The Record Read and Record Process Sequence

This example works through a load_from_csv sequence. The load_from_csv function takes a parameter $p with these properties that deal with the processing of input records:

```
$p.file_field_process
$p.file_field_lookup
$p.file_add_fields
$p.file_not_null_fields
```

All of the above could be missing in $p (therefore regarded as null), or specifically set as null. These properties will be described in detail when they are relevant to the insert/load sequence. But first we need to deal with the field names that will be assigned to the input records read.

Getting the Field Names:
The following properties deal with assigning a field name to an ordinal position to the input value sequence.

```
$p.file_fields
```

If given (is not null or not missing and not null) then this list is the field names sequence. In which case the following properties should be omitted or set to null.

```
$p.field_line
$p.file_title_lines
$p.skip_to_field_line
```

If file_fields is null or omitted then the above properties should be set:

$p.field_line is true, says there is a line in the input which defines the field names in ordinal position, and it can be resolved using one-of these two methods – either:
$p.file_title_lines is the number of lines that should be skipped before the field line is found, and after these skips the next line is expected to be the field line, or
$p.skip_to_field_line is the pattern to expect that indicates the field line.

The input field line must be assigned before the data input sequence can begin. That is, the field line must have been explicitly given (by file_fields) or resolved using file_title_lines or skip_to_field_line.

The reading of data lines and insertion into the DB then begins with this sequence, and note that each line/record is dealt with one at a time; that is, a data line is read and then inserted into the DB before the next data line is read.

For this example let us assume that the field line (resolved or explicit) is:

```
$field_line.1 = “Customer Ref”;
$field_line.2 = “Withdrawn Amount”;
$field_line.3 = “Balance”;
```

Therefore, for this example we expect data lines to have three fields, let us say that a typical data line looks like this:

```
“CUST_001”,”£10.50”;”£101.40”
```

A Data Line is Read and Assigned to Field Names:
A data line is read, each value in the line is assigned, in ordinal sequence, to a structure indexed by the property given by field_line (resolved or explicit). Let us assume that the structure has been named $record, in which case the structure will look like this:

```
$record.”Customer Ref” = “CUST_001”;
$record.”Withdrawn Amount” = “£10.50”;
$record.”Balance” = “£101.40”;
```

Record Values are Processed:
The record is processed by the property file_field_process. See above, this is one of the properties that deal with processing data. If it is present in the structure p and is not null it will have the following format (for this example):

```
$file_field_process.”Withdrawn Amount” = “NUMBER”;
$file_field_process.”Balance” = “NUMBER”;
$p.file_field_process = $file_field_process;
```

This says, for all field properties in the structure file_field_process lookup that field in the data input record and process it in the way specified by the value of file_field_process. In the example, the value in the input record for “Withdrawn Amount” will be processed as a NUMBER – that is the currency symbol will be stripped off, as will the value for “Balance” in the data input record.

There is one extension to this functionality concerning the structure file_field_lookup. If this structure is provided (that is, not null) then it is the ‘mapping’ from the input file fields to the target DB field names. It is of the form:

```
$file_field_lookup.”<source field name>” = “<DB field name>”
```

So in our example we might have the following structures set up for file_field_lookup:

```
$file_field_lookup.”Withdrawn Amount” = “DebtAmount”;
$file_field_lookup.”Customer Ref” = “C_ref”;
```

```
$file_field_lookup.”Balance” = “Balance”;
$p.file_field_lookup = $file_field_lookup;
```

So, if file_field_lookup exists and it has a property “Withdrawn Amount” then it is allowed that file_field_process can have properties in the name of the target field name. So that, in this case, both of these setups for file_field_process will have the same result:

```
$file_field_process.”Withdrawn Amount” = “NUMBER”;
```

Would be exactly the same as saying:

```
$file_field_process.”DebtAmount” = “NUMBER”;
```

But for the latter to work there must be a:

```
 $file_field_lookup.”Withdrawn Amount” = “DebtAmount”;
```

See an additional important note regarding file_field_lookup later.

Each Field Value is Validated:
Each field in the structure then does through a validation stage.
Each field value causes the system validation function to be called, it will always use the target DB field name as a reference, so the global validation routines (as set up in the Globals dialog) will be passed the target DB name and the value for that field from the record structure. So in our example the following validations will be attempted:

```
Validate(“C_ref”, “CUST_001”)
Validate(“DebtAmount”, “10.50”)
Validate(“Balance”,”101.40”)
```

Points to note: the values have been processed (because of file_field_process) so the numbers have their currency symbols stripped.

Optionally Each Field Value is Custom Validated:
The validation stage continues if there is a function in the callers code called: file_custom_validate
If this function exists then it is called [for each field in the record structure], so that function will receive the following calls:

```
file_custom_validate(“C_ref”,”CUST_001”);
file_custom_validate(“DebtAmount”,”10.50”);
file_custom_validate(“Balance”,”101.40”);
```

That function is required to validate the value for the target DB field name. It returns a value to replace the given value (if the value is valid then it just returns the given value). If it is unable to validate or “correct” the value then it can cause the code to exit.

Optionally Deal with NULL Field Values:

If the property file_not_null_fields is set then it will have the structure like this:

```
$file_not_null_fields.Balance = "SKIP";
$file_not_null_fields."Withdraw Amount"="ABORT";
$p.file_not_null_fields = $file_not_null_fields;
```

The structures can have one of two values: SKIP or ABORT. If a field value is null and the property for that field in file_not_null_fields is SKIP then the entire record insert is skipped. If a field value is null and the property for that field in file_not_null_fields is ABORT then the total load is terminated with an error return back to the calling profile.

Optionally Adjust Field Values:
The process then calls the call-back function file_custom_fields. If the calling code has a function file_custom_fields then this function is called with the record structure. The custom function is required to return the record. It is the opportunity for the calling code to adjust any values in the record's fields. The record structure is passed with properties still named in terms of the source data (that is, still in terms of the field line) – not in terms of the target db field names.
So, the custom call-back file_custom_fields might look like this:

```
function file_custom_fields($record) begin
   // convert to euros
   $record.Balance = $record.Balance * #GBP_EURO_rate;
   return $record;
   end
```

Optionally Insert Record:
The process then calls the call-back function file_conditional_insert. If the calling code has a function file_conditional_insert then this function is called with the record structure. The function can return one of three values:
Returns TRUE – continue to insert this record as provided
Returns FALSE – skip the insertion of this record
Returns a record structure – continue to insert, but use the values in the returned record

The record structure is passed with properties (field names) termed as specified in the field line (that is, the source field names) not the target DB field names.

Field Names Mapped to Target DB Field Names:
If the record insert has not been skipped (by file_conditional_insert or file_not_null_fields) then the process continues to set up the record structure for insertion into the DB.

The field names in the record structure are mapped to those to be used by the target DB if the structure file_field_lookup is set. Again the file_field_lookup has the structure like this:

```
$file_field_lookup."Withdrawn Amount" = "DebtAmount";
$file_field_lookup."Customer Ref" = "C_ref";
$file_field_lookup."Balance" = "Balance";
$p.file_field_lookup = $file_field_lookup;
```

So, the record structure will now look like this:

```
$record.DebtAmount = "10.50";
$record.C_ref = "CUST_001";
$record.Balance = "132.24";
```

(Note the Balance field name is as the source, there is no mapping from source to target for this field – but – this is important – because the structure file_field_lookup exists then it must include all fields that are required to be inserted, also that the actual value has changed because of a conversion function provided in file_custom_fields).

Important note: if file_field_lookup exists then it must have a property for each field that is required to be inserted into the DB. If file_field_lookup is null or omitted from the $p structure then all fields (with their source field line names) will be inserted into the DB.

Optionally Add Additional Fields before Insert:
If the parameter structure has the structure file_add_fields then these fields with their values are added to the record structure. A file_add_fields structure might look like this:

```
$file_add_fields.Date = "2012-09-26";
$file_add_fields.Bank = "Raphaels";
$p.file_add_fields = $file_add_fields;
```

The record structure will now look like this:

```
$record.DebtAmount = "10.50";
$record.C_ref = "CUST_001";
$record.Balance = "132.24";
$record.Date = "2012-09-26";
$record.Bank = "Raphaels";
```

Important note:
Fields to be added as specified in the file_add_fields must be in terms of their target DB field name.

Important note: Fields to be added with values that are "fixed" – that is, set for the duration of all inserts for all records – like those in the example

above for Bank and Date should use the file_add_fields as shown. However, it is possible to add fields ‘on the fly’ in two places:

In file_custom_fields, the function is passed a record and returns a record, you can add a new field and value before the record is returned.
In file_conditional_insert, the function is passed a record and if the insert is to take place then it can return a record with a new field added (it must return the record in this case and not just return true).

In both cases the new field(s) that are added must appear in either the structure file_field_lookup or file_add_fields.

It is in this form that the insert query is created.