InterSystems
IRIS Data Platform

Specialized System Tools and Utilities

Version 2023.1
2024-07-11

InterSystems Corporation   1 Memorial Drive   Cambridge MA 02142   www.intersystems.com

Specialized System Tools and Utilities
InterSystems IRIS Data Platform Version 2023.1 2024-07-11

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Tables

1

Using System Classes for National Language Support

Modern applications can adapt to various languages and regions without engineering changes. This process is called *internationalization*. The process of adapting an application to a specific region or language by adding specific components for that purpose is *localization*.

The set of parameters that defines the user language, country, and any other, special variant preferences is a *locale*. Locales specify the conventions for the input, output and processing of data. These are such things as

- Number formats
- Date and time formats
- Currency symbols
- The sort order of words
- Automatic translation of strings to another character set

A locale often is identified by noting the language in use and its geographic region (or other variation). These are usually given by the International Standards Organization (ISO) abbreviations for language and location. For example, `en-us` can represent the conventions of the English language as it is used in the United States, and `en-gb` as English is used in Great Britain.

Note: The InterSystems IRIS instances on all members of a mirror must have the same locale and collation. See Mirroring.

1.1 The %SYS.NLS Classes

InterSystems IRIS supports localization via classes in the package %SYS.NLS (NLS refers to National Language Support). These classes contain the information InterSystems IRIS needs to adapt an internationalized program to its runtime circumstances. This section summarizes your options; for additional detail, see the class documentation for each class.

Note: Using any of these classes, an application can obtain the values currently set for the system or the process. Changing the values associated with the process takes effect immediately. To change the system settings, your application must define a new locale with the appropriate values and direct InterSystems IRIS to start using the new locale.

1.1.1 %SYS.NLS.Locale

The properties in %SYS.NLS.Locale contain information about the current locale that you might need to consult. Changing any of them will not affect any behavior of the system.

1.1.2 %SYS.NLS.Device

The class %SYS.NLS.Device contains some properties for the current device, not necessarily the device that was current when the object was instantiated.

Usually, the properties for a specific device are set when the device is opened. This guarantees that the correct translations will be used. It is possible to change the translation table once the device is open by changing the XLTTable property in the process instance of this class, but this is not recommended without a solid reason for doing so.

Other properties in %SYS.NLS.Device enable you to handle errors that occur during translations. By default, when a character cannot be handled by the current table, no error is triggered and the offending character is translated as a question mark (`?`). This character, called the *replacement value* or *replacement string* can be changed to any other string. Furthermore, instead of silently translating undefined characters, it is possible to issue an error. This behavior is called the *default action*, and the possible choices are:

- 0 — Generate error
- 1 — Replace the untranslatable character with the replacement value
- 2 — Ignore the error and pass the untranslatable character through

There are separate properties for the input and output operations in the properties of this class:

- InpDefaultAction
- InpReplacementValue
- OutDefaultAction
- OutReplacementValue

1.1.3 %SYS.NLS.Format

The class %SYS.NLS.Format contains properties that affect the behavior of **$ZDATE()** and related functions. These properties are inherited from the values defined for the current locale but can be altered at the process level without affecting other users. The properties *DateSeparator* and *TimeSeparator*, for example, hold the characters that separate date and time items respectively.

The documentation for $ZDATE, $ZDATEH, and $FNUMBER describes the effect of changing these values.

Locale Property

The Locale property in the class %SYS.NLS.Format allows control of the “look” of values in the current process. For example:

- If Locale is a empty string, the system default formats (usually US English) are in effect.
- If Locale is a locale name such as `rusw` or `csy8`, the formats come from that locale.
- If Locale is `Current`, the formats come from the system.

The property can be changed after the object is instantiated or by passing the desired locale to the **%New()** method as in the following:

ObjectScript

```
Set fmt = ##class(%SYS.NLS.Format).%New("jpnw")
```

These changes affect only the current process.

1.1.4 %SYS.NLS.Table

The class %SYS.NLS.Table can instantiate objects that reflect either the system default or the current process settings for the various categories of tables. A table is the basic NLS mechanism that allows application data to be accepted as input, ordered, and displayed in the format appropriate to the specified locale. As with %SYS.NLS.Locale, changing any property of a system object will not affect the system. However, changing a property from a process object will cause the associated behavior to change immediately.

NLS tables can be classified into I/O and Internal tables. Each table type has its own set of related data:

I/O Tables

The I/O tables (also called translation tables) translate between the basic underlying character set supported by the current locale in which the systems is operating and a foreign character set supported by some entity outside InterSystems IRIS. The locale character set might be, for example, Latin2 (more properly known as ISO 8859-2) and the foreign character set might be UTF-8, generally used to communicate with the Terminal. Thus, on output, a table like Latin2–to-UTF8 would be used and, on input, a reverse mapping table would be needed, UTF8–to-Latin2.

Although there are two tables involved here (one for input and another for output), these tables usually complement one another. For simplicity, when speaking of locale definitions and system defaults, InterSystems IRIS uses a single name for a pair of I/O tables. This name is usually the name of the foreign character set, with the tacit assumption that the other half is the locale character set. However, when creating custom tables, any name that conveys the meaning of the exchange can be chosen.

I/O tables are used in *devices*; in this case, the word *device* refers to any interface where InterSystems IRIS meets the external world and where translation is needed, including the process and system call interfaces.

- Terminal
- Other terminal connections
- External files
- TCP/IP connections
- Printer
- InterSystems IRIS processes
- System call

For more information, see the Translation Tables reference page.

Internal Tables

The internal tables also map strings of characters from the current local character set to some other value, but they are not intended to be used in communication with the external world. The internal tables identify characters that are part of:

- Pattern matching

  Identify the characters that match certain pattern codes such as letters, numbers, punctuation, and so on.

- Identifiers

  Identifier tables indicate which characters can be used in identifiers.

- Uppercase, lowercase alphabets, and uppercase when used in titles.

These similar in structure to the I/O tables; they map from one character set to another which just happens to be the same set. However, they are used in the context of $ZCONVERT(), not with some I/O operation.

- Collation ordering

  These tables map a string of characters into an internal representation of that string suitable for use in global subscripts. Different languages have differing rules about how words should collate in dictionary order; these rules are encapsulated in a collation table.

- $X/$Y action

  These tables map characters into values that indicate how they interact with the $X and $Y special variables. Should $X and/or $Y be incremented after this character is output? Is the character printable? These are questions that a $X/$Y table answers.

Note: The list of available collations in any version of InterSystems IRIS is fixed. If your needs are not met by an existing collation, please contact the InterSystems Worldwide Support Center for assistance.

1.2 Examples Using %SYS.NLS

Important: These examples are all executable but none have a **RunIt** button, because they manipulate process-default values for the current locale. Also, many require administrative privileges and/or write access to the %SYS namespace. If you wish to execute them, please run them in a separate process, such as the InterSystems Terminal facility (Windows), or via a TCP/IP connection, and with the appropriate privileges.

1.2.1 Display Current Locale Information

This example displays information about the current system locale:

ObjectScript

```
Set Info = ##class(%SYS.NLS.Locale).%New()
Set Items = "Name" _
            "/Description" _
            "/Country" _
            "/CountryAbbr" _
            "/Language" _
            "/LanguageAbbr" _
            "/Currency" _
            "/CharacterSet"

Write !
For i = 1 : 1 : $LENGTH(Items, "/")
{
  Set Item = $PIECE(Items, "/", i)
  Write $JUSTIFY(Item, 15),": ", $PROPERTY(Info, Item), !
}
```

1.2.2 Display System and Process Table Data

This should display the same values for the system and process tables unless some properties have been externally altered before running this example.

ObjectScript

```
Set IOTables = "Process" _
               "/IRISTerminal" _
               "/OtherTerminal" _
               "/File" _
               "/TCPIP" _
               "/SystemCall" _
               "/Printer"
Set IntTables = "PatternMatch" _
                "/Identifier" _
                "/Uppercase" _
                "/Lowercase" _
                "/Titlecase" _
                "/Collation" _
                "/XYAction"

// iterate over the systems, and then the process data
For Type = "System", "Process"
{
  Write !
  Set Table = ##class(%SYS.NLS.Table).%New(Type)
  Write "Type: ", Type, !

  Write "I/O Tables", !
  For i = 1 : 1 : $LENGTH(IOTables, "/")
  {
    Set PropName = $PIECE(IOTables, "/", i)
    Write $JUSTIFY(PropName, 15), ": ", $PROPERTY(Table, PropName), !
  }

  Write "Internal Tables", !
  For i = 1 : 1 : $LENGTH(IntTables, "/")
  {
    Set PropName = $PIECE(IntTables, "/", i)
    Write $JUSTIFY(PropName, 15), ": ", $PROPERTY(Table, PropName), !
  }
}
```

1.2.3 Changing Date and Time Displays

The %SYS.NLS.Format class contains the properties DateSeparator and TimeSeparator, for example, hold the characters used to separate the components of date and time items respectively. In the United States default locale, `enu8` (or `enuw` for Unicode systems), these are the slash character (`/`) and the colon (`:`), respectively. The following example shows how these may be altered:

ObjectScript

```
// display the current defaults
// date is 10 April 2005
// time is 6 minutes 40 seconds after 11 in the morning
Write $ZDATE("60000,40000"), !

// now change the separators and display it again
Set fmt = ##class(%SYS.NLS.Format).%New()
Set fmt.DateSeparator = "_"
Set fmt.TimeSeparator = "^"
Write !, $ZDATE("60000,40000")
```

This following example changes the month names to successive letters of the alphabet (for demonstration purposes). To do this, it sets the property *MonthName* to a space-separated list of the month names. Note that the list starts with a space:

ObjectScript

```
// get the format class instance
Set fmt = ##class(%SYS.NLS.Format).%New()

// define the month names
Set Names = " AAA BBB CCC DDD EEE FFF GGG HHH III JJJ KKK LLL"
Set fmt.MonthAbbr = Names
Set rtn = ##class(%SYS.NLS.Format).SetFormatItem("DATEFORMAT", 2)

// show the result
Write $ZDATE(60000, 2)
```

1.2.4 Changing the Way Numbers Are Displayed

Some properties in %SYS.NLS.Format control how numbers are interpreted by $Number(). In English locales, the decimal point is used to separate the integer from the fractional part of a number, and a comma is used to separate groups of 3 digits. This too can be altered:

ObjectScript

```
// give the baseline display
Write $Number("123,456.78"), !

Set fmt = ##class(%SYS.NLS.Format).%New()
// use "/" for groups of digits
Set fmt.NumericGroupSeparator = "."

// group digits in blocks of 4
Set fmt.NumericGroupSize = 4

// use ":" for separating integer and fractional parts
Set fmt.DecimalSeparator = ","

// try interpreting again
Write $Number("12.3456,78"), !
```

1.2.5 Setting the Translation for a File

The following shows that an application can control the representation of data written to a file.

ObjectScript

```
// show the process default translation (RAW, no translation performed)
Set Tbl = ##class(%SYS.NLS.Table).%New("Process")
Write "Process default translation: ", Tbl.File, !

// create and open a temporary file
// use XML for the translation
Set TempName = ##class(%Library.File).TempFilename("log")
Set TempFile = ##class(%Library.File).%New(TempName)
Do TempFile.Open("WSNK\XML\")
Write "Temp file: ", TempFile.CanonicalNameGet(), !

// write a few characters to show the translation
// then close it
Do TempFile.WriteLine(("--" _ $CHAR(38) _ "--"))
Do TempFile.Close()

// now re-open it in raw mode and show content
Do TempFile.Open("RSK\RAW\")
Do TempFile.Rewind()
Set MaxChars = 50
Set Line = TempFile.Read(.MaxChars)
Write "Contents: """, Line, """", !

// finish
Do TempFile.Close()
Do ##class(%Library.File).Delete(TempName)
Set TempFile = ""
```

For more information on translation tables, see the section on “Three-Parameter Form: Encoding Translation” in the documentation for the **$ZCONVERT** function.

1.3 The Config.NLS Classes

Unlike %SYS.NLS, which is available everywhere and is intended for general use, the classes in Config.NLS can be used only in the %SYS namespace and only by a user with administrative privileges. Normally, administrators who need to create custom locales and tables would use the NLS pages in the Management Portal. Only users with very special requirements should need to use Config.NLS.

There are three classes in package Config.NLS:

- Locales – Contain all the definitions and defaults for a country or geographical region.
- Tables – Contain a high level description of tables, but not the mapping itself.
- SubTables – Contain the character mappings proper and may be shared by more than one Table.

The main reason for having separate Tables and SubTables classes is to avoid duplication of data. It is possible to have Tables for different character sets that happen to share the same mappings and thus the same SubTable. Also, the classes in Tables define a default action and a replacement value (see description of these properties in %SYS.NLS above). Therefore, it is possible to have separate Tables in which these attributes are different even though they share the same SubTable. This flexibility adds some complexity in managing the correct relationships between Tables and SubTables, but the gains make it worthwhile. The separation of Tables from SubTables is kept hidden from users in the Management Portal and the %SYS.NLS classes, where all the housekeeping is done. However, when working with Config.NLS this needs to be done explicitly.

1.3.1 Conventions for Naming User-Defined Locales and Tables

To differentiate your custom items from the system items, and to simplify upgrades, use a `y` at the start of the name of your items; for example: `XLT-yEBCDIC-Latin1` and `XLT-Latin1-yEBCDIC`.

CAUTION: User-defined tables, sub-tables and locales that do not follow this convention may be deleted during a system upgrade. The way to avoid this is to export user-defined tables and locales to XML files and re-import them after the upgrade.

When a custom SubTable is created from a copy of some InterSystems SubTable, the utilities that perform this task automatically use the same name and append a numeric suffix. Thus, copies of the Latin2-to-Unicode SubTable would be named `XLT-Latin2-Unicode.0001` and `XLT-Unicode-Latin2.0001`, and so on.

1.4 Examples Using Config.NLS

This section presents the following examples:

- Listing the Available Locales
- Listing the Tables in a Specific Locale
- Creating a Custom Locale

1.4.1 Listing the Available Locales

This example uses the **List()** class query in Config.NLS.Locales and displays a list of the available locale identifiers and descriptions.

ObjectScript

```
  new $namespace
  set $namespace="%SYS"

  set stmt=##class(%SQL.Statement).%New()
  set status=stmt.%PrepareClassQuery("Config.NLS.Locales","List")
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set locales=stmt.%Execute("*")
  if (locales.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", locales.%SQLCODE, ": ",
locales.%Message quit}

  while locales.%Next()
  {
    write locales.%Get("Name"), " - ", locales.%Get("Description"), !
  }
  if (locales.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", locales.%SQLCODE, ": ",
locales.%Message quit}
```

1.4.2 Listing the Tables in a Specific Locale

The following example shows the tables that make up the Unicode locale for United States English (if it is available).

ObjectScript

```
  new $namespace
  set $namespace="%SYS"

  // establish the locale identifier, try
  // United States - English - Unicode
  // United States - English - 8-bit
  Set Loc = "enuw"
  Do ##class(Config.NLS.Locales).Exists(Loc, .Ref, .Code)
  If (##class(%SYSTEM.Status).IsError(Code))
  {
    Set Loc = "enu8"
    Do ##class(Config.NLS.Locales).Exists(Loc, .Ref, .Code)
    If (##class(%SYSTEM.Status).IsError(Code))
    {
      Do ##class(%SYSTEM.Status).DisplayError(Code)
      Quit
    }
  }

  // get the local array of table names
  Write "Tables for locale: ", Loc, !
  Do Ref.GetTables(.Tables)
  Set Type = $ORDER(Tables(""))
  While (Type '= "")
  {
    Set Name = $ORDER(Tables(Type, ""))
    While (Name '= "")
    {
      Set Mod = $ORDER(Tables(Type, Name, ""))
      While (Mod '= "")
      {
        Write Type, " - ", Name, " - ", Mod, !
        Set Mod = $ORDER(Tables(Type, Name, Mod))
      }
      Set Name = $ORDER(Tables(Type, Name))
    }
    Set Type = $ORDER(Tables(Type))
  }
```

1.4.3 Creating a Custom Locale

This example will provide a template for creating a custom locale with a custom table. The custom table will translate between EBCDIC (the common form used in the US) and Latin-1 (ISO-8859–1). For more details, see the documentation for the respective classes.

As for any other table, first we need to get the definition for the character mappings. For this example we are using the data file from the web site http://source.icu-project.org (International Components for Unicode). The relevant data file is a text file with comment lines starting with a pound sign (#) and then a series of translation definition lines of the form:

```
<Uuuuu>  \xee |0
```

A small excerpt of the file looks like:

```
#
#UNICODE EBCDIC_US
#_______ _________
<U0000>  \x00 |0
<U0001>  \x01 |0
<U0002>  \x02 |0
<U0003>  \x03 |0
<U0004>  \x37 |0
<U0005>  \x2D |0
...
```

The lines indicate that Unicode character Uaaaa maps to EBCDIC character \xbb (where aaaa and bb are expressed in hexadecimal). We assume that the table is reversible and that EBCDIC character \xbb maps back to Unicode character Uaaaa. This allows us to create both sides (that is, EBCDIC-to-Latin1 and Latin1-to-EBCDIC) from the same data file in a single scan. Because the Unicode range is just from 0 to 255, this is actually a Latin-1 table.

The process first creates the SubTable object, then the Table, and finally the Locale. For the first step, the process creates two SubTables objects, initializes their Name and Type properties, and then fills in the FromTo mapping array with data read from the definition file.

SubTable names take the form, *Type–FromEncoding–ToEncoding*. The `Type` for regular I/O translations is “XLT” and so the SubTable names will be `XLT-yEBCDIC-Latin1` and `XLT-yLatin1-EBCDIC.`

The following code creates the SubTables objects. In a real world program, the code would perform a number of consistency checks that omitted here for the sake of clarity. This example deletes an existing previous versions of the same objects (SubTables, Tables and Locales) so that you can run the example multiple times. More properly, you should check for the existence of previous objects using the class method **Exists()** and take a different action if they are already present.

ObjectScript

```
  // Names for the new SubTables (save for later)
  Set nam1 = "XLT-Latin1-yEBCDIC"
  Set nam2 = "XLT-yEBCDIC-Latin1"

  // Delete existing SubTables instances with same ids
  Do ##class(Config.NLS.SubTables).Delete(nam1)
  Do ##class(Config.NLS.SubTables).Delete(nam2)

  // Create two SubTable objects
  Set sub1 = ##class(Config.NLS.SubTables).%New()
  Set sub2 = ##class(Config.NLS.SubTables).%New()

  // Set Name and Description
  Set sub1.Name = nam1
  Set sub1.Description = "ICU Latin-1->EBCDIC sub-table"
  Set sub2.Name = nam2
  Set sub2.Description = "ICU EBCDIC ->Latin-1 sub-table"
```

The SubTables object contains a property, type, that is a small integer indicating whether we are dealing with a multibyte translation or not. This example sets type to zero indicating a single-byte mapping. The mapping is initialized so that code points (characters) not defined in the data file are mapped to themselves.

ObjectScript

```
  // Set Type (single-to-single)
  Set sub1.Type = 0
  Set sub2.Type = 0

  // Initialize FromTo arrays
  For i = 0 : 1 : 255
  {
    Do sub1.FromTo.SetAt(i, i)
    Do sub2.FromTo.SetAt(i, i)
  }
```

Next the application reads the file. Definitions in the file override those set as the default mapping. The function $ZHEX() converts the codes from hexadecimal to decimal.

ObjectScript

```
  // Assume file is in the mgr directory
  Set file = "glibc-EBCDIC_US-2.1.2.ucm"

  // Set EOF exit trap
  Set $ZTRAP = "EOF"

  // Make that file the default device
  Open file
  Use file
  For
  {
    Read x
    If x?1"<U"4AN1">".E
    {
      Set uni = $ZHEX($E(x,3,6)),ebcdic = $ZHEX($E(x,12,13))
      Do sub1.FromTo.SetAt(ebcdic,uni)
      Do sub2.FromTo.SetAt(uni,ebcdic)
     }
  }

EOF  // No further data
  Set $ZT = ""
  Close file

  // Save SubTable objects
  Do sub1.%Save()
  Do sub2.%Save()
```

The character mappings are now complete. The next step is to create the Table objects that reference the SubTables objects just defined. Table objects are really descriptors for the SubTables and have only a few properties. The following code makes the connection between the two:

ObjectScript

```
  // Delete existing Tables instances with same ids
  Do ##class(Config.NLS.SubTables).Delete("XLT", "Latin1", "yEBCDIC")
  Do ##class(Config.NLS.SubTables).Delete("XLT", "yEBCDIC", "Latin1")

  // Create two Table objects
  Set tab1 = ##class(Config.NLS.Tables).%New()
  Set tab2 = ##class(Config.NLS.Tables).%New()

  // Set description
  Set tab1.Description = "ICU loaded Latin-1 -> EBCDIC table"
  Set tab2.Description = "ICU generated EBCDIC -> Latin-1 table"

  // Set From/To encodings
  Set tab1.NameFrom = "Latin1"
  Set tab1.NameTo = "yEBCDIC"
  Set tab2.NameFrom = "yEBCDIC"
  Set tab2.NameTo = "Latin1"

  // Set SubTable
  Set tab1.SubTableName = nam1
  Set tab2.SubTableName = nam2

  // Set Type
  Set tab1.Type = "XLT"
  Set tab2.Type = "XLT"
```

```
// Set Default Action
// 1 = Replace with replacement value
Set tab1.XLTDefaultAction = 1
Set tab2.XLTDefaultAction = 1

// Set Replacement value of "?"
Set tab1.XLTReplacementValue = $ASCII("?")
Set tab2.XLTReplacementValue = $ASCII("?")

// Set Reversibility
// 1 = Reversible
// 2 = Generated
Set tab1.XLTReversibility = 1
Set tab2.XLTReversibility = 2

// Set Translation Type
// 0 = non-modal to non-modal
Set tab1.XLTType = 0
Set tab2.XLTType = 0

// Save Table objects
Do tab1.%Save()
Do tab2.%Save()
```

With the Tables defined, the last step of the construction is to define a locale object that will incorporate the new tables. The application creates an empty Locale object and fills in each of the properties as was done for the Tables and SubTables. A Locale, however, is bigger and more complex. The easiest way to make a simple change like this is to copy an existing locale and change only what we need. This process uses `enu8` as the source locale and names the new one, `yen8.` The initial `y` makes it clear this is a custom locale and should not be deleted on upgrades.

ObjectScript

```
// Delete existing Locales instance with the same id
Do ##class(Config.NLS.Locales).Delete("yen8")

// Open source locale
Set oldloc = ##class(Config.NLS.Locales).%OpenId("enu8")

// Create clone
Set newloc = oldloc.%ConstructClone()

// Set new Name and Description
Set newloc.Name = "yen8"
Set newloc.Description = "New locale with EBCDIC table"
```

With the locale in place, the process now adds the EBCDIC table to the list of I/O tables that are loaded at startup. This is done by inserting a node in the array property XLTTables, as follows:

```
XLTTables(<TableName>) = <components>
```

- *tablename* identifies the pair of input and output tables for this locale.

  Because the name does not need to start with `y`, we use `EBCDIC`.

- *components* is a four-item list as follows:
  1. The input “From” encoding
  2. The input “To” encoding
  3. The output “From” encoding
  4. The output “To” encoding

The following code adds the table to the list of available locales:

ObjectScript

```
// Add new table to locale
Set component = $LISTBUILD("yEBCDIC", "Latin1", "Latin1", "yEBCDIC")
Do newloc.XLTTables.SetAt(component, "EBCDIC")
```

Before the locale is usable by InterSystems IRIS, it must be compiled into its internal form. This is also sometimes called validating the locale. The **IsValid()** class method does a detailed analysis and returns two arrays, one for errors and one for warnings, with human-readable messages if the locale is not properly defined.

ObjectScript

```
// Check locale consistency
If '##class(Config.NLS.Locales).IsValid("yen8", .Errors, .Warns)
{
  Write !,"Errors: "
  ZWrite Errors
  Write !,"Warnings: "
  ZWrite Warns
  Quit
}

// Compile new locale
Set status = ##class(Config.NLS.Locales).Compile("yen8")
If (##class(%SYSTEM.Status).IsError(status))
{
  Do $System.OBJ.DisplayError(status)
}
Else
{
  Write !,"Locale yen8 successfully created."
}
```

1.5 Using %Library.GlobalEdit to Set the Collation for a Global

The collation of newly created InterSystems IRIS globals is automatically set to the default collation of the database in which the global is created. The databases created by InterSystems IRIS installation are all set to the InterSystems IRIS standard collation, except USER, which is set to the default collation for the locale with which InterSystems IRIS is installed.

After you create a database, you can edit its properties to change its default collation. You can select InterSystems IRIS standard, the default collation for the locale, or any other collation loaded in the instance. Once the default collation of the database is set, any globals created in this database are created with this default collation.

InterSystems IRIS also supports the ability to override this behavior and specify a custom collation for a global. To do this, use the **Create()** method in the class %Library.GlobalEdit supplying the collation desired:

ObjectScript

```
Set sc = ##class(%Library.GlobalEdit).Create(ns,
                                             global,
                                             collation,
                                             growthblk,
                                             ptrblock,
                                             keep,
                                             journal,
                                             .exists)
```

where:

- *ns* — Specifies the namespace, where `""` indicates the current namespace, or `^^directoryname` references a specific directory.
- *global* — Specifies the global name, including leading ^, such as *^cz2*.
- *collation* — Specifies the collation, where collation is one of the supported collations.
- *growthblk* — Specifies the starting block for data.
- *ptrblk* — Specifies the starting block for pointers.

- *keep* — Specifies whether or not to keep the global’s directory entry when the global is killed. Setting this to 1 preserves the collation, protection, and journal attributes if the global is killed.
- *journal* — This argument is no longer relevant and is ignored.
- *exists* — Specifies, by reference, a variable that indicates whether the global already exists.

In environments in which some globals require different collations from other globals, InterSystems recommends that you set up a database for each different collation, and that you add a global mapping within the namespace to map each global to the database with its required collation. This method allows mixed collations to be used without changing application code to specially use the **Create()** method call.

1.5.1 Supported Collations

The following are supported in InterSystems IRIS, for use in the *collation* argument of the **CreateGlobal^%DM** subroutine:

- 5 — InterSystems IRIS standard
- 10 — German1
- 11 — Portuguese1
- 12 — Polish1
- 13 — German2
- 14 — Spanish1
- 15 — Danish1
- 16 — Cyrillic1
- 17 — Greek1
- 18 — Czech1
- 19 — Czech2
- 20 — Portuguese2
- 21 — Finnish1
- 23 — Cyrillic2
- 24 — Polish2
- 27 — French1
- 28 — Finnish2
- 29 — Hungarian1
- 30 — German3
- 31 — Polish3
- 32 — Spanish2
- 33 — Danish2
- 34 — Greek2
- 35 — Finnish3
- 36 — Lithuanian1
- 41 — Danish3

- 44 — Czech3
- 45 — Hungarian2
- 47 — Spanish3
- 49 — Spanish4
- 51 — Spanish5
- 52 — Finnish4

Note: To see a similar list, including which collations have been loaded into the instance, open a Terminal window, change to the `%SYS%` namespace, and enter the command **DO ^COLLATE**.

1.5.2 Default Collation for the Installed Locale

The default collation for the locale of a new installation of InterSystems IRIS is always the most recent version of the collation, that is, the one with the highest numeric suffix (as shown in the list in the previous section). For example, when installing with a Spanish locale, the default collation is Spanish5. Older versions of the collation are supported for compatibility with existing databases.

When an InterSystems IRIS instance is upgraded, the default collation is preserved when the updated locale uses a new default. For example, if the existing instance's locale uses Finnish3 as the default collation and the updated instance would use Finnish4, the upgrade preserves Finnish3 as the default, but makes Finnish4 available for new globals and databases.

2

Customizing Start and Stop Behavior with ^%ZSTART and ^%ZSTOP Routines

InterSystems IRIS can execute your custom code when certain events occur. Two steps are required:

1. Define the **^%ZSTART** routine, the **^%ZSTOP** routine, or both.

   Note: **^%ZSTART** and **^%ZSTOP** are not included with InterSystems IRIS, and must be created by the user.

   In these routines, you can define subroutines to execute when the certain activities start or stop.

   ^%ZSTART and **^%ZSTOP** must be created and defined in the %SYS namespace, although they can be mapped to a non-default database.

2. Use the Management Portal to configure InterSystems IRIS to invoke the desired subroutines.

Specifically, if you define the routine **^%ZSTART** and **^%ZSTOP** and you include subroutines with specific names, the system automatically calls these subroutines when the activity is beginning or ending. The subroutine names are as follows:

- **SYSTEM** — Executed when InterSystems IRIS as a system starts or stops
- **LOGIN** — Executed when a user performs a login or logout using the **`%Service_Console`** or **`Service_Telnet`** services.
- **JOB** — Executed when a **JOB** begins or ends
- **CALLIN**: — Executed when an external program begins or completes a **CALLIN**

For example, when a user logs in, the system automatically invokes **LOGIN^%ZSTART**, if that is defined and if you have used the Management Portal to enable this subroutine.

These subroutines are not intended to do complex calculations or run for long periods of time. Long calculations or potentially long operations like network accesses will delay the completion of the activity until the called routine returns. In this case, users may take a long (elapsed) time to login, or JOB throughput may be curtailed because they take too long to start.

Note: These subroutines are called as part of normal InterSystems IRIS operation. This means that an external event which terminates InterSystems IRIS abnormally, such as a power failure, will not generate a call to **^%ZSTOP**.

Note: If a system implements **^%ZSTOP**, and an application implements one or more $HALT routines, the **^%ZSTOP** code is not executed until the last $HALT terminates with a HALT command. The failure of a $HALT routine to issue its own HALT command can prevent the **^%ZSTOP** code from running.

2.1 Design Considerations

Because **^%ZSTART** and **^%ZSTOP** run in a somewhat restricted environment, the designer must keep several things in mind, namely:

- The routines must be written in ObjectScript.
- Since **^%ZSTART** is essentially run as though it is started with an argumentless new command, it can not be used to perform tasks such as initializing local variables for users.
- There are no values passed as arguments when any of the routine entry points are called. If different algorithms are applicable in various circumstances, the called entry point must determine what to do by examining data external to the routine: global, system variables, and so on.
- Make sure that the routines are well-behaved under all possible conditions. They should be written defensively. That is, they should check to make sure that all the resources needed to complete their task are at hand and, if possible, reserved to them before computation starts. Errors which occur are reported as failures of that system function so it is important to think about the design from the viewpoint of error containment and handling. Failure to properly account for recovery in the face of missing resources or the presence of errors has varied consequences: InterSystems IRIS may fail to start; major functions such as Studio may act strangely; or more subtle and insidious consequences may occur which are not immediately detected. It is strongly recommended that these routines be carefully written and debugged under simulated conditions, and then tested under simulated environment conditions before being put into production systems.
- No assumption should be made that conditions found during a previous invocation or a different entry point are still valid. Between successive calls to **JOB^%ZSTART**, for example, a file used by the prior call could have been deleted before this call occurred.
- Each entry point should perform its task efficiently. If part of the task is potentially long-running, it is recommended that you queue enough information to complete the task for later completion by another part of your application.
- If an entry point wishes to have data around persistently for, say, statistical purposes, it must use something like a global or an external file to hold the data.
- The routines should make minimal assumptions about the environment they are running in. A developer of one of these routines cannot, for example, assume that the program will always be executed under a specific job number. The designer cannot assume that the various entry point will be called in a specific order. The sequence of bringing up the multiple processes that implement InterSystems IRIS is rarely deterministic.
- The routine cannot assume that it is being called at a specific point during the system startup. The sequence of events during startup may change from release to release, or even from restart to restart.
- With a few exceptions, the routine must leave things as it found them. As an illustration of this principle, reassigning the value of *$IO* in the subroutine without saving and restoring it upon entry and exit is an almost certain source of error. The calling routine has no way of knowing that such things are changed, and it is very difficult for the caller to defend against any possible change to the execution environment. Therefore, the burden of not disturbing the system processing context lies on the subroutine being called.

  The general exceptions to the no-changes rule are that changing process-local values specific to an application or installation are allowed. For example, the **SYSTEM^%ZSTART** entry point may set system-wide defaults. Similarly, for application testing, it could set the date to a specific value to validate end-of-month processing.
- **^%ZSTOP** cannot contain references to globals in remote databases. At the time it is called, some of these may no longer be accessible.
- Once **SYSTEM^%ZSTOP** is invoked during the shutdown process, a user is no longer able to use the JOB command to start new processes. Existing processes are allowed to finish.

- If these routines are mapped to a database different from IRISSYS, then InterSystems IRIS will attempt to execute them from that database, not from IRISSYS. InterSystems IRIS will, of course, make certain that the calling routine has the appropriate access to that database beforehand. It is the responsibility of the administrator to ensure that the routine has access to any application globals and mappings that it requires from that namespace.
- **SYSTEM^%ZSTART** and **SYSTEM^%ZSTOP** are run with *$USERNAME* set to **`%SYSTEM`** and *$ROLES* set to **`%All`**. To run your code with a different username, use **$SYSTEM.Security.Login()** to set the desired name and then continue with your custom code. If you use JOB in your **SYSTEM^%ZSTART** code to launch any additional processes, those processes will inherit the same username (and roles) as the initiating process.

  CAUTION: All the entry points in **^%ZSTART** and **^%ZSTOP** are invoked at critical points in system operation and can have widespread effects on the operation of a system or even on its data. The specified purpose of these routines makes this high level of privilege necessary. Because of this, you must make sure that all code that can be invoked by these entry points has been thoroughly tested. Further, do not allow any user-specified code to be run via **XECUTE** or indirection.

- The exiting (that is, halting) process may get a <FUNCTION> error on any reference that requires an answer from a distributed cache cluster data server.

Note: On upgrades, InterSystems IRIS preserves only the %Z* routines that are mapped to the IRISSYS database, and if the .INT or .MAC code is available, recompiles them. Preservation of routines in other databases are the responsibility of the site administrator.

2.2 Enabling %ZSTART and %ZSTOP

Once the routines have been designed, developed, compiled, and are ready to be tested, individual entry points may be enabled through the Management Portal. Navigate to the Startup Settings page by selecting **System Administration**, then **Configuration**, then **Additional Settings**, then **Startup Settings**, and edit the appropriate individual settings:

- SystemStart, SystemHalt
- ProcessStart, ProcessHalt
- JobStart, JobHalt
- CallinStart, CallinHalt

To deactivate one or more of the entry points, use the same procedure but change the value to **false**.

2.3 Debugging ^%ZSTART and ^%ZSTOP

The opportunities for debugging **^%ZSTART** and **^%ZSTOP** in their final environment are very limited. If an error occurs, errors are written to the operator messages log, which is the current device while these routines are running. This file is messages.log and is found in the Manager’s directory.

The message indicates the reason for failure and location where the error was detected. This may be different from the place where the error in the program logic or flow actually occurred. The developer is expected to deduce the nature and location of the error from the information provided, or modify the routine so that future tests provide more evidence as to the nature of the error.

2.4 Removing %ZSTART and ^%ZSTOP

It is strongly recommended that you disable the entry point options via the Management Portal before deleting the routines. If the portal warns that a restart of InterSystems IRIS is needed for them to take effect, do this as well before proceeding. This guarantees that none of the entry points are being executed while they are being deleted.

Remember that **^%ZSTART** and **^%ZSTOP** (as well as any supporting routines) are stored persistently. To remove all traces of them, delete them through the Management Portal.

2.5 Example

The following example demonstrates a simple log for tracking system activity. It shows examples for **^%ZSTART** and **^%ZSTOP**, both of which use subroutines of a third example routine, **^%ZSSUtil**, for convenience.

2.5.1 ^%ZSSUtil Example

This routine has two public entry points. One writes a single line to the operator messages log file. The other writes a list of name-value pairs to a local log file. Both files reside in the Manager's directory, which is returned by the **ManagerDirectory()** method of the %Library.File class.

ObjectScript

```
%ZSSUtil ;
    ; this routine packages a set of subroutines
    ; used by the %ZSTART and %ZSTOP entry points
    ;
    ; does not do anything if invoked directly
    quit

#define Empty ""
#define OprLog 1

WriteConsole(LineText) PUBLIC ;
    ; write the line to the messages log
    ; by default the file messages.log in the MGR directory
    new SaveIO

    ; save the current device and open the operator console
    ; set up error handling to cope with errors
    ; there is little to do if an error happens
    set SaveIO = $IO
    set $ZTRAP = "WriteConsoleExit"
    open $$$OprLog
    use $$$OprLog
    ; we do not need an "!" for line termination
    ; because each WRITE statement becomes its
    ; own console record (implicit end of line)
    write LineText
    ; restore the previous io device
    close $$$OprLog
    ; pick up here in case of an error
WriteConsoleExit ;
    set $ZTRAP = ""
    use SaveIO
    quit

WriteLog(rtnname, entryname, items) PUBLIC ;
    ; write entries into the log file
    ; the log is presumed to be open as
    ; the default output device
    ;
    ; rtnname: distinguishes between ZSTART & ZSTOP
    ; entryname: the name of the entry point we came from
    ; items: a $LIST of name-value pairs
    new ThisIO, ThisLog
    new i, DataString

    ; preserve the existing $IO device reference
    ; set up error handling to cope with errors
    ; there is little to do if an error happens
    set ThisIO = $IO
    set $ZTRAP = "WriteLogExit"

    ; construct the name of the file
    ; use the month and day as part of the name so that
    ; it will create a separate log file each day
    set ThisLog = "ZSS"
                _ "-"
                _ $EXTRACT($ZDATE($HOROLOG, 3), 6, 10)
                _".log"

    ; and change $IO to point to our file
    open ThisLog:"AWS":0
    use ThisLog

    ; now loop over the items writing one line per item pair
    for i = 1 : 2 : $LISTLENGTH(items)
    {
        set DataString = $LISTGET(items, i, "*MISSING*")
        if ($LISTGET(items, (i + 1), $$$Empty) '= $$$Empty)
        {
            set DataString = DataString
                           _ ": "
                           _ $LISTGET(items, (i + 1))
        }
        write $ZDATETIME($HOROLOG, 3, 1),
              ?21, rtnname,
              ?28, entryname,
              ?35, DataString, !
    }

    ; stop using the log file and switch $IO back
    ; to the value saved on entry
    close $IO
    ; pick up here in case of an error
WriteLogExit ;
    set $ZTRAP = ""
    use ThisIO
    quit
```

Here is an description for each label:

^%ZSSUtil

This routine (as well as the others) begins with a QUIT command so that it is benign if invoked via

ObjectScript

```
do ^%ZSSUtil
```

The #DEFINE sequence cosmetically provides named constants in the body of the program. In this instance, it names the empty string and the device number of the operator messages log.

WriteConsole^%ZSSUtil

The entry point is very simple. It is designed for low volume output, and as a minimally intrusive routine to use for debugging output.

It takes a single string as its argument and writes it to the operator messages log. However, it must take care to preserve and restore the current *$IO* attachment across its call.

Each item sent to the device results in a separate record being written to the messages log. Thus, the following results in four records being written.

```
WRITE 1, 2, 3, !
```

The first three consist of a single digit and the fourth is a blank line. If multiple items are desired on a line, it is the responsibility of the caller to concatenate them into a string.

WriteLog^%ZSSUtil

This subroutine can be called by any entry point within **^%ZSTART** or **^%ZSTOP**. The first two arguments supply the information needed to report how the subroutine was launched. The third argument is a **$LIST** of name-value pairs to be written to the log.

This entry point first builds the name of the file it will use. To make log management easier, the name contains the month and day when the routine is entered. Therefore, calls to this subroutine create a new file whenever the local time crosses midnight. because the name is determined only at the time of the call. All the name-value pairs passed as the argument will be displayed in the same file.

Once the name has been constructed, the current value of *$IO* is saved for later use and the output device is switched to the named log file. The parameters used for the OPEN command ensure that the file will be created if it is not there. The timeout of zero indicates that InterSystems IRIS will try a single time to open the file and fail if it cannot.

Once the file has been opened, the code loops over the name value pairs. For each pair, the caller routine name and the entry point name are written followed in the line by the name-value pair. (If the value part is the empty string, only the name is written.) Each pair occupies one line in the log file. The first three values on each line are aligned so they appear in columns for easier scanning.

When all the pairs have been written, the log file is closed, the previous value *$IO* is restored and control returns to the caller.

2.5.2 ^%ZSTART

This routine contains the entry point actually invoked by InterSystems IRIS. It uses the services of **^%ZSSUtil** just described. All the entry points act more or less the same, they place some information in the log. The **SYSTEM** entry point has been made slightly more elaborate than the others. It places information in the Operator messages log as well.

ObjectScript

```
%ZSTART ; User startup routine.

#define ME "ZSTART"
#define BgnSet "Start"
#define Empty ""

    ; cannot be invoked directly
    quit

SYSTEM ;
    ; InterSystems IRIS starting
    new EntryPoint, Items

     set EntryPoint = "SYSTEM"

     ; record the fact we got started in the messages log
     do WriteConsole^%ZSSUtil((EntryPoint
                               _ "^%"
                               _ $$$ME
                               _ " called @ "
                               _ $ZDATETIME($HOROLOG, 3)))

    ; log the data accumulate results
     set Items = $LISTBUILD($$$BgnSet, $ZDATETIME($HOROLOG, 3),
                           "Job", $JOB,
                           "Computer", ##class(%SYS.System).GetNodeName(),
                           "Version", $ZVERSION,
                           "StdIO", $PRINCIPAL,
                           "Namespace", $NAMESPACE,
                           "CurDirPath", ##class(%File).ManagerDirectory(),
                           "CurNSPath", ##class(%File).NormalizeDirectory(""),
                           "CurDevName", $System.Process.CurrentDevice(),
                           "JobType", $System.Process.JobType(),
                           "JobStatus", $ZHEX($ZJOB),
                           "StackFrames", $STACK,
                           "AvailStorage", $STORAGE,
                           "UserName", $System.Process.UserName())
    do WriteLog^%ZSSUtil($$$ME, EntryPoint, Items)

    quit
```

ObjectScript

```
LOGIN ;
    ; a user logs into InterSystems IRIS
    new EntryPoint, Items

    set EntryPoint = "LOGIN"
     set Items = $LISTBUILD($$$BgnSet, $ZDATETIME($HOROLOG, 3))
    do WriteLog^%ZSSUtil($$$ME, EntryPoint, Items)
    quit

JOB ;
    ; JOB'd process begins
    new EntryPoint, Items

     set EntryPoint = "JOB"
     set Items = $LISTBUILD($$$BgnSet, $ZDATETIME($HOROLOG, 3))
    do WriteLog^%ZSSUtil($$$ME, EntryPoint, Items)
    quit

CALLIN ;
    ; a process enters via CALLIN interface
    new EntryPoint, Items

     set EntryPoint = "CALLIN"
     set Items = $LISTBUILD($$$BgnSet, $ZDATETIME($HOROLOG, 3))
    do WriteLog^%ZSSUtil($$$ME, EntryPoint, Items)
    quit
```

Here is an description for each label:

^%ZSTART

This routine begins with a QUIT command so that it is benign if invoked as a routine rather than beginning its execution properly at one of its entry points.

This routine also defines named constants (as macros) for its own name, a starting string and the empty string.

SYSTEM^%ZSTART

This subroutine constructs a string consisting of the calling routine name, entry point, and the date and time it was invoked. Then it calls **WriteConsole^%ZSSUtil** to place it in the operator messages log.

Afterward, it constructs a list of name-value pairs that it wishes to be displayed. It passes this to **WriteLog^%ZSSUtil** to place into the local log file. Then it returns to its caller.

LOGIN^%ZSTART, JOB^%ZSTART, and CALLIN^%ZSTART

These subroutines do not place any information in the operator messages log. Instead, they construct a short list of items, enough to identify that they were invoked, and then use **WriteLog^%ZSSUtil** to record it.

2.5.3 ^%ZSTOP

This routine contains the entry points actually invoked by InterSystems IRIS and it uses subroutines in **^%ZSSUtil**. This example is similar to the example for **^%ZSTART**. See the previous section for details.

ObjectScript

```
%ZSTOP ; User shutdown routine.

#define ME "ZSTOP"
#define EndSet "End"
#define Empty ""

    ; cannot be invoked directly
    quit
```

```
SYSTEM ; InterSystems IRIS stopping
    new EntryPoint

    set EntryPoint = "SYSTEM"
    ; record termination in the messages log
    do WriteConsole^%ZSSUtil((EntryPoint
        _ "^%"
        _ $$$ME
        _ " called @ "
        _ $ZDATETIME($HOROLOG, 3)))
    ; write the standard log information
    do Logit(EntryPoint, $$$ME)
    quit

LOGIN ; a user logs out of InterSystems IRIS
    new EntryPoint

    set EntryPoint = "LOGIN"
    do Logit(EntryPoint, $$$ME)
    quit

JOB ; JOB'd process exits.
    new EntryPoint

    set EntryPoint = "JOB"
    do Logit(EntryPoint, $$$ME)
    quit

CALLIN ; process exits via CALLIN interface.
    new EntryPoint

    set EntryPoint = "CALLIN"
    do Logit(EntryPoint, $$$ME)
    quit

Logit(entrypoint, caller) PRIVATE ;
    ; common logging for exits

    new items

    set items = $LISTBUILD($$$EndSet, $ZDATETIME($HOROLOG, 3))
    do WriteLog^%ZSSUtil(caller, entrypoint, items)
    quit
```

3

Extending Languages with ^%ZLANG Routines

You can use the **^%ZLANG** feature to add custom commands, functions, and special variables to the ObjectScript and other languages. Your extensions are invoked in the same way as standard features, and follow the same rules for syntax, operator precedence, and so on. **^%ZLANG** features generally do not execute as rapidly as standard InterSystems IRIS features. Consider this point when coding performance-critical routines.

To add such extensions:

1. Define routines with the following names, as needed:

Routine Name	Purpose
^%ZLANGC00	Defines custom ObjectScript commands.
^%ZLANGF00	Defines custom ObjectScript functions.
^%ZLANGV00	Defines custom ObjectScript special variables.

2. In these routines, define public subroutines as follows:

   - For the subroutine label, use the name of the command, function, or special variable that you are defining.

     The name must start with the letter `Z` and can include only letters. Unicode characters are permitted, if the locale defines them as alphabetic. All letters must be in uppercase, though execution is not case-sensitive. The maximum length of a name is 31 letters.

     The name cannot be the same as an existing command, function, or special variable (if it is, InterSystems IRIS ignores it). InterSystems also strongly recommends that you do not use an SQL reserved word.

     If you are defining a function, even a function with no arguments, the label must include parentheses.

     A label for a special variable can include parentheses.

   - Optionally include an additional label to define an abbreviation. Be careful that it is not already used by InterSystems IRIS.
   - Define the subroutine appropriately. See the "Notes" section for details.

3. As good programming practice, the first command in the parent routine should use QUIT so that nothing occurs if a user invokes the routine directly.

   It is also helpful for the routine to include a comment at the top that indicates the name of the routine itself.

Note: It is customary (but inaccurate) to refer to a routine as if the caret is part of its name. This documentation follows this custom.

CAUTION: For other subroutines, including ones invoked by the public subroutines, make sure that the labels for those are in lowercase or mixed case (or do not start with `Z`). Or implement them as private subroutines.

That is, because a **^%ZLANG** routine extends the language, it is important to make sure that only the desired subroutines are available outside of it.

3.1 Notes

Commands are handled like a DO of a routine or procedure. Arguments are passed as call parameters.

Your code should preserve the values of system state such as **$TEST** and **$ZREFERENCE** unless you intend for them to be a result of your code. (But note that for functions and special variables, the system automatically preserves **$TEST**.)

You can SET the value of a special variable. There is only one entry point for the variable. To determine whether to set the value or retrieve the value, your code should check whether an argument is given. For example:

ObjectScript

```
ZVAR(NewValue) public {
         if $DATA(NewValue) Set ^||MyVar=NewValue Quit
         Quit $GET(^||MyVar)
}
```

Then, a user can either set this variable or retrieve it, as demonstrated here:

```
USER>w $ZVAR

USER>s $ZVAR="xyz"

USER>w $ZVAR
xyz
```

To return an error code from a command or function, use **$SYSTEM.Process.ThrowError()**.

3.2 Examples

For example, to define a custom special variable for use in ObjectScript, you define the routine **^%ZLANGV00**, which could look like the following:

ObjectScript

```
 ; implementation of ^%ZLANGV00
 ; custom special variables for ObjectScript
  QUIT

ZVERNUM        ; tag becomes name of a special variable
ZVE
  QUIT $PIECE($ZVERSION,"(Build")
```

Then, for demonstration, you can use the new variable in the Terminal as follows:

```
USER>w $zvernum
InterSystems IRIS for Windows (x86-64) 2018.1
USER>w $zve
InterSystems IRIS for Windows (x86-64) 2018.1
```

For another example, suppose that you define the **^%ZLANGF00** routine as follows:

ObjectScript

```
 ; implementation of ^%ZLANGF00
 ; custom functions for ObjectScript
 QUIT

ZCUBE(input) public {
 Quit input*input*input
}
```

Then, for demonstration, you can use the new function in the Terminal as follows:

```
USER>w $zcube(2)
8
```

The following example shows **^%ZLANGC00**, which creates a command that executes the system status utility **^%SS**:

ObjectScript

```
 ; %ZLANGC00
 ; custom commands for ObjectScript
  QUIT

ZSS       ; tag name of a command to check system status
  DO ^%SS
  QUIT
```

4

Controlling InterSystems IRIS from a Windows Client

InterSystems IRIS® data platform provides a mechanism for Windows client programs to control an InterSystems IRIS configuration and to start up InterSystems IRIS processes. This allows you to deliver applications that automatically start InterSystems IRIS processes with the correct configuration information without requiring the standard InterSystems IRIS tools. The tools allow you to:

- Find InterSystems IRIS directories, paths, and service name for a given configuration name.
- Get the status of the InterSystems IRIS system.
- Control an InterSystems IRIS configuration directly or through the InterSystems IRIS Control Service, depending on which version of Windows is running
- Start an InterSystems IRIS process with the appropriate settings.

You can perform these actions by dynamically loading irisctl.dll and using its functions.

4.1 IRISctlGetDirs

Finds configuration, binary, and manager directory paths, and service name for a given configuration name.

Syntax

```
IRISctlGetDirs(char *config, IRISCTL_DIR_INFO *dirinfo)
```

config	The name of the desired configuration.
dirinfo	A pointer to a C structure where the directory information will be stored.

Return Values

Returns (char *0) on ERROR.

4.2 IRISctlConfigStatus

Returns the status of the InterSystems IRIS configuration.

Syntax

```
IRISctlConfigStatus(char* config)
```

config	The name of the desired configuration

Return Values

Returns a value from 0 through 4 as follows:

0	Configuration is up and running.
1	Configuration is starting or stopping.
2	Configuration startup or shutdown aborted.
3	Configuration is down.
4	ERROR

4.3 IRISctlControl

Controls an InterSystems IRIS configuration through the InterSystems IRIS Control Service on Windows.

Syntax

```
IRISctlControl(char *command, char *config)
```

command	Use one of the following commands: • start — starts a configuration • stop — shuts down a configuration gracefully • stopnoshut — shuts down a configuration without running the user-supplied shutdown routine • force — forces down a configuration; equivalent to **irisforce** on UNIX® systems • stopstart — shuts down a configuration gracefully and immediately restarts it
config	The name of the desired configuration.

Return Values

IRISCTL_SUCCESS	Operation succeeded
IRISCTL_ERROR	Generic error
IRISCTL_INVALID_COMMAND	Invalid command argument
IRISCTL_INVALID_CONFIGURATION	Undefined configuration
IRISCTL_CONTROL_STU_ERROR	**^STU** failed

Following an error return, **IRISctlGetLastError** returns a pointer to an informational error string.

4.4 IRISctlRun

Starts an InterSystems IRIS process in the indicated configuration, and namespace, using the indicated principal I/O device and invoking the indicated routine.

Syntax

```
IRISctlRun(char *config, char *routine, char *namespace, char *IOtype)
```

config	The name of the running configuration.
routine	The name of the desired routine to start.
namespace	The name of the desired namespace.
IOtype	How I/O is to be handled, with possible values of: • terminal — The process starts a new InterSystems IRIS programmer's terminal. • none — No I/O. The process runs in the background with NUL: used for *$Principal*. Writes to *$Principal* are discarded. Reads from *$Principal* produce an error.

Return Values

IRISCTL_SUCCESS	Operation succeeded
IRISCTL_ERROR	Generic error
IRISCTL_INVALID_COMMAND	Invalid command argument
IRISCTL_INVALID_CONFIGURATION	Undefined configuration
IRISCTL_CONTROL_STU_ERROR	**^STU** failed

Note: On Windows, the specified configuration must be running. If you are not sure if the configuration is running, use **IRISctlConfigStatus** and **IRISctlControl** to check and start the desired configuration. This prevents InterSystems IRIS from trying to start a configuration without using the control service.

4.5 IRISctlRunIO

Starts an InterSystems IRIS process in the indicated configuration, and namespace, using the indicated principal I/O device type, invoking the indicated routine and additional IO specifications for input, output and error devices.

Syntax

```
IRISctlRunIO(
        char *config,
        char *routine,
        char *namespace,
        char *IOtype,
        HANDLE *hIO,
        char *cwd,
        char *options,
        HANDLE *child,
        DWORD *childPID))
```

config	The name of the running configuration in all capital letters.
routine	The name of the desired routine to start.
namespace	The name of the desired namespace.
IOtype	How I/O is to be handled, which must have a value of TCP, because the process uses a TCP socket.
hIO	An array of three handles to be used as the standard input, output, and error devices of the InterSystems IRIS process.
cwd	The working directory path of the child process. If the directory argument is zero, the working directory of the current process is used.
option	Additional irisdb.exe command line options appended to the generated command line. For example, you could define a larger process memory size (-b 1024).
child	The pointer to a variable of type HANDLE, where the handle to the child process will be returned. If the value of handle is zero, the handle to the child process will be closed by this function.
childPID	The pointer to the PID of the created irisdb.exe process. This argument can be zero if the child’s PID is not required.

Return Values

IRISCTL_SUCCESS	Operation succeeded
IRISCTL_ERROR	Generic error
IRISCTL_INVALID_COMMAND	Invalid command argument
IRISCTL_INVALID_CONFIGURATION	Undefined configuration
IRISCTL_CONTROL_STU_ERROR	**^STU** failed

Note: The handles in the *hIO*array must be inheritable. Use **DuplicateHandle** to make the handle inheritable by the child process.

On Windows, the specified configuration must be running. If you are not sure if the configuration is running, use **IRISctlConfigStatus** and **IRISctlControl** to check and start the desired configuration. This prevents InterSystems IRIS from trying to start a configuration without using the control service.

5

Using ^GBLOCKCOPY for Fast Global Copies

^GBLOCKCOPY is an InterSystems IRIS routine that performs fast global copies between databases. It has two modes of operation: Interactive and Batch. Interactive mode is a single process, while Batch mode allows you to run configure and run multiple processes in parallel. **^GBLOCKCOPY** contains a built-in monitor and several reports to track the progress of global copies. You can restart **^GBLOCKCOPY** at the point it left off if there is a system failure.

Note: Because there is no locking or integrity checking for database blocks that are being copied, **^GBLOCKCOPY** should be used to copy globals only when they are *not* being actively modified . Although **Set** or **Kill** operations may be performed on *other* globals in the source database where the copy is being performed, as well as in the destination global, database, or namespace without affecting the copy, results in the destination global are unpredictable if **Sets** or **Kills** occur in the source global that is being copied to another database or namespace.

When **^GBLOCKCOPY** copies a global to a new database, it creates the global there with the same properties of the source global, including Protection, Journal attributes, Collation type, and Keep attributes.

Note: The **SYS.Database.Copy()** class method provides functionality similar to **^GBLOCKCOPY**.

5.1 Uses of ^GBLOCKCOPY

^GBLOCKCOPY can be used for several different operations as follows:

- *Copy single or multiple globals from a database to another database or namespace* — You can select one or several globals to be copied into a destination database or namespace. If the global already exists in the destination database, data from the source global is merged into the existing data.
- *Split a global from a single database into multiple databases using subscript level mapping* — By setting up a namespace with subscript level mapping (SLM) of a global, you can copy a global from a database into this new namespace and cause it to be split amongst the database which make up SLM.
- *Move a subscript-mapped global in many databases into one database* — Create a new database which contains the entire global. Then set up several copies in a batch which will copy the global from all the different SLM databases into the new database.
- *Make a copy of a database* — You may copy a database to another directory by copying all the globals to it.

- *Copy a global to another machine across ECP* — **^GBLOCKCOPY** supports copying a global across an ECP network connection to another machine. You need to set up an ECP connection to a remote machine, and a namespace mapping which points to it. Then select the "Copy from Database to Namespace" option and select the remote namespace as the destination of the copy.
- *Reclaim unused space in a database* — If a large global is created then killed in a database, there may be a large excess of unused space in the database. You can remove this space by copying all the globals in the database to a new one, and then replacing the old database with the new database.
- *Reorganize the pointers in a database* — If a database becomes fragmented because of block splits, you may want to reorganize the data in it to speed performance. You can do this by copying all the globals in a database into a new database, and then replacing the old database with the new database.
- *Change the collation of a global* — If you want to change the collation of an existing global you are copying, you can create the global in the destination database with the desired default collation before running **^GBLOCKCOPY**.
- *Import a Caché or legacy database or namespace into InterSystems IRIS* — If you have a CACHE.DAT file or legacy database file you want to import to an IRIS.DAT database or namespace, simply select the directory where it exists as the source directory for the copy. The database is renamed IRIS.DAT, and the data is available to copy to the destination database or namespace.

5.2 Running ^GBLOCKCOPY

Before you run **^GBLOCKCOPY** (or for that matter before you perform an upgrade), make a full operating system backup of your databases, and run an integrity check to ensure there is no corruption in any of the databases.

Note: To make **^GBLOCKCOPY** run faster, kill off any temporary and scratch data as well as any old data you do not require.

When running **^GBLOCKCOPY**, you can copy all or only some globals from a specific database. The routine prompts for the names of the globals to copy. To refer to only some globals, the syntax is:

- Inputting *glonam* selects the global "glonam" to be copied.
- Inputting '*glonam* deselects the global "glonam" if it is selected.
- Inputting *glonam1-glonam2* selects the range of globals from "glonam1" to "glonam2".
- Inputting '*glonam1-glonam2* deselects any global in the range from "glonam1" to "glonam2".

The following are wildcard characters for selecting globals:

- Use an ampersand (&) to match any single letter.
- Use a number sign (#) to match single digit.
- Use a question mark (?) to match any single character.
- Use an asterisk (*) to match any number of characters.

Note: You cannot use wildcard characters when specifying a range of globals.

When selecting a subsection of globals, you can also use the following inputs:

- **?** — lists informational inputs.
- **?L** — lists all available globals; specifies which have been selected with a number sign (#).

- **?S** — lists the currently selected globals.
- **?H** — lists the global selection syntax.

You can use the batch functionality of **^GBLOCKCOPY** to set up multiple operations to run at the same time. When configuring a batch, you will be prompted for the total number of copy processes and the maximum number of copy processes per directory. You can run a maximum of one process per global. For Batch mode, the duration of the copy operation is proportional to the size of the largest global; storage latency and bandwidth are additional factors. While the batch of operations is running, you can monitor progress using the Monitor or Batch Report.

Note: Users should be kept from accessing the databases while they are being processed by **^GBLOCKCOPY**. The result of the database operations are unpredictable if they are accessed while **^GBLOCKCOPY** is running. Databases on the same system which are not being processed by **^GBLOCKCOPY** can be used safely.

6

Using Switches

InterSystems IRIS switches are per-instance flags that can be used for a variety of purposes. They are especially useful for inhibiting various system processes when doing backups or trying to recover crashed systems. The **^SWSET** routine is used to directly manipulate the values of the switches.

Background

Switches in InterSystems IRIS have their genesis in the physical contacts once part of computer operator consoles or included in the front panel of microcomputers. By setting one of these switches, an operator could convey a single bit of information to the programs running on the machine at that time. Since InterSystems IRIS implements a "virtual machine", the concept of the switch for this machine has been similarly abstracted.

Today, switches in InterSystems IRIS are represented as individual bit settings in the shared, common memory of an InterSystems IRIS instance; they are visible to all InterSystems IRIS processes. While several have been set aside for users, most influence the operation of InterSystems IRIS itself.

Note: Users should view the switches as being local to an InterSystems IRIS instance. Although InterSystems IRIS itself provides mechanisms to propagate the meaning of certain settings to other members of a cluster, these are for InterSystems internal use only. The values of the user switches cannot be moved to other systems.

6.1 Currently Defined Switches

All switches are identified by number. They are initialized to zero (off) when InterSystems IRIS starts. The following table gives the switch number(s) and their effect:

Switch	Meaning / Use
0 — 7	Reserved for use by applications programs.
8	Inhibits existing InterSystems IRIS daemons from responding to network requests.
9	Inhibits the creation of new daemons to process network logins.
10	Inhibit all global access except by the process that sets this switch. Also inhibit routine accesses that causes disk IO except for this process.
11	Inhibit all global access except for the system job that sets this switch. This overrides switch 10 and is reserved for use by the system. This switch is set, for example, by the backup process to quiesce system activity before copying.

Switch	Meaning / Use
12	Inhibits the ability to login to InterSystems IRIS. Users who attempting to login will receive a message: "Sign-on and JOB inhibited: Switch 12 is set".
13	Inhibits all global SETs, KILLs and ZSAVE commands; only read access is allowed to globals and routines.
14	Inhibits all access to all globals and all routines.
15	Allow network references from peers, even if switch 10,13, or 14 would normally prevent the access.
16	Used internally by InterSystems IRIS to coordinate shutdown activity.
17	Bypass wait for completion of journal flush on clusters.
18	Inhibits pausing added processes if the queue for a block gets too long.
19	Inhibit the start of new transactions.
20 — 31	Undefined and reserved for InterSystems use.

CAUTION: Customer applications should confine any switch activity to the set reserved for applications programs (switches 0–7), except when specifically directed otherwise by InterSystems personnel or its documented procedures.

6.2 Manipulating Switches

The **^SWSET** routine is used to directly manipulate the values of the switches. In addition, other InterSystems IRIS facilities, such as those that work with journals on clustered systems and system backup, also set them on behalf of their callers.

6.2.1 Routine SWSET

This routine provides an interactive way to set the value of the switches from, for example, a terminal session.

```
SWSET
```

Parameters

None.

Remarks

When invoked as in the example below, the routine will prompt for the switch number and then prompt for the value to be set in the switch (0 or 1).

Examples

The following example demonstrates the use of **SWSET** . After executing

```
DO ^SWSET
```

the user will successively see the following:

```
Set/Clear switch #:

Set/Clear switch #: 2

Set/Clear switch #: 2 to value (0 or 1):

Set/Clear switch #: 2 to value (0 or 1): 1

Set/Clear switch #: 2 to value (0 or 1): 1...done
```

6.2.2 Function %swstat^SWSET

This function returns the current setting for the switch.

`%swstat^SWSET(switch)`

Parameters

- *switch* — The number of the switch.

Remarks

There are three possible return values:

- 0 — Indicates the switch was not set to its intended value.
- 1 — Indicates the switch was properly set to its new intended value.
- -1 — Indicates the switch was set to an impossible value (something other than 0 or 1).

Examples

The following example prints the value of switch number 1.

```
Write $$%swstat^SWSET(1)
```

6.2.3 Function %swset^SWSET

This function sets the switch to the specified value.

`%swset^SWSET(switch, value)`

Parameters

- *switch* — The number of the switch.
- *value* — The value it should have, 0 or 1.

Remarks

If the switch is a valid number and value is either a 0 or 1, this function sets the switch to that value and returns:

- 0 — the switch is now reset (off)
- 1 — the switch is now set (on)

otherwise it returns a value of –1 indicating that an error has occurred.

Examples

The following example sets the value of switch number 1 to off.

```
Write $$%swset^SWSET(1, 0)
```

6.3 Failure Modes

An InterSystems IRIS process which sets one of the system-reserved switches and terminates without properly cleaning up its work can leave the system in a restricted operating mode. For example, a process that sets switch 12 and then suffers a catastrophic failure (or even merely HALTs) will leave InterSystems IRIS in a state where no further users can login. If this situation occurs, the administrator or operator is urged to call the InterSystems Worldwide Response Center (WRC).

Note: The only situation for which InterSystems IRIS implements an automatic recovery is for switch 10. If a process sets this switch and then HALTs, InterSystems IRIS will automatically reset the switch to zero.

7

Notes on Managing InterSystems IRIS via Routines

The Management Portal provides a convenient, browser-based interface for controlling the system. However, to cover those scenarios when the system cannot be managed from a browser, InterSystems IRIS provides interactive routines that perform the same tasks. When run in the Terminal, these routines act as command-line interfaces. Note that within legacy documentation, these routines are referred to as *character-based routines*. These routines are described elsewhere in detail; the purpose of this page is to provide additional information that applies to all these routines.

CAUTION: There is nothing to prevent multiple instances of the same routine from being executed at the same time by different system administrators (or even the same administrator). If this happens, it is the responsibility of the administrators to coordinate their activity to avoid conflicts and achieve their objectives with regard to the coherence of the affected data.

Important: Administrators who elect to use the routines described in the documentation are assumed to have a detailed operating knowledge of how InterSystems IRIS works and what parameter values are appropriate for the options they choose.

To use any of these routines from the Terminal, you must be in the `%SYS` namespace and must belong to at least the **`%Manager`** role. To start a routine, use the **DO** command. For example, to invoke the **^LEGACYNETWORK** routine (a routine that supports configuration of legacy networking tools), use the following command:

ObjectScript

```
DO ^LEGACYNETWORK
```

Upon start, each of these routines first presents you with a list of options. Select an option by entering its number after the "Option?" prompt. In most cases, the choices on the initial menu choice lead to further menus, or to requests for information until the routine has sufficient information to accomplish its task.

Note the following additional points:

- Each option has a numeric prefix. Select an option by typing its number. The option-number pattern is used throughout the routines.
- All option lists have an item to exit this level of menu and return to the previous level. You may also reply to the "Option?" prompt with **Enter**. This is interpreted as if you had chosen the "Exit" option, that is, you are finished with that section and you are presented with the next "upper" level of options. An **Enter** reply to the top-level of options exits the routine.

- Many of the prompts for information have a default value which is selected by typing the **Enter** key. When there is a default value available, it is shown after the prompt message and followed by the characters “=>” as in

  ```
  Unsuccessful login attempts before locking user? 5 =>
  ```

  where the default value is 5 for the number of times a user may try to login and fail before the system locks their username.
- Prompts whose defaults are “Yes” or “No” also accept any matching partial response such as “yE” or “n”. The match is done ignoring the case of the response.
- In options whose intent is to alter the characteristics of existing user, roles, services, and so on, the existing value of the item is displayed as the default. Typing **Enter** preserves that value and moves on to the next prompt.
- Some prompts ask for a pattern to use when matching items such as usernames. The default pattern is usually “*” that matches all items. In such patterns the asterisk matches any sequence of characters, much like it does in DOS. A pattern may also consist of a comma-separated list of items each of which is treated as its own pattern. An item is treated as being selected if it matches any pattern in the list.

8

Process Management

All InterSystems IRIS programs and applications are run by a process. A process is identified by a system-generated integer process ID (referred to as a pid).

A process can be either a foreground (interactive) process, or a background (non-interactive) process. A background process is initiated in ObjectScript using the JOB command. The process issuing the **JOB** command is known as the parent process, the initiated background process is known as the child process. A background process may also be referred to as a “jobbed process” or a “spawned process”.

Processes hold and release shared resources using locks. In ObjectScript a process acquires and releases locks using the LOCK command. To view the current locks, refer to the system-wide lock table, described in Lock Management.

This topic describes process batch mode and process priority. For other aspects of process management, refer to the %SYS.ProcessQuery class.

Note: InterSystems IRIS provides optimal defaults for all aspects of process management. Changing these process defaults should be done with caution, and only for specific special circumstances.

8.1 Batch Mode

A process executes either in the default mode, sometimes referred to as Interactive Mode (this is unrelated to user interaction), or in Batch Mode. Batch Mode should be used only for special circumstances where no better tool is available.

A process that will access large portions of a database may be set as a Batch Mode process to limit its impact on other (non-batch) processes running on the system. Specifically, a Batch Mode process is prevented from overwhelming the database cache with the database blocks that it reads or modifies. For example, a data compaction utility might appropriately run in Batch Mode.

A Batch Mode process is not the same as a jobbed process. A jobbed process is generated using the **JOB** command. A jobbed process is a non-interactive process running in background.

You can use the **%SYSTEM.Process.BatchFlag()** method to establish the current process as executing in batch mode. **BatchFlag()** takes a Boolean argument: 0=direct mode (the default), 1=batch mode. **BatchFlag()** with no argument returns the current mode setting.

8.2 Priority

Process priority determines how multiple concurrent processes compete for CPU resources. A process with a higher priority has preferential access to CPU time.

Priority is an integer value. The range of integer values is platform-dependent, as follow: Windows has a priority range of 1 to 15, with NORMAL=8. UNIX® has a priority range of -20 to 20, with NORMAL=0. Commonly, only three integer values are used: LOW, NORMAL, and HIGH. NORMAL priority is the default for user processes. NORMAL priority uses load balancing to adjust CPU usage.

You can determine the current priority of a process using the *Priority* property of the %SYS.ProcessQuery class.

Note: Changing the priority of a process is almost never needed and can compromise system stability.

8.2.1 The SetPrio() Method

You can use the **%SYSTEM.Util.SetPrio()** method to change the priority for the current process or another process identified by pid. You specify a positive or negative integer to increment or decrement the current priority by that amount. On Windows, the minimum priority is 1; attempting to decrement a priority to less than 1 sets priority to 1. NORMAL priority is 8. The maximum priority is 15. **SetPrio()** returns the new priority level set. (Note that attempting to increment past 15 may reset priority to 1 or have other unpredictable results. A **SetPrio()** return value of >15 does not reflect the actual priority set.)

8.2.2 The ^%PRIO Utility

You can use the older ^%PRIO utility to determine the priority for the current process and to set the priority to a LOW, NORMAL, or HIGH value.

To make calls and changes using ^%PRIO, you must have access to the **`%DB_IRISSYS`** resource with write permissions.

To determine the current priority, you can issue the following command: `DO ^%PRIO`. A subsequent argumentless **WRITE** command will return the current priority as follows: `%PRIO=8`.

To set the priority of the current process, issue one of the following commands: `DO LOW^%PRIO`, `DO NORMAL^%PRIO`, or `DO HIGH^%PRIO` (commands are case-sensitive). The set priority values returned (on Windows) are: LOW=4, NORMAL=8, HIGH=13.

%SYSTEM.Process.BatchFlag() changes the current process priority. However, if a process is in Batch Mode (1), using the above commands to raise the current process priority changes the process mode from Batch Mode (1) to Direct Mode (0).

To set both the Batch Mode and the priority for the current process, specify DO SET^%PRIO(priority,mode), where priority,mode is a case-sensitive string, with priority=LOW, NORMAL, or HIGH, and mode=BATCH or NOBATCH. You can specify either one or both of the comma-separated options. For example: `DO SET^%PRIO("LOW,BATCH")`. This command allows you to change the process priority without affecting the mode status of a Batch Mode process.

8.2.3 Jobbed Process Priority

The **JOB** command provides an option to set the priority for a jobbed (background) process. If not specified, the child process takes the parent process base priority adjusted by a system-defined job priority modifier. For example, on Windows a parent process with NORMAL priority (priority=8) initiates a child process with a priority=7, due to the job priority modifier. A jobbed process with a priority of HIGH competes on an equal basis with interactive processes for CPU resources.

9

Using cvendian for Byte Order Conversion

This page describes a utility to convert the byte order of an InterSystems database for migration between Big-endian and Little-endian platforms. It also provides an option to report on the byte order of a given database.

9.1 Introduction to cvendian

InterSystems provides a utility to convert the byte order of an InterSystems database from Big-endian (that is, most-significant byte first) to Little-endian (that is, least-significant byte first), and vice versa. It is called **cvendian**, for convert *endian*. This is useful when moving a database among platforms of the two types. It also provides an option to report on the byte order of a given database.

For information about the Endianness of supported platforms, see Platform Endianness.

Important: You cannot use this utility on a mounted database.

9.2 Location of Utility

The **cvendian** utility is the file *install-dir*\Bin\cvendian.exe.

9.3 Conversion Process

You can run **cvendian** on either the system that has the files to be converted or the system that will be using the converted files.

For example, to convert a database from a Little-endian to a Big-endian system, you can perform the conversion on the Little-endian system and then transfer the database to the Big-endian system, or you can transfer the file first, and then convert it.

Note:

- The cvendian utility uses simple buffered I/O to read and write the target file. For best performance, ensure that operating system (OS) file-system caching is not disabled.
- cvendian does not work for backup and journal files. You must restore databases on a platform of the same endian, move the restored databases to the different endian platform, and then use the cvendian utility to convert the databases.

To convert a database, the process is:

1. Make a copy of your database files, because the utility replaces the source files with the converted files.
2. Run **cvendian** using the syntax described in the Utility Syntax section.

9.4 Utility Syntax

With the **cvendian** endian utility, you can specify the desired byte order, or you can report the current byte order without conversion. Use the following syntax:

```
cvendian [-option] file
```

The *option* argument is one of the following:

- `-big` — convert the database to Big-endian
- `-little` — convert the database to Little-endian
- `-report` — report the byte order of the database

You can shorten the options to their initial letter. If this is a conversion request (*-big* or *-little*), and the database already has the specified byte order, the utility displays a warning message and stops processing.

If you do not provide the *option* argument, the utility converts the database from the existing byte order to the other byte order. It is recommended, however, that you use the *option* argument.

The *file* argument is the file to convert, and can include a complete pathname.

The utility performs the following actions:

- Auto-detects the byte order of the database
- Displays endian information and other information
- Performs the conversion
- Displays a message indicating success or failure

For example, suppose you are converting a database for use on IBM AIX® for Power System-64 from Windows Server 2016. This means you must convert from Little-endian (for Intel) to Big-endian (for POWER). The output from running **cvendian** on the Windows system before moving the file to the AIX system looks similar to this:

```
C:\IrisSys\Bin>cvendian -big c:\temp\powerdb\iris.dat

This database is little-endian.
This database has a block size of 8192 bytes.

This database has 1 volume and 1 map.
The last block in the primary volume is 18176.

Original manager directory is c:\temp\powerdb\

No extension volumes.

Done converting c:\temp\powerdb\iris.dat to big-endian

C:\IrisSys\Bin>
```

You can now move the converted database file to the AIX system.

10

Converting FileMan Files into InterSystems IRIS Classes

Note: This feature is available and fully supported only for existing users. InterSystems recommends against starting new projects with FileMan.

InterSystems IRIS® data platform provides a utility to convert applications based on FileMan files into InterSystems IRIS classes, thus providing object and SQL access to the data. Specifically, this utility (the FileMan Mapping Utility) generates InterSystems IRIS class definitions that map the data. This page assumes that the FileMan globals (*^DD* and *^DIC*) are already loaded into the desired namespace in your system.

10.1 Overview of the FileMan Mapping Utility

FileMan enables you to quickly and easily create custom class mappings for your FileMan files. The classes created by the mapping utility enable you to access the file data via object access and via SQL. You can map just one, many, or all FileMan files in a namespace. For each file, you can control details such as the following:

- Which fields are mapped
- Whether the classes are read-only or have write access
- Formats for the names of the generated classes and properties (tables and fields)
- The superclass list for the generated classes
- Whether word-processing fields are mapped as list collections or as child tables

10.1.1 System Requirements

You need a working FileMan installation.

10.2 Specifying and Modifying the Default Settings

FileMan uses many settings that control how it generates classes. It is worthwhile to review these settings, modify them to values that you use most of the time, and save these as the default settings.

To specify or modify the default settings, use a command like the following:

```
Set ^%SYS("sql","fm2class",setting) = value
```

Where *setting* is an internal setting name as shown in the previous list, and *value* is the value to assign to it.

When you use the methods in **$SYSTEM.OBJ.FM2Class**, you can pass in an array of the following format:

```
arrayName(setting) = value
```

For example:

```
set fmsettings("display")=1
set fmsettings("package")="VISTA"
set fmsettings("superClasses")="%XML.Adaptor"
set fmsettings("wpIsList")= 1
```

The available settings are as follows:

childTableNameFormat

Specifies the format of the generated child table names. For this setting, specify a string that uses the following keywords along with any characters that are valid to use as table names:

- `<FILENAME>` — Replaced with the name of the FileMan file
- `<FILENUMBER>` — Replaced with the file number
- `<PARFILENAME>` — Replaced with the name of the parent file
- `<PARFILENUMBER>` — Replaced with the file number of the parent file

If you use `<FILENAME>` or `<PARFILENUMBER>`, any decimal place characters in the file number are converted to underscore characters.

Some examples of this setting:

- `SUB_<FILENAME>` — In this example, the table name of a child table is the string `SUB_`followed by the name of the file. For example: `SUB_ACCESSIBLE_FILE`
- `f<PARFILENUMBER>c<FILENUMBER>` — In this example, the table name of a child table is the string `f`, followed by the number of the parent file, followed by `c`, followed by the number of this file. For example: `f200c200_032`
- `<FILENAME>` — In this example, the child table name is simply the same as the filename.

ClassParameters

Specifies class parameters and values for an OS object, in JSON format.

compile

Specifies whether to compile the classes after creating them. Permitted values are as follows:

- 1 — Compile the classes.
- 0 — Do not compile the classes.

compileQSpec

Specifies any class compiler qualifiers and/or flags.

dateType

Specifies the data type to use when mapping FileMan DATE fields. The default is %Library.FilemanDate.

datetimeType

Specifies the data type to use when mapping FileMan DATE/TIME fields. The default is %Library.FilemanTimeStamp.

deleteQSpec

Specifies any class deletion qualifiers and/or flags.

display

Controls how the results are displayed. Permitted values are as follows:

- 0 — No display.
- 1 — Minimal display.
- 2 — Full display. This is the default.

expandPointers

Specifies whether the utility creates additional computed properties to expand the pointer field. Permitted values are as follows:

- 0 — The utility does not add any computed properties to expand the pointer field.
- 1 — The utility adds a computed property that expands the pointer field and is equal to the NAME (.01) field in the referenced file.
- 2 — The utility adds a computed property that expands the pointer field and is equal to the NAME (.01) field in the referenced file. The naming conventions used for this option is the pointer field will be named after the referenced file with “_ID” appended to the field name.
- 3 – This option is the same as 2 except that the name of the reference field will be `<Field>_f<pointed-to-file-id#>ID`.

expandSetOfCodes

Specifies whether to define an expanded SOC (SetOfCodes) table for each Set Of Codes field that is mapped. Permitted values are as follows:

- 1 — For each Set of Codes field that is mapped to a class, the utility will generate a read-only class/table that maps to the CODE and MEANING of the Set Of Codes. The name of the table is <tablename>_SOC_<field-name>. The SOC table has two fields. CODE maps to the name of the CODE and MEANING is the external meaning for the CODE. This table is mapped directly to the ^DD global, so any updates to the definition in the ^DD global will be immediately reflected in the data returned by the table.
- 2 — These SOC class/tables will not be created. This is the default.

extendedMapping

Specifies a string to be inserted into the global name in the map definitions for extended mapping purposes. For example, you might specify `["SD"]` and your global is mapped as `^["SD"]LR(...)` instead of `^LR(...)`.

This can be any valid string that can be used for extended global mapping and must include the `[...]` or `|...|` brackets.

`fieldNameFormat`

Specifies how the utility will generate the SQL field name. Permitted values are as follows:

- `Exact` — The SQL field name will use exactly the same case as the FileMan field name. For example, FileMan field 'DEA EXPIRATION DATE' becomes SQL field name 'DEA_EXPIRATION_DATE'.
- `Upper` — Letters in the FileMan field name are folded to upper case to generate the SQL field name. For example, FileMan field 'DEA expiration date' becomes SQL field name 'DEA_EXPIRATION_DATE'.
- `Lower` — Letters in the FileMan field name are folded to lower case to generate the SQL field name. For example, FileMan field 'DEA EXPIRATION DATE' becomes SQL field name 'dea_expiration_date'.
- `Pascal` — The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. Also, spaces and underscored are removed. For example, FileMan field 'DEA EXPIRATION DATE' becomes the SQL field name 'DeaExpirationDate'.
- `Camel` — Thee first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. Also, spaces and underscored are removed. For example, FileMan field 'DEA EXPIRATION DATE' becomes the SQL field name 'deaExpirationDate'.

`ienFieldName`

Specifies the name of the IEN field, which is `IEN` by default.

`JSONAdaptor`

Enables JSON Adapter functionality for use with mapped classes. The value of `JSONAdaptor` is a JSON-format string of class parameters and values. Permitted parameters are as follows:

- `%JSONENABLED`
- `%JSONIGNOREVALIDFIELD`
- `%JSONNULL`
- `%JSONIGNORENULL`
- `%JSONREFERENCE`
- `ComputedAlwaysJSONINCLUDE`

For details about `JSONAdaptor` parameters, see %JSON.Adaptor in the class reference.

`ComputedAlwaysJSONINCLUDE` allows specification of the `%JSONINCLUDE` property parameter for properties defined as `Calculated` and `SqlComputed`.

For best results, the following settings are recommended:

- Set `%JSONEREFERENCE` to `ID`.
- Set `ComputedAlwaysJSONINCLUDE` to `OUTPUTONLY`.

`logFile`

Specifies the name of the file into which the utility should log output. Type a filename.

`nameLength`

Specifies the maximum length of property names, foreign key names, and trigger names produced by this utility. The default is 180, which corresponds to an increase in the length of InterSystems IRIS class member names in a recent release. Use this option if you want to keep the names of these items at the previous shorter maximum (31 characters).

`owner`

Specifies the username to use as the owner of the classes created. The default is the current value in *$Username*.

`package`

Specifies the name of the package to create the classes in.

`readonly`

Specifies whether the generated classes should be read-only. Permitted values are as follows:

- 1 — Generate read only classes.
- 0 — Generate read/write classes.

`recursion`

Controls whether sub-files and pointer are also mapped. Permitted values are as follows:

- 0 — No recursion. Only this file is mapped. No sub-files or pointers are mapped.
- 1 — Partial recursion. The file is mapped, along with one level of sub-files and pointers.
- 2 — Full recursion. The file is mapped, along with all sub-files and pointers. This is the default.

`requiredType`

Controls which required FileMan fields are defined as required properties. Permitted values are as follows:

- 1 — FileMan fields marked as required are defined as required properties. This is the default.
- 0 — Only FileMan fields defined as Required Identifiers are defined as required properties.

`retainClass`

Specifies whether to recreate the entire class if it already exists. Permitted values are as follows:

- 0 — The utility deletes and recreates the class, which means that SQL privileges and any add-ons to the class are lost.
- 1 — The utility recreates the properties, storage, indexes, foreign keys, and so on, rather than the entire class. Notes:

Note:

- You must have run the FM2Class utility at least once with the InterSystems IRIS 2010.2 or higher before manually adding your own items to the class definition; this is necessary because earlier versions do not save the required metadata needed for this feature.
- After mapping a class, if you move the class to another namespace or another system, and attempt to map it again from the new location with the `RetainClass` set to 1, you must also manually move/copy the ^oddFMD global too, because this global stores the metadata required by the `RetainClass` setting.
- The `RetainClass` setting is not meant to work with SOC classes produced when you set `expandSetOfCodes` to 1; that is, if you add your own parameters, properties, indexes, and so on to these classes, those changes are not retained, even if `RetainClass` is set to 1.

`setOfCodesEnum`

Specifies the data type to use when mapping a Set Of Codes field. The options are %Library.EnumString (the default) and %Library.String.

Using %Library.EnumString provides an advantage because it provides the **OdbcToLogical()** and **LogicalToOdbc()** methods, which allow you to use the meaning of the Set Of Codes, rather than the code value from xDBC client applications.

`strictData`

Specifies whether the generated classes include *STRICTDATA*=1 in the definitions of any properties of type %Library.FilemanDate and %Library.FilemanTimeStamp.

This *STRICTDATA* parameter affects the **LogicalToOdbc()** and **LogicalToDisplay()** methods for these data types. When *STRICTDATA*=0, the default, the methods will contain the same code they did previously. When *STRICTDATA*=1, the code in the **LogicalToFormat()** methods that handles invalid Logical date and time values is removed. Use *STRICTDATA*=1 if your database contains invalid or incomplete date or time values, and you do not want assumptions made about what the correct data should be.

`superClasses`

Specifies the superclass list for each of the mapped classes. Specify a string with a comma-separated list of class names.

`tableNameFormat`

Specifies the format of the generated table name. For this setting, specify a string that uses the following keywords along with any characters that are valid to use as table names:

- `<FILENAME>` — Replaced with the name of the FileMan file
- `<FILENUMBER>` — Replaced with the file number

If you use `<FILENAME>`, any decimal place characters in the file number are converted to underscore characters.

See the examples for `childTableNameFormat`.

`variablePointerValueField`

Specifies how to handle variable pointer fields. Permitted values are as follows:

- 1 — For each variable pointer field mapped, FM2Class will also create a computed property that expands the variable pointer field value and is equal to the .01 field in the referenced file. This field will be named <Variable_Pointer_Field_VALUE>.

For example, an FM variable pointer field WHO is defined to point to either the EMPLOYEE file or the PATIENT file. By default, FM2Class creates two fields, one that points to the EMPLOYEE file and another one that points to the PATIENT file. Only one of these fields will be non-NULL. If WHO is an EMPLOYEE, the WHO_EMPLOYEE field will contain the ID/IEN of the EMPLOYEE. If WHO is a PATIENT, WHO_PATIENT will contain the ID/IEN of the PATIENT.

FM2Class *also* creates a third field, WHO_VALUE, which computes to the .01 field of the EMPLOYEE record if the WHO points to an EMPLOYEE, or the .01 field of the PATIENT record if WHO points to a PATIENT.

- 0 — FM2Class does not create this third field.

`wpIsList`

Specifies how to map word-processing fields. Permitted values are as follows:

- 0 — Convert as child tables.
- 1 — Convert as list collections.
- 2 — Convert as both list collections and child tables.
- 3 — Convert as a single long string field.

10.3 Mapping FileMan Files

Use methods in %SYSTEM.OBJ.FM2Class, as follows:

- To map all FileMan files in the current namespace:

  ObjectScript

  ```
  Do $SYSTEM.OBJ.FM2Class.All(.fmSettings, .classCount)
  ```

- To map one FileMan file:

  ObjectScript

  ```
  Do $SYSTEM.OBJ.FM2Class.One(fileNumber,.fmSettings,.fmFields,.classCount)
  ```

- To map some of FileMan files in the current namespace:

  ObjectScript

  ```
  Do $SYSTEM.OBJ.FM2Class.Some(fileList,.fmSettings,.fmFields,.classCount)
  ```

The arguments are as follows:

- *fmSettings* is an optional array passed by reference with any settings you would like the utility to use, as described earlier in this section.
- *classCount* is optional and is also passed by reference. It returns the number of classes created by the mapper utility.
- *fileNumber* is the FileMan file number of the file you want to map.

- *fmFields*, if defined, limits the fields in the file that are mapped. This is an array of the form `fmFields(`*file-number*`,`*field-number*`)`. Any required fields and fields defined in this array are mapped in the class definition. If this array is empty or not defined, all fields in the file are mapped. This array is passed by reference.
- *fileList* is the FileMan file numbers of the files you want to map. Specify a comma-delimited list of file number or ranges of file numbers. Or specify an array of file numbers passed by reference.

The class %SYSTEM.OBJ.FM2Class also provides the methods **Version()** and **GetVersion()**.

10.4 Next Steps

After using the conversion utility, you should verify that you have SQL access to the generated classes. To do so, use Management Portal. At a minimum, check that you can do the following:

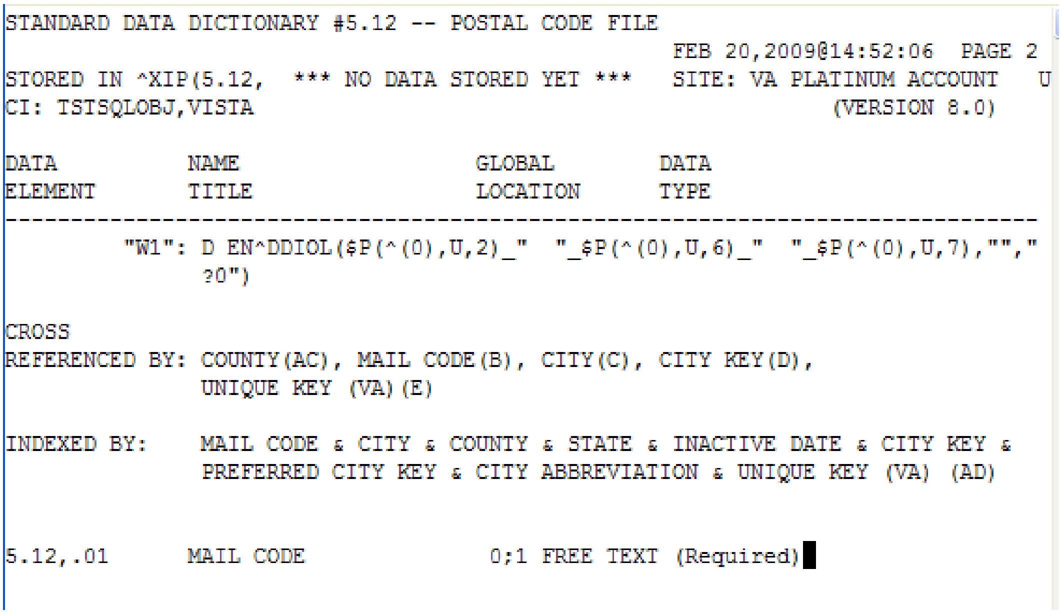
- Browse the schema.
- Execute SQL queries against the new classes.

It may also be necessary to create global and routine mappings in the namespace. Check the routines used in your generated classes to be sure that all of them are available in this namespace.

10.5 Comparison of a FileMan File to an InterSystems IRIS Class Definition

To orient yourself, you may find it helpful to compare a simple FileMan file to the resulting InterSystems IRIS class definitions.

Consider the simple example of a file called `PostalCode`. If we look at the file attributes for this file (via the FileMan Data Dictionary Utility), we see the cross-references and indexes for this file. For example:



```
STANDARD DATA DICTIONARY #5.12 -- POSTAL CODE FILE
                                                   FEB 20,2009@14:52:06  PAGE 2
STORED IN ^XIP(5.12,  *** NO DATA STORED YET ***   SITE: VA PLATINUM ACCOUNT   U
CI: TSTSQLOBJ,VISTA                                            (VERSION 8.0)

DATA          NAME                  GLOBAL        DATA
ELEMENT       TITLE                 LOCATION      TYPE
-------------------------------------------------------------------------------
         "W1": D EN^DDIOL($P(^(0),U,2)_"  "_$P(^(0),U,6)_"  "_$P(^(0),U,7),"","
               ?0")

CROSS
REFERENCED BY: COUNTY(AC), MAIL CODE(B), CITY(C), CITY KEY(D),
               UNIQUE KEY (VA)(E)

INDEXED BY:    MAIL CODE & CITY & COUNTY & STATE & INACTIVE DATE & CITY KEY &
               PREFERRED CITY KEY & CITY ABBREVIATION & UNIQUE KEY (VA) (AD)


5.12,.01      MAIL CODE              0;1 FREE TEXT (Required)
```

The same utility also shows us the definitions of the fields in this file. In this case, the titles of the fields are as follows (when listed alphabetically)

- `City`
- `City Abbrevation`
- `City Key`
- `County`
- `Inactive Date`
- `Mail Code`
- `Preferred City Key`
- `State`
- `Unique Key (VA)`

The FileMan Data Dictionary Utility also shows that the `PostalCode` file includes two pointers:

- A pointer to the `County Code` file
- A pointer to the `State` file

If we map this file, five classes are created. If we had specified `TEST` as the package name, the five classes would be as follows:

- `TEST.COUNTY`
- `TEST.COUNTYCODE`
- `TEST.POSTALCODE`
- `TEST.STATE`
- `TEST.ZIPCODE`

The class definition for `TEST.POSTALCODE` is too long to show, but the details are summarized here:

- The *DATECREATED* class parameter is set equal to the time when the class was created or updated.
- The *FILEMANFILENAME* parameter is set equal to `"POSTAL CODE"` and the *FILEMANFILENUMBER* parameter is set equal to `5.12`. These two parameters indicate the FileMan file from which this class was mapped.
- The class includes the following properties:
  - CITY
  - CITYABBREVIATION
  - CITYKEY
  - COUNTY
  - IEN
  - INACTIVEDATE
  - MAILCODE
  - PREFERREDCITYKEY
  - STATE
  - UNIQUEKEYVA

The definitions of these properties are derived from the definitions of the corresponding fields. Also note that the FileMan Internal Entry Number is represented explicitly as a property.

Descriptive text is carried over and is used as comments. For example:

```
/// FileMan Field Label: 'STATE'  FileMan Field Number: '3'
/// This field contains a pointer to the State File to represent the state
/// associated with this Postal Code.
Property STATE As TEST.STATE [ SqlColumnNumber = 6, SqlFieldName = STATE ];
```

Several of these properties (like this one) are references to the other generated classes. This representation addresses the cross-references defined in the `PostalCode` file.

- This utility creates foreign key constraints for each pointer field created. The foreign key constraint simply references the ID of the referenced table.

  In this example, the class includes the following foreign keys to represent the pointer references in the `PostalCode` file:

  - **FKeyCOUNTY(COUNTY)**, which references TEST.COUNTYCODE()
  - **FKeySTATE(STATE)**, which references TEST.STATE()

  Also see the section “Mapping of Variable Pointer Fields,” later in this section.

- The class includes one index:

  ```
  Index IDKeyIndex On IEN
  ```

- Finally, the class includes three triggers that act when filing is performed through the FileMan filer:

  - **BeforeDeleteFiling**
  - **BeforeInsertFiling**
  - **BeforeUpdateFiling**

10.6 Mapping Details

This section contains notes about how the methods in **$SYSTEM.OBJ.FM2Class** generate InterSystems IRIS class definitions.

- How the utility maps FileMan datatype fields
- How the utility maps variable pointer fields
- How the utility maps new-style cross-references
- How the utility maps cross-references that use DUZ(9)

10.6.1 Mapping of FileMan Datatype Fields

The FileMan mapping utility supports conversion of certain field types, as follows:

- BOOLEAN
- LABEL REFERENCE
- TIME

- YEAR
- UNIVERSAL TIME
- FT POINTER
- FT DATE
- RATIO

10.6.1.1 Supporting InterSystems IRIS Datatype Classes

The following InterSystems IRIS datatype classes exist to enable support of certain field types:

- %Library.FilemanTime
- %Library.FilemanYear
- %Library.FilemanTimeStampUTC

Note: The conversion methods of %Library.FilemanTimeStampUTC assume a full FileMan 22.2 run-time environment is installed, and that the DUZ(2) variable is defined and references ro INSTITUTION that includes defined COUNTRY and LOCATION TIMEZONE values.

The main purpose of these datatype classes is to provide *internal* (logical) and *external* (display/ODBC) values for the type. They are not set up with the SQL engine to be compatible with other date/time/timestamp types. For example, %Library.FilemanTime will not compare properly with a %Time type in a query, because the logical values of the two types are different. The logical value for these three types are strings, and they use string collation t force the compared value to be a string.

If you want to do something like compare a %Library.FilemanTime field with the current time, you must first convers the FileMan time field to a %Time format so the comparison is valid. An example of how to do this is:

```
SELECT ... FROM ... WHERE CAST(%external(FM_TIME_FIELD) AS TIME) > CURRENT_TIME
```

10.6.1.2 Supported FileMan Field Types

`BOOLEAN` Fields

A field defined as a `BOOLEAN` data type can have only two entry choices:

Table 10–1:

Internal	*External*
1	YES
0	NO

`LABEL REFERENCE` Fields

A field defined as a `LABEL REFERENCE` data type is designed to store a tag and routine entry of the format, *<TAG>*^*<ROUTINE>*. It is stored as a free-text field.

`LABEL REFERENCE` fields map to InterSystems IRIS as %Library.String.

`TIME` Fields

A field defined as a `TIME` data type can accept many of the date/time entries, but only stores the time portion. For example:

- *Internal* — 150943
- *External* — 15:09:43

`TIME` fields map to InterSystems IRIS as %Library.FilemanTime.

`YEAR` Fields

A field defined as a `YEAR` data type can accept many of the date entries, but only stores the year portion. For example:

- *Internal* — 3160000
- *External* — 2016

`YEAR` fields map to InterSystems IRIS as %Library.FilemanYear.

`UNIVERSAL TIME` Fields

A field defined as a `UNIVERSAL TIME` data type can accept many of the date/time entries. It stores the data/time in a format with the local time, including an indicator showing the offset from Universal Time. The first 14 characters of the internal storage of the `UNIVERSAL TIME` data type are exactly like the current `DATE/TIME` data type that includes seconds. The three characters in positions 15, 16, and 17 indicate the UTC time offset in 5–minute increments. For example:

- *Internal* — 3160106.080336440
- *External* — JAN 6,2016@08:03:36 (UTC-5:00)

In this example, (440–500)/12=-5. This is a negative five hour offset from UTC.

`UNIVERSAL TIME` fields map to InterSystems IRIS as %Libary.FilemanTimestampUTC.

`FT POINTER` Fields

A field defined as a `FT POINTER` data type works like the `POINTER` data type, but internally stores the free text that was returned from the pointed-to value. For example:

- *Internal* — PATCH,USER
- *External* — PATCH,USER

`FT POINTER` fields map to InterSystems IRIS as %Library.String.

`FT DATE` Fields

A field defined as a `FT DATE` data type works like the `DATE/TIME` data type, but internally stores the free text that was input by the user to determine the date. For example:

- *Internal* — T-1
- *External* — T-1

`FT DATE` fields map to InterSystems IRIS as %Library.String.

`RATIO` Fields

A field defined as a `RATIO` data type is designed to accept two numbers with a colon character (`:`) between the two numbers. It is formatted and stored like a mathematical ration. For example:

- *Internal* — 1:14

- *External* — 1:14

`RATIO` fields map to InterSystems IRIS as %Library.String.

10.6.2 Mapping of Variable Pointer Fields

Each variable pointer field in the file that is mapped to a InterSystems IRIS class defines a property in the class that is marked with the SqlComputed keyword. For each file to which this variable pointer field refers, an additional property is created in the class, and this property is marked with the SqlComputed and Calculated keywords. If *VariablePointerFieldName* is the name of the variable pointer field, and *PointerFileName* is the name of the file to which it points, the added property is *VariablePointerFieldNamePointerFileName*; the SQL field name for this property is *VariablePointerFieldName_PointerFileName*.

For example, suppose the file `ABC` includes a variable pointer field called `VP`. The `VP` field can point to the `Red`, `White`, or `Blue` files. This creates the following properties in the class definition:

Property	SqlFieldName	Details
`VP`		SQL computed upon INSERT and UPDATE. Stored in the map definition. Contains the internal value for the variable pointer field. Usually something like "41;DIC(40.7,", which means it points to row 41 in the 40.7 file.
`VPRed`	`VP_Red`	A computed field that is null unless the `VP` field for this row points to the `Red` file. To update the value of `VP` via SQL INSERT or UPDATE to point to a row in the `Red` table/file, set the value of `VP_Red` to the ID of that row.
`VPWhite`	`VP_White`	A computed field that is null unless the `VP` field for this row points to the `White` file. To update the value of `VP` via SQL INSERT or UPDATE to point to a row in the `White` table/file, set the value of `VP_White` to the ID of that row.
`VPBlue`	`VP_Blue`	A computed field that is null that is NULL unless the `VP` field for this row points to the `Blue` file. To update the value of `VP` via SQL INSERT or UPDATE to point to a row in the `Blue` table/file, set the value of `VP_Blue` to the ID of that row.

In all cases, the `VP` field is computed and stored automatically.

In addition to defining the three reference fields `VP_Red`, `VP_White`, and `VP_Blue`, the utility also creates three foreign key constraints in the ABC class for these references. This is done to ensure referential integrity is maintained between the files.

10.6.3 Mapping of New-Style Cross-References

This utility converts new-style cross-references to index maps if the cross-reference is defined by simple set/kill logic. This allows the InterSystems IRIS query optimizer to choose a new-style cross-reference index (if one exists) and increase query performance in some cases.

10.6.4 Mapping of Cross-References That Use DUZ(9)

FM2Class maps a cross-reference (original or new style) that was previously not mapped because the cross-reference subscripts included a reference to the variable DUZ(9) or the subscript had the form +$G(varname), where *varname* represents any variable reference.

If you have such cross-reference definitions, make sure that the DUZ(9) variable, or possibly any variable referenced by +$G(varname) is defined in any process running SQL queries against tables with indexes mapped from these cross-references. To set DUZ(9) for a process connecting via xDBC, use $SYSTEM.SQL.SetServerInitCode().

10.7 Advanced Queries

You can use the classes %FileMan.File and %FileMan.Field to query the FileMan data dictionary.

Also, you can use the classes %FileMan.MappedFile and %FileMan.MappedField to view metadata regarding the mapping between FileMan and the class definition.

For information on the %FileMan classes, see the InterSystems Class Reference.