY  {
   rsti { TYPE(PM RAM) START(0x020000) END(0x0200ff) WIDTH(48) }
   dmd1 { TYPE(DM RAM) START(0x30000) END(0x30063) WIDTH(32) }
        }
 OUTPUT($COMMAND_LINE_OUTPUT_DIRECTORY\P1.dxe)
 SECTIONS{
  osmp1{ INPUT_SECTIONS(lport1.doj(pm_irq_svc1))} >rsti
  osmd1{ INPUT_SECTIONS(lport1.doj(dm_dat1))} >dmd1
        }
}
PROCESSOR P1{
 MEMORY  {
   rsti { TYPE(PM RAM) START(0x020000) END(0x0200ff) WIDTH(48) }
   dmd2 { TYPE(DM RAM) START(0x30000) END(0x30063) WIDTH(32) }
        }
 OUTPUT($COMMAND_LINE_OUTPUT_DIRECTORY\P2.dxe)
 SECTIONS{
  osmp2{ INPUT_SECTIONS(lport2.doj(pm_irq_svc2))} >rsti
  osmd2{ INPUT_SECTIONS(lport2.doj(dm_dat2))} >dmd1
        }
}
```

Команда *RESOLVE* (*symbol_name, resolver*) заставляет компоновщик игнорировать команду LINK_AGAINST для указанного символа *symbol_name* (для всех остальных символов LINK_AGAINST по-прежнему будет выполнена). В качестве параметра *resolver* указывается абсолютный адрес или имя исполняемого файла (.dxe или .sm), в котором следует искать описание символьного имени (имени переменной или метки). Если компоновщик не находит заданный символ в исполняемом файле, то он сигнализирует об ошибке.

Команда *SEARCH_DIR ()* определяет один или несколько директориев, в которых ведется поиск входных файлов для компоновщика. Если указаны несколько каталогов, то они должны быть разделены точкой с запятой. Имена директориев должны быть заключены в двойные кавычки. Порядок поиска соответствует порядку перечисления каталогов. Команда действует, начиная с того места в LDF-файле, где она описана. Пример использования команды:

```
SEARCH_DIR( $ADI_DSP\21k\lib; $ADI_DSP\21k\include )
```

Здесь $ADI_DSP – встроенный макрос, возвращающий имя директория, в который установлен VisualDSP++.

Команда *SECTIONS* определяет положение секций программы в памяти, используя при этом сегменты, описанные в команде MEMORY. LDF-файл может содержать команды SECTIONS внутри каждой команды PROCESSOR и SHARED_MEMORY, однако команда SECTION должна следовать после команды MEMORY, описывающей сегменты, в которые компоновщик поместит секции программы.

Синтаксис команды SECTIONS:

```
SECTIONS {операторы секции}
```


где

```
операторы секции { выражение
                 { имя секции [тип секции] {команды секции} [> сегмент памяти]
```

Имя секции начинается с буквы, подчеркивания или точки и не должно совпадать с каким-либо ключевым словом компоновщика. Специальное имя секции .plit соответствует PLT-секции, генерируемой компоновщиком при размещении символов в оверлейной памяти. Эта секция должна быть размещена вне оверлейной памяти. Тип секции является необязательным параметром и используется только в том случае, когда необходимо указать, что секция содержит неинициализированные данные (SHT_NOBITS). Сегмент памяти определяет, размещается ли данная секция в каком-либо определенном сегменте памяти. Некоторые секции (например, используемые при отладке) не обязательно должны присутствовать в исполняемом файле, однако они необходимы некоторым утилитам среды разработки, читающим этот исполняемый файл. Не задавая сегмент памяти, мы указываем компоновщику на необходимость включения его в исполняемый модуль, с отметкой "ненужный" с тем, чтобы в последующем его можно было исключить из окончательного варианта карты памяти системы. В качестве команд секции могут использоваться в любых сочетаниях следующие команды:

```
команды секции { INPUT_SECTIONS (исходн.файл [модуль в архиве (входн.метки)])
               { FILL (шестнадцатеричное число)
               { PLIT (plit-команды)
               { OVERLAY_INPUT(команды оверлея) [ >оверлейный сегмент памяти]
               { Выражение
```

Команда INPUT_SECTIONS() в данном случае определяет исходные объектные или библиотечные файлы (для этого может использоваться LDF-макрос, содержащий список таких файлов, например, макрос $COMMAND_LINE_OBJECTS). Длинные имена записываются в кавычках. Имя архивного модуля в квадратных скобках указывается только в том случае, если первым аргументом является имя библиотеки. Входные метки должны содержаться в программе на языке ассемблера. Именно эти входные метки и пытается найти компоновщик в указанных объектных и библиотечных модулях.

Команда FILL используется для заполнения "пустого" пространства, которое может образоваться вследствие выравнивания текущего счетчика адреса. По умолчанию компоновщик заполняет такие "пустоты" нулями.

Команда PLIT внутри команды SECTIONS определяет PLT-таблицу с локальной (внутрикомандной) областью видимости.

Команда OVERLAY_INPUT идентифицирует те части программы, которые должны быть размещены в исполняемом оверлее командами оверлея, имеющими следующий синтаксис:

```
команды оверлея { OVERLAY_OUTPUT(имя_файла.ovl)
                  INPUT_SECTIONS(команды описания входной секции)
                  символ = OVERLAY_ID()
                  ALGORITHM(ALL_FIT)
                  SIZE(выражение)
```

Раздел "команды оверлея" должен содержать как минимум одну из возможных указанных команд. Имя сегмента памяти определяет, должна ли секция размещаться в оверлейном сегменте. По аналогии с "обычными" (неоверлейными) секциями не все оверлейные секции должны быть включены в образ памяти окончательного исполняемого файла, однако могут понадобиться для работы отдельных утилит среды разработки.

Команда OVERLAY_INPUT указывает компоновщику, на необходимость генерации соответствующего выходного оверлейного файла (с расширением .ovl). Необходимо, чтобы команды OVERLAY_OUTPUT была записана раньше команды INPUT_SECTION внутри команды OVERLAY_INPUT.

Синтаксис команды INPUT_SECTIONS аналогичен случаю, когда она используется в качестве самостоятельной команды, за исключением того, что .plit-секция не может быть помещена в оверлейную память.

Команда OVERLAY_ID возвращает идентификатор оверлея.

Команда ALGORITHM задает алгоритм компоновки оверлеев, по которому компоновщик пытается поместить все описания OVERLAY_INPUT в один оверлей, загружаемый в сегмент памяти выходной секции.

Команда SIZE позволяет задать верхний предел объема памяти, который может быть занят оверлеем.

Команда *SHARED_MEMORY* позволяет получить выходные исполняемые файлы (с расширением .sm), которые будут размещены в разделяемой памяти многопроцессорной системы. LDF-файл может содержать произвольное количество команд SHARED_MEMORY, однако количество процессоров, которые могут получить доступ к этой памяти ограничено (например, для SHARC-процессоров ADSP-2106x количество процессоров не может превышать шести). Команда SHARED_MEMORY должна иметь ту же область видимости, что и команды MEMORY и PROCESSOR, соответственно описывающие и использующие эту разделяемую память.

Синтаксис команды SHARED_MEMORY:

```
SHARED_MEMORY {
        OUTPUT (<file_name>.sm)
        SECTIONS {команды описания секций}
            }
```

Команда OUTPUT() задает имя выходного файла. Команда SECTIONS определяет секции для размещения в разделяемую память.

Пример использования команды SHARED_MEMORY при разработке LDF-файла для многопроцессорной системы:


```
MEMORY  {
 mem_rth  { TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48) }
 mem_pmco { TYPE(PM RAM) START(0x00040100) END(0x000401ff) WIDTH(48) }
 ...
       }
MPMEMORY {
 ID1 { START(0x00100000) }
 ID2 { START(0x00200000) }
       }
SHARED_MEMORY {
 OUTPUT(shared.sm)
 SECTIONS {
   dxe_dmex { INPUT_SECTIONS(shared.doj(sram_da)) } > mem_dmex
          }
              }

PROCESSOR ID1 {
 LINK_AGAINST(shared.sm, ID2.dxe)
 OUTPUT(ID1.dxe)
 SECTIONS {
   dxe_rth { INPUT_SECTIONS (ID1.doj(seg.rth) $LIBRARIES(seg_rth))
           }  > mem_rth
          }
 ...
}

PROCESSOR ID2 {
 LINK_AGAINST(shared.sm, ID1.dxe)
 OUTPUT(ID2.dxe)
 SECTIONS {
   dxe_rth { INPUT_SECTIONS (ID2.doj(seg.rth) $LIBRARIES(seg_rth))
           }  > mem_rth
          }
 ...
}
```

5.4. Утилита Expert Linker

5.4.1. Назначение и основные функциональные возможности Expert Linker

Начиная с версии VisualDSP++ 3.0, в состав интегрированной среды входит утилита Expert Linker, представляющая собой графическое средство для упрощения манипуляций с картой памяти, размещения программного кода и данных, создания оверлеев и разделяемой памяти. Expert Linker отображает в графическом виде информацию, содержащуюся в LDF-файле: описание целевой архитектуры, объектные файлы, LDF-макросы, подключаемые библиотеки и их размещение в пространстве физической памяти системы.

При открытии проекта, уже имеющего файл описания архитектуры, Expert Linker анализирует LDF-файл и строит для него графическое представление карты памяти и отображения объектов. Модифицировать описание архитектуры системы можно как через графический интерфейс Expert Linker, так и непосредственно путем изменения LDF-файла в текстовом редакторе.

Для создания нового LDF-файла можно воспользоваться мастером, генерирующим стандартный LDF-файл с учетом типа и количества процессоров в системе, а также используемого языка программирования.

5.4.2. Интерфейс Expert Linker

Окно утилиты Expert Linker состоит из двух частей: Входные секции (Input Sections) и Карта памяти (Memory Map). Левая панель содержит информацию о входных секциях, находящихся в объектных файлах и файлах библиотек. Правая панель отображает карту физической памяти в виде графического представления (рис.30) или древовидного списка (рис.31).

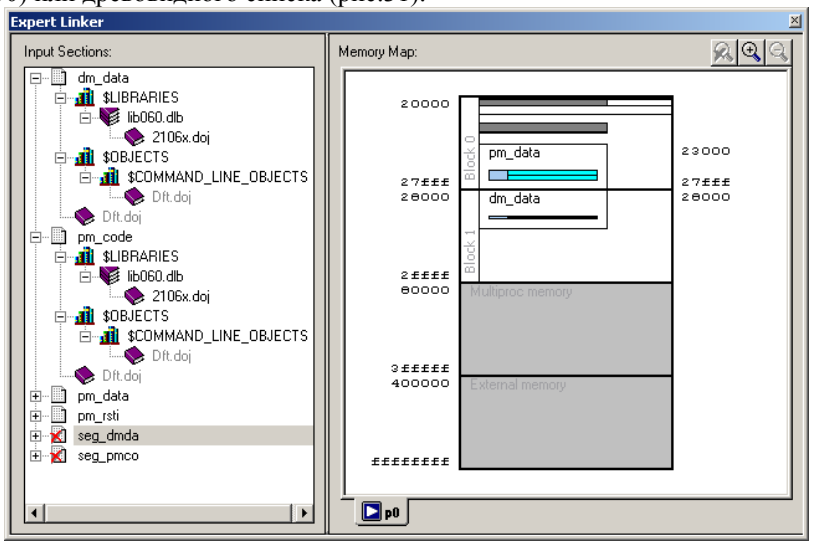


Рис.30. Окно утилиты Expert Linker

Панель Input Sections автоматически формируется компоновщиком на основании включенных в проект файлов и библиотек. Если необходимо добавить к проекту объектный или библиотечный файл, LDF-макрос или входную секцию, то это можно сделать с использованием контекстного меню.

Панель Memory Map позволяет с использованием контекстного меню задавать расположение физических сегментов в памяти каждого процессора.

Если в LDF-файле уже заданы расположения, размеры и типы физических сегментов памяти, в соответствии с командами LDF-файла, компоновщик читает входные секции (логические сегменты) из файлов с расширением *.doj, файлов библиотек с расширением *.ldb и макрокоманды LDF-файла и размещает их в указанные выходные секции (физические сегменты) исполняемого файла. В каждой выходной секции может размещаться одна или несколько входных секций

(один физический сегмент может содержать один или несколько логических сегментов).

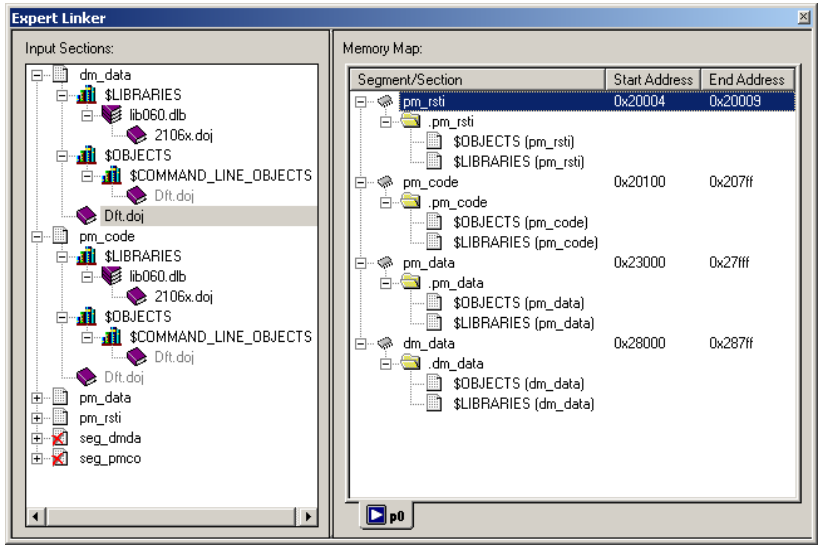


Рис.31. Отображение карты памяти в виде иерархического дерева

Использование собственных пиктограмм для каждого типа и состояния объекта LDF-файла (например, не отображенные в физическую память элементы левой панели помечаются крестиком) и различных цветовых фонов для генерации элементов карты памяти существенно упрощают процесс создания LDF-файла и делают распределение памяти более наглядным.

Связывание входных и выходных секций (логических и физических сегментов) в Expert Linker может выполняться визуально в интерактивном режиме с использованием технологии drag-and-drop. Для размещения объектного кода из входной секции в физической памяти системы достаточно перетащить пиктограмму объекта из панели Input Sections на нужную выходную секцию или сегмент в панели Memory Map.

Expert Linker позволяет посмотреть карту памяти в двух режимах: до компоновки (Pre-Link) и после компоновки (Post-Link). Первый режим используется для размещения входных секций. Второй дает возможность проверить фактическое размещение входных секций в памяти системы после компоновки. К тому же в режиме Post-Link доступна информация о фактическом размере и содержимом каждой секции. Естественно, что пока не выполнена генерация исполняемого модуля для проекта, доступен только режим Pre-Link. Переключение между режимами осуществляется с помощью контекстного меню View.

При создании нового LDF-файла может использоваться стандартный мастер

(Wizard), который генерирует некоторый стандартный вариант LDF-файла, который затем по необходимости можно изменять.

5.4.3. Пример создания LDF-файла

Процесс создания LDF-файла в "ручном" режиме рассмотрим на примере простой программы, вычисляющей поэлементное произведение двух векторов. Текст программы на языке ассемблера приведен ниже:

```
#include "def21060.h"
#define N 64

.SECTION/DM     dm_data;
.VAR input1[N]= "array1.dat";   /* элементы первого вектора */
.VAR output1[N];                /* элементы вектора-результата */

.SECTION/PM     pm_data;
.VAR input2[N]= "array2.dat"; /* элементы второго вектора */

.SECTION/PM     pm_rsti;      /* обработчик прерывания по сбросу */
               NOP;
               JUMP start;

.SECTION/PM     pm_code;       /* "полезный" программный код */
start:  M0=1;
               M9=1;
               M4=1;
               B0=input1;
               CALL dot (DB);  /* вызов функции перемножения векторов */
                  B9=input2;
                  B4=output1;
end:          IDLE;

/*------- Функция Dot - поэлементное перемножение векторов --------*/
dot:
       F0=DM(I0,M0), F5=PM(I9,M9);
       LCNTR=N, DO outer UNTIL LCE;
              F12=F0*F5,  F0=DM(I0,M0), F5=PM(I9,M9);
outer:        DM(I4,M4)=F12;
       RTS;
```

В качестве стартового шаблона LDF-файла будем использовать следующий вариант, отображаемый в окне Expert Linker, как:


```
ARCHITECTURE(ADSP-21060)

SEARCH_DIR( $ADI_DSP\21k\lib )

$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY
{

}

PROCESSOR P0
{

OUTPUT($COMMAND_LINE_OUTPUT_FILE)
  SECTIONS
    {

    }
}
```

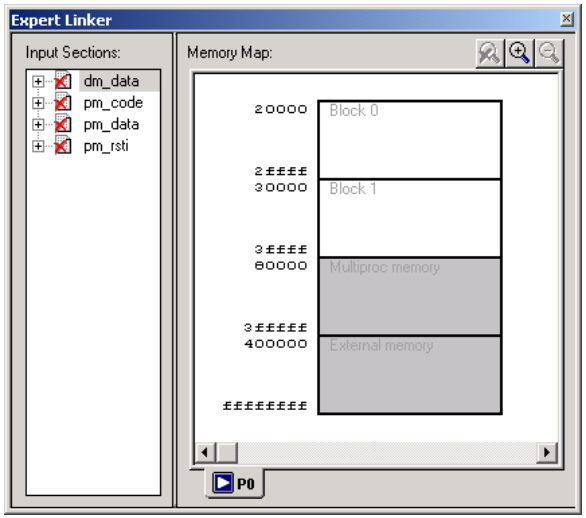


Рис.32. Начальный вариант LDF-файла проекта

Следует обратить внимание, что появление в левой части окна входных секций dm_data, pm_code, pm_data и pm_rsti (описанных в asm-файле) связано с включением в LDF-файл макрокоманды

```
$OBJECTS = $COMMAND_LINE_OBJECTS
```

которая выбирает имена всех секций во входных (объектных) файлах, передаваемых в командной строке. Зачеркнутые значки показывают, что входные секции не имеют отображения в память системы.

Создание полноценного LDF-файла включает в себя несколько основных шагов:

- создание сегментов памяти;
- создание выходных секций в сегментах памяти;
- проецирование (отображение, mapping) входных секций на выходные секции;
- проверка результатов в режиме Post-Link.

Шаг 1. Создание сегментов памяти.

Выбор пункта New в контекстном меню позволяет добавить сегмент памяти, описав его местоположение в памяти (рис.33).

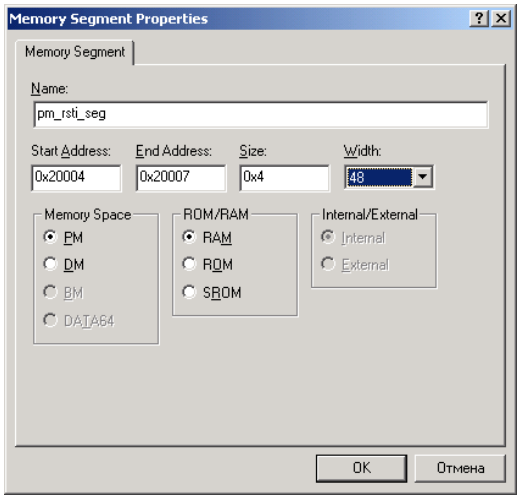


Рис.33. Окно описания сегмента памяти

После определения всех необходимых сегментов карта памяти выглядит следующим образом (рис.34):

```
…
// изменение команды MEMORY
MEMORY
{
pm_rsti_seg {TYPE(PM RAM)
      START(0x20004)
END(0x20007)
      WIDTH(48) }
PM_CODE { TYPE(PM RAM)
      START(0X20100) END(0X201FF)
     WIDTH(48)}
pm_data { TYPE(PM RAM)
     START(0x23000) END(0x23200)
     WIDTH(32)}
dm_data { TYPE(DM RAM)
    START(0x30000) END(0x30100)
    WIDTH(32)}
}

…
```

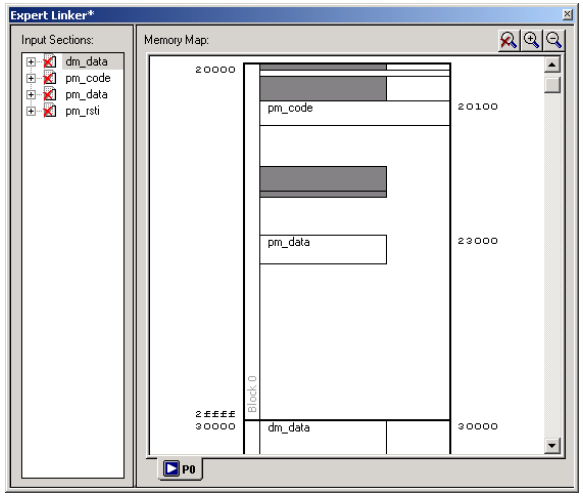


Рис.34. Карта памяти и соответствующий фрагмент LDF-файла после создания сегментов памяти

Серым цветом на карте памяти помечается неиспользуемое пространство. Ширина сегмента соответствует его разрядности (16, 32 или 48 бит).

Шаг 2. Создание выходных секций в сегментах памяти.

Вызвав контекстное меню для требуемого сегмента памяти, можно добавить в

этот сегмент новую выходную секцию. При этом на экране появится окно, в котором необходимо будет указать параметры этой секции, необходимость упаковки и выравнивания границ секции в памяти (рис.35).

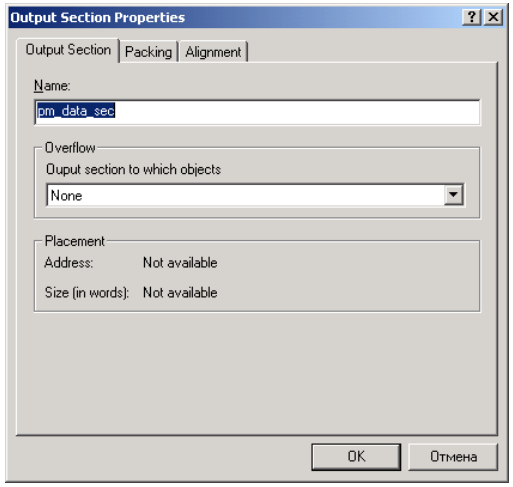


Рис.35. Параметры выходной секции

Например, после создания секции pm_code_sec она отображается внутри того сегмента памяти, в который будут занесены ее данные (рис.36).

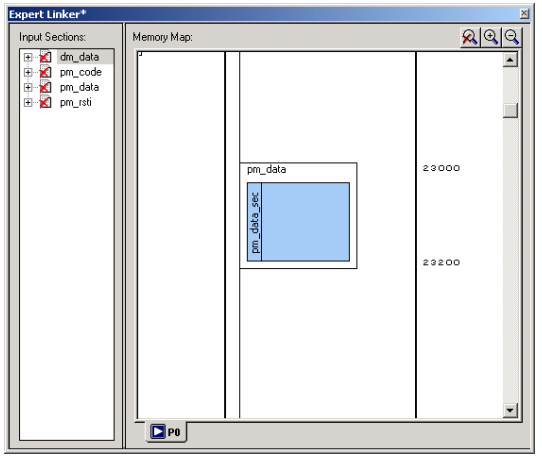


Рис.36. Выходная секция внутри сегмента памяти

Выберем простейший вариант, когда в каждом сегменте памяти будет храниться только одна секция. Тогда после создания всех выходных секций

команда SECTIONS будет содержать 4 выходных секции ("правила отображения") и выглядеть примерно следующим образом:

```
SECTIONS    {
     pm_rsti_sec { … } >pm_rsti_seg
     pm_code_sec { … } >pm_code
     pm_data_sec { … } >pm_data
     dm_data_sec { … } >dm_data
            }
```

Шаг 3. Проецирование входных секций на выходные секции.

Проецирование входных секций на выходные осуществляется простым перетаскиванием объекта входной секции (не имени секции) из левой части окна Expert Linker в правую на нужную выходную секцию. Например, после отображения входной секции pm_rsti на выходную секцию pm_rsti_sec окно Expert Linker будет иметь вид (рис.37):

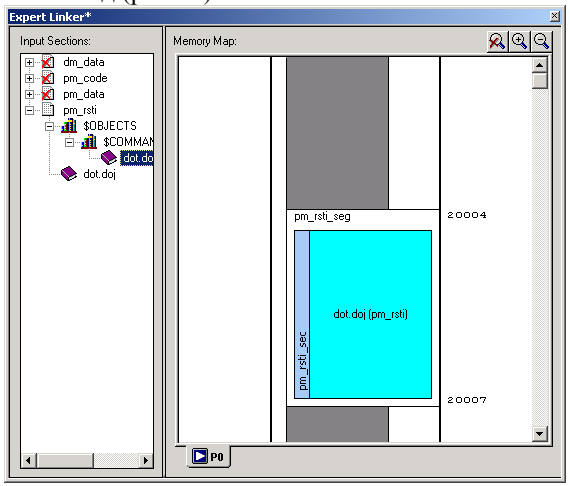


Рис.37. Отображение входной секции pm_rsti на выходную pm_rsti_sec

Результат отображения всех входных секций на выходные выглядит так, как показано на рис.38.

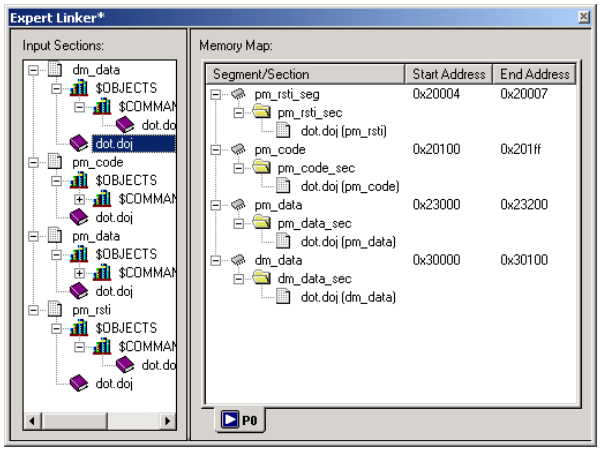


Рис.38. Итоговый LDF-файл и его представление в виде дерева сегментов и секций

```
ARCHITECTURE(ADSP-21060)
SEARCH_DIR( $ADI_DSP\21k\lib )
$OBJECTS = $COMMAND_LINE_OBJECTS;
MEMORY {
 pm_rsti_seg { TYPE(PM RAM) START(0x20004) END(0x20007) WIDTH(48)}
 pm_code     { TYPE (PM RAM) START (0x20100)  END(0x201ff) WIDTH(48)}
 pm_data     { TYPE(PM RAM) START(0x23000) END(0x23200) WIDTH(32)}
 dm_data     { TYPE(DM RAM) START(0x30000) END(0x30100) WIDTH(32)}
}
PROCESSOR P0 {
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
  SECTIONS   {
       pm_rsti_sec { INPUT_SECTIONS(dot.doj(pm_rsti))} >pm_rsti_seg
       pm_code_sec { INPUT_SECTIONS(dot.doj(pm_code))} >pm_code
       pm_data_sec { INPUT_SECTIONS(dot.doj(pm_data))} >pm_data
       dm_data_sec { INPUT_SECTIONS(dot.doj(dm_data))} >dm_data
    }
}
```

Шаг 4. Проверка результатов в режиме Post-Link.

Для проверки правильности размещения программного кода и данных в памяти системы необходимо откомпилировать LDF-файл (или выполнить генерацию исполняемого файла для всего проекта) и переключиться в режим Post-Link. На рис.39 видно, что сегмент памяти, выделенный для хранения входной секции pm_rsti из объектного модуля dot.doj оказался велик: часть этого сегмента не используется.

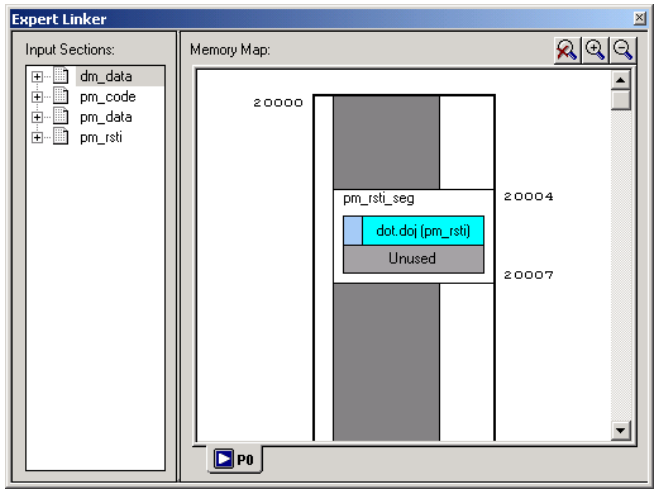


Рис.39. Фрагмент карты памяти в режиме Post-Link

В этом режиме также можно посмотреть фактическое содержимое каждой секции и список имен переменных, меток, секций размещенных в выходной секции или сегменте памяти (рис.40).

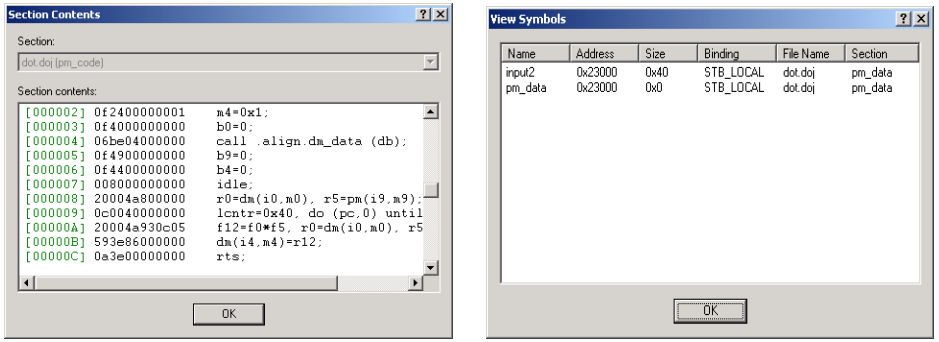


Рис.40. Содержимое секции

Контрольные вопросы

1. Объекты каких типов отображаются в левом и правом окнах утилиты Expert Linker?
2. Какие параметры системы с DSP-процессором можно задавать в окне Global Properties?
3. Зачем в шаблонных LDF-файлах при указании места расположения данных в объектном коде указываются не только сформированный файл, но и файлы стандартных библиотек VisualDSP++?

СОДЕРЖАНИЕ

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Солонина А., Улахович Д., Яковлев Л. Алгоритмы и процессоры цифровой обработки сигналов. СПб:БХВ-Петербург, 2001.
2. Сергиенко А.Б. Цифровая обработки сигналов. Учебник для вузов. СПб.:Питер, 2002.
3. Куприянов М.С., Матюшкин Б.Д. Цифровая обработка сигналов: процессоры, алгоритмы, средства проектирования. СПб.: Политехника, 1998.
4. Гончаров Ю. Технология разработки ExpressDSP //Серия статей в журнале ChipNews за 2000.
5. Ануфриев И.Е. Самоучитель MatLab 5.3/6.x. СПб.: БХВ-Петербург, 2003.
6. Мак-Кракен, Дорн У. Численные методы и программирование на Фортране. М.: Мир, 1977.
7. Bateman A., Peterson-Stephens I. The DSP Handbook. Prentice Hall, 2002.
8. Marvin C., Ewers G. A Simple Approach to Digital Signal Processing. John Wiley and Sons Inc., New York, 1996.
9. Tomarakos J. Advanced digital audio demands larger word widths in data converters and DSPs (Part 1 of 3). – www.chipcenter.com.
10. Tomarakos J. Consider a 24- or 32-bit DSP to Improve Digital Audio (Part 2 of 3). – www.chipcenter.com.
11. Tomarakos J. A 32-bit DSP ensures no impairment of 16-bit audio quality (Part 3 of 3) – www.chipcenter.com.
12. Tomarakos J. The Relationship of Data Word Size to Dynamic Range and Signal Quality in Digital Audio Processing Applications. – DSP Field Applications, Analog Devices Inc., Norwood, MA.
13. VisualDSP++ 3.0 User's Guide for SHARC DSPs. – Analog Devices Inc., Norwood, MA, 2003.
14. VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs. – Analog Devices Inc., Norwood, MA, 2003.
15. VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs. – Analog Devices Inc., Norwood, MA, 2003.
16. Code Composer Studio Getting Started Guide. – Texas Instruments Inc., 2001.

Хусаинов Наиль Шавкятович

Учебно-методическое пособие
по курсу
АРХИТЕКТУРА И ПРОГРАММИРОВАНИЕ
СИГНАЛЬНЫХ ПРОЦЕССОРОВ

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРОЦЕССОРОВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

Часть 1

Для студентов специальностей 220400, 351500

Ответственный за выпуск Хусаинов Н.Ш.
Редактор Лунева Н.И.
Корректор Селезнева Л.И., Чиканенко Л.В.

ЛР № 020565 от 23.06.1997г. Подписано к печати . .03г.
Формат 60x84 1/16 Бумага офсетная.
Офсетная печать. Усл.п.л. – 5,0. Уч.-изд. л. – 4,8.
Заказ № Тираж 100 экз.

"С"

Издательство Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Некрасовский, 44
Типография Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Энгельса, 1