# IRIS Exchequer v7.0 – SQL Data Migration Notes

## Command Line Parameters

e.g.

ConvertToSQL.Exe /Threads:6 /EnableODSLogging /MaxMemMB:300 /Yahoo

### /Threads:XX

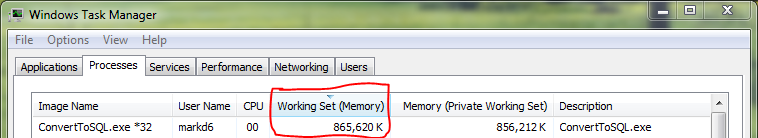
This option allows the default number of SQL Write Threads (3) to be overridden.

### /MaxMemMB:XX

This option allows the default memory usage (50% of available memory at the point the Data Copying starts) to be overridden with a required number of megabytes. If you specify less than 50 it will be ignored and 50Mb will be used, if you specify more than the system can handle it will probably crash with an Out of Memory exception.

Max : 1000

*Note: This is based on the Working Set which includes the memory used by the.Exe’s and DLL’s in memory as well, this is not shown by Task Manager as standard but can be added in via the View + Select Columns:-*

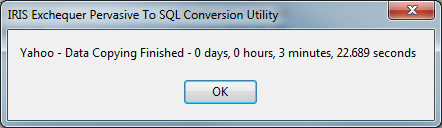


### /EnableODSLogging

This causes messages to be written to OutputDebugString for debug purposes – this should not be used unless instructed as writing the messages slows it down.

### /Yahoo

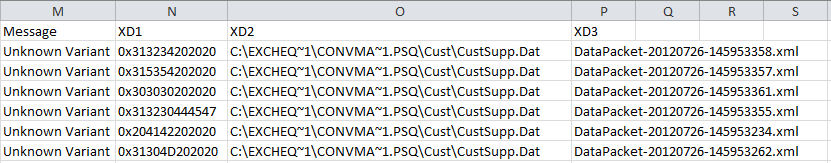
At the end of the Data Copying phase of the conversion it causes a message to be displayed with the elapsed time between the start and completion of the Data Copying process:-



## Logging

### SQLConvert.csv

Errors and Warnings are written to SQLConvert.csv in the \Logs\ folder off of the SQL Edition installation of IRIS Exchequer. This logging uses the same technology and format as the logging sub-system added into IRIS Exchequer v6.8 for the SQL Improvement Plan.

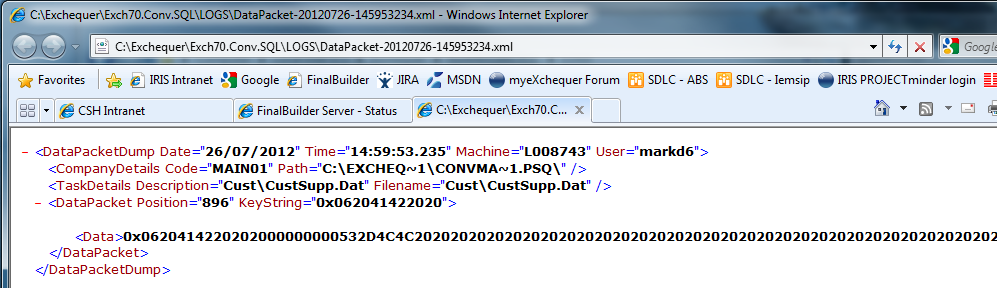


At this point the following types of warning are generated:-

|  |  |
| --- | --- |
| Unknown Variant | A record that cannot be unidentified was read from the Pervasive Database |
| SQL Execution Exception | An exception occurred whilst executing a SQL Statement |
| SQLExecutionError | An error was returned whilst executing a SQL Statement |

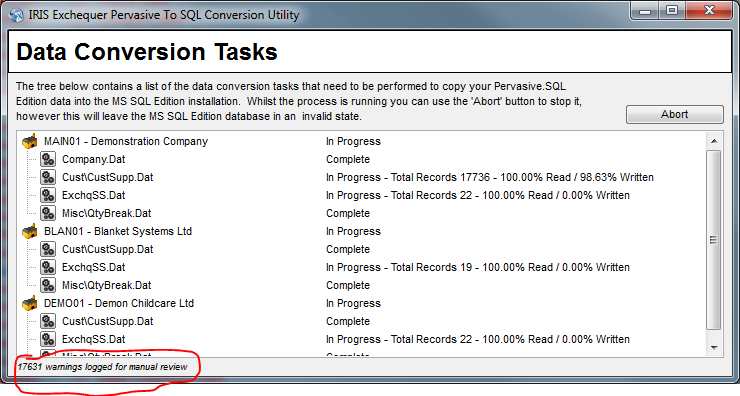
### Data Packet Dump Files

A Data Packet Dump File is an XML format file which contains the data being processed at the point the error/warning occurred. Hopefully having access to this data will make identifying issues easier and the Position and Keystring may help us find the issue in the source Pervasive Edition data.

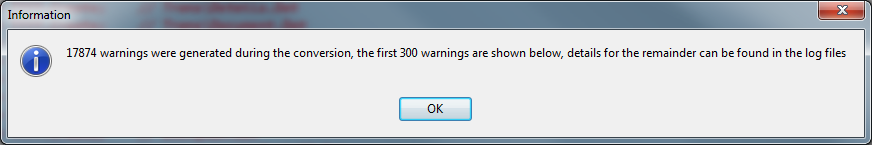


## Conversion Warnings

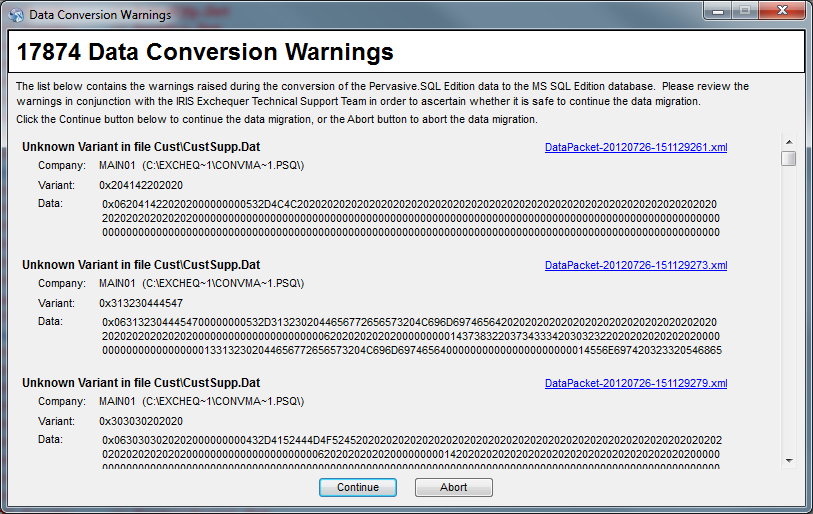
As the Data Copying phase progresses an ongoing count of the warnings is shown at the bottom:-



In order to limit memory usage only 300 warnings are actually held in memory for display, but all warnings are logged to file. In this situation the following message is displayed prior to the Data Conversion Warnings dialog:-



The Data Conversion Warnings dialog displays information about the warnings and a link to the Data Packet Dump File where applicable:-



All conversion warnings will be logged.

## Skipped Records

The following record types are ignored by the new SQL Data Migration:-

|  |  |  |
| --- | --- | --- |
| File | RecPFix/SubType | Reason |
| ExchqCk | H | DOS Help - not relevant to current Exchequer |
|  | EF | DOS Export - not relevant to current Exchequer |
| ExStkChk | DC/DQ/DS | Legacy Qty Break Records – Moved to QtyBreak.Dat in v6.10 |
| SentSys | All | This file is used by running Sentimail engines to register their capacities – if no records are found, an engine will add new records as required. Consequently there is no need to convert it. |

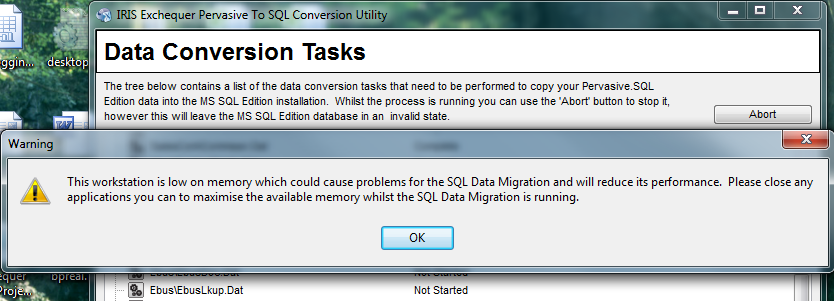
The following record types are not converted but will raise a warning if found by the new SQL Data Migration:-

|  |  |  |
| --- | --- | --- |
| **File** | **RecPFix/SubType** | **Reason** |
| ExStkChk | AL | Last Back To Back Order settings – these records are temporary records created whilst a B2B order is being created, shouldn’t exist in an Exchequer system being converted anyway |
| MLocStk | EN/EA | Dos Euro Converter Records - No Examples to test against, not relevent to current Exchequer as far as we know |

## Memory Monitoring (Build 103)

In order to reduce problems we have extended the data copying process to monitor available memory in Windows, as SQL Server is a bit of a greedy guts and tends to grab gigabytes of RAM whenever you aren’t looking which can cause the SQL Data Migration to crash with Out of Memory errors.

Basically every time the Read Thread tries to queue a record against a Write Thread it will check that there is 50Mb of Available Memory, if not it will pause the Read Thread and cause a message to be displayed by the Progress Tree window:-



*Note: The message locks the UI thread until the user clicks OK, so no updates to the progress will be seen.*

It is recommended that the user closes stuff down whilst the message is displayed, if they click OK straight away and then go to closedown other applications they may get multiple messages.

Whilst the Read Thread is paused the Write Threads will continue processing their queued items in the background which will free up memory.

# Programmer Notes

## How to Code Support for a New Table

1. In oConvertOptions.Pas extend the TDataConversionTasks enumeration to include an element for the new table.
2. In oDataConversionTask.Pas extend the case statement in CreateDataConversionTask to create a TDataConversionTask object for the new file with the appropriate configuration information.

E.g.

dmtLBin: Result := TDataConversionTask.Create(TaskId,

ttsAnyThread,

fpAllCompanies,

fosOptional,

'Trade\LBin.Dat',

'TradeCounter.zip');

The constructor takes the following parameters:-

*TaskId : TDataConversionTasks*

The enumeration value from TDataConversionTasks.

*ThreadStatus : TTaskThreadingStatus*

This parameter controls how the Btrieve Read Thread allocates the data across the SSQL Write Threads. Generally this will be set to *ttsAnyThread* unless there is a reason that the data needs to be written sequentially, for example the Trigger on the ExchqSS table causes errors to be raised if data is inserted into it from multiple threads at the same time.

*ttsAnyThread = 0*

The data will be added into whichever thread has the shortest queue.

*ttsSingleThread=1*

The data will be added into the first SQL Write Thread so the records are processed sequentially.

*PresenceStatus : TFileCompanyPresenceStatus*

This tells the scanning and processing routines about where the SQL Table will be – so it knows which connection string to use – and tells the Pervasive Installation scanning routines about where to expect the files to be found.

*fpRootOnly=0*

The SQL table is part of the Common Schema and will not be found in the Company schemas.

*fpAllCompanies=1*

The SQL table will be found in each Company schema.

*OptionalStatus : TFileOptionalStatus*

This tells the Pervasive Installation scanning routines whether the file is mandatory or optional:-

*fosMandatory=0*

*fosOptional=1*

*PervasiveFilename : ShortString*

This is the path of the Pervasive data file as a relative path from the company directory, e.g. ‘Cust\CustSupp.Dat’.

*ImportZIPFile : ShortString = ''*

This optional parameter defines the name of the SQL Emulator Import .ZIP file required to create the table. Generally this is required for optional modules and plug-ins.

1. Add a new Data Write Object for the data – see ‘How to Code a Data Write Object’ below.
2. In oBaseDataWrite.pas:-
   1. Add your new Data Write Object module into the Implementation section Uses clause.
   2. Extend the case statement in GetDataWriteObject to create and return an instance of your new Data Write Object for your new enumation value, e.g.

dmtLBin: Result := TLBinDataWrite.Create; // Trade\LBin.Dat

### How to Code a Data Write Object

A Data Write Object is a plug-in into the new SQL Data Migration process which has knowledge of a particular table and provides methods for generating a prepared SQL Insert Statement and for populating the parameters on that prepared SQL Insert Statement.

Data Write Object’s descend from TBaseDataWrite (oBaseDataWrite.pas) and implements four routines:-

*Constructor Create*

For non-variant files this creates an ADO Query for use with the data, for variant files this creates a set of sub-objects for converting the individual variants.

*Destructor Destroy;*

Destroys the ADO Query or sub-objects.

*Procedure Prepare (Const ADOConnection : TADOConnection; Const CompanyCode : ShortString);*

This is called once by the SQL Write Threads after the Data Write object has been created to allow the SQL Insert Statement to be constructed and prepared. For Variant files which routine will call the Prepare methods on the sub-objects used for the different variant records.

*Procedure SetQueryValues (Var ADOQuery : TADOQuery; Const DataPacket : TDataPacket;*

*Var SkipRecord : Boolean);*

This is called for each record/row by the SQL Write Threads to:-

1. Return the TADOQuery instance to be used for this data.
2. Populate the parameters on the SQL Insert Statement.

For variant files this routine will use the RecPFix/SubType to identify the type of data and call the SetQueryValues method on the appropriate variant sub-object. If the data cannot be identified from the RecPfix/SubType then a warning will be logged. In some cases we won’t bother processing the data, we will either set SkipRecord to TRUE to ignore it or raise an Unknown Variant warning – see Skipped Records earlier in this document for examples.

Variant files require a suite of sub-objects covering the different variant records. These sub-objects descend from TDataWrite\_BaseSubVariant and implement their own versions of the *Prepare* and *SetQueryValues* methods.

*To save time it is probably easiest to copy an existing module, for non-variant files I’d suggest oNominalDataWrite.Pas and for variant files oCompanyDataWrite.Pas.*

### Code Gen Tool – DWOCodeGen.Exe

A code generation tool has been knocked up to assist in the generation of the code, it can talk to the Exchequer database to import column information for a specific table (once you give it a valid connection string) and can parse a Delphi record structure in order to link the columns to fields in a record and generates the following code:-

1. The SQL Query using parameters for the values:-

sqlQuery := 'INSERT INTO [COMPANY].Nominal (' +

'glCode, ' +

'glName, ' +

<SNIP>

'glSpare, ' +

'glNomStr' +

') ' +

'VALUES (' +

':glCode, ' +

':glName, ' +

<SNIP>

':glSpare, ' +

':glNomStr' +

')';

*NOTE: We are using parameters in order to allow the Insert statement to be prepared which improves performance.*

1. A matching list parameter variables to be declared on the class:-

glCodeParam, glNameParam, glParentParam, glTypeParam, glPageParam,

glSubtotalParam, glTotalParam, glCarryFwdParam, glRevalueParam, glAltCodeParam,

glPrivateRecParam, glCurrencyParam, glForceJobCodeParam, glInactiveParam,

glClassParam, glSpareParam, glNomStrParam : TParameter;

1. A block of code to link the parameter variables to the parameter objects in the SQL Query:-

With FADOQuery.Parameters Do

Begin

glCodeParam := FindParam('glCode');

glNameParam := FindParam('glName');

<SNIP>

glSpareParam := FindParam('glSpare');

glNomStrParam := FindParam('glNomStr');

End; // With FADOQuery.Parameters

NOTE: Although the small horde of TParameter variables uses memory (4 bytes each) not having to find the parameter in the ADO Query for every row will improve performance.

1. A block of code to populate the parameter objects with values:-

glCodeParam.Value := NomCode; // SQL=int, Delphi=Longint

glNameParam.Value := Desc; // SQL=varchar, Delphi=String[40]

glParentParam.Value := Cat; // SQL=int, Delphi=LongInt

glTypeParam.Value := ConvertCharToSQLEmulatorVarChar(NomType); // SQL=varchar, Delphi=Char

glPageParam.Value := NomPage; // SQL=bit, Delphi=Boolean

<SNIP>

glClassParam.Value := NomClass; // SQL=int, Delphi=Byte

\*\*\* BINARY \*\*\* glSpareParam.Value := CreateVariantArray (@Spare, SizeOf(Spare));// SQL=varbinary, Delphi=Array[1..47] of Byte

glNomStrParam.Value := NomCodeStr; // SQL=varchar, Delphi=Str20

Note 1: Char’s have to be run through the ConvertCharToSQLEmulatorVarChar function as the SQL Emulator stores Char’s set to #0 as empty strings but ADO stores the #0, this function emulates the SQL Emulator behaviour for compatibility.

Note 2: Although the Code Gen Tool has a go at Binary fields they need to be confirmed manually, so it puts ‘\*\*\* BINARY \*\*\*’ at the start of the line so it won’t compile.

Note 3: Binary fields are supported by ADO through variant arrays, I have knocked up several functions in SQLConvertUtils.pas to assist in creating them:-

Function CreateVariantArray (Const StartAddr : Pointer;

Const ArrayLength : Integer) : Variant;

Returns a variant array of the specified length starting from the specified address.

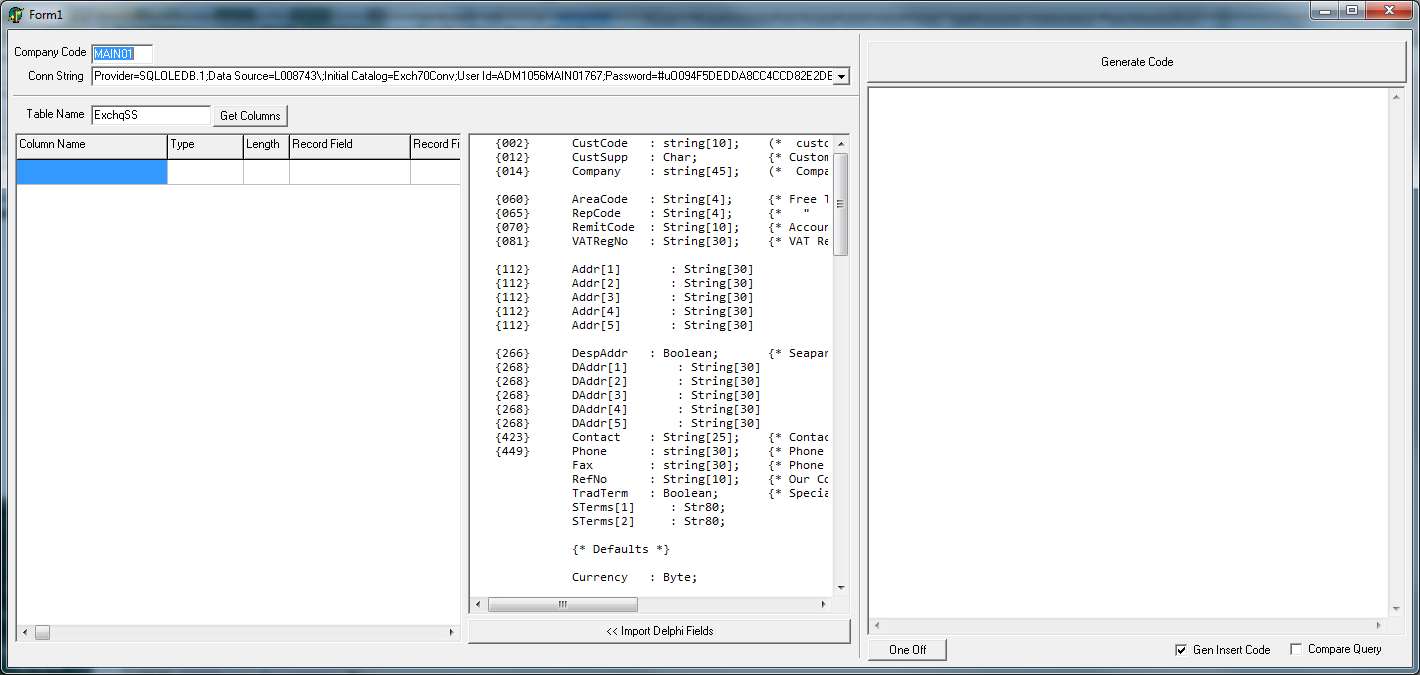
Function CalculateBinaryBlockSize (Const StartAddr : Pointer;

Const NextFieldAddr : Pointer) : Integer;

This can be used when binary fields in the SQL schemas cover multiple fields in the Delphi record structure. If cCalculates the size in bytes of a binary block between the address of two fields in a record structure, StartAddr is the address of the first field, NextFieldAddr is the address of the

next field AFTER the block. The resulting size can then be passed into CreateVariantArray in order to generate the variant array.

1. If the Compare Query check box is selected it will also generate a query that can be pasted into SQL Server Management Studio and tweaked in order to compare the contents of two tables. The query will need tweaking in order to match the rows correctly across the two tables/databases.



#### Step-by-step

1. Enter the correct Company Code – use Common for common schema tables
2. Enter the correct Connection String
3. Enter the table name, e.g. Nominal, in the Table Name field and click the Get Columns button.
4. For variant files you can use the ‘Delete Table Columns’ option on the popup menu to cut-down the list of columns to only those required for the data in question.
5. In the central memo paste in the Delphi record structure – the record structure must have any un-terminated curly-bracket comments removed and any sub-structures such as CCDepType must be expanded.
6. Click Import Delphi Fields to parse the record structure and import the fields into the list of columns. If there is a problem in the parsing it will tell you and if there is a mismatch between the number of columns it will tell you.
7. Check the SQL Columns and Delphi Fields match up – if not edit the record structure as required and goto 6).
8. Click the Generate Code button and update your code.

### Testing

Definitely worth doing at some point!

#### Limit the files being processed

You can modify TBtrieveReadThread.ReadBtrieveFile (oReadThread.Pas) to control which files are actually processed to reduce the time a test takes,.e.g.

If (oTask.dctTotalRecords > 0) Then

Begin

{ TODO : remove this! }

// For testing purposes limit the data being processed to reduce testing times

If (oTask.dctTaskId In [dmtNominal]) Then

Begin

// NOTE: Defaults to Index 0 so we don't need to set it

iRecCount := 0;