УДК 681.3.06
У912

№ 4401

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего профессионального образования
"Южный федеральный университет"

Технологический институт Южного федерального университета в г.Таганроге

Работа выполнена в соответствии с Программой развития Южного федерального университета на 2007-2010гг. Научно-образовательный проект "Разработка образовательных контентов для специальностей 010503 - "Математическое обеспечение и администрирование информационных систем", 230105 - "Программное обеспечение вычислительной техники и автоматизированных систем", подготовки бакалавров и магистров по направлению "Информатика и вычислительная техника"

Н.Ш. Хусаинов

Методическое пособие по курсу "Теория кодирования информации"

РАБОТА С БИТОВЫМ ПОТОКОМ В КОДЕКАХ ЭФФЕКТИВНОГО И ПОМЕХОУСТОЙЧИВОГО КОДИРОВАНИЯ СООБЩЕНИЙ

Для студентов специальностей

230105 – Программное обеспечение вычислительной техники и автоматизированных систем

010503 – Математическое обеспечение и администрирование информационных систем

КАФЕДРА МАТЕМАТИЧЕСКОГО ОБЕСПЕЧЕНИЯ И ПРИМЕНЕНИЯ ЭВМ

ФАВТ

Таганрог 2007

УДК 681.3.06 (076.5) + 681.325.5. (076.5)

Хусаинов Н.Ш. Методическое пособие по курсу “Теория кодирования информации". Работа с битовым потоком в кодеках эффективного и помехоустойчивого кодирования сообщений.– Таганрог: ТТИ ЮФУ, 2007. 24с.

Пособие посвящено вопросам разработки одного из элементов программной системы кодирования/декодирования информации в современных системах обработки информации и связи с возможностью передачи и хранения данных.

Рассматривается организация и принципы функционирования битового буфера. Приводятся различные по сложности и эффективности способы организации записи (чтения) кодовых слов в битовый буфер.

Кратко анализируются достоинства и недостатки стандартных библиотечных реализаций методов работы с битовыми массивами, входящих в состав средств разработки Microsoft Visual Studio и Borland Builder, а также открытых источников из Интернет.

Предлагается вариант программной реализации класса cBitStream для использования в рамках лабораторных работ по курсу "Теория кодирования информации".

Пособие предназначено для студентов специальностей 220400, 351500, изучающих курс "Теория кодирования информации". Представляет интерес для инженеров, разработчиков и программистов в области систем обработки и кодирования текстовой, аудио- и визуальной информации.

СОДЕРЖАНИЕ

1. Назначение битового буфера. Требования к функциональным возможностям

Алгоритмы обработки данных, предполагающие использование кодовых слов переменной длины и/или кодовых слов фиксированной длины некратной одному байту, повсеместно применяются в системах связи, компрессии, защиты информации, компьютерной графики, обработки аудио и видеоинформации. Одним из элементов таких систем является подсистема приема/передачи данных по каналу связи или хранения/восстановления данных на жестком диске или ином носителе информации.

Программная реализация алгоритмов формирования или чтения пакетов/буферов выходных и входных данных зачастую предполагает работу с порциями данных (кодовых или информационных слов, битовых полей), размер которых отличается от 8 (16, 24, 32) битов. Примером могут служить форматы представления видеоданных. Например, стандарт цифрового кодирования телевизионных изображений ITU-R.601 предполагает представление каждого пиксела изображения в виде триплета из трех цветовых компонент, причем на каждую компоненту отводится 10 битов. Такой способ представления данных, хотя и требует выполнения некоторого унифицированного набора побитовых логических операций и сдвигов, но в целом достаточно прост для программной реализации.

Кроме того, размеры битовых полей, отводимых для хранения различных слов данных, могут различаться (так называемые "кодовые слова переменной длины"). Пример – алгоритмы компрессии данных, например, алгоритм Хаффмана, который заменяет каждое информационное слово (символ сообщения) на уникальное кодовое слово, причем длина последнего зависит от частоты появления символа в исходном сообщении: чем чаще встречается символ, тем короче должно быть соответствующее ему кодовое слово. В этом случае процедуры записи/чтения слова в буфер несколько усложняются.

И, наконец, формируемый битовый поток, сформированный кодером, не может целиком размещаться в оперативной памяти – его требуется периодически выдавать в канал связи или сбрасывать на жесткий диск. Например, если речь идет об обработке сигнала в реальном масштабе времени (телефонный разговор, видео в реальном времени), то это объясняется следующими причинами:

- оперативная память вычислительного модуля любого устройства связи ограничена по объему и может обычно хранить порцию обработанных аудио или видеоданных длительностью от нескольких миллисекунд до нескольких минут;

- чем больше объем фрагмента, который накапливается в памяти, тем больше задержка при передаче данных. Слишком большая задержка для большинства систем связи в реальном времени является недопустимой.

В качестве временного хранилища информации закодированной информации перед ее передачей в канал связи или сохранением на диск используется некоторая область памяти, называемая битовым буфером (Bit Stream, Bit Array).

Во многих случаях в реальных системах связи на основе многозадачных ОС или систем с прерываниями в качестве битового буфера могут использоваться сразу два массива (см. рисунок 1).

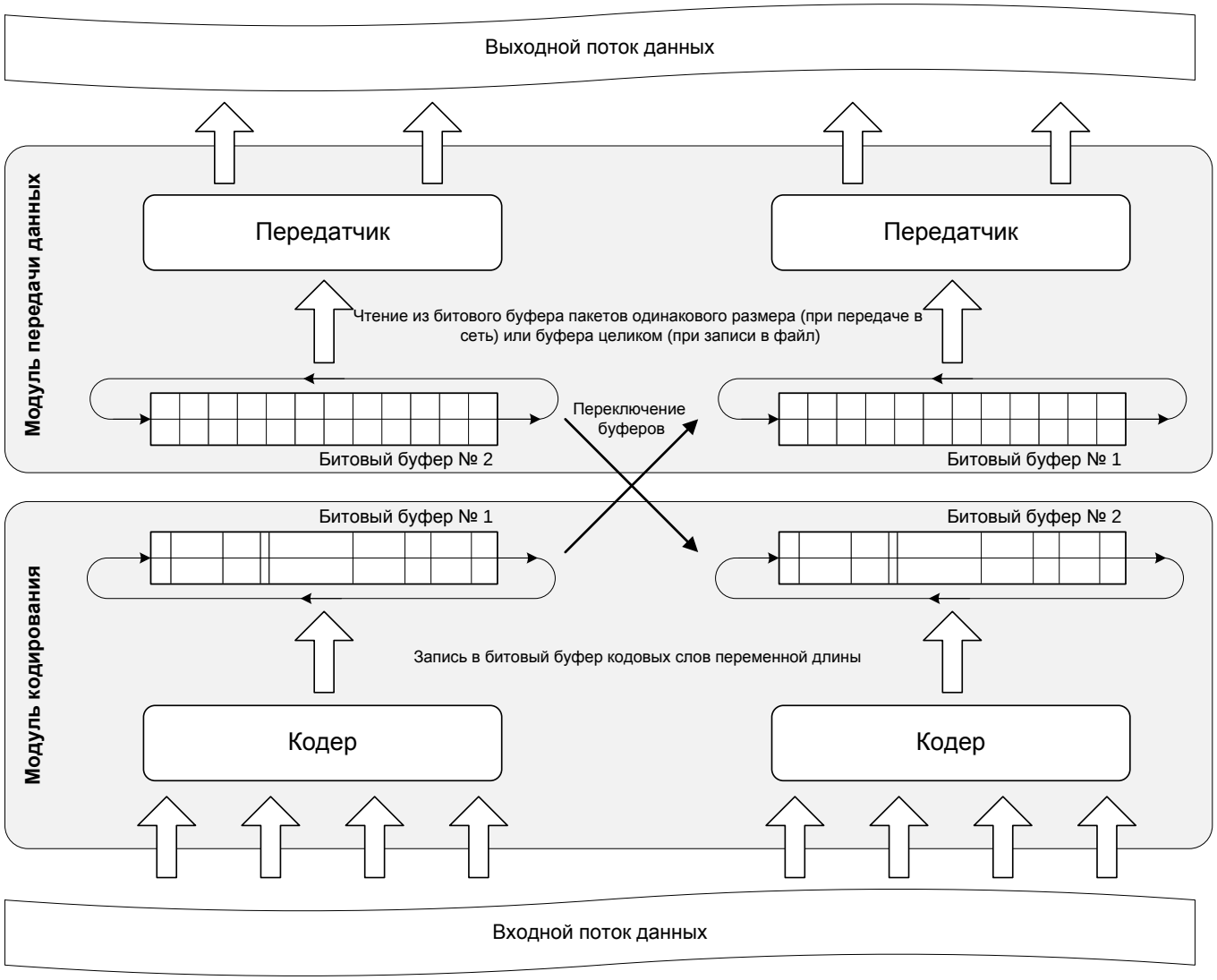


Рисунок 1 – Реализация ввода-вывода данных на основе двух массивов данных в системах связи с кодированием информации в реальном масштабе времени

Пока кодер последовательно записывает обработанные данные в один массив, модуль передачи данных одновременно передает в канал сформированный ранее другой массив. После завершения обеих процедур указатели на массивы обмениваются значениями. Аналогичные рассуждения справедливы также при буферизации приема данных из канала связи или при чтении с диска. Основной проблемой при такой реализации является согласование длительностей выполнения процедур заполнения битового буфера (текущего массива) и вывода

буфера (ранее заполненного массива) между собой, а также между устройством передачи данных и приемником.

Если оставить в стороне вопросы реализации системы кодирования/декодирования данных в реальном масштабе времени, то можно сформулировать основные требования, предъявляемые к программной реализации битового буфера для создания моделей систем кодирования информации и связи:

- фиксированный размер битового буфера, который зависит от ограничений по объему оперативной памяти и допустимому времени задержки при передаче информации;
- возможность записи/чтения в битовый буфер битовых полей различной длины (обычно от 1 до N битов, причем значение N зависит от конкретного алгоритма обработки данных);
- контроль состояния заполненности битового буфера и автоматический его сброс на диск (или в канал связи) при переполнении;
- использование битового буфера либо только для записи (при кодировании данных), либо только для чтения (при декодировании данных).

В качестве дополнительных требований, предъявляемых к разрабатываемым структурам данных и методам работы с ними, можно выделить требование универсальности, т.е. возможность использования кода во всех приложениях, ориентированных на разработку программных моделей, связанных с кодированием или перекодированием информации.

2. Принципы хранения и доступа к данным в битовом буфере

Принципы функционирования битового буфера в режиме записи (кодирование) и чтения (декодирование) проиллюстрированы на рисунке 2.

Рассмотрим более подробно принципы реализации каждого из блоков схем, приведенных на рисунке 2.

Описание структур данных битового буфера не представляет сложности. Для хранения и доступа к данным можно использовать класс cBitStream со следующими скрытыми (private) данными:

```
class cBitStream {
     unsigned char *Buffer;             // временный буфер
     ModeType CurMode;                  // режим работы с буфером
     ofstream ofp;                      // файл (при записи)
     ifstream ifp;                      // файл (при чтении)
     unsigned long int CurBufferBitSize; // тек. размер буфера в битах
     unsigned long int ptrCurrentBit;   // указатель на текущий бит
    }
```

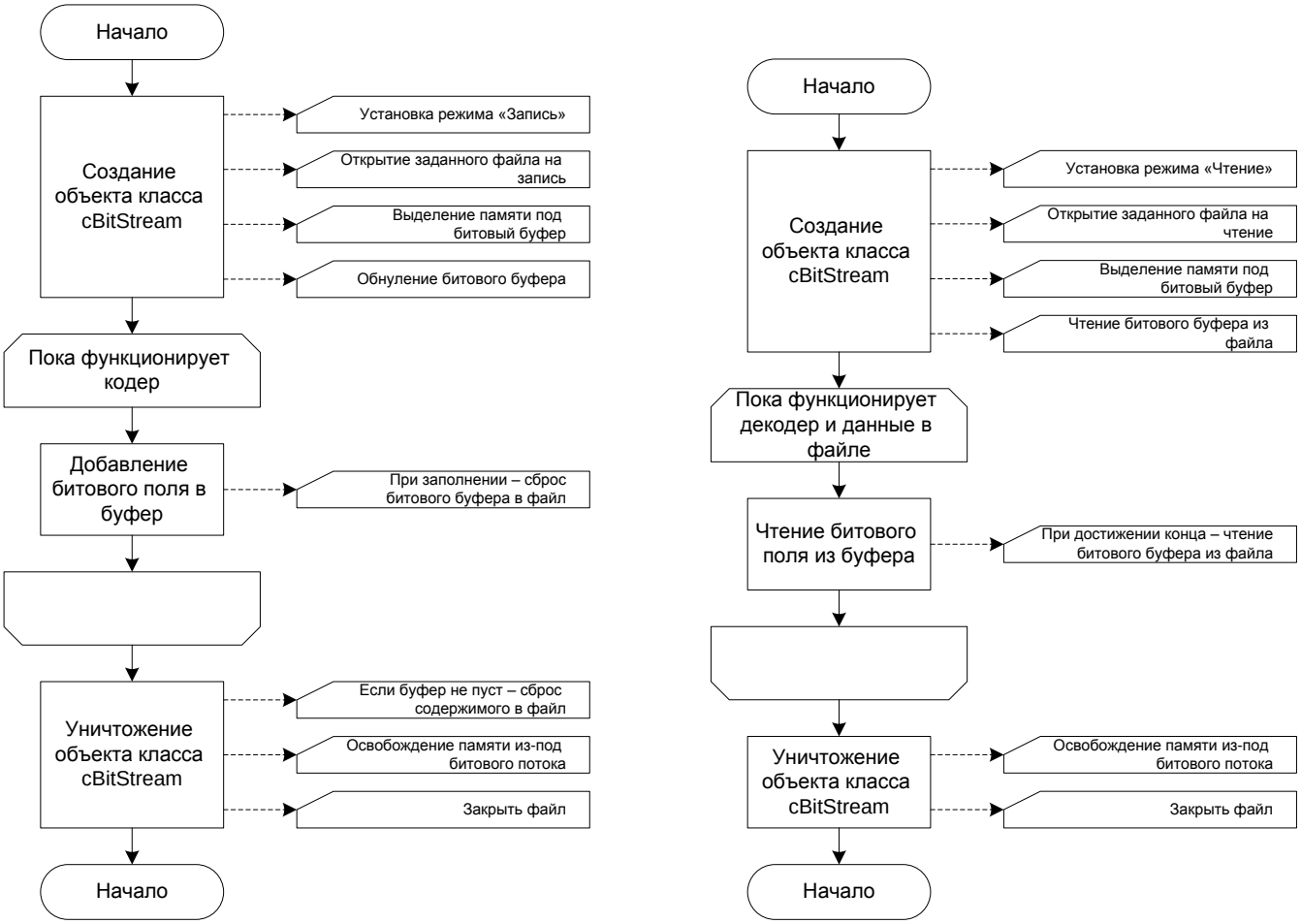


Рисунок 2 – Схема функционирования битового буфера в режиме записи (а) и в режиме чтения (б)

Для хранения данных удобно использовать массив байтов. Выделение памяти под массив (битовый буфер) может выполняться стандартным способом (командами new и delete). Ниже будет рассмотрен вариант записи/чтения данных в буфер посредством битовых полей (нескольких смежных битов), при котором доступ к буферу осуществляется порциями по 4 байта. Поэтому при выделении памяти под битовый буфер целесообразно выделить на 3-4 байта больше заданного размера буфера. Эта дополнительная память не будет использоваться для хранения данных, но позволит избежать наложения записываемых/читаемых порций на другие области памяти.

Если объект класса cBitStream создан для записи, то целесообразно вызвать метод обнуления выделенной области памяти – повторение этой процедуры после каждого сбрасывания буфера в файл позволяет несколько повысить производительность процедуры записи битовых полей.

Переменная CurMode также инициализируется при создании объекта класса cBitStream и может принимать одно из двух значений, определенных перечисляемым типом ModeType и соответствующих текущему режиму работы буфера – "чтение" или "запись":

```
enum ModeType {
     mRead,
     mWrite
    };
```

Из переменных ofp и ifp при создании буфера инициализируется только одна – в зависимости от того, в каком режиме будет работать буфер[1].

Переменная CurBufferBitSize инициализируется значением, равным количеству битов в буфере. Ее значение может измениться при выполнении чтения файла в битовый буфер (режим чтения буфера), когда буфер заполняется не полностью. Контроль данной границы в методах чтения битовых полей позволяет предотвратить чтение "лишних" битов, не записанных в битовый поток кодером.

Для перемещения по буферу (с целью чтения/записи данных, а также контроля состояния заполненности буфера) требуется хранить указатель на текущую позицию в буфере. Делать это можно двумя способами:

- хранить смещение текущего байта (ptrCurrentByte) в байтах от начала буфера и номер первого свободного бита в нем (BitIndex).

- хранить указатель на текущий бит в буфере (ptrCurrentBit) в байтах от начала буфера.

При использовании двух указателей (на текущий байт и на бит в нем) мы всегда имеем готовый указатель на текущее слово в буфере, однако каждая операция доступа к буферу для записи/чтения очередного битового поля длиной CodeBitLength требует согласованной модификации двух переменных:

```
ptrCurrentByte += (BitIndex + CodeBitLength) >> 3;
BitIndex = (BitIndex + CodeBitLength) & 0x7;
```

Проверка возможности размещения в буфере очередного битового поля длиной CodeBitLength может быть выполнена, например, следующим образом:

```
if (((ptrCurrentByte << 3) + BitIndex + CodeBitLength) >=
cnBufferBitSize)
     {
          ...
     }
```

Использование варианта с одним указателем позволяет хранить в памяти только одну переменную для обозначения текущей позиции, что снимает проблему согласования ptrCurrentByte и BitIndex. Адрес текущего байта и бита при каждом обращении к буферу в этом случае придется вычислять следующим образом:

```
ptrByte = ptrCurrentBit >> 3;           // ptrCurrentBit / 8
BitIndex = ptrCurrentBit & 0x7;         // ptrCurrentBit % 8
```

[1] Вместо двух объектов классов ifstream или ofstream можно было использовать один объект класса fstream.

А запись/чтение битового поля длиной CodeBitLength в битовый буфер потребует единственной модификации:

```
ptrCurrentBit += CodeBitLength;
```

Проверка возможности размещения в буфере очередного битового поля длиной CodeBitLength может быть еще проще, чем в предыдущем случае:

```
if ( (ptrCurrentBit + CodeBitLength) >= cnBufferBitSize )
 {
      ...
 }
```

В дальнейшем для указания на первый свободный бит буфера будем использовать указатель ptrCurrentBit.

3. Варианты реализации процедур записи/чтения данных в битовый буфер

3.1. Соглашение о способе представления данных в битовом буфере

При рассмотрении способов записи/чтения данных (кодовых слов) в битовый буфер будем придерживаться порядка следования байтов и битов в битовом буфере и кодовом слове, приведенного на рисунке 3.

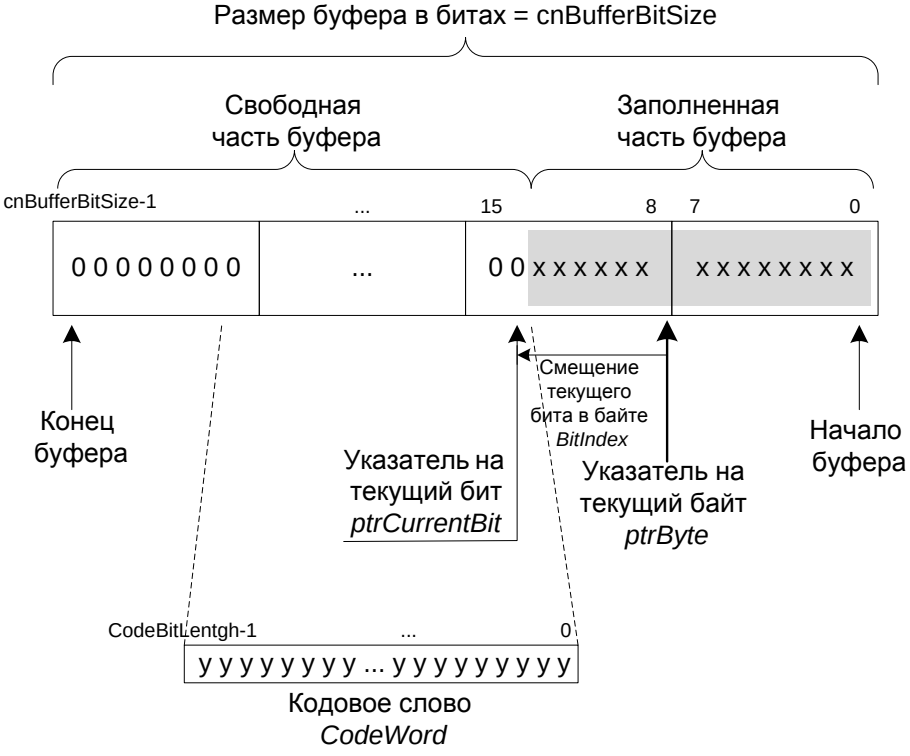


Рисунок 3 – Иллюстрация порядка следования байтов и битов в битовом буфере и кодовом слове

Такой способ представления порядка следования байтов несколько отличается от традиционного способа описания содержимого памяти ЭВМ, однако соответствует действительности и, главное, не требует "мысленного" переупорядочивания байтов, что упрощает разработку и анализ процедур работы с буфером.

Процедура записи кодового слова (битового поля) в битовый буфер обычно называется "депонированием", а процедура чтения кодового слова (битового поля) из битового буфера – "выделением.

3.2. Запись/чтение кодовых слов по одному биту

Наиболее простым способом программной реализации битового буфера является побитовый способ добавления/извлечения данных (см. рисунок 4). Преимущества данного подхода:

- быстрота разработки программного кода за счет того, что при записи/чтения битового поля выполняется циклический вызов универсальной простой процедуры записи/чтения одного бита;

- проверка возможности размещения очередного битового поля в буфере вырождается в проверку выхода указателя за границу буфера после записи/чтения очередного бита.

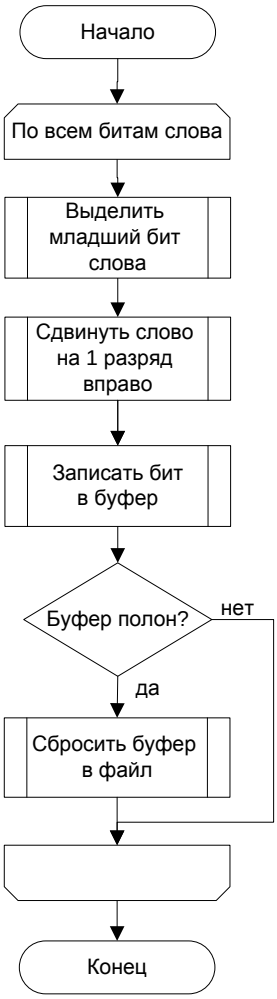
Программный код, выполняющий собственно запись бита DataBit по текущему положению указателя буфера, можно записать так:

```
ptrByte = ptrCurrentBit >> 3;           // текущий байт (начало порции)
BitIndex = ptrCurrentBit & 0x7;         // смещение бита в байте

if (DataBit == 1)                       // записываем если бит = 1
     *(Buffer + ptrByte) |= (1 << BitIndex); // наложить бит
ptrCurrentBit++;                        // сместить указатель
```

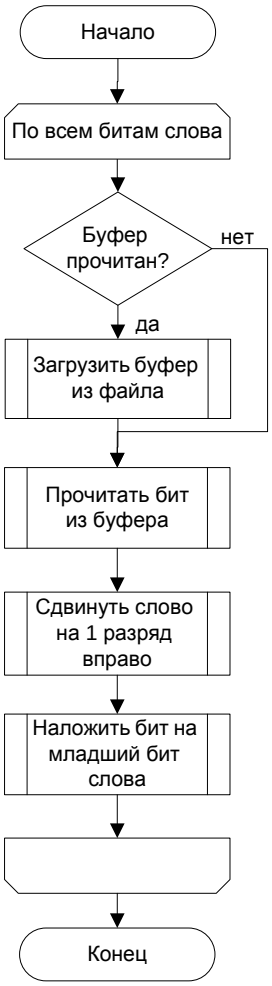
Аналогично для чтения бита по текущему положению указателя буфера:

```
ptrByte = ptrCurrentBit >> 3;           // текущий байт (начало порции)
BitIndex = ptrCurrentBit & 0x7;         // смещение бита в байте

*DataBit = (*(Buffer + ptrByte) >> BitIndex) & 0x1;    // в младший бит
ptrCurrentBit++;                        // сместить указатель
```

Обратите внимание, что на блок-схемах проверка момента достижения конца буфера выполняется при записи потока – после записи очередного бита, а при чтении потока – до чтения очередного бита. Если не придерживаться этого правила, то возможны ситуации, когда в файл будет записан неполный буфера или же будет предпринята попытка чтение из файла несуществующих данных.



а)



б)

Рисунок 4 – Блок-схемы алгоритмов записи (а) и чтения (б) очередного битового поля в битовый буфер

Максимальная длина битового поля, которое можно записать/прочитать в буфер с использованием приведенных на блок-схемах алгоритмов и в примерах программного кода, составляет 32 бита (тип данных unsigned long int).

Недостатком данного варианта является снижение быстродействия работы с битовым буфером из-за необходимости постоянного вызова процедур побитовой записи/чтения, что особенно проявляется при работе записи/чтения битовых полей большой длины (более 8-10 битов). Впрочем, при inline-реализации этих процедур снижение быстродействия оказывается менее заметным. Поэтому в большинстве случаев для практической реализации бывает достаточно именно данного подхода. Больше того, для тех алгоритмов декодирования (именно декодирования), которые предполагают побитовый анализ входного битового потока и не позволяют заранее определить длину читаемого битового поля – например, для алгоритма Хаффмана – такой подход при декодировании данных является наиболее приемлемым.

3.3. Запись/чтение кодовых слов битовыми полями

Вторым вариантам программной реализации методов работы с битовым буфером является использование универсальных процедур записи/чтения битовых полей произвольной длины. Такой вариант более предпочтителен при записи/чтении длинных битовых полей. Более высокая производительность может достигаться за счет записи/чтения битового поля целиком, а не по одному биту. При этом из буфера каждый раз выделяется некоторая рабочая область (порция), равная разрядности регистра процессора, внутри которой и выполняются процедуры наложения или извлечения битового поля (см. рисунок 5).

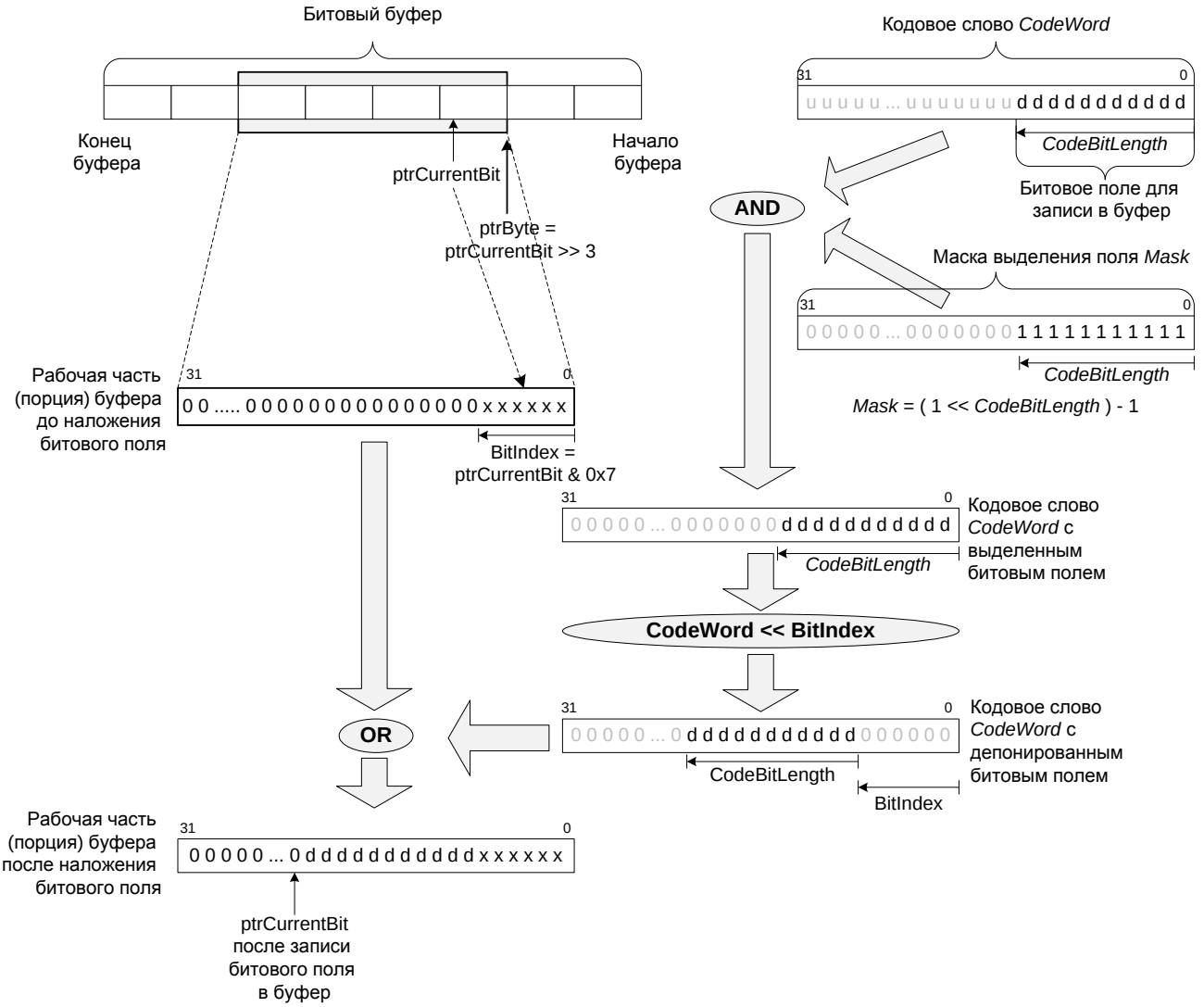


Рисунок 5 – Иллюстрация работы с битовым буфером при использовании универсальных процедур записи/чтения битовых полей произвольной длины

В виде программного кода приведенную на рисунке схему записи битового поля в битовый буфер можно записать следующим образом:

```
ptrByte = ptrCurrentBit >> 3;          // текущий байт (начало порции)
```

```
BitIndex = ptrCurrentBit & 0x7;        // смещение бита в байте

Mask = ((1 << CodeBitLength) - 1);     // генерируем маску

CodeWord = CodeWord  & Mask;           // накладываем маску
CodeWord = CodeWord << BitIndex;       // депонируем поле

CurPortion = *(unsigned long int *)(Buffer + ptrByte); //взять порцию
CurPortion = CurPortion | CodeWord;              // накладываем поле
*(unsigned long int *)(Buffer + ptrByte) = CurPortion; // обратно

ptrCurrentBit += CodeBitLength;        // сдвигаем указатель

if (ptrCurrentBit == cnBitStreamBitSize) // проверяем полноту буфера
    Сбросить_Буфер_В_Файл();
```

Операция выделения битового поля в младших CodeBitLength битах кодового слова CodeWord используется для "сброса" неиспользуемых битов кодового слова. Если вызывающая программа сама обнуляет "лишние" биты, то эту операцию можно опустить, тем более, что формирование маски наряду с операцией депонирования битового поля (позиционирования его на "нужное" место) являются, пожалуй, наиболее трудоемкими фрагментами процедуры записи битового поля из-за необходимости выполнения команды сдвига на CodeBitLength[2]. Особенно осторожно следует исключать операцию маскирования в том случае, если обрабатываемые слова являются знаковыми или могут содержать отсекаемые старшие биты.

Аналогично можно записать программный код для чтения битового поля из буфера:

```
ptrByte = ptrCurrentBit >> 3;          // текущий байт (начало порции)
BitIndex = ptrCurrentBit & 0x7;        // смещение бита в байте

Mask = ((1 << CodeBitLength) - 1);     // генерируем маску

*CodeWord = 0;                         // обнулить кодовое слово

if (ptrCurrentBit == CurBufferBitSize) // проверить наличие данных
    Загрузить_Буфер_Из_Файла();

*CodeWord = *(unsigned long int *)(Buffer + ptrByte);// читаем порцию
*CodeWord = (*CodeWord ) >> BitIndex;  // выделяем поле
*CodeWord = (*CodeWord) & Mask;        // обнулить лишние биты

ptrCurrentBit += CodeBitLength;        // сдвигаем указатель
```

[2] Скорость выполнения команды сдвига с фактором сдвига N>1 зависит от архитектуры процессора: часть процессоров выполняют команду сдвига вида shl REG, N за несколько тактов, в зависимости от значения фактора сдвига N (грубо говоря, за один такт выполняется сдвиг только на один разряд), а, например, DSP-процессоры выполняются команду сдвига регистра всегда за один такт независимо от величины фактора сдвига.

Если сравнивать приведенные выше процедуры записи и чтения битовых полей, то можно заметить, что завершение буфера контролируется по-разному: при записи указатель текущего бита сравнивается с числом битов в буфере (константа cnBitStreamSize), а при чтении – с числом битов, реально прочитанных из файла (переменная CurBufferBitSize). Это позволяет отследить ситуацию, когда декодер пытается прочитать (обычно ошибочно) из битового буфера данные, которых там нет.

Основным недостатком варианта работы с битовым буфером с использованием битовых полей является усложнение записи/чтения данных в конце битового буфера, когда записываемое/читаемое битовое поле не помещается в буфер целиком. В такой ситуации:

- при записи: следует разбить записываемое слово на две части (с учетом остающихся свободных битов буфера) и выполнить две процедуры записи, между которыми вызвать процедуру сброса буфера на диск;

- при чтении: дважды выполнить процедуру чтения битового поля, между которыми вставить вызов операции загрузки следующего буфера из файла, и затем "склеить" два битовых поля.

Результирующие блок-схемы алгоритмов записи и чтения битового поля в буфер приведены на рисунках 6 и 7. Программную реализацию этих блок-схем можно найти в описании класса BitStream в Приложении.

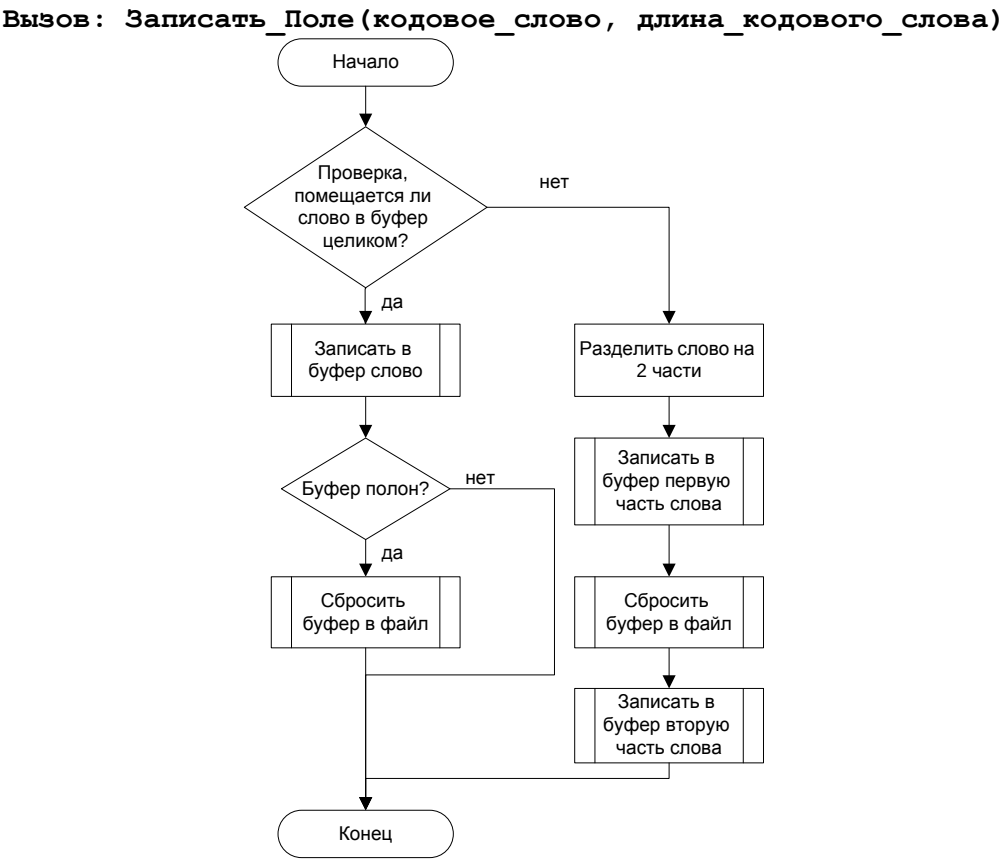


Рисунок 6 – Блок-схема алгоритма записи очередного битового поля в битовый буфер

Вызов: Прочитать_Поле(адрес(кодовое_слово), длина_кодового_слова)

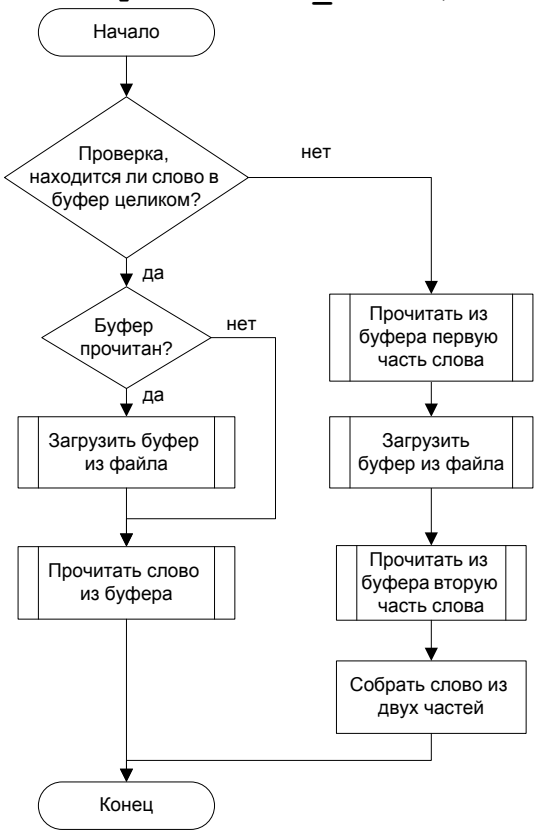


Рисунок 7 – Блок-схема алгоритма чтения очередного битового поля в битовый буфер

При написании методов работы с битовым буфером, основанным на записи/чтении битовых полей произвольной длины следует оценить максимальную длительность кодовых слов (битовых полей), которые могут быть записаны и прочитаны из битового буфера за одну операцию записи/чтения. Поскольку разрядность регистров общего назначения большинства современных процессоров составляет 32 бита, то в общем случае (если слово помещается в буфер целиком, т.е. еще не конец буфера) за одну операцию всегда можно записать/прочитать в буфер слово размером до (32-7) = 25 битов[3].

Сравнительный анализ приведенных в данном пособии примеров программной реализации двух рассмотренных выше способов записи/чтения битовых полей в битовый буфер показал, что в общем случае вариант с записью/чтением битовых полей в несколько раз превосходит по производительности вариант с побитовой записью/чтением кодовых слов. При этом надо понимать, что выигрыш увеличивается с увеличением средней длины битовых полей, подлежащих записи в битовый поток. Если же, например, большинство

[3] Такой "минимальный" размер битового поля, который гарантированно помещается в рабочую порцию буфера, имеет место, когда указатель текущего бита ptrCurrentBit указывает на самый последний бит байта. В этом случае рабочая порция начинается с начала этого байта и 7 битов из 8 этой порции уже использованы. В другом крайнем случае, если указатель ptrCurrentBit указывает на начало байта, то за одну операции можно легко записать/прочитать в буфер кодовое слово размером до (32-0) = 32 битов.

битовых полей будут однобитовыми, то преимущества по производительности второго варианта будут минимальны.

3.4. Запись/чтение длинных битовых полей

При реализации некоторых алгоритмов кодирования (например, в области защиты информации и помехоустойчивого кодирования), появляется необходимость записывать/читать в буфер кодовые слова длиной до 100 и более битов. Понятно, что битовое поле такого размера не может быть размещена в регистре общего назначения процессора. Поэтому в данном случае целесообразно в цикле записывать/читать кодовое слово в битовый буфер по частям, вызывая в цикле процедуры записи/чтения битовых полей.

В приведенном примере в качестве таких "частей" используются 16-битовые составляющие целого длинного битового поля. В этом случае для записи/чтения 16-битовых частей поля в буфер можно использовать рассмотренные выше процедуры записи/чтения битовых полей[4].

Примеры программного кода для записи и чтения длинного битового поля частями по 16 битов приведен ниже. Для хранения длинного кодового слова используется массив элементов типа unsigned long int (в процедуру передается адрес массива), в котором биты данных упакованы по 32 бита в каждом 4-байтовом слове (а не по одному биту в слове!)[5]. Полная длина битового поля по-прежнему указывается в битах.

Локальные структуры данных, общие для процедур записи и чтения длинных битовых полей:

```
unsigned int NumWords;        // количество полных 16-битовых частей
unsigned int RemainBits;      // количество оставшихся битов

unsigned long int PartWord;   // хранение текущей 16-битовой части
unsigned int *ptrWord;        // текущая 16-битовая часть
```

Программный код процедур записи и чтения длинных битовых полей можно записать следующим образом:

[4] Чем больше длина поле, тем выше быстродействие процедур работы с буфером. Рассмотренные выше процедуры записи/чтения битовых полей работают с полями длиной до 25 битов. Однако разбиение кодового слова на 24-битовые части несколько сложнее, чем на 16-битовые. Поэтому 2-х байтовые слова, по-видимому, представляют собой разумный компромисс между производительностью и сложностью реализации.

[5] Опыт показывает, что зачастую программисты, которые впервые реализуют кодирования/декодирования данных с использованием буферизации, представляют сформированное кодером кодовое слово в виде массива элементов char, где в каждом байте хранится только один информационный бит (обычно в бите № 0). Вместо того, чтобы разрабатывать дополнительные процедуры упаковки/распаковки такого "массива кодовых битов" в кодовое слово, проще переписать методы класса cBitStream таким образом, чтобы процедуры записи/чтения кодового слова в буфер на вход принимали данные требуемого типа.

```
Записать_ДлинноеПоле (unsigned long int *CodeWord,
                      unsigned int CodeBitLength)
{
 ...
 NumWords = CodeBitLength >> 4;
 RemainBits = CodeBitLength & 0xF;
 ptrWord = (unsigned int *)CodeLongWord;

 for (int nw = 0; nw<NumWords; nw++)    // полные 16-битовые части
 {
     PartWord = *ptrWord++;             // читать + сдвинуть указатель
     Записать_Поле(PartWord, 16);       // записать 16-битную часть
 }
 if (RemainBits)                        // если остались биты
 {
     PartWord = *ptrWord;               // читать
     Записать_Поле(PartWord, RemainBits);    // записать биты
 }
}
```

```
Прочитать_ДлинноеПоле (unsigned long int *CodeWord,
                       unsigned int CodeBitLength)
{
 ...
 NumWords = CodeBitLength >> 4;
 RemainBits = CodeBitLength & 0xF;
 ptrWord = (unsigned int *)CodeLongWord;

 for (int nw = 0; nw<NumWords; nw++)    // полные 16-битовые части
 {
     Прочитать_Поле(&PartWord, 16);     // прочитать 16-битное поле
     *ptrWord++ = PartWord;             // писать + сдвинуть указатель
 }
 if (RemainBits)                        // если остались биты
 {
     Прочитать_Поле(&PartWord, RemainBits);  // прочитать биты
     *ptrWord = PartWord;               // записать
 }
}
```

3.5. Дополнительные способы повышения быстродействия процедур работы с битовым буфером

На практике часто приходится иметь дело с приложениями, в которых размер кодовых слов, формируемых кодером, фиксирован (алгоритмы ИКМ, ДИКМ, ДМ при компрессии аудио и видеосигналов, обработка 12/24-битных отсчетов в компьютерной графике и системах мультимедиа и т.п.). В таких кодерах/декодерах подавляющая часть кодовых слов имеют один и тот же размер, и лишь (возможно)

малая доля информации (заголовки, управляющие коды и т.п.) должна записываться в буфер битовыми полями иного размера.

Выше отмечалось, что одной из наиболее "медленных" операций при выделении и депонировании кодовых слов является операция формирования битовой маски для "обрезания" битового поля и сброса неиспользуемых битов путем использования команды сдвига. Когда размер битового поля фиксирован, маску можно задать константой и получить за счет этого некоторый выигрыш по производительности (правда, очень незначительный).

Если имеются кодовые слова двух размеров, то придется использовать для каждого типа кодового слова собственную процедуру записи/чтения в битовый поток.

При этом снижается универсальность кода: если алгоритм кодирования будет изменен и появится кодовое слово "новой" длины, то потребуется модернизация процедур работы с битовым буфером. Тем не менее, применение такого подхода обосновано, например, при блочном кодировании с использованием кодовых слов одинаковой длины – когда подавляющий процент записываемых в битовый буфер сообщений имеет одинаковую длину.

Еще одним приемом для повышения производительности работы с битовым буфером, является использование порций размером 64 бита при записи и чтении битовых полей. Для хранения таких порций можно использовать MMX-регистры процессора.

4. Особенности реализации методов работы с битовыми массивами в стандартных библиотеках

В набор библиотек среды Visual Studio входит модуль, содержащий описание класса bitset. Этот класс позволяет работать с битовым потоком как с массивом данных фиксированного размера, содержит различные методы не только для добавления/извлечения битов, но и для выполнения логических операций и сдвигов над битами. С точки зрения использования данного класса при разработке кодеров/декодеров данных он имеет два существенных недостатка:

- класс bitset никак не связан с внешним устройством передачи или хранения данных (файлом). Поэтому программисту необходимо будет разрабатывать собственные методы контроля состояния заполненности буфера и самостоятельно сбрасывать его на диск (при записи) или загружать с диска (при чтении);

- все операции (кроме инициализации битового буфера при создании) выполняются на уровне битов, а не битовых полей, что, естественно, снижает производительность такой реализации.

Аналогичный класс (bitset) имеется в библиотеке Borland Builder. Он имеет большие возможности, нежели bitset от Microsoft, позволяет работать с битовыми полями, но также не связан с файлом.

В Интернете можно найти множество примеров реализации классов, подобных bitset. Часть из них позволяет связать битовый буфер с файлом и по заполнении буфера автоматически сбрасывать его в файл (или загружать из файла при опустошении). Особенностью других реализаций является динамическое изменение размера битового буфера при выходе указателя за его границу, т.е. переменные размер битового буфера (в контексте рассматриваемых приложений такой вариант неприемлем, однако для других областей применения он может быть весьма полезен).

В приложении А приведен пример программной реализации класса cBitStream для работы с битовым потоком, который содержит основные поля данных и методы для работы с битовым буфером. Данный набор средств достаточен для выполнения типовых операций, которые используются при построении программных моделей алгоритмов эффективного и помехоустойчивого кодирования (и декодирования) данных.

Для других приложений может потребоваться модификация программного кода.

Список использованных источников

1. Microsoft MSDN Library for Visual Studio 2005 - http://msdn2.microsoft.com/ru-ru/default.aspx, 2006.

2. Hatem Mostafa Bits Array Encapsulation - http://www.codeproject.com/cpp/BitArray.asp, 2004.

3. Вахтеров М., Орлов С. Четвертый BORLAND C++ и его окружение - http://cs.mipt.ru/docs/comp/rus/os/windows/software/develop/bcpp4/index.htm, 1994г.

Приложение А. Пример описания и программной реализации класса cBitStream для работы с битовым буфером

```
/* ------------------------------------------------------------------------------- */
/*                      Класс работы с битовым потоком                             */
/* Запись/чтение битовых полей - побитовое.                                        */
/* Максимальный размер кодового слова - 25 битов при записи/чтении полями          */
/*                                    - 32 бита при записи/чтении побитово         */
/*                                    - неогр. при записи/чтении "длинными полями" */
/* Минимальный размер буфера - не менее 32 битов (4 байта)                         */
/* Буфер работает или только на запись (при кодировании) или только на чтение      */
/* (при декодировании)                                                             */
/* ------------------------------------------------------------------------------- */
```

Содержимое файла cBitStream.h

```
#pragma once
#include <fstream>
using namespace std;

// если доступ в побитовом режиме - то определить ACCESSMODE_BIT
// если доступ в режиме полей, то убрать директиву
#define ACCESSMODE_BIT

// режимы работы с буфером (чтение/запись)
enum ModeType {
	mRead,
	mWrite
};

// Размеры битового буфера в битах и байтах
const unsigned long int cnBitStreamByteSize = 65536;
const unsigned long int cnBitStreamBitSize = cnBitStreamByteSize*8;

//---------------------------------------------------------------------------
// Описание класса cBitStream
//---------------------------------------------------------------------------
class cBitStream {
	unsigned char *Buffer;                          // временный буфер
	unsigned long int ptrCurrentBit;                // указатель на текущий (свободный) бит
	ofstream ofp;                                   // файл (при записи)
	ifstream ifp;                                   // файл (при чтении)
	ModeType CurMode;                 // текущий режим работы с буфером (запись/чтение)
	unsigned long int CurBufferBitSize;             // текущий размер буфера в битах
public:
	cBitStream(unsigned char *FileName, ModeType mode);  // конструктор
	~cBitStream(void);                                   // удаление буфера

	#ifdef ACCESSMODE_BIT
		// добавить/прочитать в буфер кодовое слово побитово
	  void WriteBitField(unsigned long int CodeWord, unsigned int CodeBitLength);
	  void ReadBitField(unsigned long int *CodeWord, unsigned int CodeBitLength);
	#else
		// добавить/прочитать в буфер кодовое слово битовым полем
	  void WriteField(unsigned long int CodeWord, unsigned int CodeBitLength);
	  void ReadField(unsigned long int *CodeWord, unsigned int CodeBitLength);
		// добавить/прочитать в буфер кодовое слово длинным битовым полем
	  void WriteLongField(unsigned long int *CodeLongWord, unsigned int CodeBitLength);
	  void ReadLongField(unsigned long int *CodeLongWord, unsigned int CodeBitLength);
	#endif

private:
	void PutBit(unsigned char DataBit);                 // добавить в буфер бит данных
	void GetBit(unsigned char *DataBit);                // взять из буфера бит данных
	void FlushBuffer(void);                             // сбросить буфер на диск
	void LoadBuffer(void);                              // загрузить буфер с диска
	void ClearBuffer(void);                             // очистить буфер
};
//---------------------------------------------------------------------------
```

Содержимое файла cBitStream.cpp

```
#include "StdAfx.h"
#include <assert.h>
#include "cBitStream.h"


//-------------------------------------------------------------------------------
// Создание битового буфера - выделение памяти, очистка
//-------------------------------------------------------------------------------
cBitStream::cBitStream(unsigned char *FileName, ModeType mode)
{
	CurMode = mode;

	// выделить память под буфер с учетом чтения последнего байта как unsigned long int
		Buffer = new unsigned char[cnBitStreamByteSize+4];

		// в зависимости от типа
	if (CurMode == mWrite)
	{
		ofp.open((char *)FileName, ios::binary);
		assert(!ofp.fail());
		ClearBuffer();                                  // очистить буфер
	}
	else if (CurMode == mRead)
	{
		ifp.open((char *)FileName, ios::binary);
		assert(!ifp.fail());
		LoadBuffer();                                   // загрузить буфер
	}
}
//-------------------------------------------------------------------------------



//-------------------------------------------------------------------------------
// Очистить буфер. Процедура вызывается только если буфер открыт на запись
//-------------------------------------------------------------------------------
void cBitStream::ClearBuffer(void)
{
	memset(Buffer, 0, cnBitStreamByteSize);             // упрощает запись битов в буфер
	ptrCurrentBit = 0;                                  // сбросить указатель
	CurBufferBitSize = cnBitStreamBitSize;              // размер буфера
}
//-------------------------------------------------------------------------------



//-------------------------------------------------------------------------------
// Сбросить содержимое битового буфера на диск
//-------------------------------------------------------------------------------
void cBitStream::FlushBuffer(void)
{
	if (CurMode == mWrite)                              // только буфер в режиме записи
	{
		if (ptrCurrentBit != cnBitStreamBitSize)        // заполненная часть буфера
			CurBufferBitSize = (ptrCurrentBit + 0x7) & 0xFFFFFFF8; // округлить вверх
			// сбросить буфер на диск
		ofp.write((char *)Buffer, (CurBufferBitSize >> 3) );
			// выполнить очистку буфера
		ClearBuffer();
	}
}
//-------------------------------------------------------------------------------
```

```
//-------------------------------------------------------------------------------
// Прочитать содержимое битового буфера с диска
//-------------------------------------------------------------------------------
void cBitStream::LoadBuffer(void)
{
	if (CurMode == mRead)                                  // только буфер в режиме чтения
	{
			// загрузить буфер из файла
		assert(!ifp.eof());              // дошли до конца файла, а данные еще требуются
		ifp.read((char *)Buffer, cnBitStreamByteSize);
		CurBufferBitSize = ifp.gcount() << 3;  // сколько реально битов было прочитано
		ptrCurrentBit = 0;
	}
}
//-------------------------------------------------------------------------------



//-------------------------------------------------------------------------------
// Прочитать очередной бит из битового буфера
//-------------------------------------------------------------------------------
inline void cBitStream::GetBit(unsigned char *DataBit)
{
	*DataBit = (*(Buffer + (ptrCurrentBit >> 3) ) ) >> (ptrCurrentBit & 0x7) & 0x01;
	ptrCurrentBit++;                                       // сместить указатель
}


#ifdef ACCESSMODE_BIT

//-------------------------------------------------------------------------------
// Прочитать кодовое слово CodeWord длиной CodeBitLength битов из битового буфера
// последовательно, по одному биту
//-------------------------------------------------------------------------------
void cBitStream::ReadBitField(unsigned long int *CodeWord, unsigned int CodeBitLength)
{
  unsigned char CurDataBit;
  unsigned long int Mask=1;

  if (CurMode == mRead)                                  // если буфер в режиме чтения
  {
	*CodeWord = 0;
	for (unsigned int i=0; i<CodeBitLength; i++)
		{
			if (ptrCurrentBit == CurBufferBitSize)  // проверить, пройден ли буфер
				LoadBuffer();
			GetBit(&CurDataBit);
			*CodeWord |= ((unsigned long int)(CurDataBit) << i);
	}
  }
}
//-------------------------------------------------------------------------------

#else

//-------------------------------------------------------------------------------
// Прочитать кодовое слово CodeWord длиной CodeBitLength битов из битового буфера
// как целое битовое поле. Максимальная длина кодового слова - 25 битов
//-------------------------------------------------------------------------------
void cBitStream::ReadField(unsigned long int *CodeWord, unsigned int CodeBitLength)
{
  unsigned long int PartLen;

  if (CurMode == mRead)                                  // если буфер в режиме чтения
  {
		// сначала обнулить кодовое слово
	*CodeWord = 0;
		// если кодовое слово целиком находится в буфере - то просто вытащить его
	if ((ptrCurrentBit + CodeBitLength) <= CurBufferBitSize)
	{
```

```
                // выполнить проверку состояния заполненности буфера
            if (ptrCurrentBit == CurBufferBitSize)
                LoadBuffer();
            *CodeWord = (*(unsigned long int *)((Buffer + (ptrCurrentBit >> 3))))
                        >> (ptrCurrentBit & 7);
            *CodeWord = (*CodeWord) & ((1 << CodeBitLength) - 1); // наложить маску
            ptrCurrentBit += CodeBitLength;
        }
        else
        {   // если кодовое слово не помещается целиком в буфер,
            // то надо его делить на 2 части или читать по одному биту
            // (эта ветка выполняется редко, один раз на буфер)
            PartLen = CurBufferBitSize - ptrCurrentBit;          // длина помещающейся части
                // читаем первую (младшую) часть кодового слова
            *CodeWord = (*(unsigned long int *)((Buffer + (ptrCurrentBit >> 3))))
                        >> (ptrCurrentBit & 7);
            *CodeWord = (*CodeWord) & ((1 << PartLen) - 1);
            ptrCurrentBit += PartLen;               // сдвинуть указатель на конец буфера
            LoadBuffer();                           // сбросить буфер на диск
                // читаем и накладываем на CodeWord оставшуюся часть кодового слова
            PartLen = CodeBitLength - PartLen;
            *CodeWord |= ((*(unsigned long int *)(Buffer)) & ((1 << PartLen) - 1))
                        << PartLen;
            ptrCurrentBit += PartLen;                           // сдвинуть указатель
                // заполненность буфера не проверяем - он больше длины одного слова
        }
    }
}
//---------------------------------------------------------------------------


//---------------------------------------------------------------------------
// Прочитать длинное кодовое слово CodeWord длиной CodeBitLength битов из битового буфера.
// Чтение выполняется битовыми полями по 16 битов + остаток.
// Длина кодового слова не ограничена.
// Массив CodeLonbgWord должен быть выровнен по границе слова (16 битов).
//---------------------------------------------------------------------------
void cBitStream::ReadLongField(unsigned long int *CodeLongWord, unsigned int CodeBitLength)
{
    unsigned long int PartWord;             // очередная 16-битовая часть кодового слова
    unsigned int NumWords = CodeBitLength >> 4;   // количество полных 16-битовых частей
    unsigned int RemainBits = CodeBitLength & 0x0A;      // количество оставшихся битов
    unsigned int nw = 0;

    unsigned int *ptrWord = (unsigned int *)CodeLongWord;

    if (CurMode == mRead)                               // если буфер для чтения
    {
        for (nw = 0; nw<NumWords; nw++)                 // сначала целые 2-байтовые слова
        {
            ReadField(&PartWord, 16);                   // прочитать 16-битное поле
            *ptrWord++ = PartWord;                      // записать + сдвинуть указатель
        }
        if (RemainBits)                                 // если остались биты
        {
            ReadField(&PartWord, RemainBits);       // прочитать оставшиеся биты
            *ptrWord = PartWord;                    // записать
        }
    }
}
//---------------------------------------------------------------------------

#endif
```

```
//-------------------------------------------------------------------------------
// Записать очередной бит в битовый буфер
// Поскольку буфер очищен, то 0 записывать не надо
//-------------------------------------------------------------------------------
inline void cBitStream::PutBit(unsigned char DataBit)
{
      if (DataBit)                                    // записываем только если бит = 1
            *(Buffer + (ptrCurrentBit >> 3)) |= (1 << (ptrCurrentBit & 0x7)); // наложить бит
      ptrCurrentBit++;                                // в любом случае сместить указатель
}


#ifdef ACCESSMODE_BIT

//-------------------------------------------------------------------------------
// Записать кодовое слово CodeWord длиной CodeBitLength битов в битовый буфер
// последовательно, по одному биту
//-------------------------------------------------------------------------------
void cBitStream::WriteBitField(unsigned long int CodeWord, unsigned int CodeBitLength)
{
  unsigned char CurDataBit;

  if (CurMode == mWrite)                                // если буфер в режиме записи
       for (unsigned int i=0; i<CodeBitLength; i++)
            {
                  CurDataBit = (unsigned char)(CodeWord & 1);   // взять младший бит
                  CodeWord >>= 1;                               // сдвинуть слово
                  PutBit(CurDataBit);
                  if (ptrCurrentBit == cnBitStreamBitSize)
                        FlushBuffer();
       }
}
//-------------------------------------------------------------------------------

#else

//-------------------------------------------------------------------------------
// Записать кодовое слово CodeWord длиной CodeBitLength битов в битовый буфер
// как целое битовое поле. Максимальная длина кодового слова - 25 битов
//-------------------------------------------------------------------------------
void cBitStream::WriteField(unsigned long int CodeWord, unsigned int CodeBitLength)
{
  unsigned long int PartLen;

  if (CurMode == mWrite)                                // если буфер в режиме записи
  {
            // сначала на всякий случай обнулить "ненужные" биты кодового слова
      CodeWord &= ((1 << CodeBitLength) - 1);
            // если кодовое слово помещается целиком - то просто сдвинуть и наложить его
      if ((ptrCurrentBit + CodeBitLength) <= cnBitStreamBitSize)
      {
            *(unsigned long int *)((Buffer + (ptrCurrentBit >> 3))) |=
                              (CodeWord << (ptrCurrentBit & 0x7));
            ptrCurrentBit += CodeBitLength;
                  // выполнить проверку состояния заполненности буфера
            if (ptrCurrentBit == cnBitStreamBitSize)
                  FlushBuffer();
      }
      else
      {     // если кодовое слово не помещается целиком в буфер, то надо его делить
            // на 2 части или записывать по одному биту
            // (эта ветка выполняется редко, один раз на буфер)
            PartLen = cnBitStreamBitSize-ptrCurrentBit;         // длина помещающейся части
                  // записываем сначала младшую часть кодового слова
            *(unsigned long int *)((Buffer + (ptrCurrentBit >> 3))) |=
                              ((CodeWord & ((1<<PartLen)-1)) << (ptrCurrentBit & 0x7));
            CodeWord >>= PartLen;                   // убрать ту часть, что уже записана
            ptrCurrentBit += PartLen;               // сдвинуть указатель на конец буфера
            FlushBuffer();                          // сбросить буфер на диск
                  // записываем оставшуюся часть кодового слова
```

```
            *(unsigned long int *)(Buffer) |= CodeWord;
            ptrCurrentBit += (CodeBitLength - PartLen);          // сдвинуть указатель
                  // заполненность буфера не проверяем - он больше длины одного слова
      }
  }
}
//---------------------------------------------------------------------------


//---------------------------------------------------------------------------
// Записать длинное кодовое слово CodeWord длиной CodeBitLength битов в битовый буфер.
// Запись выполняется битовыми полями по 16 битов + остаток.
// Длина кодового слова не ограничена.
// Массив CodeLonbgWord должен быть выровнен по границе слова (16 битов).
//---------------------------------------------------------------------------
void cBitStream::WriteLongField(unsigned long int *CodeLongWord, unsigned int CodeBitLength)
{
      unsigned long int PartWord;            // очередная 16-битовая часть кодового слова
      unsigned int NumWords = CodeBitLength >> 4;   // количество полных 16-битовых частей
      unsigned int RemainBits = CodeBitLength & 0xF;       // количество оставшихся битов
      unsigned int nw = 0;
      unsigned int *ptrWord = (unsigned int *)CodeLongWord;

      if (CurMode == mWrite)                               // если буфер в режиме записи
      {
            for (nw = 0; nw<NumWords; nw++)                // сначала целые 2-байтовые слова
            {
                  PartWord = *ptrWord++;                   // читать + сдвинуть указатель
                  WriteField(PartWord, 16);                // записать 16-битное поле
            }
            if (RemainBits)                                // если остались биты
            {
                  PartWord = *ptrWord;                     // читать
                  WriteField(PartWord, RemainBits);        // записать оставшиеся биты
            }
      }
}
//---------------------------------------------------------------------------

#endif

//---------------------------------------------------------------------------
// Завершение работы с буфером - сброс остатков на диск и освобождение памяти
//---------------------------------------------------------------------------
cBitStream::~cBitStream(void)
{
      // если буфер открыт на запись и там есть данные, то сначала надо сбросить их на диск
      if ((CurMode == mWrite) && (ptrCurrentBit > 0))
            FlushBuffer();
            // закрыть поток и связь с файлом
      ofp.flush();
      ofp.close();
            // после этого освободить память
      delete[cnBitStreamByteSize+4] Buffer;
}
//---------------------------------------------------------------------------
```