Окончание. Начало в КиТ № 1`2010

Проектирование с использованием процессоров Analog Devices. Расширенные возможности симулятора

Александр СОТНИКОВ

Компилятор C/C++ процессоров Blackfin

Поскольку создаваемый нами проект будет написан на языке С, целесообразно вначале вкратце рассмотреть основные отличительные черты компилятора С/С++ для процессоров Blackfin. Компилятор С/С++ предназначен для преобразования исходных файлов, написанных на языках С/С++, в исходные файлы на языке ассемблера и представляет собой отдельную утилиту, запускаемую автоматически из IDDE VisualDSP++ или вручную из командной строки ОС.

Он поддерживает стандарт ISO ANSI, однако имеет ряд расширенных возможностей, специфических для процессоров Blackfin, среди которых, например:

- Поддержка встраиваемых ассемблерных вставок при помощи конструкции asm ().
- Поддержка квалификаторов банка памяти ***bank*** («имя_банка»), при помощи которых компилятору можно указать на то, что переменные располагаются в разных банках и обращение к ним может быть выполнено одновременно.
- Поддержка квалификаторов секций памяти ***section*** («имя_секции»), которые указывают компилятору на то, что переменную необходимо разместить в формируемом объектном файле в определенную секцию.
- Поддержка ***прагм #pragma*** — директив, модифицирующих поведение компилятора (например, управляющих выравниванием данных по определенной границе в памяти процессора, указывающих на то, что объявляемая и/или определяемая функция является обработчиком прерывания, и ее адрес должен быть помещен в соответствующий элемент таблицы векторов прерываний, и т. п.).
- Наличие встроенных (***built-in***) функций, которые распознаются и заменяются компилятором на одну или несколько машинных команд, как если бы это были простые операторы типа “+” или “–”. Встроенные функции позволяют более эффективно использовать аппаратные ресурсы процессора и имеют имена, начинающиеся с ***__builtin_***.

Полный перечень расширений стандарта ISO ANSI, реализованных в компиляторе С/С++ процессоров Blackfin, можно найти в документе [1].

Проекты, создаваемые на языке С/С++ для процессоров Blackfin, должны включать в свой состав так называемый исполняемый заголовок С/С++ (CRT, C/C++ run-time header) — код на языке ассемблера, который выполняется после сброса или включения питания процессора. Задача CRT состоит в том, чтобы инициализировать состояние процессора и вызвать функцию _main (код, формируемый компилятором для функций на С/С++; сопровождается меткой, которая состоит из символа подчеркивания и имени функции). Выполнение CRT автоматически гарантирует, что состояние процессора перед вызовом _main подчиняется двоичному интерфейсу прикладных программ (ABI, application binry interface), который обеспечивает переносимость прикладных программ с платформы на платформу.

Код CRT (в последних версиях VisualDSP++ для него используется название startup-код) может быть сконфигурирован в интерактивном режиме и автоматически включен в проект при помощи «мастера» создания проектов. Также пользователь может вручную создать собственный файл с кодом CRT и подключить его на этапе компоновки при помощи соответствующего макроса в файле описания линкера. И, наконец, если пользователь вообще не подключает CRT к проекту, то среда VisualDSP++ автоматически использует на этапе компоновки один из стандартных, предварительно скомпилированных CRT.

Исходный файл на языке ассемблера basiccrt.s, из которого получены стандартные CRT для разных конфигураций и платформ, находится в установочном каталоге VisualDSP++ в папке ***Blackfin\lib\src\libc\crt***.

В остальном, на первый взгляд, процесс программирования на языках С/С++ для процессоров Blackfin не сильно отличается от программирования под другие платформы. Некоторые сложности в освоении у новичков может вызвать разве что организация обработки прерываний, о которой речь пойдет чуть позднее. Однако для написания действительно эффективного и компактного кода нужно хорошо разбираться в архитектуре процессора и различных нюансах работы компилятора. Так, например, поскольку процессор Blackfin не имеет аппаратной поддержки типов с плавающей точкой и 64-битных данных, то для работы с переменными типа ***float***, ***double***, ***long long*** и т. п. используется программная эмуляция, что неизбежно приводит к увеличению объема кода и снижению производительности. Другой пример — это работа с памятью. В языке С используется модель унифицированного адресного пространства, однако на аппаратном уровне доступ к различным областям памяти процессора существенно различается, что может сильно влиять на быстродействие. Подобных примеров можно привести очень много, однако ввиду ограниченного объема вопросы повышения эффективности кода на языках С/С++ для процессоров Blackfin рассматриваться здесь не будут. Заинтересованные читатели могут найти более подробное обсуждение данных вопросов в [1], а также в серии статей [2].

Создание проекта

Рассматриваемый в этой статье проект, как уже отмечалось ранее, будет осуществлять ввод в процессор блока данных через последовательный порт в режиме DMA и вычисление быстрого преобразования Фурье (БПФ). Для упрощения программы мы будем предполагать, что внешнее устройство, которое является источником данных, выдает 32-битные слова, содержащие в старшей половине (биты 31–16) отсчеты синфазной составляющей сигнала, а в младшей половине (биты 15–0) отсчеты квадратурной составляющей сигнала. Данные синфазной и квадратурной составляющих комплексного сигнала будут представлены в знаковом 16-битном формате с фиксированной точкой, который является основным форматом, поддерживаемым процессорами Blackfin. По завершении приема блока из 128 комплексных отсчетов процессор будет генери-

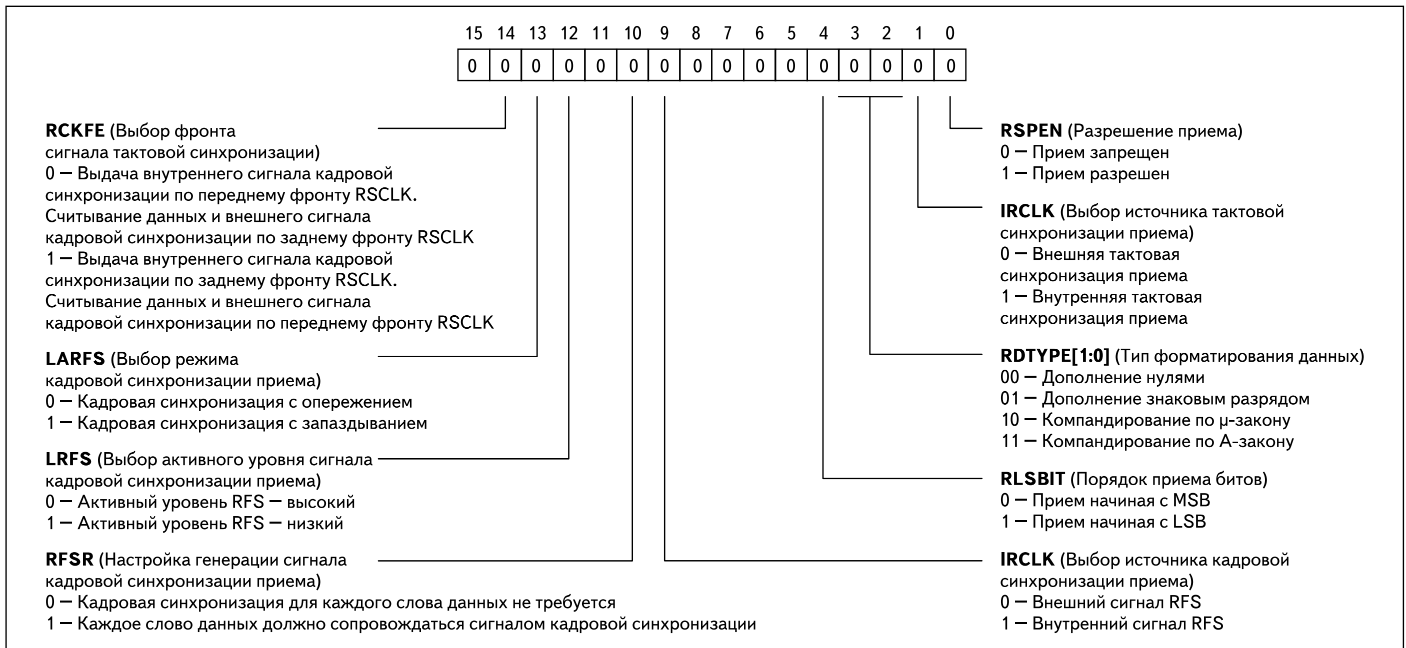

Рис. 1. Назначение полей регистра SPORT0_RCR1

Рис. 2. Назначение полей регистра SPORT0_RCR2

ровать прерывание, которое сигнализирует о готовности массива входных данных для вычисления БПФ.

Для создания нового проекта вызовите пункт меню ***File → New → Project***, выберите в открывшемся окне «мастера» создания проектов тип ***Standard application*** и задайте имя проекта (например, ***TestSport***) и каталог, в котором он будет располагаться. На вкладке ***Select Processor*** укажите процессор ADSP-BF537 и автоматический выбор ревизии кристалла. На вкладках ***Application Settings*** и ***Add Startup Code/LDF*** оставьте предлагаемые варианты настроек без изменений и завершите создание проекта, нажав кнопку ***Finish***. В результате работы «мастера» создания проектов будет сгенерирован соответствующий файл проекта ***TestSport.dpj*** и исходный файл на языке С, ***TestSport.c***, который пока что содержит лишь каркас функции ***main***.

Первое, что необходимо сделать в исходном коде,— это добавить в функцию ***main()*** инициализацию соответствующих аппаратных модулей процессора.

Для настройки последовательного порта (для определенности в проекте мы будем использовать SPORT0) на прием необходимо инициализировать четыре регистра: два регистра управления приемом (SPORT0_RCR1 и SPORT0_RCR2), регистр делителя сигнала тактовой синхронизации приема (SPORT0_RCLKDIV) и регистр делителя кадровой синхронизации приема (SPORT0_RFSDIV).

Регистры управления приемом порта SPORT0 расположены по адресам внутренней памяти 0xFFC00820 и 0xFFC00824, а назначение их полей показано на рис. 1 и 2 соответственно. Для нашего проекта мы выберем режим приема 32-битных данных с внутренней генерацией сигналов кадровой и тактовой синхронизации. Также в регистре SPORT0_RCR1 необходимо установить бит RFSR, который указывает на необходимость генерации сигнала кадровой синхронизации для каждого принимаемого слова. Остальные настройки приема по последовательному порту можно оставить без изменений, поскольку с точки зрения моделирования работы порта в симуляторе они не важны.

Значения частот сигналов тактовой (F_{RSCLK}) и кадровой (F_{RFS}) синхронизации приема определяются при помощи следующих формул:

$$F_{RSCLK} = F_{SCLK}/(2\times(\text{SPORTx_RCLKDIV}+1)),$$

$$F_{RFS} = F_{RSLK}/(\text{SPORTx_RFSDIV}+1),$$

где F_{SCLK} — частота тактового сигнала системы. В реальной системе значения, записываемые в регистры SPORT0_RCLKDIV и SPORT0_RFSDIV, необходимо подбирать с учетом конкретного значения F_{SCLK} и требований внешнего устройства, однако для ускорения работы модели мы зададим максимально возможные тактовые частоты сигналов (SPORT0_RCLKDIV= = 0 и SPORT0_RFSDIV = 31).

Для упрощения доступа к регистрам, отображенным во внутренней памяти процессора Blackfin, из программы на языке C можно подключить файл cdefBF537.h. Этот файл и подключаемый внутри него файл cdefBF532.h содержат определения символьных констант, соответствующих указателям на регистры процессора. Так, например, конструкцию ****(volatile unsigned short*)0xFFC00820***, необходимую для обращения к регистру управления SPORT0_RCR1, при использовании файла cdefBF537.h можно заменить менее громоздкой и более наглядной записью *pSPORT0_RCR1.

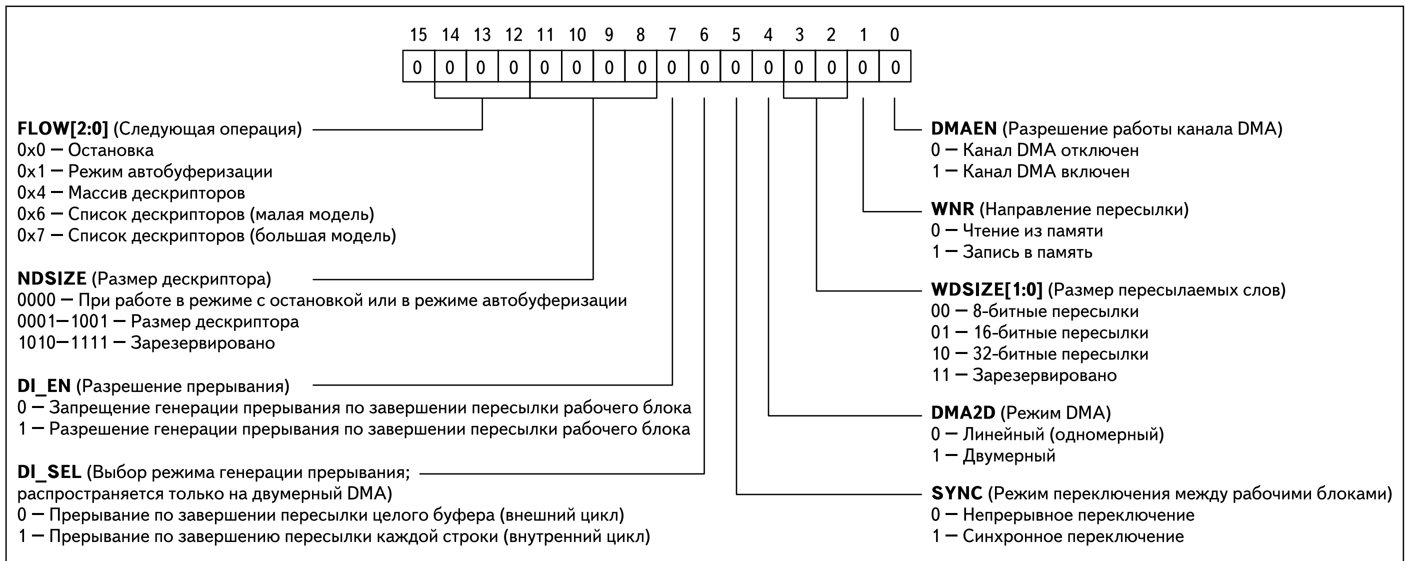

Рис. 3. Назначение полей регистра DMAx_CONFIG

Таким образом, для настройки последовательного порта нам потребуется добавить в исходный файл строки:

```
*pSPORT0_RCR1 = RFSR| IRFS | IRCLK;
*pSPORT0_RCR2 = 0x1F;
*pSPORT0_RCLKDIV=0;
*pSPORT0_RFSDIV=31;
```

Обратите внимание на то, что, пока для последовательного порта не настроен канал DMA и не разрешено соответствующее прерывание, устанавливать в единицу бит RSPEN регистра SPORT0_RCR1 не следует.

По умолчанию за прием данных по последовательному порту SPORT0 в режиме DMA отвечает канал DMA номер 3. Для настройки параметров канала нам потребуется инициализировать содержимое регистров конфигурации (DMA3_CONFIG), начального адреса (DMA3_START_ADDR), счетчика (DMA3_X_COUNT) и модификатора (DMA3_X_MODIFY) канала DMA.

Назначение полей регистра конфигурации канала 3 DMA показано на рис. 3. Для нашего проекта мы выберем одномерный режим DMA с остановкой по завершению пересылки блока данных и генерацией прерывания. Данные в память будут записываться 32-битными словами.

В качестве приемника данных будет использоваться массив данных типа ***complex_fract16*** под названием ***rx_buf*** длиной 128 элементов. Тип данных ***complex_fract16*** представляет собой структуру из двух элементов типа ***fract16*** (знаковое 16-битное число с фиксированной точкой), соответствующих вещественной и мнимой составляющей комплексного числа. Для работы с этими двумя типами данных в исходном файле необходимо подключить заголовочные файлы ***fract.h*** и ***complex.h***. Массив ***rx_buf*** следует объявить глобально с квалификатором ***volatile***:

```
volatile complex_fract16 rx_buf[128];
```

Квалификатор ***volatile*** указывает компилятору на то, что он не должен трогать эту переменную в процессе оптимизации, и наиболее часто используется в следующих случаях:

- для обращения к регистру процессора или внешнему устройству, отображенным в карте памяти;
- для данных, к которым осуществляется доступ в режиме DMA;
- для объектов (например, глобальных переменных), модифицируемых в обработчиках асинхронных прерываний.

В регистр счетчика DMA записывается полное количество пересылаемых слов, а в регистр модификатора — количество байтов в пересылаемом слове, то есть 4. Таким образом, фрагмент кода, отвечающий за инициализацию канала DMA, будет иметь вид:

```
*pDMA3_CONFIG = SYNC| WNR | WDSIZE_32 | FLOW_STOP |
DI_EN;
*pDMA3_X_COUNT = 128;
*pDMA3_X_MODIFY = 4;
*pDMA3_START_ADDR = (int)rx_buf;
```

Опять же, как и в случае инициализации последовательного порта, бит регистра DMA3_CONFIG, отвечающий за активацию канала DMA, мы пока устанавливать не будем.

Последний этап процедуры инициализации аппаратных средств процессора состоит в настройке прерывания, которое будет генерироваться по заполнению блока данных. Прерывание канала DMA номер 3 по умолчанию направляется на вход IVG9 контроллера событий ядра. Эту настройку можно изменить, записав новое значение в поле регистра SIC_IAR0, соответствующее данному прерыванию.

Далее необходимо назначить обработчик прерывания (подпрограмму обслуживания) и зарегистрировать его в таблице векторов прерываний. Для этих целей в библиотеке компилятора C/C++ процессоров Blackfin имеются два механизма. Первый из них — это функция ***register_handler (IVGx, ISR_Name)***, которая ставит в соответствие подпрограмму обслуживания прерывания с именем ISR_Name прерыванию ядра IVGx. Она осуществляет запись адреса функции, обработчика прерывания, в требуемый элемент таблицы векторов прерываний, а также устанавливает в регистре маскирования прерываний ядра (IMASK) бит, соответствующий прерыванию IVGx, и бит глобального разрешения прерываний. В нашем случае вызов функции будет иметь вид:

```
register_handler(ik_ivg9, Sport0_RX_ISR);
```

где ik_ivg9 — это символьная константа, определяющая номер прерывания, а Sport0_RX_ISR — имя функции-обработчика.

Второй механизм — это макрос ***EX_INTERRUPT_HANDLER (ISR_Name)***. Обычно он используется для одновременного объявления и определения обработчика и имеет следующий синтаксис:

```
EX_INTERRUPT_HANDLER(ISR_Name)
{
//Код обработчика прерывания
}
```

Также данный макрос может быть использован для объявления прототипа функции-обработчика:

```
EX_INTERRUPT_HANDLER(ISR_name);
```

Применение макроса ***EX_INTERRUPT_HANDLER*** указывает компилятору на необходимость сохранения контекста процессо-

ра при входе в функцию и восстановления контекста при выходе из нее, а также на то, что последней командой в коде на языке ассемблера, который будет сгенерирован компилятором, должна быть команда возврата из прерывания (RTI;).

Для использования описанных выше механизмов в исходном файле необходимо подключить заголовочный файл ***sys\exception.h***.

И, наконец, последний шаг в настройке прерывания — это написание кода функции-обработчика. В нашем случае он будет включать в себя три операции: снятие бита прерывания канала DMA номер 3 (осуществляется записью единицы в соответствующий бит регистра состояния канала DMA), отключение приема по последовательному порту SPORT0 и установка в единицу глобально объявляемой переменной-флага ***DMA_done***, которая будет сигнализировать о завершении приема блока данных:

```
volatile int DMA_done = 0;
EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
      *pDMA3_IRQ_STATUS = 0x0001;
      *pSPORT0_RCR1 &= 0xFFFE;
      DMA_done = 1;
}
```

После того как инициализация завершена, необходимо снять маскирование прерывания в контроллере системных прерываний установкой соответствующего бита регистра SIC_IMASK, активизировать работу канала DMA и разрешить прием по последовательному порту:

```
*pSIC_IMASK = 0x00000020;
*pDMA3_CONFIG |= 1;
*pSPORT0_RCR |= 1;
```

Основная часть программы будет включать в себя цикл, в котором осуществляется проверка флага ***DMA_done***, и расчет комплексного БПФ принятого массива данных. Для вычисления БПФ в библиотеке функций цифровой обработки сигналов (ЦОС) компилятора С имеется ряд функций, прототипы которых приведены в заголовочном файле ***filter.h***. Эти функции работают с данными типа ***fract16*** и ***complex_fract16***, и для нашего примера мы воспользуемся функцией ***cfft_fr16***, которая имеет следующий синтаксис:

```
void cfft_fr16(const complex_fract16 input[],
      complex_fract16 output[],
      const complex_fract16 twiddle_table[],
      int twiddle_stride,
      int fft_size,
      int *block_exponent,
      int scale_method);
```

Назначение первых двух и пятого аргумента функции очевидно из их названий. Третий аргумент функции — это указатель на массив, содержащий таблицу синусов и косинусов, которая необходима для вычисления БПФ. Четвертый аргумент, ***twiddle_stride***, позволяет использовать одну такую таблицу большого размера для вычисления нескольких БПФ разной размерности и должен быть равен отношению размера таблицы к размерности БПФ. Поскольку в нашем случае такая возможность не требуется, этот аргумент должен быть равен единице. Последний аргумент функции указывает вариант масштабирования, используемый для предотвращения переполнения при вычислении функции БПФ. Он может принимать три значения: 1, 2 или 3, что соответствует статическому масштабированию (данные на входе каждого каскада БПФ делятся на 2), динамическому масштабированию (данные на входе каскада БПФ делятся на два, если модуль любого элемента данных больше 0,5) или отсутствию масштабирования. И, наконец, предпоследний аргумент — это указатель на переменную, в которую будет записан масштабирующий множитель для выходного массива. В случае ***scale_method = 1*** и ***scale_method = 3*** возвращаемое значение множителя всегда будет равно $log_2(fft_size)$ и 1 соответственно.

Для инициализации массива ***twiddle_table*** может быть использована функция:

```
void twidfftf_fr16(complex_fract16 twiddle_table [], int fft_size);
```

которую достаточно вызвать однократно при выполнении процедуры инициализации.

И, наконец, еще одна функция, которая потребуется нам для представления результатов в удобной форме,— это функция вычисления модуля комплексного числа:

```
fract16 cabs_fr16(complex_fract16 a);
```

Окончательный вид исходного кода программы представлен в листинге 1. Следует отметить, что этот код далеко не оптимален, и опытный разработчик легко найдет в нем ряд моментов, которые могут быть подвергнуты дальнейшей оптимизации. В то же время он хорошо подходит для ознакомления с общими принципами создания проектов для процессоров Blackfin на языке С и расширенных возможностей моделирования.

```
/************************************************************
 * TestSport.c
 ************************************************************/
#include "cdefBF537.h"
#include "sys\exception.h"
#include "fract.h"
#include "complex.h"
#include "filter.h"

volatile   complex_fract16 rx_buf[128];
volatile   int DMA_done = 0;
fract16    result [128];

EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
      *pDMA3_IRQ_STATUS = 0x0001;
      *pSPORT0_RCR1 &= 0xFFFE;
      DMA_done = 1;
}

int main (void)
{
      /* Begin adding your custom code here */
      complex_fract16 twiddle_table [128];
      complex_fract16 out_buf [128];
      int block_exp, i=0;

      twidfftrad2_fr16(twiddle_table,128);

      //SPORT0 initialization
      *pSPORT0_RCR1 = RFSR | IRFS | IRCLK;
      *pSPORT0_RCR2 = 0x1F;
      *pSPORT0_RCLKDIV = 0;
      *pSPORT0_RFSDIV = 31;

      //DMA initialization
      *pDMA3_CONFIG = SYNC|WNR|WDSIZE_32|FLOW_STOP|DI_
EN;
      *pDMA3_X_COUNT = 128;
      *pDMA3_X_MODIFY = 4;
      *pDMA3_START_ADDR = (int)rx_buf;

      //Interrupt initialization
      register_handler(ik_ivg9, Sport0_RX_ISR);

      //Interrupt, DMA and SPORT RX enable
      *pSIC_IMASK = 0 x00000020;
      *pDMA3_CONFIG  |= 1;
      *pSPORT0_RCR1   |= 1;

      //Wait for DMA completion
      while (! DMA_done);

      cfft_fr16((complex_fract16*)rx_buf, out_buf, twiddle_table,1,
            128,&block_exp,0);

      for(i=0; i<128; i++)
            result[i] = cabs_fr16 (out_buf[i]);

      return 0;
}
```

Листинг 1

Моделирование работы программы

После запуска сессии симулятора и успешной компоновки исполняемого файла ***TestSport.dxe*** необходимо настроить ввод входных данных для моделирования. В среде VisualDSP++ это делается при помощи «потоков» (streams). Настройка потока осуществляется в диалоговом окне, которое вызывается через команду меню ***Settings → Streams*** (рис. 4). Нажатие на кнопку ***Add*** («Добавить») вызывает новое диалоговое окно ***Add New Stream*** («Добавить новый поток»), которое разделено на две части — настройка источника (***Source***) и приемника (***Destination***). При моделировании ввода данных источником является текстовый файл, а приемником — внутренняя память или один из поддерживаемых симулятором периферийных модулей процессора ADSP-BF537. При моделировании вывода данных назначение изменяется на противоположное. Для

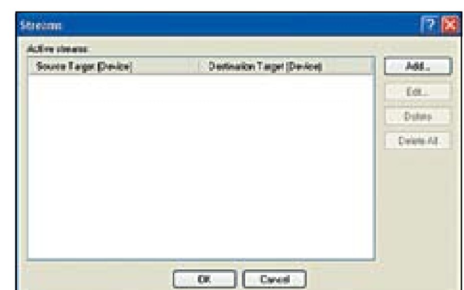

Рис. 4. Диалоговое окно Streams

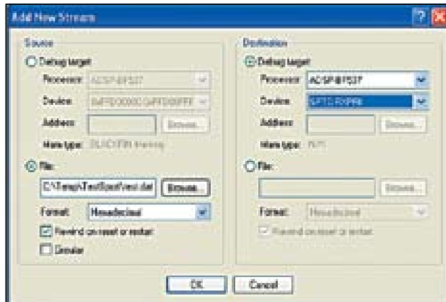

Рис. 5. Настройка потока ввода данных

файла-источника может быть выбран формат данных, а также опции автоматической инициализации потока при загрузке или перезапуске исполняемого файла (флажок ***Rewind on reset or restart***) и автоматического перехода к началу файла при достижении его конца (***Circular***). В нашем проекте необходимо выбрать настройки источника и приемника, как показано на рис. 5.

Текстовый файл должен содержать текстовые строки с данными в формате, который соответствует настройкам потока, где каждый элемент данных начинается с новой строки. Разработанная нами программа предназначена для работы с данными в формате fract16, и каждое 32-битное слово, принимаемое по последовательному порту, будет содержать как синфазную, так и квадратурную составляющую сигнала. Поэтому целесообразно выбрать шестнадцатеричный формат данных в файле-источнике и записать в каждую строку файла сначала шестнадцатеричное значение квадратурной составляющей отсчета, а затем шестнадцатеричное значение синфазной составляющей. Это может быть сделано, например, в среде MATLAB при помощи скрипта, представленного в листинге 2.

```
clear
t = 0:1:127;
x = 0.7*(exp(j*2*pi*0.1*t));
s = awgn(x,20);
re = fi(real(s),1,16,15);
im = fi(imag(s),1,16,15);
file = fopen('test.dat','w');
for i = 1:128
  fprintf(file,'%s%s\n',hex(im(i)),hex(re(i)));
end
fclose(file)
```

Листинг 2

Для наглядного представления входных данных и результатов расчета амплитудного спектра мы воспользуемся средствами графического отображения VisualDSP++. Для создания нового графика необходимо выбрать пункт меню ***View*** → ***Debug Windows*** → ***Plot*** → ***New***. В открывшемся диалоговом окне, которое показано на рис. 6, из выпадающего списка ***Type*** («Тип графика») выберите ***Line Plot*** («Линейный график») и задайте название графика (например, ***Input Data***) в поле ***Title***. Чтобы добавить на график линию данных синфазной составляющей, нужно нажать на кнопку ***Browse***, выбрать

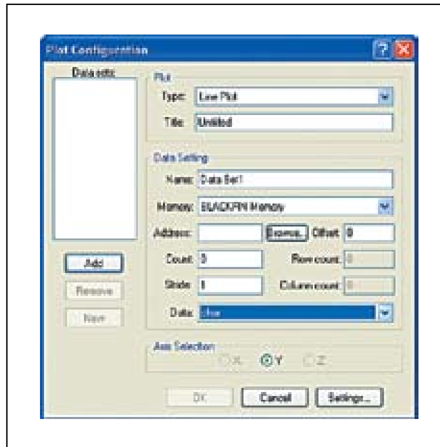

Рис. 6. Диалоговое окно Plot Configuration

в открывшемся списке позицию ***rx_buf*** и закрыть список нажатием кнопки ***OK***. В выпадающем списке типов данных (***Data***) для построения линейного графика отсутствует возможность выбора знаковых дробных чисел, поэтому выберите в нем пункт ***short***. В поле ***Name*** можно задать имя отображаемого набора данных (например, ***In-Phase***). В поле ***Offset*** задается смещение первого элемента данных относительно выбранного ранее адреса, и для синфазной составляющей изменять значение по умолчанию не нужно. В поле ***Count*** необходимо указать количество элементов данных на графике (в нашем примере — 128). И, наконец, поскольку отсчеты синфазной и квадратурной составляющих сигнала упорядочены в массиве через один, в поле ***Stride*** необходимо ввести число 2. При нажатии кнопки ***Add*** в списке ***Data Sets*** появится позиция с указанным ранее именем. Эти же действия необходимо повторить для добавления на график набора данных квадратурной составляющей. Единственное отличие будет заключаться в том, что в поле ***Offset*** потребуется изменить 0 на 1.

Аналогичным образом можно создать график с данными амплитудного спектра. При настройке графика необходимо выбрать из списка адресов позицию, соответствующую массиву ***result***, тип данных ***short***, Offset = 0, Count = 128 и Stride = 1.

Если теперь поставить точку останова на последней команде функции ***main()*** и запустить программу на исполнение, то по прошествии некоторого времени в окнах графиков отобразятся графики входного сигнала и спектра наподобие показанных на рис. 7.

Заключение

В этой статье на примере проекта, реализующего чтение данных по последовательному порту в режиме DMA, были рассмотрены основные принципы создания проектов для процессоров Blackfin на языке C, а также некоторые расширенные возможности симулятора, такие как моделирование ввода/вывода данных и графическое отображение содержимого памяти процессора. Несмотря на то, что рассматриваемый пример является довольно упрощенным и, в некоторой степени, искусственным, он позволяет продемонстрировать такие важные аспекты программирования, как обращение к внутренним регистрам процессора, организация обмена данными в режиме DMA и обработка прерываний. Этот пример может использоваться начинающими разработчиками в качестве отправной точки для создания собственных приложений, более приближенных к реальности. ■

Литература

1. VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors. Revision 5.2, September 2009 — www.analog.com
2. Anderson A. Programming and Optimizing C code — http://www.dspdesignline.com/197006981

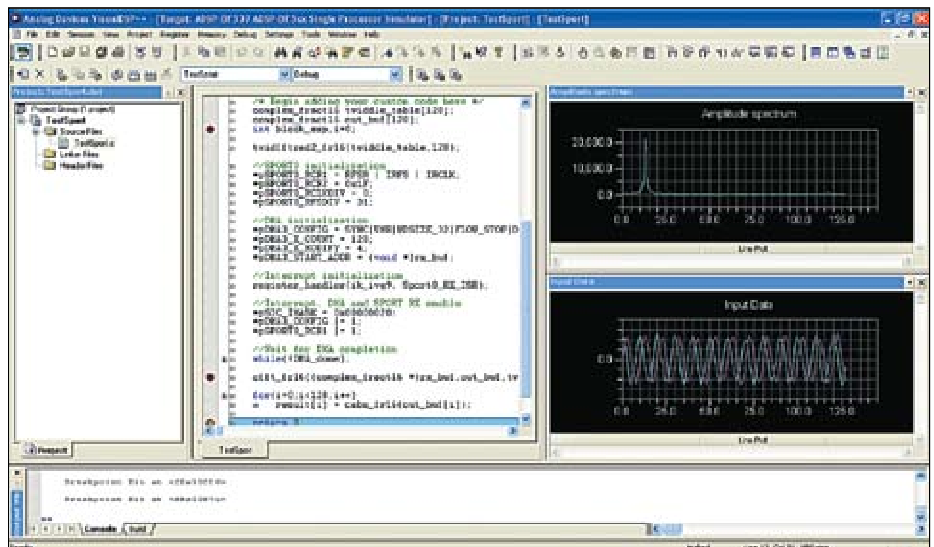

Рис. 7. Окно VisualDSP++ с графиками входных данных и результатов расчета амплитудного спектра