МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Н.Э.БАУМАНА
Кафедра «Компьютерные системы и сети»

УТВЕРЖДАЮ
Заведующий кафедрой ИУ-6

_____В.В. Сюзев
«___» _____ 2011 г.

Романовский А.С., Хохлов С.А.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению лабораторной работы
по курсу «Системы реального времени»

«Изучение инструментальной среды разработки Code Composer Studio»

Москва 2011

Цель работы: знакомство со средой разработки программ для ЦСП Code Composer Studio фирмы Texas Instruments Inc., в ходе которого модернизируется готовый проект.

Продолжительность лабораторной работы – 13 часов.

Продолжительность самостоятельной подготовки – 16 часов.

Содержание работы: в соответствие с выданным заданием модифицируется текст несложной программы на языке С, добавляется в проект модифицированный файл, производится компиляция и запуск программы.

Форма представления результатов: демонстрируется работа и текст программы.

Приобретаемые навыки: запуск среды Code Composer Studio; редактирование текста программы; подгрузка используемых библиотек; сохранение, компиляция и запуск пользовательской программы.

1 Среда разработки Code Composer Studio

1.1 Описание среды разработки

Для работы с сигнальными процессорами, в том числе и с TMS320C6711, компанией TI разработана инструментальная среда Code Composer Studio (CCS). CCS — это интегрированный функционально законченный продукт, включающий в себя всё необходимое для редактирования, компиляции, отладки и анализа программ. CCS имеет удобный реконфигурируемый пользовательский интерфейс с удобными средствами редактирования кода и развитой системой контекстной помощи, в которую автоматически включается система команд отлаживаемого ЦСП. Для удобства ведения сложных разработок в CCS входят графические средства конфигурирования и ведения проектов.

1.2 Конфигурация аппаратной части

Первым шагом работы с CCS является задание конфигурации аппаратной части отлаживаемой ЦСП системы (в литературе ещё употребляется термин "целевая система" - target system).

Существует множество вариантов подключения отлаживаемого ЦСП - как вариантов внутрисхемных эмуляторов, через которые подключается ЦСП, так и вариантов подключения самих процессоров к конкретному эмулятору.

Кроме того, каждый из типов внутрисхемных эмуляторов имеет свои параметры конфигурации и свои функциональные возможности.

Идёт непрерывный процесс совершенствования как самих ЦСП и встроенных в них средств внутрисхемной эмуляции, так и самих внутрисхемных эмуляторов. Изменяются и интерфейсы подключения их к компьютеру.

Внутрисхемные эмуляторы производятся многими фирмами в рамках программы третьих поставщиков. При этом разные модели даже в пределах одной линейки фирмы-производителя, имеющие один и тот же интерфейс подключения, могут отличаться по своим функциональным возможностям, яркий пример тому - изделия фирмы Spectrum Digital.

Учитывать все эти нюансы и максимально использовать

функциональные возможности устройств позволяет принятая в CCS архитектура подключения отлаживаемой подсистемы.

Модульность построения CCS проявляется уже на этом начальном этапе. Основной блок - сервер отладки (target server). Это модуль, который отвечает за обмен со средствами внутрисхемной эмуляции. С одной стороны, к нему подсоединяются все внутренние модули CCS и дополнительные модули пользователя (Plugin), а с другой - совокупность отлаживаемых систем

Для каждого типа внутрисхемного эмулятора поставляется свой драйвер - это программный модуль, имеющий, с одной стороны, стандартный интерфейс для взаимодействия с отладочным сервером, а с другой - интерфейс для подключения соответствующего JTAG-эмулятора. Именно связка "внутрисхемный эмулятор + драйвер" и образует канал между ЦСП и средствами отладки. Соответственно, эта связка в большой степени и определяет параметры, скорость обмена и функциональные возможности средств внутрисхемной эмуляции.

1.3 Менеджер проектов

Итак, мы запустили CCS. Первое открываемое по умолчанию окно и первое понятие, с которым мы сталкиваемся - это проект и, соответственно, система управления проектами - менеджер проектов. Разрабатываемые проекты представляют собой достаточно сложные структуры, которые состоят из целого набора, объектов, таких как библиотеки, файлы заголовков, GEL-скрипты, командные файлы и множество исходных текстов. Число файлов может достигать нескольких десятков. Для удобного управления всем этим конгломератом объектов и его структуризации и служит менеджер проектов, который организует все файлы проекта в разбитое по категориям дерево с удобной навигацией. Добавления и изменение состава проекта производятся простым перемещением файлов методом dragn-drop, в том числе, и из стандартного File Explorer, а двойное нажатие мышкой приводит к открытию или редактированию выбранного файла. Окно менеджера проектов даёт чёткое графическое представление о структуре и проекта и его составных частей.

В составе проекта хранятся и все конфигурационные файлы, включая конфигурацию DSP/BIOS, а также опции средств генерации кода, которые задаются теперь не просто ключами командной строки, а при помощи специальной графической утилиты

1.4 Интегрированный редактор

Для редактирования файлов в CCS используется встроенный многооконный редактор, аналогичный используемому в MS Visual C++, но имеющий специфические необходимые для DSP расширения. Встроенный редактор имеет систему подсветки синтаксиса для всех типов используемых в CCS файлов, включая GEL-скрипты. Написание, редактирование и отладка кода производятся в едином интерфейсе. Удобству работы способствуют такие функциональные возможности как контекстно зависимые меню, вызываемые правой кнопкой мыши, и плавающие конфигурируемые панели инструментов.

1.5 Встроенный язык скриптов GEL

При разумной сложности конфигурирования системы всегда найдётся тот случай, когда нельзя будет чего-либо добиться стандартными средствами. С другой стороны - нельзя бесконечно усложнять конфигурационные возможности.

В CCS встроен язык описания сценариев - GEL (General Extention Language) - в буквальном переводе, язык расширения. Это внутренний язык с С-подобным синтаксисом. Использование языка GEL даёт разработчику возможность описания поведения системы на алгоритмическом уровне с использованием готовых библиотечных элементов. При этом функции языка GEL могут встраиваться практически во все элементы интерфейса и во все функции CCS. По сути это лишь слегка ограниченная возможность дописывать к CCS свои участки кода.

Возможности использования языка расширения GEL образуют гибкую прослойку между элементами CCS и позволяют придать готовому программному продукту гибкость специально написанной для данного конеретного случая программы.

При его помощи можно решать гораздо более сложные задачи,

практически формируя конфигурацию среды разработки:

- создание элементов графического пользовательского интерфейса (GUI) для управления отлаживаемой ЦОС-программой;
- диагностика аппаратной части;
- начальная установка и конфигурирование;
- автоматизация часто выполняемых последовательностей команд;
- добавление пунктов в меню;
- функции работы с отлаживаемым ЦСП: изменение содержимого памяти и регистров, загрузка программы, добавление и удаление точек останова, сброс ЦСП;
- возможности работы с создаваемыми в рамках интерфейса CCS окнами ввода/вывода.

То есть фактически при помощи GEL можно сделать всё то, что можно сделать при помощи пользовательского интерфейса CCS, но только в автоматизированном режиме, что, в сочетании с возможностью добавления дополнений (Plugin), позволяет строить на базе CCS уже практически программируемые тестовые и управляющие комплексы для разрабатываемого устройства.

1.6 Конфигурация объектов

Разработка новой системы на начальном этапе требует для начала работы хотя бы минимального набора сервисов, обеспечивающих возможности планировщика задач, 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 себя 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 CCS предлагается промежуточная модель: статически конфигурируемое ядро реального времени DSP/BIOS, которое даёт базовый набор функций реального времени, таких как переключение нитей, ввод/вывод с малой задержкой и обмен между отлаживаемой

программой и отладчиком. Также в DSP/BIOS входят средства анализа реального времени, которые позволяют реализовывать взаимодействие с ЦСП программой непосредственно во время выполнения без использования точек останова. Полностью DSP/BIOS занимает около 2 К слов памяти и требует менее 1 MIPS производительности. DSP/BIOS построен как полностью статическое ядро. Все его ресурсы конфигурируются перед компиляцией программы.

Для задания конфигурации ресурсов DSP/BIOS используется специальная графическая утилита, с помощью которой конфигурируются и инициализируются все объекты DSP/BIOS.

1.7 Компиляторы CCS

Для компиляции программ в CCS используются оптимизирующие компиляторы ассемблера и Си. Использование языка высокого уровня Си и оптимизирующих компиляторов при написании ЦОС-приложений позволяет сочетать гибкость и скорость написания программы на языке высокого уровня и оптимальность написания ассемблерных кодов. Эффективность выходного кода достигается максимальным использованием ресурсов ЦСП при построении кода с учётом специфики его архитектуры.

Как показывает практика, написание программ для ЦСП на С в современных условиях при использовании оптимизирующих компиляторов, эффективность которых непрерывно повышается, является оптимальным решением по соотношению времени разработки и получаемой эффективности кода. А с учётом всё более жёстких временных рамок и растущей сложности алгоритмов - единственно возможным методом.

Для своих ЦСП фирма TI выпускает свободно доступные DSPLib - библиотеки оптимизированных ассемблерных ЦОС-функций с заголовками для вызова их из С-программ, использование которых существенно повышает оптимальность кода.

1.8 Отладчик CCS

1.8.1 Описание отладчика CSS

В состав CCS входит мощный многооконный отладчик, который имеет ряд особенностей, ориентированных именно на системы ЦОС. Отладочные средства CCS позволяют работать одновременно с несколькими ЦСП как по отдельности, так и интегрируя их в единую систему. Отладчик CCS имеет гибкую систему задания точек останова, вплоть до задания выражения на Си-подобном языке. Уникальной особенностью CCS является наличие точек подключения (Probe Points). Фактически это подключаемый к программе канал ввода/вывода. Точка подключения может быть использована для подачи и снятия сигналов, что совместимо со средствами файлового ввода/вывода и визуализации данных. Использование точек подключения даёт пользователям CCS мощный инструмент анализа и тестирования программы.

Добавление к этому средств анализа реального времени, базирующихся на технологии RTDX, превращает CCS в средство анализа и тестирования в реальном времени.

1.8.2 Точки останова и контроля

Встроенный отладчик CCS имеет три типа точек останова, вернее даже точек контроля и воздействия на выполнение программы, поскольку точки останова только одного типа являются точками останова в чистом виде:

- точки останова (Break Points);
- точки пробники (Probe Points);
- профильные точки (Profile Points).

Точки останова служат для останова программы в требуемом месте. В CCS различаются два типа точек останова: **программные** – по попаданию программы на определённый оператор и **аппаратные** – по выполнению определённой операции на шине, что позволяет как отслеживать последовательность выполнения программы, так и быстро локализовывать, например, такие неприятные ситуации, как порча по непонятным причинам данных в памяти или выход индекса массива за

его пределы. Достаточно просто поставить точку останова на операцию с подозрительной ячейкой ОЗУ. Точка останова может быть ещё и условной. Для такой точки задаётся GEL-выражение, например "gain > 11", в случае выполнения которого происходит останов программы. Такой подход ещё раз иллюстрирует широкие возможности задания поведения отладчика для максимальной адаптации его к конкретным условиям.

Следующим типом точки контроля является **точка пробника** (Probe Points). Если наличие точек останова - это, в принципе, стандартная возможность для любого отладчика, хотя в CCS возможности её применения расширены, то точка пробника - это специальная возможность CCS, ориентированная именно на отладку ЦОС-программ. Фактически точка пробника - это программный аналог щупа осциллографа или логического анализатора, подключаемого к схеме. Точка пробника представляет собой подключаемый к определённой точке программы канал для подачи или снятия данных. Обмен данными с точками пробника происходит прозрачно для исполняемой программы при помощи средств внутрисхемной эмуляции и средств отладчика. Опять же точки пробника могут быть условными и задаваться GEL-выражением.

В отличие от точки останова, при прохождении точки пробника выполнение программы не останавливается. При попадании на точку пробника может быть выполнено множество различных функций: поданы или сняты данные, перерисованы окна визуализации сигнала или выполнен GEL-скрипт.

2 DSP/BIOS

2.1 Задачи DSP/BIOS

Избавление разработчика от рутинных операций подразумевает наличие предназначенных для этого средств, которые стоят между ним и непосредственно ЦСП. К рутинным задачам относятся организация многозадачной обработки, работа с аппаратным обеспечением, средства ввода/вывода и организация памяти, то есть практически полноформатная операционная система реального времени.

DSP/BIOS представляет собой ядро реального времени, которое предоставляет разработчику следующие сервисы:

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

Практически, это оптимизированная для ЦОС-приложений мультизадачная операционная система реального времени.

В отличие от большинства существующих коммерческих ядер, DSP/BIOS поставляется бесплатно как составная часть среды разработчика Code Composer Studio, что весьма немаловажно, с учётом цен на продукты такого класса. При этом DSP/BIOS включает в свой состав оптимизированные для работы с конкретным ЦСП библиотеки и компоненты.

2.2 Цели DSP/BIOS

DSP/BIOS является статически конфигурируемым ядром реального времени со статически определяемой приоритетной моделью исполнения процессов. Такой моделью 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,

достигается оптимизация производительности процессора, поскольку большинство статических вызовов после компиляции упаковываются в несколько команд. Далее, поскольку время выполнения команд ЦСП известно, конфигурация системы задаётся статически и тоже известна, модель исполнения и система приоритетов также задаётся статически, то мы имеем полностью предсказуемое и однозначно определённое поведение системы.

2.3 Компоненты DSP/BIOS

2.3.1 Визуальный редактор компонентов DSP/BIOS

DSP/BIOS - достаточно сложная статически конфигурируемая система, для задания конфигурации которой используется специальная графическая утилита, интегрированная в Code Composer Studio. Утилита конфигурации - это специализированный визуальный редактор, который позволяет выбирать, какие модули DSP/BIOS будут включены в систему, а какие нет, а также задавать их параметры. Все параметры задаются статически до компиляции. При этом утилита конфигурации позволяет оценить объём требуемой под служебные нужды памяти, а также проверить соответствие заданных параметров, что позволяет избежать ошибок уже на начальном этапе конфигурации системы и сэкономить время на старте. Ещё одной функцией утилиты конфигурации является привязка проекта к конкретной аппаратной платформе. Именно в утилите конфигурации задаются параметры карты памяти, распределения прерываний и привязки тактовой частоты процессора к системным часам реального времени. При этом для различных ЦСП существуют уже готовые начальные схемы конфигурации DSP/BIOS.

Все модули ядра реального времени DSP/BIOS могут быть разбиты на 6 групп, каждая их которых предоставляет свой интерфейс (API) пользовательским приложениям:

- группа функций анализа реального времени и обмена данными;
- функции аппаратной абстракции;
- функции ввода/вывода, независимые от аппаратных устройств;
- функции управления выполнением нитей;

- функции взаимодействия и синхронизации нитей;
- прочие функции.

На хост-компьютере пишутся исходные файлы проекта (на C, C++ или ассемблере), использующие интерфейсы DSP/BIOS API. Затем с помощью утилиты конфигурирования определяются те компоненты, которые будут использоваться в проекте, и их параметры. Исходные тексты, библиотека DSP/BIOS и файлы конфигурации образуют проект, из которого средствами генерации кода получается исполняемый файл, загружаемый в ЦСП. Встроенные средства анализа Code Composer Studio позволяют производить мониторинг исполнения программы.

2.3.2 Нити DSP/BIOS

В DSP/BIOS существует четыре типа нитей, которые разбиваются на два класса - **прерывания** и **задачи**. Эти классы отличаются моделью исполнения и видами вызова и завершения.

Первый тип - **аппаратные прерывания**. Они обслуживаются модулем HWI и представляют собой выделенный класс объектов, привязанных к аппаратным источникам прерываний, имеющихся в основных ЦСП-платформах.

Второй тип нитей - **программные прерывания**. Их модель исполнения аналогична аппаратным прерываниям, но программные прерывания не связаны с каким-либо физическим устройством. Их исполнение инициируется программными средствами. Как и аппаратные, программные прерывания выполняются в режиме "исполнение до завершения", но при этом их исполнение может быть приостановлено. Исполнение программных прерываний базируется уже на приоритетной основе.

В DSP/BIOS существует выделенный тип **программных прерываний, управляемых периодическим тактовым сигналом**, используемым для планирования запуска периодических функций, которые должны стартовать регулярно с различной частотой.

Эти функции выполняются модулем PRD. В нём имеются системные часы - 32-разрядный счётчик, изменяющийся каждый раз при вызове функции PRD_tick(). Функция PRD_tick может вызываться как

обработчиком прерывания от системного таймера, который будет управлять системными часами, так и любым другим периодическим процессом, например, тактовой частотой приёма данных. Далее менеджер PRD позволяет разработчику планировать исполнение процессов с различной частотой. Можно создавать несколько PRD-объектов с разным периодом вызова. Поскольку все PRD-объекты управляются одним и тем же системным тактовым сигналом, то период вызова определяется целым числом тактов системного тактового сигнала или вызовов PRD_tick().

В отличие от аппаратных и программных прерываний, рассмотренных выше, **синхронизируемые задачи** в DSP/BIOS могут выполняться, пока они не будут завершены, прерваны нитью с большим приоритетом или не заблокируют своё исполнение до тех пор, пока не произойдёт требуемое событие или не освободится требуемый ресурс. В отличие от задач, прерывания, как аппаратные так и программные, не могут приостановить своё выполнение, освободить ресурсы или перейти в спящее состояние - они должны выполняться до конца или до того момента, когда управление перейдёт к нити с более высоким приоритетом.

Задачи образуют базис традиционной конкурентной обработки с разделением приложения на независимые нити с синхронизацией исполнения семафорами или по системным часам. Задачи могут динамически изменять своё исполнение, используя семафоры или другие средства межзадачной синхронизации.

Работой с задачами в DSP/BIOS занимается модуль TSK. Задачи исполняются в соответствии со своим приоритетом. Имеется 15 уровней приоритета плюс остановленное состояние (отрицательный приоритет).

Нельзя забывать и ещё об одном типе задач - **фоновом процессе**, который имеет минимальный приоритет и выполняется в свободное от отработки других задач время. В очередь на выполнение в фоновом процессе ставятся задачи, которые выполняются, когда больше исполнять нечего, например, задачи ввода/вывода не в реальном времени, а также всевозможные сервисные процессы.

В отличие от аппаратных и программных прерываний, которые конфигурируются статически в утилите конфигурации, задачи могут быть созданы как статически, так и динамически в процессе работы программы.

Каждая задача может находиться в одном из четырёх состояний исполнения:

- выполнение - выполнением данной задачи занимается процессор;
- готовность - задача поставлена в очередь на исполнение и ждёт освобождения процессора;
- блокировка - задача не исполняется, а ожидает, когда произойдёт определённое системное событие;
- удаление - задача удалена и не будет продолжать выполняться снова.

Как было сказано выше, задачи ставятся на выполнение в соответствии с уровнем приоритета.

В отличие от аппаратных и программных прерываний, каждая задача имеет свой конфигурируемый стек. Это очень важное отличие от программных прерываний, которые все исполняются с использованием одного стека, что должно учитываться разработчиками при разбиении приложения на несколько исполняемых нитей. В основном, прерывания используются для критических нитей, а задачи - для нормальной мультизадачной работы. Разделением модели достигается меньший суммарный размер требуемого стека приложения и лучшие условия переключения контекстов, чем при использовании единой модели.

2.3.3 Средства межпроцессорной коммуникации и синхронизации

Кроме возможности обмена потоками данных общего вида, которая будет рассмотрена позже, в DSP/BIOS предусмотрены специальные средства синхронизации процессов и специальные средства быстрого и простого обмена небольшими объёмами данных.

Базовым средством синхронизации задач являются **семафоры**. В DSP/BIOS реализованы счётные семафоры. Счётный семафор хранит внутренний счётчик состояния. Если счётчик отличен от нулевого

значения, то задача при запросе семафора не блокируется. Наличие внутри семафора не логического значения, а счётчика даёт возможность применять различные схемы межпроцессорной синхронизации.

При создании семафор инициализируется определённым значением. Обычно - это количество ресурсов, которые синхронизирует семафор. Операции с семафором включают ожидание (pending) - уменьшение счётчика семафора - и отправку (posting) - увеличение счётчика. SEM_pend() ожидает семафора, а SEM_post() используется для его установления. Если задача ожидает семафора, то вызов SEM_post() переводит её из очереди задач, ожидающих семафора, в очередь задач, имеющих статус "готовность" и стоящих в очереди на исполнение.

Семафор имеет также параметр времени ожидания, который позволяет задаче не ждать события неопределённо долго, а продолжить или завершить своё исполнение через указанное время.

Семафоры используются задачами для внутренней синхронизации. Типичным примером может служить ожидание задачей готовности блока данных от периферийного устройства или синхронизация доступа к общему блоку памяти или к критичным данным другой задачи.

Следующим средством синхронизации является выделенный объект ядра для синхронизации доступа к разделяемым ресурсам - модуль LCK. Средства захвата ресурсов, предоставляемые модулем LCK, позволяют производить арбитраж доступа к разделяемым ресурсам между несколькими конкурирующими задачами. Фактически захват ресурсов - это семафоры с дополнительными функциональными возможностями.

Кроме средств синхронизации исполнения и доступа к ресурсам, имеется и **два типа средств обмена данными**, ориентированных на быстрый обмен малыми объёмами данных, предназначенных не для передачи больших потоков данных, которые обрабатываются алгоритмами, а на передачу небольших сообщений, содержащих управляющую или координирующую информацию.

Первый тип - **почтовые ящики** (Mailboxes). Обычно они используются для обмена сообщениями или данными между разными задачами, исполняющимися на различных уровнях приоритета. Содержание почтового ящика не фиксировано и определяется разработчиком. Задача может положить сообщение в почтовый ящик или ожидать прихода сообщения в него.

Второй тип - **очереди**. В отличие от почтового ящика, очередь может содержать несколько элементов (сообщений), которые могут быть поставлены в очередь или взяты из неё. Все сообщения в очереди обрабатываются по принципу FIFO: первым поставлен в очередь - первым считан из очереди.

2.3.4 Модуль RTA - средства анализа реального времени

Средства анализа реального времени дают разработчикам возможность анализа контроля поведения приложения во время его исполнения. Все эти средства фактически работают через один и тот же JTAG-интерфейс, через который работает и отладчик, используя его как относительно низкоскоростной канал реального времени. Большая часть средств анализа реального времени требует наличия в памяти процессора ядра реального времени DSP/BIOS. Кроме предоставления выполняемых сервисов для средств анализа, ядро DSP/BIOS поддерживает физический канал связи реального времени с хост-компьютером. Кроме того, используя ядро DSP/BIOS для построения приложений с целью использовать его мультизадачные возможности и сервисы ввода/вывода, разработчики автоматически получают инструмент для снятия и передачи в реальном времени информации, которая используется средствами анализа и визуализации среды разработки Code Composer Studio. Предоставляемые модулем RTA сервисы можно разделить на следующие группы:

- Журнал сообщений (Message Event Log) - средства отображения упорядоченной по времени последовательности событий, записываемых в журнал ядра независимыми нитями, что позволяет отслеживать и протоколировать исполнение программы. Приложение может записывать события в журнал через вызовы DSP/BIOS. Также

события могут заноситься в журнал средствами ядра при отслеживании состояния нити.

- Сбор статистики (Statistic Accumulators) - средства отображения статистики исполнения, собираемой во внутреннем аккумуляторе ядра. Статистика отображает динамические параметры процесса выполнения ядра, начиная от простых счётчиков и изменения во времени данных, до вычисления ожидаемого интервала исполнения независимых нитей. Сбор статистики выполняется непосредственно через вызовы DSP/BIOS или же средствами самого ядра, отслеживающими распределение и исполнение нитей, а также выполнение ими операций ввода/вывода.
- Каналы обмена данных с хост-компьютером (Host Data Channels) - средства связи объектов ввода/вывода ядра и файлов - расположены в хост-компьютере и позволяют организовывать между ними стандартные потоки данных. Они используются для тестирования и анализа поведения алгоритмов. Эта группа сервисов также позволяет перехватывать на лету любые другие пересылки потоков данных через модули ядра и отсылать их в хост-компьютер для последующего анализа. Сервер управляющих команд (Host Command Server) - средства контроля трассировки и сбора статистики в программе.

Технология RTDX позволяет производить обмен данными между хост-компьютером и системой на базе ЦСП исполняемому на ЦСП приложению без помех. Этот двунаправленный канал реального времени позволяет как производить сбор и анализ данных хост-компьютером, так и взаимодействовать с исполняемым ЦОС-приложением. Данные, принятые от ЦСП, могут использоваться для анализа и визуализации на хост-компьютере. При этом параметры приложения могут подстраиваться по командам хост-компьютера без остановки приложения. RTDX-каналы позволяют производить полное тестирование алгоритмов при помощи стандартных отладочных средств - подачу тестовых воздействий на ЦОС-приложение и анализ получаемых результатов.

RTDX имеет два уровня - программный и аппаратный. Небольшая RTDX библиотека исполняется на ЦСП. ЦОС-приложение обращается к вызовам этой библиотеки, когда необходимо считать или записать данные в RTDX-канал. Аппаратный уровень образуется средствами внутрисхемной JTAG-эмуляции, которые обеспечивают обмен данных по отладочному интерфейсу между ЦСП и хост-компьютером.

На хост-платформе RTDX библиотека работает интегрированно с Code Composer Studio. Средства анализа и визуализации данных работают с библиотекой RTDX через стандартный интерфейс COM API. Возможна работа с RTDX не только средств Code Composer Studio, но и других стандартных средств анализа и визуализации, работающих через интерфейс COM, например, пакета LabVIEW фирмы National Instruments или электронных таблиц Microsoft Excel.

Хост-библиотека RTDX поддерживает два режима работы - напрерывный и периодический. В непрерывном режиме данные просто накапливаются во внутреннем буфере RTDX библиотеки и не записываются в файл. Этот режим используется, когда надо непрерывно снимать и отображать данные ЦОС-приложения и не надо сохранять их в файл. В периодическом режиме данные записываются в файл. Этот режим используется, когда надо собирать большие объёмы данных и хранить их в файле.

2.3.5 Уровень абстрагирования аппаратного обеспечения

Одной из составляющих DSP/BIOS являются функции для работы с базовыми аппаратными компонентами, независимо от их физического воплощения. Интерфейс абстрагирования аппаратного обеспечения даёт возможность работы с основным аппаратным обеспечением через простой логический интерфейс, независимый от самого устройства. При абстрагировании таких компонентов, как, например, таймер и аппаратные прерывания, существенно облегчается переход от устройства к устройству. Поскольку разные ЦСП имеют разный набор периферии, то в состав DSP/BIOS включена библиотека поддержки конкретного ЦСП - Chip Support Library (CSL), в которой содержатся функции работы со всеми его функциональными модулями.

Ещё одним компонентом абстрагирования от аппаратного обеспечения является система управления распределением памяти. Через конфигурационную утилиту разработчик устанавливает границы, тип и наименования физических блоков памяти, создавая логическую карту памяти.

Входящие в состав DSP/BIOS средства аппаратно-независимого ввода/вывода позволяют унифицировать потоки ввода/вывода и отделить их от конкретного устройства. Потоки данных передаются в реальном времени и базируются как на фреймовой структуре, так и на системных часах. Внутри DSP/BIOS потоки данных используются как для обмена с периферией, так и для обмена данными между нитями. Важной частью модели независимого ввода/вывода являются как драйверы конечных устройств, например кодеков, так и драйверы портов ввода/вывода и контроллеров ПДП, входящие в библиотеку CSL.

2.3.6 Часы реального времени

Модуль CLK позволяет работать с функциями часов и таймеров реального времени, создавая аппаратно-независимую временную базу для всех временно-зависимых процессов обработки и периодических функций. Через утилиту конфигурирования устанавливаются параметры встроенного таймера и задаётся установленная временная база, обычно 1 мс. При этом можно как самому установить регистры таймера, так и воспользоваться встроенным интерфейсом и просто установить нужное время, оставив содержимое регистров на усмотрение средств конфигурации.

Использование такой привязки абстрагирует системные часы от конкретного ЦСП и привязывает их к реальному времени. А поскольку к системным часам привязывается планировщик нитей и планировщик периодических процессов, то и вся система привязывается к реальному времени. Модуль CLK позволяет иметь два вида системных часов - высокого и низкого разрешений. Часы низкого разрешения обычно используются для планировщиков нитей и периодических функций. Часы высокого разрешения - прямая функция от такта ЦСП. Они показывают число, на которое изменился регистр таймера за

время выполнения какой-либо операции. Часы высокого разрешения обычно используются для измерения времени исполнения функций или для измерения временных интервалов.

3 Задание

1. Подключите отладочную плату DSK6711 к ПК.
2. Включите питание отладочной платы.
3. Запустите Code Composer Studio Setup.
4. Проверьте, что My System содержит сведения об эмуляторе для отладочной платы DSK6711. Если таких сведений нет, их нужно добавить вручную из списка доступных плат.
5. Запустите на ПК среду CCS. При необходимости в окне Parallel Debug Manager следует выбрать эмулятор для отладочной платы DSK6711.
6. Загрузите проект <Директория CCS>\examples\ dsk6711\csl\timer\timer2_useBios\timer2_useBios.pjt . В данном примере используются два таймера. По прерываниям от этих таймеров вызывается обработчик аппаратного прерывания, который в свою очередь инициирует исполнение одного из двух программных прерываний. Каждому таймеру соответствует свое программное прерывание.
7. Скомпилируйте и соберите проект (Project->Build). Загрузите .out-файл в память процессора (File->Load Program...). В дальнейшем данные действия следует повторять после любого изменения кода программы.
8. Запустите программу на исполнение.
9. Исследуйте работу программы, устанавливая точки останова в аппаратном и программных прерываниях.
10. Период первого таймера установите равным 100 мс, второго — 1с. В первом обработчике программного прерывания реализуйте имитацию вычислений с помощью задержки (60-70 мс), в другом — включение/выключение светодиода USER_LED1. Исследуйте совместную работу программных прерываний при равных приоритетах с помощью RTA и визуального контроля переключений светодиода.
11. Понизьте приоритет второго программного прерывания. Исследуйте совместную работу программных прерываний при

разных приоритетах с помощью RTA и визуального контроля переключений светодиода.

12. Установите приоритет первого программного прерывания ниже, чем приоритет второго прерывания. Исследуйте совместную работу программных прерываний при разных приоритетах с помощью RTA и визуального контроля переключений светодиода.
13. Реализуйте переключение светодиодов с помощью программных прерываний, управляемых периодическим тактовым сигналом. Для этого необходимо настроить модули CLK и PRD с помощью визуального редактора компонентов DSP/BIOS. Обратите внимание, что оба таймера зарезервированы модулем обработки аппаратных прерываний HWI, поэтому предварительно необходимо «освободить» один из таймеров.
14. Подготовьте отчет по лабораторной работе.

Описание платы DSK6711: <Директория CCS>\docs\hlp\6711DSK.HLP

Описание процессора TMS320C6711:

http://focus.ti.com/docs/prod/folders/print/tms320c6711c.html

4 Текст модифицируемой программы

/* В данном примере используются два таймера. По прерываниям от этих таймеров вызывается обработчик аппаратного прерывания, который в свою очередь инициирует исполнение одного из двух программных прерываний. Каждому таймеру соответствует свое программное прерывание. */

```
#include <std.h>
#include <swi.h>
#include <csl.h>
#include <csl_timer.h>
#include <csl_irq.h>
#include "timer2_useBioscfg.h"

void TimerEventHandler(int Arg);
```

```
static TIMER_Handle hTimer1, hTimer2;
static Uint32 TimerEventId1,TimerEventId2;

static Uint32 TimerControl = TIMER_CTL_RMK(
  TIMER_CTL_INVINP_NO,
  TIMER_CTL_CLKSRC_CPUOVR4,
  TIMER_CTL_CP_PULSE,
  TIMER_CTL_HLD_YES,
  TIMER_CTL_GO_NO,
  TIMER_CTL_PWID_ONE,
  TIMER_CTL_DATOUT_0,
  TIMER_CTL_INVOUT_NO,
  TIMER_CTL_FUNC_GPIO
);

extern far SWI_Obj SwiMain;
extern far SWI_Obj SwiIsr1;
extern far SWI_Obj SwiIsr2;

void main() {
  CSL_init();
  /* post the main SWI */
  SWI_post(&SwiMain);

}

void SwiMainFunc(int arg0, int arg1) {
  hTimer1 = TIMER_open(TIMER_DEVANY, TIMER_OPEN_RESET);
  hTimer2 = TIMER_open(TIMER_DEVANY, TIMER_OPEN_RESET);

  /* Obtain the event IDs for the timer devices */
  TimerEventId1 = TIMER_getEventId(hTimer1);
  TimerEventId2 = TIMER_getEventId(hTimer2);
```

```
  /* Enable the timer events */
  IRQ_enable(TimerEventId1);
  IRQ_enable(TimerEventId2);

  /* Configure the timer devices */
  TIMER_configArgs(hTimer1,
    TimerControl, /* use predefined control value  */
    0x00100000,   /* set period                    */
    0x00000000    /* start count value at zero     */
  );

  TIMER_configArgs(hTimer2,
    TimerControl, /* use predefined control value  */
    0x00080000,   /* set period                    */
    0x00000000    /* start count value at zero     */
  );

  /* Start the timers */
  TIMER_start(hTimer1);
  TIMER_start(hTimer2);
}

void SwiIsr1Func(void)
{
  return;
}

void SwiIsr2Func(void)
{
  return;
}

void TimerEventHandler(int Arg) {
```

```
  static int cnt[2] = {0,0};
  /* process timer event here */
  cnt[Arg]++;
  if (Arg) SWI_post(&SwiIsr2);
  else SWI_post(&SwiIsr1);
}
```

5 Использованные источники информации

1. Конспект лекций по курсу «Системы реального времени»
2. ChipNews /Инженерная микроэлектроника, №4, 2001 г. Ю.Гончаров. Технология eXpressDSP. Часть II. Интегрированная среда разработчика Code Composer Studio.
3. ChipNews /Инженерная микроэлектроника, №2, 2001 г. Ю.Гончаров. Технология разработки eXpressDSP.
4. ChipNews /Инженерная микроэлектроника, №10, 2001 г. Ю.Гончаров. EXpressDSP. Часть IV. Ядро реального времени DSP/BIOS.
5. ChipNews /Инженерная микроэлектроника, №6, 2001 г. Ю.Гончаров. EXpressDSP. Часть III. Ядро реального времени DSP/BIOS.
6. http://focus.ti.com/docs/prod/folders/print/tms320c6711c.html