ПОЧТОВЫЙ КЛИЕНТ

Задание

Разработать почтовый клиент, позволяющий пользователю отправлять электронную почту по протоколу SMPT+ESMTP, используя существующие в сети Internet почтовые сервера. Программа должна поддерживать возможность аутентификации пользователя на сервере при отправке электронной почты. Программа должна поддерживать возможность отправки одного и того же письма нескольким пользователям и приложение к письму как минимум одного файла.

Адреса получателей, тема и текст письма, месторасположение прикладываемого к письму файла, адрес почтового сервера, имя пользователя, пароль и номер порта указываются пользователем.

При необходимости осуществлять отправку с шифрованием по протоколу SSL/TLS.

При сдаче работы необходимо продемонстрировать отправку письма на почтовый ящик преподавателя и на тестовый сервер Denwer. Отправлять можно с любого адреса.

Показать исходный код программы.

Отчет оформлять необязательно.

Баллы:

6 баллов – рабочий клиент с возможностью отправки письма по протоколу SMTP: без авторизации, шифрования и без вложений;

8 баллов – рабочий клиент с возможностью отправки письма без вложений с опциональным шифрованием SSL/TLS;

10 баллов – рабочий клиент с возможностью отправки письма с одним вложением;

12 баллов – рабочий клиент с возможностью отправки письма с несколькими вложениями;

+ 1 балл за аккуратную разметку исходного кода;

+ 1 балл за комментарии к коду;

+ 1 балл за каждую функцию сервера, реализованную не как в методичке (по другому принципу)

+ 3 балла за каждую дополнительную функцию сервера, разработанную полностью самостоятельно.

Для лучшего понимания работы программы рекомендуется установить любой бесплатный почтовый клиент и изучить его.

Пояснения

В данной работе разрабатывается почтовый клиент, работающий по протоколам SMTP и ESMTP, т.е. реализующий только отправку писем с возможностью авторизации и вложений файлов в письма.

Процесс отправки письма включает три стадии:

1) соединение с сервером, авторизация;
2) отправка письма или нескольких писем;
3) завершение соединения.

Важное отличие SMPT от HTTP – это наличие диалога между клиентом и сервером. Клиент должен «помнить» предыдущий диалог, т.е. реакция клиента на входящее сообщение зависит не только от содержания этого сообщения, но и от предыдущих.

Если в прежних лабораторных мы просто посылали ответ на входящий запрос, то теперь этого недостаточно.

Есть два основных подхода для программной реализации такого диалога:

а) в специальной переменной хранить состояние клиента (какое было отправлено сообщение до этого), при каждом входящем сообщении от сервера проверять текущее состояние;

б) весь диалог вести в одной процедуре, используя функцию ожидания ответа от сервера.

Мы воспользуемся первым подходом, т.к. у второго есть недостатки.

1. Во время диалога программа будет «висеть», не реагируя на действия клиента, т.к. она не может обрабатывать две процедуры сразу. Для преодоления этого недостатка необходимо выделить отправку письма в отдельный поток (thread), что выходит за рамки данного курса.

2. За одну сессию можно отправить только одно письмо.

Более подробно этапы диалога по отправке письма представлены в таблице ниже:

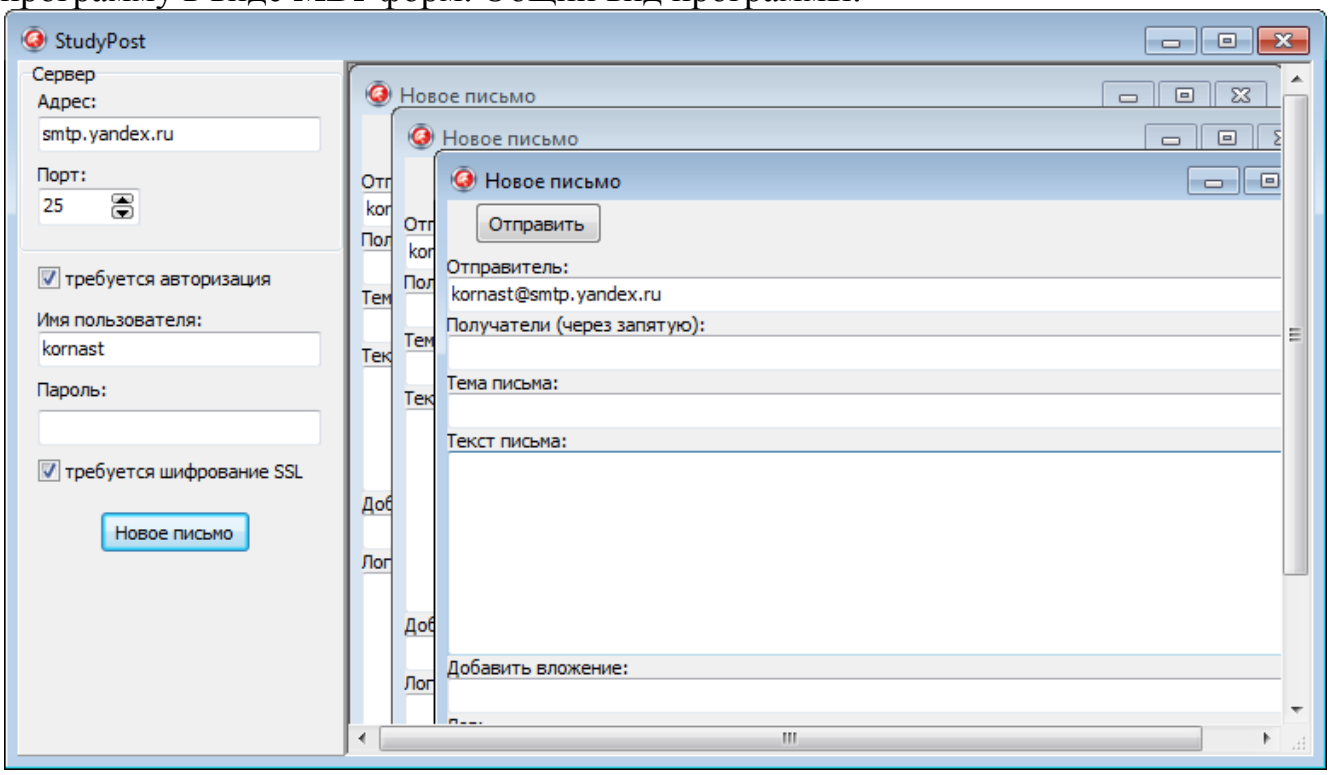
№	*Этап*	*Команда клиента*	*Код отклика сервера*
1.	Соединение	-	220
2.	Приветствие	HELO или EHLO	250 + доступные расширения
3.	Авторизация (опционально): – запрос авторизации – передача имени пользователя – передача пароля	 AUTH LOGIN имя пользователя пароль	 334 334 235
4.	Адрес отправителя	MAIL FROM: <адрес>	250
5.	Адреса получателей (1 или более)	RCPT TO: <адрес>	250
6.	Тело письма	DATA .	354 250
7.	Завершение соединения	QUIT	221

Ход работы

Интерфейс программы

Интерфейс программы удобно разбить на две формы: первая для ввода общих сведений и настоек (сервер, логин, пароль), а вторая собственно для составления и отправки письма.

Для одновременного написания и отправки нескольких писем реализуем программу в виде MDI-форм. Общий вид программы:



Сначала подготовим первую форму `FStart`. Разместим на ней панель `TPanel` со свойством `Align: alLeft`. Цвет формы `Color` измените на `clAppWorkSpace` – цвет рабочего пространства, по умолчанию – темно-серый.

Если после этого цвет панели также изменился, выставьте его обратно в `clBtnFace` (цвет кнопки), а свойство `ParentBackground` установите в False.

На панели разместите:

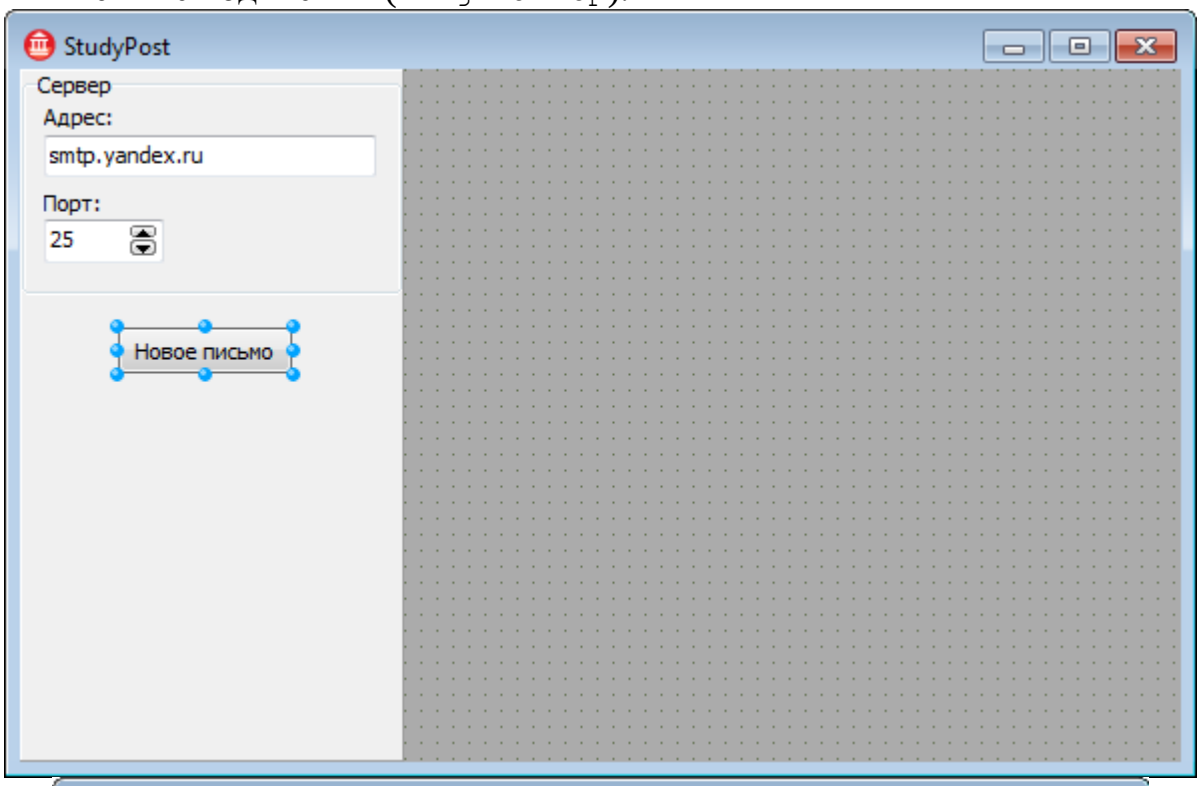
- поля для ввода адреса и порта сервера `EdServerAdress: TEdit` и `EdServerPort: TSpinEdit`;
- кнопку для создания нового письма `BtnNewMail: TButton`;
- метки `TLabel` с подписями ко всем полям.

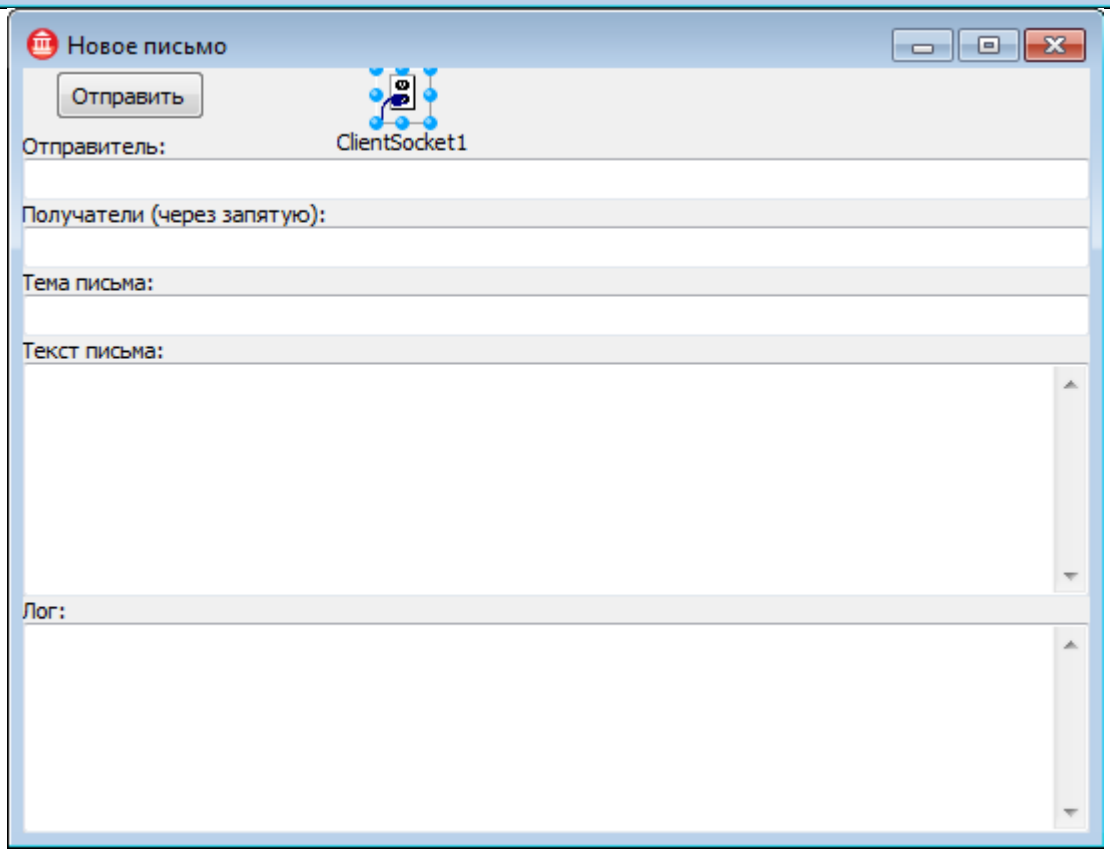
Можно вписать в поля для ввода значения по умолчанию. Порт SMTP по умолчанию – 25.

Вторая форма `FMail` предназначена для ввода и редактирования письма. Сначала реализуем клиент для отправки писем без шифрования и без вложений, соответствующие компоненты добавим на форму позже.

- панель `Panel1: TPanel` (`Align=alTop`) для размещения на ней кнопки «Отправить» `BtnSend: TButton`;

- поля `TEdit` для ввода адреса отправителя `EdFrom`, получателей `EdTo`, темы письма `EdSubject` (`Align=alTop`, все поля `Margins=3`, `AlignWithMargins=True`);
- поле `TMemo` для ввода текста письма `MmBody` (`Align=alClient`);
- поле `TMemo` для лога отправки `mmLog` (`Align=alBottom`);
- клиентский сокет `ClientSocket1: TClientSocket`;
- метки с подписями (`Align=alTop`).





Теперь сделаем наши формы MDI-формами, т.е. форма `FMail` будет располагаться внутри `FStart`.

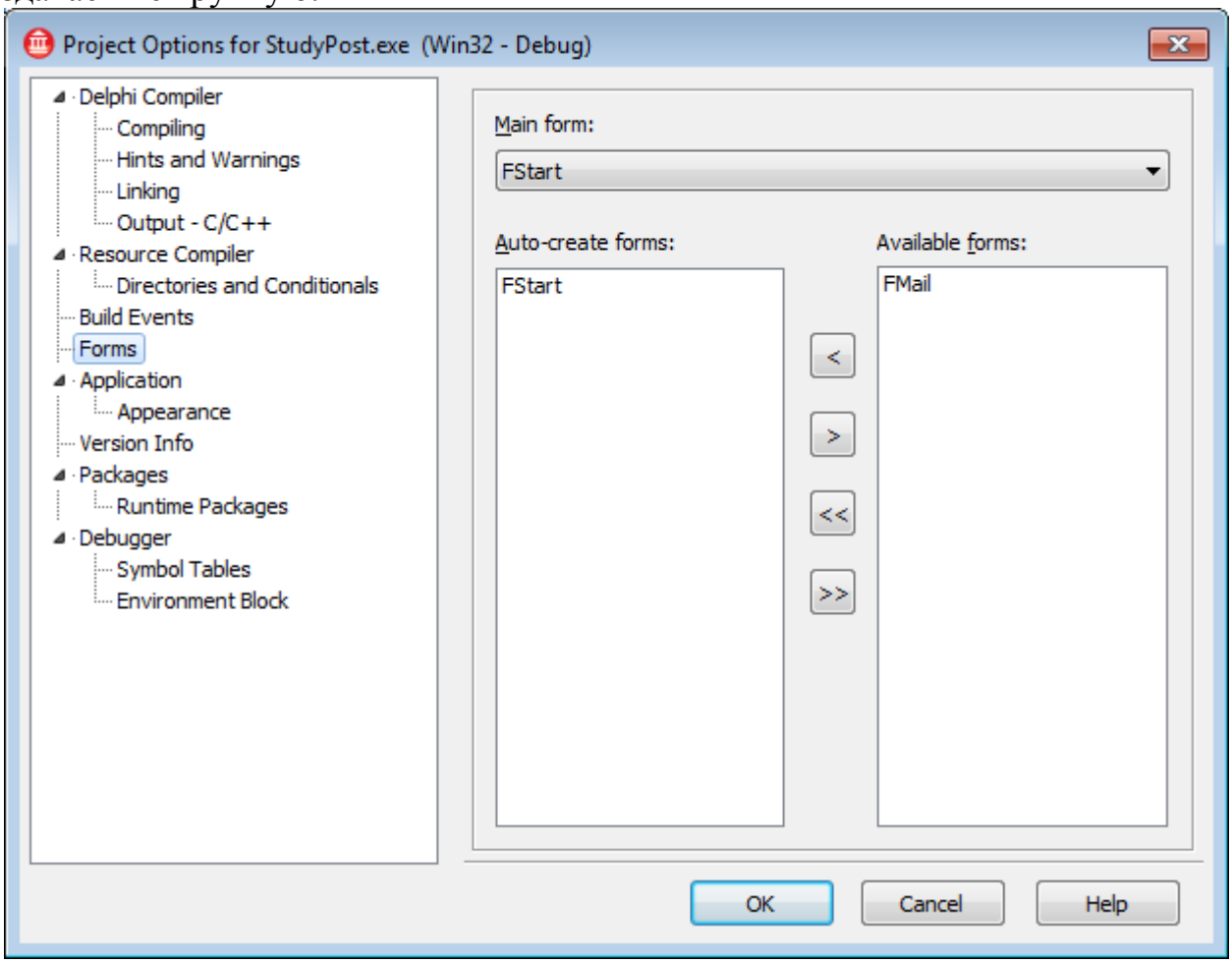
Для этого установите свойства:

`FStart.FormStyle = fsMDIForm` – главная форма;

`FMail.FormStyle = fsMDIChild` – дочерняя форма.

Дочерние формы будем создавать по запросу пользователя, при нажатии на кнопку «Новое письмо», т.е. в момент запуска программы `FMail` еще не должно быть.

Откройте свойства проекта (меню Project – Options...). В открывшемся окне на вкладке Forms перенесите `FMail` из автоматически создаваемых форм в создаваемые вручную.



Сразу же впишем код для создания дочерней формы по нажатию на кнопку «Новое письмо», вписав соответствующий модуль в `uses` в разделе `implementation`:

```
implementation

{$R *.dfm}

uses
  UMail;    //модуль с описанием FMail

procedure TFStart.BtnNewMailClick(Sender: TObject);
begin
  TFMail.Create(Application);  //создание дочерней формы
end;
```

Примечание

Обратите внимание, допустимо вписать модуль UMail и в раздел interface к другим файлам. Но обычно в interface добавляются только те модули, которые непосредственно используются при описании формы.

Кроме того, нельзя чтобы два файла одновременно ссылались друг на друга через interface. Т.е., если в файле Unit1 вписать в uses в разделе interface модуль Unit2, и одновременно в файле Unit2 вписать в uses в разделе interface модуль Unit1, то это приведет к ошибке.

Фактически, это единственное действие, которое выполняет главная форма.

Однако, для удобства пользователя, следует сразу вписать в поле «Отправитель» сведения из имени пользователя и адреса сервера (user_name@server).

Для этого нам нужно с главной формы обратиться к дочерней форме. В обычном случае для этого можно было бы использовать переменную FMail (не путать с типом TFMail):

```
FMail.EdFrom.Text := EdUsername.Text + '@' + EdServerAdress.Text;
```

Но для MDI-форм это не подходит. Этих форм много, и они не заносятся в переменную FMail (ее можно вообще удалить).

Для доступа к вновь созданной форме, используем локальную переменную:

```
procedure TFStart.BtnNewMailClick(Sender: TObject);
var NewFMail: TFMail;
begin
  NewFMail := TFMail.Create(Application);
  NewFMail.EdFrom.Text := EdUserName.Text + '@' + EdServerAdress.Text;
end;
```

Также добавим кое-какое оформление для дочерней формы.

Во-первых, по умолчанию, при нажатии кнопки «Закрыть» (крестик), дочерние формы не закрываются, а сворачиваются. Исправить это можно через событие формы onClose. Параметр Action определяет, какое действие необходимо выполнить:

caNone – ничего не делать (запрет на закрытие формы);

caHide – форма не будет закрыта, а лишь спрятана с экрана, при необходимости ее можно будет открыть вновь;

caFree – закрыть и полностью удалить форму из памяти

caMinimize – свернуть форму (по умолчанию для дочерних MDI-форм).

Таким образом:

```
procedure TFMail.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Во-вторых, заголовок формы «Новое письмо» не всегда удачен. При открытии нескольких писем трудно понять, какое из них какое. Желательно дописать в заголовок тему письма, указанную пользователем. Будем менять заголовок при редактировании EdSubject, событие onChange:

```
procedure TFMail.EdSubjectChange(Sender: TObject);
begin
  if (EdSubject.Text = '') then
```

```
    Caption := 'Новое письмо'
  else
    Caption := EdSubject.Text;
end;
```

`Caption` в данном случае относится к текущей форме. Опять же, если бы форма была одна и создавалась автоматически, можно было бы написать `FMail.Caption`, но у нас переменная `FMail` не используется.

Еще один вариант записи, который работает всегда: `self.Caption`, где `self` – это текущий объект, обрабатывающий процедуру, т.е. текущая форма.

```
procedure TFMail.EdSubjectChange(Sender: TObject);
begin
  if (EdSubject.Text = '') then
    self.Caption := 'Новое письмо'
  else
   self.Caption := EdSubject.Text;
end;
```

Ведение лога

В лог необходимо вносить любые сообщения, проходящие через сокет, указав входящие они (от сервера, метка S) или исходящие (от клиента, метка C), а также подключение и отключение от сервера и возникшие ошибки. Все эти действия, кроме исходящих сообщений, соответствуют событиям клиентского сокета `onRead`, `onConnect`, `onDisconnect`, `onError`.

Событие `onWrite` хоть и срабатывает при готовности сокета к отправке сообщения, но не содержит информации о передаваемых данных, поэтому его мы использовать не будем.

В `onConnect` добавим в лог текст «Подключение к серверу» с указанием времени и разделительной линией.

```
procedure TFMail.ClientSocket1Connecting(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  mmLog.Lines.Add(TimeToStr(Now())
    + ': Подключение к серверу ------------------------');
end;
```

В `onDisconnect` добавьте аналогичное сообщение с текстом «Отключение от сервера».

Событие `onError` содержит информацию о возникшей ошибке в виде параметра `ErrorEvent` (во время какого действия возникла ошибка) и `ErrorCode` (десятичный код ошибки, расшифровку можно посмотреть в документации на MSDN). Выведем в лог код ошибки:

```
procedure TFMail.ClientSocket1Error(Sender: TObject; Socket: TCustomWinSocket;
  ErrorEvent: TErrorEvent; var ErrorCode: Integer);
begin
  mmLog.Lines.Add(TimeToStr(Now())
    + ': Ошибка №' + IntToStr(ErrorCode));
  ErrorCode := 0;
end;
```

В событие `onRead` добавим переменную типа `AnsiString` (ACSII-кодировка) для получения текста и полностью выведем его в лог:

```
procedure TFMail.ClientSocket1Read(Sender: TObject; Socket: TCustomWinSocket);
var
  msg: AnsiString;
begin
  msg := Socket.ReceiveText;
  mmLog.Lines.Add(TimeToStr(Now())
    + ' S: ' + msg);
end;
```

Для добавления в лог отправленных сообщений введем процедуру `SendText` (объявите ее в разделе `public` формы). В конце каждого сообщения, передаваемого в сокет, необходимо добавить символы конца строки `#13#10`.

```
procedure TFMail.SendText(Msg: AnsiString);
begin
  mmLog.Lines.Add(TimeToStr(Now())
    + ' C: ' + Msg);
  ClientSocket1.Socket.SendText(Msg + #13#10);
end;
```

В дальнейшем будем использовать эту процедуру для отправки сообщений.

Отправка письма по протоколу SMTP

Классический SMTP поддерживает только отправку текстовых писем в открытом виде (без шифрования) и без авторизации пользователя. На сегодняшний день вам вряд ли удастся найти в Интернете почтовый сервер с такими возможностями – на всех общеизвестных серверах (mail, yandex, google, rambler, yahoo) требуют авторизации и шифрования по протоколам SSL/TLS.

Поэтому на начальных этапах разработки программы мы будем проверять ее работу на тестовой «заглушке» на базе почтового MTA-сервера **sendmail** из пакета **Denwer**. Программа прилагается к лабораторной работе, либо вы можете скачать Denwer с официального сайта denwer.ru. После установки Denwer, Нужную программу можно найти в его директории в `denwer\tools\sendmail`.

Для запуска сервера используйте программу `sendmail_daemon_start.exe`. В результате в системном трее появится значок  . Двойной клик по нему открывает консоль с логом работы сервера.

Для остановки сервера: правый клик по значку, команда `Terminate`, либо запустите `sendmail_daemon_stop.exe` из той же директории.

Письма с этого сервера никуда не отправляются, а сохраняются в папку `tmp\!sendmail` в виде файлов `.eml`.

Состояния клиента

Как было сказано выше, нам необходимо запомнить, какое сообщение отправлял клиент до этого (или не отправлял ничего). Можно хранить его в виде обычного текста, но анализировать такое сообщение неудобно.

Как правило, для хранения состояний в программировании используются перечислимые типы данных – пользовательский тип, для которого явно перечислены все возможные значения – состояния (см. Приложение 1).

Наш клиент может находиться в одном из следующих состояний:

Обозначение	Пояснение
NONE	Ничего не было отправлено
HELO	Отправлено приветствие
MAIL_FROM	Отправлена команда MAIL FROM (от кого)
RCPT_TO	Отправлена команда RCPT TO (кому)
DATA	Отправлен запрос на отправку тела письма
MAIL	Отправлено содержимое письма (заголовки и тело)
QUIT	Завершение соединения

Запишем эти состояния в код программы.

В секции `type` до описания формы добавьте собственный тип с перечнем этих значений в круглых скобках. Однако обозначения NONE, QUIT, DATA являются достаточно распространенными, и могут перекрыть уже использующиеся где-то в другом месте. Поэтому (а еще, чтобы удобнее было выводить подсказку по Ctrl+Пробел), к каждому состоянию в начале припишем префикс `smtp`.

```
type
  //состояния SMTP-сессии
  TSmtpStates = (smtpNONE, smtpHELO, smtpMAIL_FROM, smtpRCPT_TO, smtpDATA,
smtpMAIL, smtpQUIT);
```

`TSmtpStates` – это *тип данных*, т.е. можно объявлять переменные типа `TSmtpStates`. Объявим ее внутри формы (*поле* – переменная, объявленная внутри класса).

```
public
  //текущее состояние SMTP-сессии
  CurrentState: TSmtpStates;
```

Полю `CurrentState` можно присвоить только значения, перечисленные в `TSmtpStates`. Например,

```
CurrentState := smtpHELO;
```

Это состояние мы присвоим после отправки сообщения HELO.

Подключение

Подключение включает собственно подключение клиентского сокета и отправку сообщения `HELO`.

Подключение сокета выполняем после клика по кнопке «Отправить» `BtnSend`. Необходимо вписать адрес и порт из `Edit`-ов в сокет, открыть соединение и записать состояние клиента `NONE`, т.е. еще не отправлено никаких сообщений.

```
procedure TFMail.BtnSendClick(Sender: TObject);
  ClientSocket1.Host := FStart.EdServerAdress.Text;
  ClientSocket1.Port := FStart.EdServerPort.Value;
  ClientSocket1.Open;
  CurrentState := smtpNONE;
end;
```

Для доступа к главной форме FStart ее модуль должен быть указан в uses в разделе implementation.

```
uses
  UStart;
```

Отправка письма – длительный процесс. Нежелательно, чтобы пользователь мог редактировать письмо, пока идет отправка. Поэтому после подключения к серверу нужно отключить все поля для ввода и кнопку «Отправить». После завершения передачи они вновь должны стать активными.

Самостоятельно добавьте соответствующий код в обработчики OnConnect и OnDisconnect клиентского сокета.

Только после того, как соединение было установлено и получено приветствие от сервера, можно отправить сообщение HELO. Т.е. действие выполняем в событии onRead сокета, если состояние было NONE. Добавьте в обработчик onRead следующий код:

```
  case CurrentState of //проверить текущее состояние
  smtpNONE:     DiaOnStart(Msg); //ничего не было отправлено
  smtpHELO:     DiaOnHELO(Msg); //ничего не было отправлено
  smtpMAIL_FROM:DiaOnMailFrom(Msg);
  smtpRCPT_TO:  DiaOnRcptTo(Msg);
  smtpDATA:     DiaOnData(Msg);
  smtpMAIL:     DiaOnMail(Msg);
  smtpQUIT:     DiaOnQuit(Msg);
  end;
```

DiaOnStart, DiaOnHELO и т.д. – это процедуры, которые необходимо объявить в разделе private формы:

```
  private
    { Private declarations }
    CurrentState: TSMTPState;
    procedure SendText(Msg: AnsiString);
    procedure DiaOnStart(Msg: AnsiString);
    procedure DiaOnHELO(Msg: AnsiString);
    procedure DiaOnMailFrom(Msg: AnsiString);
    procedure DiaOnRcptTo(Msg: AnsiString);
    procedure DiaOnData(Msg: AnsiString);
    procedure DiaOnMail(Msg: AnsiString);
    procedure DiaOnQuit(Msg: AnsiString);
```

Реализацию всех этих функций мы напишем позже. Нажмите Ctrl+Shift+C в описании формы, чтобы Delphi автоматически сформировал объявление этих процедур в implementation и оставьте их пока пустыми.

Отправка приветствия

Напишем процедуру DiaOnStart. Она возникает как реакция на стартовое сообщение сервера после подключения.

Если сервер прислал стартовое сообщение с кодом 220 – то все в порядке, отправляем команду HELO. В противном случае – ошибка, необходимо отключиться от сервера. Но просто оборвать соединение будет «нетактично», нужно попытаться завершить его командой QUIT.

```
procedure TFMail.DiaOnStart(Msg: AnsiString);
begin
```

```
  if copy(Msg, 1, 3) = '220' then //успешное соединение
  begin
    CurrentState := smtpHELO;
    SendText('HELO StudyPost');
  end
  else //неудача
  begin
    mmLog.Lines.Add('Ошибка при соединении! Сервер отказал в подключении. '
      + 'Возможно, он не поддерживает SMTP.');
    //отправка команды QUIT
    CurrentState := smtpQUIT;
    SendText('QUIT');
  end;
end;
```

По такому же принципу будут строиться и все остальные процедуры ответа клиента. Поскольку команда QUIT может быть использована уже на этом этапе, реализуем ее следующей.

Завершение соединения

Если сервер прислал код 221 – все в порядке, в противном случае возникла ошибка. Сообщение об ошибке нужно добавить в лог, но соединение в любом случае оборвется.

```
procedure TFMail.DiaOnQuit(Msg: AnsiString);
begin
  if copy(Msg, 1, 3) <> '221' then
    mmLog.Lines.Add('Ошибка при отключении! Неверный код состояния.');
  ClientSocket1.Close;
end;
```

Отправка адреса отправителя

В нормальном режиме работы эта команда отправляется после `HELO`, соответственно, ее необходимо вписать в `DiaOnHELO`. Признаком успешного ответа сервера на HELO является код 250.

```
procedure TFMail.DiaOnHELO(Msg: AnsiString);
begin
  if copy(Msg, 1, 3) = '250' then
  begin
    CurrentState := smtpMAIL_FROM;
    SendText('MAIL FROM: <' + EdFrom.Text + '>');
  end
  else
  begin
    mmLog.Lines.Add('Ошибка при подключении! Сервер не ответил на приветствие. '
      + 'Возможно, он не поддерживает SMTP.');
    CurrentState := smtpQUIT;
    SendText('QUIT');
  end;
end;
```

После этого необходимо отправить перечень получателей (пока только одного).

Отправка адреса получателя

Обработчик пишем в ответе на MAIL FROM, код ответа сервера должен быть 250.

```
procedure TFMail.DiaOnMailFrom(Msg: AnsiString);
begin
  if copy(Msg, 1, 3) = '250' then
  begin
    CurrentState := smtpRCPT_TO;
    SendText('RCPT TO: <' + EdTo.Text + '>');
  end
  else
  begin
    mmLog.Lines.Add('Ошибка при отправке письма! '
      + 'Неверное поле "Отправитель": ' + EdFrom.Text);
    CurrentState := smtpQUIT;
    SendText('QUIT');
  end;
end;
```

Отправка запроса на передачу тела письма

Команда DATA отправляется после RCPT TO, если код ответа сервера – 250.

```
procedure TFMail.DiaOnRcptTo(Msg: AnsiString);
begin
  if copy(Msg, 1, 3) = '250' then
  begin
    CurrentState := smtpDATA;
    SendText('DATA');
  end
  else
  begin
    mmLog.Lines.Add('Ошибка при отправке письма! '
      + 'Неверное поле "Получатель": ' + EdFrom.Text);
    CurrentState := smtpQUIT;
    SendText('QUIT');
  end;
end;
```

Передача тела письма

В ответ на команду DATA сервер должен вернуть код 254 – готов к передаче.

Тело письма содержит поля отправителя, получателя, темы. Затем построчно передается текст письма, и в конце – строка с единственным символом точки.

```
procedure TFMail.DiaOnData(Msg: AnsiString);
var i: Integer;
begin
  if copy(Msg, 1, 3) = '354' then
  begin
    CurrentState := smtpMAIL;
    //заголовки - от кого, кому, тема
    SendText('From: ' + EdFrom.Text);
    SendText('To: ' + EdTo.Text);
    SendText('Subject: ' + EdSubject.Text);
    //текст письма построчно
    for i := 0 to MmBody.Lines.Count - 1 do
      SendText(MmBody.Lines[i]);
    //конец письма
    SendText('.');
  end
  else
  begin
```

```
    mmLog.Lines.Add('Ошибка при передаче письма! '
      + 'Сервер отклонил передачу.');
    CurrentState := smtpQUIT;
    SendText('QUIT');
  end;
end;
```

При передаче письма может возникнуть ошибка: если в какой-то из строк содержится только символ точки, то это будет считаться концом письма. Самое простое решение – дописать к такой строке пробел. Самостоятельно добавьте такую проверку.

Завершение передачи

После передачи письма сервер должен вернуть код 250 с указанием размера письма в байтах. В любом случае, после этого завершаем соединение командой QUIT.

```
procedure TFMail.DiaOnMail(Msg: AnsiString);
begin
  if copy(Msg, 1, 3) <> '250' then
    mmLog.Lines.Add('Ошибка при отправке письма! '
      + 'Не удалось отправить письмо.');
  CurrentState := smtpQUIT;
  SendText('QUIT');
end;
```

На данном этапе клиент готов к работе. Протестируйте его с помощью сервера Denwer. Письма необходимо отправлять на `localhost:25`. Отследите лог работы сервера и проверьте, чтобы письмо появилось в виде текстового файла в `tmp\!sendmail`.

Попробуйте отправить письмо через какой-нибудь известный сервер и посмотрите, каким будет ответ сервера.

Отправка письма нескольким получателям

Добавим возможность отправки письма нескольким получателям. Получатели перечисляются через запятую в поле `edTo`.

Необходимо сформировать из него список всех получателей. Хранить его будем в переменной типа `TStringLines` (объявите ее в разделе `private` формы `FMail`).

```
    Recipients: TStringLines;
```

Это объект, т.е. его необходимо создавать вручную и освобождать память, когда он больше не нужен.

Создавать будем в событии `OnCreate` формы `FMail`, т.е. сразу после создания формы.

```
  Recipients := TStringList.Create;
```

В обработчик закрытия формы добавьте удаление списка:

```
  Recipients.Free;
```

Заполнять список будем непосредственно при нажатии кнопки «Отправить», до соединения, чтобы к моменту отправки список уже был готов.

```
  Recipients.Clear; //очистка списка: он мог уже быть создан
  str := Trim(EdTo.Text); //содержимое EdTo с удаленными пробелами
  while pos(',', str) > 0 do //если есть запятые, нужно вырезать получателя
  begin
    Recipients.Add(copy(str, 1, pos(',', str)-1));//добавление в список
    Delete(str, 1, pos(',', str)); //удаление из строки
    str := Trim(str);//обрезка пробелов
  end;
//после последней запятой может быть еще один получатель
//если запятых не было, то он - единственный
  if str <> '' then
    Recipients.Add(str);
//если ни один получатель не указан - ошибка
  if Recipients.Count = 0 then
  begin
    //сообщение об ошибке
    MessageDlg('Ошибка! Необходимо указать получателя.',mtError,[mbOK],0);
    //установить курсор
    EdTo.SetFocus;
    //прервать выполнение процедуры отправки
    exit;
  end;
```

Не забудьте объявить `str` в локальных переменных.

По аналогии с последним блоком, самостоятельно добавьте проверку, был ли указан отправитель письма.

Основные изменения коснутся отправки команды RCPT TO и ответа на нее. Если раньше после RCPT TO сразу шла отправка команды DATA, то теперь нужно повторить команду RCPT TO, если еще не всех получателей сообщили серверу. Нам потребуется переменная, в которой будем хранить номер текущего получателя. Объявите ее в разделе `private` формы `FMail`.

```
    CurrentRecipient: Integer;
```

В процедуре `DiaOnMailFrom`, где мы отправляем адрес первого получателя, укажем, что текущий номер 0. Отправлять теперь нужно не весь текст из `EdTo`, а только одну строку из `Recipients`.

```
    CurrentRecipient := 0;
    SendText('RCPT TO: <' + Recipients[CurrentRecipient] + '>');
```

В `DiaOnRcptTo` необходимо увеличить счетчик на `1`. Если есть еще получатели, отправить следующего, иначе – команду `DATA`.

```
  CurrentRecipient := CurrentRecipient + 1;
  if copy(Msg, 1, 3) = '250' then
  begin
    if CurrentRecipient < Recipients.Count then
      SendText('RCPT TO: <' + Recipients[CurrentRecipient] + '>')
    else
    begin
      CurrentState := smtpDATA;
      SendText('DATA');
    end;
  end
```

Еще одно отличие заключается в обработке ошибок. На данный момент отправка письма прервется, если хотя бы один получатель недоступен. Правильнее будет сообщить о невозможности доставки конкретному получателю, а отправку прервать только если все получатели недоступны.

Нам потребуется еще одна переменная – счетчик успешно принятых получателей `AcceptedRecipients: Integer`. Объявите ее в разделе private формы, а в `DiaOnMailFrom` присвойте значение 0.

В `DiaOnRcptTo` внесите следующие изменения.

При успешной отправке необходимо увеличить `AcceptedRecipients` на 1, в противном случае – добавить запись в лог, но не обрывать соединение.

```
  if copy(Msg, 1, 3) = '250' then
  begin
    AcceptedRecipients := AcceptedRecipients + 1;
    ...

  end
  else
  begin
    mmLog.Lines.Add('Получатель недоступен: ' + Recipients[CurrentRecipient-1]);
end;
```

Команду DATA следует отправлять только, если число получателей больше 0, в противном случае – сообщить об ошибке и завершить соединение.

```
      if AcceptedRecipients > 0 then
      begin
        CurrentState := smtpDATA;
        SendText('DATA');
      end
      else
      begin
        mmLog.Lines.Add('Ошибка при отправке письма! '
         + 'Ни один получатель не принят сервером.');
       CurrentState := smtpQUIT;
       SendText('QUIT');
      end;
```

Проверьте работу клиента через Denwer. Его почтовый сервер примет всех получателей. Файл письма будет создан один.

Отправка письма с авторизацией и шифрованием. Indy

К сожалению, используя предыдущий подход, будет затруднительно реализовать клиент, способный работать с реальными серверами в Интернет. Мы реализовали его в большей степени для учебных целей.

Его можно было бы развить еще немого, добавив авторизацию по протоколу ESMTP, но сервер Denwer ее не поддерживает, а на сегодняшний день все серверы в Интернет требует шифрования по протоколу SSL/TLS. Стандартные средства Delphi не позволяют подключить шифрование к `TClientSocket`.

Поэтому мы перенесем программу на более продвинутый уровень и реализуем ее работу через компоненты Indy. Краткие сведения об Indy можно найти в Приложении 2.

Мы существенно перепишем и упростим приложение, поэтому создайте новую копию всего разработанного проекта. К сдаче необходимо предоставить оба варианта программы.

Также допустимо сделать одну общую программу, в которой будет опция «использовать Indy» для управления способом отправки (+1 балл).

Подключение компонентов Indy

В новой копии программы компонент `ClientSocket1` удалите. Также удалите или закомментируйте связанные с ним процедуры, процедуры `Dia...`, тип `TSMTPState` и все переменные на форме, а также все вызовы. Фактически, мы возвращаем проект в первоначальное состояние. Добейтесь, чтобы проект компилировался без ошибок.

Разместите на форме `FMail` два компонента:

- `IdSMPT1: TIdSMTP` – клиент SMTP, в него вносятся настройки подключения (хост, порт, логин, пароль);
- `IdMessage1: TIdMessage` – компонент, содержащий само письмо.

Все настройки для них оставьте по умолчанию.

Все команды для отправки письма будут находиться в обработчике нажатия кнопки «Отправить».

Сначала заполним письмо.

Адрес отправителя:

```
IdMessage1.From.Address := EdFrom.Text;
```

Перечень получателей составляется также, как и раньше, только вместо собственной перемененной мы используем список получателей в `IdMessage1`:

```
IdMessage1.Recipients.Clear;
str := Trim(EdTo.Text);
while pos(',', str) > 0 do
begin
  IdMessage1.Recipients.Add.Address := copy(str, 1, pos(',', str)-1);
  Delete(str, 1, pos(',', str));
  str := Trim(str);
end;
if str <> '' then
  IdMessage1.Recipients.Add.Address := str;
if IdMessage1.Recipients.Count = 0 then
begin
  MessageDlg('Ошибка! Необходимо указать получателя.',mtError,[mbOK],0);
  EdTo.SetFocus;
  exit;
end;
```

Тема письма и дата отправки:

```
IdMessage1.Subject := EdSubject.Text;
IdMessage1.Date:= Now;
```

Текст письма добавляется одной общей командой:

```
IdMessage1.Body.Clear;
IdMessage1.Body.AddStrings(MmBody.Lines);
```

Далее, необходимо указать хост и порт сервера в настройках `IdSMTP1`:

```
IdSMTP1.Host := FStart.EdServerAdress.Text;
IdSMTP1.Port := FStart.EdServerPort.Value;
```

Наконец, осуществляем подключение, отправку и отключение:

```
  IdSMTP1.Connect;
  try
    IdSMTP1.Send(IdMessage1);
  finally
    IdSMTP1.Disconnect;
  end;
```

Нам не удастся вести лог так же подробно, как раньше. Через Indy мы не увидим самих сообщений, отправленных клиентом и сервером. Можно лишь отметить основные шаги передачи через событие `OnStatus` сокета `IdSMTP1`. В параметрах его обработчика имеется поле `AStatusText`, которое уже содержит пояснение на английском языке.

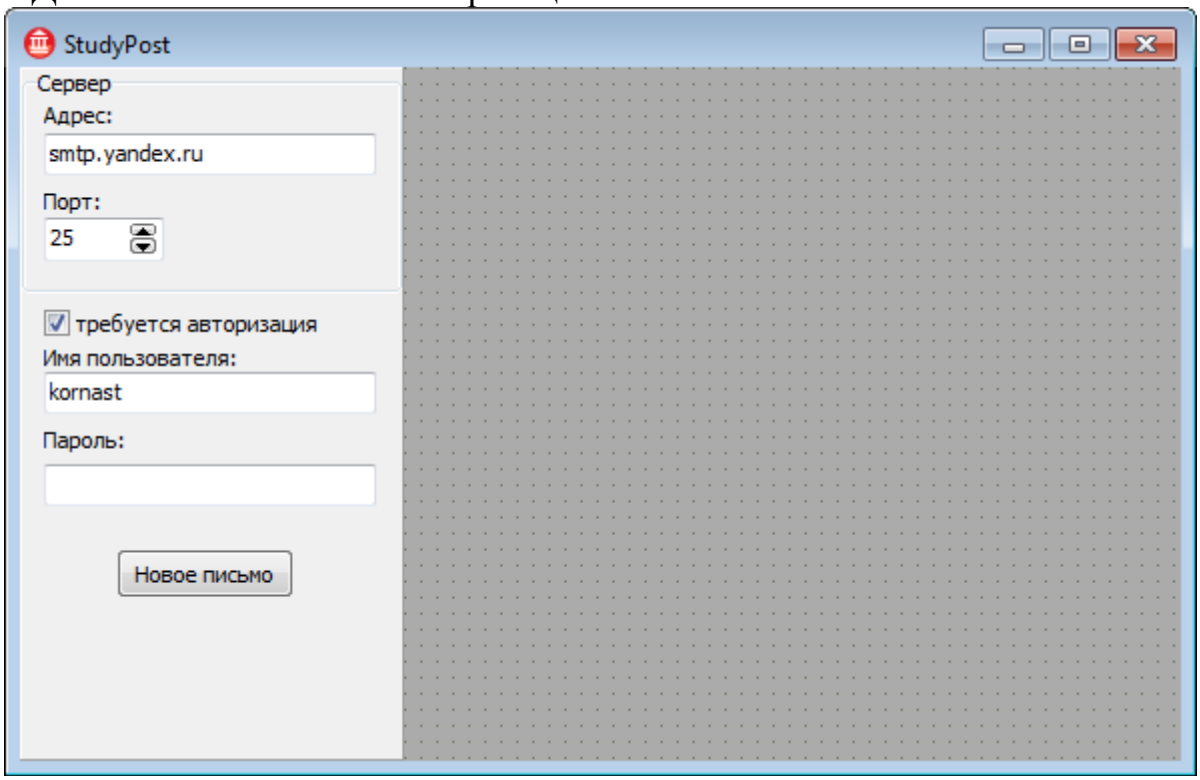
```
procedure TFMail.IdSMTP1Status(ASender: TObject; const AStatus: TIdStatus;
  const AStatusText: string);
begin
  mmLog.Lines.Add(TimeToStr(Now())+ ' ' + AStatusText);
end;
```

Для вывода пояснений на русском языке нужно расшифровать `AStatus`.

Теперь наша программа может выполнять все те же действия, что и раньше. Проверьте ее работу с помощью Denwer.

Авторизация

Добавим возможность авторизации.



Разместите на главной форме:

- поле для ввода имени пользователя `EdUserName: TEdit;`
- поле для ввода пароля `EdPassword: TEdit`, в свойство `PasswordChar` введите символ, скрывающий символы пароля[1];

[1] например * или символ • (скопируйте из Word)

- флажок `ChkBoxAuth: TCheckBox`, свойство `Checked = True`, надпись «требуется авторизация»;
- метки с пояснениями.

В обработчик кнопки «Отправить» добавьте ввод логина и пароля, если отмечен `ChkBoxAuth`.

```
if FStart.ChkBoxAuth.Checked then
begin
  IdSMTP1.Username := FStart.EdUserName.Text;
  IdSMTP1.Password := FStart.EdPassword.Text;
end;
```

Если испытать программу через Denwer, авторизация не сработает, т.к. его сервер не поддерживает расширения ESMTP.

Чтобы испытать ее работу на реальном сервере осталось добавить возможность шифрования.

Шифрование

Добавьте на форму FMail еще один компонент `IdSSL1: TIdSSLIOHandlerSocketOpenSSL` со вкладки Indy I/O Handlers, а на форму FStart еще один флажок `ChkBoxSSL: TCheckBox` с текстом «требуется шифрование SSL/TLS».

В обработчик кнопки «Отправить», если отмечен `ChkBoxSSL`, необходимо подключить `IdSSL1` к `IdSMTP1`.

```
if FStart.ChkBoxSSL.Checked then
begin
  IdSSL1.Destination := IdSMTP1.Host+':'+IntToStr(IdSMTP1.Port);
  IdSSL1.Host := IdSMTP1.Host;
  IdSSL1.Port := IdSMTP1.Port;
  IdSMTP1.IOHandler := IdSSL1;
  IdSMTP1.UseTLS := utUseExplicitTLS;
end;
```

Теперь письмо можно отправить с вашего обычного почтового ящика. Чтобы не дискредитировать свой логин и пароль, можно создать временный тестовый аккаунт.

Настройки для подключения SMTP можно узнать в справке почтового сервиса. Например, для Яндекса:

- адрес почтового сервера — smtp.yandex.ru;
- защита соединения — SSL;
- порт — 465.

Вложения в письмо

Осталось добавить еще одну важную возможность – добавление вложений в письма.

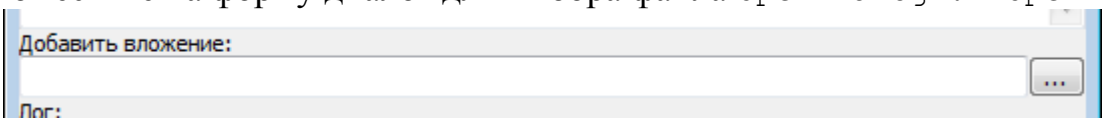
Разместите на форме `FMail` панель высотой 21 пиксель с `Align=alBottom`.

На панели разместите:

- поле для ввода имени файл `EdAttachment: TEdit` с `Align=alClient`;
- кнопку для выбора файла `BtnAttachOpen: TButton`;

Метку с подписью поместите на самой форме с `Align=alBottom`.

Поместите на форму диалог для выбора файла `OpenDialog1: TOpenDialog`.



Обработчик клика по кнопке выбора файла:

```
if OpenDialog1.Execute then
  EdAttachment.Text := OpenDialog1.FileName;
```

Для добавления вложения к письму потребуется объект `TIdAttachmentFile`. Это не компонент, его нельзя разместить на форме, а только создать в программе. Для работы с ним необходимо объявить `IdAttachmentFile` в `uses`.

В код отправки письма добавьте:

```
if EdAttachment.Text <> '' then //если пользователь указал файл
 if FileExists(EdAttachment.Text) then //и если этот файл существует
   //создать вложение
   TIdAttachmentFile.Create(IdMessage1.MessageParts, EdAttachment.Text)
 else //иначе - файл не найден
 begin
  MessageDlg('Ошибка! Указанный файл не существует.',mtError,[mbOK],0);
  EdAttachment.SetFocus;
  exit;
 end;
```

Самостоятельно добавьте возможность приложения к письму нескольких файлов.

Для этого разместите на форме кнопку «Добавить» и `TMemo`, в котором будет перечень всех файлов. При нажатии на кнопку «Добавить» текст из `EdAttachment` добавляется в `Memo`, а сам `EdAttachment` очищается.

В коде отправки письма необходимо организовать цикл, в котором будут добавлять все строки из `Memo`.

Для более продвинутой организации вложений (на дополнительные баллы) используйте `TListBox` или `TListView`, добавьте кнопку «Удалить».

Приложение 1. Перечисляемые типы

Перечисляемые (или перечеслимые) типы данных – это типы данных, для которых явно указаны все допустимые значения. Обычно таких значений немного, максимально – 256.

Например, тип данных – дни недели:

```
type
  TWeekDays = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
var
  Day: TWeekDays;
...

Day := Monday;
```

Переменной `Day` можно присвоить значения только из перечня в `TWeekDays`.

Каждое из значений соответствует его порядковому номеру при перечислении, т.е.

```
Monday = TWeekDays(0)
Tuesday = TWeekDays(1)
Wednesday = TWeekDays(2)
Thursday = TWeekDays(3)
Friday = TWeekDays(4)
Saturday = TWeekDays(5)
Sunday = TWeekDays(6)
```

По-другому, то же самое можно записать через константы и целочисленную переменную:

```
const
  Monday = 0;
  Tuesday = 1;
  Wednesday = 2;
  Thursday = 3;
  Friday = 4;
  Saturday = 5;
  Sunday = 6;
var
  Day: Byte;
...

Day := Monday;
```

Но в такой записи нет контроля допустимых значений, т.е. в переменную `Day` можно записать, например, значение 100, хотя такого дня недели не бывает.

На практике используются оба подхода в равной мере.

Приложение 2. Компоненты Indy

Indy – это кроссплатформенная открытая библиотека для различных языков программирования, официальный сайт The Indy Project: indyproject.org.

В RAD Studio входит VCL-библиотека Indy.Sockets, которая включает множество компонентов для работы с сокетами и различными прикладными протоколами – SMTP, POP3, IMAP, NNTP, HTTP, FTP и др. На сайте Indy Project заявлено более 100 протоколов.

В компоненты Indy уже «зашиты» все команды протоколов, которые мы реализовывали вручную. Достаточно внести необходимую информацию, все остальное выполняется автоматически. Иными словами, сокеты Indy имеют высокий уровень абстракции.

Важное отличие Indy от использованных нами сокетов – они работают в **блокирующем** (синхронном) режиме. TClientSocket и TServerSocket по умолчанию работают в **не блокирующем** (асинхронном) режиме.

В чем разница? В не блокирующем режиме мы открывали соединение или отправляли сообщение, а потом ждали, когда придет ответ и сработает событие OnRead (OnClientRead).

В блокирующем режиме все команды пишутся подряд и срабатывают сразу. Т.е. после команды отправки текста в том же участке кода пишется команда получения ответа.

Команда Connect у Indy-сокетов, в отличие от Open обычных сокетов, возвращает значение True при успешном подключении, и False в противном случае.

Типичный пример сессии с условным Indy-клиентом:

```
with IndyClient do
begin
  Host := 'example.com'; // хост
  Port := 80; // порт
  Connect;
  try
    // здесь пишем все нужные команды
  finally
    Disconnect;
  end;
end;
```

Все время, пока выполняются команды, приложение будет занято, а при длительном выполнении может «подвиснуть». Если в процессе, например, выводить записи в лог, они отобразятся только после завершения соединения. Также не будет отображаться прогресс в процентах и т.п.

Чтобы избежать этого, следует вызвать принудительную обработку событий в программе с помощью следующей команды:

```
Application.ProcessMessages;
```

В идеале, использование блокирующих сокетов следует дополнять потоками (Thread), которые обрабатываются отдельно от основного приложения и не мешают его работе.

В практике программирования на Delphi для работы через сеть чаще всего используют именно Indy.