Surviving Client/Server: Credit Payments, Part 2

by Steve Troxell

Last month we started looking at a credit payment processing system with a few twists. I'll briefly recap the requirements of the system. A customer could have one or more outstanding credits. A single payment could be applied to any one, some, or all of the customer's outstanding credits. For example, a $3500 payment is recorded as a single payment but could be applied to pay off a $1000 credit and a $2500 credit.

Further, a payment can be composed of different types of currency: cash, personal checks, money order, cashier's check, etc. Each element of the payment must be distinguished in the database and any portion of these elements may be freely disbursed across any of the credits being paid off. Continuing with the example above, a $3500 payment consists of $900 in cash, $2100 in the form of one personal check and $500 in the form of one cashier's check. The payment is applied to the two credits as follows: the $1000 credit is paid off with $500 in cash and $500 from the personal check; the $2500 credit is paid off with the remaining $400 in cash, $1600 from the personal check and the $500 cashier's check. All of this must be tracked and all of this is considered a single payment.

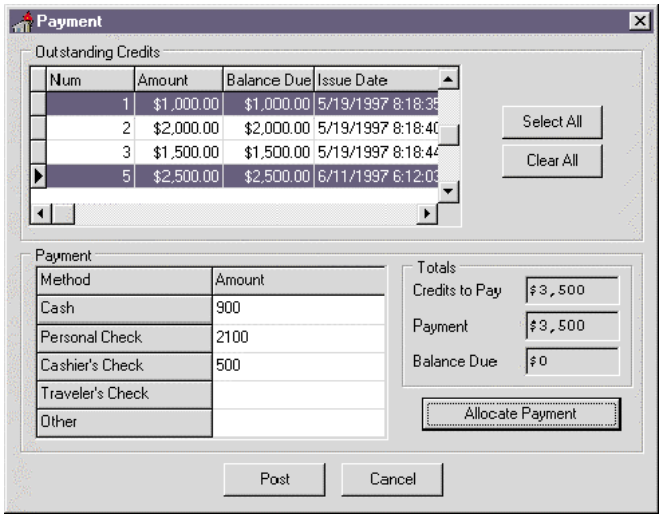
Last time, we discussed ways to handle the issuance of the credits and handling the different payment methods (check, cash, etc) in the payment screen. This month we'll complete the payment processing by covering the two-dimensional allocation of the payment across multiple methods and multiple credits.

The main payment screen is shown in Figure 1. The user selects one or more credits to apply the payment to from the grid at the top. The totals of the various methods of payment are entered in the grid at the bottom.

Now the challenge becomes how to distribute the payment given in the lower grid across the selected credits in the upper grid. For that we click the `Allocate Payment` button and reveal the screen shown in Figure 2. The upper grid is read only and updates automatically. It shows the breakdown of the payment by method and how much remains in each payment method "bucket" after allocating payment to the credits shown in the lower grid. The lower grid lists each of the credits to which this payment is applied and allows the user to enter the portion of each payment method "bucket" to disburse to each credit.

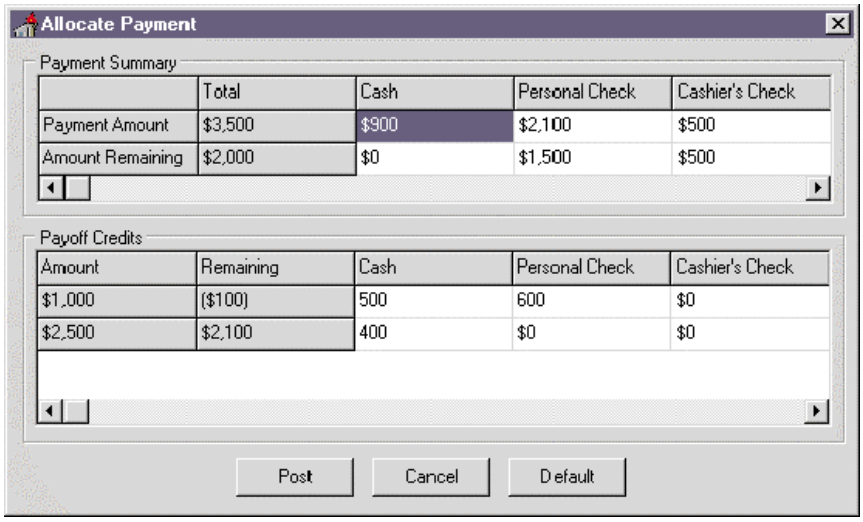
As amounts are entered in the lower grid, the `Amount Remaining` entries for the payment methods

➤ *Figure 1*

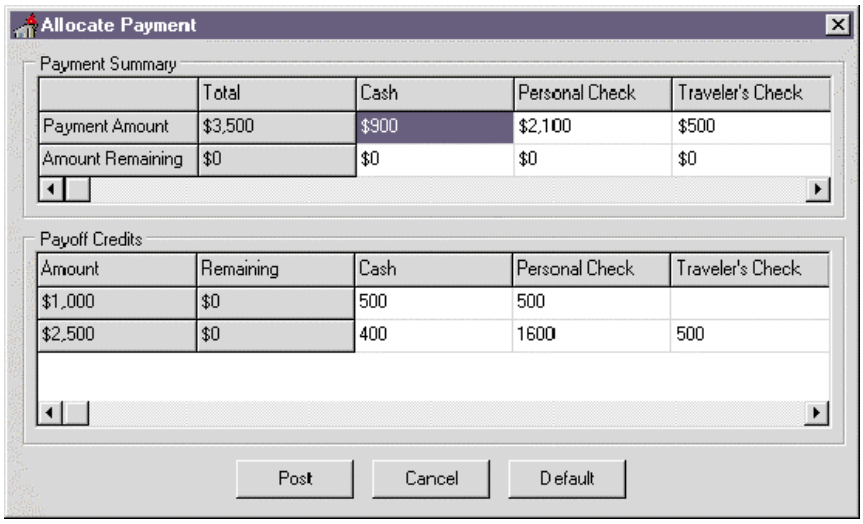


➤ *Listing 1*

```
create table Credits
(
  CreditNo        integer not null primary key,
  Status          char(1) default 'V' not null,
  CustNo          integer not null,
  Amount          float default 0 not null,
  BalanceDue      float default 0 not null,
  IssueDateTime   date default 'now' not null
);
create table Payments
(
  PaymentNo       integer not null primary key,
  CustNo          integer not null,
  Amount          float default 0 not null,
  PaymentDateTime date default 'now' not null
);
create table PaymentCredits
(
  PaymentNo       integer not null,
  CreditNo        integer not null,
  Amount          float not null,
  BalanceDue      float not null,
  primary key (PaymentNo, CreditNo)
);
create table PaymentAllocation
(
  PaymentNo       integer not null,
  CreditNo        integer not null,
  PayMethodCode   char(2) not null,
  Amount          float not null,
  primary key (PaymentNo, CreditNo, PayMethodCode)
);
```



➤ *Figure 2*



➤ *Figure 3*

```
PAYMENTS:
  PAYMENTNO       AMOUNT
============ ============
          1         3500

PAYMENTCREDITS:
  PAYMENTNO   CREDITNO       AMOUNT    BALANCEDUE
============ ========== ============ ============
          1          6         1000            0
          1          5         2500            0

PAYMENTALLOCATION:
  PAYMENTNO   CREDITNO PAYMETHODCODE       AMOUNT
============ ========== ============= ==========
          1          6 CS                    500
          1          6 CK                    500
          1          5 CS                    400
          1          5 CK                   1600
          1          5 TC                    500
```

➤ *Figure 4*

and the credits are automatically updated. This screen has been partially filled out. In a real system, errors would be generated if the user tried to post this screen and the entire payment was not allocated out to all the credits or any of the `Amount Remaining` values went negative. For example, in Figure 2 the user has over allocated the payment towards the $1000 credit.

To support this concept on the backend, the tables that record the payment information are a master-detail-detail configuration. One payment record is linked to one or more credits to be paid, and each of these is linked to one or more payment amounts, one for each payment method applied to that credit. The table definitions are shown in Listing 1 (I'll repeat the `Credits` table from last month for completeness).

By way of illustration, Figures 3 and 4 show the arrangement of data for the example given at the beginning of this article.

Now you may think that the amount of the payment is stored redundantly but this was done to facilitate reporting of this information. Summing the amounts from `PaymentAllocation` to generate the values we've stored in the other tables would get awkward in SQL, so we pre-calculate them and make a little redundancy to simplify the reporting end of the system.

The `BalanceDue` in the `PaymentCredits` table can be thought of as redundant with the `BalanceDue` in the `Credits` table. Keep in mind that the one in `Credits` always reflects the current balance regardless of how many payments have been made. The one in `PaymentCredits` always reflects the balance at the time that particular payment was made. This was necessary due to a reporting requirement of the system. If we joined to `Credits` to get the balance each time, it would be inaccurate as additional payments are made against the credit.

As you can see from the specification of the user interface and the layout of the payment tables, it would be a terrible idea to have the payment amounts post directly into the database tables via data-aware controls. That would produce a lot of unnecessary network traffic and database activity, and it would be extremely difficult to map the data fields in the dialogs directly to the table rows and columns. Besides, all the manipulation done in the Payment Allocation dialog is worthless to the database until the user has returned to the main payment dialog and posted the payment.

It is more effective to store all the data for the payment dialogs in

Delphi until the user is done, then translate the internal data into something suitable for the database. In the actual application, we used an Orpheus grid, which has no inherent storage capacity of its own. So I built a fairly elaborate class structure to contain all the payment information. Here, we're using TStringGrids which do hold their own data, so we don't need to go to great lengths to describe a fancy external class structure. It doesn't matter how the user information is stored prior to posting to the database, the point is that we do hold it in the Delphi app and programmatically post it to the database.

One big advantage to using a data structure to hold the payment separate from the user-interface controls is that it is much easier to provide a "cancel" operation for the Payment Allocation dialog and abandon any changes the user may have made to the payment allocation. But we're going to stick with our TStringGrids for now. We need to keep track of which credits were selected for payment, the payment amounts broken down by category, and the distribution of all the payment methods across each of the selected credits.

All our payment data is stored within the two grids in the Payment Allocation dialog (Figure 2) except for the ID numbers of the selected credits. For that we simply use a TList in the allocation dialog and use its pointer fields to hold the ID numbers of the credits we've selected. If we've created an instance of TList called Credits, then we would add the credit ID number like this:

```
Credits.Add(Pointer(aCreditID));
```

typecasting the LongInt ID number into a pointer. Now, our list simply holds a list of integer numbers rather than a list of pointers to allocated memory.

OnSetEditText

With one exception, I won't go into details about how the data is stored internally in the grids and transferred between the two dialogs, our concern here is getting it into the database. The full source code is available on the disk.

The one thing I would like to point out is how the Amount Remaining values are made to automatically update as the user keys in data. For this, we use the OnSetEditText event handler for TStringGrid. Each time the data in a cell changes, character by character, the OnSetEditText event handler fires and gives you an opportunity to do something with the cell data just changed. The parameters of OnSetEditText tell us which cell changed as well as its new value. Knowing this we can simply recalculate the appropriate totals for that row and column in the grids.

This same technique is used on the Main Payment Dialog (Figure 1). Whenever the user changes the payment breakdown in the lower grid, we must change the Totals display on the right. Again, we use OnSetEditText to tell us when any grid data has changed, but in this case we really aren't concerned with which cell has changed. We simply recalculate the total payment and change the display as shown in Listing 2.

Posting The Payment Data

To get the payment data from the two dialogs into the data tables requires four steps.

Firstly, add a single record in the Payments table noting the total amount of the payment and generating a payment ID number. All the data associated with a single payment will be marked with the same payment ID number.

Secondly, for each credit selected to pay, reduce the balance due on the credit in the Credits table by the amount of the payment.

➤ *Listing 2*

```
procedure TfrmPayment.grdPaymentSetEditText(Sender: TObject; ACol, ARow: Longint;
  const Value: string);
var
  I: Integer;
begin
  { Update total payment amount }
  TotalPaid := 0;  { form variable }
  with grdPayment do begin
    for I := 1 to RowCount - 1 do
      if Cells[1, I] <> '' then
        TotalPaid := TotalPaid + StrToFloat(Cells[1, I]);
  end;
  UpdateTotals;  { local method to update the screen }
end;
```

➤ *Listing 3*

```
create procedure PaymentSave(iCustNo integer, iAmount integer)
  returns (oPaymentNo integer)
as begin
  oPaymentNo = gen_id(Gen_PaymentNo, 1);
  insert into Payments (PaymentNo, CustNo, Amount)
    values (:oPaymentNo, :iCustNo, :iAmount);
end
```

➤ *Listing 4*

```
create procedure PaymentCreditSave(iPaymentNo integer,
  iCreditNo integer, iAmount integer)
as
  declare variable NewBalance float;
begin
  select BalanceDue from Credits
    where CreditNo = :iCreditNo
    into :NewBalance;
  NewBalance = NewBalance - :iAmount;
  update Credits
    set BalanceDue = :NewBalance
    where CreditNo = :iCreditNo;
  insert into PaymentCredits (PaymentNo, CreditNo,
      Amount, BalanceDue)
    values (:iPaymentNo, :iCreditNo, :iAmount, :NewBalance);
end
```

```
procedure TfrmPayment.btnPostClick(Sender: TObject);
var
  PaymentNo: LongInt;
  Amount,
  TotalPaidThisCredit: LongInt;
  C, P: Integer;
begin
  with dmDataModule.dbDemo do begin
    StartTransaction;
    try
      { post the main payment record }
      with dmDataModule.spPaymentSave do begin
        ParamByName('iCustNo').AsInteger := CustomerNo;
        {note TotalPaid was calculated in Listing 2}
        ParamByName('iAmount').AsFloat := TotalPaid;
        ExecProc;
        PaymentNo :=
          ParamByName('oPaymentNo').AsInteger;
      end;
      { Post the payment amounts at finest granularity }
      with frmPaymentAllocation do begin
        with grdCredits do begin
          { for each credit selected to pay }
          for C := FixedRows to RowCount - 1 do begin
            TotalPaidThisCredit := 0;
            { for each payment method for that credit }
            for P := FixedCols to ColCount - 1 do begin
              Amount := GetCellAmount(Cells[P, C]);
              if Amount <> 0 then
                with dmDataModule.qryPaymentAllocSave do
                begin
                  ParamByName('PaymentNo').AsInteger :=
                    PaymentNo;
                  ParamByName('CreditNo').AsInteger :=
                    LongInt(Credits[C - FixedRows]);
                  ParamByName('PayMethodCode').AsString :=
                    PChar(dmDataModule.
                      PaymentMethodsList.Objects[
                      P - FixedCols]);
                  ParamByName('Amount').AsFloat :=
                    Amount;
                  Inc(TotalPaidThisCredit, Amount);
                  ExecSQL;
                end;
              end;
              if TotalPaidThisCredit <> 0 then
                with dmDataModule.spPaymentCreditSave do
                begin
                  ParamByName('iPaymentNo').AsInteger :=
                    PaymentNo;
                  ParamByName('iCreditNo').AsInteger :=
                    LongInt(Credits[C - FixedRows]);
                  ParamByName('iAmount').AsFloat :=
                    TotalPaidThisCredit;
                  ExecProc;
                end;
            end;
          end;
        end;
        Commit;
      except
        Rollback;
        raise;
      end;
    end;
end;
```

➤ *Listing 5*

Thirdly, for each credit selected to pay, add a single record into the `PaymentCredits` table, noting the payment ID, the credit ID, the total amount of payment applied to that credit and the balance remaining on the credit.

Lastly.for each different breakdown of payment (cash, check, etc) against a single credit, add a record to the `PaymentAllocation` table noting the payment ID, the credit ID, the code identifying the payment method, and the amount of that portion of the payment.

All of these steps should be wrapped up within a single transaction to ensure everything is posted consistently.

The first step is handled by the stored procedure shown in Listing 3. We use an InterBase generator to produce the unique payment ID and return that to the application so we can bind all the other pieces of information under the same ID.

Steps 2 and 3 involve each credit that is being paid and are handled by the stored procedure shown in Listing 4.

Step 4 is a simple insert into the `PaymentAllocation` table and is handled by a straight SQL query in the application. Listing 5 shows how we pull all this together. Notice that we actually add the `PaymentAllocation` records before the `PaymentCredits` record in order to give us the opportunity to sum up the payment total for the credit.

Conclusion

Well, that wraps things up for my little tour around our credit payment system. My hope is that you've picked up a few odds and ends for the reasoning of some of the choices made here. In next month's column I'll begin to look at the new database access component hierarchy in Delphi 3 and how we may derive custom dataset classes.

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@turbopower.com or on CompuServe at STroxell.