Оглавление

1. Фреймворк Carbon

Фреймворк предназначен для создания переносимых программных компонентов и приложений. Основными целями при разработке фреймворка были:

- Избавиться от дублирования кода;
- Реализовать принцип: написано один раз - работает везде;

Приложение, написанное с помощью фреймворка, представляет собой набор компонентов, взаимодействующий между собой по опрелеленным правилам. Компоненты, в свою очередь, состоят из нескольких слоев. Благодаря этому достигается разделение приложения на платформо-зависимую и платформо-независимую части.

Также как и компоненты, фреймворк состоит из платформо-зависимой (от ОС и аппаратуры) и платформо-независимой частей. Это позволяет относительно просто портировать приложения на новую программно-аппаратную платформу: достаточно заменить зависимые от платформы части фреймворка и пользовательских компонентов.

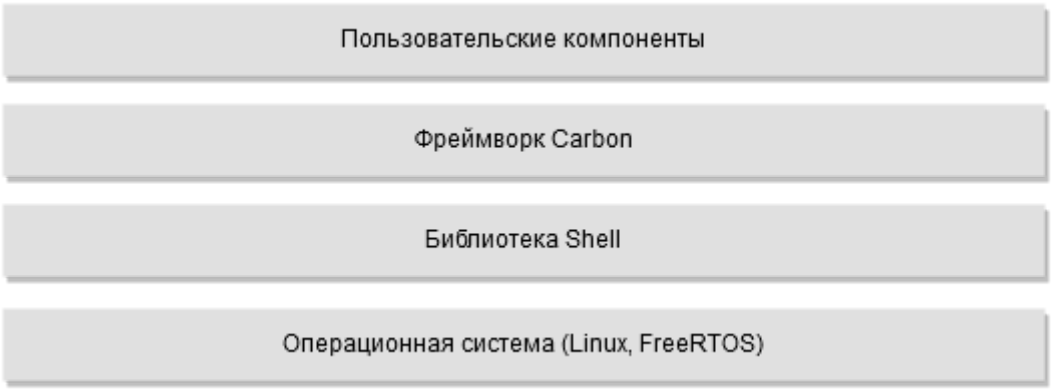
Фреймворк состоит из следующих частей:

- Набор библиотек, реализующих часто используемые функции, например, сетевой ввод/вывод;
- Набор классов, который определяет архитектуру приложения, средства и методы взаимодействия ее отдельных компонентов;

Фреймворк отличается от простой библиотеки функций тем, что библиотека может быть использована в существующей программе для упрощения реализации какого-либо функционала, а фреймворк используется в обратном порядке: он выступает как основа приложения, на которую программист накладывает специфический для него функционал. Кроме этого, фреймворк устанавливает правила взаимодействия между отдельными частями. Все это позволяет строить более крупные, многократно используемые компоненты, чем при использовании простых библиотек.

1.1 Архитектура приложения

Приложение, написанное с помощью фреймворка состоит из нескольких слоев и имеет следующую архитектуру:



Операционная система предоставляет низкоуровневые функции: создание и управление потоками, примитивы синхронизации, дисковый и сетевой ввод/вывод, и т. д.

Библиотека Shell предоставляет унифицированный, независимый от платформы API для фреймворка и пользовательских компонентов.

Фреймворк Carbon представляет собой набор классов и шаблонов, которые используются для построения пользовательских компонентов. Фреймворк построен на следующих принципах:

- *Реактивное (событийно-ориентированное) программирование* - выполнение программы состоит из обработки событий, например, нажатие клавиши, получение пакета данных из сетевого интерфейса и т.д;

- *Автоматное программирование* - это метод программирования, при котором время выполнения программы разбивается на шаги (состояния) и выполнение программы сводится к изменению состояния под действием определенных событий (более детально про автоматное программирование можно почитать здесь: Автоматное программирование);

Более подробно об архитектуре приложений рекомендуется прочитать в Microsoft Application Architecture Guide.

1.2 Жизненный цикл приложения

Жизненный цикл любого приложения состоит из следующих основных стадий:

1.2.1 Инициализация приложения

Работа приложения начинается с инициализации в несколько этапов:

- Создается глобальный и единственный объект приложения (объект класса, наследованного от `CApplication`) ;
- Инициализируются необходимые внутренние данные и компоненты фреймворка (например, журнал);
- Приложение инициализирует свои компоненты;

Инициализация приложения происходит в функции `CApplication::init()`. Приложение переопределяет эту функцию, добавляя к ней необходимый код, например:

```
result_t CUserApplication::init()
{
      result_t   nresult;

      nresult = CApplication::init();
      if ( nresult == ESUCCESS )  {

            /* User application initialisation code */
```

```
            if ( nresult != ESUCCESS )  {
                  /* UNDO */
                  CApplication::terminate();
            }
      }

      return nresult;
}
```

Функция возвращает `ESUCCESS`, если инициализация прошла успешно. В противном случае возвращается код ошибки и работа приложения прекращается.

После инициализации приложение находиться в состоянии готовности выполнения своего API.

1.2.2 Цикл обработки сообщений

Основная работа приложения происходит в цикле обработки сообщений. Приложение переопределяет функцию CApplication::processEvent(), в которой обрабатываются сообщения, например:

```
boolean_t CUserApplication::processEvent(CEvent* pEvent)
{
      boolean_t  bProcessed;

      switch ( pEvent->getType() )  {
            case EV_USER_EVENT:

                  /* User event processing code */

                  bProcessed = TRUE;
                  break;

            default:
                  /* System default event handler */
                  bProcessed = CApplication::processEvent(pEvent);
                  break;
      }

      return bProcessed;
}
```

По умолчанию создается единственный цикл и в нем обрабатываются сообщения.

В каждом цикле обработки сообщений есть очередь, в которую попадают сообщения, направленные к обработчикам, работающим в контексте этого цикла. Все сообщения выполняются последовательно, в порядке поступления в очередь. Это накладывает ограничения на использование функций в обработчиках сообщений: синхронные функции (например, `CSocket::connect()`) в обработчиках выполнять нельзя, так как это заблокирует выполнение следующих сообщений в очереди. Такие функции нужно выполнять в отдельных циклах или отдельных потоках.

Пользователь может создавать свои циклы обработки сообщений. Работа приложения завершается, когда основной цикл получает сообщение `EV_QUIT`.

1.2.3 Завершение приложения

Этап завершения приложения и освобождения ресурсов выполняет функции, обратные инициализации:

- Приложение завершает работу своих модулей;
- Фреймворк завершает работу внутренних модулей и осуществляет проверки на утечку ресурсов (например, не удаленные объекты);
- Удаляется объект приложения и возвращается код завершения;

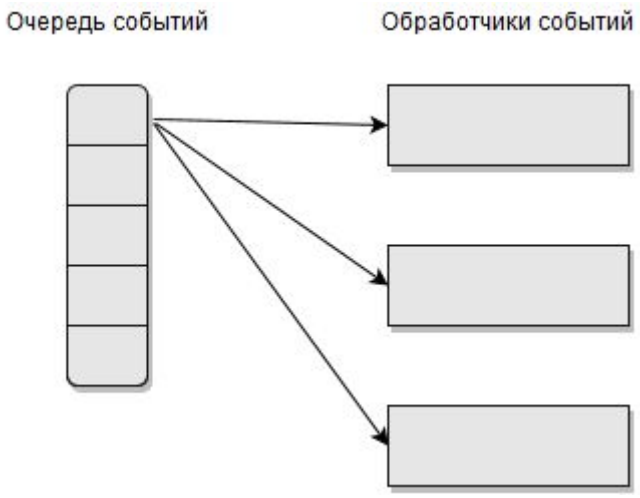
Пользовательское приложение переопределяет функцию `CApplication::terminate()` добавляя к ней необходимый код (как и в случае `init()`, нужно не забыть вызвать метод `terminate()` родительского объекта). Например:

```
void CUserApplication::terminate()
{
      /* User application termination code */

      CApplication::terminate();
}
```

1.3 Создание и обработка сообщений

Цикл обработки сообщений (событий) состоит из очереди событий и обработчиков событий, класс `CEventLoop`.



Все обрабатываемые сообщения помещаются в очередь для последующего выполнения. Если очередь сообщений пустая, то поток, в котором выполняется цикл, останавливается до получения сообщения. Все сообщения выполняются в том порядке, в котором они попадают в очередь.

Сообщения представляют собой объекты класса `CEvent` (или наследованных от него). Конструктор объекта:

```
CEvent(event_type_t type, CEventReceiver* pReceiver,
          PPARAM pParam, NPARAM nParam, const char* strDesc);
```

`type`
: тип события, системный или определенный пользователем

`pReceiver`
: указатель на объект-обработчик события

`pParam`
: первый пользовательский параметр события (указатель `void*`)

`nParam`
: второй пользовательский параметр события (целочисленное значение `natural_t`)

`strDesc`
: текстовое описание события (не обязательно, только для отладки)

Обработчиками сообщений (событий) может быть любой объект, класс которого наследован от класса `CEventReceiver`. Для того, чтобы компонент (объект) мог обрабатывать сообщения, его класс не только должен быть наследован от `CEventReceiver`, но и должен переопределить чистую виртуальную функцию `processEvent()`. Например:

```
boolean_t CReceiverModule::processEvent(CEvent* pEvent)
{
	boolean_t  bProcessed = FALSE;

	switch ( pEvent->getType() )  {
		case EV_USER1:
			/* Event EV_USER1 processing code */

			bProcessed = TRUE;
			break;

		case EV_USER2:
			/* Event EV_USER2 processing code */

			bProcessed = TRUE;
			break;
	}

	return bProcessed;
}
```

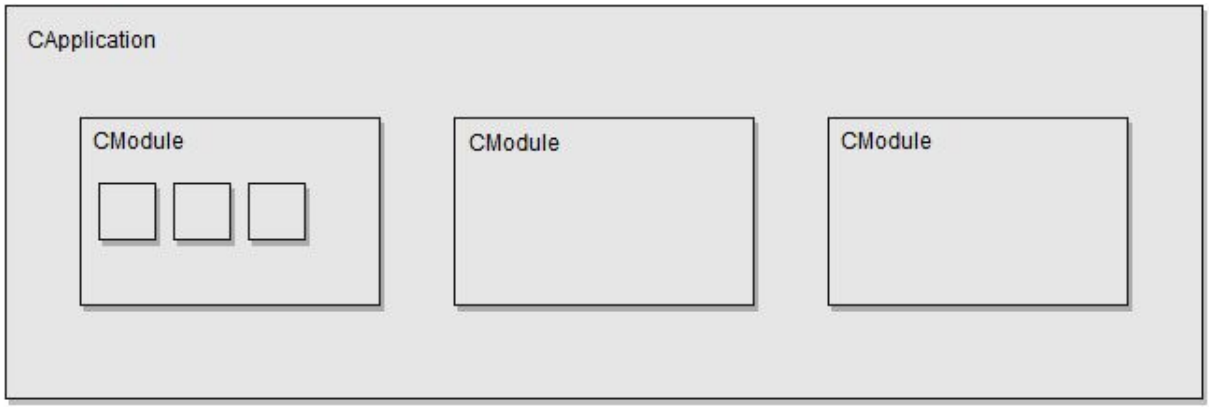
Необходимо учитывать, что каждый объект `CEventReceiver` может принимать сообщения только от одного цикла сообщений `CEventLoop`.

1.4 Компоненты

Компоненты являются основными составными частями приложения. Для того, чтобы компоненты были независимы от аппаратной части и операционной системы необходимо, чтобы:

- Код компонента разделялся на платформо-зависимый и платформо-независимый;
- Использовался только API фреймворка (библиотеки shell и carbon);

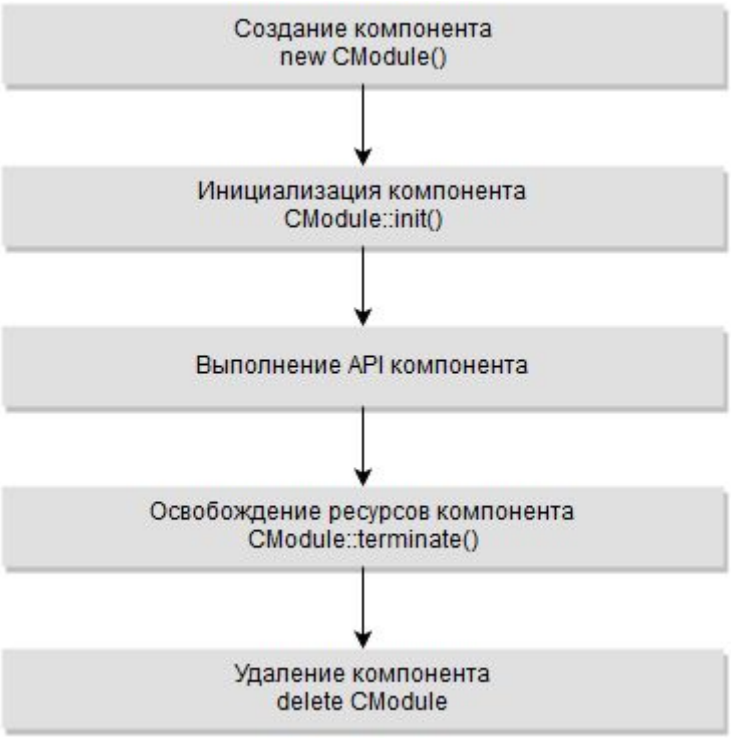
Приложение состоит из компонентов, в свою очередь компоненты могут быть вложенными. Общая структура приложения:



1.4.1 Жизненный цикл компонента

Компонент представляет собой экземпляр класса `CModule` (или наследованный от него) . Класс `CModule` определяет виртуальные методы, соответствующие следующим стадиям жизненного цикла:

- создание в конструкторе объекта;
- инициализация, `CModule::init();`
- выполнения функций API;
- Завершение работы и освобождение ресурсов, `CModule::terminate();`
- удаление с вызовом деструктора объекта;



Создание объекта происходит в конструкторе. На этом этапе этапе также происходит первоначальная инициализация данных компонента (любая инициализация, которая не может возвратить ошибку, например, установка начальных параметров).

Полная инициализация компонента происходит в функции `init()`. Если функция `init()` завершилась без ошибок, то компонент считается работоспособным и готовым выполнять свой API. Если функция возвратила ошибку, то дальнейшая инициализация компонентов прерывается и вся процедура инициализации приложения завершается с ошибкой. Например:

```
result_t CUserModule::init()
{
	result_t   nresult;

	nresult = CModule::init();
	if ( nresult != ESUCCESS )  {
		return nresult;
	}

	/* User module initialisation code */

	if ( nresult != ESUCCESS )  {
		CModule::terminate();
	}
```

```
	return nresult;
}
```

После инициализации компонент выполняет свой API. Хорошим стилем архитектуры приложения считается стиль, при котором каждый компонент выполняет только одну логическую функцию.

При завершении работы происходит вызов функции `terminate()`. В ней происходят необходимые действия до удаления объекта, например, остановка потоков, закрытие файловых дескрипторов, освобождение памяти и т.д. Например:

```
void CUserModule::terminate()
{
	/* User module termination code */

	CModule::terminate();
}
```

1.4.2 Взаимодействие компонентов

Каждый из компонентов выполняется в контексте цикла обработки сообщений или отдельного потока. Взаимодействие компонентов может осуществляться:

- прямым вызовом функция API, если компоненты выполняются в контексте одного цикла обработки сообщений или потока. Для доступа к данным нет необходимости использовать примитивы синхронизации (например, мьютексы), все события выполняются последовательно в том порядке, в котором они поступили в очередь сообщений цикла;

- посылкой сообщений (`CEvent`), если компоненты выполняются в контексте разных циклов обработки сообщений или разных потоков. В этом случае при необходимости доступа к общим данным необходимо использовать механизмы синхронизации, например, мьютексы;

2. Инсталляция и конфигурация

2.1 Структура исходного текста

Исходный текст фреймворка состоит из нескольких каталогов:

```
carbon
      app
      backend
      example
      src
            carbon
            module
            shell
      thparty
      tool
```

Подкаталоги:

- подкаталог **app** содержит дополнительные программы (сервисы, необходимые для функционирования программ, написанных с использованием фреймворка);
- подкаталог **backend** содержит исходный текст целевых операционных систем (кроме linux);
- подкаталог **example** содержит исходный текст примеров программ, написанных с использованием фреймворка и демонстрирующих работу API;
- подкаталог **src/shell** содержит исходный текст библиотеки shell;
- подкаталог **src/carbon** содержит исходный текст фреймворка carbon;
- в подкаталоге **src/module** содержаться дополнительные модули, которые пользователь может использовать в своих приложениях;
- подкаталог **thparty** содержит стороннии библиотеки, которые могут понадобиться для сборки фреймворка;
- подкаталог **tool** содержит сборочные скрипты;

2.2 Сборочные скрипты

Для компиляции фреймворка необходимо создать конфигурационный файл `config.mak`, с помощью которого настраиваются опции периода компиляции. В качестве шаблона можно использовать файл `config.in` (переименовав его в `config.mak`) изменив необходимые опции.

В файле конфигурации определены две make переменные:

- **`MODULE_DEP`** содержит список необходимых компонентов в каталоге `src/module`;
- **`THPARTY_DEP`** содержит список необходимых для работы сторонних библиотек в каталоге `thparty`;

Пример:

```
MODULE_DEP += net_media

THPARTY_DEP += mp4v2
```

В каталогах с исходным текстом может находиться необязательный файл `depend.mak`, который содержит дополнительные зависимости того модуля, в каталоге которого он находиться.

2.3 Компиляция

Компиляция фреймворка осуществляется с помощью make-файла из корневого каталога. Командная строка:

```
make [MACHINE=<machine[-backend]>] [CROSS_COMPILE=<gcc-prefix>]
[RELEASE=1]
```

- Переменная `MACHINE` определяет целевую платформу и может быть “unix”, “embed-tneo” или “embed-freertos”. По умолчанию “unix”;
- Переменная `CROSS_COMPILE` определяет префикс используемого компилятора GCC;
- Переменная `RELEASE` определяет режим компиляции release/debug. По умолчанию используется компиляция в режиме debug (RELEASE=0);

3. Примеры приложений

Примеры приложений расположены в каталоге **example**.

01minimal
: Демонстрирует минимальное приложение.

02event
: Демонстрирует создание, отправку и прием сообщений.

03timer
: Демонстрирует создание таймера и отправку сообщений по его срабатыванию.

04thread
: Демонстрирует создание дополнительного потока и посылку из него сообщений в основной цикл обработки.

05module
: Демонстрирует создание нескольких, взаимодействующих между собой модулей и разделение модуля на платформо-зависимую и платформо-независимую части.

06net_server
: Демонстрирует клиент-серверный обмен данными с использованием компонента `net_server` в асинхронном режиме.

07remote_event
: Демонстрирует отправку и прием сообщений между приложениями.

08shell_execute
: Демонстрирует выполнение внешних команд.

09net_sync
: Демонстрирует клиент-серверный обмен данными с использованием компонента `net_connector` в синхронном режиме.

10net_server_sync
: Демонстрирует клиент-серверный обмен данными с использованием компонента `net_server` в синхронном режиме.

11udp_server
: Демонстрирует отправку и прием широковещательных UDP сообщений с использованием компонента `net_connector`.

12driver_interface

Демонстрирует создание интерфейса и обмен данными пользовательской программы и драйвера для ОС UNIX (Linux).

3.1 Минимальное приложение

Исходный текст примера находится в каталоге: **example/01minimal.**

Минимальное приложение демонстрирует жизненный цикл приложения, состоящего из единственного компонента - объекта приложения:

```
class CMinimalApp : public CApplication
{
      public:
            CMinimalApp(int argc, char* argv[]);
            virtual ~CMinimalApp();

      public:
            virtual result_t init();
            virtual void terminate();
};

CMinimalApp::CMinimalApp(int argc, char* argv[]) :
      CApplication("Minimal App", MAKE_VERSION(1, 0), argc, argv)
{
}

CMinimalApp::~CMinimalApp()
{
}

result_t CMinimalApp::init()
{
      result_t   nresult;

      nresult = CApplication::init();
      if ( nresult != ESUCCESS )  {
            return nresult;
      }

      /* Custom application initialisation code */

      return ESUCCESS;
}

void CMinitmalApp::terminate()
{
      /* Custom application termination code */
```

```
	CApplication::terminate();
}

int main(int argc, char* argv[])
{
	CMinitmalApp	app(argc, argv);
	int		nExitCode;

	nExitCode = app.run();

	return nExitCode;
}
```

Как и любой другой компонент (объект класса, наследованный от `CModule`), пользовательское приложение проходит несколько этапов жизненного цикла в функции `main()`:

1. Создание объекта приложения:
   `CMinitmalApp	app(argc, argv);`
   При этом вызывается конструктор `CMinimalApp::CMininalApp()`;

2. В функции `app.run()` происходит инициализация, при этом вызывается функция `CMinimalApp::init()`, затем запускается цикл обработки сообщений. После обработки сообщения `EV_QUIT` происходит остановка цикла и вызывается функция завершения `CMininalApp::terminate()`;

3. Объект автоматически удаляется при выходе из функции `main()`, вызывая деструктор `CMininalApp::~CMinimalApp()`;

Нужно помнить, что при переопределении виртуальных функций, таких как `init()`, `terminate()`, необходимо вызывать соответствующие функции родительского класса.

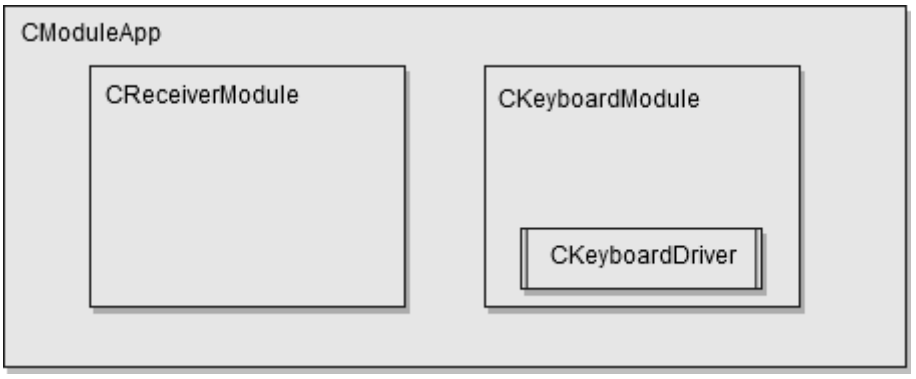
Приложение завершается при обработке сообщения `EV_QUIT`. В минимальном приложении это сообщение непосредственно не посылается. Но в версии фреймворка для ОС UNIX при старте приложения в фоне запускается сигнальный сервер (`CSignalServer`), который обрабатывает системные сигналы `SIGxxx`. По умолчанию, сигнальный сервер при приеме сигналов `SIGINT` и `SIGTERM` посылает приложению сообщение `EV_QUIT`, которое завершает приложение. Поэтому, для завершения минимального приложения нужно послать ему один из этих сигналов (например, нажав Ctrl-C).

3.2 Многокомпонентное приложение

Исходный текст примера находится в каталоге: **example/05module.**

Приложение реализует алгоритм асинхронного приема данных с устройства (для примера используется клавиатура) и обработка принятых данных (вывод кода нажатой клавиши в журнал).

Приложения состоит из двух компонентов:



1. Приложение состоит из двух модулей, каждый из которых выполняет свою задачу: `CKeyboardModule` получает данные с клавиатуры, `CReceiverModule` обрабатывает полученные данные;

2. Модули взаимодействуют друг с другом посредством передачи сообщения `EV_KEY`. Это необходимо, так как они выполняются в разных контекстах: `CKeyboardModule` выполняется в отдельном потоке, `CReceiverModule` выполняется в контексте основного цикла обработки сообщений приложения;

3. Вся аппаратно-зависимая часть приложения выделена в отдельный класс `CKeyboardDriver`. В этом классе определены только минимально необходимые функции. При переносе на другую аппаратуру достаточно реализовать этот класс;

4. Интерфейс программирования (API)

Интерфейс программирования состоит из набора классов, шаблонов и небольшого количества глобальных объектов и функций.

Обычно пользовательское приложение не обращается к глобальным объектам напрямую, а использует для этого существующие вспомогательные функции.

Все классы API могут быть разделены на классы со счетчиком ссылок и без счетчика. Классы без счетчика ссылок имеют обычные конструкторы и деструкторы. Классы со счетчиком ссылок наследуются от класса `CRefObject`, который реализует функциональность объекта со счетчиком. Объект такого класса имеет счетчик ссылок и две функции: `reference()` для инкремента счетчика и `release()` для декремента счетчика. Если модуль хочет использовать объект со счетчиком долговременно (для нескольких функций своего API), то он в начале работы с объектом увеличивает счетчик ссылок, а в конце работы уменьшает. Как только счетчик объекта уменьшается до 0, то объект автоматически удаляется. Деструктор объекта со счетчиком недоступен для вызова вне объекта.

4.1 Базовые типы данных

Заголовочный файл: **`shell/types.h`**

В качестве примитивных базовых типов данных вместе со стандартными используются следующие типы данных:

```
int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t,
int64_t, uint64_t
```

Целочисленное значение с размерностью процессора: `natural_t, unatural_t`.
Указатель: `ptr_t (void*)`. Определена константа: `NULL`.
Логический тип данных: `boolean_t`. Определены константы: `TRUE, FALSE`.

4.2 Коды ошибок

Заголовочный файл: **`shell/error.h`**

Все функции, возвращающие свой статус, используют тип данных `result_t`.

Предопределены коды: `ESUCCESS` – нет ошибки, успешно. Остальные коды ошибок совпадают с кодами ошибок GNU/Linux, их можно посмотреть здесь: https://www.gnu.org/software/libc/manual/html_node/Error-Codes.html

4.3 Журнал (логирование)

Заголовочный файл: **carbon/logger.h**

Функции журналирования могут быть разных типов: INFO, WARNING, ERROR, DEBUG и разных каналов. Каналы статически закрепляются за подсистемами/компонентами и управляются независимо друг от друга. Это позволяет динамически настраивать получение необходимых сообщений.

4.3.1 Функции журналирования

```
void log_info(int chan, const char* strMessage, …);
void log_info2(int chan1, int chan2, const char* strMessage, …);

void log_warning(int chan, const char* strMessage, …);
void log_warning2(int chan1, int chan2, const char* strMessage,…);


void log_error(int chan, const char* strMessage, …);
void log_error2(int chan1, int chan2, const char* strMessage, …);

void log_debug(int chan, const char* strMessage, …);
void log_debug2(int chan1, int chan2, const char* strMessage, …);

void log_debug_bin(int chan, const void* pData, size_t length,
                   const char* strMessage, …);
void log_debug_bin2(int chan1, int chan2, const void* pData,
                   size_t length, const char* strMessage, …);
```

`chan`, `chan1`, `chan2` - номера каналов. Функции вида `log_xxx2()` используются для вывода сообщения по двум каналам одновременно.

Определены каналы:

`L_ALWAYS_ENABLED`
: Сообщения по этому каналу всегда включены

`L_ALWAYS_DISABLED`
: Сообщения по этому каналу всегда выключены

`L_GEN`

Канал для общих сообщений, может использоваться для компонентов без выделения для них отдельных каналов. По умолчанию включен.

`L_GEN_FL`
Канал аналогичен `L_GEN`, но используется для более подробных сообщений. По умолчанию выключен.

`L_SOCKET, L_SOCKET_FL`
Используется сетевым вводом/выводом.

`L_FILE, L_FILE_FL`
Используется файловым вводом/выводом.

Список всех каналов можно посмотреть в заголовочных файлах: `shell/logger.h` и `carbon/logger.h`. Максимальное количество каналов 1024.

Функции `log_debug_bin()/log_debug_bin2()` используются для вывода вместе с текстовой строкой двоичного дампа памяти в формате HEX. Параметры `pData` и `length` определяют адрес памяти и длину в байтах.

4.3.2 Устройства вывода

Функции журналирования работают с одним или несколькими устройствами вывода. Эти устройства называются аппендерами. Если определено несколько аппендеров, то сообщение выводится на все аппендеры. Определены несколько типов аппендеров:

`stdout`
Вывод информации в стандартный поток stdout.

`file`
Запись информации в файл. Максимальный размер файла и количество записываемых файлов ограничено. Самые старые записи удаляются при переполнении.

`syslog`
Запись информации в syslog.

`tcp_server`
Запись информации по сети. Клиенты могут подключаться по сети к определенному порту и получать сообщения.

По умолчанию создается один аппендер `stdout`. Аппендеров одного и того же типа может быть неограниченное количество (например, запись в разные файлы). Возможно создание новых типов устройств вывода на основе класса `CAppender`.

4.3.3 Управления журналированием

Включение логирования:

```
void logger_enable(unsigned int type_channel);
```

type_channel

тип сообщений: LT_ALL, LT_INFO, LT_WARNING, LT_ERROR, LT_DEBUG,
канал сообщения: один из определенных каналов L_xxx.
тип/канал объединяются арифметическим “и”, например:

```
logger_enable(LT_DEBUG|L_GEN_FL);
```

Выключение логирования:

```
void logger_disable(unsigned int type_channel);
```

Проверка состояния:

```
boolean_t logger_is_enabled(unsigned int type_channel);
```

Добавление аппендера “stdout”:

```
appender_handle_t logger_addStdoutAppender();
```

Добавление аппендера “file”:

```
appender_handler_t logger_addFileAppender(const char* strFilename,
size_t nFileCountMax, size_t nFileSizeMax);
```

strFilename

Полное имя файла для записи сообщений.

nFileCountMax

Количество записываемых файлов (формат: filename.N).

nFileSizeMax

Максимальный размер файла в байтах.

Добавление аппендера “syslog”:

```
appender_hande_t logger_addSysLogAppender(const char* strIdent);
```

strIdent

Строка, добавляемая в сообщение (см. Формат сообщений).

Добавление аппендера “tcp_server”:

```
appender_handle_t logger_addTcpServerAppender(uint16_t nPort);
```

nPort

Локальный номер сетевого порта для ожидания клиентов.

4.3.4 Форматирование сообщений

Формат сообщений журнала может изменяться пользователем. Формат одинаков для всех аппендеров, но для каждого типа сообщения (LT_INFO, LT_WARNING, LT_ERROR, LT_DEBUG) может быть свой.

Установка формата сообщения:

```
void logger_set_format(unsigned int type, const char* strFormat);
```

`type`
: Тип сообщения (LT_INFO, LT_WARNING, LT_ERROR, LT_DEBUG)

`strFormat`
: Формат строки

Формат строки задается аналогично форматному выводу семейства функции `printf()`. Используются следующие спецификаторы, которые при вызове функции журналирования заменяются на текущие значения:

%F	имя файла, в котором вызвана функция log_xxx() (короткое, без каталогов)
%N	номер строки в файле, в котором вызвана функция log_xxx()
%P	имя функции, из которой вызвана функция log_xxx()
%T	время вызова функции log_xxx() (формат времени см. ниже)
%s	сообщение пользователя, заданное в функции log_xxx()
%l	тип сообщения: “INF”, “WRN”, “ERR”, “DBG”
%p	идентификатор текущего процесса
%t	идентификатор текущего потока
%n	возврат каретки
%%	символ ‘%’

Формат вывода времени аналогичен формату, принятому для функции `strftime()` (man 3 strftime). Устанавливается функцией:

```
void logger_set_time_format(unsigned int type, const char*
strFormat);
```

По умолчанию для всех типов сообщений используется формат:

- текст сообщения: “`%T [%l] %P(%N): %s`”
- время сообщения: “`%d.%m.%y %H:%M:%S`”

Общая длина сообщения не может превышать 1024 символа.

4.3.5 Создания новых устройств вывода

Пользователь может создавать свои собственные устройства вывода. Аппендер представляет собой класс, наследованный от `CAppender` и определивший чистые виртуальные функции:

- `result_t init()` - начальная инициализация аппендера;
- `void terminate()` - освобождение ресурсов перед удалением объекта;
- `result_t append(const void* pData, size_t nLength)` - запись строки на целевое устройство;

После создания объекта аппендера он может быть добавлен в список рабочих аппендеров функцией `logger_insert_appender_impl()`.

В качестве примера создания аппендера можно использовать определение самого простого аппендера stdout, src/shell/logger/appender_stdout.cpp.

Важное примечание: При реализации функций `init()/terminate()/append()` нельзя использовать функции из библиотек shell/carbon. Так как, если какая-либо функция вызывает внутри себя функцию логирования, то получится бесконечная рекурсия.

4.4 Функции отладки

Приложение может быть скомпилировано в отладочном варианте и релизном. В отладочном варианте определена глобальная константа DEBUG=1, в релизном варианте определено RELEASE=1. По умолчанию компилируется отладочный вариант, для релиза при компиляции нужно указать для утилиты make переменную RELEASE=1.

4.4.1 Проверка данных на этапе выполнения

Заголовочный файл: **`shell/shell.h`**

Проверка данных на этапе выполнения программы может быть выполнена с помощью специальных функций:

```
void shell_assert(boolean_expr);
void shell_assert_ex(boolean_expr, text, …);
void shell_verify(boolean_expr);
```

В качестве `boolean_expr` может выступать любое выражение, возвращающее логический результат. Если возвращаемый результат `TRUE`, то выполнение приложения продолжается, `FALSE` - приложение аварийно завершается с выводом выражения `boolean_expr`, именем файла и номером строки, где это произошло, а также стеком текущего потока. Вариант `shell_assert_ex()` отличается от `shell_assert()` возможностью напечатать пользовательское сообщение с необязательными параметрами.

Функции `shell_assert()/shell_assert_ex()` работают только в отладочном варианте приложения, в релизном варианте `boolean_expr` не вычисляется. Функция `shell_verify()` отличается тем, что в отладочном варианте она работает аналогично `shell_assert()`, а в релизе она только вычисляет значение выражения `boolean_expr`, но не проверяет результат и не завершает программу при результате `FALSE`.

4.4.2 Вывод диагностической информации

Заголовочный файл: **carbon/logger.h**

Диагностическая информация всегда выводится в журнал (logger).

Вывод сообщения `strMessage` в журнал (`log_dump()`), и вывод сообщения вместе с дампом памяти (`log_dump_bin()`):

```
void log_dump(const char* strMessage, ...);
void log_dump_bin(const void* pData, size_t nLength, const char*
strMessage, …);
```

Формат сообщения: “%T: %s” (см. формат вывода сообщений в журнал).

Распечатка стека (должна быть включена отладочная информация), заголовочный файл shell/debug.h:

```
void printBackTrace();
```

Вывод в журнал двоичного дампа памяти в HEX формате, заголовочный файл shell/debug.h:

```
void dumpHex(const void* pData, size_t nLength);
```

`pData`
: адрес буфера данных

`nLength`
: Размер буфера данных, байтов

Вывод в строку двоичного дампа памяти в HEX формате (заголовочный файл shell/debug.h):

```
void getDumpHex(char* strBuf, size_t nBufLength, const void*
pData, size_t nLength);
```

`strBuf`
: выходной буфер для строки

`nBufLength`
: максимальная длина буфера для строки, символов

`pData`
: адрес буфера данных

`nLength`
: Размер буфера данных, байтов

4.5. Операции со строками

Заголовочный файл: **carbon/string.h**

4.5.1 Базовые строковые функции

Базовые строковые функции совпадают со строковыми функциями стандартной библиотеки C (`man 3 string`). Имена функции имеют префикс `_t`:

```
_tstrlen, _tstrcpy, _tstrcat, _tstrcmp, _tstrchr,
_tstrrchr, _tstrstr,
_tstrspn, _tstrcspn, _tstrcasecmp, _tstrncasecmp,
_tmemset, _tmemcmp, _tmemcpy, _tmemmove, _tmemmem, _tmemchr
```

Функции заполнения области памяти нулями:

```
void _tbzero(void* pData, size_t nLength);

#define _tbzero_object(__object) _tbzero(&(__object),
                          sizeof(__object))
```

Функции копирования строки:

```
void copyString(char* strDst, const char* strSrc, size_t nDstLen);
```

`strDst`
адрес буфера, в который копируется строка;

`strSrc`
адрес копируемой строки;

`nDstLen`
размер буфера, байтов;

```
void copySubString(char* strDst, const void* memSrc, size_t
nStrLen, size_t nDstLen);
```

Функция `copyString()` отличается от функции `copySubstring()` способом задания строки для копирования, для `copyString()` это ASCIIZ строка, а для `copySubstring()` это пара адрес/длина.

Функция объединения путей в файловой системе:

```
void appendPath(char* strPath, const char* strSubPath, size_t
nMaxLen);
```

`strPath`
буфер, содержащий путь в файловой системе;

`strSubPath`
строка, содержащая дополнительный путь в файловой системе;

`nMaxLen`
максимальный размер буфера, содержащего результирующий путь (включает в себя и размер `strPath`);

Функции для удаления заданных символов в начале/конце строки:

```
void rTrim(char* strSrc, const char* chars);
```
Удаление с конца строки всех символов, которые входят в строку chars;

```
void lTrim(char* strSrc, const char* chars);
```
Удаление с начала строки все символов, которые входят в строку chars;

```
void rTrimEol(char* strSrc);
```
Удаление на конце строки символов “\r” и “\n”;

```
void rTrimWs(char* strSrc);
```
Удаление на конце строки символов “\t” и “ “;

4.5.2 Класс CString

Основной класс для работы со строками. Конструкторы:

```
CString();
CString(const char* string);
CString(const void* data, size_t length);
CString(const CString& string);
```

Функции-члены	**Описание**
CString::isEmpty()	Проверяет, пустая строка или нет.
CString::c_str()	Возвращает указатель на строку const char*.
CString::cs()	Аналог c_str().
CString::equal()	Сравнивает строки
CString::equalNoCase()	Сравнивает строки без разбора высоты
CString::clear()	Очищает строку.
CString::free()	Очищает строку и освобождает занятую область памяти.
CString::size()	Возвращает размер строки.
CString::append()	Добавить строку к строке.
CString::appendPath()	Добавить строку-подкаталог к строке, содержащей путь в файловой системе.
CString::format()	Форматировать строку.
CString::getNumber()	Возвратить целочисленное значение, соответствующее содержимому строки.
CString::ltrim()	Удалить первые символы в строке.
CString::rtrim()	Удалить последние символы в строке.
CString::dump()	Диагностический вывод параметров объекта.

4.5.3 Дополнительные функции работы со строками

Заголовочный файл: **carbon/utils.h**

Функция разделения строки на подстроки по заданному символу-разделителю:

```
size_t strSplit(const char* strData, char chSep, str_vector_t*
strAr, const char* strTrim = 0);
```

`strData`
: Исходная строка

`chSep`
: Символ-разделитель

`strAr`
: выходной массив подстрок типа
`typedef std::vector<CString> str_vector_t;`

`strTrim`
: Необязательная строка, содержащая символы, которые необходимо удалить по краям разделенных подстрок;

Функция возвращает количество подстрок в массиве `strAr`.

Функция разделение адреса URL на составные части:

```
boolean_t splitUrl(const char* strUrl, split_url_t* pData);
```

`strUrl`
: Исходная строка адреса URL;

`pData`
: Результат, структура данных:

```
typedef struct {
      CString         scheme;
      CString         host;
      int             port;
      CString         path;
      CString         query;
      CString         fragment;
      CString         username;
      CString         password;
} split_url_t;
```

Функция возвращает `FALSE`, если в строке URL синтаксическая ошибка, и `TRUE`, если разделение прошло успешно и `pData` содержит составные части исходной строки.

4.6. Атомарные переменные

Заголовочный файл: **shell/atomic.h**

Атомарные переменные применяются в тех случаях, когда необходимо гарантировать безопасность выполнения операций в многопотоковой среде.

Тип данных атомарной переменной `atomic_t`. Во внутреннем представлении переменная представлена значением типа `int32_t`.

Функции, гарантирующие потоковую безопасность при работе с атомарными переменными:

```
void atomic_set(atomic_t& var, int32_t value);
```
Установить значение переменной.

```
int32_t atomic_get(atomic_t& var);
```
Получить значение переменной.

```
int32_t atomic_add(atomic_t& var, int32_t value);
```
Прибавить значение к переменной и возвратить результат.

```
int32_t atomic_sub(atomic_t& var, int32_t value);
```
Вычесть значение из переменной и возвратить результат.

```
int32_t atomic_inc(atomic_t& var);
```
Инкрементировать значение переменной и возвратить результат.

```
int32_t atomic_dec(atomic_t& var);
```
Декрементировать значение переменной и возвратить результат.

```
boolean_t atomic_cas(atomic_t& var, int32_t oldValue, int32_t
newValue);
```
Если текущее значение переменной равно `oldValue`, то записать значение `newValue` в переменную и возвратить `TRUE`. Иначе ничего не делать и возвратить `FALSE`.

4.7 Функции динамической памяти

Заголовочный файл: **carbon/memory.h**

Функции выделения/освобождение памяти аналогичны стандартным функциям из библиотеки C (`man 3 malloc`):

```
void* memAlloc(size_t size);
```
Выделение буфера памяти указанного размера в байтах.

```
void memFree(void* p);
```
Освобождение ранее выделенного буфера памяти.

```
void* memRealloc(void* p, size_t size);
```
Изменение размера ранее выделенного буфера памяти.

```
void* memAlloca(size_t size);
```
Выделение временного буфера памяти на стеке.

4.8 Функции работы со временем

Заголовочный файл: **shell/hr_time.h**

Все функции, оперирующие задержками, используют тип данных `hr_time_t`. Этот тип данных определяет независимый от используемой ОС отсчет времени с достаточным разрешением (константа `HR_TIME_RESOLUTION`, для ОС UNIX это 1 микросекунда).

Определены следующие функции:

```
int64_t HR_TIME_TO_MICROSECONDS(hr_time_t hrTime);
```
Преобразование `hrTime` в микросекунды

```
int64_t HR_TIME_TO_MILLISECONDS(hr_time_t hrTime);
```
Преобразование `hrTime` в миллисекунды

```
int64_t HR_TIME_TO_SECONDS(hr_time_t hrTime);
```
Преобразование `hrTime` в секунды

```
hr_time_t hr_time_now();
```
Получить текущее значение времени в `hr_time_t`. Счетчик времени, монотонно увеличивается, независимо от текущего календарного времени, установленного в системе.

```
hr_time_t hr_time_get_elapsed(hr_time_t hrStart);
```
Определить интервал времени, прошедший с момента `hrStart`;

4.9 Файловые функции

Заголовочный файл: **shell/file.h**

Файловые функции определены в классах `CFileAsync` и `CFile`.

Класс `CFileAsync` предоставляет базовые функции для работы с файлами и асинхронные версии функций `read()/write()`. Класс `CFile` наследует и расширяет возможности класса `CFileAsync` синхронными функциями `read()/write()`.

4.9.1 Класс CFileAsync

Конструктор класса:
`CFileAsync::CFileAsync();`

Функции-члены	**Описание**
CFileAsync::open()	Открытие файла
CFileAsync::create()	Создание нового файла
CFileAsync::close()	Закрытие файла
CFileAsync::getHandle()	Получить файловый дескриптор
CFileAsync::isOpen()	Проверить, открыт ли файл
CFileAsync::read()	Асинхронное чтение из файла
CFileAsync::readLine()	Асинхронное чтение из файла до конца строки
CFileAsync::write()	Асинхронная запись в файл
CFileAsync::getPos()	Получение текущей позиции указателя в файле
CFileAsync::setPos()	Установка позиции указателя для чтения/записи
CFileAsync::getSize()	Получение размера файла
CFileAsync::getMode()	Получение режимов доступа к файлу
CFileAsync::setHandle()	Установка файлового дескриптора

4.9.2 Класс CFile

Класс `CFile` наследует класс `CFileAsync`.
Конструктор класса:
`CFile()::CFile();`

Функции-члены	**Описание**

CFile::read()	Чтение из файла
CFile::readLine()	Чтение из файла до конца строки
CFile::write()	Запись в файл

Статические функции-члены	**Описание**
CFile::readFile()	Чтение из файла
CFile::writeFile()	Запись в файл
CFile::copyFile()	Копирование файла
CFile::renameFile()	Переименование файла
CFile::makeDir()	Создание каталога
CFile::validatePath()	Создание каталога и всех промежуточных каталогов
CFile::fileExists()	Проверить, существует ли файл
CFile::pathExists()	Синоним CFile::fileExists()
CFile::removeFile()	Удалить файл
CFile::unlinkFile()	Синоним CFile::removeFile()
CFile::sizeFile()	Получить размер файла

4.10 Сетевые функции

Заголовочный файл: **`shell/net/socket.h`**, **`shell/net/netaddr.h`**

Базовые функции сетевого ввода-вывода определены в классах `CSocketAsync` и `CSocket`. Существует еще класс `CSocketRef`, который отличается от класса `CSocket` только наличием счетчика ссылок.

Класс `CSocketAsync` реализует асинхронные сетевые функции ввода-вывода, класс `CSocket` наследован от `CSocketAsync` и реализует синхронные сетевые функции.

Для упрощения работы с сетевыми функция реализованы два класса, заголовочный файл shell/net/netaddr.h.

- класс `CNetHost` содержит необходимые функции для работы с IP адресом;

- класс `CNetAddr` содержит необходимые функции для работы с парой: IP адрес и порт;

4.10.1 Класс CSocketAsync

Конструктор:
`CSocketAsync::CSocketAsync();`

Функции-члены	**Описание**
CSocketAsync::open()	Открытие сокета
CSocketAsync::connect()	Асинхронное соединение с удаленным хостом
CSocketAsync::close()	Закрытие сокета
CSocketAsync::send()	Асинхронная посылка данных
CSocketAsync::receive()	Асинхронное чтение данных
CSocketAsync::receiveLine()	Асинхронное чтение строки данных
CSocketAsync::getHandle()	Получить дескриптор сокета
CSocketAsync::isOpen()	Проверить, открыт сокет или нет
CSocketAsync::getAddr()	Получить локальный адрес сокета
CSocketAsync::setOption()	Установить опцию сокета
CSocketAsync::getOption()	Получить значение опции сокета
CSocketAsync::getError()	Получить последнюю ошибка сокета

4.10.2 Класс CSocket

Класс `CSocket` наследует класс `CSocketAsync` и расширяет его функциями синхронного ввода-вывода.

Для прерывания ожидания синхронных функций класс пользуется классом CFileBreaker.

Конструктор:
`CSocket::CSocket();`

Функции-члены	**Описание**
CSocket::connect()	Соединение с удаленным хостом
CSocket::send()	Посылка данных
CSocket::receive()	Чтение данных
CSocket::receiveLine()	Чтение строки данных (по признака конца строки)
CSocket::listen()	Перевод сокета в режим прослушивания соединений
CSocket::accept()	Прием нового соединения на сокете
CSocket::select()	Ожидания активности на сокете
CSocket::breakerEnable()	Включить возможность прерывания блокирующих вызовов (connect(), send(), receive())
CSocket::breakerDisable()	Выключить возможность прерывания блокирующих вызовов
CSocket::breakerBreak()	Прервать блокирующий вызов с возвратом результата ECANCELED
CSocket::breakerReset()	Сброс прерывателя блокирующих вызовов в начальное состояние

4.11 Функции создания потоков

Заголовочный файл: **carbon/thread.h**

Для создания и управления потоками используется класс `CThread`.
Конструктор:

```
CThread::CThread(hr_time_t hrStartTimeout,
                 hr_time_t hrStopTimeout);
CThread::CThread(const char* strName, hr_time_t hrStartTimeout,
                 hr_time_t hrStopTimeout);
```

`hrStartTimeout`

При старте потока функция `CThread::start()` ожидает начала выполнения процедуры потока. Ожидание завершается при вызове функции `CThread::bootCompleted()` из процедуры потока, или при превышении времени ожидания `hrStartTimeout`;

`hrStopTimeout`

При завершении потока функция `CThread::stop()` ожидает завершения процедуры указанное время. Если процедура не завершилась, то применяется функция принудительной остановки процедуры потока;

Функции-члены	**Описание**
CThread::getName()	Возвращает название потока
CThread::setName()	Установить название потока (для отладки)
CThread::getData()	Получить пользовательские данные потока
CThread::start()	Запустить поток
CThread::stop()	Остановить поток
CThread::isRunning()	Определить, работает ли поток
CThread::isStopping()	Определить, поток сейчас завершается или нет
CThread::isAlive()	Определить, работает ли поток и не останавливается

4.12 Механизмы синхронизации доступа к данным

Заголовочный файл: **`carbon/lock.h`**

4.12.1 Мьютексы

Класс для создания мьютексов `CMutex`, наследуется от класса `CLock()`.
Конструктор:

```
CMutex::CMutex();
CMutex::CMutex(MutexType type);
```

`type`
: Тип мьютекса: `mutexNormal, mutexRecursive, mutexErrorCheck`;

Функции-члены	**Описание**
CMutex::lock()	Захватить мьютекс
CMutex::unlock()	Освободить мьютекс
CMutex::trylock()	Захватить мьютекс, если он свободен

4.12.2 Условные переменные

Класс для создания условных переменных `CCondition`, наследуется от класса `CLock()`.

Конструктор:
`CCondition::CCondition();`

Функции-члены	**Описание**
CCondition::lock()	Захватить мьютекс условной переменной
CCondition::unlock()	Освободить мьютекс условной переменной
CCondition::wait()	Ожидание сигнала на условной переменной
CCondition::waitTimed()	Ожидание сигнала на условной переменной, но не позже указанного времени
CCondition::wakeup()	Сигнализировать всем потокам, ожидающим сигнала на условной переменной

4.12.3 Автоматическое освобождение примитивов синхронизации

Для удобства программирования добавлен класс-обертка `CAutoLock`, который в своем конструкторе захватывает, например, мьютекс, а в деструкторе освобождает его. Это используется для автоматического освобождения примитива синхронизации при выходе из функции:

```
CMutex     mutex;

void sync()
{
      CAutoLock  locker(mutex);

      /* mutex захвачен */

      ...

      return;

      /* mutex освобожден */
}
```

4.13 Генерация событий (сообщений)

Для взаимодействия между компонентами, которые исполняются в разных контекстах (потоках) используется метод передачи сообщений (событий). Основной класс для создания событий `CEvent`.

Конструктор:

```
CEvent(event_type_t type, CEventReceiver* pReceiver,
      PPARAM pparam, NPARAM nparam, const char* strDesc);
```

`type`
: тип события, системный или определенный пользователем

`pReceiver`
: указатель на объект-обработчик события

`pParam`
: первый пользовательский параметр события (указатель `void*`)

`nParam`
: второй пользовательский параметр события (целочисленное значение `natural_t`)

Основными параметрами события является тип и адрес обработчика. Тип события - это один из предопределенных системных идентификаторов `EV_xxx` (carbon/event.h) или определенный пользователем. Идентификаторы пользовательских событий начинаются с номера `EV_USER`, и могут быть определены как:

```
#define EV_USER_APP1        (EV_USER+0)
#define EV_USER_APP2        (EV_USER+1)
…
```

В качестве пункта назначения любого события выступает пара указателей: адрес объекта-приемника события (`pReceiver`) и адрес цикла обработки сообщений, в контексте которого этот приемник работает. Так как объект приемника сообщения однозначно определяет цикл обработки, в контексте которого он работает, то в конструкторе события достаточно указать только адрес приемника.

Событие может нести с собой два дополнительных параметра, используемые пользователем по своему усмотрению. Параметр `pparam` имеет тип указателя `void*`, параметр `nparam` имеет целочисленный тип `natural_t`.

Если с объектом нужно передать больший размер данных, то это можно сделать двумя способами. Например, необходимо передать структуру:

```
struct big_data_t {
     int  buf[100];
};
```

1) Передача структуры с помощью создания нового класса:

```
class CBigEvent : public CEvent
{
     private:
          big_data_t m_data;
          int        m_param;

     public:
          CBigEvent(const big_data_t* pData, int param,
               event_type_t type, CEventReceiver* pReceiver)
          :
               CEvent(type, pReceiver, NULL, 0, “”),
               m_data(*pData),
               m_param(param)
          {
          }

          virtual ~CBigEvent() {}

          big_data_t* getData() {
               return &m_data;
          }

          int getParam() {
               return m_param;
          }
}

CBigEvent*      pEvent;
big_data_t      data;

_tbzero_object(data);
pEvent = new CBigEvent(&data, 55, EV_XXX, pReceiver, “big”);
appSendEvent(pEvent);
```

2) Передача структуры с использованием шаблона:

```
typedef CEventT<big_data_t, EV_XXX>  CBigEvent;

CBigEvent*      pEvent;
big_data_t      data;

_tbzero_object(data);
pEvent = new CBigEvent(&data, pReceiver, "desc");
appSendEvent(pEvent);
```

Первый способ можно использовать, если нужно передать несколько дополнительных параметров, второй способ, если параметр единственный.

Отправляется событие с помощью функции:

```
void appSendEvent(CEvent* pEvent);
```

Возможно отправить событие всем обработчикам событий, работающим в контексте одного цикла обработки событий:

```
void appSendMulticastEvent(CEvent* pEvent, CEventLoop* pLoop);
```

При создании события в качестве обработчика указывается `EVENT_MULTICAST`.

Пример работы с событиями: **`example/02event`**.

4.14 Таймеры

Заголовочный файл: **`carbon/timer.h`**

Для выполнения периодических или отложенных по времени операций используются таймеры. Таймер и его callback функция всегда работает в контексте какого-либо цикла обработки сообщений.

Основной класс для создания таймеров `CTimer`.
Конструктор:

```
CTimer::CTimer(hr_time_t hrPeriod, timer_cb_t callback,
          int options, void* pParam, const char* strName);

CTimer::CTimer(hr_time_t hrPeriod, timer_cb_t callback,
          int options, const char* strName);

CTimer::CTimer(hr_time_t hrPeriod, timer_cb_t callback,
```

```
            void* pParam, const char* strName);

CTimer::CTimer(hr_time_t hrPeriod, timer_cb_t callback,
            const char* strName);
```

`hrPeriod`
Период срабатывания таймера;

`callback`
Функция, вызываемая при срабатывании таймера;

`options`
Опции таймера. Определены опции:
`CTimer::timerPeriodic` периодический таймер (по умолчанию таймер срабатывает 1 раз и удаляется);

`pParam`
Пользовательские данные, передаваемые в функцию таймера при срабатывании;

`strName`
Имя/описание таймера. Используется для отладки;

Функции-члены	**Описание**
CTimer::getTime()	Получить время следующего срабатывания таймера
CTimer::isPeriodic()	Определить, таймер периодический или однократный
CTimer::pause()	Временно остановить таймер
CTimer::restart()	Запустить таймер с начала периода срабатывания

Для задания функции callback в конструкторе таймера применяется макрос:

```
TIMER_CALLBACK(__class_member, __pEventLoop);
```

`__class_member`
Функция-член объекта в виде: `имя_класса::имя_функции`;

`__pEventLoop`
Указатель на цикл обработки сообщений, в контексте которого будет выполняться функция таймера.

Функция callback таймера имеет следующий вид:

```
void callback(void* pParam);
```

Старт таймера осуществляется функцией:

```
CEventLoop::insertTimer(CTimer* pTimer);
```

Для завершения/удаления таймера используется макрос:

```
SAFE_DELETE_TIMER(__pTimer, __pEventLoop);
```

`__pTimer`

Удаляемый таймера;

`__pEventLoop`

Цикл обработки сообщений, в контексте которого выполняется таймер (который использовался при старте таймера);

Пример работы с таймером:

```
class CTimerModule : public CModule
{
	private:
		CTimer*	m_pTimer;

	public:
		CTimerModule() :
			CModule("timer example"),
			m_pTimer(0)
		{
		}

		virtual ~CTimerModule()
		{
			shell_assert(m_pTimer == 0);
		}

	public:
		virtual result_t init();
		virtual void terminate();

	private:
		void timerCallback(void* p);
		void cancelTimer();
};

result_t CTimerModule::init()
```

```
{
	result_t   nresult;

	nresult = CModule::init();
	if ( nresult != ESUCCESS )  {
		return nresult;
	}

	m_pTimer = new CTimer(HR_1SEC,
		TIMER_CALLBACK(CTimerCTimeModule::timerCallback, this),
		0, NULL, "example");

	appMainLoop()->insertTimer(m_pTimer);

	return ESUCCESS;
}

void CTimerModule::terminate()
{
	cancelTimer();
	CModule::terminate();
}

void CTimerModule::cancelTimer()
{
	SAFE_DELETE_TIMER(m_pTimer, appMainLoop());
}

void CTimerModule::timerCallback(void* p)
{
	m_pTimer = 0;
	log_debug(L_GEN, "timer callback\n");
}
```

Полный пример работы с таймерами: **example/03timer**.