Руководство программирования на C для семейства процессоров ADSP-2100

Сборник советов, статей DSPatch, вопросов/ответов и примеров из технических замечаний от проектной группы DSP Analog Devices

Версия 1.0

Автор перевода: Силантьев В.И.

1. «Программирование на C для DSP». DSPatch

Перепечатка статей «Программирование на C для DSP» из DSPatch №№26-33 затрагивает темы основ C для DSP, оперативных средств, использования встроенного кода на языке ассемблера, использования библиотечных функций и заканчивается примером реализации БПФ, который выполняется на плате EZ-LAB.

Последующие статьи этого раздела предоставляют советы и рекомендации по программированию на C для DSP. Отдельные статьи касаются написания эффективного C кода, который показывает преимущества архитектуры семейства процессоров ADSP-2100 и ADSP-21000. Эта статья приводит как аргументы в пользу использования C, так и недостатки использования C, в то время как последующие рассказывают о конкретных реализациях.

1.1. Введение

(DSPatch #26)

Язык программирования C получил широкое распространение как основной язык программирования при создании программного обеспечения для встроенных микропроцессоров и программно-аппартной разработки. Продемонстрированное использование C при разработки встроенных приложений может быть отнесено и к DSP приложениям. Приводимые аргументы неотразимы:

- Многие инженеры и большинство недавно дипломированных инженеров уже знают C, так что разработку приложений можно начать быстро, без утомительного изучения тонкостей нового языка ассемблер и набора инструкций для отдельных DSP.

- C позволяет приложениям быть разработанными, макетированными, анализированными и протестированными на ПК с использованием коммерчески доступными C компиляторами и средствами. Этот прообразный код может быть иногда использован без изменений.

- Язык C определен. Поэтому код на C легко переносим с на любой DSP процессор, где существует поддержка языка C. Стабильность языка в дальнейшем гарантируется введением ANSI стандарта для C.

- Большинство программного обеспечения, разрабатываемого для DSP приложений, является кодом управления, требующего сложных структур и алгоритма управления. Этот код управления обычно представляет собой малую часть времени выполнения программы, но несет важную часть в успехе инженерных усилий при создании программного обеспечения. Код управления удобно создавать на C. К тому же, C предоставляет простой доступ к аппаратной части. Код, который должен получить доступ к управлению аппаратной частью, регистрам состояния и портам ввода/вывода, может быть написан на C. Использование C легче, чем язык ассемблера для кода управления и уменьшит время, необходимое для разработки приложения.

Аргументы против C

Несмотря на аргументы, приведенные выше, достижения C как языка программирования для разработки DSP приложений не так хороши.

В ранние 80-е, сразу же после того. Как приложения на DSP получили практику, пользователи начали испытывать потребность в утонченности средств разработки DSP кода. В исключительных случаях пользователям была нужна поддержка C. Производители ответили C компиляторами. Эти C компиляторы первого поколения часто не отличались мощной организацией циклов и возможностью адресацией массивов, используемой архитектурой DSP, и в целом были неэффективны для производительности, обладаемой DSP.

Темы, затрагиваемые в следующих разделах.

В следующих разделах мы обсудим элементы C программ и использования DSP/C для создания наиболее эффективного кода. Темы будут касаться организации циклов, массивов данных, использования встроенного ассемблера и передачи параметров.

1.2. Инициализация управляющих регистров, отображенных в памяти ADSP-2101

(DSPatch #27)

Встроенная периферия процессоров семейства ADSP-2101, такая как, последовательные порты, таймер и интерфейс хост-порта, настраивается с помощью управляющих регистров, отображенных в памяти, расположенных в верхнем 1K внутренней памяти данных. Для установки этих регистров, ваша C программа должна записать в них инициализирующие значения после включения питания системы.

Вместо обращения к языку ассемблера (путем внедрения кода или компиляцией с внешним модулем на ассемблере), вы можете записать значения в управляющие регистры, отображенные в памяти, непосредственно из C. Следующий пример программы для ADSP-2101 демонстрирует, как это можно сделать.

Основная программа содержится в файле `example.c`, показанной на Листинге 1. Эта программа выводит счетчик через SPORT0 (последовательный порт 0). Заголовочный файл `cdef2101.h`, показанный на Листинге 2, использует директиву препроцессора #define для назначения символьного имени каждому регистру, отображаемому в памяти.

Листинг 1. example.c

```
/* example.c*/
#include "header.h"

#pragma inline_data
.VAR/DM loop_count_;
#pragma inline_data

extern ushort loop_count;
#define CONSTANT 0
void init2101();
void transmit(short i);

/* Этот пример демонстрирует инициализацию */
/* регистров, отображенных в памяти ADSP-21xx */
/* с помощью функции init2101(). Он также */
/* показывает функцию transmit(), которая */
/* используется для передачи данных через */
/* последовательный порт. */

main()
{
      short i;
      init2101();       /* инициализация регистров */
                        /* включая регистры SPORT0. */
      loop_count=5;     /* установка счетчика */
      for(i=CONSTANT; i<loop_count; i++)
      {
            transmit(i);/* Вызов встроенной ассемблерной функции */
                        /* для передачи через SPORT0. */
      }
}
```

```
void transmit(short i)
{
      #pragma inline
      #include "asm_sprt.h"
      readstack(SI,1,DM);/* чтение значения i из стека C */
      TX0=SI;           /* передача */
      #pragma inline
}
```

Листинг 2. cdef2101.h

```
/* cdef2101.h */

/* Это заголовочный файл, который назначает регистрам, отображенным в памяти */
/* ADSP-2101, символьные имена. */

#define sp1_autobuf           *(int *) 0x3fef
#define sp1_rfsdiv            *(int *) 0x3ff0
#define sp1_sclkdiv           *(int *) 0x3ff1
#define sp1_control_reg       *(int *) 0x3ff2
#define sp0_autobuf           *(int *) 0x3ff3
#define sp0_rfsdiv            *(int *) 0x3ff4
#define sp0_sclkdiv           *(int *) 0x3ff5
#define sp0_control_reg       *(int *) 0x3ff6
#define sp0_multiTX0          *(int *) 0x3ff7
#define sp0_multiTX1          *(int *) 0x3ff8
#define sp0_multiRX0          *(int *) 0x3ff9
#define sp0_multiRX1          *(int *) 0x3ffa
#define tscale                *(int *) 0x3ffb
#define tcount                *(int *) 0x3ffc
#define tperiod               *(int *) 0x3ffd
#define dm_wait_reg           *(int *) 0x3ffe
#define system_control_reg    *(int *) 0x3fff
```

Директива препроцессора:

```
#define system_control_reg    *(int *) 0x3fff
```

приводит 0x3FFF как указатель на integer, и далее разыменовывает его. Другими словами, символ system_control_reg становится эквивалентом целого значения, указываемого 0x3FFF. Теперь он может быть использован в операторе C, например,

```
system_control_reg=0x1000;
```

который записывает значение 0x1000 в ячейку 0x3FFF, отображенную в памяти. Этот оператор компилируется в ассемблерную инструкцию:

```
IO=0X3FFF;
DM(IO,M2)=0X1000; /*M2=0*/
```

Регистры, отображенные в памяти, инициализируются функцией `init2101()`, которая использует символы из `cdef2101.h`. Функция `init2101()` не принимает аргументов и не возвращает значения. В `example.c` мы вызываем эту функцию оператором:

```
init2101();
```

Функция находится во внешнем файле `init2101.c` (см. Листинг 3), который должен быть соединен с основной программой. Это можно сделать включением файла в командную строку при вызове компилятора.

```
CC21 example init2101
```

Листинг 3. init2101.c

```
/* init2101.c */

/* Здесь представлена подпрограмма, которая инициализирует */
/* регистры, отображенные в памяти ADSP-2101. */

#include "cdef2101.h"
void init2101()
{
      dm_wait_reg = 0x0;      /* нет режимов ожидания */
      sp1_autobuf = 0x0;
      sp1_rfsdiv = 0x0;
      sp1_sclkdiv = 0x0;
      sp1_control_reg = 0x0;
      sp0_autobuf = 0x0;
      sp0_rfsdiv = 0x255;     /* деление на 256 для выборки 8 КГц */
      sp0_sclkdiv = 0x2;      /* вырабатывать 2.048 МГц SCLK */
      sp0_control_reg = 0x6b27; /* внутренний тактовый генератор */
                              /* синхронизация принимаемых кадров, норм. */
                              /* синхронизация передаваемых кадров, норм. */
                              /* внутренняя синхронизация разрешена */
                              /* u-law компандирование, 8-разрядное слово */
      sp0_multiTX0 = 0x0;
      sp0_multiTX1 = 0x0;
      sp0_multiRX0 = 0x0;
      sp0_multiRX1 = 0x0;
      tscale = 0x0;
      tcount = 0x0;
      tperiod = 0x0;
      system_control_reg = 0x1000; /* разрешить SPORT0 */
                                   /* SPORT1 = IRQ1, IRQ0 */
}
```

1.3. Первые шаги

(DSPatch #28)

В этой главе мы остановимся на основных понятиях, связанных с написанием хорошо структурированной программы. Перед написанием вашей программы очень важно понять разницу в производительности программы, написанной на C и программой написанной на ассемблере. Также важно понять вопросы, относящиеся к написанию алгоритмов для встроенных DSP систем на C.

Рекомендуется, чтобы вы были хорошо знакомы с рабочей моделью C компилятора и хорошо осведомлены о регистрах ADSP-21xx, чтобы избежать конфликтных ситуаций.

Оценка ассемблера и C

При каждом проектировании встроенной DSP системы реального времени встает вопрос о языке программирования. При использовании DSP от Analog Devices у вас имеется выбор между C и ассемблером. Выбор оптимального решения осуществляется между языком ассемблера, C или комбинацией этих двух языков. Оно включает в себя требования по скорости обработки и необходимой памяти, план разработки, переносимость, существование предварительно написанного кода и опыт программирования.

Во-первых, всегда следует осознавать, что код ассемблера всегда более эффективен и занимает меньший объем в памяти программ, чем код, написанный на C. Если вы не пишите программу для системы реального времени, скорость обработки может быть не так важна (объем памяти может остаться критичным). Разница эффективности между C кодом и кодом ассемблера может быть порядка три к одному. Между тем, существует много причин для выбора C как языка

программирования. Если у вас уже есть значительная часть кода, написанного на C, вы, очевидно, захотите использовать C компилятор Analog Devices. Если вы заинтересованы в переносимости кода между различными платформами, вы также решите писать ваш код на C. Язык ассемблера Analog Devices будет работать только на DSP Analog Devices, при этом нет каких-либо эффективных программ для трансляции кода между различными производителями DSP.

Опыт программирования и ваш план разработки могут внести большой вклад в решение выбора между C и ассемблером. Хотя язык ассемблера Analog Devices является наиболее легким в освоении и программировании, многие программисты уже имеют хорошие знания языка C.

Поэтому, если вы столкнулись с активным планом разработки, то программирование на C сохранит ваше время.

Важно помнить, что написание кода на C будет быстрым, только при условии, что вы не тратите время на оптимизацию кода, чтобы удовлетворить требования по объему занимаемой памяти и скорости обработки.

Если вы ограничены в скорости обработки или ограничены в объеме занимаемой памяти, вы захотите посмотреть и проверить C библиотеки Analog Devices. Библиотека представляет собой коллекцию функций, написанных на языке ассемблера, которые можно вызывать из ваших C программ. Общие функции включают файлы `math.h` (`cos`, `sin`, `sqrt` и т.д.), подпрограммы фильтров, БПФ и т.п. Это может стать первым шагом при оптимизации вашего кода.

Для дальнейшей оптимизации вашего C кода можно выбрать смешанный код из C и ассемблера.

Наиболее эффективный путь, это определение узких мест вашего кода. Данную операцию можно провести с помощью профилирования вашего кода в программе моделирования. Разбейте эти части C кода и выделите их в вызываемые функции. Функции могут находиться в другом файле/модуле, или могут представлять собой подфункции, вызываемые внутри файла/модуля. После того как вы определитесь в выборе функций, вы найдете полезным написание некоторых подпрограмм на ассемблере, что увеличит скорость обработки. При смешанном коде, дополнительная возможность C компилятора позволяет вам обращаться к переменным, не заботясь об их расположении.

Компилятор может попытаться переместить ассемблерные инструкции для создания более эффективного C кода. Спецификатор **volatile** говорит компилятору, что такие инструкции не должны перемещаться.

Тонкости встроенной системы

Если вы привыкли писать программы на C для компьютерных платформ, как PC, вы можете пользоваться такими преимуществами, как стандартный ввод/вывод и виртуальная память. Если вы пишите программу для встроенных приложений, как DSP процессор, вы будете заинтересованы в использовании структуры процессора, его внутренних прерываний, наборе регистров, как он управляет стеком C, и количеством доступной памяти программ и данных.

Стандартные функции ввода/вывода содержатся в заголовочном файле `stdio.h`. Стандартный ввод/вывод включает в себя символьный ввод/вывод (`getchar`, `putchar`), форматированный ввод/вывод (`printf`, `scanf`) и файловый ввод/вывод. Библиотека C Analog Devices не поддерживает функции `stdio.h`, так как они некорректны для использования во встроенных приложениях.

Поэтому, как программист для встроенных систем, вы должны быть более осведомленным со структурой ввода/вывода DSP. Структура ввода/вывода DSP может включать в себя: последовательные порты, интерфейс хост-порта, встроенные кодеки, или отраженные в памяти АЦП и ЦАП. В зависимости от конфигурации ввода/вывода DSP, вы можете получать доступ через один из двух вариантов, как через регистры, отраженные в памяти (что вы можете делать и на C), так и через внутренние регистры DSP (что требует встроенной ассемблерной инструкции в коде). Так как

функции форматированного ввода/вывода не доступны во встроенных системах, нет и функции `printf` (которая позволяла вывод на экран компьютера), помогавшей многим программистам проводить отладку кода. Если вы переносите на встроенное приложение существующий C код, который содержит эти функции, вы можете закомментировать эти строки или написать подпрограмму-заглушку `printf()`, которая заменит функции из `stdio.h`.

Пространство памяти DSP основано на модифицированной гарвардской архитектуре, противоположной фон-неймановской архитектуре микропроцессоров и микроконтроллеров. При фон-неймановской архитектуре инструкции программы и данные хранятся в общей области памяти. В гарвардской архитектуре пространство данных и программы разделено. При модифицированной гарвардской архитектуре можно хранить программные данные (например, коэффициенты фильтра) в памяти программ, также как и инструкции программы. Почему это важно? Архитектура DSP позволяет проводить чтение двух слов данных и захват инструкции за один цикл.

Многие из этих возможностей, необходимые для встроенных систем, управляются в заголовке исполняемой программы. Заголовок выполняемой программы устанавливает таблицу векторов прерываний, задает необходимые переменные и объявления особенных параметров процессора, устанавливает стек C и область памяти, выделяемая программе для динамически размещаемых структур данных, и вызывает основную программу. Компилятор C Analog Devices автоматически добавляет этот заголовок к коду, но вы должны быть осведомлены в его содержимом. Вы можете изменить заголовок исполняемой программы для достижения эффективности. Для получения дополнительной информации обращайтесь к *ADSP-2100 Family C Compiler and Runtime Library Manua*l.

Использование регистров компилятором C ADSP-21xx

C компилятор создает код, который использует определенные регистры. Вы должны понять, как компилятор работает с регистрами для избежания конфликтных ситуаций.

Функции, которые возвращают одно слово (`int`, `char`, `short` и однословные структуры), помещают результат в регистр AR.

Функции, которые возвращают два слова (`float`, `double` и двухсловные структуры), помещают результат старшего слова в SR1 и младшего слова в SR0.

Следующие рабочие регистры не требуют сохранения функцией:

```
AR, AF, AY1, MR2, MR1, MR0, MF, MY1, I1,
M3, I6, M5, SB, SE, SI, SR1, SR0, PX
```

Параметры, передаваемые при вызове функции

Компилятор передает параметры в функции через регистры всякий раз, когда это возможно. Компилятор использует следующие правила для определения, как передавать параметры:

- Первый однословный параметр передается через регистр **AR** (исключая, если справедливо #4).
- Второй однословный параметр передается через регистр **AY1** (исключая, если справедливо #4).
- Все другие параметры передаются через·стек.
- Если функция объявлена с переменным числом параметров, последний аргумент и все переменные аргументы передаются через стек.
- Многословные параметры и все последующие параметры передаются через стек.

Регистры с фиксированным значением компилятора C ADSP-21xx

Следующие регистры имеют фиксированные значения. Эти регистры не должны изменятся в коде пользователя.

```
M1 == 1
M2 == 0
I4 == указатель стека (может изменяться во время операций со стеком)
M4 == указатель фрейма (может изменяться во время операций со стеком)
M6 == 0
```

L регистры должны быть равны нулю в среде C, но могут временно изменяться пользовательским кодом, до тех пор пока возвращаются в ноль. Диспетчер прерывания устанавливает все L регистры в ноль, за исключением L2 и L3, перед вызовом подпрограммы прерывания.

Регистры L2 и L3 не изменяются диспетчером прерывания, так как они часто используются как регистры автобуферизации, и изменение может вызвать некорректную операцию.

Исполняемая библиотека C не использует регистры I2-L2 или I3-L3. Пользователь должен с осторожностью устанавливать регистры L2 или L3 в значения отличные от нуля, из-за резервирования регистров I2 и I3.

Регистры пользователя

Регистры DAG I2 и I3 могут быть зарезервированы. Когда пользователь резервирует эти регистры, они не используются компилятором. Исполняемая библиотека C не использует регистры I2 или I3. Эти два регистра часто используются в приложениях автобуферизации. Если регистры зарезервированы, они могут быть модифицированы функциями исполняемой библиотеки.

Если пользователь не резервирует регистры I2 или I3, то регистры L2 и L3 должны изменяться с особой осторожностью.

Использование опции *-loader* ППЗУ распределителя.

Семейство процессоров ADSP-2100 обладает восемью программно выбираемыми загрузочными страницами. Они позволяют вам иметь восемь программ, которые могут быть загружены во внутреннюю память.

Большинство C программ используют и внутреннюю. И внешнюю память. Отдельные программы, которым необходима внешняя память или инициализированная память данных, могут использовать опцию `-loader` ППЗУ распределителя (PROM splitter).

ППЗУ распределитель создает код в загрузочной странице, который будет инициализировать вашу память программ и данных. Последняя загрузочная страница содержит ваш образ внутренней памяти программ. Опция `-loader` не использует ни память программ, ни память данных.

Для использования опции `-loader` ППЗУ распределителя:

- Не помещайте любой код в загрузочную память
- Скомпонуйте образ (.exe файл), который содержит только память программ и данных. Свободно используйте инициализированную память данных и внешнюю память.
- Вызовите ППЗУ распределитель (`spl21`) с опцией `-loader`. Он создаст загружаемый образ, который необходимо предварительно загрузить, а затем запустить программу.

Эти основные указания помогут вам создать программу с использованием как языка C, так и языка ассемблера, которую вы можете запустить на своей DSP системе с минимальным количеством проблем. Более подробное описания можно найти в *C Compiler User's Manual.*

В следующей главе мы обсудим технику настройки ваших оперативных средств. Последующие разделы предложат характерные примеры.

1.4. Настройка оперативных средств ADSP-2100

(DSPatch #29)

В последней главе мы уделили внимание основным понятиям, необходимых для начала работы, написанию хорошо структурированной программы и обсудили различия между кодом C и ассемблерным кодом. В этой главе мы вспомним все чему научились и рассмотрим простой пример, от начала до конца. Во время рассмотрения мы обсудим, как и где можно настроить ваши оперативные средства.

Примечание: предполагается, что вы установили, как минимум, версию 5.1 программного обеспечения разработки приложений для семейства процессоров ADSP-2100. Это обновление содержит новый G21 C компилятор, который заменяет компилятор CC21.

Пример C программы

Давайте начнем с создания небольшой C программы. Программа `simple.c` инициализирует две переменные и вызывает подпрограмму для их сложения. Листинг 4 показывает эту программу.

Листинг 4. simple.c

```
/* simple.c */

int dm x_variable;
int pm y_variable;
int sum;
main()
{
      x_variable=1;      /* инициализация значения x */
      y_variable=5;      /* инициализация значения y */
                         /* вызов подпрограммы сложения */
      sum=add_routine(x_variable,y_variable);
      return(0);         /* завершение основной программы */
}

int add_routine(x,y)     /* подпрограмма сложения */
int x;
int y;
{
      int z;
      z=x+y;             /* произвести сложение */
      return(z);         /* возвращение результата */
}
```

Мы поместили одну переменную в область памяти данных, а другую в область памяти программ. DSP процессоры Analog Devices позволяют двойную организацию модели памяти модифицированной гарвардской архитектуры (память программ содержит данные и код, память данных содержит только данные). Это дает преимущество, так как за один программный цикл осуществляется чтение памяти программ и памяти данных и захват инструкции, в отличие от двух последовательных захватов. Большинство микропроцессоров работают на фон-неймановской архитектуре и имеют общую память

для программ и данных. Компилятор G21 имеет расширение ANSI языка для поддержки двойной модели памяти DSP. Зарезервированы два ключевых слова (`dm` и `pm`), используемые для объявления переменных. Объявление переменной, хранящей сумму, не использует ключевых слов, по умолчанию находясь в память данных (`dm`).

Программа инициализирует глобальные переменные x_variable в памяти данных и y_variable в памяти программ, затем передает их значения в подпрограмму `add_routine()`. Подпрограмма `add_routine()` принимает передаваемые параметры и возвращает из сумму в основную программу. Локальные переменные подпрограммы хранятся в стеке.

Теперь, когда у нас есть программа, давайте решим, какой C компилятор должен преобразовать инструкции в машинный код процессоров семейства ADSP-2100. Для начала мы должны определить системную конфигурацию для нашей простой системы. Будем использовать файл `simple.sys`, показанный на Листинге 5, для описания системной конфигурации ADSP-2105, использующего только внутреннюю память процессора. Объявляется сегмент загрузочной памяти ПЗУ длиной 1024 слова, следующий за ним сегмент памяти программ ОЗУ длиной 1024 слова и сегмент памяти данных ОЗУ длиной 512 слов.

Листинг 5. simple.sys

```
.system simple_ach;
.adsp2105;
.mmap0;
.seg/rom/boot=0 boot_page_0[1024];
.seg/pm/ram/abs=0/code/data int_pm[1024];
.seg/dm/ram/abs=14336/data int_dm[512];
.endsys;
```

Используйте команду:

```
bld21 simple.sys
```

для создания файла архитектуры `simple.ach` для компилятора G21.

Команда G21 управляет операциями других программ средств разработки, проводя процесс трансляции (который может включать препроцессирование C, компиляцию, ассемблирование и/или компоновку). Мы будем использовать ключ `-v` (`verbose`), для иллюстрации, что процесс управляется G21, и ключ `-a` для указания файла архитектуры.

```
g21 simple.c -v -a simple.ach
```

Эта команда компилирует, ассемблирует и компонует наш файл `simple.c` и создает исполняемый файл `simple.exe`. Она также производит необходимые операции, которые прозрачны программисту. Они включают заголовок исполняемой программы и распределение памяти под стек и «кучу».

Заголовок исполняемой программы

Заголовок исполняемой программы содержит ассемблерный код, содержащийся в CRTL (исполняемая библиотека C программ), который компилятор G21 добавляет к нашей программе `simple.c`. Заголовок начинается с адреса 0x0000 памяти программ и содержит таблицу векторов прерываний для управления прерываниями DSP. Он также устанавливает оперативные средства C, вызывает основную программу `main()` и включает программу обработки, которая следует после функции `return()` из `main()`.

Давайте посмотрим заголовок исполняемой программы для ADSP-2105, показанный на Листинге 6.

Листинг 6. $ADI_DSP/lib/2105_hdr.dsp

```
.MODULE/ABS=0           ADSP-2105_Runtime_Header;
.ENTRY                  ___lib_prog_term;
.EXTERNAL               ___lib_setup_everything;
.EXTERNAL               main_;
.EXTERNAL               ___lib_int2_ctrl, ___lib_sp0x_ctrl, ___lib_sp0r_ctrl;
.EXTERNAL               ___lib_int1_ctrl, ___lib_int0_ctrl, ___lib_tmri_ctrl;
__Reset_vector:             CALL ___lib_setup_everything;
                            CALL main_; {запуск C программы}
___lib_prog_term:           JUMP ___lib_prog_term;
                            NOP;
__Interrupt2:               JUMP ___lib_int2_ctrl;NOP;NOP;NOP;
__Sport0_trans:             JUMP ___lib_sp0x_ctrl;NOP;NOP;NOP;
__Sport0_recv:              JUMP ___lib_sp0r_ctrl;NOP;NOP;NOP;
__Interrupt1:               JUMP ___lib_int1_ctrl;NOP;NOP;NOP;
__Interrupt0:               JUMP ___lib_int0_ctrl;NOP;NOP;NOP;
__Timer_interrupt:          JUMP ___lib_tmri_ctrl;NOP;NOP;NOP;

.ENDMOD
```

При сбросе питания DSP начинает исполнять код с адреса PM[0x0000], вектор прерывания по сбросу заголовка исполняемой программы. Он вызывает подпрограмму инициализации `____lib_setup_everything`, которая устанавливает процессор по умолчанию, создает стек и «кучу» и обрабатывает аргументы командной строки.

Далее заголовок исполняемой программы вызывает подпрограмму `main()`, и начинается выполнение программы `simple.c`. Если мы разрешим (размаскируем) прерывание и оно произойдет, счетчик команд перейдет к соответствующему адресу в таблице векторов прерывания. Показанный заголовок исполняемой программы содержит вектора прерываний IRQ0, IRQ1 и IRQ2 (внешнее прерывание для ADSP-2105), прерывание таймера и последовательного порта. Например, прерывание таймера вызовет перемещение счетчика команд на адрес PM[0x0018] и начнется выполнение подпрограммы обработки прерывания (CRTL `interrupt`)

При выполнении инструкции `return()` в основной программе, вызывается подпрограмма обработки, производящая зацикливание программы на себя по метке `____lib_prog_term`.

Приведенный заголовок исполняемой программы используется по умолчанию компилятором G21. Для особых приложений его можно изменить. Одной из причин может быть вставка своей таблицы векторов прерываний и подпрограмм обработки.

Для создания своего заголовка исполняемой программы, скопируйте соответствующий файл (например, `2105_hdr.dsp` для системы ADSP-2105) из каталога `$ADI_DSP/21xx/lib` в ваш рабочий каталог. Измените его по своим требованиям и запустите программу ассемблирования:

```
asm21 -c 2105_hdr.dsp
```

для создания `.obj` файла. Компилятор использует по умолчанию заголовок `2105_hdr.dsp`. Для указания другого заголовка используйте ключ `-runhdr`. Команда будет выглядеть следующим образом:

```
g21 simple.c -v -a simple.ach -runhdr 2105_hdr.dsp
```

Стек

Стек представляет собой буфер зарезервированной памяти, используемой C компилятором для передачи переменных между подпрограммами. Количество требуемой переменным памяти меняется в зависимости от числа записанных в стек данных. По умолчанию размер стека составляет 1024 слова памяти данных ОЗУ. Листинг инициализации стека приведен ниже.
Обычно размер стека требует модификации для размещения приложения. Давайте рассмотрим нашу программу `simple.c`. Следует заметить, что 1024 слова, зарезервированных для стека, слишком много. Поэтому, мы можем скопировать подпрограмму `stack.dsp` из каталога `$ADI_DSP/21xx/lib/src` в рабочий каталог, уменьшить размер стека и откомпилировать `stack.dsp` вместе с `simple.c`.

Листинг 7. $ADI_DSP/lib/src/stack.dsp

```
/* Этот модуль располагается в CRTL. Если вы хотите изменить размер стека
скопируйте этот файл в ваш рабочий каталог и компилируйте его там.*/

.MODULE                         Stack_Declaration;
.VAR/DM/RAM                     stack[1024], ____top_of_stack;
.GLOBAL                         ____top_of_stack;
.ENDMOD;
```

Динамическая память, «куча»

Динамическая память, или «куча», также как стек, является памятью, резервируемой C компилятором. «Куча» резервируется для программ, вызывающих функцию `malloc()`. Размер памяти по умолчанию равен 128 словам памяти данных ОЗУ. Вы можете изменить размер памяти, действуя таким же образом, как и в случае со стеком. Листинг 8 показывает модуль резервирования динамической памяти.

Листинг 8. $ADI_DSP/lib/src/heap.dsp

```
/* Этот модуль располагается в CRTL. Если вы хотите изменить размер «кучи»
скопируйте этот файл в ваш рабочий каталог и компилируйте его там.
*/

.MODULE                         Heap_Declaration;
.VAR/DM/RAM                     ___heap[128];
.GLOBAL                         ___heap;
.ENDMOD;
```

Тестирование нашей программы *simple.c*

Теперь вы знаете, что компилятор C делает автоматически, и что вы можете изменить в заголовке исполняемой программы, стеке и «куче». Давайте вернемся к программе `simple.c`. Следующая команда компилирует наш файл.

```
g21 simple.c -g -v -a simple.ach -runhdr 2105_hdr.dsp
```

Ключ `-g` создает отладочную информацию, используемую в программе моделирования. Для вызова программы моделирования и тестирования нашей C программы наберите команду:

```
sim2101 -a simple -e simple
```

Ключ `-a` говорит программе моделирования, что следует использовать файл архитектуры `simple.ach`, а ключ `-e`, что следует загружать исполняемый файл simple.exe (выход компилятора). В следующей главе мы более подробно рассмотрим использование встроенного ассемблерного кода в вашей C программе. Этот метод позволит увеличить эффективность программы.

1.5. Согласование C и ассемблерных подпрограмм

(DSPatch #29)

В предыдущей главе мы обсудили, как настроить оперативные средства C, изменить настройки стека, динамической памяти и заголовка исполняемой программы. В этой части будет обсуждаться применение ассемблера для упрощения и оптимизации вашей C программы. Мы также рассмотрим настройку автобуферизации последовательного порта на C.

Пример программы, `main.c`, показывает принципы согласование языка ассемблера (см. Листинг 9). Для нашей системы мы будем использовать также файлы `asm_fn.dsp`, `init2101.c`, `mreg2101.h`, `2101_hdr.dsp`, `stack.dsp` и `simple.sys`. исходные тексты этих файлов доступны для загрузки через FTP.

Листинг 9. main.c

```
/* Это основная программа для примера автобуферизации на C. */

#include "mreg2101.h"

extern init2101();
extern int rx_buf[];                /* установка памяти для автобуферизации */

int add(int a, int b);              /* прототип функции сложения */

void main()
{
      int i,j,sum,result;
      asm("i2=^rx_buf_; \
           l2=%rx_buf_;");          /* установка указателя и длины буфера */

      init2101();                   /* инициализация регистров, включая SPRT0 */

      asm("ifc=0x3f;
                                    /* очистка ожидающих прерываний */ \
      nop;                          /* один цикл ожидания для ifc */ \
      imask=0x8;");                 /* размаскировывание прерывания SPRT0 RCV */

      while (1)                     /* бесконечный цикл CPB */
      {
            asm("idle;");           /* ожидание прерывания */
                                    /* при прерывании буфер rx_buf содержит 256
                                    /* слов. Принятых от SPORT0 */
            sum=rx_buf[0];          /* чтение первого слова буфера */
            for (i=1;i<LENGTH-1;i++) /* цикл суммирования данных буфера */
            {
                  j=rx_buf[i];
                  sum=add(j,sum);   /* вызов подпрограммы сложения */
            }
            asm volatile ("tx0=%0;" :: "a" (result));
                                    /* вывод данных через SPORT0 */
      }
}

/* подпрограмма на ассемблере для сложения двух чисел */

int add(int x, int y)
{
      asm("ar=ar+ay1;");
}
```

Имеется несколько способов добавления кода на языке ассемблера в C программу и несколько причин делать это. Одной из причин является исполнение операций, которые существуют только в языке ассемблера (например, инструкция IDLE) и доступ к отдельным регистрам DSP (например, IMASK). Другой причиной является критичное время выполнения, так как язык ассемблера более эффективный, чем C. Вы можете усилить мощность языка ассемблера использованием библиотеки рабочих C программ (CRTL), которые написаны на нем (мы обсудим это в следующей главе). Вы можете также встроить инструкции ассемблера в ваш C код и компоновать объектные файлы, полученные с языка ассемблера, вместе с объектными файлами, полученными с языка C.

Понимание синтаксиса ассемблерного кода и регистров

Во-первых, перед тем, как использовать ассемблер вместе с C, вы должны получить начальные знания языка ассемблера для семейства процессоров ADSP-2100. *ADSP-2100 Family User's Manual* является хорошо написанным руководством

Во-вторых, вы должны понять, как компилятор G21 резервирует внутренние регистры процессора ADSP-21xx. Никогда не меняйте регистры, используемые для операций со стеком I4, M4 и L4. Для автобуферизации последовательного порта используйте только регистры I2 и I3 (вместе с M1, который устанавливается в 1 компилятором).

Зарезервированные регистры G21

Эти регистры зарезервированы компилятором. Используйте их временно и всегда восстанавливайте.

Регистр	***Значение***
L0	0
L1	0
L5	0
L6	0
L7	0
M1	1
M2	0
M6	0

Эти регистры зарезервированы компилятором, но без определенного значения. Используйте их временно и всегда восстанавливайте.

M0	AX1
M7	AY0
MX0	I0
MX1	I5
MY0	I7
AX0	

Эти регистры могут быть зарезервированы для автобуферизации.

I2	определяется пользователем
I3	определяется пользователем

Эти регистры зарезервированы для работы со стеком. Не изменяйте их.

I4	указатель стека
M4	указатель фрейма
L4	0

Это регистры общего пользования. Их не нужно сохранять.

Регистр	***Значение***
AF	определяется пользователем
AR	определяется пользователем
AY1	определяется пользователем
I1	определяется пользователем
I6	определяется пользователем
M3	определяется пользователем
M5	определяется пользователем
MF	определяется пользователем
MR0	определяется пользователем
MR1	определяется пользователем
MR2	определяется пользователем
MY1	определяется пользователем
PX	определяется пользователем
SB	определяется пользователем
SE	определяется пользователем
SI	определяется пользователем
SR0	определяется пользователем
SR1	определяется пользователем

DSP содержит регистры, отраженные в памяти, которые располагаются во внутренней зарезервированной памяти данных. Для инициализации регистров, отраженных в памяти, на C, используйте заголовочный файл `mreg2101.h` (см. Листинг 10). Этот файл определяет символьные имена адресам регистров. Регистр установки автобуферизации последовательного порта 0 находится по адресу 0x3FF3.

Листинг 10. mreg2101.h

```
/* Это заголовочный файл определяет регистры, отраженные в памяти ADSP-2101 */

#define SP1_Autobuf           *(int *) 0x3fef
#define SP1_RFSDIV            *(int *) 0x3ff0
#define SP1_SCLKDIV           *(int *) 0x3ff1
#define SP1_Control_Reg       *(int *) 0x3ff2
#define SP0_Autobuf           *(int *) 0x3ff3
#define SP0_RFSDIV            *(int *) 0x3ff4
#define SP0_SCLKDIV           *(int *) 0x3ff5
#define SP0_Control_Reg       *(int *) 0x3ff6
#define SP0_MultiTX0          *(int *) 0x3ff7
#define SP0_MultiTX1          *(int *) 0x3ff8
#define SP0_MultiRX0          *(int *) 0x3ff9
#define SP0_MultiRX1          *(int *) 0x3ffa
#define TSCALE                *(int *) 0x3ffb
#define TCOUNT                *(int *) 0x3ffc
#define TPERIOD               *(int *) 0x3ffd
#define DM_Wait_Reg           *(int *) 0x3ffe
#define System_Control_Reg    *(int *) 0x3fff

#define LENGTH 64
```

```
#define SP0_Autobuf *(int *) 0x3ff3
```

Программа `init2101.c` (см. Листинг 11) устанавливает регистры, отраженные в памяти. Для автобуферизации мы будем использовать регистр I2 для хранения указателя, и регистр M1 для хранения шага или пост-модификации. Запишите значение 0x25 по адресу 0x3FF3.

Листинг 11. init2101.c

```
/* Эта C подпрограмма инициализирует регистры, отраженные в памяти,
устанавливает SPORT0 для автобуферизации приема с использованием регистров I2 и
M1. SPORT0 инициализируется для внутренней синхронизации, нужны внутренние RFS и
TFS с нормальной шириной кадра, данные в формате u-law 8-разрядов. */

#include "mreg2101.h"

void init2101()
{
      SP1_Autobuf       = 0x0;
      SP1_RFSDIV        = 0x0;
      SP1_SCLKDIV       = 0x0;
      SP1_Control_Reg   = 0x0;
      SP0_Autobuf       = 0x0025;
      SP0_RFSDIV        = 255;
      SP0_SCLKDIV       = 3;
      SP0_Control_Reg   = 0x6b27;
      SP0_MultiTX0      = 0x0;
      SP0_MultiTX1      = 0x0;
      SP0_MultiRX0      = 0x0;
      SP0_MultiRX1      = 0x0;
      TSCALE            = 0x0;
      TCOUNT            = 0x0;
      TPERIOD           = 0x0;
      DM_Wait_Reg       = 0x0000;
      System_Control_Reg = 0x1018;
}
```

```
      SP0_Autobuf = 0x0025;
```

Так как мы выбрали регистр I2 для хранения указателя, мы должны использовать регистр L2 для хранения длины нашего кольцевого буфера. Вы не нуждаетесь и не должны резервировать L регистры, так как они резервируются автоматически, при использовании I регистра.

Теперь, когда мы определили регистры автобуферизации, мы должны зарезервировать указатель, для предотвращения использования его компилятором C. Ключ `-mreserved=i2` в командной строке резервирует регистр I2. Не пытайтесь таким образом зарезервировать регистр L2.

Теперь, когда мы инициализировали наши регистры для автобуферизации, необходимо создать буфер для данных последовательного порта. Область памяти должна представлять собой кольцевой буфер. Это обеспечит размещение компоновщиком буфера в подходящих границах. Лучшим способом определить кольцевой буфер является использование директивы ассемблера. Мы поместили ассемблерные директивы в файл `asm_fn.dsp` (см. Листинг 12), который содержит необходимые подпрограммы на языке ассемблера. Следующая директива объявляет кольцевой буфер `rx_buf_`.

Листинг 12. asm_fn.dsp

```
.MODULE asm_fn;
.VAR/DM/RAM/CIRC  rx_buf_[64];
.GLOBAL           rx_buf_;

.ENTRY add_;

add_:
      ar=ar+ay1;
      rts;
.ENDMOD;
```

```
      .VAR/DM/RAM/CIRC  rx_buf_[64];
      .GLOBAL           rx_buf_;
```

Добавление простой инструкции ассемблера в код C

Для добавления единственной строки ассемблерного кода в ваш C с GNU компилятором (G21 версии 5.0 и выше), используйте простую форму конструкции `asm()`. Встроенная инструкция IDLE выглядит следующим образом:

```
asm("idle;"); /* ждать прерывания */
```

Для внедрения более одной инструкции вы можете расположить их на одной строке, или поместить их на отдельных строках (что сделать код более читабельным) и использовать обратную косую черту (\) для отображения продолжения линии. Обратная косая черта (\) должна быть последним символом в строке, включая комментарии. Приведенный ниже код использует оператор `asm()` для инициализации указателя автобуферизации (I2) к началу буфера `rx_buf_` и регистра длины (L2) .

```
asm("i2=^rx_buf_; \
     l2=%rx_buf_;");
```

Следующий пример кода очищает очередь ожидающих прерываний записью значения 0x3F в регистр IFC, затем ждет окончания выполнения операции один цикл и размаскировывает прерывание по приему последовательным портом 0 записью значения 0x8 в регистр IMASK.

```
asm("ifc=0x3f;          /* очистить ожидающие прерывания */ \
     nop;               /* один цикл ожидания ifc */ \
     imask=0x8;");      /* размаскировать прерывание SPRT0 RCV */
```

Шаблоны ассемблерных конструкций

Как вы могли заметить, добавить ассемблерную инструкцию в C код достаточно просто. Сложность состоит в том, чтобы не пересечься с регистрами, используемыми рабочим окружением C. Конструкция `asm()` предоставляет сложную форму, которая позволяет C компилятору выбрать, какие регистры DSP будут задействованы. Шаблон для сложной формы конструкции `asm()` выглядит следующим образом:

```
asm ("template":
"constraint" (out_ops):
"constraint" (in_ops):
"clobber");
```

Смотрите *ADSP-2100 Family C Tools Manual* для получения полной информации об использовании сложной формы конструкции `asm()`. Большинство приложений не требуют гибкость, предоставляемой шаблонами – следующая глава показывает простой пример этому.

Добавление модулей на языке ассемблера

Лучшим способом добавления кода на языке ассемблера в C программу является создание отдельного модуля. Если они написаны для выполнения в рабочей модели C, объектные файлы, созданные с языка ассемблера могут быть скомпонованы с объектными файлами, созданными с языка C.

Для создания ассемблерной подпрограммы, которая будет вызываться из C, вам необходимо понять, как компилятор передает параметры и возвращает значения из функций.

При передаче параметров, компилятор записывает первый параметр в регистр AR, а второй в регистр AY1. Если число параметров превышает два, остальные аргументы записываются в стек C. Все 16-разрядные результаты возвращаются в AR, а 32-разрядные результаты записываются в регистры SR1 и SR0.

Давайте создадим небольшую подпрограмму, которая принимает два аргумента: сумму и слагаемое. Первый параметр передается через регистр AR, в то время как второй через регистр AY1. Мы можем воспользоваться этим и произвести сложение с помощью инструкции `AR=AR+AY1`.

Вызов функции выглядит следующим образом:

```
extern int add(int x, int y); /* прототип для функции сложения */

void main()
{
...
      sum=add(sum, j);
...
}
```

Ассемблерный код для этой подпрограммы выглядит следующим образом:

```
.module asm_fn;
...
/* другие функции и объявления */

.entry add_;
add_:
ar=ar+ay1;
rts;

.endmod
```

Для более больших ассемблерных подпрограмм, вы должны запоминать состояние ваших регистров.

Ассемблерные модули, которые инициализируют оперативные средства

Файлы `stack.dsp` (см. Листинг 13) и `2101_hdr.dsp` (см. Листинг 14) являются примерами ассемблерных модулей, которые ассемблируются и компонуются вместе с объектными файлами C кода. Скопируйте их в ваш рабочий каталог. Оба этих файла изменены для этого примера (см. предыдущие главы). Они используются для инициализации оперативных средств.

Листинг 13. stack.dsp (модифицированная версия)

```
/* Это модифицированная версия файла 2101_hdr.dsp, который находится в каталоге
$ADI_DSP\21XX\LIB\SRC. Он был изменен для уменьшения размера стека и скопирован
в рабочий каталог. */

.MODULE                         Stack_Declaration;
.VAR/DM/RAM                     stack[200], ____top_of_stack;
.GLOBAL                         ____top_of_stack;
.ENDMOD;
```

Листинг 14. 2101_hdr.dsp (модифицированная версия)

```
/* Это модифицированная версия файла 2101_hdr.dsp, который находится в каталоге
$ADI_DSP\21XX\LIB. Он был изменен для удаления библиотечного вызова прерывания
по приему SPORT0 и скопирован в рабочий каталог. */

.MODULE/ABS=0           ADSP-2101_Runtime_Header;
.ENTRY                  ___lib_prog_term;
.EXTERNAL               ___lib_setup_everything;
.EXTERNAL               main_;
.EXTERNAL               ___lib_int2_ctrl, ___lib_sp0x_ctrl, ___lib_sp0r_ctrl;
.EXTERNAL               ___lib_int1_ctrl, ___lib_int0_ctrl, ___lib_tmri_ctrl;
```

```
__Reset_vector:              CALL ___lib_setup_everything;
                             CALL main_; {запуск C программы}
___lib_prog_term:            JUMP ___lib_prog_term;
                             NOP;
__Interrupt2:                JUMP ___lib_int2_ctrl;NOP;NOP;NOP;
__Sport0_trans:              JUMP ___lib_sp0x_ctrl;NOP;NOP;NOP;
__Sport0_recv:               RTI; NOP; NOP; NOP;
__Interrupt1:                JUMP ___lib_int1_ctrl;NOP;NOP;NOP;
__Interrupt0:                JUMP ___lib_int0_ctrl;NOP;NOP;NOP;
__Timer_interrupt:           JUMP ___lib_tmri_ctrl;NOP;NOP;NOP;

.ENDMOD
```

Файл `stack.dsp` не содержит ассемблерных инструкций, он инициализирует буфер, используя директивы ассемблера. Мы уменьшили размер стека до 200 слов.

Файл `2101_hdr.dsp` является измененным заголовком исполняемой программы, который используется вместо стандартного. Необходимо указать этот заголовок в командной строке ключом `-runhdr 2101_hdr.dsp`. Мы модифицировали заголовок для упрощения вектора прерывания по приему SPORT0.

После инициализации программа main.c запускает бесконечный цикл и выполняет инструкцию IDLE, которая переводит процессор в режим ожидания прерывания. Когда `rx_buf` полный, вызывается прерывание приема последовательного порта 0. DSP автоматически переходит на вектор прерывания, выполняет инструкцию RTI (возврат из прерывания) и возвращается к инструкции после IDLE.

Файл `2101_hdr.dsp` содержит также автоматический вызов основной программы. Заметьте, что метки и переменные в C содержат подчеркивание (_), добавленное при использовании в коде ассемблера (`main` становится `main_`).

Так как внешний модуль содержит одну и более ассемблерную подпрограмму, все правила по передаче параметров и управлению стеком остаются в силе.

Построение файла системы примера

Теперь, когда мы сделали основную работу, остается скомпилировать наш пример системы. Во-первых, запустите построитель системы (*system builder*), описанной в файле `simple.sys` (см. Листинг 15) для создания файла `simple.ach`. Далее запустите G21 для трансляции вашего исходного кода в исполняемый. Во время процесса трансляции, C компилятор запускает несколько средств в следующем порядке: препроцессор C, компилятор C, препроцессор ассемблера, ассемблер, компоновщик. Исходя из этого, вы можете указать компилятору в командной строке, что необходимо скомпилировать ваши C модули, ассемблировать модули на ассемблере и скомпоновать все объектные файлы вместе. Будем использовать следующую командную строку в этом примере:

Листинг 15. simple.sys

```
.system simple_ach;
.adsp2101;
.mmap0;
.seg/rom/boot=0               boot_page_0[2048];
.seg/pm/ram/abs=0/code/data   int_pm[2048];
.seg/dm/ram/abs=14336/data    int_dm[1024];
.endsys;
```

```
g21 main.c init2101.c stack.dsp asm_fn.dsp -a simple.ach -g -save-temps
    -Wall -o output -map -mreserved=i2 -runhdr 2101_hdr.dsp
```

- Входные файлы `main.c`, `init2101.c` и `stack.dsp`.
- `-runhdr 2101_hdr.dsp` указывает заголовок исполняемой программы.
- `-a simple.ach` указывает файл архитектуры системы.
- `-mreserved=i2` резервирует регистры I2 и L2 для автобуферизации.
- `-g -save-temps` необходимы для использования отладчика C в программе моделирования.
- `-Wall` выводит все предупреждения на экран.
- `-o output -map` указывает имя выходного файла `output.exe` и создает файл распределения `output.map`.

1.6. Использование библиотечных функций C

(DSPatch #31)

Эта глава представляет использование и синтаксис библиотечных функций и макросов C.

Используя библиотечные вызовы и информацию, затронутая в последних двух главах, мы создадим пример программы, которая принимает блок данных через последовательный автобуферизированный порт, производит быстрое преобразование Фурье над этими данными, обрабатывает их, производит обратное преобразование и выводит результат. Программа представляет собой простой, но полезный пример техники цифровой обработки сигнала, которая может быть осуществлена на плате ADSP-2101 EZ-LAB.

Этот пример программы был скомпилирован с помощью программ средств разработки Analog Devices версии 5.01. Для определения номера версии ПО, которое вы имеете, наберите `ASM21 -H` в командной строке и проверьте номер версии. Если вы планируете серьезные разработки на C с использованием DSP Analog Devices и не имеете ПО версии 5.01, необходимо подумать об обновлении. Существующая версия 6.0 компилятора C Analog Devices (G21) имеет множество преимуществ над версиями 5.1 и 5.01. Для резюмирования возможностей GNU компиляторов версий 5.x и 6.0 смотрите сравнительные характеристики, приведенные ниже.

Какая разница между CC21 и G21?

Если у вас был опыт программирования с компилятором C ADDS-21XX-BUN (CC21), здесь приведены некоторые ключевые отличия:

G21 (новый)	**CC21 (старый)**
нет дробей	дробный тип
asm ( )	#pragma
G21 вызывает компоновщик	компоновщик вызывается отдельной командой
IEEE формат плавающей точки	нестандартный тип плавающей точки

C программы используют библиотечные функции для выполнения операций, которые не предусмотрены языком. Вызываемые C подпрограммы по настоящему написаны на ассемблере и включают в себя размещение памяти, цифровую обработку и математические функции. Не использование этих функций, означает, что вам предстоит более тяжелая работа по достижению оптимизированного кода. Давайте разделим библиотечные функции на две категории, те, которые созданы Analog Devices и те, которые созданы пользователями.

Существующие библиотечные функции

ПО средств разработки Analog Devices обеспечивает большинство основных функций C необходимых для программирования DSP. Эти функции, названные C библиотекой рабочих программ описаны в вашем руководстве *C Runtime Library Manual*.

Библиотека рабочих программ состоит из подпрограмм и макросов, определенных ANSI стандартом C и расширениями к ANSI C, сделанными Analog Devices.
Функции, которые имеют общее назначение, сгруппированы в заголовочные файлы. Список заголовочных файлов приведен ниже.

Заголовочные файлы библиотеки рабочих программ

Стандартные ANSI макросы и определения оперативных средств

Header	*Purpose*
error.h	обработка ошибок
stddef.h	стандартные определения
limits.h	ограничения
float.h	плавающая запятая

Стандартные ANSI функции

Header	*Purpose*
math.h	математические функции
signal.h	обработка сигнала
stdarg.h	переменные аргументы
ctype.h	управление символами
string.h	управление строками
stdlib.h	стандартная библиотека
locale.h	локализация
assert.h	диагностика
setjump.h	нелокальные переходы
stdio.h*	ввод/вывод

Расширения C библиотеки программ

Header	*Purpose*
asm_sprt.h	макросы языка ассемблера
circ.h	подпрограммы кольцевого/автобуфера
filters.h	DSP фильтры
ffts.h	БПФ
float.h	плавающая запятая
sport.h	управление последовательным портом

* содержит только EOF символ, так как стандартный ввод/вывод не имеет смысла во встроенных приложениях.

Для примера EZ-LAB мы будем использовать функции из заголовочных файлов `fft.h`, `signal.h` и `sport.h`.

Заголовочный файл `sport.h` новый для ПО версии 5.01. Он обеспечивает C макросы для запуска, останова, чтения и записи для обоих последовательных портов на ADSP-2101.

Листинг 16. init2101.c

```
/* Эта C подпрограмма инициализирует регистры, отраженные в памяти ADSP-2101,
устанавливает SPORT0 для автобуферизации с использованием регистров I2,M1.
SPORT0 инициализируется для внутренней синхронизации SCLK, необходима внутренняя
синхронизация кадров приема и передачи с нормальной шириной кадра, формат данных
8-разарядов u-law. */

#include "mreg2101.h"

void init2101()
{
      SP1_Autobuf       = 0x0;
      SP1_RFSDIV        = 0x0;
      SP1_SCLKDIV       = 0x0;
      SP1_Control_Reg   = 0x0;
      SP0_Autobuf       = 0x06a7;
      SP0_RFSDIV        = 255;
      SP0_SCLKDIV       = 3;
      SP0_Control_Reg   = 0x6927;
      SP0_MultiTX0      = 0x0;
      SP0_MultiTX1      = 0x0;
      SP0_MultiRX0      = 0x0;
      SP0_MultiRX1      = 0x0;
      TSCALE            = 0x0;
      TCOUNT            = 0x0;
      TPERIOD           = 0x0;
      DM_Wait_Reg       = 0x0000;
      System_Control_Reg= 0x0;
}
```

Для использования SPORT0 и автобуферизации вам необходимо сделать следующее:

1. Инициализировать управление SPORT0 и регистры для автобуферизации (см. Листинг 16)
2. Размаскировать прерывание по приему SPORT0
3. Инициализировать указатели для автобуферизации (I2 и I3)
4. Разрешить SPORT0
5. Записать начальное значение в SPORT0 для запуска автобуферизованной передачи.

Пункты 4 и 5 могут быть реализованы с помощью макросов из `sport.h`

Функция `sport_write()` записывает значение в указанный буфер передачи последовательного порта. Функция `sport_start()` разрешает использование последовательного порта путем установки соответствующего бита в регистре управления.. Синтаксис функций из sport.h указан ниже:

```
#include <sport.h>
sport_write(0,0);
sport_start(0);
```

Для обработки данных буфера в C мы будем использовать функции `memset()` и `memset()` из заголовочного файла `string.h`. Функция `memset()` позволяет инициализировать буферы некоторой величиной. Это может быть полезным, если мы хотим установить значения ячеек буфера в ноль или передать автобуфер и мнимую часть на вход БПФ. Функция memcpy() позволяет передать содержимое одного буфера в другой. Мы будем использовать буфер `rx_buf[16]` для приема входных данных последовательного порта. После этого необходимо передать содержимое в буфер `real_time_buf[16]`, который будет содержать реальную часть для БПФ, а `rx_buf[16]` будет свободным для приема следующего блока данных. Синтаксис этих функций приведен в Листинге 17.

Листинг 17. Функции из string.h

```
#include <string.h>
memset(tx_buf,'\0',N);
memset(imag_time_buf,'\0',N);
memset(real_freq_buf+N-3,'\0',N-3);
memset(imag_freq_buf+N-3,'\0',N-3);
memcpy(real_time_buf,rx_buf,16);
memcpy(tx_buf,real_time_buf,16);
```

Мы будем также использовать заголовочный файл `fft.h` для выполнения быстрого преобразования Фурье и его инверсии (обратного преобразования). Заголовочный файл `fft.h` исключен из библиотеки рабочих программ датированной 11/1993. Для получения дополнительной информации обратитесь к замечанию, прилагаемому к ПО версии 5.01.

Заголовочный файл `fft.h` содержит функции преобразования для выборок размером 8, 16, 32, 64, 128, 256, 512, 1024. Для нашего примера выбрано 16-точечное преобразование и его обратная форма. Синтаксис функций из файла `fft.h` показан ниже:

```
#include <ffts.h>
fft16(real_time_buf,
      imag_time_buf,
      real_freq_buf,
      imag_freq_buf);
ifft16(real_freq_buf,
      imag_freq_buf,
      real_time_buf,
      imag_time_buf);
```

Небольшой урок по теории дискретизации DSP

Последовательный кодек (который включает А/Ц и Ц/А преобразователи) дискретизирует вход микрофона , включенный в разъемы на плате EZ-LAB. Этот 8-разрядный кодек дискретизирует данные с частотой 8000 Гц.

Так как 4 КГц является максимальной частотой, которая может быть представлена из дискретизации в 8 КГц (по Найквисту), используемый диапазон включает частоты от 0 до 4000 Гц. При этом кодеке, область частот, подлежащих фильтрации встроенной фильтрации, включает диапазон от 200 Гц до 3400 Гц. Для нашего примера этого достаточно, так как частота голоса человека лежит в этих пределах.

DSP принимает дискретные значения от кодека через последовательный порт. 16 таких значений накапливаются автобуфером последовательного порта.

Эти 16 значений будут `real_time_buf[16]` для 16-точечного БПФ. Мы выполняем комплексное БПФ только на реальной части, поэтому мнимая часть будет обнулена.

Говоря проще, БПФ определяет наличие определенных частот в сигнале, представленном во временной области. 16-точечное БПФ делит диапазон от 0 до 4 КГц на 8 «интервалов». Значение на каждом интервале через 500 Гц будет представлено комплексным числом.

Увеличение размера БПФ будет увеличивать число интервалов и , соответственно, разрешение в диапазоне от 0 до 4 КГц. 1024-точечное БПФ поделит диапазон на 512 интервалов. В этом случае разрешение составит 8 КГц. Дальнейшее увеличение размеров БПФ даст еще более лучшее разрешение, но займет значительно большее время преобразования и количество памяти.

16-точечное БПФ выбрано в этом примере, потому что будет использоваться только внутренняя память DSP для работы с платой EZ-LAB, а также, потому что оно демонстрирует библиотечный вызов и позволяет увидеть эффект цифровой обработки сигнала

Листинг 18. 2101_hdr.dsp (модифицированная версия)

```
/* Это модифицированная версия файла 2101_hdr.dsp, который находится в каталоге
$ADI_DSP\21XX\LIB. Он был изменен для удаления библиотечного вызова прерывания
по приему SPORT0 и скопирован в рабочий каталог. */

.MODULE/ABS=0           ADSP-2101_Runtime_Header;
.ENTRY                  ___lib_prog_term;
.EXTERNAL               ___lib_setup_everything;
.EXTERNAL               main_;
.EXTERNAL               ___lib_int2_ctrl, ___lib_sp0x_ctrl, ___lib_sp0r_ctrl;
.EXTERNAL               ___lib_int1_ctrl, ___lib_int0_ctrl, ___lib_tmri_ctrl;
__Reset_vector:             CALL ___lib_setup_everything;
                            CALL main_; {запуск C программы}
___lib_prog_term:           JUMP ___lib_prog_term;
                            NOP;
__Interrupt2:               JUMP ___lib_int2_ctrl;NOP;NOP;NOP;
__Sport0_trans:             JUMP ___lib_sp0x_ctrl;NOP;NOP;NOP;
__Sport0_recv:              RTI; NOP; NOP; NOP;
__Interrupt1:               JUMP ___lib_int1_ctrl;NOP;NOP;NOP;
__Interrupt0:               JUMP ___lib_int0_ctrl;NOP;NOP;NOP;
__Timer_interrupt:          JUMP ___lib_tmri_ctrl;NOP;NOP;NOP;

.ENDMOD
```

Листинг 19. mreg2101.h

```
/* Это заголовочный файл определяет регистры, отраженные в памяти ADSP-2101 */

#define SP1_Autobuf           *(int *) 0x3fef
#define SP1_RFSDIV            *(int *) 0x3ff0
#define SP1_SCLKDIV           *(int *) 0x3ff1
#define SP1_Control_Reg       *(int *) 0x3ff2
#define SP0_Autobuf           *(int *) 0x3ff3
#define SP0_RFSDIV            *(int *) 0x3ff4
#define SP0_SCLKDIV           *(int *) 0x3ff5
#define SP0_Control_Reg       *(int *) 0x3ff6
#define SP0_MultiTX0          *(int *) 0x3ff7
#define SP0_MultiTX1          *(int *) 0x3ff8
#define SP0_MultiRX0          *(int *) 0x3ff9
#define SP0_MultiRX1          *(int *) 0x3ffa
#define TSCALE                *(int *) 0x3ffb
#define TCOUNT                *(int *) 0x3ffc
#define TPERIOD               *(int *) 0x3ffd
#define DM_Wait_Reg           *(int *) 0x3ffe
#define System_Control_Reg    *(int *) 0x3fff

#define LENGTH 64
```

Построение примера системы с помощью компилятора G21

Теперь, когда мы в основном все сделали, остается только скомпилировать наш пример системы. Файл примера называется `main.c` (см. Листинг 20). Другие файлы, используемые в нашей системе, называются `init2101.c`, `mreg2101.h`, `2101_hdr.dsp`, `stack.dsp` и `ezlab.sys` (на основе который создается файл описания архитектуры `ezlab.ach`).

Листинг 20. main.c

```
/* Это основная программа примера обработки речевого сигнала для EZ-LAB.*/

#define N 16

#include "mreg2101.h"
#include <string.h>
#include <ffts.h>
#include <sport.h>

asm(".var/dm/ram/circ rx_buf_[16]; \
     .global rx_buf_; \
     .var/dm/ram/circ tx_buf_[16]; \
     .global tx_buf_;");
int real_time_buf[N];
int imag_time_buf[N];
int real_freq_buf[N];
int imag_freq_buf[N];

extern init2101();
extern int rx_buf[];          /* буфер в памяти данных для автобуферизации */
extern int tx_buf[];          /* буфер в памяти данных для автобуферизации */

void main()
{
/* инициализация регистров ADSP-2101, включая автобуферизацию SPORT0 */
      init2101();

/* очистка ожидаемых прерываний, установка IRQ и размаскирование прерывания */
      asm("ifc=0x3f;          /* очистка ожидающих прерываний */ \
           icntl=0x7;         /* установить пороги irq */ \
           imask=0x8;");      /* размаскировать прерывание SPRT0 RCV */

/* установить указатель и длину автобуфера */
      asm("i2=^rx_buf_; \
           l2=%rx_buf_;");
      asm("i3=^tx_buf_; \
           l3=%tx_buf_;");

/* инициализировать tx_buf[16] */
      memset(tx_buf,'\0',N);

/* запись нулей в tx0 для начала автобуферизации */
      sport_write(0,0);

/* разрешить SPORT0 */
      sport_start(0);

/* бесконечный цикл для СРВ */
      while (1)
      {
      /* ждать прерывания */
            asm volatile("idle;");

      /* копировать буфер sport0 в буфер реальной части для БПФ */
            memcpy(real_time_buf,rx_buf,16);
```

```
        /* обнулить мнимую часть для БПФ */
            memset(imag_time_buf,'\0',N);

        /* выполнить 16-точечное комплексное БПФ */
            fft16(real_time_buf,
                  imag_time_buf,
                  real_freq_buf,
                  imag_freq_buf);

        /* очистить верхние интервалы результата БПФ (НЧ-фильтр) */
            memset(real_freq_buf+N-3,'\0',N-3);
            memset(imag_freq_buf+N-3,'\0',N-3);

        /* выполнить 16-точечное ОБПФ */
            ifft16(real_freq_buf,
                   imag_freq_buf,
                   real_time_buf,
                   imag_time_buf);

        /* копировать результат ОБПФ в буфер передачи */
            memcpy(tx_buf,real_time_buf,16);
        }
}
```

Командная строка для создания исполняемого файла выглядит следующим образом:

```
g21 @allfiles -a ezlab.ach -g -save-temps -Wall -o output -map
    -mreserved=i2,i3 -runhdr 2101_hdr.dsp
```

ASCII файл `allfiles` содержит список всех входных файлов, `main.c, init2101.c, stack.dsp`. Описание ключей компилятора приводится ниже.

- `-runhdr 2101_hdr.dsp` указывает заголовок исполняемой программы.
- `-a ezlab.ach` указывает файл архитектуры системы.
- `-mreserved=i2,i3` резервирует регистры I2/L2 и I3/L3 для автобуферизации.
- `-g -save-temps` необходимы для использования отладчика C в программе моделирования.
- `-Wall` выводит все предупреждения на экран.
- `-o output -map` указывает имя выходного файла `output.exe` и создает файл распределения `output.map`.

Листинг 21. stack.dsp

```
/* Это модифицированная версия файла stack.dsp, который находится в каталоге
$ADI_DSP\21XX\LIB\SRC.  Он  был  изменен  для  уменьшения  количества  памяти,
отводимой под стек. */

.MODULE                     Stack_Declaration;
.VAR/DM/RAM                 stack[200], ____top_of_stack;
.GLOBAL                     ____top_of_stack;
.ENDMOD;
```

Запуск программы на ADSP-2101 EZ-LAB

Необходимо знать две вещи перед запуском примера на плате EZ-LAB. Во-первых, нужно запрограммировать EPROM. Для этого следует использовать ППЗУ распределитель `SPL21`. Для получения дополнительной информации обратитесь к руководству.

Во-вторых, необходимо сделать некоторые изменения платы перед запуском кода на EZ_LAB.

Выводы 4 и 5 коннектора последовательного порта должны быть соединены. Это соединит синхронизацию передаваемых кадров (TFS0) с синхронизацией принимаемых кадров (RFS0), и необходимо для правильной работы последовательного порта. *ADSP-2100 Family EZ Tools Manual* содержит детальное описание этой операции.

В следующей главе мы рассмотрим эффективное использование стандартного ввода/вывода.

1.7. Управление STDIO – пример для ADSP-2171

В этой главе мы обсудим, как управлять STDIO в C программах и поговорим о написании C кода для одного из последних DSP Analog devices, ADSP-2171.

STDIO.H является заголовочным файлом ANSI C, который определяет типы и макросы для управления вводом/выводом на уровне потоков для C системы. Например, широко используемая функция printf находится в библиотеке STDIO.
STDIO.H имеет несколько вариантов применения при написании C кода для PC компиляторов C. Однако, во встроенных системах очень редко необходим вывод на экран потока информации или прием символов с клавиатуры. Поэтому STDIO.H не поддерживается в ПО Analog Devices. Имеется файл STDIO.H, включенный в ПО ADDS-21XX-SW-PC, но он пуст и содержит только символ EOF.

Большинству программистов, пишущих программы для процессоров семейства ADSP-2100, не нужна библиотека STDIO. Поэтому, существует два случая, когда необходимо иметь функции STDIO в вашем коде.
Во-первых, вы можете проверять производительность предварительного C кода на ADSP-21xx, и ваш код уже имеет несколько потоковых функций ввода/вывода (`scanf`, `printf`). Вы должны, вероятно, вручную удалять или комментировать все экземпляры функций потокового ввода/вывода.

Во-вторых, вы можете хотеть писать и тестировать вашу программу с использованием PC компилятора (например, Borland Turbo C++) перед тем, как компилировать код с помощью ПО средств разработки Analog Devices. Вы можете использовать `printf` для вывода сообщений об ошибках и состоянии системы.

В любом случае, у вас есть два варианта действий. Можно, конечно, удалить все вызовы `printf` и других функций перед запуском компилятора G21. Другим способом является редактирование пустого файла STDIO.H (или создание нового) и добавление следующего кода для функций потокового ввода/вывода, необходимых вашей системе.

```
#define printf myprintf
int myprintf (char *fmt,...)
{return 1;}
```

Эти изменения, которые вы включили в новый заголовочный файл STDIO.H, позволяют игнорировать все вызовы функции `printf()`. Функция myprintf вызывается вместо стандартной функции `printf()`. Если необходима какой-либо вид обработки , функция `myprintf` может быть дополнена ля выполнения этого. Такая обработка может включать, например, подачу напряжения на Ц/А преобразователь. После модификации STDIO.H вы можете скомпилировать вашу C программу.

Определение `printf` новым символьным именем `myprintf` необходимо только в случае, если имеется настоящее объявление `printf`. Если его нет, можно пропустить директиву `#define` и использовать имя printf в строке с `int`.

Однако имеются некоторые обстоятельства, которые следует учесть. Во-первых, так как функции потокового ввода/вывода игнорируются, ваша программа не будет работать так, как изначально задумывалось. Если вы хотите ввод/вывод DSP, необходимо работать с функциями заголовочного файла `sport.h`, который управляет последовательными портами, или использовать свои собственные функции.

Во-вторых, хотя функция `printf` игнорируется, она остается вызовом, поэтому ваш код будет избыточным, включая сохранение и восстановление регистров для каждого вызова функции. С точки зрения производительности, было бы уместно убрать все вызовы `printf`. Если вы не используете ключ `–msmall-code`, избыточность будет составлять несколько циклов. Давайте посмотрим пример, иллюстрирующий это.

Программа `test.c` написана для процессора ADSP-2171. Основная программа использует встроенные ассемблер для проверки бита в регистре AR.

Подпрограмма error_routine использует функцию `printf` для вывода сообщения об ошибке на экран (полезно для отладки на PC) и выводит шестнадцатеричное число 0xDEAD на параллельный порт (полезно при использовании логического анализатора).

Тестирование бита в основной программе можно легко реализовать и на C, и на самом деле, вы не будете использовать встроенный ассемблер при компиляции с использованием PC средств. Приводимое использование ассемблера иллюстрирует новую инструкцию, доступную в ADSP-2171.
Так как ADSP-2171 поддерживает инструкции недоступные в других процессорах семейства ADSP-21xx (например, побитовая обработка, возможность возведения в квадрат и др.), нужно использовать специальный ключ при ассемблировании.

Порт `log_ana_port` описывается в файле архитектуры системы. Следующая инструкция добавляется в файл описания системы `2171.sys`.

```
.port/dm/abs=0x2000
log_ana_port_;
```

Подчеркивание добавлено к имени порта, так как мы будем использовать эту метку и в ассемблере, и в C. Порт необходимо объявить в ассемблерном модуле. Скопируйте файл `2171.dsp` в ваш рабочий каталог из каталога `C:\adi_dsp\21xx\lib` и добавьте следующие строки:

```
.PORT log_ana_port_;
.GLOBAL log_ana_port_;
```

Эти директивы объявляют порт и делают его глобальным. Это можно сделать в любом ассемблерном модуле. Файл `2171_hdr.dsp` был выбран для этой цели, так как он, скорее всего, потребует изменений для вашей системы. Теперь имя порта доступно и может использоваться в C:

```
extern volatile int
log_ana_port;
```

Следует заметить, что подчеркивание, как последний символ, не используется в синтаксисе C. Компилятор при генерации ассемблерного кода добавляет подчерк к любому символьному имени, используемого в C. Необходимо помнить об этом при использовании смешанного кода. Так как компилятор G21 может выполнять компиляцию, ассемблирование и компоновку вашего кода, мы проведем это отдельными действиями.

Командный файл `build.bat` содержит последовательные шаги, необходимые для компиляции C программы под процессор ADSP-2171. Во-первых, необходимо создать файл архитектуры системы, используемый далее на этапе компоновке. Во-вторых, наш файл `2171_hdr.dsp` ассемблируется программой. На третьем шаге компилируется C программа. Ключ `-S` (это должна быть прописная S) останавливает G21 перед этапом ассемблирования. Далее запускается C препроцессор, CPP. Это необходимо при использовании директив препроцессора в вашем коде, например, `asm («#include <def2171.h»)`. После препроцессора запускается ассемблер ASM2. Не забудьте про ключ `-2171`. Это необходимо при использовании инструкций процессора ADSP-2171, например, TSTBIT. Наконец, можно скомпоновать объектный файл `test.obj` для создания исполняемого файла (`output.exe`).

Листинг 22. Программа для ADSP-2171 с использованием STDIO

```
/* Это пример программы для ADSP-2171 с использованием STDIO */

#include "stdio.h"

extern volatile int log_ana_port;

void main()
{
      while (1)                    /* бесконечный цикл */
      {

      /* этот код очищает регистр AR (для проверки наличия ошибки), тестирует
         бит и вызывает подпрограмму обработки ошибок */

      asm ("ar=0; \
            af=tstbit 5 of ar; \
            if eq call error_routine_; ");

      asm ("idle;");          /* ждать прерывания */
      }
}

void error_routine ()
{
      printf("Error Occurred");
      log_ana_port=0xDEAD;
}
```

Листинг 23. Файл архитектуры для ADSP-2171

```
.SYSTEM example_system;
.ADSP2171;
.MMAP0;
.SEG/ROM/BOOT=0               boot_mem[2048];

.SEG/RAM/PM/ABS=0/CODE/DATA   int_pm[2048];
.SEG/RAM/DM/ABS=0x3000        int_dm[2048];
.PORT/DM/ABS=0x2000           log_ana_port;
```

Листинг 24. build.bat

```
bld21 -c 2171.sys
asm21 2171_hdr.dsp -c -s
g21 test.c -S -a 2171 -save-temps -g -Wall
c:\adi_dsp\21xx\etc\cpp -P -undef -D__GNUC__=2 test.s test.is
c:\adi_dsp\21xx\etc\asm2 -c -s -o test test.is -2171
g21 -runhdr 2171_hdr.obj test -a 2171.ach -o output -map
```

2. Программа сквозной передачи для ADSP-21xx EZ-Lite

(DSPatch #35: Anatomy of a 2100 family C program)

Этот раздел подробно объясняет основы C программы для семейства процессоров ADSP-2100. Эта программа представляет собой простой пример для использования с ADDS-21XX-EZ-LITE. EZ-Lite является недорогой макетной (оценочной) платой с процессором ADSP-2181. Так как EZ-Lite поставляется вместе программным обеспечением, он не содержит компилятора. Поэтому предполагается, что у вас есть полная версия ПО Analog Devices (ADDS-21XX-SW-PC). В настоящее время доступна версия 6.0.

Основная программа называется `CTIP35.C`. Как вы можете видеть из листинга, отдельные линии пронумерованы. Эти строки поясняются в последующих разделах. При запуске `CTIP35.C` на EZ-Lite читает вход стерео кодека AD1847, названный LEFT_IN и RIGHT_IN и возвращает его на AD1847, записывая в LEFT_OUT и RIGHT_OUT. Вы можете изменить программу сквозной передачи, каким-либо образом обрабатывая входные данные перед выходом.

Файл `CTIP35.ZIP` включает в себя все системные файлы, необходимые для тестирования приложения на EZ-Lite. `CTIP35.ZIP` доступен на FTP Analog Device и включает в себя:

CTIP35.C
SIGNAL.H,
2181_HDR.DSP
TALK_47.DSP
BUILD.BAT
2181.ACH

Файл `TALK_47.DSP` является ассемблерной подпрограммой, которая вызывается в `CTIP35.C` и инициализирует интерфейс между ADSP_2181 и AD1847. Заголовочный файл `SIGNAL.H` включает в себя макросы для управления прерываниями. Файл `2181_HDR.DSP` включает в себя модифицированный заголовок исполняемой программы для процессора ADSP-2181. Командный файл `BUILD.BAT` ассемблирует файл `2181_HDR.DSP` и запускает компилятор C со всеми необходимыми ключами. Наконец, файл `2181.ACH` содержит описание архитектуры системы, которая используется на EZ-Lite.

ctip35.c

```
/* Эта программа может быть использована как оболочка для C программ,
запускаемых на плате EZ-LITE и являться примером C кода. */

3     #include <signal.h>
2     #define DM_Wait_Reg     *(int *) 0x3ffe

5     extern init_1847();
3     void new_sample();

4     asm(".var/dm/ram/circ rx_buf_[3];   /* статус  + данные L/R */\n"\
4         "\t.var/dm/ram/circ tx_buf_[3]; /* команда + данные L/R */\n"\
4         "\t.init tx_buf_: 0xc002, 0x0000, 0x0000; /* инициализация MCE */\n"\
4         "\t.global rx_buf_, tx_buf_;");

1     volatile int left_in, right_in;
1     volatile int left_out, right_out;
      int stat_flag;

      void main()
      {
5           init_1847();                  /* инициализация интерфейса 1847 */

4           asm("set fl1;");              /* установка светодиода на EZ-LITE */

2           DM_Wait_Reg=0x0fff;
```

```
3           interrupt(SIGSPORT0RECV, new_sample);

6           while(1){
6                 asm("idle;");           /* ожидание прерывания */

6                 right_out=right_in;     /* вернуть обратно */
6                 left_out=left_in;
            }
      }

3     void new_sample()
      {
4           asm("ena sec_reg;");
4           asm("ax0 = dm(rx_buf_+1);");  /* прием новых данных */
4           asm("ax1 = dm(rx_buf_+2);");
4           asm("dm(left_in_) = ax0;");   /* запись в память */
4           asm("dm(right_in_) = ax1;");

4           asm("ax0 = dm(left_out_);");  /* чтение данных */
4           asm("ax1 = dm(right_out_);");
4           asm("dm(tx_buf_+1) = ax0;");  /* передача на кодек */
4           asm("dm(tx_buf_+2) = ax1;");

4           asm("dis sec_reg;");
      }
```

2.1. Расширения к ANSI C компилятору

Analog Devices GNU C компилятор соответствует ANSI C стандарту. При этом имеются некоторые расширения к ANSI C языку, которые характеризуют ADSP процессоры. Инструкция:

```
volatile int left_in, right_in;
```

это пример расширения к типу данных `int`. Использование `volatile` принуждает DSP помещать переменные left_in, right_in в область памяти, так чтобы работать с данными через регистры. Другое расширение ANSI C это указание `pm`:

```
int pm coeff[10];
```

это пример создает буфер в памяти программ DSP для хранение коэффициентов. Без этого указания буфер разместился бы в памяти данных.

2.2. Доступ к регистрам, отраженным в памяти на C

DSP резервирует часть внутренней памяти для настройки регистров. Конфигурация последовательных портов, внутреннего памяти, IDMA, BDMA, состояния ожидания и др. осуществляется через запись в эти регистры, отраженные в памяти. Для реализации этого в C, используйте директиву #define для создания метки или указателя на область памяти. В нашей программе `CTIP35.C` мы инициализируем состояния ожидания для использования области памяти ввода/вывода:

```
#define DM_Wait_Reg *(int *) 0x3ffe
```

Далее в программе можно записать значение в эту область памяти используя инструкцию:

```
DM_Wait_Reg=0x0fff;
```

2.3. Управление прерываниями

DSP является процессором, управляемым прерываниями. Для программирования этих прерываний на C необходимо понять, какие прерывания доступны на вашем DSP. Например, ADSP-2181 имеет прерывания для последовательных портов, внешних сигналов, таймера, понижения питания и др. Если вы не хотите пользоваться руководствами и спецификациями, вы можете взять `CTIP35.C` за основу и пропустить этот раздел. Если вы хотите понять, как настраиваются прерывания, или научиться их настраивать прерывания для своей системы, идем дальше. Первым делом, необходимо подключить заголовочный файл `SIGNAL.H` в вашу программу:

```
#include <signal.h>
```

Файл `SIGNAL.H` включает макросы для каждого прерывания процессоров семейства ADSP-2100.

Заметьте, что файл `SIGNAL.H` включен в ваше программное обеспечение, поэтому вы можете не иметь последней версии этого файла, например, если используете ПО версии 5.1. Последняя версия всегда доступна на FTP Analog Devices.

Для CTIP35.C необходимо установить прерывание по приему последовательного порта 0. макрос выглядит следующим образом:

```
interrupt(SIGSPORT0RECV, new_sample);
```

Эта строка регистрирует функцию `new_sample` как новый обработчик прерывания для приема последовательного порта 0.

2.4. Внедрение ассемблерных инструкций в C код

Общеизвестно, что как только вы добавляете ассемблерную инструкцию в C код, вы приносите в жертву переносимость кода. При этом существуют некоторые инструкции, которые можно выполнить только на ассемблере. Создание кольцевого буфера может быть проведено так:

```
asm(".var/dm/ram/circ rx_buf_[3];");
```

Инструкция IDLE и доступ к системным регистрам, например, IMASK, должны также осуществляться посредством ассемблера. Основной формат вставки ассемблерного кода в C:

```
asm("ассемблерный код;");
```

Для включения нескольких строк ассемблера используйте формат:

```
asm("первая инструкция;" \
    "вторая инструкция;");
```

Обратная косая черта (\) используется последним символом (включая комментарии). Вы можете использовать символы `\n` и `\t` для создания переносов и табуляции в вашем файле листинга.

2.5. Вызов подпрограмм DSP

Другим способом добавить ассемблерный код в вашу C программу является помещение DSP кода в подпрограмму. Программа `TALK_47.DSP` содержит код для инициализации кодека AD1847. `TALK_47.DSP` включает подпрограмму `init_1847()`, которая вызывается из C. Необходимо помнить о добавлении символа подчеркивания к именам переменных и меток, которые инициализируются в C и доступны из ассемблера. DSP метка в `TALK_47.DSP` называется `init_1847_`.

2.6. Управление оперативными средствами

Большинство DSP программ основаны на внешних сигналах. Эта программа ждет прерывания по приему последовательного порта и производит обратную передачу. Инструкция WHILE используется для создания бесконечного цикла. Ассемблерная инструкция IDLE переводит процессор в режим ожидания прерывания. Необходимо успеть закончить ваш код до тех пор, пока наступит следующее прерывание. Например, если вы принимаете данные от AD1847 с частотой 16 КГц, у вас есть 1/16 КГц, или 62.5 мкс между прерываниями. При использовании 33 МГц ADSP-2181, это составит 2062.5 цикла DSP.

build.bat

```
7     asm21 2181_hdr -c -2181
7     g21 -a 2181 -o ctip35 -I. -mreserved=i2,i3 -runhdr 2181_hdr.obj ctip35.c
          talk_47.dsp -g -save-temps -mlistm
```

2.7. Компиляция программы

Перед использованием команды G21 для компиляции программы `CTIP35.C`, вам необходимо ассемблировать новый заголовок исполняемой программы для ADSP-2181. Используйте ключ `-c` для создания файла `2181_HDR.OBJ` с учетом регистра. Ключами, которые нужны только для отладки программы, являются: `-g`, `-save-temps`, и `-mlistm`.

После запуска командного файла для компиляции C кода, вы можете загрузить исполняемый файл `CTIP35.EXE` в EZ-Lite, используя программу монитор, которая работает под управлением ОС Windows.

3. Серия технических замечаний EE – руководства по программированию на C для процессоров ADSP-21xx

Эта серия тенических замечаний EE (Engineer-to-Engineer) задумывается как руководство для пользователей, программирующих DSP серии ADSP-21xx на C. Она состоит из нескольких примеров в противоположность целому проекту. Эти примеры преследуют цель изучить основные положения, позволяющие пользователю запустить их без необходимости в аппаратной части, работая только в программе моделирования (симуляторе). Цель этого руководства показать некоторые основы с соответствующими примерами, делая изучение теории полее понятным.

Мы рассмотрим следующие темы:

- Доступ к регистрам, отраженным и неотраженным в памяти, в C (EZ-Notes #01)
- Расширения языка: типы хранения в памяти, конструкции *asm* и *inline* (EZ-Notes #02)

Выбранным DSP для реализации примеров, чаще всего, является ADSP-2181. Совместимость кода процессоров семейства ADSP-21xx делает примеры адаптируемыми для любых других процессоров.

Примеры кода для этих EZ-Notes доступны для загрузки на FTP сайте под именем C_Ezxx.zip, где xx – номер соответствующего замечания.

Файл C_EZ01.zip включает в себя следующие файлы:

regs.c	основная программа
cdef2181.h	заголовочный файл, определяющий регистры, отраженные в памяти
regs.bat	командный файл для компиляции примера
2181.ach	файл архитектуры для ADSP-2181

Приведенные примеры пронумерованы через каждые пять строк для ссылки на комментарии, сделанные в дальнейшем текстовом описании.

Для получения дополнительной информации по программированию на C для процессоров семейства ADSP-21xx обращайтесь к C Tools Manual, ADDS-21XX-SW-PC Release Notes и другим EZ-Notes.

3.1. Доступ к регистрам, отраженным и неотраженным в памяти, в C.

Доступ к регистрам, отраженным в памяти

Регистры, отраженные в памяти, размещаются в зарезервированной области внутренней памяти DSP. Эти регистры используются для настройки внутреннего таймера, последовательных портов, режимов ожидания, программируемых флагов и др. Заголовочный файл `cdef2181.h` использует директиву препроцессора `#define` для назначения символьного имени адресу каждого регистра, отраженного в памяти, ADSP-2181. Этот заголовочный файл будет применяться почти в каждом примере, приводимых в этом техническом замечании. Символьные имена одинаковы с теми, которые приведены в файле `def2181.h`, включенного в ПО средств разработки. Для более легкой ссылки на эти имена, содержимое файла `cdef2181.h` приведено в этом замечании.

Команда

```
#define Prog_Flag_Data *(int *) 0x3fe5
```

создает метку для ячейки памяти DM(0x3FE5), определяя 0x3FE5 как указатель на integer и далее разыменовывая его. Это означает, что имя `Prog_Flag_Data` эквивалентно целому значению, содержащемуся в ячейке памяти 0x3FE5.

C команда создает

```
Prog_Flag_Data=0x0000;
```

записывает значение «0» в ячейку памяти 0x3FE5 и является эквивалентной применению ассемблерных инструкций:

```
AX0=0x0000;
DM(0x3fe5)=AX0;
```

cdef2181.h

```
/* Заголовочный файл назначает символьные имена регистрам отраженным в памяти */
/* DSP ADSP-2181 */

#define Sys_Ctrl_Reg                *(int *) 0x3fff
#define Dm_Wait_Reg                 *(int *) 0x3ffe
#define Tperiod_Reg                 *(int *) 0x3ffd
#define Tcount_Reg                  *(int *) 0x3ffc
#define Tscale_Reg                  *(int *) 0x3ffb
#define Sport0_Rx_Words1            *(int *) 0x3ffa
#define Sport0_Rx_Words0            *(int *) 0x3ff9
#define Sport0_Tx_Words1            *(int *) 0x3ff8
#define Sport0_Tx_Words0            *(int *) 0x3ff7
#define Sport0_Ctrl_Reg             *(int *) 0x3ff6
#define Sport0_Sclkdiv              *(int *) 0x3ff5
#define Sport0_Rfsdiv               *(int *) 0x3ff4
#define Sport0_Autobuf_Ctrl         *(int *) 0x3ff3
#define Sport1_Ctrl_Reg             *(int *) 0x3ff2
#define Sport1_Sclkdiv              *(int *) 0x3ff1
#define Sport1_Rfsdiv               *(int *) 0x3ff0
#define Sport1_Autobuf_Ctrl         *(int *) 0x3fef
#define Prog_Flag_Comp_Sel_Ctrl     *(int *) 0x3fe6
#define Prog_Flag_Data              *(int *) 0x3fe5
#define BDMA_Word_Count             *(int *) 0x3fe4
#define BDMA_Control                *(int *) 0x3fe3
#define BDMA_External_Address       *(int *) 0x3fe2
#define BDMA_Internal_Address       *(int *) 0x3fe1
#define IDMA_Control                *(int *) 0x3fe0
```

Следующая демонстрационная программа regs.c показывает использование заголовочного файла `cdef2181.h` для доступа к регистрам, отраженным в памяти. Она состоит из двух функций. Первая функция конфигурирует внутренний таймер ADSP-2181, который может использоваться через прерывание таймера (см. EZ-Notes #05). Другая функция устанавливает программируемый флаг ADSP-2181 в регистре флагов для вызова программного сброса EZ-Kit Lite.

regs.c

```
/* демонстрационная программа показывает доступ к регистрам, отраженным в
памяти, и регистрам неотраженным в памяти. */

#include "cdef2181.h"

      void set_timer(void);          /* прототипы используемых функций */
      void reset_kit(void);

      void main(void)
{
            set_timer();
            reset_kit();
            asm ("dis timer;");      /* отключение таймера */
}
```

```
      /* Подпрограмма reset_kit выполняет программный сброс EZ-KIT LITE */
      /* Необходимо: соединить вывод PF0 к цепи сброса с 0-Ом резистором */
      /* на расположение R28. */
15    /* Это реализуется через программирование флага PF0 */
      /* Одновременно показывается работа с флагами PF */

      void reset_kit(void)
      {
            Prog_Flag_Comp_Sel_Ctrl=0x7b01;     /* PF0=выход */
20          Prog_Flag_Data=0;                   /* PF0 низкий уровень */
      }

      void set_timer(void)
      {
            asm(²ifc=0xff;          /* очистка ожидаемых прерываний */ \
25          nop;²);                 /* задержка */

            Tscale_Reg =1;
            Tcount_Reg =2000;
            Tperiod_Reg =2000;

30          asm("imask=1;           /* размаскировывание прерывание таймера */ \
            ena timer;");           /* запуск таймера */
      }
```

Строка 1: заголовочный файл

Директива препроцессора `#include "cdef2181.h"` используется для установления доступа к файлу `cdef2181.h`. Это *пользовательский* заголовочный файл, в противоположность *системному* заголовочному файлу, который осуществляет сопряжение с операционной системой и содержит определения регистров, отраженных в памяти, для C программы.

Строки 19-20, 26-28: доступ к регистрам

Строки 19-20 программируют флаг PF0, записывая соответствующие значения в регистры, отраженные в памяти. Строки 26-28 конфигурируют внутренний таймер записью необходимых значений в регистр таймера, отраженный в памяти.

Доступ к регистрам, неотраженным в памяти

Регистры, неотраженные в памяти, являются специализированными DSP регистрам, например, IFC, IMASK, ASTAT и др. Доступ к этим системным регистрам может быть осуществлен только через ассемблерный код. Программа `regs.c` показывает пример внедренной ассемблерной инструкции в C программу.

Строки 24-30

Эти инструкции используют конструкцию `asm()`:

```
asm(²assembly instruction;²);
```

для соответствующего доступа к системным регистрам IFC и IMASK:

Обратная косая черта (\) используется для продолжения строки более чем одной ассемблерной инструкции в одной конструкции `asm()`. Она позволяет делить строки с различными инструкциями, делая код более читабельным и комментируемым. Символ (\) должен быть последним в строке, включая комментарии.

Следующий командный файл используется для компиляции программы:

regs.bat

```
g21 regs.c -a 2181 -g -o regs -save-temps -mlistm -v -Wall
```

- Эта командная строка управляет другим ПО средств разработки, выполняя процесс трансляции (C препроцессор, компиляция, ассемблирование и компоновка) для создания исполняемого файла `regs.exe`. Некоторые другие операции включают размещение памяти для стека и добавление заголовка исполняемой программы. Ключ `-v` позволяет контролировать эти операции, заставляя компилятор выводить команды, выполняемые на каждом этапе.
- Так как не указывается заголовок исполняемой программы, компилятор использует директиву ADSP2181 файла архитектуры `2181.ach` для определения типа процессора (=какой заголовок исполняемой программы использовать).

- `-a 2181.ach` указывает файл архитектуры системы.
- `-o regs` указывает имя выходных файлов `regs.*`.
- `-g -save-temps` необходимы для использования отладчика C в программе моделирования.
- `-mlistm` необходим для создания объединенного файла листинга.
- `-Wall` выводит все предупреждения на экран.

3.2. Расширения языка: типы хранения в памяти, использование конструкций *asm* и *inline*.

Расширения языка

Компилятор G21 поддерживает набор расширений к ANSI стандарту для языка программирования C. Эти расширения специфицируют ADSP процессоры и включают:

- Поддержку для разделенных памяти программ и памяти данных (ключевые слова `pm`, `dm`)
- Поддержку `inline` функций
- Поддержку `asm()` конструкции

Следующий код помогает понять использование этих расширений и сопровождается дальнейшим пояснением.

lang_ext.c

```
/* программа показывает использование расширений G21 к языку C */

1     #include "pointdef.h"          /* содержит определение указателей */

      int aux;
      static int dm x ;              /* размещение переменных в указанных */
      static int dm y ;              /* областях памяти */
5     static int pm z ;
      static int dm i[3]={10,20,30};

      DINT pointer;                  /* тип DINT определен в pointdef.h */
                                     /* для указателей на dm integer */
```

```
      static inline int add(void)   /* определение add() as inline */
      {
10          z=x+y;
            return z;
      }

      static inline void Set_Fl(short flag)     /* определение Set_Fl() */
                                                /* эта функция устанавливает */
                                                /* Flag 0, 1 или 2 */
      {
15          switch (flag)
            {
                  case 0: asm volatile("set fl0;");
                          break;
                  case 1: asm volatile("set fl1;");
20                        break;
                  case 2: asm volatile("set fl2;");
                          break;
            }
      }

25    static inline void reset_fl1(void)        /* сброс Flag 1 и счетчик */
                                                /* числа сбросов */
      {
            static int pm rst_fl1_count;
            rst_fl1_count++;
            asm volatile("reset fl1;");
30    }

      void main (void)
      {                       /* некоторые операции с указателями */
                              /* основаны на использовании ключевых слов */
                              /* pm/dm и файла pointdef.h */

            pointer=i;        /* указатель указывает на i[0]=10 */
            x = *pointer++;   /* x=10 ; указатель указывает на i[1]=20 */
35          aux=(*pointer)++; /* aux=20; i[1]=21; */
            y=i[1];           /* y=21 */
            add();

            Set_Fl(0);        /* вызов inline функций */
            asm volatile("reset fl0;");

40          Set_Fl(1);
            reset_fl1();

            asm volatile("toggle fl1;");
            reset_fl1();
      }
```

Поддержка двойной памяти

Два ключевых слова (pm, dm) поддерживают двойную организацию памяти (отдельные память программ и память данных, где память программ может содержать данные и код, а память данных только данные), модифицированную гарвардскую архитектуру процессоров ADSP-2100, в противоположность фон-неймановской архитектуре, где данные и код хранятся в одной области памяти. Ключевые слова (pm, dm) используются для определения расположения статических (или глобальных) переменных. Таким образом, имея явный доступ к области памяти программ (например, для чтения коэффициентов фильтра) и памяти программ, вы можете воспользоваться преимуществами DSP архитектуры, когда за один цикл можно выбрать два слова данных и одну инструкцию.

Строки 3-6

Эти операторы иллюстрируют размещение переменных в области памяти данных и памяти программ.

Некоторые правила использования ключевых слов следует соблюдать при двойной памяти:

- Без указания pm/dm, G21 по умолчанию использует память данных для размещения переменных, например, переменная aux (**строка 2**) будет размещена в памяти данных.
- Память программ является выделенной областью памяти для функций.
- Автоматические переменные (= локальные переменные внутри функции) располагаются в стеке, который находится в памяти данных. Для использования ключевого слова pm внутри функции, необходимо присвоить переменной статический класс памяти, указав соответствующий спецификатор (**строка 27**).

Определение указателей с использованием *pm*/*dm*.

Еще одно использование ключевых слов поддержки двойной памяти – определение указателя в пределах пространства двойной памяти. Например, следующий оператор:

```
int dm * pm pd;
```

объявляет pd как указатель, размещенный в памяти программ и указывающий на целое типа integer, размещенное в памяти данных. Программа lang_ext.c также использует определения указателей, включая (**строка 1**) пользовательский заголовочный файл pointdef.h, который содержит некоторые определения типов указателей. Этот файл может быть использован как пример, который может быть изменен для кода, применяющего указатели.

pointdef.h

```
typedef int pm * PINT;              /* PINT определяет тип указателя на
                                       integer, расположенного в pm */

typedef int * DINT;                 /* DINT определяет тип указателя на
                                       integer, расположенного в dm */
                                    /* т.ж.: typedef int dm * DINT; */

typedef float pm * PFLOAT;          /* PFLOAT определяет указатель на
                                       float, расположенного в pm */

typedef float * DFLOAT;             /* DFLOAT определяет тип указателя на
                                       float, расположенного в dm */
                                    /* т.ж.: typedef float dm * DFLOAT; */
```

Строка 7

```
DINT pointer;
```

Показывает пример использования типа DINT из файла pointdef.h для объявления переменной pointer как указателя, расположенного в dm (по умолчанию) и указывающего на целое типа integer, которое размещается в памяти данных. Возникает вопрос: почему нельзя определить область памяти (pm/dm) указателя внутри typedef? Это невозможно. Компилятор выявит ошибку на такой конструкции:

```
typedef int * pm DINTP; /* попытка объявить pm указатель на dm integer */
```

Тип памяти указателя должен быть определен программистом, например,

```
#include ²pointdef.h²

DINT pm dd1;              /* = int * pm dd1 */
                          /* dd1 это pm указатель на dm integer */
```

Строки 33-36

В этих строках показаны некоторые операции с указателями. Для завершения этой части, следует заметить, что так как функции занимают память программ, указатели на функции всегда указывают на область памяти программ.

Поддержка встроенных функций (*inline*)

Ключевое слово `inline` указывает G21 интегрировать код функции, объявленной как встроенной в код вызывающей программы. Это делает выполнение более быстрым, избегая задержки вызова функции.

Строки 8, 13 и 25 примера C программы показывают определение функций, использующих ключевое слово `inline`. Заметьте, что определение предшествует вызовам функций. Это предупреждает вызов функции вместо ее внедрения.

Как inline функции влияют на код можно увидеть в программе моделирования в окне C кода (CBUG) и кода ассемблера (окно Program Memory).

Альтернативой использования встроенных функций является явное указание в командной строке компилятора ключа `–finline-functions`. Этот ключ вызывает интеграцию всех простых функций в код вызывающей программы. Какие функции являются простыми решает сам компилятор.

Поддержка встроенного ассемблера (*asm*)

Конструкция `asm()` позволяет внедрять ассемблерные инструкции в C код и может быть полезна для вызова ассемблерных инструкций, которые не могут быть вызваны совсем или эффективно на C. Приведем некоторые примеры:

```
asm(²ifc=0xff²)                /* доступ к регистрам, неотраженным в памяти
                                  осуществляется только на ассемблере/
                               /* обратитесь к EZ Notes #01 */

asm volatile (²set fl0;²);     /* строка 17 lang_ext.c */
                               /* установка fl0 является специальной
                                  инструкцией процессора */
```

Расширение `volatile` используется для предотвращения удаления или перемещения инструкции. Отдельные переменные могут назначены с помощью `asm()`. Следующий пример назначает регистр AX0 переменной `x`:

```
register int x asm(²AX0²);
```

Более сложные формы конструкции asm(), позволяющие указывать операнды ассемблерных инструкций с помощью C выражений рассматриваются в *EZ Notes #03*, темой которой является *Interfacing C and Assembly Language – Интерфейс языка C и ассемблера*.
Следующий командный файл используется для компиляции примера программы:

lang_ext.bat

```
g21 lang_ext.c -a lang_ext -g -v -Wall -save-temps -mlistm
    -fkeep-inline-functions –o lang_ext -map
```

- `-map` указывает компоновщику создавать файл распределения памяти. Это полезно при отладке.

- `-fkeep-inline-functions` указывает компилятору выводить разные версии времени исполнения для встроенных функций. Это необходимо, например, для установки точек останова внутри функций, так как G21 выдает собственный ассемблерный код для функций (что компилятор C не делает).

После компиляции примера, его можно запустить с использованием C отладчика (CBUG). Руководство по применению CBUG включено в *C Tools Manual and the Release Notes*.

3.3. Пространство ввода/вывода ADSP-218x DSP – конвейеризация переменных C на порт ввода/вывода

Если необходимо послать значение переменной C в порт ввода/вывода, отраженный в памяти, используйте встроенный ассемблер, как показано ниже.

Доступ к пространству ввода/вывода

```
int myvar;
/* переменная, объявленная глобальной */

void main(void )
{
      myvar=256 ;
      asm(".external myvar_;");
      asm("ax0=dm(myvar_);");
      asm("IO(0x0100)=ax0;");
}
```

Код, приведенный в листинге, передает значение переменной `myvar` в порт ввода/вывода по адресу 0x1000. Заметьте, что переменная объявлена глобальной, чтобы сделать ее доступной ассемблерному коду. Необходимо также объявить переменную внешней до попытки доступа к ней. Наконец, заметьте, что для хранения значения был выбран регистр AX0. Этот регистр является рабочим регистром, используемым C компилятором, и отсюда, безопасным в применении.

Таким же образом можно использовать встроенный ассемблер для передачи значения от порта ввода/вывода в переменную C.

4. DSPatch вопросы/ответы по программированию на C

4.1. Загрузочные страницы в C

Вопрос: Имеется ли способ создать некоторое число C программ и хранить их в различных загрузочных страницах в загрузочной ППЗУ? Я знаю, что имеются директивы ассемблера для указания загрузочной страницы, но я не видел ничего похожего в C компиляторе?

Ответ:
Процессоры ADSP-2101, ADSP-2105, ADSP-2111 и ADSP-21msp50 имеют возможность загрузки в одну из восьми страниц внешней ППЗУ, например 27512. Для создания C программ, которые генерируют код для размещения в различных страницах, следуйте следующим шагам:

1) С каждой C программой должен быть уникальный заголовок исполняемой программы. Стандартный заголовок, поставляемый с ПО средств разработки приведен ниже (с одним добавлением, спецификатором BOOT=0).

Заголовок исполняемой программы ADSP-2101

```
.MODULE/BOOT=0 ADSP2101_Runtime_Header;

.EXTERNAL   main_, ____top_of_ram, ____top_of_stack;
.EXTERNAL   __lib_setup_heap, __lib_setup_errno, __lib_setup_exit;
.EXTERNAL   __lib_setup_interrupts_2101;
.EXTERNAL   __lib_restart_vector
.ENTRY __lib_code_start;

__lib_code_start:

      IMASK=0;                      {отключить все прерывания}
      L0=0; L1=0; L2=0; L3=0;       {все регистры длины должны быть}
      L4=0; L5=0; L6=0; L7=0;       {  установлены в 0}
      M0=0; M1=1; M2=0; M3=-1;      {все регистры модификации установлены в}
      M5=1; M6=0; M7=-1;            {  различные значения}
      M4=^____top_of_stack;         {указатель фрейма = вершина стека}
      I4=^____top_of_stack;         {указатель стека  = вершина стека}
      CALL __lib_setup_heap;
      CALL __lib_setup_interrupts_2101;
      CALL __lib_setup_errno;
      CALL __lib_setup_exit;
      CALL main_;
__lib_code_hang:
      JUMP __lib_code_hang;
.ENDMOD;
```

Заголовок исполняемой программы находятся в каталоге ADI_DSP\LIB\SRC. Имена файлов для различных DSP процессоров следующие:

2105_HDR.DSP
2101_HDR.DSP
2111_HDR.DSP
2150_HDR.DSP

Первая C программа, которая должна размещаться в загрузочной странице 0, может использовать стандартный заголовок, сопровождаемый спецификатором /BOOT=0, добавленным к директиве MODULE. Компоновщик по умолчанию использует стандартный файл заголовка. Для любых других дополнительных C программ, заголовок исполняемой программы должен быть скопирован в отдельный файл, мы назовем его HDR1.DSP, и претерпеть следующие изменения:

.MODULE/BOOT=1 ADSP2101_Runtime_Header1;

(Заметьте, что спецификатор /BOOT=1 используется для C программы, которая должна размещаться в первой загрузочной странице. Каждый модуль должен сопровождаться различной цифрой, добавляемой к именам, как показано в следующих четырех строках заголовка.

```
.EXTERNAL   main1_, ____top_of_ram, ____top_of_stack;
.ENTRY      __lib_code_start1;

__lib_code_start1:
      CALL main1_;
```

По такому принципу создается необходимое число файлов заголовков исполняемой программы.

2) Ассемблируйте программы заголовков по одной, включая стандартный, используя ключ зависимости от регистра.

```
ASM21 2101_HDR.DSP -c
ASM21 HDR1 -c
и т.д.
```

3) Создайте C программы для каждой из загрузочных страниц, используя следующие имена для основных программ:

```
main(), main1(),
main2(), main3(),
и т.д.
```

4) Скомпилируйте C программы по одной, используя ключ загрузочной страницы, как показано ниже. Спецификатор в директиве MODULE должен совпадать с номером, указанном в ключе.

```
CC21 prog0 -b0
CC21 prog1 -b1
CC21 prog2 -b2
и т.д.
```

5) Скомпонуйте скомпилированные C программы с заголовками исполняемой программы:

```
LD21 prog0 prog1 prog2 HDR1 HDR2 -e C_CODE -a C_SYS -c -lib
```

Заметьте, что C программа PROG0 компонуется со стандартным заголовком. Файл описания архитектуры системы также должен включать объявления загрузочных страниц.

4.2. Командная строка C компилятора

Вопрос: В моем приложении я указываю слишком много файлов в командной строке компилятора. Предел командной строки в ДОС составляет 128 символов. Имеется ли более рациональный путь указания входных файлов для G21/G21k?

Ответ: Лучшим способом указания большого списка файлов для G21/G21k является использование параметра командной строки `@filename`. Эта возможность позволяет указывать в командной строке файл, содержащий список файлов, посылаемых компилятору. Формат файла состоит из указания одного имени (пути) файла в каждой строке. Пример:

```
g21 @files_all -a 21.ach
```

4.3. Проблемы G21 с результатами умножения

Вопрос: У меня есть проблемы с подпрограммой, написанной для ADSP-2101, когда я вызываю ее из C программы. Когда я моделирую ассемблерную подпрограмму или использую другие подпрограммы, все работает нормально. При этом когда я вызываю ассемблерную подпрограмму из C программы, результаты умножения не те, которые ожидаются?

Ответ: ADSP-2101 выполняет умножение в одном из режимов – дробном и целом. Режим управляется 4 битами в регистре MSTAT. В дробном режиме, процессор сдвигает результат умножения на один разряд перед записью в регистр результата. Этот сдвиг устраняет дополнительный знаковый бит, генерируемый, если оба входных операнда являются дробными знаковыми числами (часто называемыми форматом 1.15). В целочисленном режиме, сдвиг влево не происходит. При сбросе ADSP-2101 по умолчанию устанавливается в дробный режим.

C компилятор работает в целочисленном режиме и меняет режим процессора по умолчанию. Вы должны изменить режим на дробный перед выполнением любой операции умножения. Вставьте инструкцию DIS M_MODE в начале ассемблерной подпрограммы для изменения режима (очистка 4 бит в регистре MSTAT). Вставьте инструкцию ENA M_MODE в конце подпрограммы, перед возвращением в C программу. Инструкция ENA M_MODE устанавливает 4 бита в регистре MSTAT и меняет режим работы умножения в целочисленный.

4.4. Доступ к физическим адресам в C

Вопрос: Я использую C компилятор для разработки кода под систему ADSP-2101. Как я могу сохранить значения по физическим (абсолютным) адресам для настройки управляющих регистров, отраженных в памяти, или записать данные в периферию, размещенную во внешней памяти.

Ответ:
Приведенный ниже код показывает простой метод записи данных по физическим адресам:

```
#define DA_Conv   *(int *) 0x2000

/* объявление регистров, отраженных в памяти, для таймера ADSP-2101 */

#define tperiod   *(int *) 0x3FFD
#define tcount    *(int *) 0x3FFC
#define tscale    *(int *) 0x3FFB

main()
{
      /* запись значения 0x100 в ЦАП */
      DA_Conv=0x100;
      tperiod=300;
      tcount=0;
      tscale=300;
}
```

4.5. Использование портов ADSP-2100, отраженных в памяти данных, в C программе

Вопрос: У меня есть C программа, которую я хотел бы использовать для ADSP-2100. Я хотел бы узнать, как воспользоваться портом ввода/вывода, соединенным с ADSP-2100 в моей C программе?

Ответ: Любое символьное назначение, которое может быть сделано на языке ассемблера ADSP-2100, может быть проведено и в C программе, включая символьные ссылки на порты ввода/вывода.

Когда запускается C компилятор для ADSP-2100, ко всем меткам, используемых C программе, добавляется символ подчеркивания на ассемблерном выходе. Поэтому, ключевой особенностью ссылки на порт ввода/вывода, является имя порта (в ассемблерных модулях и файле описания системы) с добавленным подчерком, а использование имени в C программе без него. Следующий пример показывает это:

```
extern DA_OUTPUT;

main()
{
      int i;
      for (i=0; i<10; i++)
      DA_OUTPUT = i;
}
```

В C программе имя DA_OUTPUT объявлено внешним и не имеет символа подчеркивания. Так как конструкция `asm()` работает только внутри функции, определяется функция `dummy()`:

```
dummy()
{
      asm(".port DA_OUTPUT_;");
      asm(".global DA_OUTPUT_;)");
}
```

Порт ЦАП в этом примере назван DA_OUTPUT_. Заметьте, что имя порта указано в верхнем регистре и с добавленным символом подчерка.

4.6. Использование прерываний с C программами для ADSP-2100

Вопрос: Как я могу управлять прерываниями при использовании C программ для ADSP-2100?

Ответ:
Прерывания могут обрабатываться в C программе при следовании следующим шагам. Во-первых, заголовок исполняемой программы должен быть изменен для включения векторов прерывания. Инструкция RTI в стандартном заголовке должна быть изменена на инструкцию перехода, где место назначения перехода является первой инструкцией подпрограммы обработки прерывания. Если переход осуществляется на C подпрограмму, метка адреса должна содержать символ подчерка. Например, если переход осуществляется на функцию с именем `func`, то оператор перехода в заголовке исполняемой программы должна быть инструкция `JUMP func_`.

Прерывание или прерывания должны быть правильно разрешены в заголовке исполняемой программы. Установка управляющего регистра (ICNTL) и регистра маски прерываний (IMASK) должна быть произведена добавлением соответствующих инструкций в модуль заголовка.

Пример измененного заголовка исполняемой программы приведен ниже.

```
.MODULE/ABS=0/RAM RTH;
.EXTERNAL   main_;
.EXTERNAL   ____top_of_ram;
.EXTERNAL   int_rtn_;

      RTI;
      RTI;
      RTI;
      JUMP int_rtn_;

top:  M4=^____top_of_ram;
      I4=^____top_of_ram - 1;
```

```
        L0=0;
        L1=0;
        L2=0;
        L3=0;
        L4=0;
        L5=0;
        L6=0;
        L7=0;

        M0=0;
        M1=1;
        M2=0;
        M3=-1;
        M5=1;
        M7=-1;

        ICNTL=h#0F;
        IMASK=h#08;

        CALL main_;
        TRAP;
.ENDMOD;
```

Этот модуль должен быть ассемблирован с использованием ключа зависимости регистра. Следующий C код показывает пример основной программы, которая будет прерываться.

```
main()
{
        int i,k;
top:
        for (i=0; i<1000; i++)
        k=i;
        goto top;
}
```

Цикл повторяется до наступления прерывания. Процесс выполнения программы переводится в подпрограмму обработки прерывания и после того, как она завершится, возвращается обратно в основной цикл. Пример подпрограммы обработки прерывания приведен ниже:

```
int_rtn()
{
        asm(" MX0=5; ");
        asm(" MY0=10 ");
        asm(" MR=MX0*MY0(SS); ");
        asm(" DM(i4,m7)=MR1; ");
        asm(" RTI; ");
}
```

C программа должна быть скомпилирована и затем скомпонована. При компоновке ключ `-c` используется для корректного создания стека этапа исполнения. Пример командной строки:

```
G21 run_hdr c_func1 c_func2 -e my_prog -x -g -c -a my_sys
```

4.7. Проблемы с C препроцессором

Вопрос: У меня вопрос о правильном использовании команды .include в программе ADSP-2101. В моем ассемблерном модуле, когда я применяю команду:

```
.include
<c:\adi_dsp\21xx\include\def2101.h>;
```

я получаю сообщение об ошибке:

```
asm21:
"c:\dsp\21xx\include\def2101.h",
line 0: illegal lhs '#'.
```

В чем проблема? Разве C препроцессор не распознает директивы #define во включаемых файлах?

Ответ:
Когда вы используете C компилятор для ассемблирования кода ADSP-2100, запускается C препроцессор. Проблема, которую вы видите, вызывается порядком, в котором случаются вызовы когда вы ассемблируете свой код. При ассемблировании случаются два вызова, один для C препроцессора, а другой для ассемблерного препроцессора. C препроцессор вызывается первым. Так как директива `.include` является ассемблерной, то она игнорируется C препроцессором. В то же время, при вызове ассемблерного препроцессора, файл `def2101.h`, который вы указываете, не содержит правильных ассемблерных директив. Ассемблерный препроцессор не распознает директиву `#define`, которая содержится в заголовочном файле. Решением может служить использование следующей директивы:

```
#include <filename>
```