Разработка программного кода для сигнальных процессоров TMS320C6000

Игорь ГУК
gii@scanti.ru

Статья открывает новый цикл, в котором будут показаны примеры реализации алгоритмов цифровой обработки сигналов (ЦОС) на базе цифровых сигнальных процессоров (ЦСП) фирмы TI. Цель предлагаемых материалов — показать технологию программирования ЦСП фирмы TI от постановки задачи на разработку программной реализации алгоритма ЦОС до получения бинарного кода для ЦСП, его отладки и оптимизации с помощью аппаратных и программных средств, предоставляемых фирмой TI. Предлагаемая программная модель реализации алгоритмов ЦОС учитывает требования стандарта eXpresDSP, внедряемого фирмой TI. Кроме того, модель предполагает максимально возможную переносимость разработанного программного кода между различными семействами ЦСП фирмы TI.

В ближайших выпусках цикла будут рассмотрены общие положения ЦОС, рекомендации по написанию программного кода алгоритмов ЦОС на примере фильтра с конечной импульсной характеристикой (в дальнейшем — КИХ-фильтр), использование интегральной среды разработки (ИСР) CCS для реализации КИХ-фильтра на ЦСП TMS320C6000, тестирование и отладка полученного кода с помощью CCS и отладочного модуля на основе ЦСП TMS320C6211. Использование младшего процессора из серии TMS320C6000 обеспечивает полную совместимость программного кода со всеми старшими ЦСП данного семейства.

Цифровая обработка сигналов (ЦОС) — это дисциплина, изучающая дискретные и цифровые сигналы, системы их обработки, а также цифровые процессоры, реализующие данные системы.

Сигнал — это изменение параметров среды распространения в зависимости от передаваемого сообщения, описываемое функцией времени. В качестве среды распространения могут выступать: электромагнитное поле (радиосигнал), воздух (звуковой сигнал), вода (гидроакустический сигнал), почва (сейсмический сигнал) и т. д.

В ЦОС различают четыре типа сигналов:

- ***Аналоговый*** (рис. 1, а) — это сигнал, непрерывный во времени и по значению. Описывается непрерывной (или кусочно-непрерывной) функцией времени $x(t)$. Аргумент и функция могут принимать любые значения из некоторых произвольных интервалов: $t_{min} \le t \le t_{max}$, $X_{min} \le x(t) \le X_{max}$.
- ***Дискретный*** (рис. 1, б) — это сигнал, дискретный во времени и непрерывный по значению. Представляет собой последовательность чисел, называемых отсчетами. Описывается решетчатой функцией $x(nT)$, где $n = 0, 1, 2, 3…$ — номер отсчета, а T — интервал между отсчетами, называемый *периодом дискретизации*. Обратную величину 1/T называют *частотой дискретизации* $f_д$. Решетчатая функция определена только в моменты времени $t = nT$ и может принимать произвольное значение из некоторого произвольного интервала $X_{min} \le x(nT) \le X_{max}$.
- ***Цифровой*** (рис. 1, в) — это сигнал, дискретный во времени и квантованный по значению. Описывается решетчатой функцией, которая может принимать только конечное число значений из некоторого конечного интервала. Эти значения называются *уровнями квантования*, а соответствующая функция — *квантованной*.
- ***Цифро-аналоговый*** (рис. 1, г) — это сигнал, непрерывный во времени и квантованный по значению. Описывается непрерывной (или кусочно-непрерывной) функцией времени $x_ц(t)$, причем аргумент может принимать любые значения из некоторого интервала $t' \le t \le t''$, а сама функция — только конечное число значений из некоторого конечного интервала $x' \le x \le x''$, то есть является квантованной.

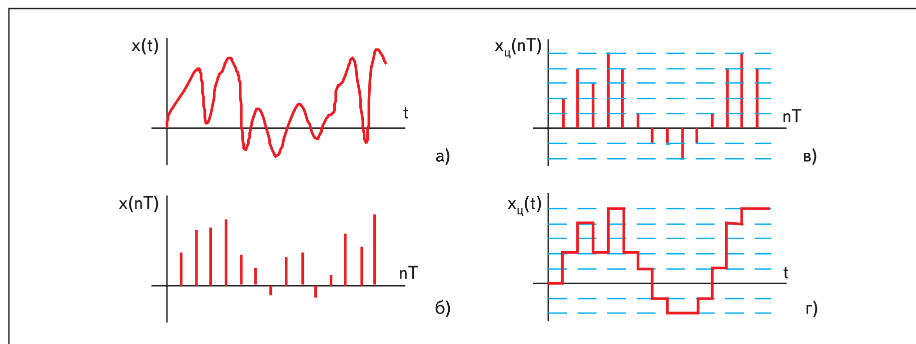

Рис. 1. Основные типы сигналов

В частотной области для описания сигналов используется преобразование Фурье. Оно представляет собой пару соотношений (функции прямого и обратного преобразования), которые устанавливают взаимнооднозначное соответствие между сигналом и спектром. Причем, под спектром понимается функция прямого преобразования Фурье, а под сигналом — функция обратного преобразования Фурье.

Обобщенная ***процедура ЦОС*** включает три этапа:

- Преобразование входного аналогового сигнала в дискретный сигнал.
- Обработка дискретного сигнала по заданному алгоритму цифровым сигнальным процессором (ЦСП) и формирование выходного дискретного сигнала.
- Преобразование дискретного сигнала в выходной аналоговый.

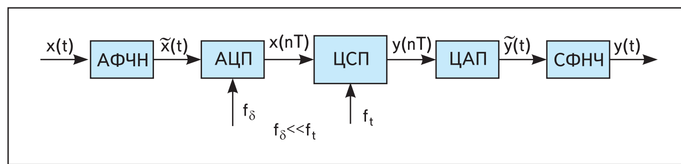

Рис. 2. Система цифровой обработки сигналов

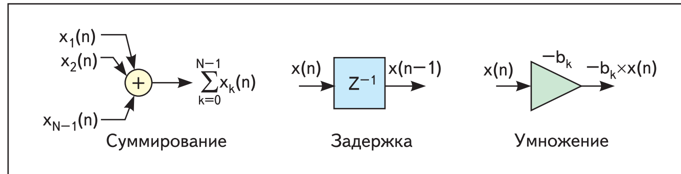

Рис. 3. Графическое изображение основных операций цифровой обработки сигналов

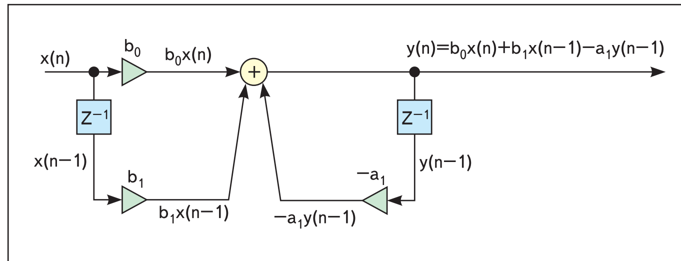

Рис. 4. Пример построения структурной схемы

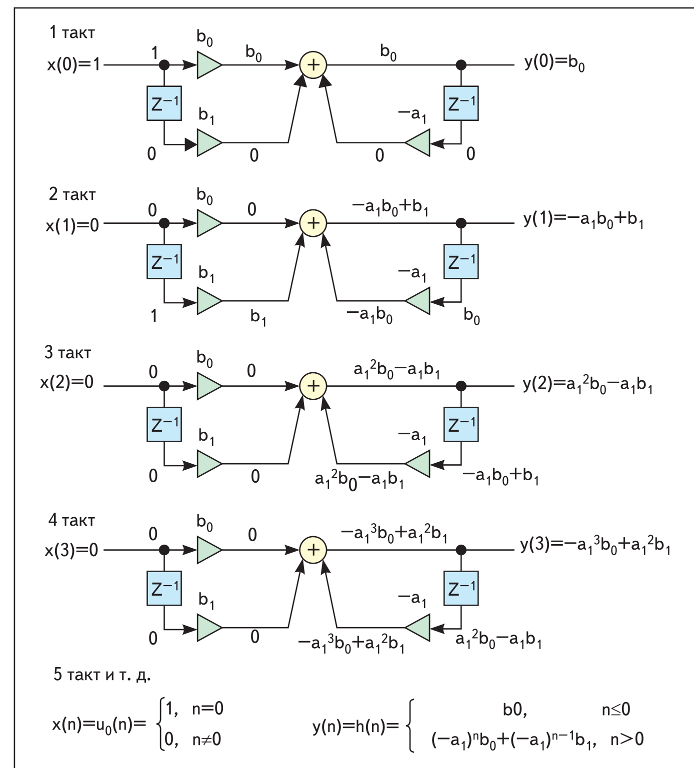

Рис. 5. Функционирование системы ЦОС

Система ЦОС, реализующая процедуру ЦОС, включает (рис. 2):

- Аналоговый антиэлайсинговый фильтр низкой частоты (АФНЧ). Обеспечивает корректность преобразования аналогового сигнала в дискретный.
- Аналого-цифровой преобразователь (АЦП). Формирует из аналогового сигнала $\tilde{x}(nT)$ цифровой $x(nT)$ и выполняет две функции: дискретизацию во времени и квантование по уровням.
- Цифровой сигнальный процессор (ЦСП). Вычисляет по заданному алгоритму выходной отсчет $y(nT)$ в зависимости от входного отсчета $x(nT)$.
- Цифро-аналоговый преобразователь (ЦАП). Формирует цифро-аналоговый сигнал $\tilde{y}(t)$.
- Аналоговый сглаживающий фильтр низкой частоты (СФНЧ). С его помощью сигнал $\tilde{y}(t)$ преобразуется в аналоговый сигнал $y(t)$.

При анализе дискретных сигналов удобно пользоваться нормированным временем:

$$\hat{t} = \frac{t}{T} = \frac{nT}{T} = n \qquad (1.1)$$

Таким образом, номер отсчета n интерпретируется как нормированное время. Переход к нормированному времени позволяет рассматривать дискретный сигнал как функцию целочисленной переменной. Представления дискретного сигнала $x(n)$ и $x(nT)$ являются равнозначными. Используется то, которое наиболее удобно в каждом конкретном случае.

Основной математической моделью для описания систем ЦОС служит разностное уравнение (уравнение вход-выход), связывающее входное воздействие $x(n)$ и реакцию системы на него $y(n)$:

$$y(n) = \sum_{i=0}^{N-1} b_i x(n-i) - \sum_{i=1}^{M-1} a_i y(n-i), \qquad (1.2)$$

где b_i и a_i — весовые постоянные коэффициенты. Конкретный набор коэффициентов определяет алгоритм обработки дискретного сигнала. Описываемая соотношением (1.2) цифровая система является линейной и называется *линейной дискретной системой* (ЛДС). Реализация ЛДС требует только трех операций:

- умножения на постоянный весовой коэффициент;
- суммирования;
- задержки на один период частоты дискретизации (сдвиг).

Поставив в соответствие каждой операции графический символ (рис. 3) и объединив их в соответствии с разностным уравнением, получим еще одну модель представления ЛДС — *структурную схему*. Пример реализации структурной схемы для разностного уравнения (РУ) вида

$$y(n) = b_0x(n)+b_1x(n-1)-a_1y(n-1) \qquad (1.3)$$

показан на рис. 4.

При изучении цифровых систем в качестве испытательных воздействий используются дискретные сигналы, называемые типовыми. Одним из примеров может служить *цифровой единичный импульс* $u_0(n-n_0)$, равный единице при $n = n_0$ и равный нулю при остальных значениях n:

$$u_0(n-n_0) = \begin{cases} 1, & n = n_0 \\ 0, & n \neq n_0 \end{cases}, \qquad (1.4)$$

где $n = 0, \pm1, \pm2, \pm3\ldots$, n_0 — целочисленная константа.

Если ЛДС до момента поступления на вход единичного цифрового импульса $u_0(n-n_0)$ находилась в нулевом состоянии, то есть $x(n) = 0$ и $y(n) = 0$ для всех $n < n_0$, тогда выходной сигнал $y(n)$ (реакция системы на сигнал $u_0(n)$) называется *импульсной характеристикой* $h(n)$ системы ЦОС (ИХ):

$$y(n)\Bigg|_{\substack{x(n)=u_0(n),\\ x(n)=0\text{и}\\ y(n)=0 \text{ при } n<n_0}} = h(n). \qquad (1.5)$$

Функционирование ЛДС происходит по тактам, равным по длительности периоду дискретизации. Каждый такт включает несколько этапов:

1 этап — поступление текущего отсчета $x(n)$ на вход системы ЦОС;

2 этап — вычисление текущего выходного отсчета $y(n)$ в соответствии с выражением (1.2);

3 этап — поступление выходного текущего отсчета $y(n)$ на выход системы ЦОС;

4 этап — формирование новой задержанной последовательности отсчетов $x(n-i)$ и $y(n-i)$ в соответствии с правилом:

$$x(i) = x(i-1) \text{ и } y(i) = y(i-1). \qquad (1.6)$$

На следующем такте процесс повторяется.

На рис. 5 показан процесс функционирования ЛДС, описываемой РУ (1.3), в случае

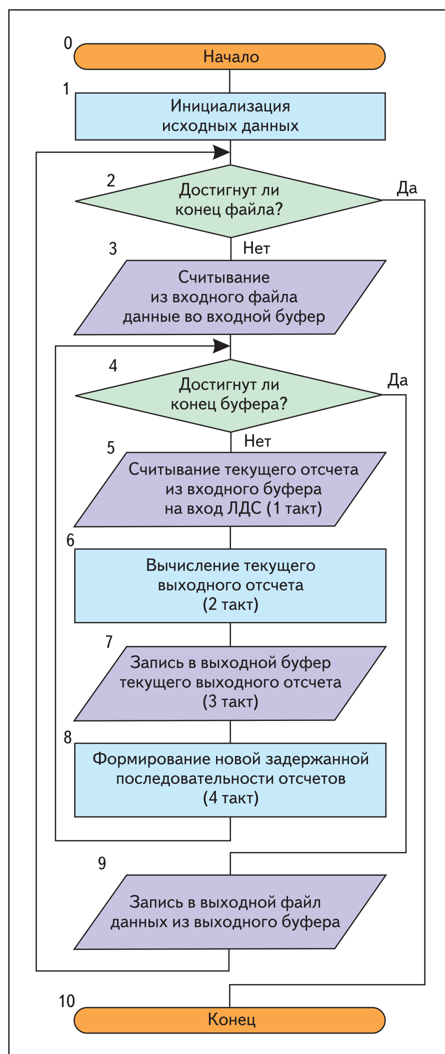

Рис. 6. Блок-схема алгоритма функционирования ЛДС

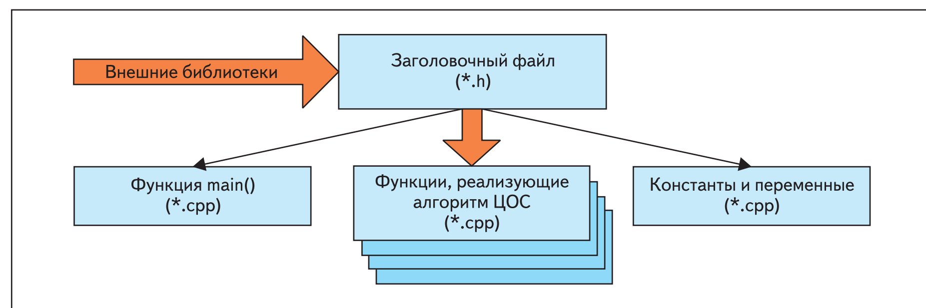

Рис. 7. Структура файловой модели программной реализации алгоритма ЦОС

поступления на вход единичного цифрового импульса, при условии $n_0 = 0$ (см. (1.2)). Блок-схема алгоритма функционирования ЛДС показана на рис. 6.

Для программного моделирования алгоритмов ЦОС может быть использована *файловая модель*. Суть данного подхода заключается в том, что входное воздействие и реакция на него находятся в файлах, которые называются *тестовыми векторами*. При разработке конкретного алгоритма ЦОС (например, цифровой фильтрации) выбирают воздействие, реакция на которое заранее известна. Таких воздействий может быть несколько. Обрабатывая входной файл при помощи программной реализации алгоритма, получают выходной файл и убеждаются в соответствии расчетного и полученного результатов.

Основные требования файловой модели просты и заключаются в следующем:

- Наличие только одного заголовочного файла в программном коде. В данном файле производится подключение всех необходимых внешних библиотек, объявление констант, массивов, функций, макросов, пользовательских типов и т. д.
- Создание контекстной структуры, включающей все необходимые для функционирования программы глобальные указатели, константы, переменные и т. д.
- Отсутствие статических и динамических переменных и констант. Если их наличие необходимо, они инициализируются как глобальные и включаются в контекстную структуру.
- Создание глобальных переменных, констант, структур и массивов производится в одном отдельном файле. Все глобальные переменные, константы и т. п. должны быть объявлены в заголовочном файле.
- Каждая функция располагается в отдельном файле и должна быть объявлена в заголовочном файле.
- Обмен данными между функциями осуществляется через указатель на контекстную структуру.
- Программный код функций основывается на базовых операторах языка С.
- Функции, реализующие алгоритм ЦОС, не должны быть привязаны к конкретному типу ЦСП. Обмен данными с периферийными устройствами ввода-вывода организуется через входной и выходной буфер.
- Обработка массивов осуществляется через указатели на эти массивы, включенные в контекстную структуру.
- Функция *main*() не должна включать программный код, реализующий алгоритм цифровой обработки сигналов. Задача данной функции — заполнить входной буфер данными из файла, вызвать основную функцию алгоритма ЦОС и записать результат из выходного буфера в выходной файл.

Структура файловой модели показана на рис. 7.

Рассмотрим пример программной реализации цифрового устройства (в дальнейшем фильтра), удовлетворяющий требованиям файловой модели и описываемый нерекурсивным разностным уравнением:

$$y(n) = M\sum_{i=0}^{N-1} b_i x(n-i), \qquad (1.7)$$

где $x(n)$— входной цифровой сигнал, $y(n)$ — выходной цифровой сигнал, M — масштабирующий коэффициент, b_i — коэффициенты фильтра.

Для генерации программного кода воспользуемся интегрированной средой разработки (ИСР) *Visual Studio* (VS) версии 5 и выше.

Необходимо запустить VS (рис. 8) и создать консольное приложение. Для этого в меню «*File*» нужно выбрать команду «*New*» (рис. 9). Появится окно, показанное на рис. 10. На вкладке «*Projects*» данного окна определяется тип приложения, его расположение на диске и задается имя.

Необходимо выбрать тип «*Win32 Console Application*», указать в окошке «Location» место размещения консольного приложения (в дальнейшем проекта), в окошке «*Project name*» ввести его имя и нажать кнопку «*OK*» (подтвердить ввод). В результате появится окно параметров создаваемого приложения.

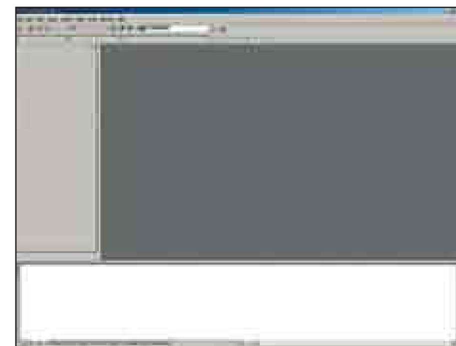

Рис. 8. Вид запущенной ИСР VS

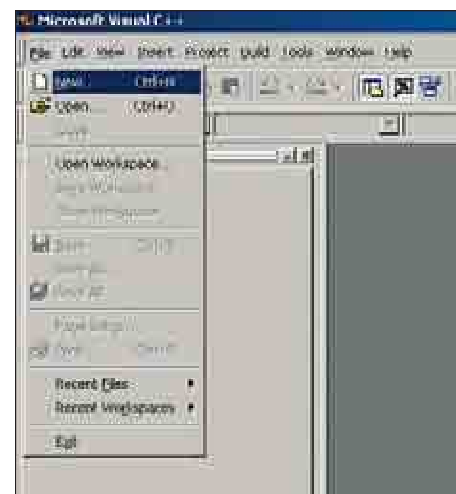

Рис. 9. Создание нового проекта

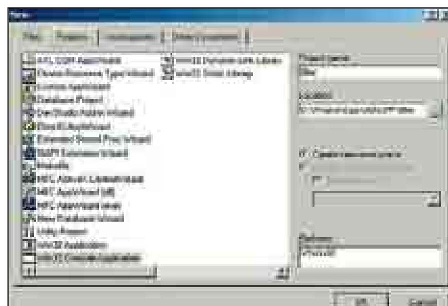

Рис. 10. Создание консольного приложения

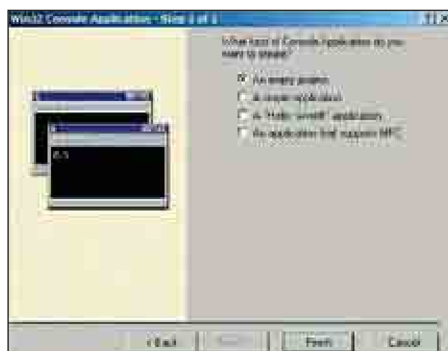

Рис. 11. Параметры создаваемого проекта

Рис. 12. Информация о параметрах созданного проекта

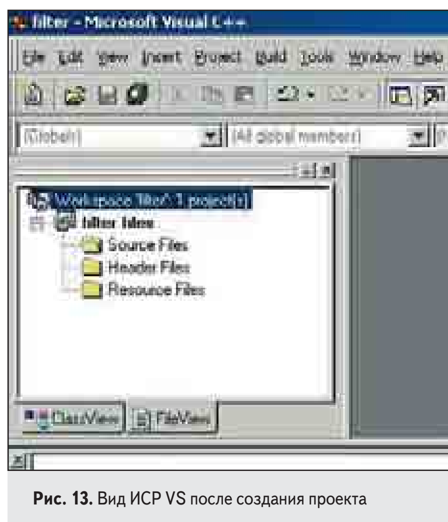

Рис. 13. Вид ИСР VS после создания проекта

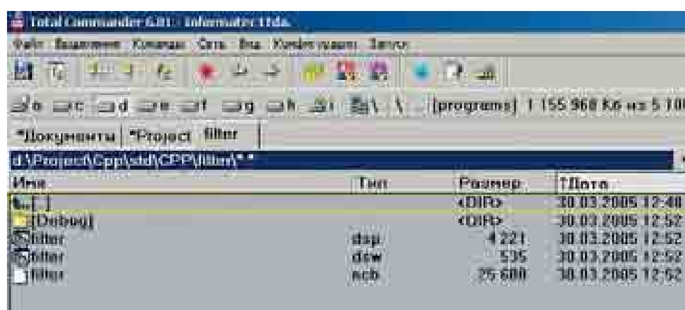

Рис. 14. Служебные файлы консольного приложения

Все настройки необходимо выставить, как показано на рис. 11, и нажать кнопку «*Finish*». Появится информационное окно, сообщающее о параметрах созданного проекта (рис. 12). Необходимо убедиться, что проект создан с требуемыми характеристиками, и подтвердить ввод.

В результате будет создано консольное приложение, и вид ИСР VS изменится (рис. 13). Станет активным окно «*WorcSpace*» и в нем две вкладки: «*ClassView*» и «*FileView*». Необходимо перейти на вкладку «FileView», где показана структура проекта.

В директории, указанной в окошке «*Location*» при создании проекта, появится папка, в которой находятся служебные файлы проекта. Эти файлы определяют тип и структуру создаваемого программного кода (рис. 14). Данные файлы можно редактировать, только имея достаточный опыт в создании подобных проектов. На начальном этапе лучше оставить их без изменений.

После создания проекта к нему подключаются заголовочный файл (с расширением **.h*) и файлы с функциями, реализующими алгоритм ЦОС (с расширением **.c* или **.cpp*). Эти файлы являются текстовыми и содержат исходный программный код алгоритма на языке С. Создать файлы можно в произвольном текстовом редакторе, а затем скопировать в папку с проектом и подключить. Или же файлы можно создать непосредственно в ИСР VS при помощи встроенного текстового редактора. В данной статье рассматривается второй вариант.

Для создания файлов с исходным текстом программного кода необходимо в меню «*File*» выбрать опцию «*New*» (рис. 9) и на вкладке «*Files*» указать тип файла (рис. 15).

Для заголовочного файла выбирают тип «*C/C++ Header File*», а для файлов с функциями — тип «*C++ Source File*». Затем необходимо указать место размещения файла в окошке «*Location*» (обычно это папка проекта) и имя файла в окошке «*File name*». Необходимо также установить флажок в позиции «*Add to project*» и убедиться, что в окошке, расположенном ниже флажка, имя проекта указано верно. Определив тип, место и имя файла, подтверждают ввод.

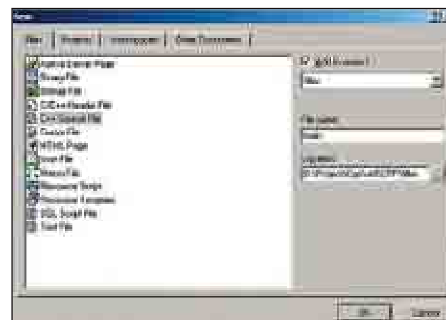

Рис. 15. Создание файлов с исходным текстом программного кода

Для реализации файловой модели алгоритма (рис. 16) необходимо создать следующие файлы:

- заголовочный файл «*filter.h*», содержащий объявления функций, массивов, пользовательских типов и т. д.;
- файл «*main.cpp*» с функцией *main*() (соответствует блокам 0, 2, 3, 9 и 10 в блок-схеме на рис. 6);
- файл «*init.cpp*» с функцией начальной инициализации контекстной структуры (блок 1 на рис. 6);
- файл «*run.cpp*» с функцией запуска алгоритма обработки входного буфера (блоки 4, 5, 7 на рис. 6);

Рис. 16. Вид ИСР VS после создания всех необходимых файлов проекта

- файл «*convolution.cpp*» с функцией, выполняющей операции свертки и формирования новой задержанной последовательности (блоки 6, 8 на рис. 6);
- файл «*const.cpp*», со всеми необходимыми глобальными константами, переменными, массивами и т. д.

Данная структура проекта удовлетворяет требованиям файловой модели и, с точки зрения автора, является наиболее рациональной, но не единственно возможной.

После создания всех необходимых файлов ИСР VS примет вид, показанный на рис. 16. Можно выбрать мышкой любой файл и в клиентской области ИСР VS откроется область его редактирования, где необходимо набрать программный код (рис. 17). Вначале поле редактирования пусто. Необходимо набрать текст программного кода, как в обычном текстовом редакторе.

Для программной реализации алгоритма (1.7) необходимо определить тип контекстной структуры со следующими полями:

- указатель на входной буфер;
- указатель на выходной буфер;
- длина входного и выходного буферов;
- указатель на буфер линии задержки;
- указатель на буфер с коэффициентами фильтра;
- длина буферов линии задержки и коэффициентов.

Тип контекстной структуры определяется в файле «*filter.h*» следующим образом:

```
typedef struct {
    word16 *pInpBuff;     // Указатель на входной буфер
    word16 *pOutBuff;     // Указатель на выходной буфер
    word16 lenBuff;       // Длина входного и выходного буферов
    word16 *pSimplBuff;   // Указатель на буфер линии задержки
    word16 *pCoeffBuff;   // Указатель на буфер с коэффициентами
                          // фильтра
    word16 lenFir;        // Длина буферов линии задержки
                          // и коэффициентов
} CONTEXTFILTER;
```

Затем в файле «*const.cpp*» необходимо создать контекстную структуру данного типа и указатель на нее:

```
CONTEXTFILTER     cntx;    // Контекстная структура фильтра
CONTEXTFILTER*    pCntx;   // Указатель на контекстную
                           // структуру фильтра
```

Кроме того, в этом же файле создаются массивы:

- для входных данных;
- для выходных данных;
- для линии задержки;
- для буфера коэффициентов.

Создание массива происходит следующим образом:

```
word16 inpBuff[LENBUFF];        // Массив для входных данных
word16 outBuff[LENBUFF];        // Массив для выходных данных
word16 simplBuff[LENFILTER];    // Массив для линии задержки
```

Размеры массивов для входных и выходных данных, а также для линии задержки указываются через макросы, которые определяются в файле «*filter.h*»:

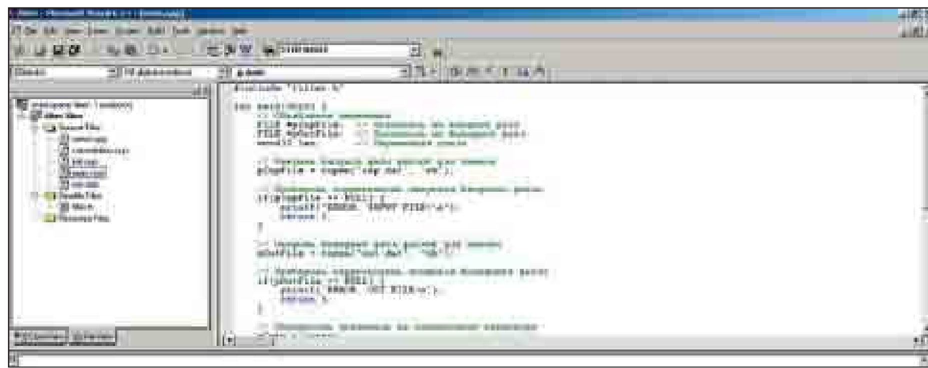

Рис. 17. Редактирование файла с исходным текстом программного кода

```
#define LENBUFF     120  // Длина входного и выходного буферов
#define LENFILTER   10   // Порядок (длина) фильтра
```

Кроме того, в этом же файле вводятся новые имена стандартных типов:

```
typedef int word32;
typedef short word16;
typedef char word8;
```

Этот шаг обусловлен тем, что разрядность стандартных типов часто зависит от используемого компилятора, а при разработке алгоритма для ЦСП необходимо строго контролировать разрядность переменных. При использовании новых имен для стандартных типов всегда точно известна разрядность переменных. Если необходимо использовать другой компилятор и там изменилась разрядность стандартных типов, то это можно легко компенсировать изменением нескольких строк программного кода в заголовочном файле.

Буфер коэффициентов непосредственно инициализируется при создании значениями коэффициентов:

```
word16 coeffBuffLF[]={1128, -2364, -2960, 4582, 14652, 14652, 4582,
-2960, -2364, 1128};
```

Полный листинг файла «*const.cpp*»:

```
#include «filter.h»

////////////////////////////////////////////////////////////////////////////////
// Создание констант
////////////////////////////////////////////////////////////////////////////////
CONTEXTFILTER cntx;       // Контекстная структура фильтра
CONTEXTFILTER* pCntx;     // Указатель на контекстную
                          // структуру фильтра

////////////////////////////////////////////////////////////////////////////////
// Создание массивов
////////////////////////////////////////////////////////////////////////////////

word16 inpBuff[LENBUFF];       // Массив для входных данных;
word16 outBuff[LENBUFF];       // Массив для выходных данных;
word16 simplBuff[LENFILTER]; // Массив для линии задержки;
// Массив для коэффициентов фильтра
word16 coeffBuff[] = {1128, -2364, -2960, 4582, 14652, 14652, 4582,
-2960, -2364, 1128};
```

Инициализацию контекстной структуры выполняет функция инициализации *initFilter*():

```
#include «filter.h»

void initFilter(CONTEXTFILTER* pCntx){
    // Создание локальных переменных
    word32 i; // Переменная цикла

        pCntx->pInpBuff = inpBuff;          // Указатель на входной
                                            // буфер
        pCntx->pOutBuff = outBuff;          // Указатель на выходной
                                            // буфер
        pCntx->lenBuff = LENBUFF;           // Длина входного
                                            // и выходного буферов
        pCntx->pSimplBuff = simplBuff;      // Указатель на буфер
                                            // линии задержки
        pCntx->pCoeffBuff = coeffBuff;      // Указатель на буфер
                                            // с коэффициентами
                                            // фильтра
        pCntx->lenFilter = LENFILTER;       // Длина буферов линии
                                            // задержки и коэффици-
                                            // ентов
        pCntx->mCoeff = MASHTAB;            // Масштабирующий
                                            // коэффициент

        // Очистка буфера линии задержки
        for(i = 0; i < pCntx->lenFilter; i++) {
            pCntx->pSimplBuff[i] = 0;
        }}
```

Обработку буфера осуществляет функция *runFilter*():

```
#include «filter.h»

void runFilter(CONTEXTFILTER* pCntx){
    // Локальные переменные
    word16* pInpBuff;     // Указатель на входной буфер
    word16* pOutBuff;     // Указатель на выходной буфер
    word16 lenBuff;       // Длина входного и выходного буферов
    word16* pSimplBuff;   // Указатель на буфер линии задержки
    word16* pCoeffBuff;   // Указатель на буфер
                          // с коэффициентами фильтра
    word16 lenFilter;     // Длина буферов линии задержки
                          // и коэффициентов
    word32 mCoeff;        // Масштабирующий коэффициент
    word32 count;         // Переменная цикла
    word32 coeff;         // Вспомогательная переменная

    // Инициализация локальных переменных
    pInpBuff = pCntx->pInpBuff;
    pOutBuff = pCntx->pOutBuff;
    lenBuff = pCntx->lenBuff;
    pSimplBuff = pCntx->pSimplBuff;
    pCoeffBuff = pCntx->pCoeffBuff;
    lenFilter = pCntx->lenFilter;
    mCoeff = pCntx->mCoeff;

    // Цикл обработки входного буфера
    for(count = 0; count < lenBuff; count++){

            // Чтение входного отсчета
            coeff = pInpBuff[count];

            // Умножение на масштабирующий коэффициент
            coeff *= mCoeff;

            // Нормирование результата
            coeff >>= 15;

            // Запись в буфер задержанных отсчетов
            pSimplBuff[0] = (word16) coeff;

            // Определение выходного отсчета
            coeff = convolution(pSimplBuff, pCoeffBuff, lenFilter);

            // Запись выходного отсчета
            pOutBuff[count] = (word16) coeff;
    }
}
```

Функции инициализации контекстной структуры и обработки входного буфера должны быть объявлены в заголовочном файле следующим образом:

```
// Функция инициализации контекстной структуры
extern void initFilter(CONTEXTFILTER* pCntx);
// Функция обработки входного буфера
extern void runFilter(CONTEXTFILTER* pCntx);
```

Вычисления выходного отсчета осуществляет функция свертки *convolution*() (см. рис. 18):

```
#include «filter.h»

word16 convolution(word16* pSimplBuff, word16* pCoeffBuff,
word16 lenFilter){
    // Объявление локальных переменных
    word16 coeffX;
    word16 coeffB;
    word32 count;
    word32 coeff;
    word32 sum;

    // Инициализация локальных переменных
    sum = 0;

    // Цикл обработки линии задержки
    for(count = 0; count < lenFilter; count++){

        // Чтение текущего отсчета
        coeffX = pSimplBuff[count];

        // Чтение соответствующего коэффициента фильтра
        coeffB = pCoeffBuff[count];

        // Умножение отсчета на коэффициент
        coeff = coeffX * coeffB;

        // Накопление результата
        sum += coeff;
    }

    // Сдвиг линии задержки
    for(count = lenFilter — 1; count > 0; count--){

        // Чтение предыдущего отсчета
        coeffX = pSimplBuff[count — 1];

        // Запись предыдущего отсчета в текущую ячейку
        pSimplBuff[count] = coeffX;
    }

    // Нормирование результата суммирования
    sum >>= 15;

    // Выход из функции
    return sum;

}
```

Данная функция также должна быть объявлена в заголовочном файле:

```
// Функция вычисления выходного отсчета
extern word16 convolution(word16*, word16*, word16);
```

Окончательный вид заголовочного файла «*filter.h*» будет иметь следующий вид:

```
/////////////////////////////////////////////////////////////////////////////
// Подключение внешних библиотек
/////////////////////////////////////////////////////////////////////////////

#include <stdio.h> // Стандартная библиотека ввода/вывода

/////////////////////////////////////////////////////////////////////////////
// Определение макросов
/////////////////////////////////////////////////////////////////////////////

#define LENBUFF 120      // Длина входного и выходного буферов
#define LENFILTER 10     // Порядок (длина) фильтра
#define MASHTAB 28093    // Масштабирующий коэффициент
                         // фильтра

/////////////////////////////////////////////////////////////////////////////
// Определение новых имен стандартных типов
/////////////////////////////////////////////////////////////////////////////
```

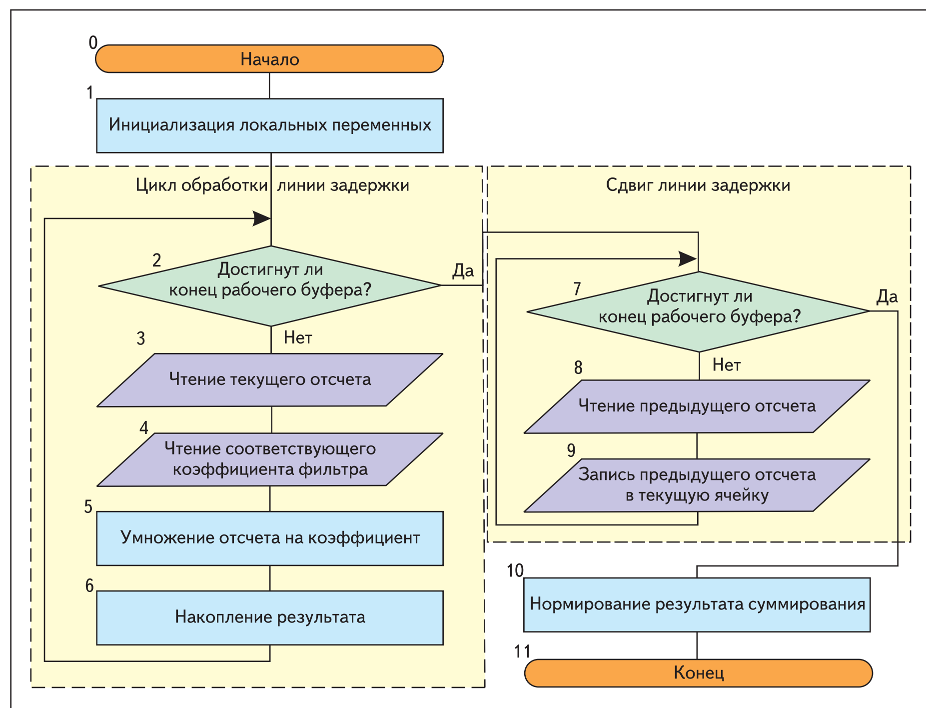

Рис. 18. Блок-схема алгоритма функции свертки convolution()

```
typedef int word32;
typedef short word16;
typedef char word8;

/////////////////////////////////////////////////////////////////////////////
// Определение пользовательских типов
/////////////////////////////////////////////////////////////////////////////

// Определение типа контекстной структуры фильтра
typedef struct {
    word16 *pInpBuff;     // Указатель на входной буфер
    word16 *pOutBuff;     // Указатель на выходной буфер
    word16 lenBuff;       // Длина входного и выходного буферов
    word16 *pSimplBuff;   // Указатель на буфер линии задержки
    word16 *pCoeffBuff;   // Указатель на буфер
                          // с коэффициентами фильтра
    word16 lenFir;        // Длина буферов линии задержки
                          // и коэффициентов
    word32 mCoeff;        // Масштабирующий коэффициент
} CONTEXTFILTER;

/////////////////////////////////////////////////////////////////////////////
// Объявление констант и массивов
/////////////////////////////////////////////////////////////////////////////

extern CONTEXTFILTER cntx;       // Контекстная структура
                                 // фильтра
extern CONTEXTFILTER* pCntx;     // Указатель на контекстную
                                 // структуру фильтра

extern word16 inpBuff[];         // Массив для входных данных;
extern word16 outBuff[];         // Массив для выходных данных;
extern word16 simplBuff[];       // Массив для линии задержки;
extern word16 coeffBuff[];       // Массив для буфера
                                 // коэффициентов;
extern word16 coeffBuffLF[];     // Массив для коэффициентов
                                 // фильтра

/////////////////////////////////////////////////////////////////////////////
// Объявление функций
/////////////////////////////////////////////////////////////////////////////

extern void initFilter(CONTEXTFILTER* pCntx);   // Функция
                                                // инициализации
                                                // контекстной
                                                // структуры
extern void runFilter(CONTEXTFILTER* pCntx);    // Функция
                                                // обработки
                                                // входного
                                                // буфера
extern word16 convolution(word16*, word16*, word16);  // Функция
                                                      // вычисления
                                                      // выходного
                                                      // отсчета
```

Завершает разработку программного кода фильтра функция *main*(). Ее главная задача — провести тестирование функций, реализующих разрабатываемый алгоритм, путем имитации функционирования разработанного кода. Это достигается за счет поступления отсчетов входного сигнала из входного файла во входной буфер, в вызове функции обработки этого буфера и сохранении результата работы программы (выходной буфер) в выходном файле. Листинг функции *main*():

```
#include «filter.h»

int main(void) {
    // Объявление переменных
    FILE *pInpFile;    // Указатель на входной файл
    FILE *pOutFile;    // Указатель на выходной файл
    word32 len;        // Переменная цикла

    // Открыть входной файл файлов для чтения
    pInpFile = fopen(«inp.dat», «rb»);

    // Проверить корректность открытия входного файла
    if(pInpFile == NULL) {
        printf(«ERROR, INPUT FILE!\n»);
        return 2;
    }

    // Открыть выходной файл файлов для записи
    pOutFile = fopen(«out.dat», «wb»);

    // Проверить корректность открытия выходного файла
    if(pOutFile == NULL) {
        printf(«ERROR, OUT FILE\n»);
        return 1;
    }

    // Инициализировать указатель на контекстную структуру
    pCntx = &cntx;

    // Инициализировать контекстную структуру
    initFilter(pCntx);

    // Инициализировать переменную цикла
    len = LENBUFF;

    // Цикл обработки входного буфера
    while(len == LENBUFF) {
        // Считать данные из файла
        len = fread(pCntx->pInpBuff,sizeof(word16), LENBUFF, pInpFile);

        // Вызвать функцию обработки входного буфера
        runFilter(pCntx);

        // Записать данные в файл
        fwrite(pCntx->pOutBuff, sizeof(word16), len, pOutFile);
    }
```

```
    // Закрыть входной и выходной файлы
    fclose(pInpFile);
    fclose(pOutFile);

    // Выход из программы
    return 0;
}
```

В рассматриваемом примере фильтр соответствует требованиям:

- частота дискретизации — 8 000 Гц;
- полоса пропускания — 1 600 Гц;
- полоса задерживания — 2 400 Гц;
- отклонение в полосе пропускания — 1 дБ;
- отклонение в полосе задерживания — –20 дБ;
- число коэффициентов — 10;
- импульсная характеристика — симметричная, с четным количеством коэффициентов.

Конкретные значения коэффициентов фильтра могут быть рассчитаны, например, в программе MatLab или FD-3. Результат расчета в программе FD-3 показан в таблице. На рис. 19 и рис. 20 показаны расчетные амплитудно-частотная и импульсная характеристики фильтра.

В рассмотренной программной реализации используется целочисленное представление коэффициентов. Для этого необходимо все коэффициенты умножить на нормирующий множитель и округлить результат до целого значения. Чем большая точность представления коэффициентов требуется, тем большее значение нормирующего множителя необходимо устанавливать. В данном примере точность ограничивается 16 разрядами, поэтому нормирующий множитель равен 2^{15}–1, так как один разряд знаковый.

Таблица. Коэффициенты фильтра

№ п/п	Коэффициент с фиксированной точкой	Целочисленное представление коэффициентов x(2^{15}−1)
0.	0,034449903800000	1128
1.	−0,072134019699999	−2364
2.	−0,090314822399999	−2960
3.	0,139855699899999	4582
4.	0,447167894399999	14652
5.	0,447167894399999	14652
6.	0,139855699899999	4582
7.	−0,090314822399999	−2960
8.	−0,072134019699999	−2364
9.	0,34449903800000	1128
M	0,857374999999999	28093

Применение целочисленного представления коэффициентов приводит к тому, что результат умножения больше истинного. Поэтому результат умножения необходимо разделить на нормирующий множитель (нормировать). Так как нормирующий множитель выбран равным степени двойки, то деление можно заменить сдвигом вправо. Это и сделано в примере программного кода.

Все набранные в текстовом редакторе VS файлы должны быть сохранены нажатием одной из двух (или) клавиш на кнопочной панели.

Программный код, набранный в текстовом виде (файлы с расширением «.h» и «.cpp»), необходимо преобразовать в исполняемый модуль (файл с расширением «.exe»). Эта операция называется *компиляцией*, выполняется *компилятором* (в данном случае VS) и включает два этапа:

- *трансляция* каждого файла с кодом программы (транслируются только файлы с расширением «.cpp», заголовочные файлы подключаются автоматически за счет директивы «*#include*») в объектные модули (файлы с расширением «.obj»);
- *компоновка* (или линковка) объектных файлов в исполняемый модуль.

На первом этапе проверяется соответствие программного кода каждого файла в отдельности требованиям языка С. На втором — взаимосвязь между частями программы (функциями, константами, массивами, подключаемыми внешними библиотеками и т. д.).

В VS определены три режима компиляции:

- трансляция отдельного файла и создание его объектного модуля;
- трансляция только тех файлов, которые были изменены, в объектные модули, линковка всего проекта и получение нового исполняемого модуля;
- трансляция и линковка всех файлов, построение нового исполняемого модуля.

Первый режим (рис. 21a) используется для проверки синтаксиса отдельного файла. Третий режим (рис. 21c) обычно используется при начальной компиляции проекта и периодического контроля правильности выполнения компиляции во втором режиме. Это обусловлено тем, что время компиляции для третьего режима самое большое и для реальных проектов может достигать десятка минут. Наиболее часто при отладке программы используется второй режим (рис. 21b). Он позволяет значительно экономить время компиляции, но иногда могут возникнуть сбои, поэтому периодически необходимо проводить компиляцию в третьем режиме.

В результате компиляции создается исполняемый модуль (файл с расширением .exe), который находится в папке проекта «*debug*». Запустить файл на выполнение можно непосредственно из операционной системы. Но лучше всего сделать это в среде VS, нажав клавишу на кнопочной панели.

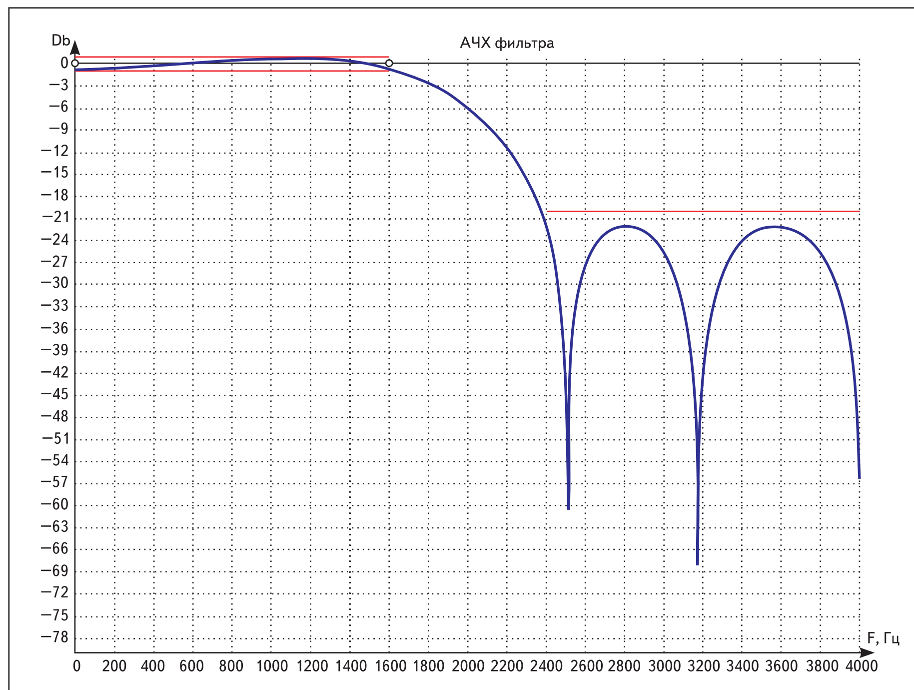

Рис. 19. Амплитудно-частотная характеристика фильтра

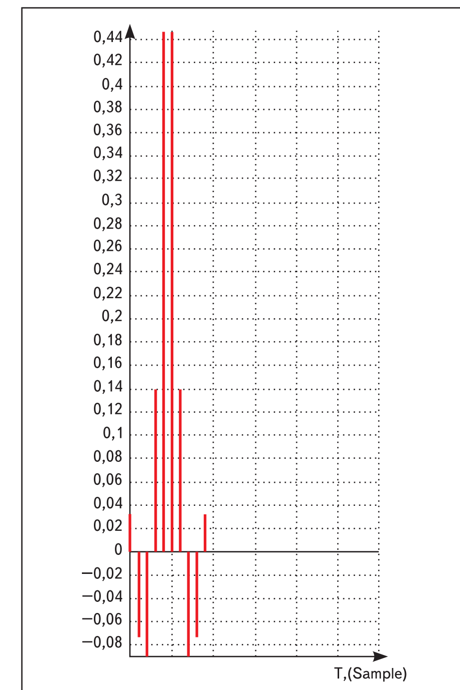

Рис. 20. Импульсная характеристика фильтра

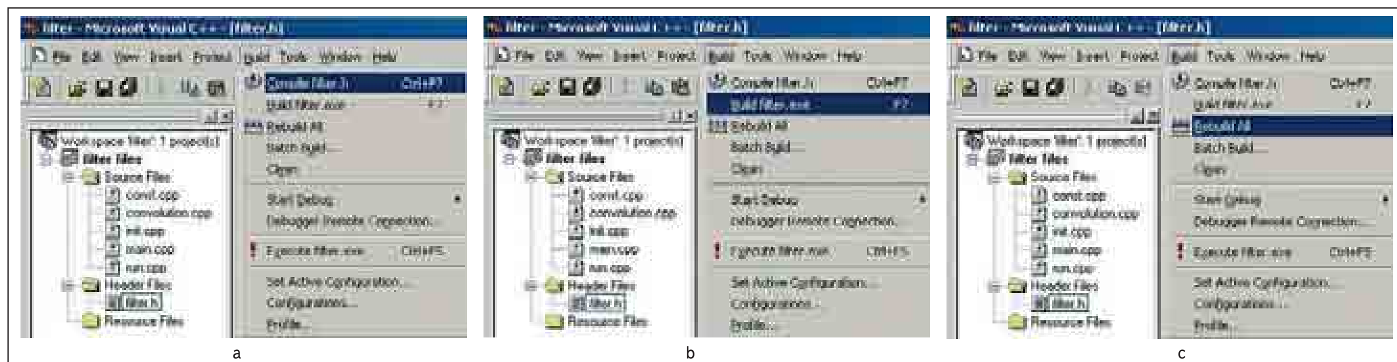

Рис. 21. Компиляция программного кода

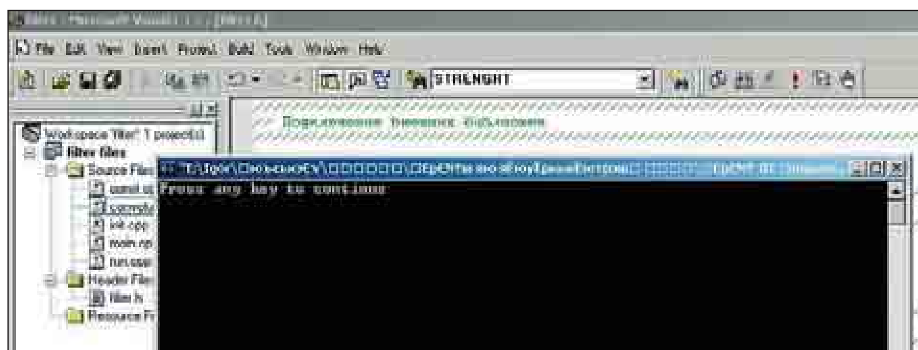

Рис. 22. Запуск проекта на выполнение

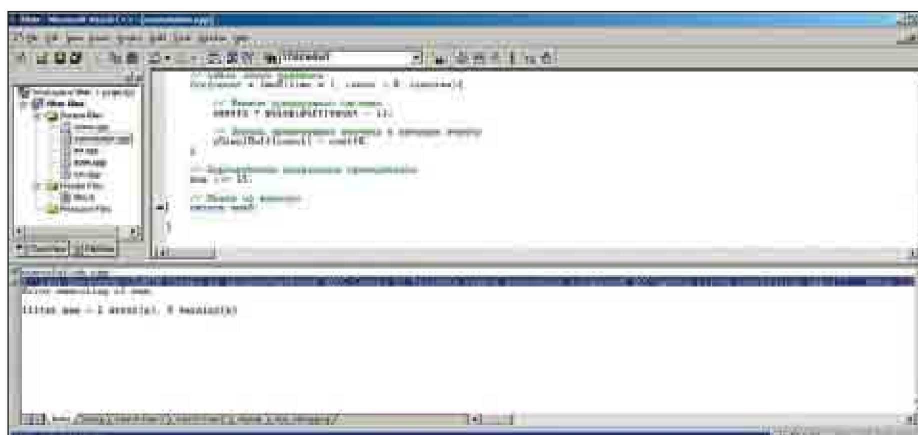

Рис. 23. Выявление синтаксических ошибок

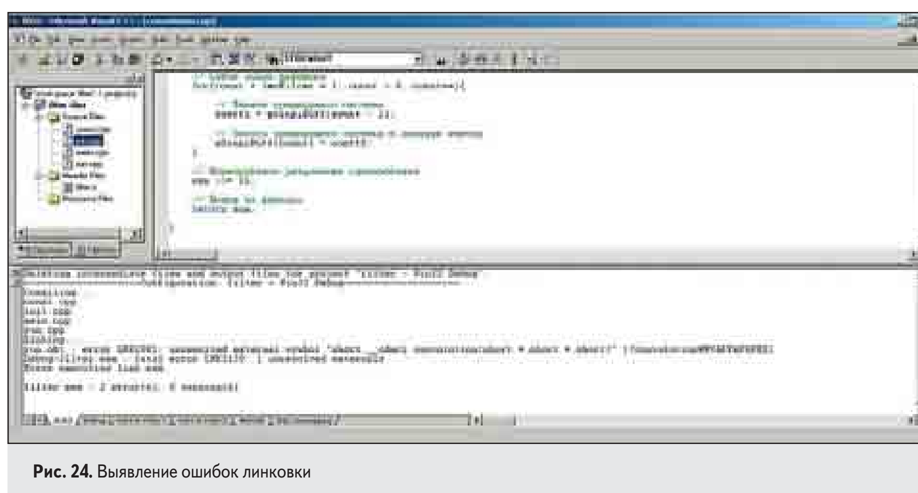

Рис. 24. Выявление ошибок линковки

После запуска программы появится консольное окно, в котором, например, можно отобразить вычисленный программой результат или информацию о самом процессе выполнения. В конце выполнения программы в консольном окне появится надпись с предложением нажать любую кнопку для выхода из консольного окна (рис. 22).

На этом построение файловой модели можно считать завершенным. Однако немаловажную роль в разработке программного кода играет процесс отладки. Во-первых, необходимо постараться выявить все ошибки в коде. Во-вторых, необходимо оптимизировать программный код для выбранного ЦСП. Вторая задача будет подробно рассмотрена в третьей статье данного цикла. А сейчас ознакомимся с некоторыми способами поиска ошибок в программном коде.

В процессе создания программного кода могут возникнуть ошибки трех типов:

- синтаксические ошибки;
- ошибки линковки;
- ошибки выполнения.

Синтаксические ошибки вызваны несоответствием требованиям языка программирования. Определяются они на первом этапе компиляции. Если ошибка данного типа произошла, появляется информационное сообщение об ошибке и ее типе в одном из окон ИСР VS (рис. 23). Необходимо навести курсор на описание выявленной ошибки и щелкнуть мышкой. Курсор переместится к месту выявленной ошибки в тексте программного кода. Однако нужно помнить, что компилятор иногда выставляет курсор не на ошибку, а на строку ниже. Поэтому необходимо проверить не только выделенную строку, но и предыдущие.

Ошибки линковки обусловлены несоответствием связей между отдельными компонентами программы. Например, была объявлена некая функция в заголовочном файле проекта, в одной из функций она была использована (вызвана), а файл с описанием самой функции отсутствует или не подключен к проекту (рис. 24). Такие ошибки выявляются на втором этапе компиляции.

Ошибки выполнения связаны, в первую очередь, с нарушением логики работы про-

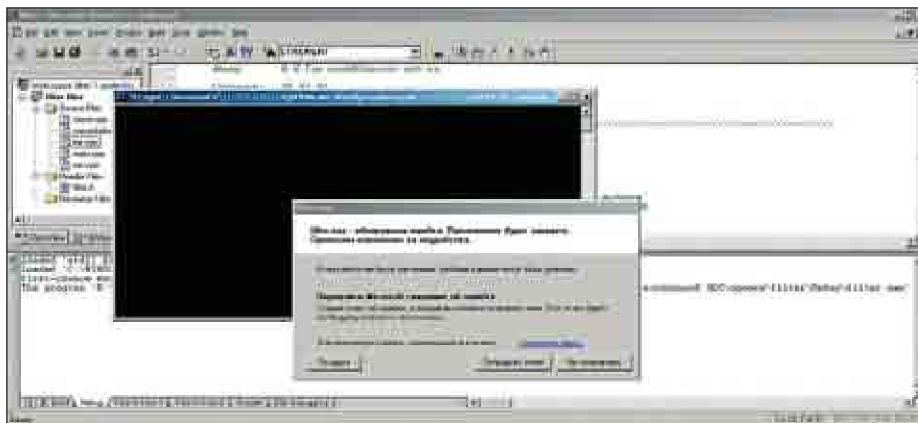

Рис. 25. Ошибки выполнения

граммы. Они выявляются в процессе работы программы на ПК. К таким ошибкам можно отнести деление на ноль, использование созданной, но не инициализированной переменной, выход за пределы массива. Перечисленные ошибки обычно приводят к аварийному завершению выполнения программы (рис. 25). Для выявления таких ошибок используют запуск программы в режиме отладки. Этот режим задается нажатием клавиши на кнопочной панели. Выполнение программы останавливается в месте сбоя. Данный вид ошибок самый сложный для устранения, так как выявление места сбоя не гарантирует наличия ошибки именно здесь. Ошибка может произойти в одном фрагменте кода, а проявиться при выполнении совершенно в другом.

В качестве метода поиска ошибки в данном случае можно предложить следующий алгоритм:

- В проверяемой функции исключить из процесса выполнения все внешние функции. Для этого вызываемые функции могут быть закомментированы, или вместо них вставляют код, имитирующий их работу. Например, если результатом работы вызываемой функции является заполнение некоторого буфера, тогда вставляем код, который просто заполняет данный буфер произвольными числами (или одним и тем же числом).
- Устранить все выявленные ошибки в проверяемой функции.
- Подключить (раскомментировать) по очереди все вызываемые функции и убедиться в работоспособности программы. Если появилась ошибка, тогда перейти к функции, вызвавшей ошибку, и повторить алгоритм проверки для нее заново.

По этому алгоритму необходимо проверить все функции разрабатываемого проекта, начав с функции *main*(). Еще существуют ошибки выполнения, обусловленные некорректной реализацией самого алгоритма. При этом программа работает без сбоев, но выдает неправильный результат. Выявляются такие ошибки путем подачи на вход цифрового устройства типового воздействия с заранее известной реакцией. Сравнивая полученную реакцию с требуемой, делают вывод о степени соответствия реализованного алгоритма поставленной задаче.

В рассматриваемом примере реализован цифровой фильтр на базе нерекурсивного разностного уравнения (2.1). Одним из его свойств является то, что его импульсная характеристика (ИХ) конечна и равна (с точностью до нормирующего множителя M) коэффициентам фильтра. Кроме этого, преобразование Фурье от ИХ для любого фильтра есть частотная характеристика (ЧХ). Модуль ЧХ — это амплитудно-частотная характеристика (АЧХ). Поэтому для контроля корректной работы фильтра можно в качестве тестового сигнала использовать единичный цифровой импульс, описываемый выражением (1.4), где $n_0 = 0$. Данный сигнал равен единице при $n = 0$ и равен нулю при остальных значениях n. Реакция фильтра на него при нулевых начальных условиях ($y(n)$ и $x(n) \equiv 0$, для $n < 0$) есть ИХ, а модуль преобразования Фурье от ИХ — это АЧХ.

В начале необходимо сформировать входной сигнал, соответствующий (1.4). Это можно сделать самостоятельно, написав программу генерации файла бинарного типа, содержащего 16-разрядные (для рассматриваемого примера) целые числа. Программа не очень сложная даже для программиста невысокой квалификации. Но лучше воспользоваться специализированным ПО. В частности, автор пользуется программой EDSW. Скачать демо-версию этой программы можно с сайта w ww. dsp-sut.spb.ru. Программа не только позволяет сгенерировать файл с нужным сигналом, но и предоставляет инструменты для анализа результата. Пример работы программы показан на рис. 26. На левом верхнем графике изображен входной сигнал (единичный цифровой импульс), на левом нижнем — реакция (ИХ), на правом — модуль преобразования Фурье (АЧХ).

Сравнивая результат работы фильтра (рис. 26) с предъявленными требованиями (рис. 19 и рис. 20) видим, что отклонение в полосе пропускания несколько больше заданных требований. Это вызвано квантованием коэффициентов фильтра. Для устранения погрешности можно повысить точность представления коэффициентов фильтра, увеличить его порядок, или изменить структуру фильтра. В каждом конкретном случае решение принимает разработчик, исходя из условий конкретной задачи.

Проект рассмотренного примера можно скачать с сайта w ww. scanti.ru. Если вы пользуетесь каким-либо другим компилятором

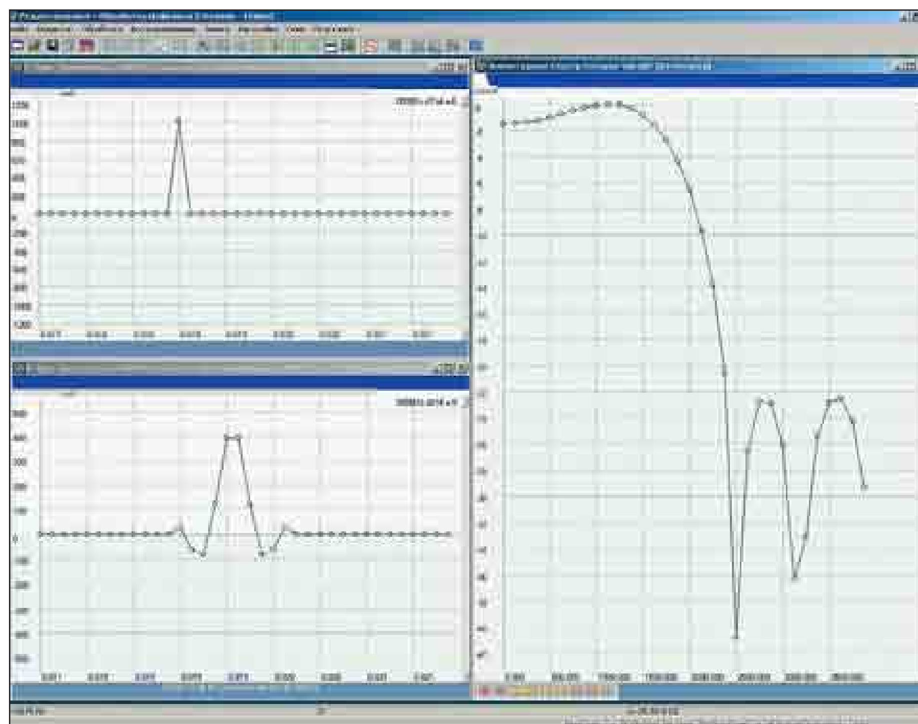

Рис. 26. Анализ результата работы фильтра