Miscellaneous Subjects

Error and Warning Handlers

The information provided here is meet as a supplement to the "Error and Warning Handlers" tutorial. If you have not studied this tutorial, you should do so before reading this additional information.

Monitoring of Jovis errors and warnings is an important and serious issue. You should never make presumptions about whether or not a particular command needs error checking. In fact, the omision of error checking should be considered poor programming style. You must NEVER pass over or ignore errors that occur. For example, you may get an error message telling you that you've run out of memory, and that the low memory reserves has been released. To ignore such a situation is asking for disastrous results!

If an error or warning occurs, Jovis will send a message starting at the card level, of the current card. The default name for the error message, appropriately, is "JovisErrorMsg", and the default name for the warning message is "JovisWarningMsg". You can install any warning or error message name that you want using the SetProperty command. (See the syntax reference for more information about the SetProperty command.) You can dynamically install a message name at any time. In this fashion, it is possible to have different error/warning handlers for different circumstances.

Several parameters are included with these messages to help you better understand both the meaning and level of importance of the errors and warnings. Jovis comes with a 'Str#' resource (ID #5301) that has a brief description of every error or warning that may occur. (They are also listing in an appendix in this manual.) When an error occurs, Jovis looks up the message and includes it as one of the parameters. The below handlers should serve as both an example and model for your own implementation(s).

Note: It is necessary that you use only the "JovisErrorCode" global for checking errors for 'CreateCollection', 'OpenCollection', 'DBInit' and 'DBOpen'. This is because initialization has not been completed.

Below are the "standard" error and warning handlers. You can always change them to better suit your requirements, such as clearer user related information. However, the actual messaging mechanism that Jovis uses is vital for informing you of errors and warnings.

-- The standard error and warning handlers:

```
function JovisErrorMsg theFile, theRel, theCmd, theMsg, ErrCode
  beep
  put "ERROR!" into temp
  put theMsg into line 3 of temp
  put "Command:"&&theCmd into line 5 of temp
  if theFile = "" then put "Unknown at this time." into theFile
  put "File:"&&theFile into line 6 of temp
  if theRel ≠ "" then put "Relation:"&&theRel into line 7 of temp
  if ErrCode ≠ "" then put "Code Number:"&&ErrCode into line 8 of temp
  answer temp with "Exit Script"
  get Jovis("SetProperty","myDB","AutoSave","true")
  if Jovis("IsTransactionOn","myDB") = "true" then
    get Jovis("CancelTransaction", "myDB")
    return "Stop"
  end if
end JovisErrorMsg


function JovisWarningMsg theFile, theRel, theCmd, theMsg, ErrCode
  global IsNetWorked
  put "WARNING!" into temp
  put theMsg into line 3 of temp
  put "Command:"&&theCmd into line 5 of temp
  if theFile = "" then put "Unknown at this time." into theFile
  put "File:"&&theFile into line 6 of temp
  if theRel ≠ "" then put "Relation:"&&theRel into line 7 of temp
  if ErrCode ≠ "" then put "Code Number:"&&ErrCode into line 8 of temp
  answer temp with "Exit Script","Continue"
  if it = "Exit Script" then
    get Jovis("SetProperty","myDB","AutoSave","true")
    if Jovis("IsTransactionOn","myDB") = "true" then
      get Jovis("CancelTransaction", "myDB")
      return "Stop"
    end if
  end if
end JovisWarningMsg
```

The below handlers 'disable' and 'enable' the Jovis warning and error function handlers. In addition, 'DisableJovisHandlers' sets the error/warning global to a new user provided global, and 'EnableJovisHandlers' re-installs the default "JovisErrorCode" global. These two routines allow you to handle errors and warnings at the local script level. Here's an example of how to use them:

```
1:    get Jovis("GoToKeysetName","myDB","ItemPicture")
2:    DisableJovisHandlers "PictErrors"
3:    get Jovis("FindAt","myDB",ItemID)
4:    EnableJovisHandlers
5:    if item 1 of PictErrors ≠ "0" then
6:          answer "Picture key not found"
7:          return "false"
8:    end if
```

Notice how we setup a 'custom' error global called "PictErrors" just before we make the 'FindAt' call. By doing so, the end-user doesn't know if an error occurred until we've restored the error/warning handlers in line 4, and in line 5 tested the custom "PictErrors" global.

```
on DisableJovisHandlers GlobalName
  if GlobalName = "" then --> sanity check
    answer "Error: Setproperty's 'GlobalName' is empty."
    exit to HyperCard
  end if
  -- disable handlers
  get Jovis("SetProperty","myDB","WarningMsg","")
  get Jovis("SetProperty","myDB","ErrorMsg","")
  -- change error code global
  get Jovis("SetProperty","myDB","ErrorGlobal",GlobalName)
end DisableJovisHandlers

on EnableJovisHandlers
 --> This must come first!
  get Jovis("SetProperty","myDB","ErrorGlobal","JovisErrorCode")
 --> reset to standard error and warnings msgs
  get Jovis("SetProperty","myDB","WarningMsg","JovisWarningMsg")
  get Jovis("SetProperty","myDB","ErrorMsg","JovisErrorMsg")
end EnableJovisHandlers
```

Emergency Memory Reserve

An important way to help ensure that Jovis always has enough memory available for essential operations is to maintain an emergency memory reserve. This memory reserve is a block of memory that Jovis uses only for essential operations and only when all other available space has been allocated. When Jovis starts up, it allocates a block of memory and reserves it. When Jovis needs to fulfill an essential memory request and there isn't enough space to satisfy the request, the reserved memory is released. This effectively ensures that Jovis will always have the memory requested, at least for essential operations. Each time one of Jovis' commands is called, it checks whether the reserve has been released. If it has, Jovis attempts to recover the reserve. If it cannot recover the reserve, it returns immediately from the call without execution and reports that memory is low and that the memory reserve has been released.

There are actually two memory reserves maintained by Jovis, one for the commands as they are executed, the other for the Jovis engine itself. If either reserve is triggered by insufficient memory, both reserves are released. With the next call to Jovis, there will be an attempt to recover both reserves.

Initializing Jovis Globals

The information provided here is a supplement to the "Creating, Opening and Closing a Data File" tutorial. If you have not studied this tutorial, you should do so before reading this additional information.

Jovis uses the same technique of initializing a shell application global when creating or opening a data file as it does when working with 'Pict', 'snd ', and 'Blob' globals. The following provides further information and assistance with these globals.

Whenever you work with 'Pict', 'snd ', and 'Blob' commands you must first initialize them to the name "Jovis". For example:

```
1: on MouseUp
2:    global myPict
3:    put "Jovis" into myPict
4:    get Joivs("ClipToPict","myPict")
5: end MouseUp
```

Notice that in line 4, after we have initialized the global 'myPict', the 'ClipToPict' command requres the global to be passed in double quotes. This means that the global's NAME is being passed to Jovis, NOT what the global contains. Once 'ClipToPict' receives the global's name it makes sure that it contains the word "Jovis", if not, an error is returned. This is done to ensure that a possible 'Pict' is not already loaded into memory for this global and is not overwritten or lost. The same thing could happen if you initialized a global that is already in use, so be careful.

When you are finished, you use one of the "Clear" commands; either 'ClearPict', 'ClearSound', or 'ClearBlob' to free memory and to reset the shell application global. Here's our example of what to do:

```
1: on MouseUp
2:    global myPict
3:    if myPict is not empty then
4:       get Joivs("ClearPict","myPict")
5:       if myPict = "Jovis" then put "" into myPict
6:    end if
7: end MouseUp
```

Here, the goal is to give yourself a way to test that the global is in use, and then, after calling one of the "Clear" commands, set the contents of the global to empty. By testing for empty, you catch any attempts to "Clear" a global after it has been cleared.

Keywords, Valid Names and Reserved Characters

Reserved Characters

Do not use the FS (ASCII 028) and GS (ASCII 029) control characters. They must be reserved for use by Jovis.

Standard Naming Convention

The names that you assign relations, fields and indexes,require a standardized format All names must consist of alpha-numeric characters such as, "Address2_alt". Case is non-sensitive. The only non alpha-numeric character allowed is the underscore, "_". You can use as many under-score characters as you would like. The maximum length of a name is 31 characters. Anything outside of these parameters is considered an error.

Keywords

Jovis considers the following words to be keywords, and reserves them strictly for it's use. You cannot use them as the name of a field, index or relation. You can embed them into your own names. Such as: "AccountDate" or "LastName_Field", etc.

The current list of Jovis keywords are:

BLOB	Integer
Boolean	Long
Current	Number
Date	Pict
Decimal	Picture
Field	Sound
Float	String
Logic	Text

Null Keys

A null key is an undefined value, it contains nothing. If you have an index or keyset, say of last names, the index entries that are empty are considered null keys. Avoiding null keys is an important factor in keeping indexes as small and efficient as possible. Empty fields within a relational record are completely acceptable, because Jovis handles this automatically by not indexing empty fields. However, there are times when you by need to test for a certain condition, such how many record have no last name entered, or what records need to have their "Account_Start" field reset. If you require null keys for a particular index, you should initialize the field to a default value. The following example should help you understand this better.

```
on CreateRecordDefaults
   put Jovis("CreateRecord","myDB","Customers") into rec
   put Jovis("SetRecordField","myDB","Customers",rec,
             "Last_Name","<<UNKNOWN>>") into rec
--
   put Jovis("SetRecordField","myDB","Customers",rec,
             "Account_Start","0/0/00") into rec
--
   put Jovis("SetRecordField","myDB","Customers",rec,
             "Zip","-1") into rec
--
   get Jovis("UpdateRecord","myDB","Customers",rec)
end CreateRecordDefaults
```

Page Selecting

Selection paging is the ability to retrieve a specified number of rows at a time based on a given criteria. For example, assume a database with 10,000 records and that you need to review records created after a given date. Your criteria might be something like this: "Field creationDate > 6/20/94". You know that this criteria is going to create a selection of over 1500 records and that once you've reviewed only a few hundred you'll be finished. With selection paging, you set the criteria as normal, but in addition you specifiy how many records you want to retrieve at a given time, say 75 records. By calling SetSelection a second time with the continue parameter set to 'true', you can retrieve the next 75 records. With each subsequent SetSelection call that has the continue parameter set to true, you can set how many records you want to retrieve.

Being able to page through a selection is not a feature of SQL databases. Nor is the ability to set the size of a selection page possible in the SQL language.

The remainder of this document presents an abstract for the SetSelection and AppendSelection commands, and provides a few examples of their usage.

SetSelection

```
get Jovis( "SetSelection", "myDB", "RelationName",
<field list>, [<criteria>], [<function callback>],
[<prompt>], [<number of rows>], [<continue flag>] )
```

Where:

"SetSelection"	the Jovis command to be executed
"myDB"	name (in quotes) of the Jovis global
"RelationName"	name of the relation to select from
<field list>	comma delimited field list. If the field list is "*" or "+", all fields in the relation are included. Note: This parameter is not optional
[<criteria>]	optional criteria. If this parameter is empty the entire relation is selected. Note if the criteria is invalid, no new selection is created; the "active" selection is not replaced nor is the previous criteria changed
[function callback]	optional parameter in which you can place the name of a function in your scripts to be called by Jovis. Note: The function is called after each record is tested
[prompt]	optional prompt message which will appear in the progress window in

which a selection is being created

[<number of rows>] optionally indicates the number of rows to return in a created selection

[continue flag] if "false", the given criteria replaces any previous criteria. If this flag is "true", the 'current' criteria is used, and further selecting continues. Note: a previous call with this parameter set to "true" replaces all records previously selected thereby producing a page-in page-out result. See "AppendSelection" for appending a current selection with the continue flag

AppendSelection

```
get Jovis( "AppendSelection", "myDB", "RelationName",
                    [<criteria>], [<function callback>], [<prompt>],
                    [<number of rows>], [<continue flag>] )
```

Where:

"AppendSelection" the Jovis command to be executed

"myDB" name (in quotes) of the Jovis global

"RelationName" name of the relation to select from

[<criteria>] criteria basis for appending records to the current selection. NOTE: Checks for duplication of records in the existing selection, and replaces any that are already contained in the selection.

[<function callback>] name of the function call in your scripts

[<prompt>] optional prompt message which will appear in the progress window in which a selection is being created

[<number of rows>] indicates the number of rows to append

[<continue flag>] if "false" the given criteria replaces any previous criteria and appends all records previously selected and checks for duplicate records in the existing selection, replacing any that already are contained in the selection. If this flag is "true", the 'current' criteria is used, and further appending continues.

EXAMPLES FOR SELECTION PAGING:

SetSelection:

```
on mouseUp
  global JovisErrorCode,relationName
  put "!RecID,UserName,UserID,UserMF,UserDate" into fldList
```

```
  put "Field !RecID > " & quote & "0" & quote into theCriteria
  get jvois("setSelection","myDB",relationName,
                  fldList,theCriteria,"","",4,false)
  if item 1 of JovisErrorCode = "error" then
    answer JovisErrorCode
    exit mouseUp
  end if
  doSelectionToVar
end mouseUp
```

Get Next Selection Page:

```
on mouseUp
  global JovisErrorCode,relationName
  set cursor to watch
  put "!RecID,UserName,UserID,UserMF,UserDate" into fldList
  get jvois("setSelection", "myDB",relationName,fldList,"","","",4,true)
  if item 1 of JovisErrorCode = "error" then
    answer JovisErrorCode
    exit mouseUp
  end if
  --
  if jvois("countSelection","myDB",relationName) = 0 then
    answer "No more records to select."
  else doSelectionToVar
end mouseUp
```

Append current selection option:

```
on mouseUp
  global JovisErrorCode,relationName
  set cursor to watch
  get jvois("appendSelection", "myDB",relationName,"","","",4,false)
  --
  if item 1 of JovisErrorCode= "error" then
    answer JovisErrorCode
  end if
  --
  if jvois("countSelection","myDB",relationName) = 0 then
    answer "No more records to select."
  else send "doSelectionToVar" to this cd
end mouseUp
```


Relational Record IDs

For each relation that is created, an "internal" index is also created that contains a unique ID number for each record in each relation. As each record is created, the ID is incremented and assigned to the header of each new record. The ID is 'Read-Only'.

The first (and oldest) record in the database is # 1, the second is #2, and so forth. Please note that the ID number is assigned when the record is created. The ID numbers are not reused, so when a record is deleted or never written to the database usng 'UpdateRecord', a permanent gap appears in the sequence of ID numbers.

The ID field name is "!RecID". (NOTE: The "!" is part of the field name. This character is not allowed in any other Jovis field name. This is a special case, controlled by Jovis.)

You can use "!RecID" in creating your criteria. You can also get its value or export it just like any other field. For example:

```
on mouseUp
   put Jovis("GetRecordField","myDB","Customers",aRecord,"!RecID") into recID
end mouseUp
```

It is automatically included as a field in any selection that you set. You can use the 'GetSelectionField' command to retrieve the ID. For example:

```
on mouseUp
   put Jovis("GetSelectionField","myDB","Customers","!RecID",5) into recID
end mouseUp
```

You can also retrieve a record using the !RecID. For example:

```
on mouseUp
   put Jovis("ReadRecord","myDB","Customers","Field !RecID = ["&recID&"]") into
aRec
end mouseUp
```

The index for the ID is used to sequentially read the database, this is done when criteria is optionally not provided. Here's an example of a criteria that uses the "!RecID":

```
   put "Field !RecID ≥ [25] and Field !RecID ≤ [50]" into myCriteria
```

There is one caveat concerning the "!RecID". It has to do with relying on the "!RecID" as a type of reference into your database. For example, suppose thgat you have created a relation for a

business that requires a series invoice number. You might think, no problem, I can use the "!RecID" field. However, during the course of day to day operations, a few records are deleted and some newly created record are not updated to the database. As a result you have 150 actual records, and your !RecID counter is at 185. Now, suppose you export all 150 record to a text file, create a new data file, and then re-import the records. When you create the next new record, the "!RecID" is #151. You may be wondering what is going on here! The answer is that by creating the new data file you also created a new "!RecID" counter. Your export/import process does not update the new "!RecID" to the last ID number of the old relation, only to the number of records imported.

The solution is not to reference your database against the "!RecID". Instead, us an external counter, such as a hidden card field, or even better, create a keyset with a key whose record is a counter that you increment each time you create a relational record. Here's an example of how to do this:

```
function getNextInvoiceNum
   get Jovis("GoToKeysetName","myDB","Counter")
   get Jovis("FindAt","myDB","Inv_Counter")
   -- now lock and read the current key's record
   put Jovis("ReadText","myDB","false","True") into myCounter
   add 1 of myCounter
   get Jovis("RewriteText","myDB",myCounter)
   return myCounter
end getNextInvoiceNum
```

In general, the !RecID field is very handy for search criteria and retrieving a specific record very quickly. However, using it as an intricate component to your database, or as a reference counter could cause problems that out weigh the benefits you think you are gaining.

Custom Function Handlers

The following commands 'ReadRecord','LastRecord', 'NextRecord", 'PriorRecord', 'SetSelection', 'AppendSelection','TrimSelection', 'CountMatches', and 'ExportData' have an optional parameter in which you can place the name of a shell application function, of which you write the script.

The function is called after each database record is tested.

The passes three parameters to the shell application function:

- the number of records tested so far, including the current one,
- the number of records matched so far, including the current one, and
- whether or not the current record matches the criteria.

The function must return two true/false items ("True,False") to the Jovis.

- whether or not to accept the current record, and
- whether or not to end the search.

The shell application function can do anything you like, but must not call any Jovis commands, unless you want a disaster!

The order of execution when using one of the above commands that has a shell application function is like this:

Jovis function handler Jovis function handler Jovis ,etc.

The loop is repeated until Jovis has finished with the criteria, or until the scripter instructs it not to continue the search. Discontinuing a search is done by returning "true" in the second parameter from the shell application function. It should be noted that because there is so much "dialog" between Jovis and the shell application function, the performance that one normally experiences is substantially reduced. If you require maximum control over how any of the above listed commands are executed, you will find that this feature is very helpful.

Below is a sample script for a shell application function. It instructs Jovis to accept the first ten matches.

```
on CreateSelection
      Put "Field zip > " & quote & "60600" & quote &
      " and Field zip < " & quote & "60699" & quote into criteria
      Put "Last_Name,Phone,Customer_ID,State,Zip" into fldList
      get Jovis("SetSelection","myDB","Customers","fldList,
                        Criteria,"CheckSelect")
end CreateSelection


function CheckSelect recsRead, recsMatched, isMatch
      If recsMatched >= 10, then
            return isMatch & ",True"
      else
            return isMatch & ",False"
      end if
end CheckSelect
```

Index Cache and the 'StartUp' Command

Jovis uses a sophisticated index caching system for swapping portions of different indexes to and from secondary storage. This allows indexes to share memory space while maintaining acceptable performance. In order to provide this caching system, there is a minimum memory requirement of 512K that must be available within your shell application. Extra memory is also required for reading records, creating selections and other general needs. Therefore, you should consider at least one megabyte of RAM as a requirement for running Jovis.

If your indexes and/or keysets have more than 100,000 keys, you will want to increase the amount of memory for them. To do so, you must use the 'StartUp' command. This command must be called BEFORE all other Jovis commands are called. It initializes the cache memory system as well as other system managers. It requires one parameter which is the size of the memory allocation for the cache memory system. The default and minimum size is 512K. If you pass anything less, Jovis uses the default size. Here is an example of how to call it:

```
on mouseUp
    get Jovis("StartUp", 512000*2)  --> 1Meg.
end mouseUp
```

Keep in mind that the 'Startup' command is optional, by calling 'CreateCollection', 'OpenCollection', 'DBInit', or 'DBOpen' the system will still be initialized, but will only use the default size. If there are errors during initialization, they are reported to you via the JovisErrorCode global, not with the standard error function handler.

Jovis creates a special shell application global called "JovisPrivateData" which you must never alter! It is used to maintain the cache system between Jovis calls. When you call the 'ShutDown' command, the global is set to empty. If this global is empty, the cache system is not in use, and no database files can be accessed.

Tips On Database Design

There are many books on database design, for the most part they all say essentially the same thing: your best database design is one that serves well defined needs. In other words, if you need certain kinds of reports, your design should reflect how the reports are going to be generated. If a retail store wanted to send out flyers to customers based on purchase history, it should have one relation with customer information, (i.e. names and address) and another relation that lists purchases that have been made by these customers. A customer ID field in both of these relations allows you to search for purchases and then find and retrieve a given customer record using the customer ID.

It is not unusual to design a database, begin using it, only to discover that you need to re-design it. This is like the old saying "keep doing it until you get it right" approach, but it is often a good one, because there are often hundreds of details that you must consider. Jovis is very easy to script, especially if you create "utility" functions for creating relations, fields and indexes. Experimentation is a good method when you are starting to design a database. There is no harm in trying several approaches to a database design.

Rarely does a database design stay static or unchanged. It is common for end-users to request additional fields, or to have fields indexed that were not originally indexed. Often databases that were intented for single-users become client/server databases. Sometimes fields need to be split into two or more fields, and on occasion a field is even deleted.

In designing you database, it is not enough to meet your current requirements, you must also consider future needs. Some helpful tips to avoid trouble later on are:

1. Try to be as modular as possible. For example, don't mix field updating with creating new records in the same routine. Use as many subroutines as you need. Always create routines for re-use.

2. Avoid complexity. Your database can get overly complex if you start creating too many relations and files. Avoid using every database feature just because it is available. Use a clear concise naming convention, and always adhere to it. Try not to be overly ambitious or grandios; databases work best when they provide precise and accurate information to the end-user. Stay organized!

3. Comment everything you do. Use dates, your initials, and explain how and what each routine does. It could be months, or even a year or more before you have to update previous work. You should do everything possible to help yourself and/or others in maintaining a database.

Relational vs: Architectural

With regard to using the Relational or Architectural version for your database, it all depends on whether you need to retrieve records individually or as selections. Of course, you can use all of the Architectural capabilities within a Relational database. However, from the standpoint of which overall capability you need to use, the following information should be considered.

An Architectural database is very fast because there is virtually no overhead in processing your data. You simply find an indexed key and read its attached record. It is more of a "nuts-and-bolts" approach since there are no tables to maintain, as is the case in a relational system. If you have records which are unique, such as pictures, sounds, or text blocks or strings, the Architectural version will be a perfect solution.

With the relational, there is the added processing of field information and interface features. So, there are some preformance considerations to be made, even though the Relational version is very fast. If your data falls into a tabular format, meaning that you have many records with the same fields, then the relational version is for you. Also, the relational version should be used if you need to use search criteria and obtain subsets of your database. A subset could be as small as a single field from one record, or hundreds of entire records. The deciding factor will be the data and the way your end-users need to work with it.