AxoNet Software GmbH

Introduction

Using the VCL client components for asanetwork

Contents

Overview

This introduction shows the basic use of the VCL client components 1.4 for asanetwork clients (test and measurement equipment). It is assumed that you have a basic knowledge of asanetwork and of the asanetwork SDK, especially the C++/Delphi Interface.

These components are available for Borland Delphi 4, 5 and 6 and Borland C++ Builder 4 and 5. The only software needed from asanetwork is the awn32b.dll together with the interface definitions in awn16/32.pas.

Installation

Please close any IDE of Borland Delphi or C++ Builder before you start the installation. Then start the installation program DVclClientx.x.exe (demo version) or VclClientx.x.exe (full version). x.x = Version number, e.g. 1.4.

The installation program offers 3 choices:

- Typical - installs everything (Delphi 4, Delphi 5, Delphi 6 and C++ Builder 4, C++ Builder 5)
- Minimal - installs only Delphi 6, C++ Builder 5
- Custom - you may choose the package yourself

After installation restart your IDE and you have a new palette called AxoNet which contains 2 entries: TAwnSimpleService and TAwnControl.

In your installation directory (typically "c:\program files\Axonet Software GmbH\VclClient") you find a sub dir called examples. There you find examples for every supported and installed version of Delphi or C++ Builder.

Before you start

First you need to define the required services for asanetwork.

You always have a customer order service based on your license name (DID). In this demo we use the name TEST_. For your applications please replace this TEST_ DID with the one you'll find in your license agreement.

Then you have one or more test and measurement services as defines in the "Design guide" of asanetwork. The VCL components support test and measurement services with XML data (these start with AWNTX) and manufacturer specific services (these start with your DID).

For our demo we assume that we have these services in our equipment:

- TEST_00000 customer order services
- AWNTXBR000 Brake test (only the general service)
- AWNTXEM000 Emission test, general service, sub divided into
  - AWNTXEM010 Emission test, gas engine, no catalyst
  - AWNTXEM020 Emission test, gas engine, open loop catalyst
  - AWNTXEM030 Emission test, gas engine, closed loop catalyst
  - AWNTXEM050 Emission test, diesel engine

Then you have to define the service location (DLOC) and the specific properties of each service. This data is mapped into an INI file structure like this:

[AWN]

DLoc=	The DLoc of your application goes here
Orders=	Order service, e.g. TEST_00000
Results=	All general result services separated by ; e.g. AWNTXBR00000;AWNTXEM00000
{Debug= optional}	0 = off, 1 = enabled, displays the debug window from the AxAwn COM library
[RESULTSERVICE]	Name of the general result services, repeated as necessary, e.g. AWNTXEM00000
SubService=	Service names of all subservices separated by ; e.g. AWNTXEM010;AWNTXEM020
DiQual=	Input quality of service, if missing defaults to 1
DoQual=	Output quality of services, if missing defaults to 1
DPrio=	Priority of service, if missing defaults to 9
[SUBSERVICE]	Name of sub service, repeated as necessary
DiQual=	Same as above
DoQual=	
DPrio=	

A working example looks like this awn.ini (from our demo project)

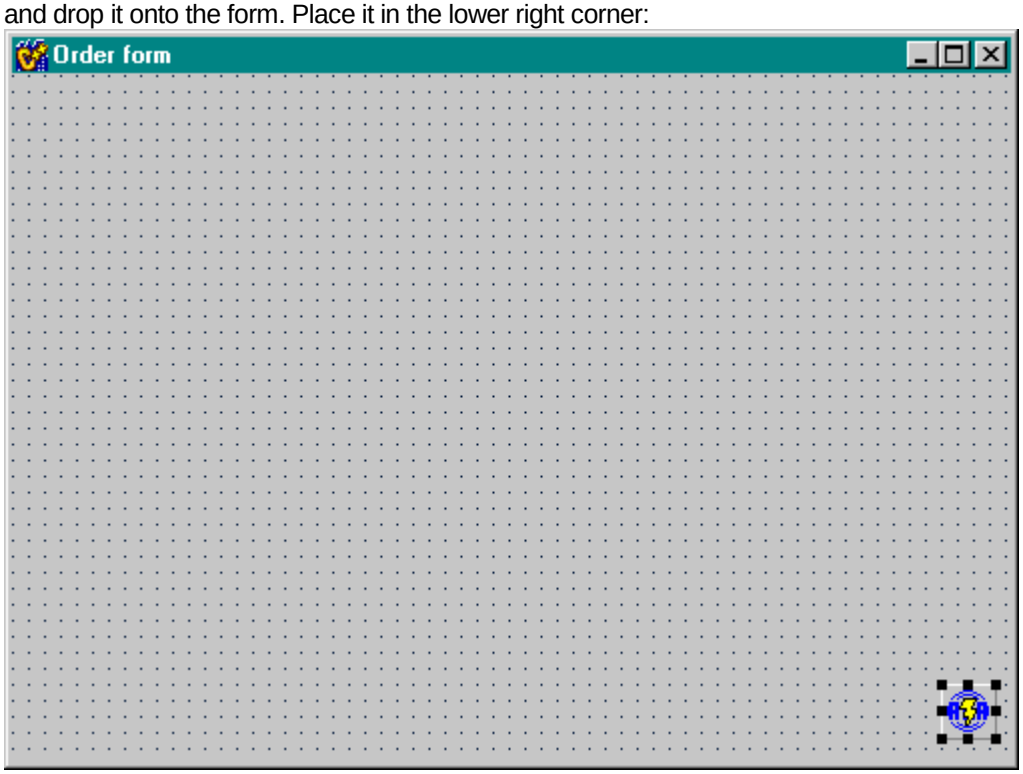
```
[AWN]
DLoc=AwnTest
Orders=TEST_00000
Results=AWNTXBR000;AWNTXEM000

[TEST_00000]
Info=customer order services for AwnTest

[AWNTXBR000]
Info=Brake test

[AWNTXEM000]
SubService=AWNTXEM010;AWNTXEM020;AWNTXEM030;AWNTXEM050
Info=Emission test, general service

[AWNTXEM010]
Info=Emission test, gas engine, no catalyst

[AWNTXEM020]
Info=Emission test, gas engine, open loop catalyst

[AWNTXEM030]
Info=Emission test, gas engine, closed loop catalyst

[AWNTXEM050]
Info=Emission test, diesel engine
```


Building a customer order client

Step 1, building the first demo application

In this introduction we'll use Borland C++Builder but you can use Borland Delphi in just the same way.
Start a new project, save the first form as order and the project as demo1.prj. Then add the **AwnCtl** component from the AxoNet palette:

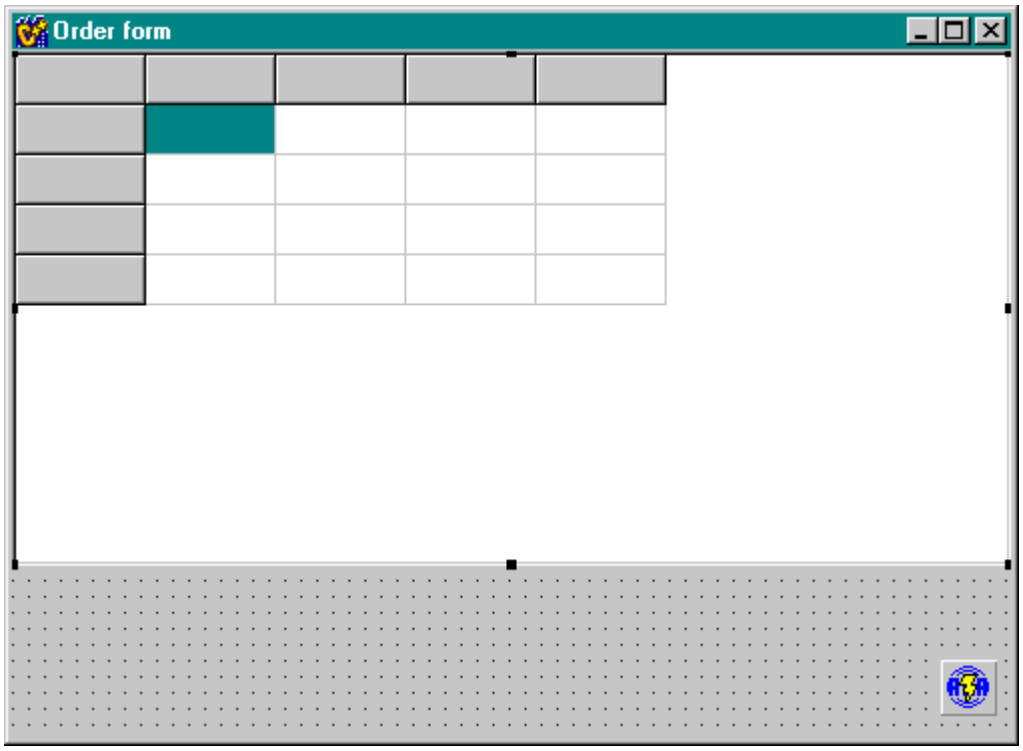
and drop it onto the form. Place it in the lower right corner:

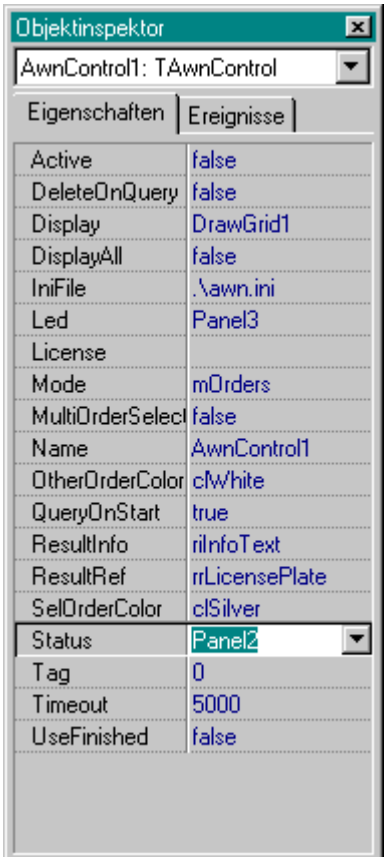


Now look at the property inspector. We'll add some data. First put in the name of our previously created INI file. Here we use awn.ini. Please note the properties Display, Status and Led. These are used to link visible components to our invisible control. We'll use this in the next step.

The Mode property is mOrders because we are building an order client.

Place a TDrawGrid on the form:



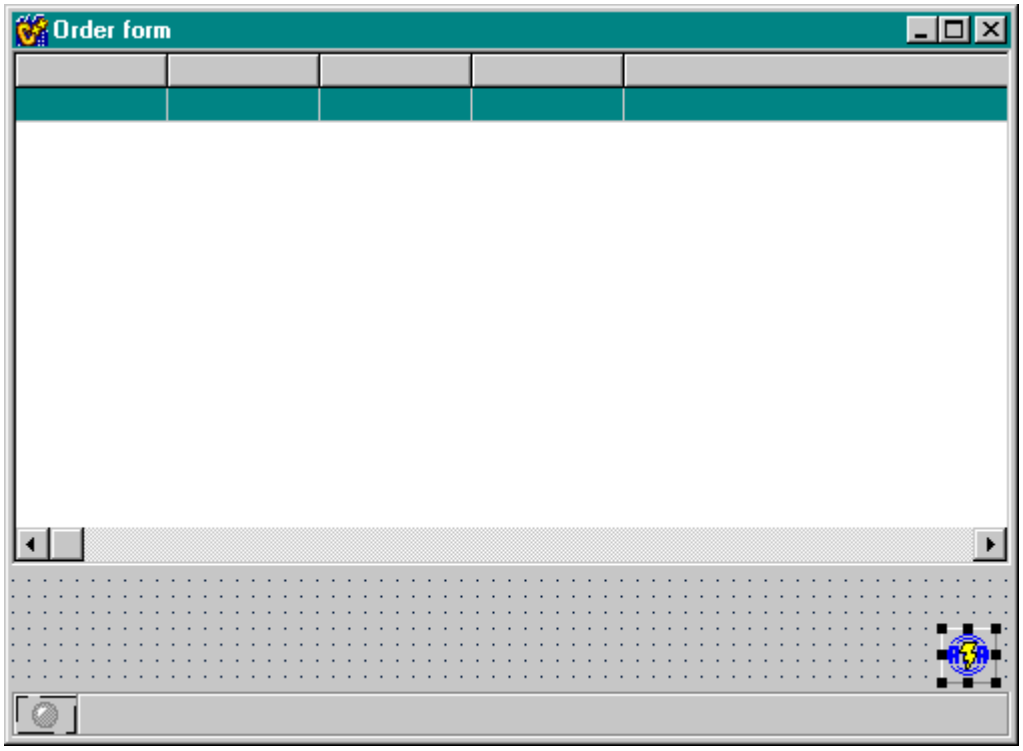


Select the control and link the display property to the DrawGrid1.

Next place a TPanel on the form (see below). Adjust the alignment to alBottom. Change BevelOuter to bvLowered. Place two TPanel inside the first panel. Align one alLeft, the other alClient. Panel3 is used for the led. The led display is grey if no connection to network manager has been made. If a connection is established, the led will change to green. If the connection is lost, the led goes to red. Panel2 is used as a status line for the TDrawGrid. For the selected order the control displays some additional information. Both panels are only used for layout purpose. You can place and align them, as you like. And if you don't want a led, simply leave the property empty.

Select the control. Go into the property inspector and link Led to panel3 and Status to panel2. Now the form should look like this. Please note the grey led and the empty panel2.

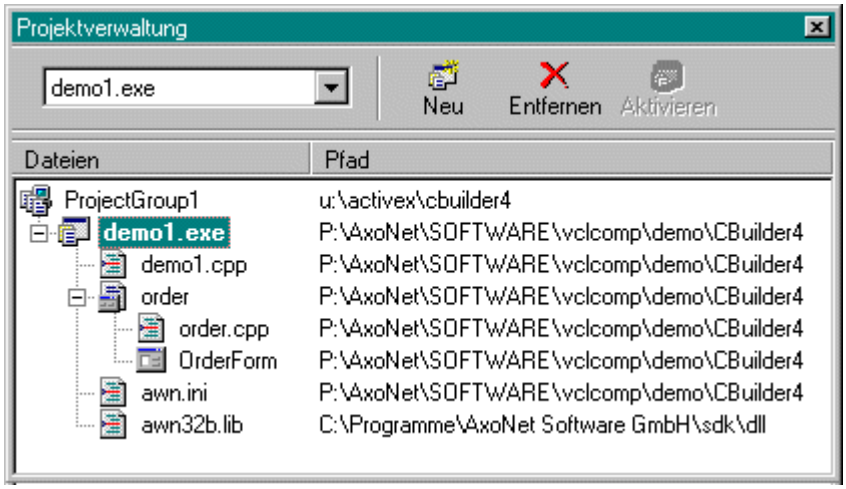
Your form should now look like this:



Before we start our application for the first time, we have to set the license key for the customer order service. Put in your license key from your asanetwork contract.

Then we need two event handlers for the form itself. In the OnCreate event we enable our control, in the OnDestroy event we disable our control:

```
//---------------------------------------------------------------------------

void __fastcall TOrderForm::FormCreate(TObject *Sender)
{
    AwnControl1->Active = true;
}
//---------------------------------------------------------------------------

void __fastcall TOrderForm::FormDestroy(TObject *Sender)
{
    AwnControl1->Active = false;
}
//---------------------------------------------------------------------------
```



The last thing that remains is the asanetwork awn32b.lib necessary for linking. Add this library to the project.

You'll find it in the directory

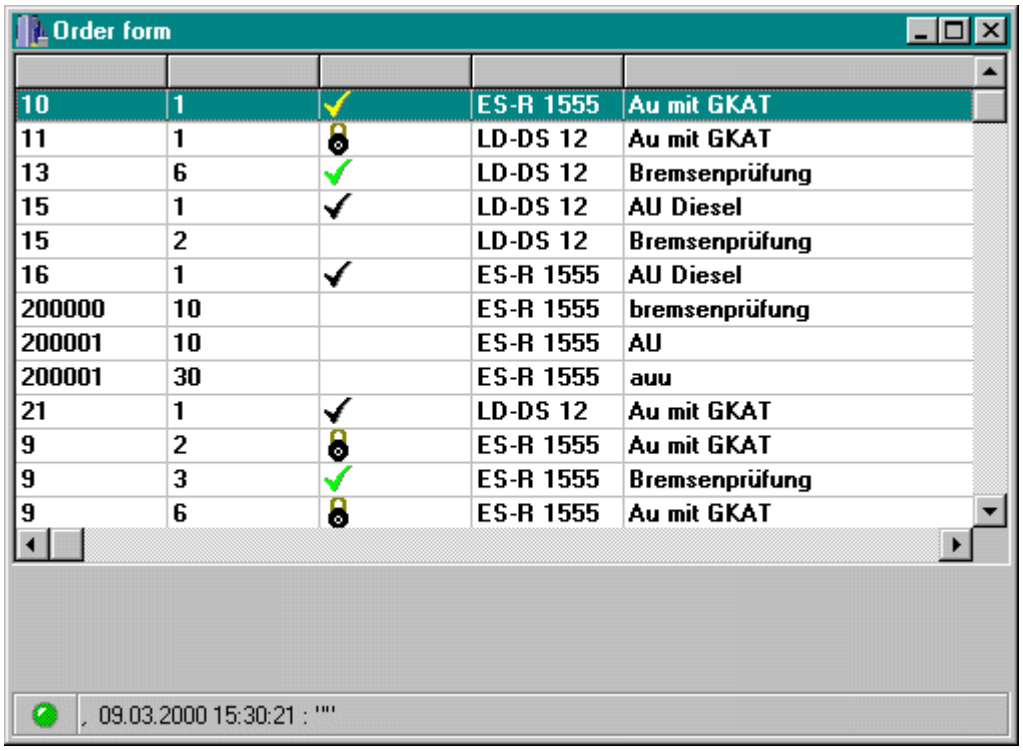
"c:\program files\AxoNet Software GmbH\VclClient"

or

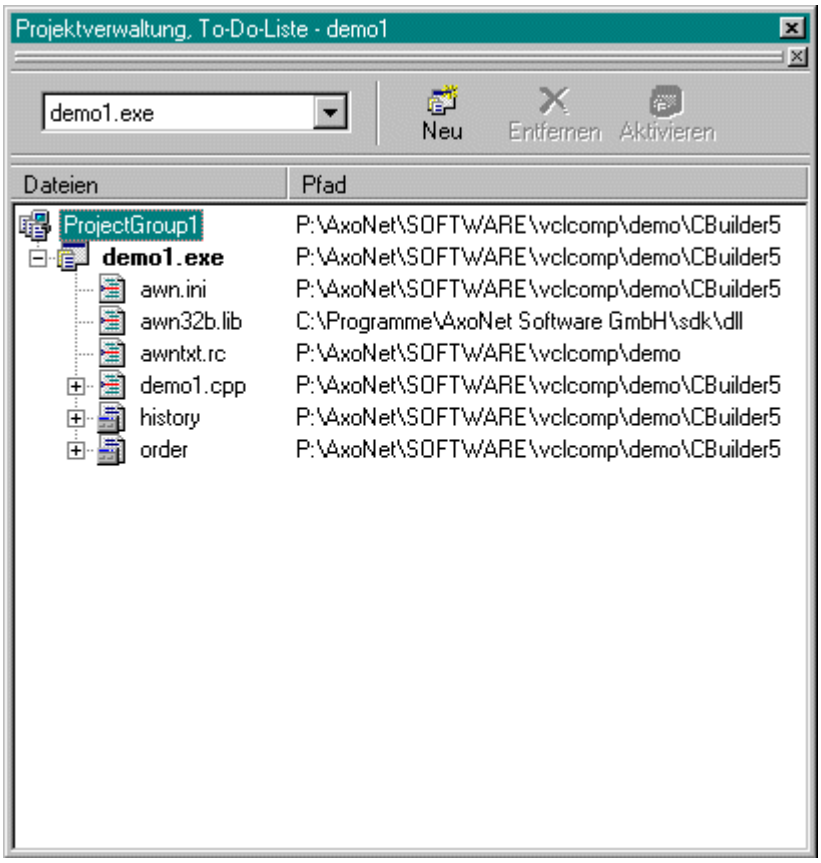
"c:\program files\AxoNet Software GmbH\sdk\dll"

if you installed the asanetwork sdk using the default paths.

Now build and run the application. You'll get a dialog similar to this one. If you don't see any orders please ensure that the network manager is running and that you have orders available. To create orders you can simply start the order generator from the asanetwork SDK.



Seems we forgot something.
Yes, the text strings of the control have been separated into a resource module for easier translation.

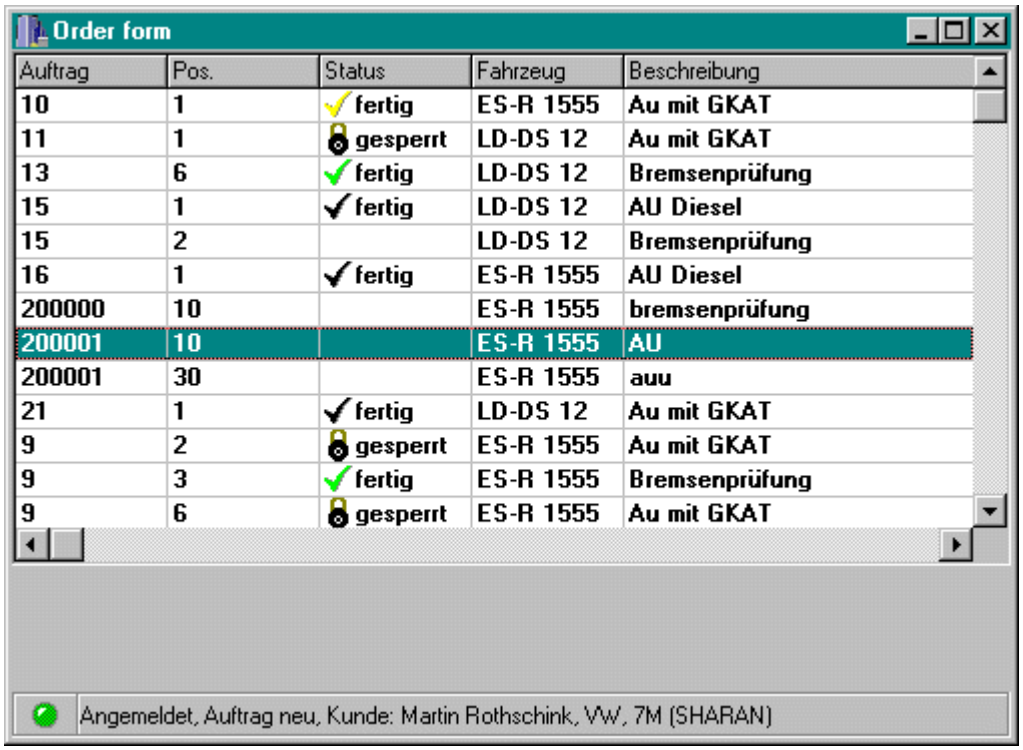


The default text file for German is called awntxt.rc and the default file for English is called awntxten.rc.

You can simply add the source (.rc) file to your project.

If you need other languages, translate either the German or English source file.

And now we get the strings:

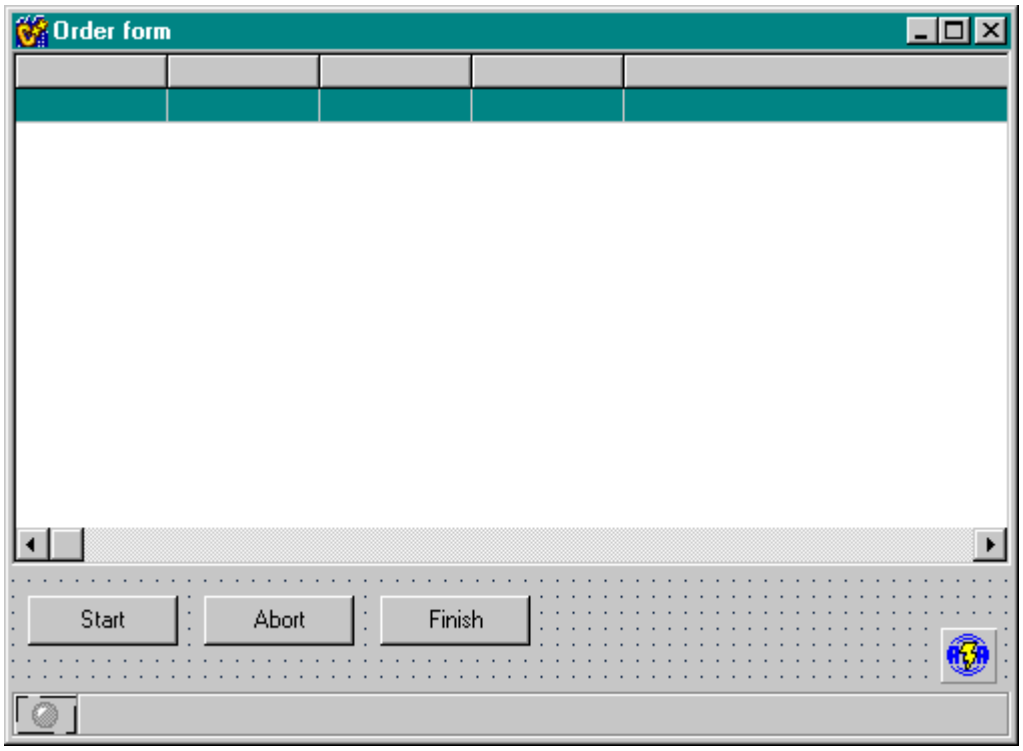


- The first column contains the order number.
- The second column is the position (item) number of this order.
- The third column displays the order state together with a bitmap (see Design guide for a description of these symbols).
- The forth column displays the license plate of the vehicle.
- The fifth and last column display a verbal description of this order position (item).

The bottom line shows a green led – we are currently connected to the network manager. The remaining space is used for a status line. Here the control displays information about the customer and vehicle for new orders. If the order is in process or finished the brief result is displayed.

Step 2, processing orders

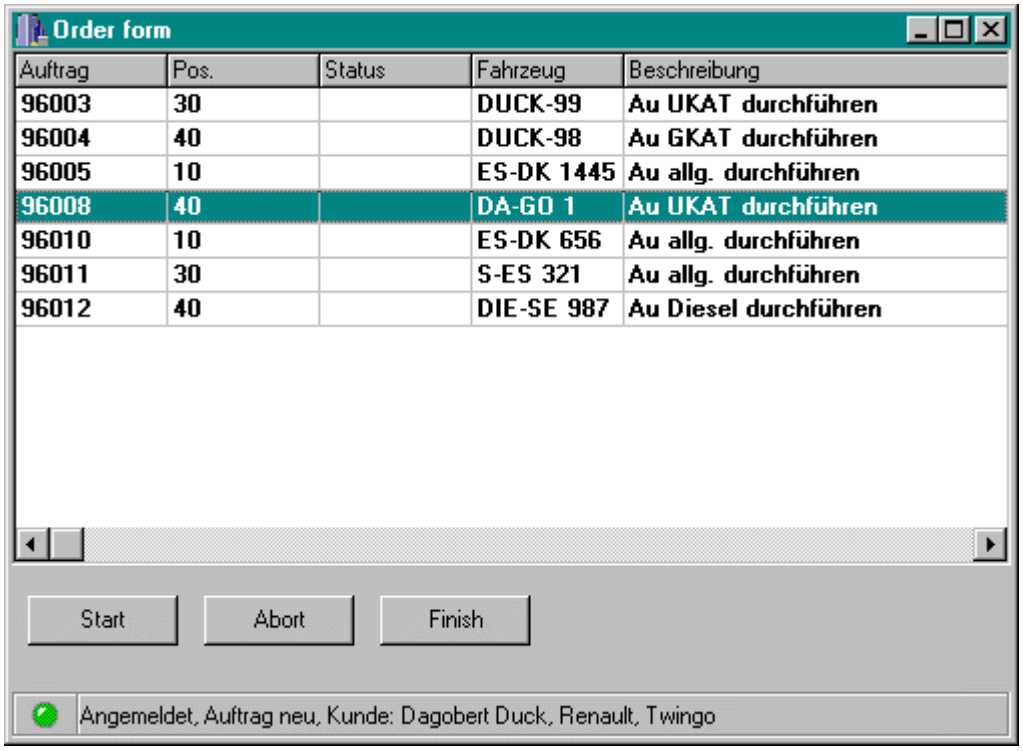
In this step we'll add some buttons to our application to start, abort and finish orders.



Add 3 buttons to your form. Call them "start", "abort" and "finish".

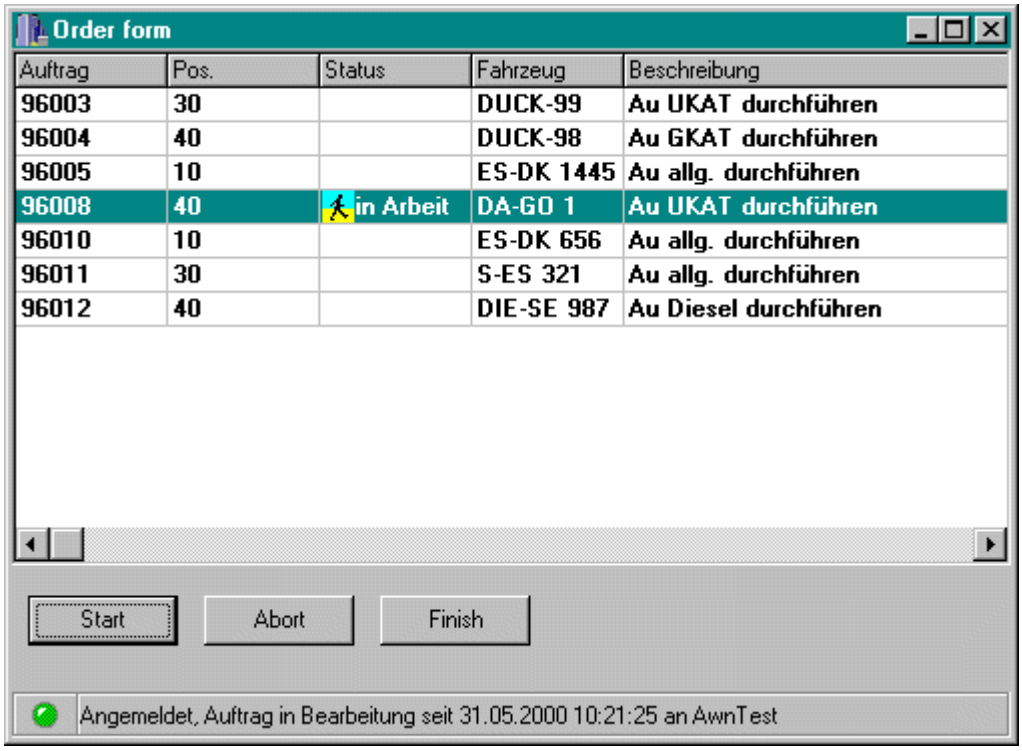
Double click on the buttons to set up the event handler. Add the code shown below to each of the button's event handlers.

```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::StartClick(TObject *Sender)
{
    AwnControl1->SelOrder = AwnControl1->DispIndex;
    AwnControl1->StartOrder();
}
//---------------------------------------------------------------------------
void __fastcall TOrderForm::AbortClick(TObject *Sender)
{
    AwnControl1->AbortOrder( R_DEFAULT, "Test aborted", "");
}
//---------------------------------------------------------------------------
void __fastcall TOrderForm::FinishClick(TObject *Sender)
{
    AwnControl1->FinishOrder( R_OK, "Test passed", "gas.xml");
}
//---------------------------------------------------------------------------
```

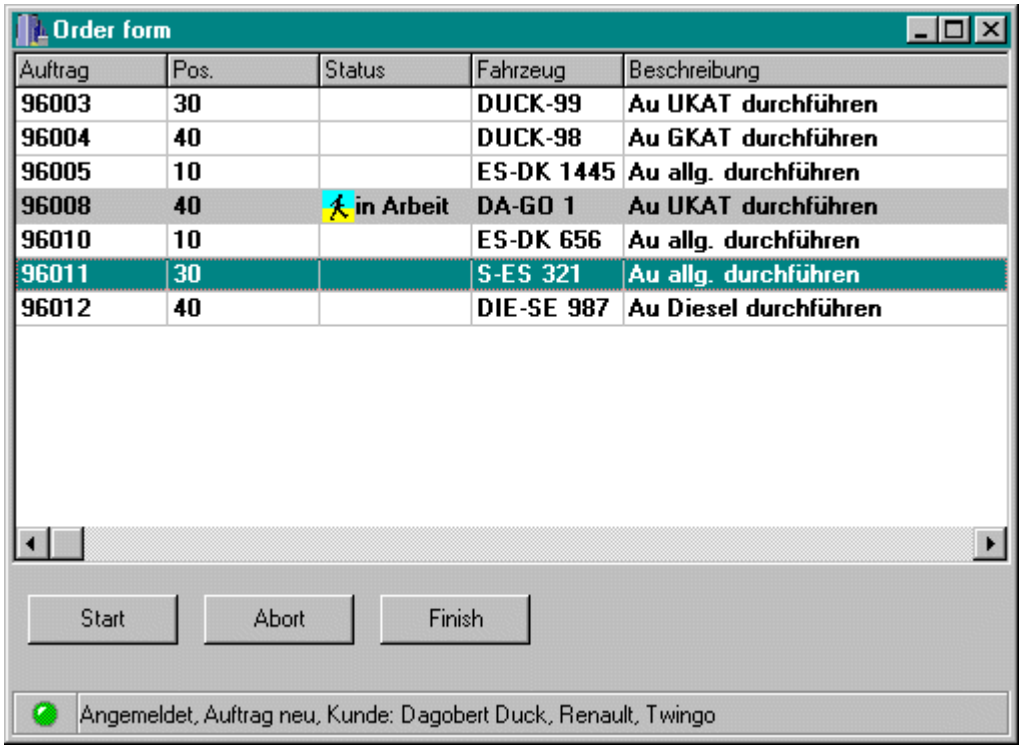



To start an order we use the StartOrder() method. Before we can successfully call this method we need to define the index of the order we want to work with. This can be done either by assigning a valid value to the SelOrder property or by calling using the DispIndex property. DispIndex looks up the index of the currently selected item in the DrawGrid. Assigns this index to SelOrder. Now we can successfully call StartOrder().

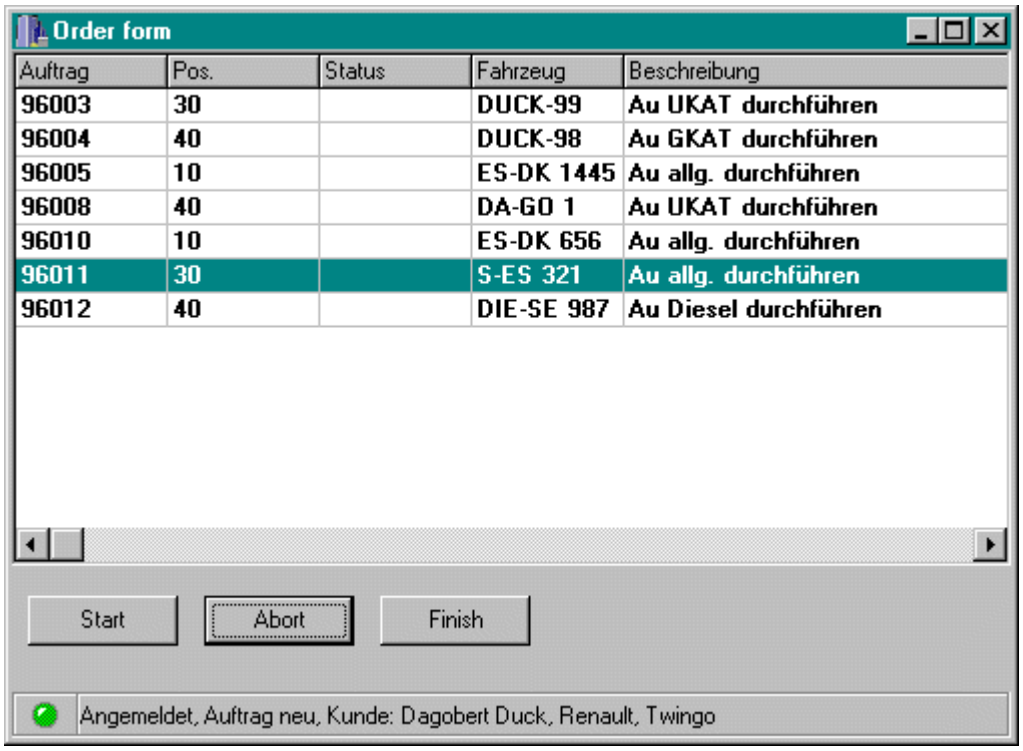
```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::StartClick(TObject *Sender)
{
    AwnControl1-> SelOrder = AwnControl1->DispIndex;
    AwnControl1->StartOrder();
}
//---------------------------------------------------------------------------
```

Select an entry in the TDrawGrid and click on the start button. If a customer order system is running (or the order generator) you'll see that the state changes to active.



If you select any other item inside the TDrawGrid, the currently assigned SelOrder is displayed with a grey background while the status line displays information about the new selected item.



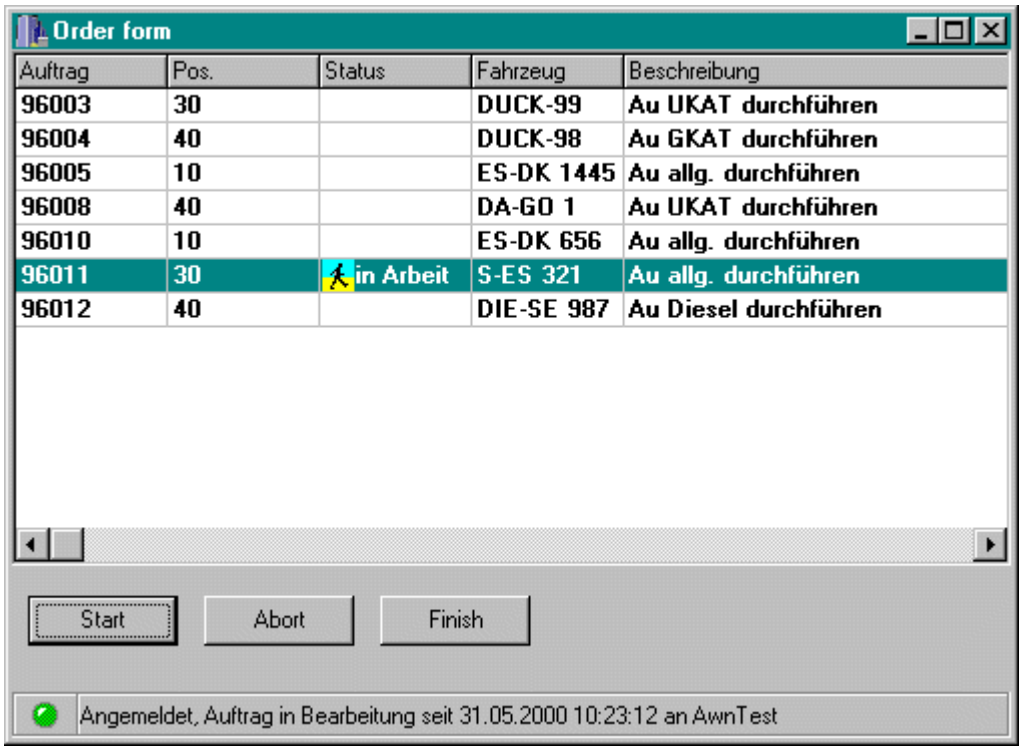
Now click on Abort. The order immediately is set to the new state. You only need to call AbortOrder().

The first parameter is the result code. We didn't started any testing or measurement so we use the default (undefined) result code R_DEFAULT.

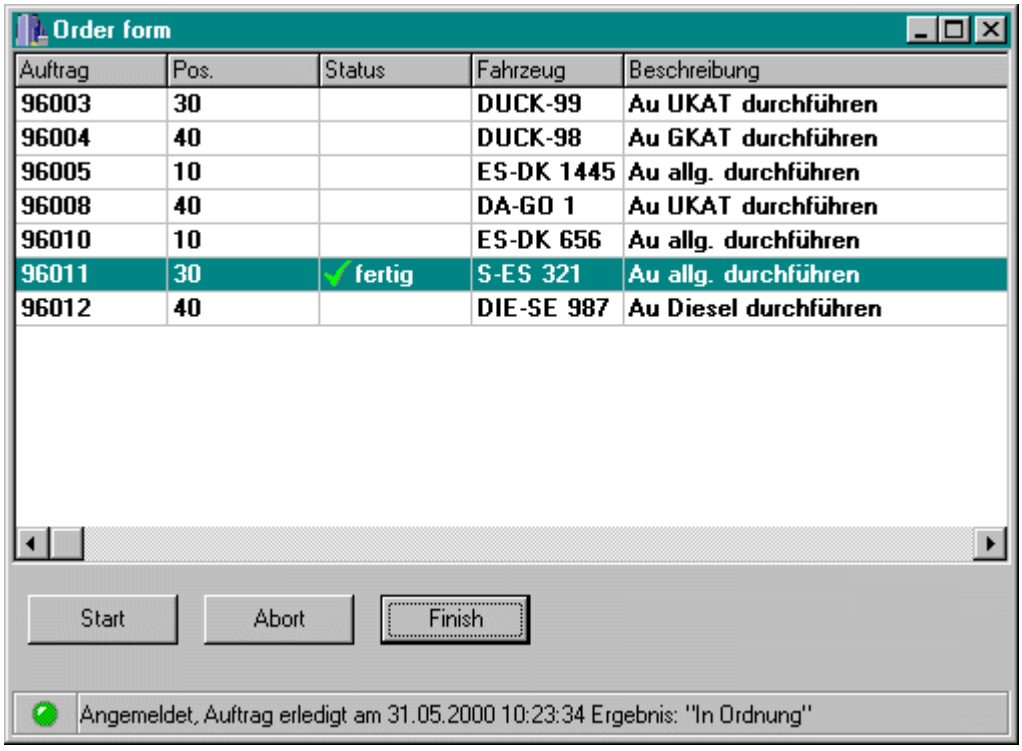
The second parameter is a brief description. Here we put in "Test aborted".

The last parameter is the path name of a file containing result data. If the order was aborted after you already stored some results you can supply the file name here. In our demo we do not use this parameter.

```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::AbortClick(TObject *Sender)
{
    AwnControl1->AbortOrder( R_DEFAULT, "Test aborted", "");
}
//---------------------------------------------------------------------------
```

Start the selected order again. The status line has changed and now displays the start time of processing and the device (DLoc).



Now we're going to finish this order. Click the finish button. The order is immediately set to the finished state. You only need to call AbortOrder().

The first parameter is the result code. In our demo we put in R_OK. Therefore we get a green check mark.

The second parameter is a brief description. Here we put in "Test passed".

The last parameter is the path name of a file containing result data. In our demo we use a static file with some results from a gas analyser.

Once again the status line changed and now displays the end time of processing and a verbal description of the result code (not the brief result).

```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::FinishClick(TObject *Sender)
{
    AwnControl1->FinishOrder( R_OK, "Test passed", "gas.xml");
}
//---------------------------------------------------------------------------
```


Step 3, more properties for orders

In this step we'll show some more properties of the control: The DisplayAll, QueryOnStart and DeleteOnQuery properties.

Let's start with QueryOnStart. If QueryOnStart is true (default) the control automatically sends a query for new and active orders to the asanetwork. In this way you always make sure, that orders are displayed. As long as you don't provide any way to update the list manually, you should not change this property.

DeleteOnQuery (which defaults to false) controls what happens with any existing list. If false the list is not changed before a query is executed. If true, the list is cleared (deleted) and rebuild from the query.

What's the difference? If the list is not changed, finished orders remain in the list. If you do any query, only new and active orders are updated. Finished orders are only removed from the list if the dealer management system deletes the whole order. If you clear the list, finished orders are removed and the list is rebuild with neutral and active orders only.

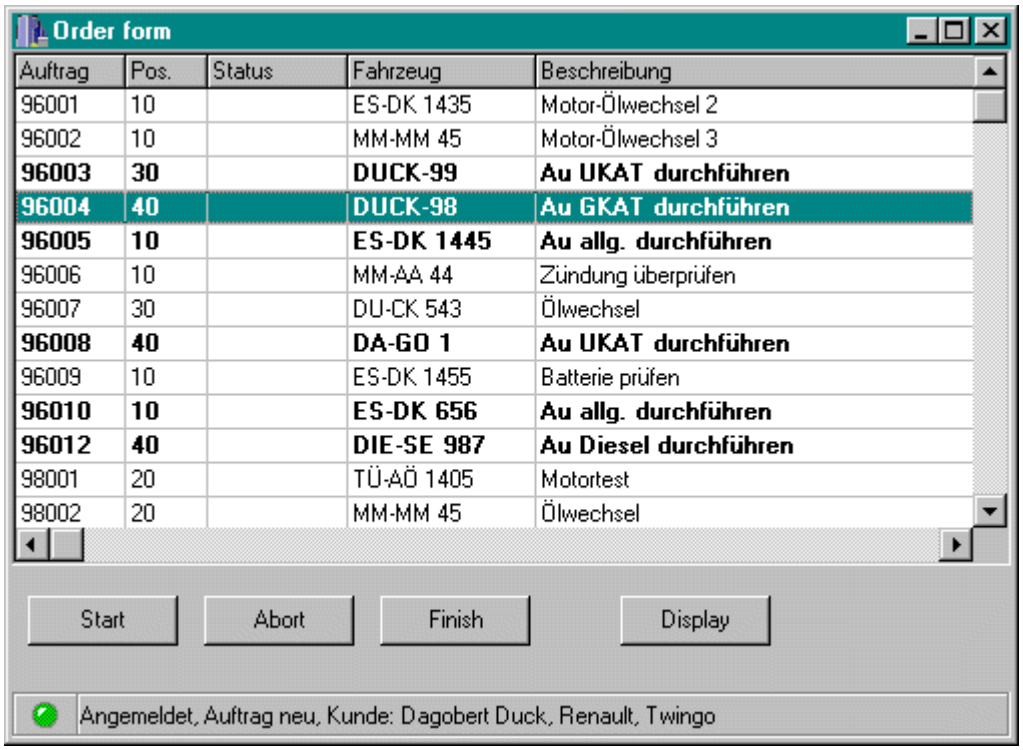
The best approach is to provide both ways to the user. Add 2 buttons called "Update list" and "Rebuild list". Set both QueryOnStart and DeleteOnQuery to false.

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::UpdateListClick(TObject *Sender)
{
    AwnControl1->QueryOrder( "*", AWN_POS_QUERY_ALL);
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::RebuildListClick(TObject *Sender)
{
    try
    {
        AwnControl1->ClearList();
    }
    catch(const EDeleteDisabled &e) {};

    AwnControl1->QueryOrder( "*", AWN_POS_QUERY_ALL);
}
//---------------------------------------------------------------------------
```

Note the exception handler for the ClearList() method. ClearList throws an EDeleteDisabled exception if an order is active.

Now let's look at DisplayAll. Add a button called display and toggle the DisplayAll property inside of the event handler:

```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::DisplayClick(TObject *Sender)
{
    AwnControl1->DisplayAll = !AwnControl1->DisplayAll;
}
//---------------------------------------------------------------------------
```

If you click on Display the grid displays all orders currently available in asanetwork. This is useful if you want to get an overview if there are some more tasks to be done for this vehicle.

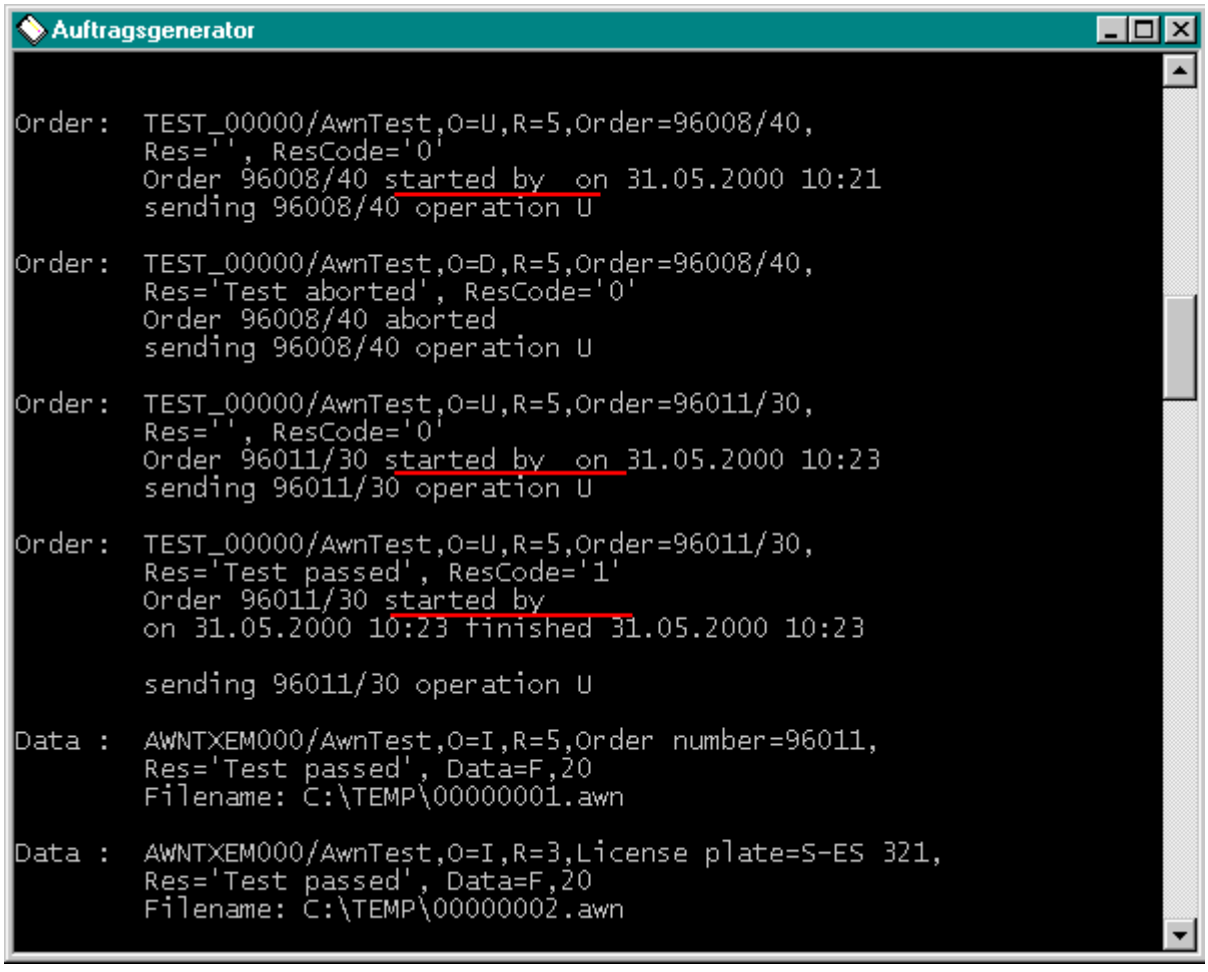
If you click again, only those orders are displayed which are processed by this application. These are displayed in a bold font. You can only select these orders.

Step 4, Accessing order data

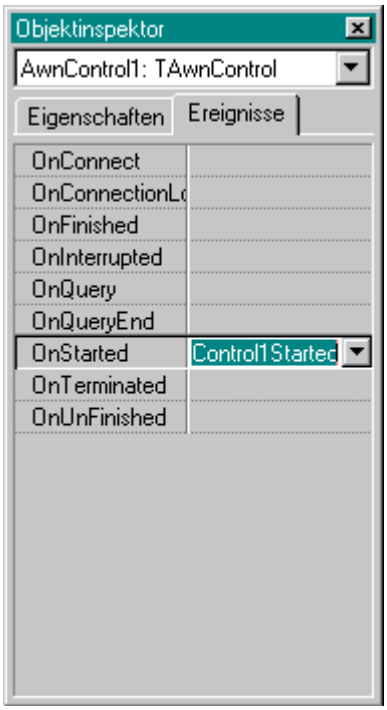
This step shows, how we can easily access the data before the order is sent to the net.

If you look at the order generator, you'll see the sequence of our previously finished order:

Start order 96008/40
Abort order 96008/40
Start other order 96011/30
Finish order 96011/30
Results for order 96011/30
Results for vehicle S-ES 321



Now if you look very carefully at the start order command, you'll notice, that the name of the operator is missing. This is true, because this information has to be supplied by the application. Let's add this information.



Select the control and go to the event page of the property inspector. For every action there's an event. Double click the OnStarted event entry.

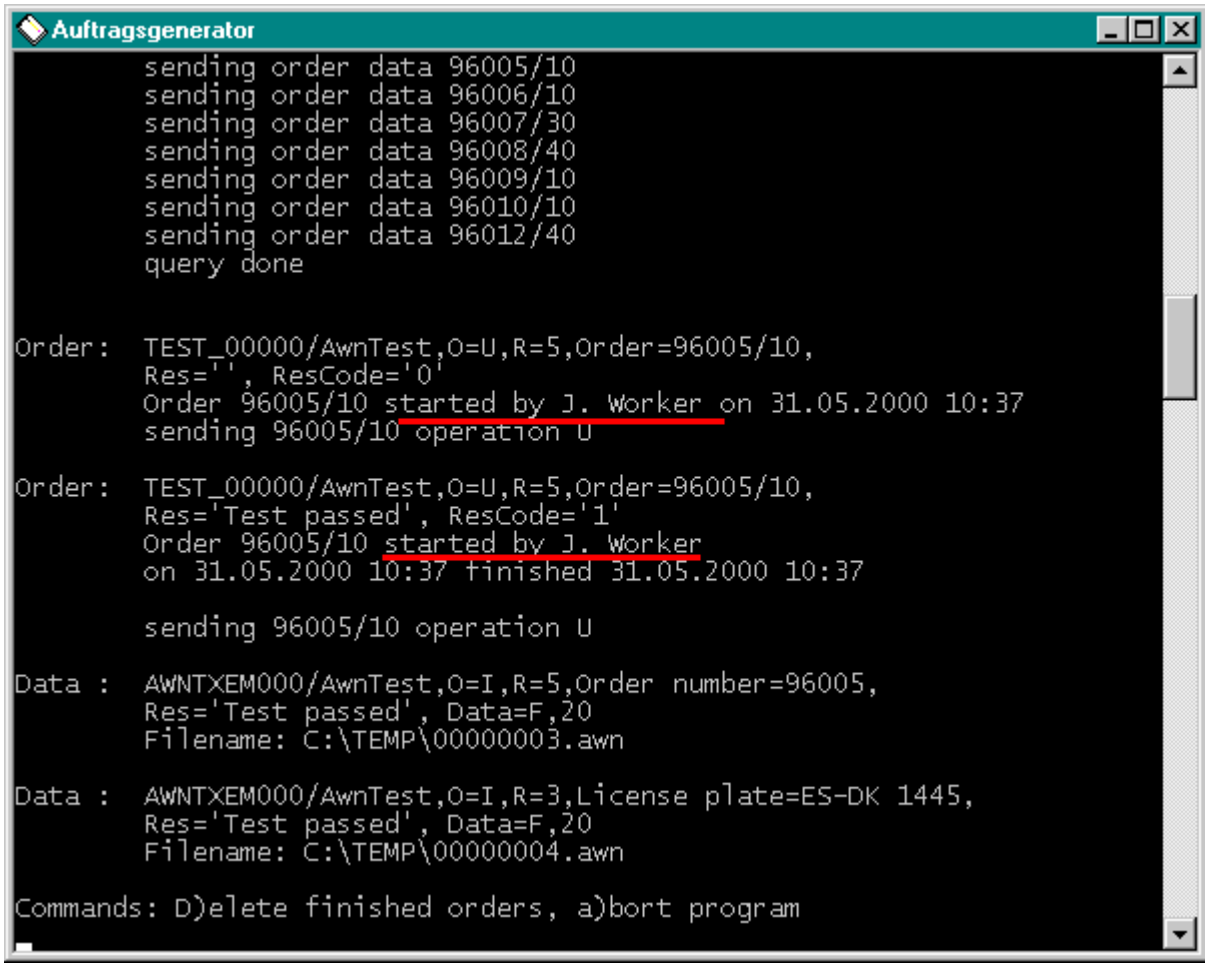
This event is created after you called the StartOrder() method but just before any data is transmitted. So it's the ideal place to do any data modification.

You have direct access to the send and order object – so be careful what you're going to change.

Here we simply add the name of the operator to the order object.

If you start the application again and start an order the order generator now contains the name of the operator (see below).

```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::AwnControl1Started(TObject *Sender,
      IAwnSend *SendObj, IAwnOrder *OrderObj, AnsiString &FileName)
{
    OrderObj->SetOrderRealWorker( "J. Worker");
}
//---------------------------------------------------------------------------
```

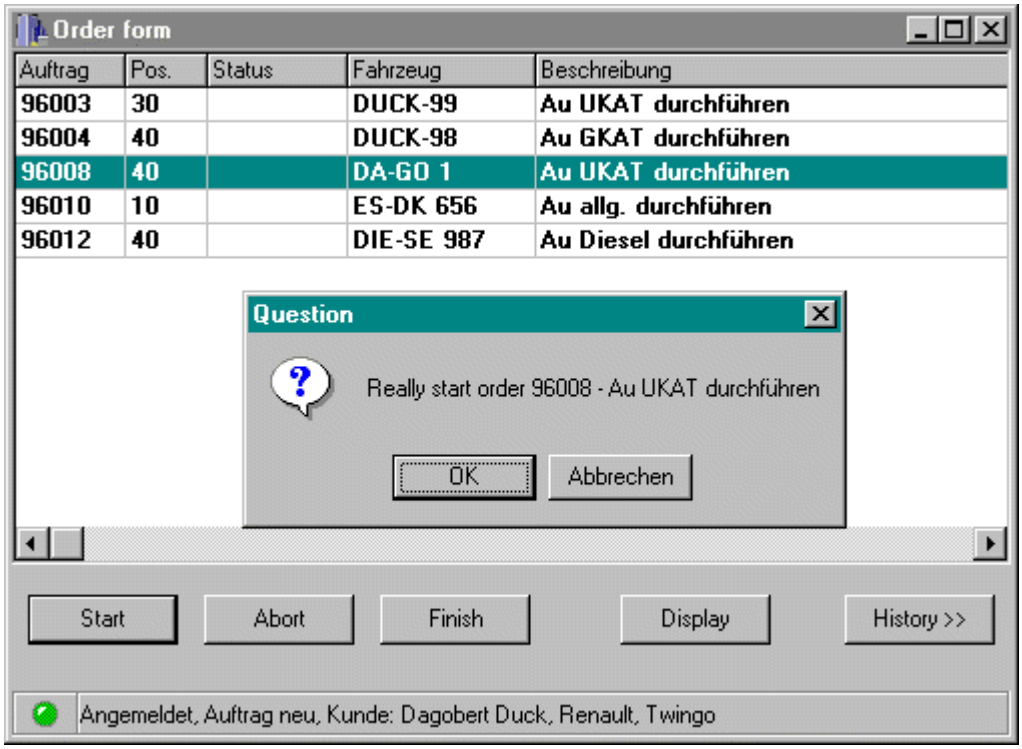
If you need to access the data before any order is active, you can always access the Order[] property. In this example we display a dialog before the order is started. Inside the dialog we display some order data. To access this data we first need the index of the order list. Here we use the DispIndex property to get the index of the currently focused item inside the draw grid. Another way of obtaining the index is the SearchOrder method.

Using the index we can access the IAwnSend object of the Order[] property. Then we extract the order object inside and create an IAwnOrder object. Please note that we are responsible to free the memory after processing.

We extract some data (order number and verbal description), delete the order object and display a dialog. If the user really wants to start this order we assign the index to the SelOrder property and call StartOrder().

You should never store the index in a variable of your own. The list of orders always may grow or shrink. So it's very likely that your index is no longer the same object as before. Always use DispIndex or SearchOrder() to determine the index number. If you already have an active order use the SelOrder property. SelOrder is automatically adopted if the list grows or shrinks.


```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::StartClick(TObject *Sender)
{
    char OrderTxt[256];
    int Index = AwnControl1->DispIndex;
    IAwnSend *Send = AwnControl1->Orders[Index];
    IAwnOrder *Order = (IAwnOrder *)Send->CreateData();

    String Msg = "Really start order "+String(Send->GetReference()->GetOrder());
    Order->GetOrderTxt( OrderTxt);
    Msg += String(" - ")+String(OrderTxt);
    Order->Delete();

    if (Application->MessageBox( Msg.c_str(), "Question",
        MB_OKCANCEL|MB_ICONQUESTION) == ID_OK)
    {
        AwnControl1->SelOrder = Index;
        AwnControl1->StartOrder();
    }
}
//---------------------------------------------------------------------------
```



Step 5, Crash handling

The control automatically keeps track of started orders in the state.dat file as described above. If any order is active, a dirty flag is written to the ini file. If your application terminates unexpectedly, the previously started order is automatically aborted on restart if the dirty flag is present. If you close your application, the dirty flag is removed and the previously started order is automatically resumed on restart. To demonstrate this feature, start an order and then close the application. If you restart the program you'll notice the previous order is displayed again in the selected and active state.

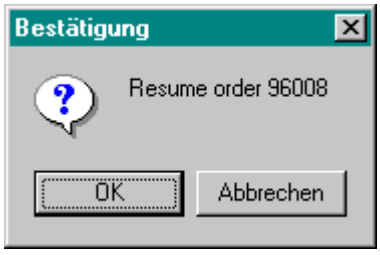
To give you more control over this process two events are available:

The OnCrash event only gives you the chance of showing information because generally it makes no sense to resume a crashed system.

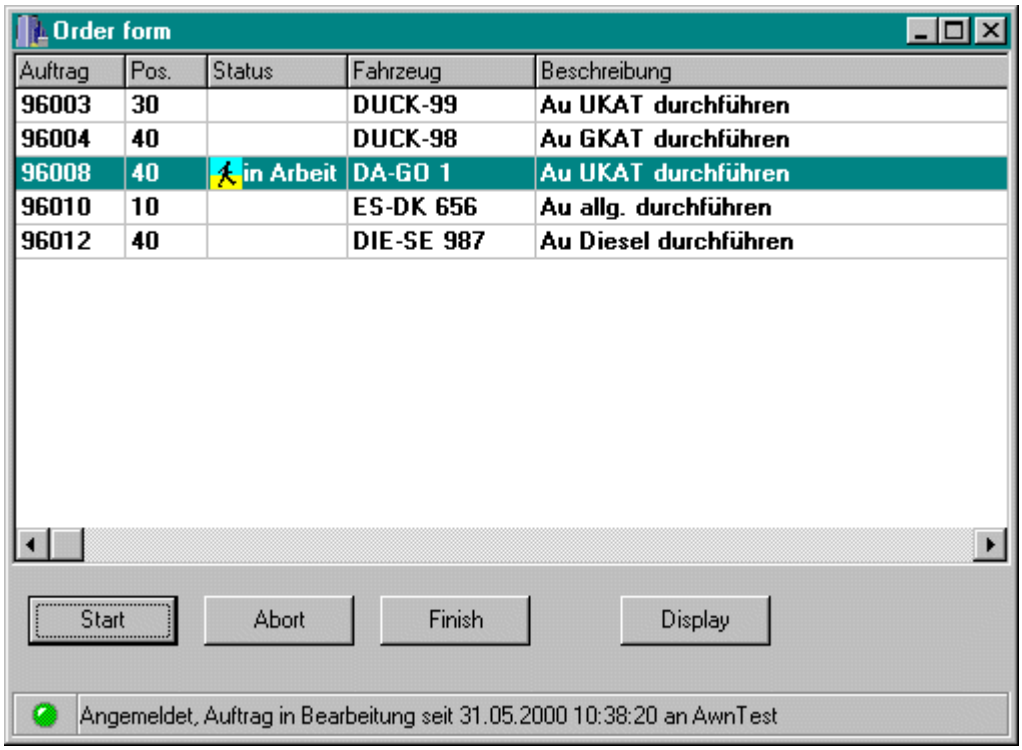
The OnResume event allows you to either resume the order or abort the processing.

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::AwnControl1Resume(TObject *Sender,
      IAwnSend *SendObj, IAwnOrder *OrderObj, AnsiString &FileName)
{
    if (MessageDlg(String("Resume order ")+SendObj->GetReference()->GetOrder(),
        mtConfirmation, TMsgDlgButtons() << mbOK << mbCancel, 0) == mbCancel)
        AwnControl1->AbortOrder( R_ABORTED, "aborted by operator", "");
    else
        ; // enable order data again …
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::AwnControl1Crash(TObject *Sender,
      IAwnSend *SendObj, IAwnOrder *OrderObj, AnsiString &FileName)
{
    MessageDlg(String("Order ")+SendObj->GetReference()->GetOrder()+
               " aborted after application crash!",
               mtInformation, TMsgDlgButtons() << mbOK, 0);
}
//---------------------------------------------------------------------------
```

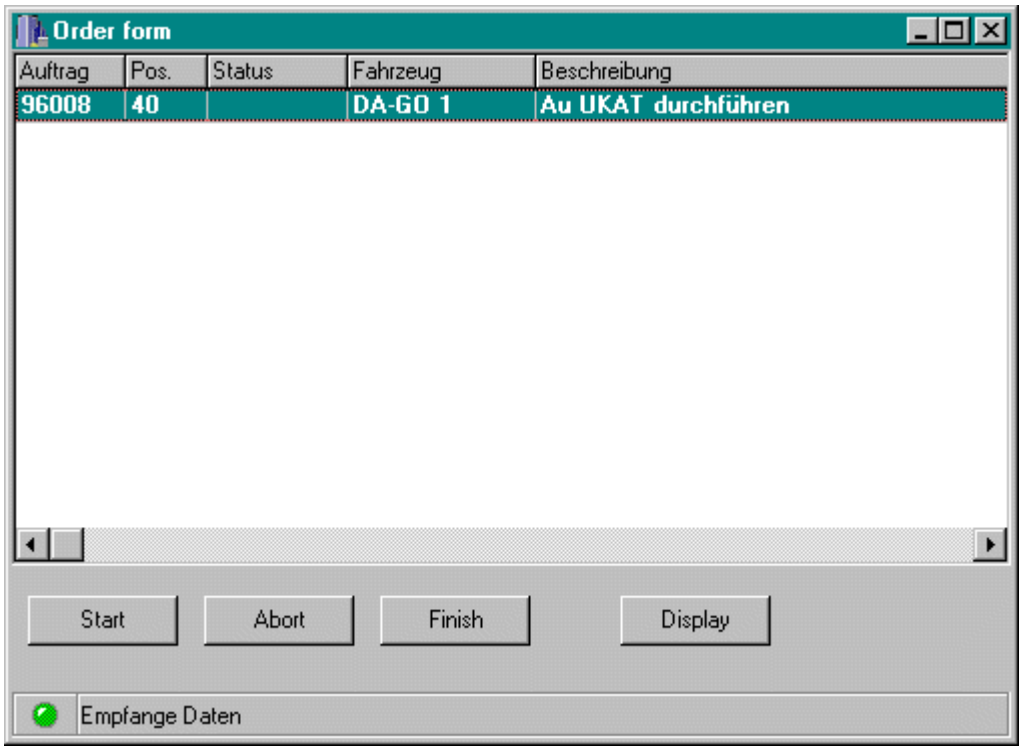
Now start an order and then close the application. If you start the application again, this dialog appears



If you click OK, the previously selected order is again selected and active:



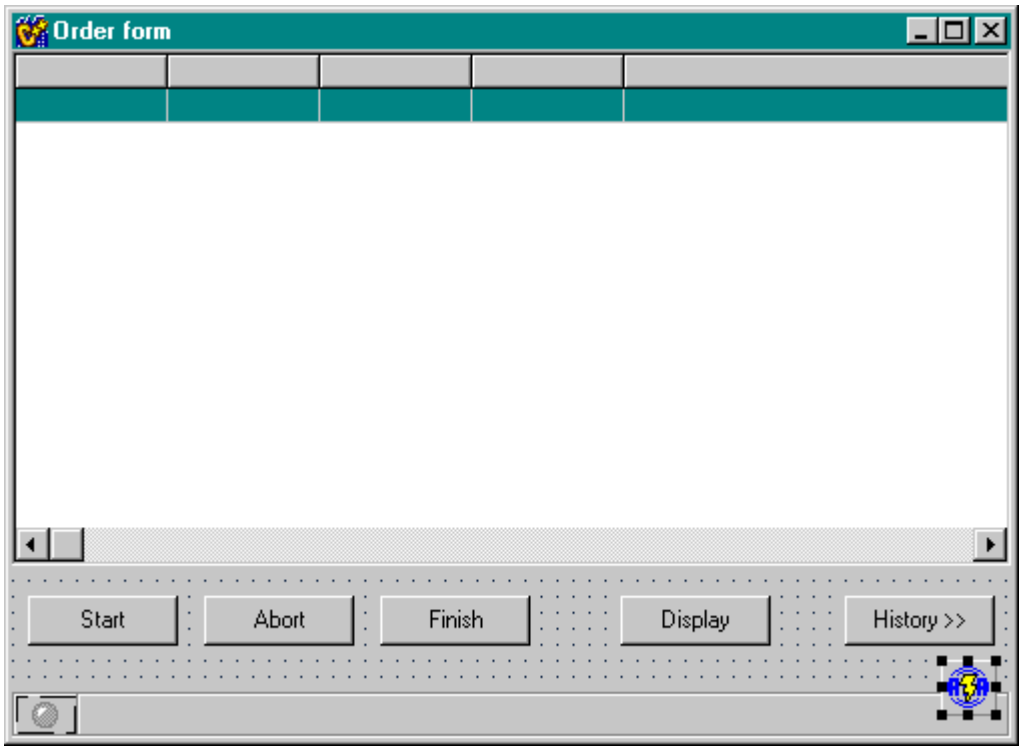
Now kill the application (e.g. use the task manager, go to the process tab, select demo1 and kill the process). Restart the application.



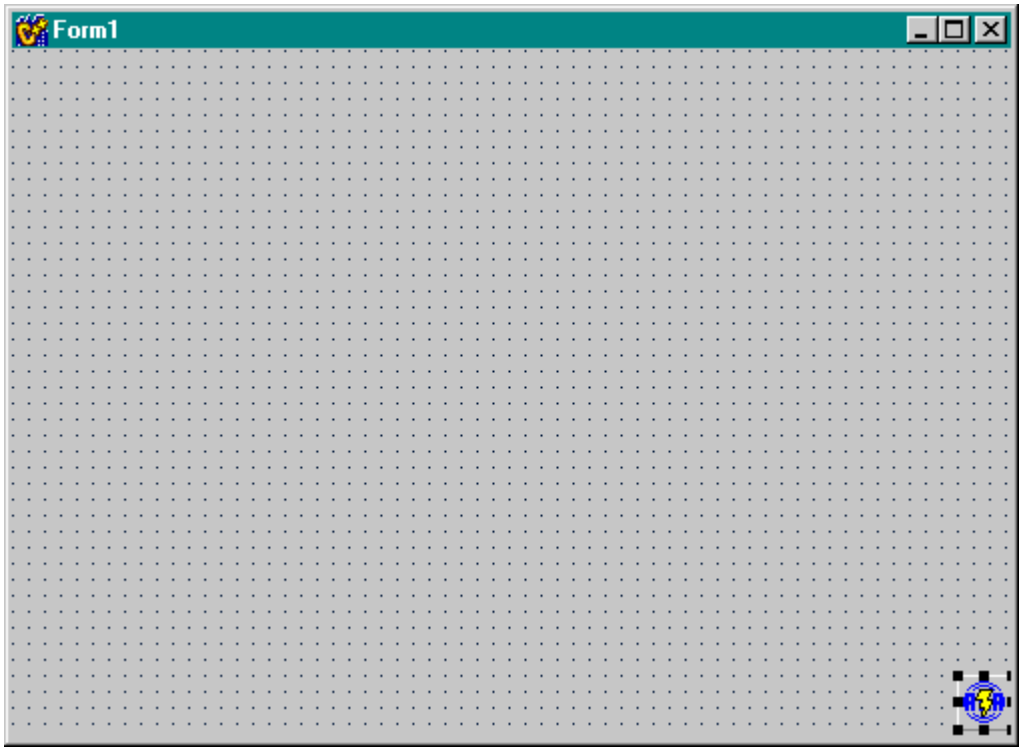
Building a history client

Step 6, the basic form

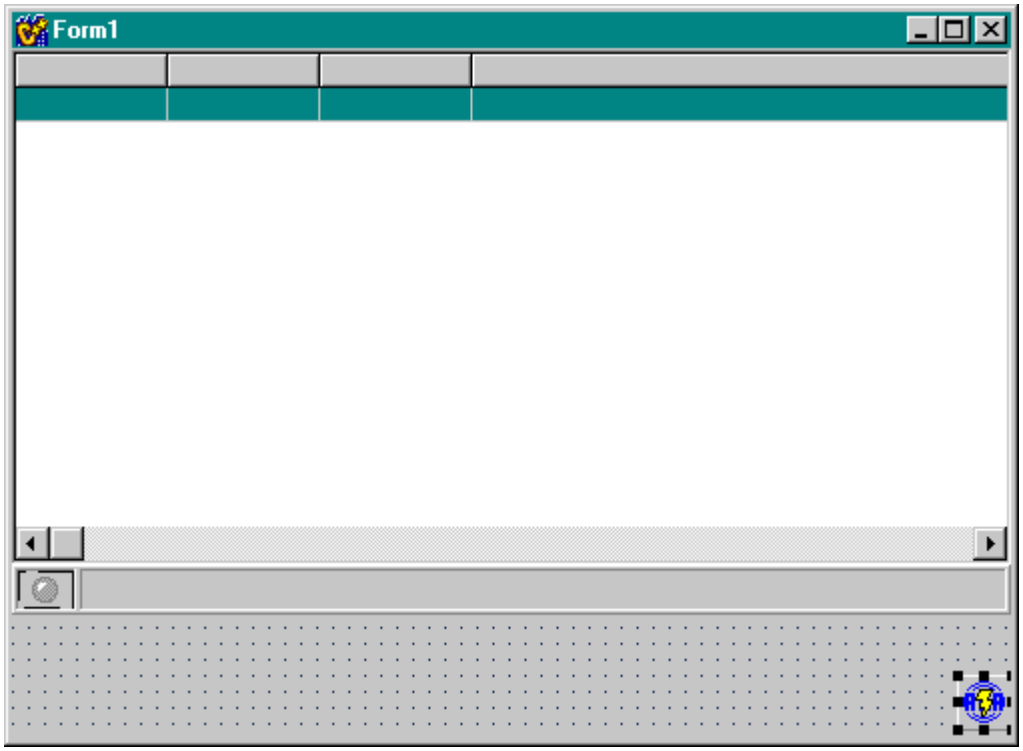
We'll now leave the order handling and start to build a history client. A history client is used to get previously save results back from the asanetwork. You can then display these old results or use it for comparison.
To display the history form, let's add a new button on the order form called History. Later on we add the OnClick code for this button.



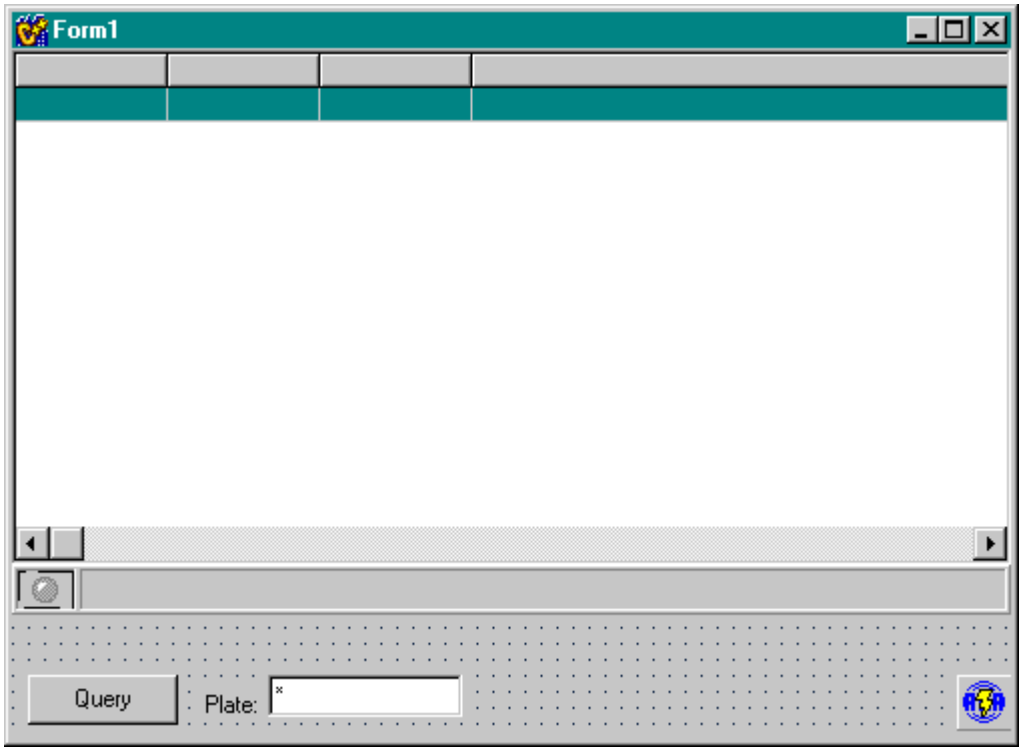
Create a new form and again add a TAwnControl:

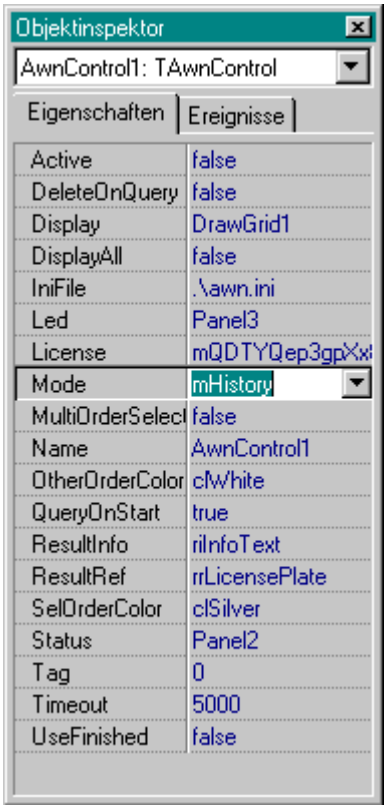


In the same way as before add a TDrawGrid and 3 TPanels for the Led and Status:



Add a button called "Query" and a TEdit for the license plate of the vehicle.





Change the mode property to mHistory and add the Ini file and License as before.

Again add some code to the OnCreate and OnDestroy event of the form to enable or disable the control.

Finally add an event handler for the query button. In this event handler we call the QueryResult method of the control. The first parameter is the service used to send the query. If you enter an empty string, all services defined in the ini file are queried. The second parameter is the license plate we're looking for and the last parameter is the time limit in months. We use 0 = unlimited.

```
//---------------------------------------------------------------------------
void __fastcall THistoryForm::FormCreate(TObject *Sender)
{
    AwnControl1->Active = true;
}
//---------------------------------------------------------------------------
void __fastcall THistoryForm::FormDestroy(TObject *Sender)
{
    AwnControl1->Active = false;
}
//---------------------------------------------------------------------------
void __fastcall THistoryForm::QueryClick(TObject *Sender)
{
    AwnControl1->QueryResult( "", Edit1->Text, 0);
}
//---------------------------------------------------------------------------
```

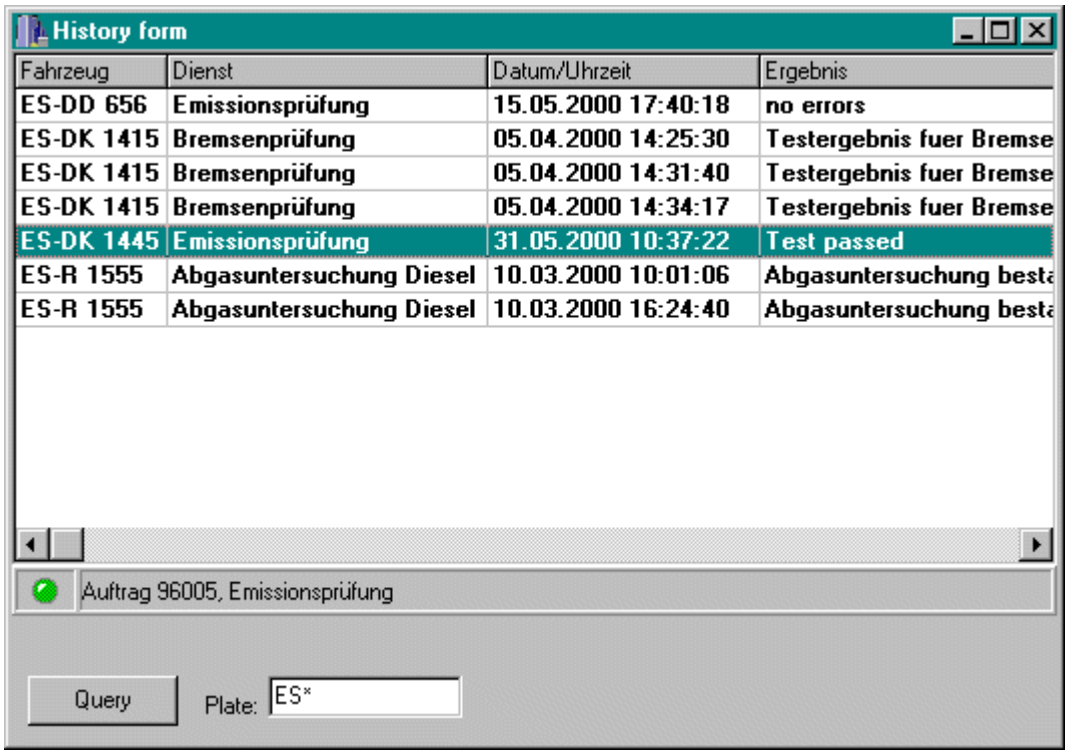
Back in the order form add this event handler for your "History" button:

```
//---------------------------------------------------------------------------
void __fastcall TOrderForm::HistoryClick(TObject *Sender)
{
    HistoryForm->ShowModal();
}
//---------------------------------------------------------------------------
```

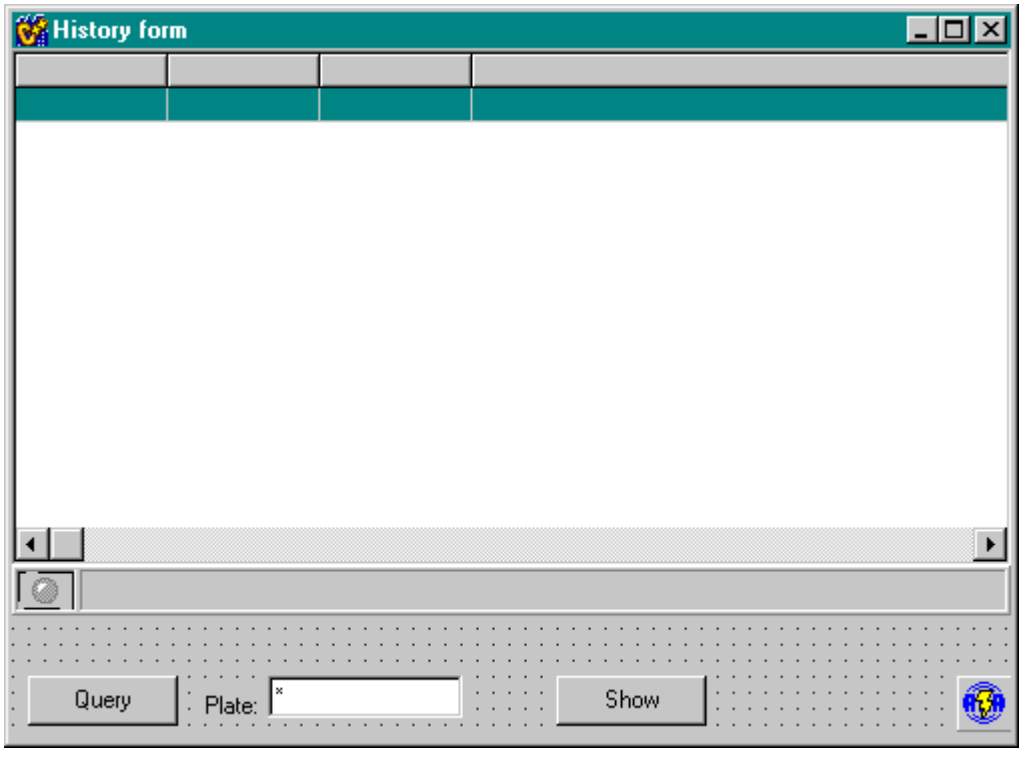
Now start your application and enter a plate number or a piece of a plate number then click query. The control creates a query for every defined service and retrieves the results. Your display now looks like this:

- The first column show the license plate
- The second column shows the type (verbal description) of the service which created this result
- The third column shows date and time of creation
- The last column shows the brief result

All results have been retrieved and are stored as files on your hard disk (in the TEMP directory). You have direct access to the IAwnSend objects and the files using the Results[] or ResultFiles[] properties.



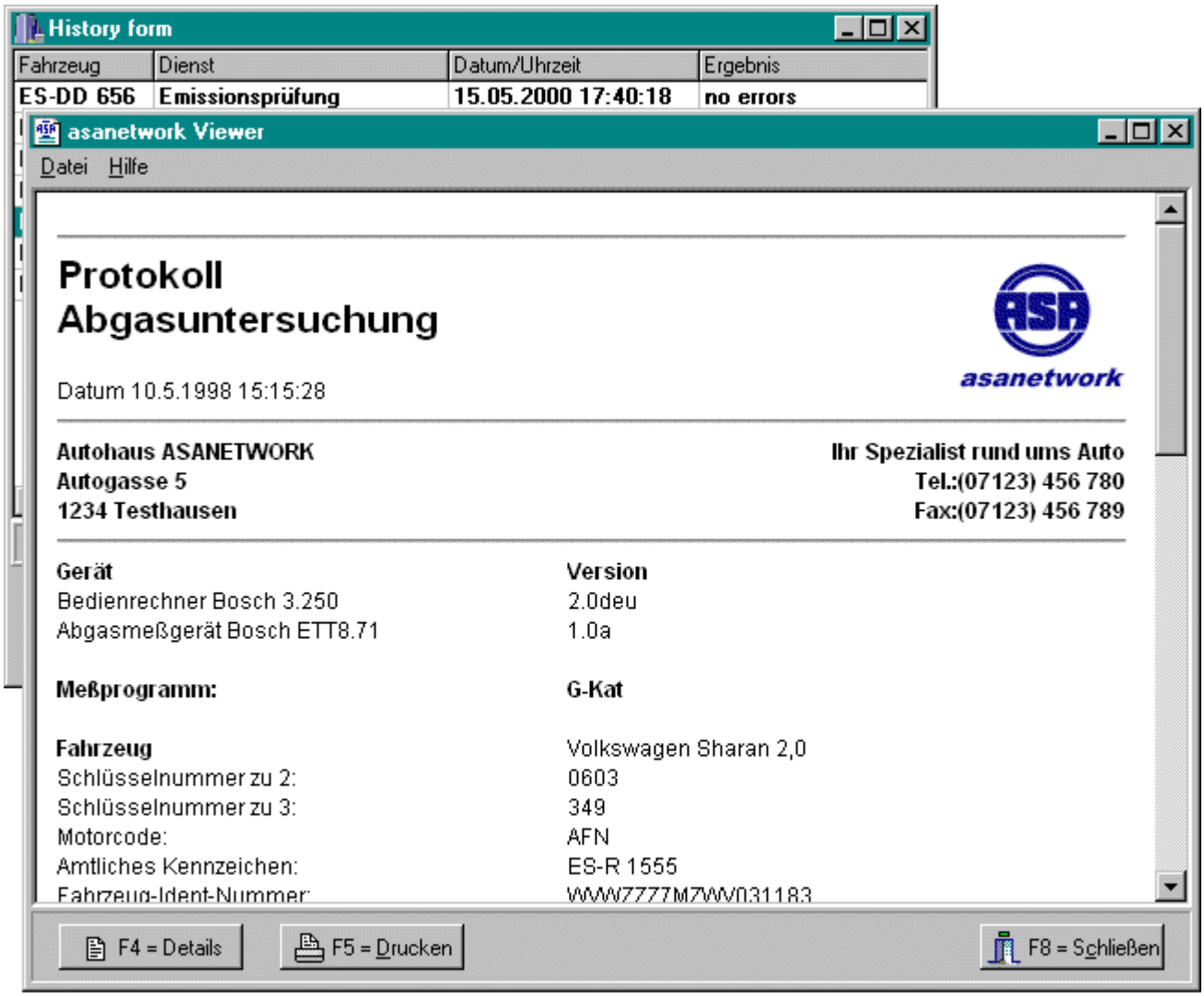
Now let's add a show button to display the selected result:



We use the DispIndex property again to get the index of the focused item. Then we directly call the asanetwork viewer with the file name of the selected result:

```
//---------------------------------------------------------------------------
void __fastcall THistoryForm::ShowClick(TObject *Sender)
{
    int Index = AwnControl1->DispIndex;
    String CmdLine = "c:\\programme\\axonet software gmbh\\awnview\\awnview.exe ";

    CmdLine += AwnControl1->ResultFiles[Index];

    WinExec( CmdLine.c_str(), SW_NORMAL);
}
//---------------------------------------------------------------------------
```

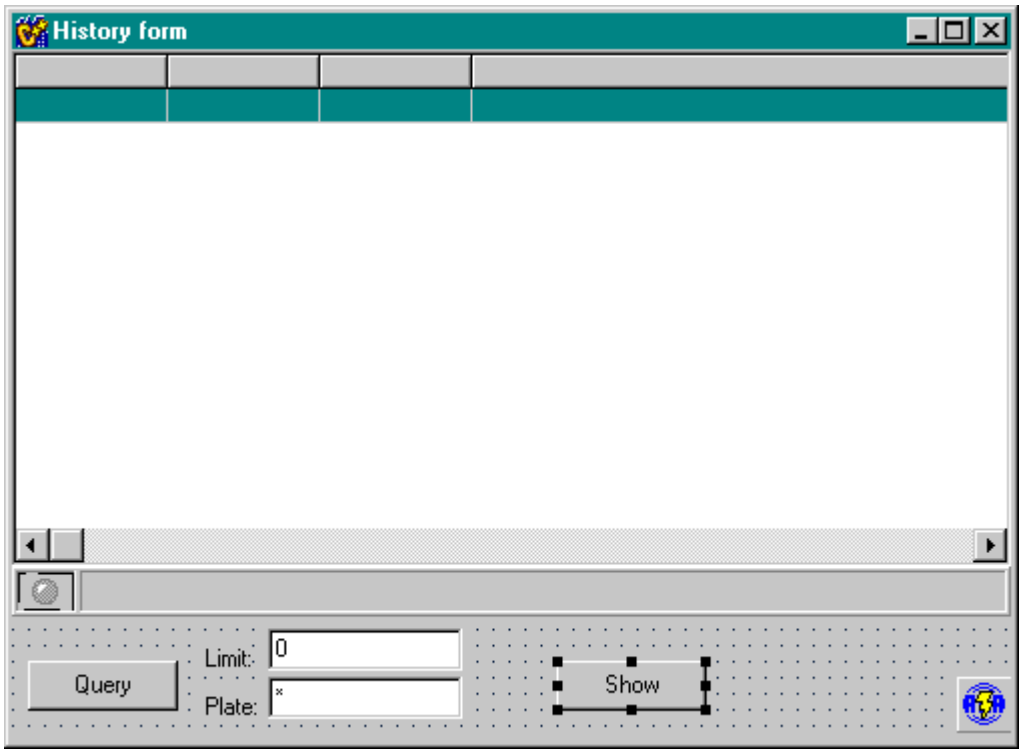
Start you application again, perform a query, select an item and click the "Show" button. The viewer starts up and displays the results:



Step 7, Adding a time limit

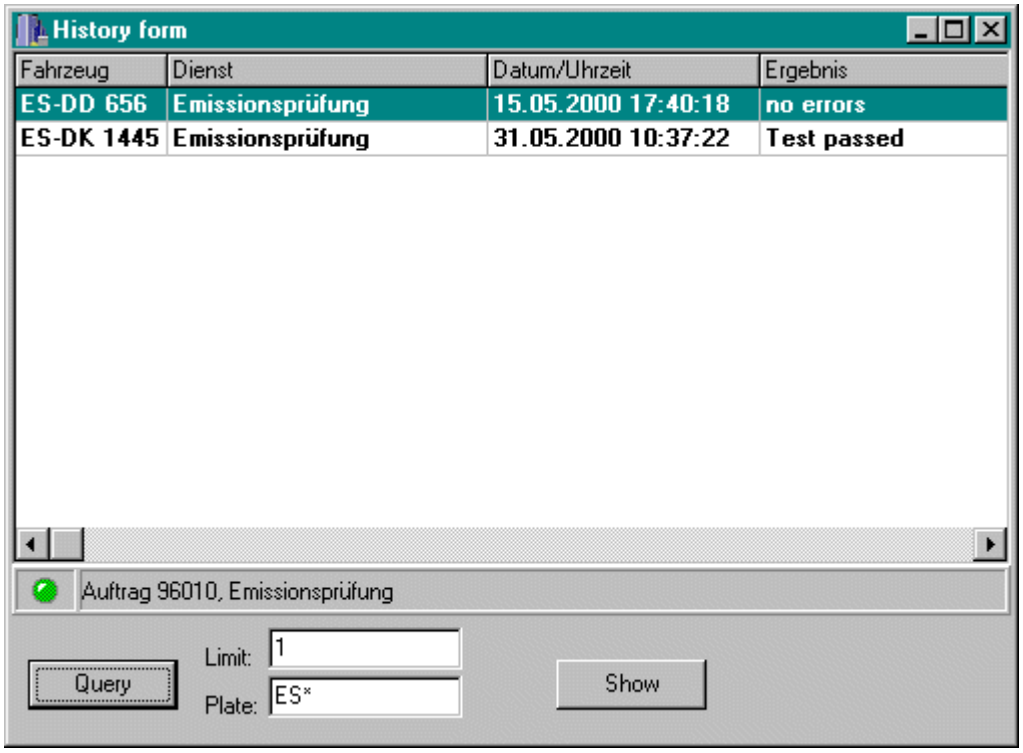
In this step we add the possibility to limit the query for a given period of time.

Add a TEdit for the input of the time limit. The time limit works in units of months. If you enter e.g. 1, you'll only receive results that are up to 1 month old. if you enter 12 you'll receive results of the last year.

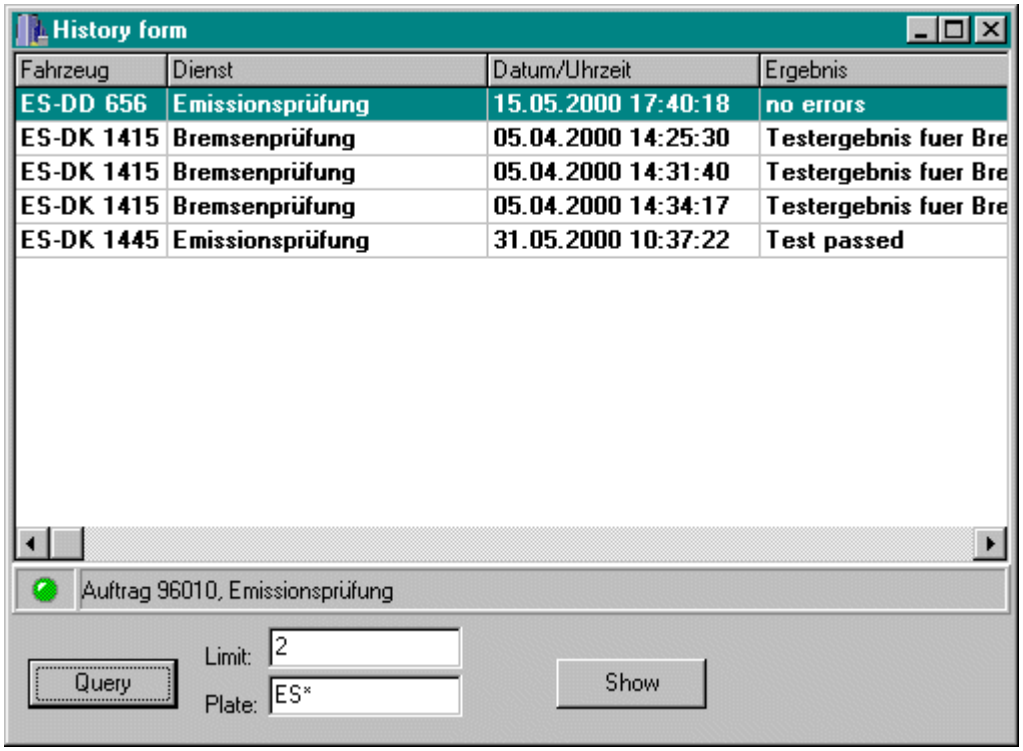



```
//---------------------------------------------------------------------------
void __fastcall THistoryForm::QueryClick(TObject *Sender)
{
    int Limit = StrToInt(Edit2->Text);
    Label2->Caption = "running";
    AwnControl1->QueryResult( "", Edit1->Text, Limit);
}
//---------------------------------------------------------------------------
```

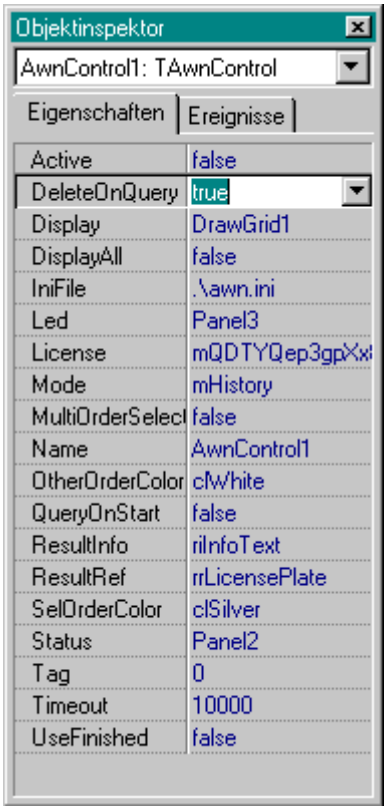
First example, limit = 1 month (compare this to the previously unlimited picture).



Second example, limit = 2 months (compare this to the previously unlimited picture).



Step 8, More properties



Again there's the DeleteOnQuery property. If false, you can accumulate different queries, if true the lists are deleted first. Try it.

Another property used with queries is the Timeout property. If you start a query the control waits for a response from the network manager. If there is no response in a given time (= Timeout value) the control aborts the query.

While the query is running every received item creates an OnQuery event. If the query is finished (either the last item was received or there was an timeout) an OnQueryEnd event is created.

We'll use these events to create a blinking panel while a query is running together with a display of the number of received items.

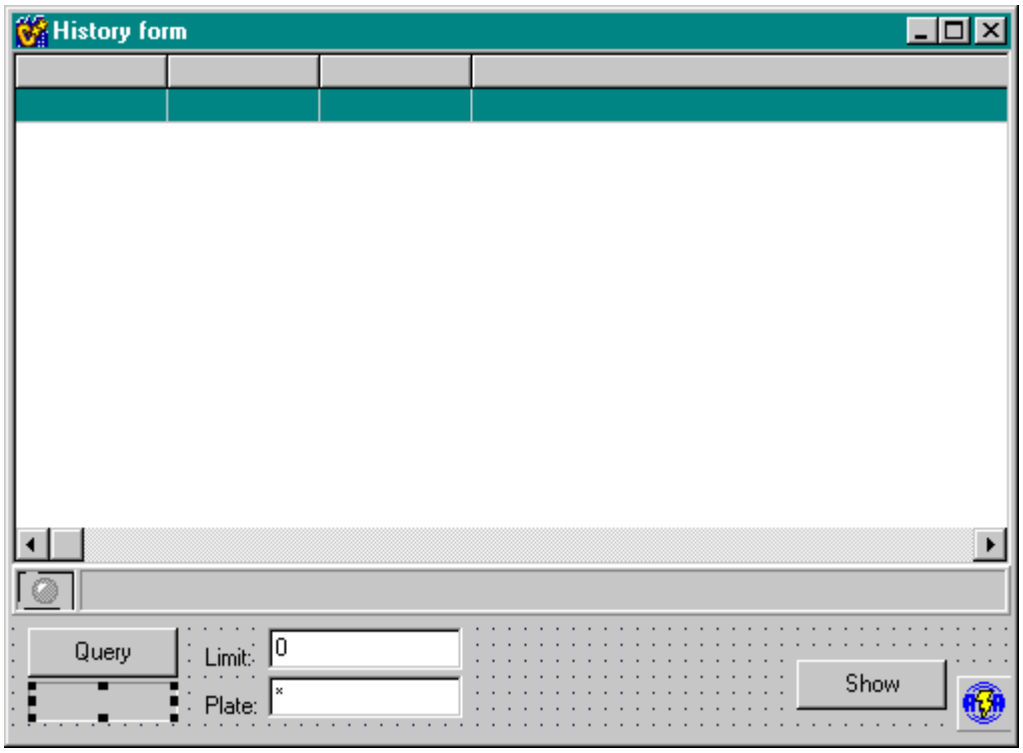
Add a TPanel and place a TLabel inside.

On start of our query we display the word "running".

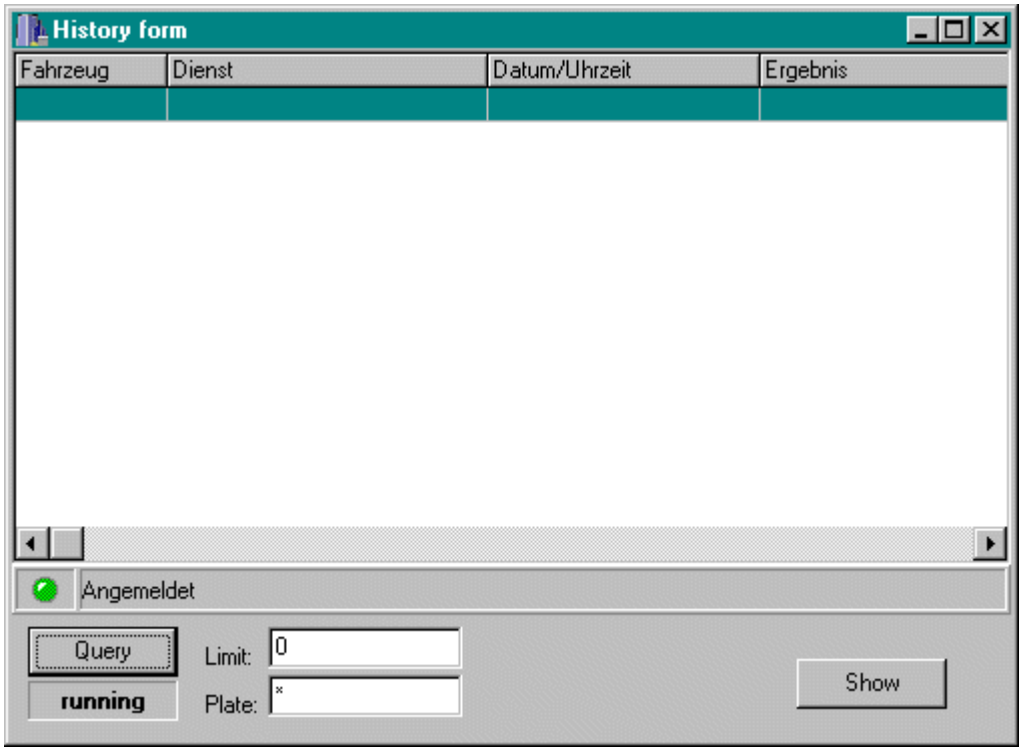
On every OnQuery event we'll toggle the background color of the label between blue and the standard button face color. The label caption displays the word running and the number of received items.

If we receive an OnQueryEnd event, we'll reset the background color and display the total number of received items.

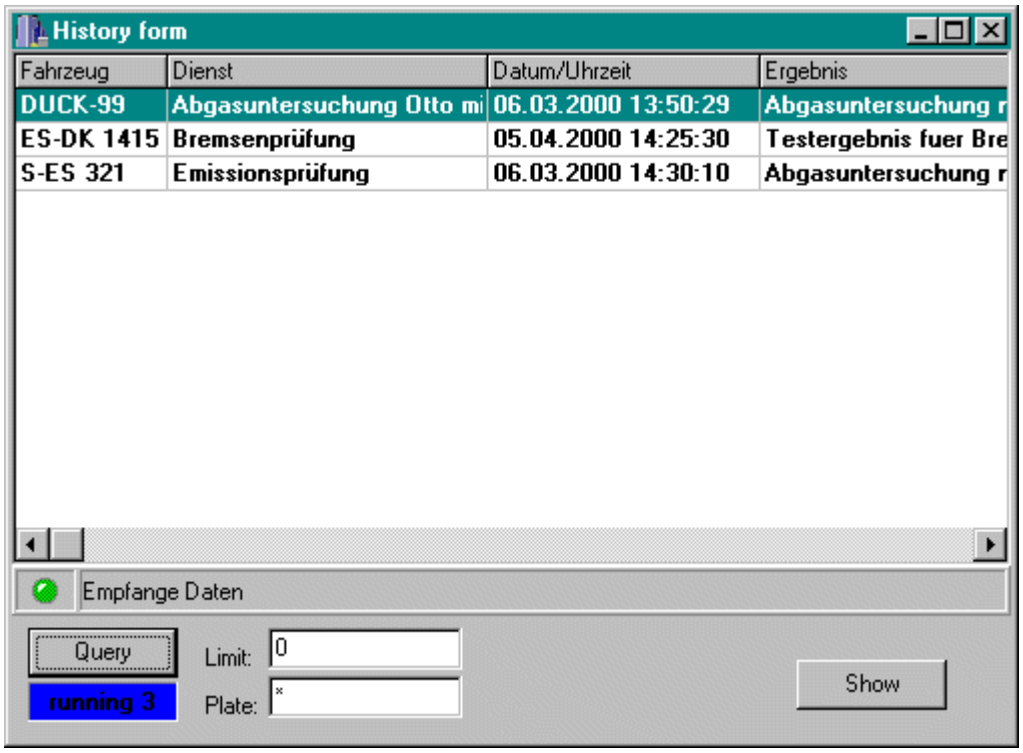
```
//---------------------------------------------------------------------------
void __fastcall THistoryForm::QueryClick(TObject *Sender)
{
    int Limit = StrToInt(Edit2->Text);
    Label2->Caption = "running";
    AwnControl1->QueryResult( "", Edit1->Text, Limit);
}
//---------------------------------------------------------------------------
void __fastcall THistoryForm::AwnControl1Query(TObject *Sender, int count)
{
    if (Label2->Color == clBlue)
        Label2->Color = clBtnFace;
    else
        Label2->Color = clBlue;
    Label2->Caption = "running "+ IntToStr(count);
}
//---------------------------------------------------------------------------
void __fastcall THistoryForm::AwnControl1QueryEnd(TObject *Sender,
      int count)
{
    Label2->Caption = "finished "+IntToStr(count);
    Label2->Color = clBtnFace;
}
//---------------------------------------------------------------------------
```



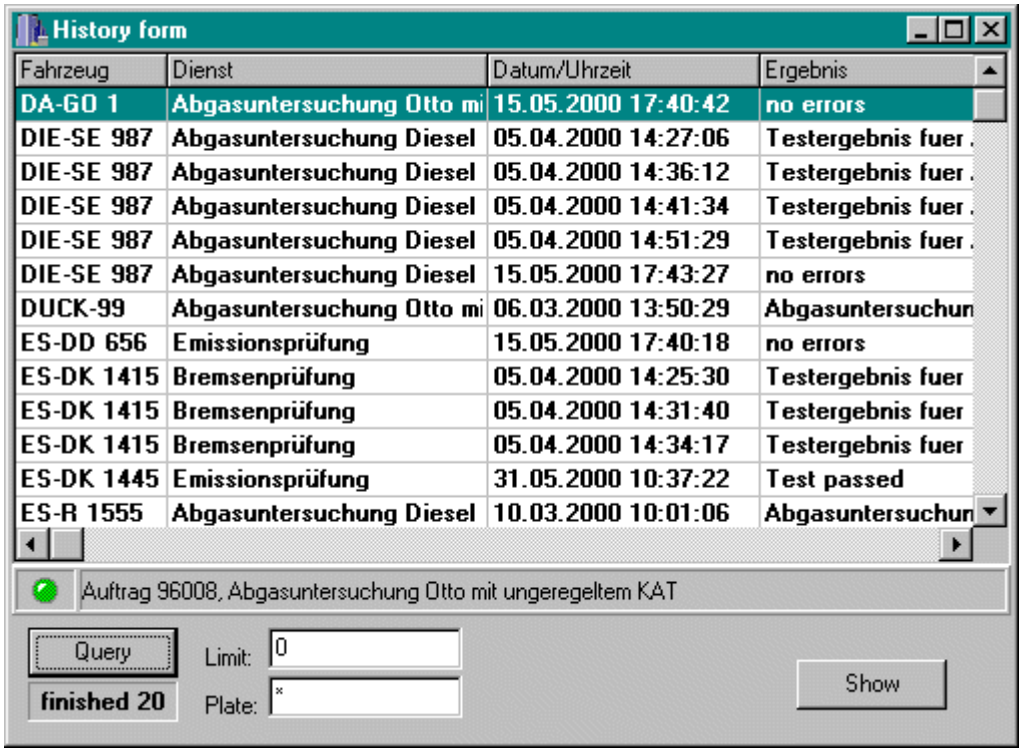
Query started



Three items received:



Query finished with 20 items:



Topics not covered

There are some additional properties and methods not mentioned in this introduction. Please have a look at the online help file for additional information:

Methods (items marked with * are for experienced users only)

StoreResult – save results without an associated order
SearchOrder – search the list for a specific order and return the index
* InsertSend – create a new position for a given order
HasVehicle – returns true if order has extended vehicle data available
GetVehicle – return extended vehicle data object

Properties

OrderCount – return number of orders in the Orders[] list
ResultCount – return number of results in the Results[] and ResultFiles[] list
* MultiOrderSelect – if true allow more than one active order, you have to store the order identification yourself and need to call SearchOrder
UseFinished – if false (default) you can't start an already finished order again
ResultRef – determines the reference used for history queries: rrLicensePlate, rrVehicleIdentification or rrOrder
ShowVehicle – if true displays a small car for any order with extended vehicle data

