Manufacturing Management Software

Scripting Guide

Making IT Work

Save Time

Save Money

Improve Performance

Comprehenisve - Proven - Affordable

www.match-it.com

Smart
Funding Innovation

Table of Contents

Scripting

This section contains reference information that you will find useful if you intend to write or modify Lua scripts.

1 Lua

This topic describes how to use the scripting language built into Match-IT. The information presented here is relevant to writing complete scripts to perform some process and also to writing PPS expressions (Parametric Product Structures) that can be embedded into your methods.

The scripting language used in Match-IT is an extension of *Lua*. You can read all about standard *Lua* at www.lua.org; this document only covers the Match-IT specific extensions. You should familiarise yourself with *Lua* before reading this document.

1.1 What is Lua?

Lua is a powerful, light-weight programming language designed for extending applications. *Lua* is also frequently used as a general-purpose, stand-alone language. *Lua* is free software.

For complete information, visit *Lua*'s web site at www.lua.org . For an executive summary, see www.lua.org/about.html.

Lua has been used in many different projects around the world. For a short list, see www.lua.org/uses.html.

1.2 What is a script?

In Match-IT, *Lua* scripts can be used in two contexts. They can be used as a PPS expression (Parametric Product Structure), or as a program to perform some function.

When used as an expression, you enter the expression directly into Match-IT via the expression editor. The expression becomes ‘embedded’ into a method.

When used as a program, the script is a text file you create using any text editor (e.g. Notepad, we recommend Notepad++ which can do syntax highlighting and folding, see notepad-plus.sourceforge.net). Once created they can be run directly or they can be ‘imported’ as a wizard. It's important to note that although the process is referred to as ‘importing’, it's only the properties that are imported and not the script itself, the script is ‘linked’ not embedded. This means whenever the wizard is executed, the script file is re-accessed and then run. So any changes you make to the script take effect immediately and it need not be re-imported. The main benefit of importing is that the script becomes part of your wizard library and as such can be connected to via wizard buttons available on many forms and/or turned into a menu button.

When 'importing' a script it's referred to as a *template* and must conform to a *syntax* like this:

TemplateFile	::=	[LanguageLine] { IgnoredLine } StartLongComment Template EndLongComment Script
LanguageLine	::=	'#lua'
IgnoredLine	::=	any line that is not a Template
Template	::=	'{{{ TEMPLATE:' TemplatePrefix,TemplateMnemonic,TemplateName,TemplateClass,TemplateUserGroups [Description] { TemplateLine } '}}}'
TemplateLine	::=	Comment | ContextSection | DefaultLine
StartLongComment	::=	'--[['
EndLongComment	::=	'--]]'
Script	::=	See Lua Language Script
Comment	::=	'{{{ COMMENT:' [CommentTitle]

{ TextLine }

'}}}'

ContextSection ::= '{{{ CONTEXT:' [SectionName]

Description

{ DefaultLine }

'}}}'

This is an import section only. It does not exist in the imported template. It is just a convenient way to visually isolate all the template defaults in the import script.

DefaultLine ::= 'DEFAULT:' DefaultName,DefaultType[,DefaultValue]

A default line defines a context default for the template. This causes a qualifier of 'DefaultName' to be attached to the template. The type of the value is 'DefaultType' and the initial value is 'DefaultValue'.

These qualifiers can be accessed within the template by the normal field access mechanism using the drSelfTemplate as the reference. For example:

```
drh = m.open(match_it.drh)             --open file
m.load(drh,m.drSelfTemplate)           --load 'self' to
gain access to qualifiers
DefaultOfInterest = drh.DefaultName    --if the name is
known to be DefaultName
DefaultOfInterest = drh[DefaultName]  --if the name is in
the variable "DefaultName"
...
m.close(drh)
```

This is a convenient mechanism to group all user tweakable stuff in one easy to get to place.

Description ::= '{{{ DESCRIPTION:'

{ TextLine | LineBreak } ['[END]' { TextLine }]

'}}}'

The Description forms the text of the confirmation dialog when the template is run. The confirmation text ends at the description end or the first line containing the '[END]' marker. If the confirmation text is blank, no confirmation dialog is shown when the template is run.

All blank lines are ignored. All text following !! on any line is ignored. This is an end of line comment for use in the import script.

LineBreak ::= '.'
This adds a line break to the description.

There must only be a single `Template` in the file. All lines beyond the `Template` are assumed to be the Lua script. To stop Lua interpreting the `Template` block, it should be enclosed in a long comment ('--[[' to the corresponding '--]]').

If the `LanguageLine` is omitted the system default language is assumed.

See Wizard Functions and Wizard DO Function for the Match-IT specific functions available within Lua. All these functions, and other facilities, are available through a Lua table called **match_it** (usually abbreviated to just '`m`').

1.3 The Match-IT module

In *Lua*, the entire Match-IT extension is available through a module called '`match_it`'. To access this module, all scripts and PPS expressions must include the line:

```
m = require('match_it')  --load Match-IT into 'm'
```

This creates a table (called '`m`' in the example above) where all functions, constants, etc. are presented as keys in the table. Unless shown otherwise all names mentioned in this document must be prefixed by the table name using the usual dot notation, e.g. `m.open(mch)`.

Lua is a case sensitive language, whereas Match-IT is not. Keys in the Match-IT table are added on demand the first time you use them. When keys are added they are added using whatever case convention you used. If you are not consistent on case usage, it just results in multiple table entries, one for each case convention. This is benign but wasteful.

1.4 Accessing files

All the databases within Match-IT are accessible from scripts. You have the same access to the databases from within scripts as Match-IT itself has. BE WARNED! It is possible to do great damage if your scripts do not work correctly.

File numbers

All database files are identified by a file number. The first is number 1. You do not need to know these numbers as each has a mnemonic that is easy to remember. The mnemonics are constant and never change, whereas the file numbers themselves may change if a new database is added to the system. All these mnemonics can be found in the reference section. All file mnemonics consist of the three letters you see on the front of each line in the file list.

In any function that requires a file number as a parameter, it can supplied by just quoting its mnemonic, e.g. `m.mch` for the material catalogue.

Field numbers

For each database, each of its fields has a number allocated to it for reference purposes. Like file numbers, these all have a mnemonic. All the mnemonics for the fields of any file can be found in the reference section. A list of all field mnemonics is available. In the help topic they are shown as `tla:name` (i.e. the file mnemonic, a colon, then the field name). In scripts the colon is dropped. E.g. `m.mchBuyable` is referring to the `Buyable` field in the `mch` database. There is an extra field that can be accessed from scripts that is not in the on-line help, that is `RecNo`. This represents the internally allocated record number. This can only be read, you cannot assign to it.

In most cases, fields are referenced directly by their name (see Accessing fields in files) without the need for these mnemonics. A few functions require the field to be referenced by its number (e.g. `m.set()`), use these mnemonics for those cases.

Key numbers

For each database, each of its keys (aka indexes) has a number allocated to it for reference purposes. These also have mnemonics, they too can be found in the reference section. All key mnemonics are of the form `tla:Field1Field2…FieldNKey`, where `tla` is the file mnemonic and `Field1…FieldN` are the names of the fields that compose the key in their order of significance. The word `Key` always appears as a suffix. This distinguishes them from field mnemonics. As with field numbers, the colon is dropped in scripts. E.g. `m.mchStepMaterialKey` is referring to the `mch` key that is composed of the `Step` field and the `Material` field.

1.5 Using virtual files

The term *virtual file* refers to temporary files that can be created in scripts. They only exist for the duration of the script and only exist in memory. They are most useful as receptacles for CSV files read in and for ad-hoc reporting. You can do almost anything with a virtual file that you can on a real file. The main difference is that a virtual file must have its fields declared before the file can be used. You do this using the `m.field()` function (see later).

The mnemonic for a virtual file number is of the form `V##`, where `##` is a number in the range 01..32, e.g `V01`, `V02`, etc.

The key (index) for a virtual file must also be declared before the file can be scanned in anything other than physical order. This is done by implication by referencing a key mnemonic for the file. For example, if virtual file `v01` has a field declared of `CustomerName`, then to create a key on that field, just reference a key (using the `m.set()` function) with a name of `m.v01CustomerNameKey`. Only one key can be defined per file and it can only consist of a single field. You can emulate multi-field keys programmatically in the script by using a structured string. Records with null (or empty) keys are not indexed.

Note: It's more efficient to use an ordinary *Lua* table if you just need working space, even if that space is very large.

1.6 Accessing fields in files

To access fields in a database, first you have to declare a *file* variable that references the file you want to access, then it must be opened. You do this using the `m.file()` and `m.open()` functions, like this:

```
mch = m.file(m.mch); m.open(mch)
```

For a virtual file, you must define the fields before the file is opened, like this:

```
myfile = m.file(m.v01)
m.field(myfile,'My string field',m.str)
m.field(myfile,'My material',m.material)
...
m.open(myfile)
```

Fields in an open file can be referenced by using their name, like this:

```
oldmanqty = mch.manqty --read a field
mch.manqty = m.measure('Metre,2') –-assign to a field
```

The fields referenced can be 'native' fields or 'qualifiers'.

Native fields are pre-defined as part of the standard Match-IT databases. A list of 'native' field names for each file is available in the reference section.

Qualifiers are fields added to the database by you and show in the 'qualifiers' tab on many forms. They are accessed in exactly the same way as a native field, e.g:

```
oldvalue = mch.myfield --read a qualifier
mch.myfield = m.measure('Metre,2') –-assign to a qualifier
```

WARNING: When you write to a qualifier field, the database is updated immediately. Contrast this to native fields that are all written at once when you execute `m.Create()` or `m.Update()`.

Drilling through

You can also access fields in objects that are referenced from some variable without having to declare and open the file. This is useful for accessing a single field but is very slow (because each access, opens the file, reads/writes the field, then closes the file again), so is not suitable when a large number of fields are required to be read or written.

To drill-through, you use the dot notation, like this:

```
sol = m.file(m.sol); m.open(sol);
... (do something to load a sales line sol)
if sol.OurPartNum.ManQty < m.measure('Each,2') then
  sol.OurPartNum.ManQty = 'Each,2'
end
```

This code fragment will update the reference quantity of the material in the sales line to `2 each` if the current reference quantity is less than 2. Note: it's the MCH that is being updated in this example, not the SOL. We're just using the SOL to get at the MCH.

This dot notation can be continued to any depth. In general the syntax is:

object '.' fieldname { '.' fieldname }

The *object* must refer to a record in some database.

Direct file buffer access

You can also access fields directly in the file buffer provided the file is open (from within a *Lua* script or from the Match-IT context that invoked the *Lua* script, e.g. PPS). Use a notation like this:

```
if m.mch.ManQty < m.measure('Each,2') then
  m.mch.ManQty = 'Each,2'
end
```

This code fragment will update the reference quantity of the material currently in the file buffer. Only the file buffer is affected, not the file itself. The file itself must be updated via `m.Create()` or `m.Update()` or in the Match-IT environment the *Lua* script returns to.

Qualifier 'records'

If your system uses a large number of qualifiers that are intended to be accessed from scripts, it is useful to group related qualifiers into 'records'. A qualifier 'record' represents all qualifier fields associated with a file that have a common prefix. A 'prefix' is any part of the name that precedes a dot (.) character. E.g. if you have qualifiers associated with your materials catalogue with names like this:

- core
- VA
- primary.machine
- primary.wire
- primary.winding1.volts
- secondary.machine
- secondary.wire

Then `primary` and `secondary` are considered to be records consisting of the fields `machine` and `wire`, etc. They can be accessed using dot notation like this:

```
    ...
    mch = m.file(m.mch)
    m.open(mch)
    m.load(mch,...)
    primaries = mch.primary         --make a record 'reference'
    if primaries.machine == ...     --referencing a value (via a record reference)
    primaries.wire = ...            --assigning a value (via a record reference)
    mch.primary.wire = ...          --the same assignment (without a record reference)
    p1 = primaries.winding1         --a sub-record reference
    p1.volts = 220                  --and an assignment though it
    ...
```

Note the use of assigning a reference to a record (`primaries=`). This is useful to aid the readability of your scripts. Note also, that there can be records within records (`primary.winding1.volts`), to any depth.

Qualifier 'arrays'

The qualifier system can also emulate arrays of records. If the name that follows a dot is entirely numeric, it can be accessed using the array notation in your scripts. E.g. if you define qualifiers like this:

- primary.1.wire
- primary.2.wire
- ...
- primary.16.wire

Then the `wire` component of `primary` can be accessed as an array like this:

```
    ...
    wire1 = mch.primary[1].wire  --get the wire for the first primary
    wire2 = mch.primary[2].wire  --get the wire for the second primary
    ...
```

or, more interestingly, like this:

```
    ...
```

```
   for winding = 1,16 do
      wire = mch.primary[winding].wire
      ...
   end
   ...
```

The numeric element can be specified by any expression that evaluates to a number. Array elements, like Lua, are 1-based, so the first element is number 1. Also, the qualifier name elements for arrays must not start with a zero, this is to ensure there can be no ambiguity with '01' not being the same as '1', etc.

1.7 Data types

Match-IT has nearly 200 data types. A data type defines what a value of that type can contain and also, critically, what can be done with it. In your scripts the type of any value must be defined. It can be defined explicitly or it can be implied from the context. To define the type explicitly, it can be 'cast' using one of the Match-IT type names.

For example, to make a value into a Match-IT string:

```
stringthing = m.str(someotherthing)
```

This mechanism can also be used to construct values of specific types from their parts. For example:

```
somemoney = m.money('Sterling,100')
```

This will make a Match-IT money type that represents £100. The parameter given to `m.money` in this case is the make symbols for the type. NB: The make symbols must be enclosed in quotes if it's a literal string, omitting the quotes can lead to strange effects as an undefined value in *Lua* is interpreted as nil. E.g.:

```
Fred = 6
areal = m.real(fred)
```

Here the case of `fred` is different to `Fred`, so `fred` is undefined. This will create a value in `areal` of 0 not 6.

The data type is implied when dealing with fields in files as each field has a data type associated with it. In this case an appropriate 'cast' is done automatically when you assign to a file field, and the type of any value read will be that defined for the field.

A list of type names, and their make symbols, for each module is available in the reference section. The names under the *LType* column are the names required in your *Lua* scripts.

Type casting

In general, you can define a data type by a 'cast' from a type number (e.g. `m.zvMaterial`), a type name (e.g. `m.TypeNam('Material')`), a file number (e.g. `m.mch`) or a file name (e.g. `m.FileNam('mch')`). The latter is useful when dealing with an owner file, owner record pair from some file. For example:

```
msc = m.file(m.msc)
…
owner = msc.OwnerFile(msc.OwnerRec)
```

Here the date type of `owner` will be whatever is appropriate for the contents of the `msc.OwnerFile` field and its value will be whatever was in the `msc.OwnerRec` field.

Type coercion

When accessing and assigning Match-IT values in your *Lua* scripts, Match-IT will try to coerce values to an appropriate type. If the coercion fails, the operation you are attempting will throw an error and the script will terminate. In most sensible cases the coercion will succeed, for example a number can be considered to be a boolean in an IF condition.

When dealing purely with Match-IT measures, all the usual rules apply, e.g. 1 metre + 1 foot will do unit conversions.

When dealing with mixed Match-IT types and *Lua* types for arithmetic cases, the *Lua* type will be coerced into the equivalent Match-IT type and then the operation performed, e.g. 1 metre + 1, the lone '1' will be converted to a unitless measure and then the addition performed (note: a unitless measure is units compatible with anything).

When dealing with mixed Match-IT types and *Lua* types for comparison cases, **no coercion is performed** (due to, IMO, limitations in the *Lua* language). This means you must specifically coerce comparison operands, e.g:

```
limit = 2
test = m.real(1)
if test < limit then ...
```

Will raise an error, but:

```
limit = 2
test = m.real(1)
if test < m.real(limit) then ...
```

Will work as expected.

Equality

Lua considers any non-nil and non-zero value to be true (like C). This in conjunction with the lack of type coercion when doing comparison operations can lead you into trouble if you are not careful. E.g:

```
x = m.flag(false)
if x then ...
```

In the above, `x` is considered to be true by *Lua* even though it's a Match-IT false flag! Here, there is an implicit == comparison, to make it work properly you must change it into an explicit comparison. E.g:

```
x = m.flag(false)
if x == m.flag(true) then ...
```

This will then work as expected.

Another situation to be wary of is testing for equality with mixed Match-IT and *Lua* types. E.g:

```
a = m.real(1)
b = 1
if a == b then ...
```

In the above, *Lua* considers `a` to be not equal to `b` because they are different types. To make this work as expected, you must again coerce the operands into Match-IT types. E.g.:

```
a = m.real(1)
b = 1
if a == m.real(b) then ...
```

This will then work as expected.

Strings

Lua uses the '\' character as the escape character (to denote special values, e.g. `\n` is the new line character). You must be wary of this when constructing file names and paths from literals. E.g.:

```
filename = 'c:\match_it\configs\welcome.wiz'
```

This will not create what you want because the `\m`, `\c` and `\w` sequences are interpreted as escape sequences, so you end up with '`c:atch_itonfigselcome.wiz`'. Instead use the '/' character like this:

```
filename = 'c:/match_it/configs/welcome.wiz'
```

Arithmetic

When doing arithmetic involving a Match-IT data type, the result is always a Match-IT type of the same type as the first operand. This applies even if the first operand was not initially a Match-IT type. E.g.:

```
now = m.now()
later = now + 3600
number = 3600 + now
```

The type of `later` is a `Clock` but the type of `number` is a `Real`.

1.8 Accessing defaults

You can access all Match-IT's defaults in a similar way to fields in files. They appear as fields in the `'m'` table. The 'field' name is the same as the defaults' *ID.* E.g.:

```
if m.asActiveTicks < m.int(10000) then          --read
      m.asActiveTicks = 10000                          --write
end
```

This will set the *gone away ticks* default to 10,000 if it's currently less than that. Note that assigning to defaults in this way just updates the in-memory copy; it is not saved in the database. Use the `m.SetDefault()` function to do that.

A list of default IDs, for each module is available in the reference section. The names under the *ID* column are the names required in your *Lua* scripts.

1.9 Accessing registers

Registers are a special set of values that can be passed to your scripts from Match-IT. They typically contain information pertinent to the context the script is being invoked from. See the Register usage conventions for details. There are 32 registers, they can contain values of any Match-IT type, and they are named `r01` to `r32`. To access them, just use their name as a field on the `'m'` table, like this:

```
if m.r01 > m.measure('Metre,1') then ... --read
m.r02 = m.money('Sterling,100')          --write
```

1.10 Accessing system globals

There are a number of context variables maintained by Match-IT that can be useful in scripts, for example the currently logged-in user. There are mnemonics for most of these that you can use in *Lua*. The variables available are all defined in the reference section. They can be accessed for reading (only) as `m.GlobalName`, where `GlobalName` is the name as you see it in the help system.

1.11 Accessing system constants

Many fields and functions in Match-IT require special values that have specific meanings. There are mnemonics for most of these that you can use in *Lua*. This means you can refer to these special constant values by a mnemonic name rather than having to know the specific value. The constants available are all defined in the reference section. They can be accessed for reading (only) as `m.ConstantName`, where `ConstantName` is the name as you see it in the help system. They are returned as native *Lua* numbers or strings.

1.12 Pre-defined functions

The Match-IT extension to *Lua* adds a large number of functions to the language that give access to many of Match-IT's internal facilities. The functions available and what they do are all defined in the reference section under Wizard Functions.

The functions referenced above only exists for use within a *Lua* script. There are another set of functions referred to as 'drops' that can also be accessed within *Lua* scripts. These give access to internal Match-IT functions.

The additional functions available as 'drops' are all defined in the reference section.

1.13 Register usage conventions

There are many places in Match-IT where a script can be invoked from a form or list to perform some function, for example on the list of customers there are '**`Add`**' and '**`Edit`**' buttons (when the user interface level is high enough) that can be attached to a script. In such cases information from the form or list is passed to the script in registers (see accessing registers). In addition a special default (`drTemplateParam`) is set to a value that represents the operation to be performed.

There is also a mechanism to 'hook' almost all events for almost all controls for almost all windows within the UI (User Interface). This mechanism allows an expression to be attached to an event that is evaluated whenever that event is triggered. This is useful to override, or augment, the standard behaviour within a window for a particular control. These 'hook' expressions are passed context information in registers.

Scripts can also be invoked from your methods to evaluate method elements (e.g. what material to use, dimensions, etc.). This context is referred to as PPS (Parametric Product Structures).

In all cases the scripts are expected to return a single result that is appropriate to the context. The scripts return a result using the return statement, e.g:

```
m = require('match_it')
result = m.material(‘1877’)
return result
```

Form/list editing

The `drTemplateParam` default will contain the word ‘Create’ if the required action is to *add* a new object (customer record, material record, etc.). It will contain the word ‘Edit’ if the required action is to change an existing object.

For a ‘Create’ operation, register `r01` is empty (actually a `m.Void()`) and register `r02` contains the parent object if there is one (e.g. the sales order header for a sales order line).

For an ‘Edit’ operation, register `r01` will contain the object to be edited and register `r02` will contain its parent (if there is one).

In the case of using scripts to create method lines from the method editor, a method line can have a parent of a material record or a process library entry, here `r02` is the material record and `r03` is the process library entry.

In the case of attaching a script to the design button in the customer part record form, `r01` contains the customer part record.

UI Hooks

When an expression is evaluated for a UI hook, the registers have the following meaning:

`r01`	The upper case window name invoking the expression.
`r02`	The upper case control name invoking the expression
`r03`	The upper case event name that was triggered, or 'ProcedureEntry' when the window is called, or 'ProcedureExit' when the window exits.
`r04`	The upper case 'when' name. One of 'pre' or 'post'. 'pre' is called before standard logic (but after 'sync') and 'post' is called after standard logic (but before 'refresh').
`r05`	For all events except 'ProcedureExit' this is the LocalRequest value. 1=InsertRecord, 2=ChangeRecord, 3=DeleteRecord, 4=SelectRecord. For 'ProcedureExit' this is the LocalResponse value. 1=RequestCompleted, 2=RequestCancelled.
`r06`	The ForceRefresh flag (only meaningful in a 'post' call). When TRUE the standard logic is calling for a complete re-draw of the window controls and their values.
`r07`	The keycode active (if any) when the event was triggered.

PPS

When a script is invoked for a PPS expression, the registers have the following meaning:

`r01` `r02` `r03`	The ‘parent’ unit quantity, length and width.
`r04`	The ‘spec’ object. This is the ‘top’ thing that is creating the demand, e.g. a sales line.
`r05`	The ‘self’ object. This is either a material record (mch) or a method record (mcb).
`r06`	The make quantity. This is how much of the parent that is required.
`r07`	The demand material. This is the material that is required by the ‘spec’ object.
`r08`	The parent method line (mcb). This is the method record in the parent that is demanding this item. If the parent is the top, this will be empty.
`r09`	The ‘parent’ height.
`r10`	The 'like' root (mcb) or void if not in a LIKE process.

1.14 Useful tools

When you install Match-IT some useful scripts are deposited in your `configs` folder. Look for files with a `.lua` extension in this folder. If you open them in a text editor (e.g. notepad++) they’ll contain comments that describe their function.

One tool in particular is useful when coding scripts involving showing the user several pages of options, that is the `Pages()` function defined in the `pages.lua` file.

The `toCSV()`, `fromCSV()` and `CSVreader()` functions in the `csv.lua` file are useful when processing CSV files.

1.15 Using the console

The console is a useful tool to help you learn to use *Lua* within Match-IT and also to help debug scripts. The console is a simple dialog that allows you to type commands and see the results immediately.

The console can be invoked directly from the standard menu via `Favourites | Functions | Standing Data | Wizards | Run a script`, this will open a form with an option to start a *Lua* shell.

It can also be invoked from a script by using the `m.console()` function.

Any `print()` commands executed in your script will appear in the console. If the console is not active when you issue a `print()` nothing will happen.

1.16 Using the debugger

A simple debugger is available to help diagnose problem in your scripts. To use it add this to your scripts:

```
require’console’
require’debugger’
```

Then you can ‘instrument’ your code using the `pause()` command. After a `pause()` your script will enter an interactive session using the console. Type `help` in that to get a list of facilities available. The debugger is automatically available if your started the console directly (as above).

You can also break into a running script from the thread viewer (press `Threads` in the desktop menu bar to open the thread viewer), select the thread that is running your script and press the **Pause** button. Your script will enter an interactive debug session (Note: if your script is currently stalled waiting for a response from a form; it will not 'pause' until you create some sort of event in the form, like clicking on a field or moving a list selection).

1.17 Example scripts

There are a number of *Lua* scripts installed as part of the Match-IT installation. They are all in the `configs` folder with either a `.lua` extension or a `.wiz`. If you look in a `.wiz` file make sure it's not an old legacy script, they look very different. If you see the marker `#!lua` in the first line then it's a *Lua* script, otherwise it's probably not.

The sample script `ReportSample.wiz` is a simple example of using scripts to create a custom report.

1.18 Wizard Functions

This is the complete list of functions available to the Lua Language wizard expressions.

For brevity, the Lua syntax does not show the **match_it** table prefix. The actual usage in a script will be of the form *match_it.name* where *name* is the function name in this list.

The function name lines are of the form: **name(parameters),results** where *name* is the name of the function, *parameters* are 0 or more parameters required and *results* are 0 or 1 or more results returned by the function. The parameter and result names used here are typically the data type name involved. Parameters enclosed in [] are optional. The data type of *bool* represents a native Lua boolean data type and the data type of *number* represents a native Lua number, otherwise type names are Match-IT logical types.

Unless the functions explicitly return an error code, any internal error will cause the script to abort. Such errors can be 'caught' using the *pcall* function (see the Lua reference manual).

Assign(dObject,sObject)

This function changes the value of the dObject to that of the sObject. The objects must be of the same type. This differs from a Lua `dObject = sObject` assign statement in that the Lua assign changes the LType referenced by dObject to be sObject, i.e after the Lua assign dObject and sObject are references to the same LType object. The assign operation here preserves the two objects but just makes their value the same.

This is most useful from within callback functions in a GUILE page, where it can be used to change the original value of controls.

Beep()

This function just makes the system default attention sound.

Bool(A),bool

This function will coerce its operand into a Lua boolean if possible. Note the following for the cases when the operand (A) is a Match-IT type:

- an empty Str is FALSE, any other Str is TRUE
- an empty RecNo is FALSE, a non-empty RecNo is TRUE
- any numeric type that is 0 is FALSE, else it's TRUE

Call(Template),Object

This function will call the given template. The object returned is the result of the template or 0 if it failed. Parameters can be passed to and from templates using the GetReg and PutReg functions.

Children(Object,Type),RecNoQ

This function returns a new 'RecNoQ' of all objects of the given type that are referencing the given object. An 'object' is a record in one of the the Match-IT databases. Only child references are returned, use LINKS to find all non-child references.

Clear(File)

This clears the file buffer for the given file. After calling this function all file buffer fields are empty.

Close(File)

This function closes a file that has been opened using the Open() function. It always returns a TRUE flag. If this was a nested open, the context of the outer open is restored to allow subsequent Next() functions to continue where it left off.

The File must refer to a file variable (see the *file* function). An outer close does not destroy a virtual file.

All open files are automatically closed when the script stops (for whatever reason).

ComStart(),Depth

Start the Windows COM sub-system. A call to this must be made before any attempt to use COM services in a script, e.g. those provided via luaCOM. A corresponding ComStop() call should be made just before the script terminates to turn it off again (although it will be automatically stopped if the script terminates without doing so). ComStart/Stop calls can be nested. Returns a native Lua number that is the COM start nesting depth prior to this call (so 0 means it was not started before).

ComStop(),Depth

Stops the Windows COM sub-system. A call to this should be made for each call to ComStart(). The COM sub-system is only actually stopped by the 'outer' stop call. Returns a native Lua number that is the COM start nesting depth after this call (so 0 means it is now stopped). Calling ComStop() when it is already stopped is benign.

Confirm(Message,[MessageName],[WindowTitle],[OrCancel]),Result

Ask a simple yes/no/cancel question of the user. Returns a native Lua number with a value of gxYes, gxNo or gxCancel to reflect the user selection. If `MessageName` is given, it is shown above the `Message`. If `WindowTitle` is not given, then 'Confirm' is used. If `OrCancel` (a boolean) is given, pressing the **`Cancel`** button does not ask for a further confirmation (use this if your script performs some specific action on cancel). If `OrCancel` is not given and the user presses the **`Cancel`** button, a further confirmation is asked warning the user the script is going to terminate and giving them the opportunity to change their mind (use this if your script terminates on cancel).

Console([options][,pause])

This starts (or re-starts) an interactive console session. If *options* are given, these are passed to the console (use '-?' to get the usage). If *pause* is present and TRUE an 'OK' message is shown before the outer console is closed. This allows examination of its contents before it disappears.

The console function provides a command line interface where you can type commands and print results to a console window. It's intended as a debug aid. If you call Console() from within your script you can then use the debugger to examine variables, try things out, etc. (Use *require('debugger')* to load a simple debug tool.) The [console:drRunScriptUI] can also be started directly from Match-IT.

When a console is started (by whatever means), the Lua initialisation string is evaluated before any options you pass. This is useful to initialise things you want done every time without having to type them.

Note: an options parameter of "-e print('message') -i" is useful when debugging to give you an indication of where the console was invoked from.

Convert(Measure,Unit[,Date][,Flag]),Measure

Returns the given Measure in the units of the given Unit. This can be used on money and time values as well as measures (e.g. to convert between currencies). If the optional Date is given it means perform the conversion for the given date, otherwise it's performed for today. This is significant when converting currencies. If the Flag is present and TRUE it means be silent about conversion errors. This is useful when probing for conversion possibilities. A conversion error will return a result of 'undefined' (test by comparing its digits to umInfinity).

Copy(object1),object2

This creates a copy of the given object. An ordinary assignment of a Match-IT object in Lua creates a new reference to the same object. This copy function creates a new object that has the same value. The significance of this is best illustrated with an example.

```
a = m.measure('Each,1')
b = a
```

After the above assignments b is a, so changing the value of a will also change the value of b. Whereas,

```
a = m.measure('Each,1')
b = m.copy(a)
```

Here, b is a copy of a, so changing a will not affect b.

Cost(Material),Money

This function causes the cost of a material to be (re-)calculated. The Material parameter is an expression that must evaluate to a material that exists in the catalogue. This function is identical to pushing the 'Re-Cost' button in the UI. The money figure it comes back with is the unit cost calculated, no setup costs will be included.

Create(File,[NoValidate]),Object

This function creates the record in the file as constructed in its file buffer. It returns the RecNo of the object created or aborts if there was an error. The file parameter must refer to a file variable created by the *file* function. The file must be open. If a record has been reserved (see the *reserve* function), then that reserved record is written, otherwise a new record is written. If *NoValidate* is present and *true,* the usual record validation is by-passed when a reserved record is written (use with caution - it becomes your responsibility to ensure value rules are not violated).

The Object RecNo is returned in the form of an LType appropriate to the file involved.

CreateCSV(File,PathName,[DoFields]),RecordCount

This function creates a CSV file from the current contents of the given virtual file

- *File* is a file variable. The file must be closed.
- *PathName* is the name of the file to create the records in.
- *DoFields* is a flag that iff TRUE will export the first line as the field list, otherwise the first line is the first record.

The returned *RecordCount* is a count of the number of output records created (excluding the field list). It's a native Lua number.

Delete(File)

Delete the current record in the file.

DestroyVFile(File,[Undeclare])

This function destroys a virtual file and, if undeclare is true, destroys its field declarations too. The file must be closed. Destroying a virtual file deletes all its records. All virtual files are destroyed when the wizard declaring them terminates (for whatever reason).

File must refer to a file variable that is a virtual file ID (ie. one of V01..V99).

Digits(Measure),Real

This function extracts the numbers from a measure. E.g. 1.6 from 1.6 Kg. The returned Real is a native Lua number.

Duplicates(RecNoQ1,RecNoQ2)

This function removes all members from RecNoQ1 that are not also members of RecNoQ2. Both RecNoQs must be of the same type.

Edit(Object),bool

This function will bring up the edit form for the 'object' referenced. An editor is typically a tlaView:Up procedure. It returns TRUE (1) if the user saved, or FALSE (0) if the user cancelled the edit. The editor runs in the same thread as the template. Template execution is suspended until you close the edit form. (Contrast this with View.)

Eq(A,B),bool

This is the same as the Lua '==' operator except the operands (A,B) can be mixed Lua and Match-IT types. This is useful for those cases where you do not know the type of the operands being compared.

Error(Message,[level])

Show the given message and/or terminate the script.

If level is nil or 0, then the user is asked if they want to carry on or abort.

If level is 1 or true, then the message is shown and the script is aborted.

If level is 2, then the script is aborted immediately.

Any other value for level is the same as 0.

When a script is aborted, the last protected function call is terminated, If no protected call is active then the whole script is terminated.

Note: When terminating, if the Lua start console on error default is set to YES, a console session will be started. This is useful to diagnose what went wrong. If your script includes a line "require('debugger')" then you can examine the local variables of the function raising the error.

Exclude(RecNoQ,Member)

This function removes the Member from the RecNoQ.

Exec(Command,[Directory],[IsDocument],[Wait]),ErrorCode

This function will execute an arbitrary command, or open an arbitrary document.

- *Command* is the command or document name to execute.
- *Directory* is the working directory to set.
- *IsDocument* is a flag that if set TRUE indicates a document is to be opened, otherwise a command is run.
- *Wait* is a flag that if set TRUE will wait for the command to complete or the document to close, otherwise the function returns as soon as the document opens.
- *ErrorCode* will be non-zero if the function failed to run the command or open the document. It's a native Lua number.

Existing(File[,KeyNo][,Flag]),Object

This function determines if the identified record exists already in the file system. It returns the RecNo of the object referenced if the record does exist, or 0 otherwise. The referenced record is constructed as if it was going to be 'created'. The constructed record is then checked for a duplicate in the file system. A 'duplicate' means at least one key is the same. Note: This does not mean that all fields are the same.

The Object RecNo is returned in the form of an LType appropriate to the file involved.

The *file* must be open. If *KeyNo* is not given, then all keys are checked, otherwise just the specified key is checked (in which case only its key fields need be set in the file buffer). If *Flag* is present and TRUE it means check for the existence of any record with the key set, if not present or FALSE it means check if a duplicate key would be posted if an 'add' or 'put' was attempted. This applies to keys that do not tolerate duplicates.

Field(File,FieldName,FieldType),FieldNo

This function declares a field in a virtual file. The *File* must refer to a file variable that is a virtual file. It creates a field with name *FieldName* and type *FieldType*. Once created the field name persists until the file declaration is destroyed. The field can be referenced in the conventional way.

The function returns the field number it was given. It's a native Lua number.

FieldExists(File,FieldName[,Mode]),bool

This function determines if the identified field actually exists. It returns TRUE (1) if the field does exist, or FALSE (0) otherwise.

This is only meaningful when applied to fields that are defined as qualifiers. Fields defined as part of the native file record always exist.

The *Mode* parameter specifies:

- 0(default) = look for the name on the object, if it does not exist there look in its context, if it does not exist there look for its default
- 1 = do not look for its default
- 2 = only look on the object itself

File(FileNo),FileRef

This function creates a *file* variable and associates it with the file buffer identified by *FileNo*. Fields in the file buffer can then be accessed by indexing the file variable. E.g.

```
mch = m.file(match_it.mch)
mch.name = 'a material name' --updates the field
mypart = mch.material        --reads the field
```

FileOf(Object),FileNo

This function extracts the file number associated with the given object. If the object does not represent a record in a file then a file number of 0 is returned.

This function is useful when using operations that require a FileNo/RecNo pair as parameters. E.g. if 'thing' is some object in your script then:

```
FileOf(Thing),Thing
```

is the FileNo/RecNo pair representing that object.

Find(File,[KeyNo],[FieldNo],[TypeNo]),RecNoQ

This function performs a *Set(File,KeyNo,FieldNo)*, then performs Next()'s until the whole file has been scanned, placing each object found in the new RecNoQ. The file must be open, and the appropriate key fields populated. See *set* for an explanation of the significance of the KeyNo and FieldNo parameters.

When no TypeNo is given the LType of the RecNoQ entries will be that implied by the file being accessed. In cases where this is ambiguous (because the file has several types), the TypeNo can be specified as the type to use. It must be appropriate.

FindMaterial(Class[,Field1Name,Value1][,Field2Name,Value2][,Field3Name,Value3][,CompareOp]),Material

This searches the material catalogue for a material with up to 3 fields (native or qualifier) that meet the given search criteria. It returns the Material found or a void if none where found. If more than one field is given, the match is an 'AND' of all the fields given. The CompareOp only applies to the last field. Other fields (if given) are always matched exactly.

The search criteria is defined by the parameters as:

- Class - defines the stock class of the materials to be searched
- Field#Name - defines the field names to look for
- Value# - an 'expression' that evaluates to the value to look for
- CompareOp - defines the type of test to apply to candidates

The possibilities for CompareOp are:

- drFindAbove = find the material whose field value is above or equal the given value by the smallest margin
- drFindBelow = find the material whose field value is below or equal the given value by the smallest margin
- drFindExact = find the material whose field value is exactly the given value (implied if CompareOp

omitted)

A CompareOp of drFindAbove or drFindBelow is only valid when the field is a measure.

Hint: To find the first available material of a class, just provide the class and no fields.

FreeQ(RecNoQ)

This function frees the RecNoQ. This releases the memory it's using. The RecNoQ cannot be referenced after a FreeQ() operation. To do so will cause an evaluation error. The function returns a void RecNoQ. If a void RecNoQ is given to FreeQ it has no effect.

Get(FileRef,KeyNo,[NoWatch]),Object

This function reads the first record from the File that matches the given KeyNo. The fields of the key must have been set in the file buffer. The file must be open.

If Watch is TRUE, the record is read ready for a subsequent update. If an attempt is made to update the record without this, an error will be raised.

The result is an object of an LType that embeds a RecNo from the file read. If a record containing the required key does not exist then nil is returned.

Include(RecNoQ,Member)

This function adds the Member to the RecNoQ if it's not there already. If member was already present the function has no effect.

Item(RecNoQ,Number),Object

This function returns the Number'th object from the given RecNoQ.

Items(RecNoQ),Count

This function returns a count of the number of members in the given RecNoQ. It's a native Lua number.

Join(RecNoQ1,RecNoQ2)

This function adds all the unique members of RecNoQ2 to RecNoQ1. The given RecNoQs must be for the same type.

KillState()

This function marks the Lua state the script is running in for 'demolition'. This means the Lua state will be completely closed and destroyed when the script terminates. If this function is not called and the script terminates without error, then the Lua state is kept in a state pool for re-use later. This considerably speeds up the execution of scripts, particularly where the same script is called repeatedly (e.g. when used for PPS expressions).

KodeClass(Kode),KClass

This function extracts the class from a Kode. The class of a kode can usually be determined even if the kode is empty (i.e. has no instance).

KodeFirst(ClassName),KClass

This function returns the first instance of a Kode with the given class name. This is usually the class record. Note: This function returns its result a KClass not a Kode. To turn it into a Kode use a form like `k = m.KodeMake(nil,KodeFirst(ClassName))`.

KodeInstance(Kode),KInst

This function extracts the instance from a Kode. If the Kode is empty the instance is empty (i.e. a void).

KodeMake(KClass,KInst),Kode

This function constructs a Kode from its parts. If KInst is null, an empty Kode of the given KClass is created. If both are null, an anonymous Kode is created.

Le(A,B),bool

This is the same as the Lua '<=' operator except the operands (A,B) can be mixed Lua and Match-IT types. This is useful for those cases where you do not know the type of the operands being compared.

Links(Object,Type),RecNoQ

This function returns a new 'RecNoQ' of all objects of the given type that are referencing the given object. An 'object' is a record in one of the the Match-IT databases. Only non-child references are returned, use CHILDREN to find all child references.

Load(File,Object,[Watch])

This function reads the record from the File for the record defined by the Object. The Object must represent a RecNo in some file. The file must be open.

If Watch is TRUE, the record is loaded ready for a subsequent update. If an attempt is made to update the record without this, an error will be raised.

Attempting to load a record that does not exist will throw an error and the script will abort unless the error is caught using a 'pcall' (see the Lua language reference manual).

LoadCSV(File,PathName,[IgnoreLine1]),RecordCount

This function loads all the records of a CSV file into the given virtual file. The virtual file is emptied first.

- *File* must refer to a file variable that is a virtual file.
- *PathName* is the name of the file to load the records from.
- *IgnoreLine1* is a flag that iff TRUE will ignore the first line of the CSV file (assumed to be a field name list).

The returned *RecordCount* is a count of the number of records loaded. It's a native Lua number.

Lookup(Class,Dimension1,Dimension2,Field,[AllowNil],[NoMake]),Value

This looks up a value in the 2D 'dependence' table. If the addressed item is not present in the table, the user is asked to define it on the spot unless `NoMake` is asserted. It is placed in the table for future reference.

The lookup is directed by the parameters as:

- Class = the lookup table class code to look in
- Dimension1 = the 'object' to form the first 'dimension' in the lookup
- Dimension2 = ditto for the second dimension
- Field = a qualifier name to lookup

The function returns the value of the qualifier addressed if it exists. If it does not exist an error is thrown unless `AllowNil` is asserted, in which case the function returns a `void()`.

Lt(A,B),bool

This is the same as the Lua '<' operator except the operands (A,B) can be mixed Lua and Match-IT types. This is useful for those cases where you do not know the type of the operands being compared.

Make(Unit,Real),Measure

This function constructs a Measure from its parts. The following identity holds: Make(Units(A),Digits(A)) == A

Member(RecNoQ,Member),bool

This function returns TRUE iff the given member exists in the given RecNoQ.

Members(RecNoQ),Function

This function creates an iterator function that can be used to traverse a RecNoQ. Typical usage is this

```
for item in members(RecNoQ) do
  ... whatever
end
```

Message(msg,[title],[delay])

Show a gxMessage. *msg* is the message to show. *title* is the message title or nil. *delay* is the maximum time to wait for a response (in seconds), 0 or nil means forever.

MonitorStart(Title),bool

This function starts a monitor session with the given title. It returns TRUE iff the session started successfully. This function is only allowed in the context of executing a wizard, it cannot be used by isolated expressions (e.g. a PPS expression). Monitor sessions are automatically stopped when the wizard completes. Nested starts are permitted.

This function will terminate the script if the monitor cannot be started. Thus it always returns TRUE.

MonitorStep([Message][,StopOnInterrupt]),bool

This function issues a monitor step message. It returns (a native bool) TRUE if the user is asking to interrupt the operation and FALSE otherwise. The function is a no-op, returning FALSE, unless a monitor session is active (by whatever means).

If StopOnInterrupt is present and true, the script will be terminated if the user cancels the operation, otherwise the function will return false.

MonitorStop()

This function stops a monitor session started by a previous call to MonitorStart. It always returns TRUE. If no sesion was active then nothing happens.

NewQ(Type),RecNoQ

This function creates a new empty RecNoQ that can contain objects of the given type.

Next(File),Object

This function finds the next (or first) record in the scan defined by the associated Open() (or Set()) function. It returns the RecNo of the record found or 0 otherwise. The RecNo is returned in the form of an LType appropriate to the file (e.g. a Material for the MCH file).

The *file* must refer to a file variable and the file must be open and have been *set* to an appropriate key.

NextMaterial(MaterialGroup,PreviousMaterial),NextMaterial

This function finds the next, or first, material that is a member of the given MaterialGroup. If PreviousMaterial is Void(), the first member is found. Otherwise the next member after the one given is found. The one given is usually the result of some previous call to this function. If there are no more members, then NextMaterial is returned as a Void().

NextResource(ResourceGroup,PreviousResource),NextResource

This function finds the next, or first, resource that is a member of the given ResourceGroup. If PreviousResource is Void(), the first member is found. Otherwise the next member after the one given is found. The one given is usually the result of some previous call to this function. If there are no more members, then NextResource is returned as a Void().

Normalise(Material,Quantity,[Length],[Width],[Height]),Measure

This function takes an 'abstracted' quantity of a material and converts it into the reference quantity units for the material. For 0D materials this is just a units conversion (e.g. Grams to Kilos). For 1D and 2D materials it may involve de-abstracting. E.g. for a 1D material of Bars of 3 Metres, a quantity of 10.5 Metres will be converted to 3.5 Bars.

The result is the *Measure* returned. If no normalisation is possible, the returned measure will have the value of *Undefined*.

Now(),Clock

This function returns the current time.

Time is represented as centi-seconds since midnight. You can perform arithmetic on time. E.g. adding 6000 to a time will advance it by 1 minute, and subtracting 3000 will take it back 30 seconds. The behaviour of a time less than 0 is undefined. 0 itself is interpreted as 'no time'.

Number(A),number

This function will coerce its operand into a native Lua number if possible.

Open(File)

Open the given file. *File* must refer to a file variable. Opens may be nested with no loss of context provided each is matched with a close.

This does not do a Set() on the file. You must explicitly set a key using the *set* function.

Pack(object),RawValue

This function returns the raw value of the object as a native Lua string. NB: The string will contain binary data, including nulls. Use string.byte(s,i).. to look at it. This function can be used when passing values to functions that require arbitrary typed values in their packed form (e.g. qaMakeField).

Page(PageTable),ReturnCode

This function creates user interface (UI) pages that display information and/or ask questions (prompts). See the GUILE manual for details on the usage of this function.

Parents(RecNoQ),RecNoQ

This function translates all the members of RecNoQ into their parents. A 'parent' of an object is the object it is attached to in the database. For example, the parent of a sales order line is its sales order header.

NOTE: The 'type' of the RecNoQ is changed by this operation.

Path([FileName]),FullPath

When a FileName is given, the function returns the fully expanded pathname for the file. A relative FileName is interpreted as being relative to the folder that Match-IT was started in (this may not be the same as the current path). If the FileName contains %registry% elements, that element is replaced by the location of the Match-IT registry.

If the FileName is enclosed in colons (:), it has special meaning as follows:

:cwd:	= return the current working directory
:appdata:	= return the user's application data path

When a FileName is not given, the function returns the folder in which Match-IT was started.

The returned path is a native Lua string.

Populate(File[,NoClear])

This populates the file buffer for the given file with its default values. Unless NoClear is TRUE the file buffer is cleared first (see Clear() above).

Previous(File),Object

This function finds the previous (or last) record in the scan defined by the associated Open() function. It returns the RecNo of the record found or 0 otherwise. The RecNo is returned in the form of an LType appropriate to the file (e.g. a Material for the MCH file).

The *file* must refer to a file variable and the file must be open and have been *set* to an appropriate key.

Prop(Control,Property),Value

Prop(Control,Property,Value)

This function reads (first form) or writes (second form) properties associated with controls on pages constructed using the *page* function. See the GUILE manual for details on the usage of this function.

Remove(RecNoQ1,RecNoQ2)

This function removes all the members of RecNoQ2 from RecNoQ1. Both RecNoQs must be of the same type.

Reserve(File,[Parent]),Object

This function must be called prior to a Create() to allocate a record in the file. If the record is a child of some parent (e.g. a sales line is a child of the associated sales order), then *Parent* must be an object that represents the parent. Parent can be omitted only iff the record is not a [CHILD]. The 'parent' is ignored for non [CHILD] records. The function returns the RecNo allocated (as an approp LType).

Round(Measure,Order),Measure

This function will round the given measure to the power of ten defined by the order parameter. E.g. Round(101.12,1) will yield 101, Round(101.12,10) will yield 100, Round(101.12,.1) will yield 101.1, etc.

Run(Process,[Param1],[Param2])

This function performs the same action as a ribbon menu button. It will run the indicated process and pass it Param1 and Param2 as parameters. If either Param1 or Param2 is not required use the Void() function in its place.

The Process must refer to a menu runnable process. Param1 and Param2 must evaluate to something appropriate to the process referenced.

ScaleFactor(TargetQty,Quantity,[Length],[Width],[Height],[Weight]),Real

TargetQty, Quantity, Length, Width, Height and Weight are measures.

This function determines how many items are implied by the TargetQty. If Length is 0, the items are assumed to be 0D (descrete things). If Length is not 0 and Width is 0, the items are assumed to be 1D (rods, bars, liquids, etc.). If Length is not 0 and Width is not 0, the items are assumed to be 2D (sheets, panels, etc.). The TargetQty can be expressed in the units of Quantity, or Length or Weight or Area (Length*Width). The function returns how many items are required to make the TargetQty. For 1D and 2D cases, this might not be a whole number. If it cannot be determined, due to missing conversions, 0 is returned. This can be exploited to test for the existance of conversions.

The returned Real is a native Lua number.

Select(OldSelection,[Title],[Protected],[Context]),NewSelection

Brings up the selector for the object implied by *OldSelection* and allows a new selection to be made.

Selected(RecNoQ),Object

This function returns the currently selected object from the given RecNoQ. An object is selected by any of: a Filter iteration, user selection (when the RecNoQ is a prompt).

SelfScript(),ScriptFileName

This function returns the name of the last script run on this thread by drLuaRunScript. For scripts started via ThreadStart() this will be the script name given to ThreadStart(). For scripts started from the Match-IT UI, it will be the name of the script so started. The script name returned is a native Lua string.

Set(File,[KeyNo],[FieldNo],[Limit])

This functions sets the key that a subsequent Next() or Previous() will use. *KeyNo* defines the key to be used, omitted means scan in physical order. *FieldNo* defines the field to fix on, omitted means no fix field. *Limit* sets the maximum number of records to SELECT, omitted means no limit. The file buffer must have been populated with the appropriate key field values.

SetDefault(DefaultName,Value,[Object])

All parameters are arbitrary expressions.

If an *object* is not given it sets the system default to the value given. If an object is given, it sets the instance default for that object (use this form to set, for e.g., customer specific defaults).

The default name must be the system name of the default. This can be found by looking at the Detail of the default in the System Defaults list.

The default is set in memory and in the database. This means the default value will remain in force even if Match-IT is shut-down and re-started.

SetPath(NewPath),OldPath

Set the current working directory to NewPath and return what it used to be (as a native Lua string).

SetVFile(File,PropertyName,PropertyValue)

This function sets a property of a Virtual File. A virtual file is a memory (only) based structure that behaves like a real file. They are intended to be useful in constructing arbitrary lists. All wizard functions that accept file objects as parameters can accept a virtual file, e.g. View.

The *File* parameter must refer to a file variable. The *PropertyName* defines the property that is to be set. It must be one of 'Name' to set a logical name for the file; 'Title' to set a display title when viewing; 'Description' to set a description when viewing. The *PropertyValue* is the value of the property to set.

ShortPath(LongPath),ShortPath

Return the short name for the given filename. The short name is the equivalent file name in DOS 8.3 format. The given filename must exist and be accessable. If the filename is already in its short form it's just returned unaltered. The returned short name is a native Lua string.

Signal(ThreadName),bool

This function sends a 'signal' to the thread identified by 'ThreadName'. The function returns TRUE (as a native Lua boolean) if the signal was sent, or FALSE otherwise (which means the thread does not exist). It's an interface to ztSignal() and can be used to terminate an m.wait() call in some other Lua thread.

String(A),string

This function will coerce its operand into a native Lua string. Note: any Match-IT type that is an object will translate into its name.

Symbols(A),string

This function will translate its operand into a native Lua string containing its symbols. The symbols of an object are a comma separated list of components that can be used to reconstruct the object. E.g. for a measure of 2 metres, its symbols are: Metre,2. This is useful when exporting fields to some file that you want to import again. Such symbols can be blindly assigned to a file field and they will be re-constructed into the proper value for the field.

ThreadGet(),Object

This function waits for and gets a message sent to this thread. The message can consist of any LType value. If no message is available the thread will stall until one is. Use the ThreadReady() function to determine if a message is available before calling this function. Use the Type() function to determine the type of object received.

ThreadName(),Name

This function returns the name of this thread. If this thread was not started by the ThreadStart() function it returns nil. Otherwise it returns the name of the thread as returned by ThreadStart(). This can be used in 'generic' scripts to determine if it is running as the parent or one of its children. The returned name is a native Lua string.

ThreadPut(ThreadName,Object)

This function sends a message to the thread identified by ThreadName. The message sent is the value of the LType identified by Object. Messages are added to a queue for the destination thread. The sending thread does not stall.

ThreadReady(),bool

This function determines if a message is ready to be read from the message queue for this thread. It returns (a native Lua) 'true' if there is or nil otherwise. Use ThreadGet() to retrieve the message.

ThreadRelock()

This should be called immediately after calling some 3rd party library that contains its own windows message loop (e.g. IUP).

ThreadRunning(ThreadName),bool

This function determines if the identified thread is running. The thread is identified by the name returned by ThreadStart(). If the thread is running the function returns (a native Lua) true, otherwise it returns nil.

ThreadSleep(SleepTime)

This function puts the calling thread to sleep for SleepTime centiseconds. During this time other threads may execute.

ThreadStart(ScriptFileName,ThreadName),ThreadName

This function starts a new thread, gives it the name ThreadName and attempts to execute the script specified by ScriptFileName. The thread terminates when the script terminates. The thread runs in its own independent Lua state and runs indepedently of the starting parent thread. In particular, the child thread will continue to run even if the parent thread terminates. Threads can communicate values of any LType using the ThreadGet() and ThreadPut() functions. The function returns the name of the thread started as a native Lua string.

ThreadUnlock()

This should be called immediately before calling some 3rd party library that contains its own windows message loop (e.g. IUP). A call to ThreadRelock() should be made immedaitely on return from the 3rd party library. While a thread is un-locked, the script must not call any native Clarion run-time library functions, in particular it must not call any UI functions. If the library calls a 'call-back' function into your script and that function requires access to the Clarion RTL, it must temporarily re-lock and un-lock the thread while it does it.

Today(),Date

This function returns the current date.

Dates are held as a day number from a (historical) reference date. Adding a number to a date advances the date by that number of days. Subtracting a number moves the date back by that number of days.

ToType(Object,LuaTypeName),NativeThing

Convert a Match-IT LType value into the equivalent native Lua value if possible. If not the result is nil.

TransAbort(Cause),ErrorCode

This function will explicitly abort the current transaction. All file write actions from here on, up to the TransEnd will be ignored. When the TransEnd function is executed, the whole transaction will be discarded. The error message shown at the TransEnd will be whatever was set by the Cause given to this function. The ErrorCode returned is the error that will be shown when the transaction ends. In Lua, it's a native Lua number.

TransAborted()

This function returns the error code associated with the current transaction. If the current transaction has been aborted, implicitly or explicitly, an error code will be returned. If there is no error, 0 is returned. It's a native Lua number.

NB: Lua does not interpret 0 as false, so do this: `if m.TransAborted() ~= 0 then...` and not `if m.TransAborted() then...`

TransBegin(),ErrorCode

This function initiates the current transaction. It returns an error code or 0 if successful. In Lua, it's a native Lua number. Every file that is going to be written to in the tranaction, either explicitly or implicitly, must have first been included. If a file write is attempted without first performing a TransInclude, the transaction will abort with an error when the TransEnd is performed. A TransPrepare must have already been performed.

TransEnd(Silent),ErrorCode

This function terminates the current transaction. It returns an error code or 0 if successful. It's a native Lua number. A TransPrepare, TransInclude(s) and a TransBegin must have already been performed. All the file write actions performed since the TransBegin function are either committed or discarded by this function. If any error was detected, or the transaction was explicitly aborted, the file write actions will be discarded. Otherwise the actions are committed and the Match-IT databases are updated **all at once**. At the end of this function, either all the actions will be done or none of them. If the *Silent* option is given and set to true, then no error message is shown, otherwise an error message is shown if the transaction failed.

TransFast(),Depth

This function puts the transaction system into {\i fast} mode. In this mode transactions are not atomic and an error during a transaction could leave the file system in an inconsistent state. **BEWARE!** For every call to TransFast there must be a corresponding call to *TransSafe*. The TransFast function is intended to be used during complex import operations and in a context where you know how to recover should it fail. In fast mode transaction performance is improved because records are not flushed to the disk immediately, they are saved in memory and flushed en-masse every now and then. I.e. fast but very dangerous. **USE WITH CAUTION**.

The *Depth* returned is the fast nesting depth. It's a native Lua number.

TransInclude(FileRef,[AccessType],[Object]),ErrorCode

TransInclude(FileNo,[AccessType],[Object]),ErrorCode

This function adds the given file to the current transaction. It returns an error code or 0 if successful. It's a native Lua number. Every file that is going to be written to in the transaction, either explicitly or implicitly, must be included between the TransPrepare and the TranBegin. If a file write is attempted without first performing a TransInclude, the transaction will abort with an error when the TransEnd is performed. A TransPrepare must have already been performed.

The file must be specified by an open FileRef (as returned by the File function) or a FileNo. The latter form is useful when the script must include the file but does not access the file buffer directly.

AccessType defines the action that will be done on the file. Possibilities are:

- (B)are - just do a bare trInclude
- (C)reate - records are going to be added to the file
- (U)pdate - records are going to be updated
- (D)elete - records are going to be deleted

They are defined by a string containing the above ()'d characters. If omitted 'CUD' is assumed.

Object defines the object that is going to be deleted. If present the files necessary to delete that record are auto included. If not present then all files that could possibly have a reference to the file are auto included. If the Object is a FileNo then the record currently in the file buffer is to be deleted. Object is ignored for access types other than delete.

TransPrepare(Title),ErrorCode

This function prepares a new transaction with the given title. It returns an error code or 0 if successful. In Lua, it's a native Lua number. Transactions are automatically ended when the script completes. Nested prepares are permitted provided each is bracketed by a corresponding TransEnd.

No user interaction is allowed between a TransPrepare/TransEnd pair. Performing functions that result in user interaction, either directly or indirectly, will have an undefined, and propably undesirable, effect.

There is a protocol to using the 'Trans' functions. A transaction is a set of file write actions that are to be performed atomically (i.e. all or nothing). Transactions are also much faster than 'one at a time' file write actions. The protocol is:

1. TransPrepare
2. TransInclude for each file that is to be *written* to (including implied writes)
3. TransBegin
4. (the file write actions as required)
5. TransEnd

Failure to follow this protocol will result in an error and the script will terminate.

TransSafe(),Depth

This function must be called subsequent to every call to *TransFast* to restore the transaction system to its usual safe mode. Safe mode is restored automatically when the wizard terminates (for whatever reason) but you should not rely on this, as other threads cannot perform normal transactions while the transaction system is in fast mode.

The *Depth* returned is the fast mode nesting depth remaining. It's a native Lua number.

Type(Object,[no-coercion]),Name

This function returns the Match-IT name for the type of the object given as a native Lua string. If the no-coercion option is present and TRUE, then the name returned for a non-Match-IT object will be blank, otherwise the name of the nearest equivalent Match-IT type will be returned. E.g. a Lua Boolean will return a name of Flag when coercing or blank when not (NB: It returns an empty name, not nil).

TypeOf(Object),TypeNo

This function extracts the type number associated with the given object.

Units(Measure),Unit

This function extracts the units from a measure. E.g. Kilo from 1.6 Kg.

UnPack(RawValue,type),Object

Turns a raw packed value into an object of the given type. The raw packed value should be the result of some previous pack(object) call. The raw packed value is assumed to be appropriate to the type being created. **NO**

CHECKING IS PERFORMED. It's is assumed you know what you are doing if you use this function.

One semi-legitimate use is:

```
RecNo = m.unpack(m.pack(m.int(1234)),m.Material)
```

This will create a material object referencing record number 1234 (whatever that might be). Sometimes useful for performing data recovery exercises when you know the RecNo's involved from somewhere else.

Unreserve(File)

This undoes the effect of the last {\i reserve} if the record is not going to be created. Unreserve is not required if *Create()* is performed on the file.

Update(File,[NoValidate]),Object

Update the current record (only) for the given file. The file parameter must refer to a file variable created by the *file* function. The file must be open. The Object RecNo is returned in the form of an LType appropriate to the file involved. If *NoValidate* is present and *true*, the usual record validation is by-passed when the record is updated (use with caution - it becomes your responsibility to ensure value rules are not violated).

Validate(File),bool

This function will validate the record implied by the File. The function returns a TRUE flag (1) if the validation is successful, or FALSE if not. Validation is automatic when a record is Created or Updated (unless inhibited). If the validation fails the user is shown an error message.

View(Object)

This function will bring up the viewer browse for the 'object' referenced. A viewer is typically a tlaView procedure. It always returns TRUE. A viewer called up like this runs in a separate thread. It will continue to run even after the template finishes unless you explicitly close it. (Contrast this with Edit.)

ViewVFile(File),Object

This function displays the current contents of a virtual file. The current file position will be selected. The returned *Object* is the record selected from the list. The *File* must refer to a file variable for a virtual file.

Void(),Void

This function just returns a Void LType. This can be used for functions that require parameters and you don't want to give it one. A Void() can stand for anything. It's a 'god' - hands out anything you want.

Wait([TimeLimit]),ThreadName

This function waits for either the specified amount of time or until it receives a signal. It's an interface to ztWait(). It returns the ThreadName (as a native Lua string) of the signal sender if it got a signal or nil if it just timed out. If no time limit is given it'll wait forever (not a good idea!).

Watch(File)

Watch() must be called prior to a Load/Next/Previous if a subsequent update is expected. Watch() is implied when in a transaction.

1.19 Wizard DO Function

The **DO** operation is one of the functions that can be used within wizard expressions. It's not a function in the normal sense, but rather it provides access to internal services exposed to Lua.

The normal function call syntax in Lua is used where the function name is one of the operations identified in the Processes and Operations list.

The number and type of parameters is dependent on the operation. There may be up to 24. Each may be an input parameter or an output parameter.

In Lua scripts, a DO function returning an error will cause an error to be thrown and the script will terminate (unless you catch the error in your script using pcall or xpcall).

The operations available, their syntax and semantics, is defined under the *DO Operations* topic in the help file for each system module, e.g. DO Operations in the Material Stock module. In those descriptions the parameters are identified by quoting the logical type name that is expected, preceeded by either a '+' symbol or a '-' symbol. When preceeded by a '+' it indicates the parameter is an input parameter (i.e. passed to the function). When preceeded by a '-' symbol it indicates the parameter is an output parameter (i.e. the function passes it out as a result). Sometimes a '-' parameter can be both an input and an output.

Its Lua usage syntax is:

R1,...,R24 = match_it.operation(P1,...,P24)

The operation is the name of the function. In Lua, the syntax of the call of a DO function is the same as any other function. Only the output parameters (if any) of the function called are returned as results (R1..R24). The passed parameters (P1..P24) may be omitted or nil, in which case an 'empty' value of the appropriate type is passed to the function. If an output parameter is given in the parameter list (P1..P24) then it supplies an initial value for the output to the function.

2 GUILE

This section describes how to create user interfaces from Lua scripts using the GUILE system.

GUILE is an acronym for: Graphical User Interface Lua Extension.

2.1 What is a User Interface?

The term user interface in this context refers to a Lua script asking questions of a user and/or presenting information to a user. A user interface can be as simple as asking for confirmation to perform an action or as complex as a product configurator asking for many specification elements.

The welcome wizard you saw when you first installed Match-IT contains a simple user interface asking for various setup options.

Almost anything you see in a built-in Match-IT window can also be done in a window created by GUILE (*GUILE* is the name used to refer to the facilities described in this manual).

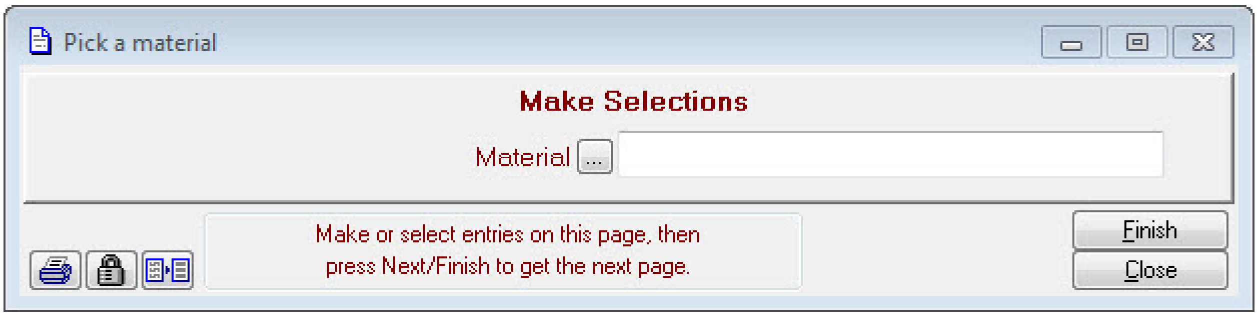
2.2 How do I create a user interface?

User interfaces are created using the `page()` function available from Lua scripts. This function takes a parameter that is a Lua table containing the interface definition (referred to as a *window*) and an optional boolean value that specifies if the window is to be previewed or run (more on that later).

Here is a very simple window definition that just asks the user to pick a material:

```
--make a selection
materialPrompt={text='Material',value=m.Material()}
m.page{title='Pick a material',name='Make Selections',prompts={materialPrompt}}
m.message('You selected '..materialPrompt.value)
```

And this is what the window this creates will look like:


Pick a material

Make Selections

Material

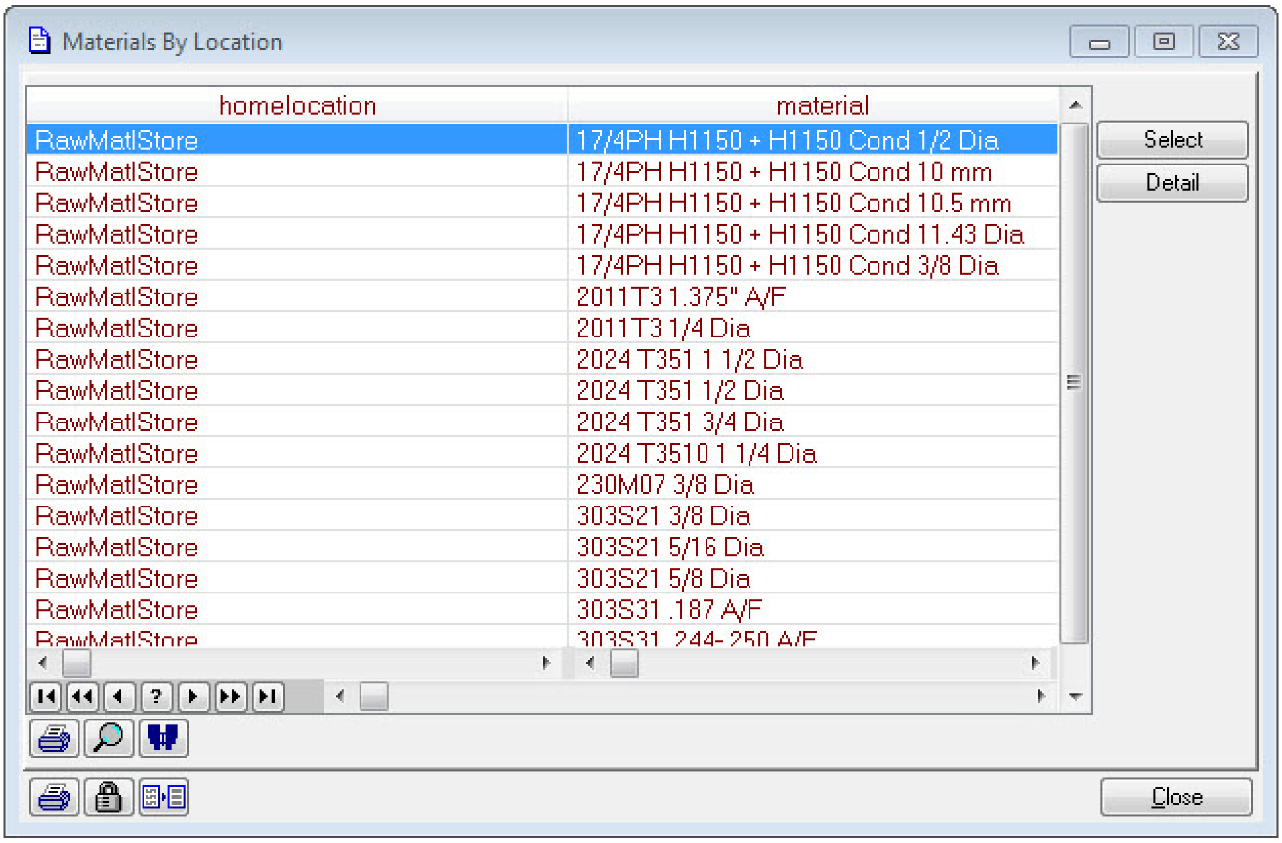
Make or select entries on this page, then press Next/Finish to get the next page.

Executing the m.page() function draws the window and waits for a user response. When the user presses the **Finish** button, the m.page() function returns to your script and, in this example, `materialPrompt.value` will contain whatever material the user selected from your catalogue.

Here's a more interesting example that partly mimics the built-in materials selector:

```
--browse a list
mch=m.file(m.mch);m.open(mch)
mch.Archived=false
m.page{title='Materials By Location',
       name='',
       nohints=true,noback=true,nonext=true,
       prompts={
                {file=mch,
                 vsize=18,
                 list={
                       {data='homelocation',width=32,},
                       {data='material',width=32,},
                      },
                 keys={
                       {key=m.mchArchivedHomeLocationMaterialStepKey,
                        fixed=m.mchArchived,},
                      },
                 buttons={
                          {type=m.drPage_Button,
                           text='Select',
                           callback=function() return m.drWizardNext end,},
                          {type=m.drPage_Button,
                           text='Detail',
                           callback=function() m.edit(m.material(mch.RecNo)) end,},
                         },
                },
               },
      }
m.message('You selected '..m.material(mch.RecNo))
m.close(mch)
```

This is what the above definition will look like (using sample data):



homelocation	material
RawMatlStore	17/4PH H1150 + H1150 Cond 1/2 Dia
RawMatlStore	17/4PH H1150 + H1150 Cond 10 mm
RawMatlStore	17/4PH H1150 + H1150 Cond 10.5 mm
RawMatlStore	17/4PH H1150 + H1150 Cond 11.43 Dia
RawMatlStore	17/4PH H1150 + H1150 Cond 3/8 Dia
RawMatlStore	2011T3 1.375" A/F
RawMatlStore	2011T3 1/4 Dia
RawMatlStore	2024 T351 1 1/2 Dia
RawMatlStore	2024 T351 1/2 Dia
RawMatlStore	2024 T351 3/4 Dia
RawMatlStore	2024 T3510 1 1/4 Dia
RawMatlStore	230M07 3/8 Dia
RawMatlStore	303S21 3/8 Dia
RawMatlStore	303S21 5/16 Dia
RawMatlStore	303S21 5/8 Dia
RawMatlStore	303S31 .187 A/F
RawMatlStore	303S31 .244-.250 A/F

All the usual Match-IT widgets operate as normal in GUILE constructed windows. The above examples are in the `configs\page_eg.lua` file.

2.2.1 Window definition table structure

The complete `page()` and window definition syntax is:

```
ExitCode = m.page(Controls[,Preview])
```

The function returns an *ExitCode* of:

drWizardContinue
drWizardNext,
drWizardPrevious,
drWizardClosed or
drWizardError

to reflect the user button press that closed the window. It's a native Lua number.

If *Preview* is present and TRUE, the page is just shown and callbacks are not executed. This is intended to allow the layout to be tested. When previewing, all hidden controls are shown and a message is shown in the console when a callback would have been made. NIL is returned for any extra results the callbacks would've returned.

The *Controls* table is used to construct the window and must contain keys as follows (the ()'d text is the data type of the key value):

Window keys:

id(str)	=	the id to use in the GXW (so its size and position can be saved/restored), if not given the window title is used.
title(str)	=	the title to use on the window (if not declared a default title of 'Wizard' is used)
name(str)	=	sub-title to use on the implied tab '0' (if not declared a default sub-title of 'Make Selections' is used), a tab '0' is implied for controls declared outside of a tab structure (see below for a description of tabs)
hint(str)	=	the help to show for the window (if not declared no help button is shown for the window)
timer(int)	=	if >0 create a timer event at this centi-seconds interval. NB: The event generation timing is approximate, do not rely on the time interval. Also, the presence of a timer inhibits the auto-next option for wizards (i.e. the drAutoNext default is ignored)
showtabs(bool)	=	if true the tabs are exposed on a multi-tab window, default is to not show them
previous(bool)	=	if true show a 'Back' button (unless noback is asserted, this is implied if more than 1 tab is defined and no button is placed in the 'back' position). This is the same as a button control with an implied 'back' callback and a 'text' property of 'Back' and a position of 'back'. The 'back' implied callback selects the previous tab if there is one, otherwise it closes the window with drWizardPrevious.
next(bool)	=	if true show a 'Next' button (unless nonext is asserted, this is implied if more than 1 tab is defined and no button is placed in the 'next' position). This is the same as a button control with an implied 'next' callback and a 'text' property of 'Next' and a position of 'next'. The 'next' implied callback selects the next tab if there is one, otherwise it closes the window with drWizardNext.
finish(bool)	=	if true show a finish button (unless nonext is asserted, this is implied if only 0 or 1 tabs are defined and no button is placed in the 'next' position). This is the same as a button control with an implied 'finish' callback and a 'text' property of 'Finish' and a position of 'next'. The implied 'finish' callback closes the window with drWizardNext.
promptruler(str)	=	the default ruler to use for prompts, if not present uses: '-34p.[E].34d.[H]?' (see Ruler format for a description of rulers)
buttonruler(str)	=	the default ruler to use for buttons, if not present uses: 'L10=R!'
stringruler(str)	=	the default ruler to use for strings, if not present uses: 'L$'
textruler(str)	=	the default ruler to use for text controls, if not present uses: '-L10$74=R'
groupruler(str)	=	the default ruler to use for groups, if not present uses: '34=10..L34=R4?'
nohint1(bool)	=	if true do not draw hint line 1 or the hints bounding box (any strings placed in position drPage_HintLine1 are also ignored)
nohint2(bool)	=	if true do not draw hint line 2 or the hints bounding box (any strings placed in position drPage_HintLine2 are also ignored)
nohints(bool)	=	if true it's a shortcut for `nohint1=true, nohint2=true`
nonext(bool)	=	if true do not draw a 'next' or 'finish' button (any buttons placed in position drPage_Next are also ignored)

noback(bool)	=	if true do not draw a 'back' button (any buttons placed in position drPage_Back are also ignored)
noclose(bool)	=	if true do not draw the default 'close' button (you can still place your own button in position drPage_Close)
noconfirm(bool)	=	if true do not ask to confirm when closing the window, just do it (beware: unsaved changes could be lost)
bare(bool)	=	if true it's a shortcut for `nohints=true, nonext=true, noback=true, noclose=true, noconfirm=true`
callback(func)	=	if declared call this Lua function to process window events or child events not explicitly handled by the child
vcr(table)	=	if declared and is a table, draws the standard 'vcr' form buttons (Back, Next, New, Reset, Del, Save, Close), keys in the table are the names of the buttons (in lower case) and their value is the callback function to call when that button is pressed, the presence of this table implies **noback**, **nonext**, **nohints** and **noclose**, if a key is not present the button performs its default behaviour when pressed, the default behaviour is: **back** - disabled **next** - disabled **new** - disabled **reset** - perform a `window.reset` (this resets all values to as they were when the window was opened) **del** - disabled **save** - disabled **close** - if the window has been touched, presses `save`, otherwise closes the window
prompts(table)	=	table of controls (the key name of 'prompts' is retained for legacy compatibility), each control can contain keys as defined in **Per Control keys:** below.

Per Control keys:

id(str)	=	the id (prepended by the window id) to use in the ASF (so it can have security associated with it), if not declared an id of the form 'C#nnn' is used, where 'nnn' is the control number in the window. Ids are useful to refer to controls from the prop() function (see later).
hide(bool)	=	true if the control is hidden, it exists but is not visible
readonly(bool)	=	true if the control is read-only, on a prompt this prevents its value from being changed, on groups, buttons, strings and tabs it disables them (disabled controls are visible but cannot be selected)
password(bool)	=	if true on a 'prompt' control, the value is displayed as asterisk (*) characters so user entry cannot be overseen
nulls(bool)	=	if true on a 'prompt' control, the value is allowed to be blank
default(bool)	=	if true on a 'button' control, the button is the default button, pressing the **`shift-Enter`** key presses the default button, behaviour is undefined if more than one default button is specified
nochevron(bool)	=	if true on a 'prompt' control, a chevron (») is not prepended when the control has been touched, if true on a save button, chevrons (»...«) are not placed around the button text when any control on the window has been touched

type(int) = the type of the control, one of:

drPage_Prompt	-	an LType editor
drPage_Button	-	a button
drPage_String	-	a (small) text string - up to 256 characters
drPage_Text	-	a (large) text string, formatted as RTF - up to 2048 characters, the string must start with the RTF header '{\rtf1' and finish with the footer '}', the paragraph terminator is '\par'.
drPage_List	-	a container for a list and its action buttons, only 1 allowed per tab, maximum of 4 per window
drPage_Tab	-	a container for any of the above (all tabs are peers), up to 9 tabs allowed per window
drPage_Group	-	a container for prompts, buttons and strings (only), use a ruler to set its size and position NB: Unlike other controls, a group does **NOT** move the Y drawing position, its the window designers responsibility to prevent overlaps/ overflows.

position(int) = where the control is to be placed, one of (see window layout):

drPage_First	-	on the front of the (next) line (forced for List and List buttons)
drPage_InLine	-	follow on from the previous control (default for strings)
drPage_Back	-	over the ‘Back’ button (ignored if **noback** is set), only 1 allowed per tab, the last declared takes precedence, if declared outside a tab, they are global (show for all tabs), otherwise they only show when the enclosing tab is selected, this position is not available if a **vcr** table is present
drPage_Next	-	over the ‘Next’ button (ignored if **nonext** is set), only 1 allowed per tab, the last declared takes precedence, if declared outside a tab, they are global, otherwise tab specific, this position is not available if a **vcr** table is present
drPage_Save	-	over the ‘Save’ button, only 1 allowed per tab, the last declared takes precedence, if declared outside a tab, they are global, otherwise tab specific, this position is not available if a **vcr** table is present
drPage_Close	-	over the 'Close' button, only 1 allowed per tab, the last declared takes precedence, if declared outside a tab, they are global, otherwise tab specific, this position is not available if a **vcr** table is present
drPage_HintLine1	-	over “hint line 1”, only 1 allowed per tab, the last declared takes precedence, if declared outside a tab, they are global, otherwise tab specific, this position is not available if **nohints** is set

drPage_HintLine2	-	over “hint line 2”, only 1 allowed per tab, the last declared takes precedence, if declared outside a tab, they are global, otherwise tab specific, this position is not available if **nohints** is set
drPage_VCRback	-	over the VCR 'Back' button, this position is only available if the **vcr** table is present
drPage_VCRnext	-	over the VCR 'Next' button, this position is only available if the **vcr** table is present
drPage_VCRnew	-	over the VCR 'New' button, this position is only available if the **vcr** table is present
drPage_VCRreset	-	over the VCR 'Reset' button, this position is only available if the **vcr** table is present
drPage_VCRdel	-	over the VCR 'Del' button, this position is only available if the **vcr** table is present
drPage_VCRsave	-	over the VCR 'Save' button, this position is only available if the **vcr** table is present
drPage_VCRclose	-	over the VCR 'Close' button, this position is only available if the **vcr** table is present

ask(bool)	=	if present and true same as type = drPage_Prompt (the type key must be nil) (for legacy 'page' compatibility)
show(bool)	=	if present and true same as type = drPage_Prompt + readonly=true (the type key must be nil) (for legacy 'page' compatibility)
option(bool)	=	if present and true same as type = drPage_Prompt + nulls=true (the type key must be nil) (for legacy 'page' compatibility)
text(str)	=	the visible text for the control (including any hot key designation preceded by an ampersand, e.g. ‘&Save’ makes the ‘S’ a hot-key, pressing the Alt key and the hot-key together will select the control). For a prompt, if text is not declared the default is 'Select a <typename>' where <typename> is the name of the LType for the prompt. For a group, if text is not declared the group is drawn without a box.
hint(str)	=	the What’s This? help for the control. For a prompt, the first line becomes its tool-tip, pressing the `[?]` button shows the help. The `[?]` is not dawn if there is no help.
name(str)	=	the sub-title to use on a tab or list (if not declared, no sub-title is shown)
callback(func)	=	if declared call this Lua function to process the controls events, otherwise use the callback associated with its parent (see below)
value(LType)	=	for a prompt, it's the LType object to be shown/edited, the presence of this key when type is not declared implies a drPage_Prompt control **NB**: While the window is open, this value must **NOT** be changed by directly assigning to it, it should only be changed by using the `assign` operation. Failure to observe this rule will result in memory corruption and undefined behaviour (which will be bad).
alias(str)	=	for a prompt, it refers to the id of another prompt whose value is to be used here, it must be declared before the alias, if the id is not unique in the window, which one is referenced is not defined, this provides an alternative prompt to view/edit the same value
context(LType)	=	for a prompt, it's the context to pass to the LType editor. Not all LType editors expect a context, for those that don't this property is ignored. For those that do, this property if present must be of the LType expected or an equivalent LType. A context

		is used by an LType editor to give it a hint when there are choices, for example with the `ClassMat` LType the hint specifies the material class to select from when an example material is not given.
ruler(str)	=	for a group, prompt, button or string, this defines how to render the control elements (see below)
tab(table)	=	for a tab, it's a table of controls to place in the tab, controls outside of any tab are placed in the implied tab '0', tabs cannot be nested, i.e. a tab cannot be placed inside another tab, the presence of this key when type is not declared implies a drPage_Tab control
file(file)	=	on a list, the file the list is looking at (it must be open), it may be a physical file or a virtual file
keys(table)	=	on a list, a table of keys the file is being browsed on, when more than 1 key is declared, a key change tab sheet is shown, if only 1 key is declared, no key change sheet is shown, if not declared the RecNoKey is implied (i.e. key number 1), on a virtual file only 1 key is allowed and **key** must be a field number in the file that has unique values or 0 for physical order, each entry must be a table with the following keys:

key(key)	=	a key number the file can be browsed on
text(str)	=	text to show on the key tab (including its hot-key designation)
id(str)	=	the key id, if omitted, use the text, useful to reference the key via the prop() function (see below)
fixed(field)	=	the field number the key is fixed on or 0 if not fixed, the key must already be loaded in the file buffer
callback(func)	=	if declared call this Lua function to process the keys events, otherwise use the callback associated with its parent (see below)

paged(bool)	=	on a list, true if the list is page loaded, otherwise it's fully loaded (default is paged)
vsize(int)	=	on a list, the minimum number of rows to show, if not declared 16 is used (which is enough to fill the default tab height), the minimum is 4
bsize(int)	=	on a list, the size of the button area to reserve to its right in Dialog Units (a Dialog Unit, or DU, is approximately a quarter of a character width), the default is the default button width (40 DU) when there are buttons or 0 when not
buttons(table)	=	on a list, a table of list buttons, each entry must be a *button* control, these are drawn vertically to the right of the main list browse area, buttons may also be grouped using a group control
columns(bool)	=	on a list, if true individual columns in a row are selectable in the list, otherwise only the whole row is selectable
group(table)	=	on a group, a table of *prompt*, *button* and *string* controls in the group, the presence of this key when *type* is not declared implies a drPage_Group control
role(int)	=	as a list button on a list with multi-select enabled, this sets the function of the button in a multi-select context, up to 4 such buttons can be defined, omitted buttons just remove that action, the role of each must be one of:

drPage_SelectAll	=	select all records in the browse that pass whatever filters are prevalent
drPage_SelectNone	=	clear all selections
drPage_SelectInclude	=	include the currently highlighted line in the selected list
drPage_SelectExclude	=	exclude the currently highlighted line from the selected

list

selected(RecNoQ)	=	on a list with multi-select enabled, use this RecNoQ instead of creating one to hold the selections (use NewQ to create it, the type must be compatible with the file being browsed), ignored if multi-select not enabled, default is to auto create a RecNoQ but it will be destroyed when the window closes, so set your own if you need it beyond that scope
list(table)	=	on a list, a list of fields to be shown in the list, the presence of this key when *type* is not declared implies a drPage_List control, each field is a table with the following keys:

name(str)	=	the column name, if omitted use the field name of the data from the file
role(int)	=	if present and set to drPage_SelectMarker it turns on the multi-select facility and marks this field as the 'include marker', it must be a calculated field of type Str, any callbacks on this are ignored
width(int)	=	the initial width of the column in characters (users can change this is in the usual way by dragging the column edge), if omitted the width of the column name is used
id(str)	=	the id (prepended by the window id) to use in the ASF (so it can have security associated with it), if omitted use the name
data(str)	=	the field name in the file to show, it's formatted according to its LType, if blank the field is calculated via a callback, for physical files, this may refer to a physical field or a qualifier on the file being browsed
ltype(type)	=	when the column is calculated, the LType of the result expected from the callback
callback(func)	=	if declared call this Lua function to process the columns events, otherwise use the callback associated with its parent (see below)
hint(str)	=	the *What's This?* help for the column

If none of type, *ask, show, option* is given but *value* is given, then type = drPage_Prompt is assumed.

If none of type, *ask, show, option* is given but *list* is given, then type = drPage_List is assumed.

If none of type, *ask, show, option* is given but *tab* is given, then type = drPage_Tab is assumed.

If none of *type, ask, show*, option is given but *group* is given, then type = drPage_Group is assumed.

If more than one of these is given, the behaviour is undefined.

2.2.2 Ruler format

A 'ruler' can be used to control how the elements of a control are drawn.

A ruler is a string with the following syntax (in BNF notation):

ruler	::=	[position]{element}[height]	
position	::=	'+' | '-'	
element	::=	gap | prompt | ellipsis | data | help | edge | nudge	
gap	::=	{[count] ('.' | '=')}	
prompt	::=	{[count] 'p'}	--prompts only

ellipsis	::=	'[E]'	--prompts only
data	::=	{[count] 'd'}	--prompts only
help	::=	'[H]'	--prompts only
edge	::=	'L' | 'R'	--strings, text and buttons only
nudge	::=	[count] '<' | '>'	
height	::=	{[count] (':' | '?' | '$' | '!')}	
count	::=	{'0'..'9'}	

Where []'s indicate optional items, {}'s indicate replicated items, ()'s represents grouping, |'s represent a choice and ''s represent literal text.

The items have the following meaning:

position:	a leading '+' means this control is to be drawn in-line, to the right of whatever was drawn last (same as drPage_InLine) a leading '-' means draw the control on the front of the (next) line (same as drPage_First). The ruler position overrides the 'position' property of the control.
gap:	each '.' moves the drawing position 1 DU to the right (a DU is approximately 1/4 of a character width) each '=' moves the drawing position 4 DU to the right (a DU is approximately 1/4 of a character width)
prompt:	each 'p' represents 1 character position to allow for the prompt text (default is 34), omitted means no prompt text if a prompt precedes the data entry field, it is right-aligned, otherwise it is left-aligned
ellipsis:	represents the ellipsis button [...] or [```], omitted means no ellipsis button, the button occupies 10 DU
data:	each 'd' represents 1 character position to allow for the data entry field (default is 34), omitted means no data entry field
help:	represents the help button [?], omitted means no help button, the button occupies 13 DU
edge:	an 'L' means place the left edge of the control here an 'R' means place the right edge of the control here, the difference between the L and the R is the control width, the absence of an 'R' means the width floats (only meaningful on string controls)
nudge:	< means draw the control 1 DU higher than normal > means draw the control 1 DU lower than normal
height:	each ':' represents 1 DU (a DU is approximately 1/8 of a character height), each '?' represents the default height of a prompt control, each '$' represents the default height of a string control, each '!' represents the default height of a button control. For string controls, this sets the height of the text as well as the vertical space to the next line of controls, setting a multi-line height and a width for a string control will cause the string text to fold across multiple lines. For group controls, this sets the height of the group but does **NOT** affect the vertical space to the next line of controls. For all other control types, it sets the vertical space to the next line of controls.
count:	a decimal number that replicates the succeeding character that many times, e.g. '10p' is the same as 'pppppppppp'

The default ruler for a prompt control is: '-34p.[E].34d.[H]?'

The default ruler for a button control is: 'L10=R!'

The default ruler for a string control is: 'L$'

The default ruler for a text controls is: '-L10$74=R'

The default ruler for a group control is : '34=10..L34=R4?'

Space characters in a ruler are ignored and can be used to visually separate elements.

2.2.3 Callback Interface

Callbacks are the mechanism by which the calling script can be notified of actions performed by the user of the window. They are ordinary Lua functions.

The callback functions are given a single parameter of the control/window table receiving an event. The event ID is lodged in control.event and identifies what has caused the callback to be made and the control/window table identifies the control it applies to, or the window if it's a non-control specific event.

If a callback is not declared for a control, the callback in its parent is called, if that is not declared, the parents parent is called, this process is repeated until the window callback is reached, if that is not present, no callback is called.

The callbacks return an exit code, the codes are the same as the page() function, anything other than drWizardContinue causes the page() function to terminate with the exit code returned by the callback.

If no exit code is returned, drWizardContinue is assumed.

The callbacks are called using a protected call (see *pcall* in the Lua manual), so errors are caught by the page() function, allowing it to gracefully shut-down the window before propagating the error upwards.

The column fill callback returns its result (of an appropriate LType) as well as an exit code. E.g.: `return result,exitcode` (or just `return result` and an implied drWizardContinue). If no result is returned, a NIL value is used.

The event ID is one of:

drPage_WindowOpened	-	the window is open and has been shown
drPage_AllowWindowClose	-	the user has asked that the window be closed, callbacks should return drWizardContinue to allow it or anything else to dis-allow it (except drWizardError)
drPage_WindowClosing	-	the window is about to be closed
drPage_WindowTimer	-	the window timer time has elapsed
drPage_TabSelected	-	a new tab has been selected
drPage_ListSelected	-	a new list row or column has been selected
drPage_Changed	-	on a prompt: the value has been changed from its previous value
		otherwise: the control's touched status has been changed
drPage_Pressed	-	a button has been clicked or a list row/column was double clicked
drPage_Popup	-	a button or a list row/column was right-clicked
drPage_FillColumn	-	notification that a calculated list column value is required
drPage_LoadFixedKey	-	notification that the fixed key elements in a list should be set
drPage_FilterRecord	-	notification that a record is about to be added to a list, return drWizardSkip to reject the record from the list
drPage_NoRecordsFound	-	notification that nothing passes the list filter

drPage_RecordsFound	-	notification that at least one record passes the list filter
drPage_AddSelection	-	notification that a multi-select list is about to add a selection to the selected list, callbacks should return drWizardContinue to allow it or anything else to dis-allow it (except drWizardError)
drPage_RemoveSelection	-	notification that a multi-select list is about to remove a selection from the selected list, callbacks should return drWizardContinue to allow it or anything else to dis-allow it (except drWizardError)
drPage_EmptySelections	-	notification that a multi-select list is about to clear all selections from the selected list, callbacks should return drWizardContinue to allow it or anything else to dis-allow it (except drWizardError)

The callback functions may manipulate the following properties of the window and its controls (via the prop() function, see later):

Properties (read/write):

window.title		
window.hint		
window.timer		
window.showtabs		
control.value	-	the <u>current</u> value of a prompt, when writing, the type of the new value must be the same as the original; also when writing, changing the value will trigger the drPage_Changed event on that control
control.hint		
control.name	-	only valid on a *tab* and *list* control
control.text		
control.hide		
control.readonly	-	on drPage_Prompt controls this sets readonly, on other types it disables them
control.paged	-	only valid on a *list* control
control.touched(bool)	-	reading returns the touched status of the control, for a prompt this is true if the current value is different to its initial value, for other control types it just echoes the last write to the property writing to touched will trigger the drPage_Changed event if the touched status <u>changes</u> as a result of that write NB: writing to touched is not valid on a prompt control
key.key	-	only valid on a key of a *list* control, when writing, changes the key being browsed to that given
key.fixed	-	only valid on a key of a *list* control, when writing, changes the fixed field of the key
key.text	-	only valid on a key of a *list* control, when writing, changes the text on the key tab (if one is being shown)

Note: In the above, the prefix (window. control. and key.) is purely descriptive. The actual property name used in the prop() function should not include this prefix.

The following properties are also maintained and can be read from the callbacks:

Properties (read only):

window.type(int)	=	type of the control, on the window this will be drPage_Window
window.event(int)	=	the current event ID (see above)
window.control(table)	=	the control triggering the event, or nil if not a control event or it's a pre-defined control (e.g the close and 'vcr' buttons)
window.touched(int)	=	the number of prompts that have been 'touched' in the window
window.context(str)	=	the callback source context of the current event, this will be one of:

'callback'	the callback is coming from a control in the controls space or the window
'save'	the callback is coming from the VCR save button
'close'	the callback is coming from the VCR close button
'back'	the callback is coming from the VCR back button
'next'	the callback is coming from the VCR next button
'new'	the callback is coming from the VCR new button
'reset'	the callback is coming from the VCR reset button
'del'	the callback is coming from the VCR del button

control.orig(LType)	=	the original (un-touched) value of a prompt
control.old(LType)	=	the previous (before last change) value of a prompt
control.parent(table)	=	the control this one is contained within (listbutton-->list-->tab-->window-->nil, etc.)
control.window(table)	=	the overall window table (useful to get at global properties)
control.row(int)	=	during a `FillColumn` or `ListSelected` or `Pressed` or `Popup` event, the list row number active, the file buffer will be loaded during the callback, or empty as appropriate
control.column(int)	=	during a `FillColumn` event, the list column number being filled during a `ListSelected` or `Pressed` or `Popup` event, the list column number selected NB: The column number here is the ordinal position within the `list` table, which may be different to the visual order if the user has moved the columns about.
control.selected(RecNoQ)	=	the selection list actually being used (either auto created or given in the list control definition) when multi-select is enabled on a list
control.key(table)	=	the list key currently active

control.id(str)	=	the id of the control (useful in prop())
control.event(int)	=	the current event ID (see above)

The following properties are write-only:

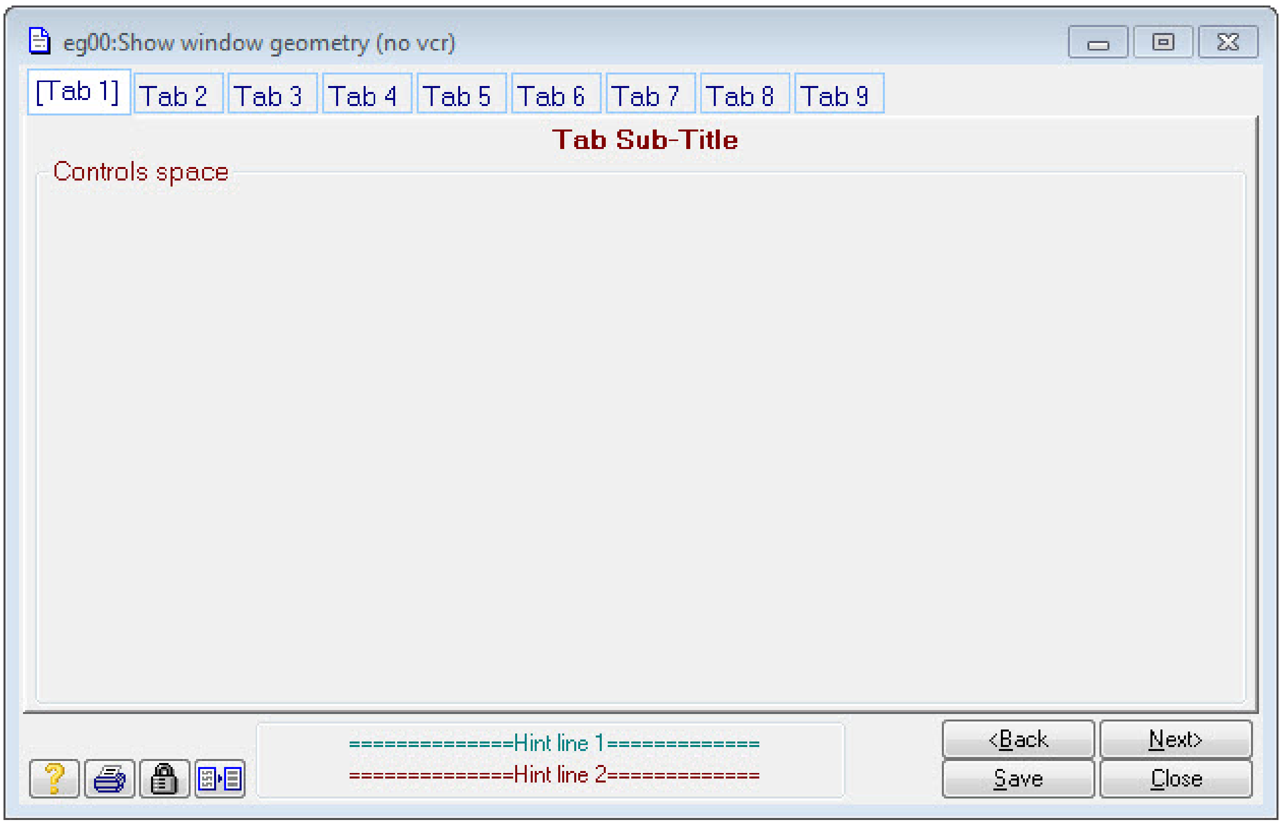
Properties (write only):

window.reset(bool)	=	set true to reset all prompts to their initial value (or whatever value their source now contains) and clear all touched statii for <u>all</u> controls NB: this does NOT trigger control changed events
control.reset(bool)	=	if on a prompt, set true to reset to its initial value (or whatever value its source now contains); on all control types, set true to clear the touched status NB: this does NOT trigger a control changed event
control.select(bool)	=	set true to change the input focus to this control or: if a button, press it if a tab, show it if a key, make its key active
control.refresh(bool)	=	set true to force a list to refresh

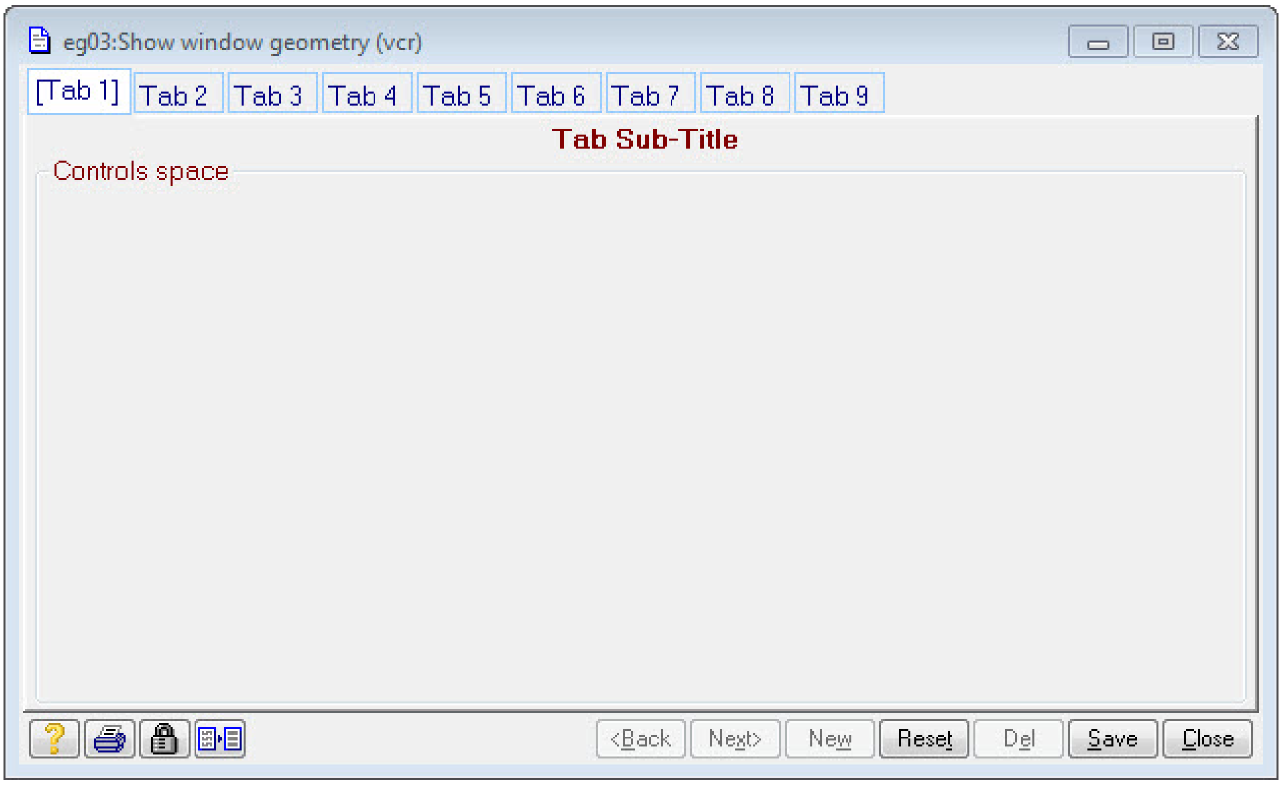
2.2.4 Window Layout

Controls are drawn in the order defined in the controls table.

When there is no VCR table, the spatial layout follows this pattern:


eg00:Show window geometry (no vcr)
[Tab 1]
Tab 2
Tab 3
Tab 4
Tab 5
Tab 6
Tab 7
Tab 8
Tab 9
Tab Sub-Title
Controls space
==============Hint line 1=============
==============Hint line 2=============
<Back
Next>
Save
Close

When the VCR table is present the layout is modified to this:

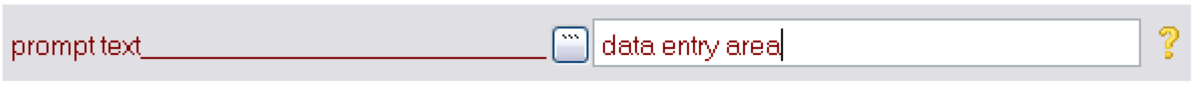


The controls are drawn in the controls space unless overridden by their *position*. The controls space has a fixed minimum size but will expand in both the X and Y direction to at least encompass the controls declared within it. Within this space there is the notion of a drawing 'position'.

The drawing position is moved on for each control drawn according to its prevailing *ruler*.

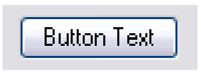
The various control types render as (the code for these samples is installed with the system in `configs\page_eg.lua`):

Prompt:



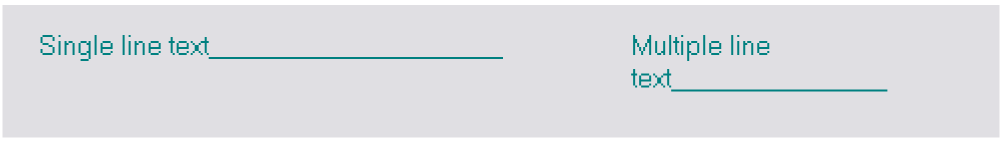
The width, relative position and presence of these elements can be controlled by a *ruler*. For the default ruler, the width of a prompt is 136 DUs (34 characters), the [...] is 10, the entry area is 136 and the [?] is 13.

Button:



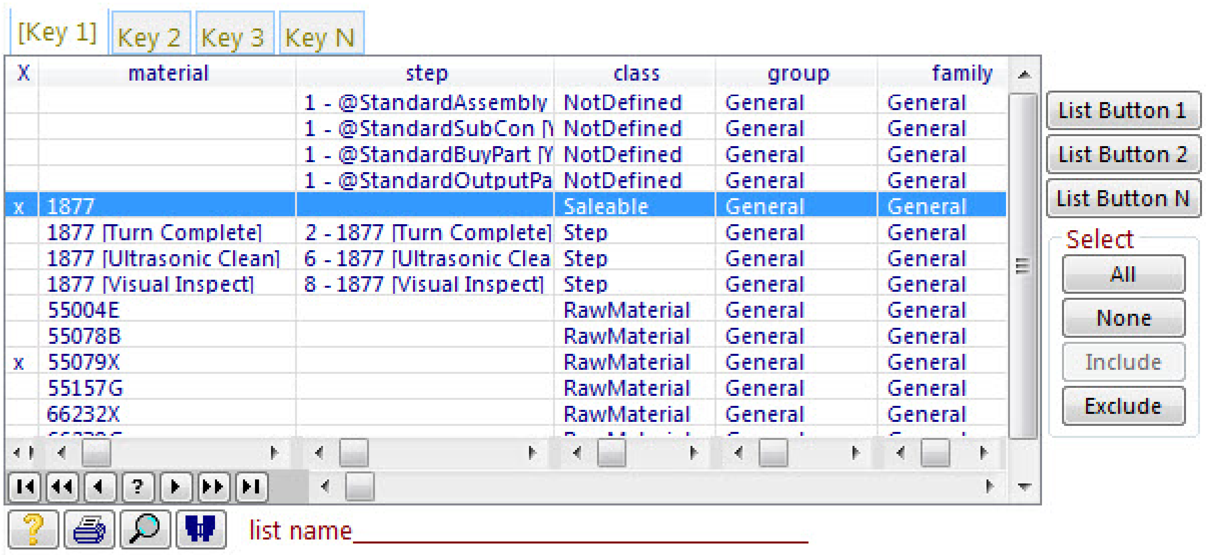
The width of a button can be controlled by a *ruler*. For the default ruler, the width is 40 DUs (10 characters). Buttons follow each other in-line unless overridden by a position of drPage_First or the *ruler*.

String:



The default width of a string is whatever is required to encompass the characters within it, the default height is a single line, strings follow each other in-line unless position 'first' or the ruler overrides it.

List:

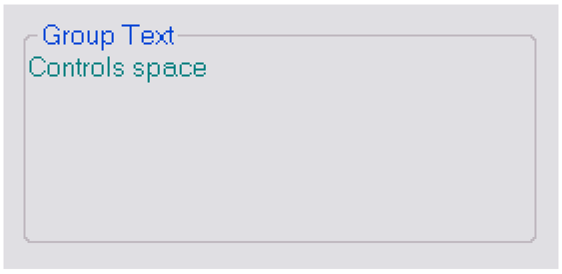


X	material	step	class	group	family
		1 - @StandardAssembly	NotDefined	General	General
		1 - @StandardSubCon [\	NotDefined	General	General
		1 - @StandardBuyPart [Y	NotDefined	General	General
		1 - @StandardOutputPa	NotDefined	General	General
x	1877		Saleable	General	General
	1877 [Turn Complete]	2 - 1877 [Turn Complete]	Step	General	General
	1877 [Ultrasonic Clean]	6 - 1877 [Ultrasonic Clea	Step	General	General
	1877 [Visual Inspect]	8 - 1877 [Visual Inspect]	Step	General	General
	55004E		RawMaterial	General	General
	55078B		RawMaterial	General	General
x	55079X		RawMaterial	General	General
	55157G		RawMaterial	General	General
	66232X		RawMaterial	General	General

The default size of a column is the size of its name. List buttons are aligned to the right of the list. The default list width (including its buttons) is the whole window width. The *where used* tool is drawn for virtual files but is disabled. The key change tabs are not drawn if only one key is declared for the list.

The 'X' column is present because the multi-select facility has been turned on, there is a helper function for this - `Selector()` in the 'utils' library, use `require('utils')` to access it. Just put `Selector('M')` in the fields table for the list to enable it and `Selector('S')` in its buttons table to provide the 'Select' button group. See `configs\page_eg.lua` for a code example.

Group:



The size and position of the group is determined by its *ruler*. NB: The enclosing box is not drawn if the group has no text.

It's the window designers responsibility to ensure the enclosed controls fit (although this does not matter if the group has no text).

2.3 How do I access properties?

Properties of the window and the controls within it can be read and written from the callback functions in response to user actions.

Static properties (i.e. those that do not change while the window is running) can be accessed directly using the normal Lua table syntax.

Dynamic properties (i.e. those whose value may change) can be read or written using the prop() function. Its syntax is:

```
Value = m.prop(Control,[id.]PropertyName) --read a property
m.prop(Control,[id.]PropertyName,Value) --write a property
```

Control is a control as passed to a call back function from page() or a direct reference to any control table in an open window.

Id is the id of the control to be affected, if omitted or empty, self is assumed. When present it must match the 'id' field of some control in the window or be one of these special values (must be lower case):

window.id	to mean the window itself (e.g. 'mywindow' if the window.id property is 'mywindow')
'window'	to mean the window itself
'tabthis'	to mean the currently selected tab, i.e. 'self'
'tabback'	to mean the first enabled/visible tab before the current one, or self if there isn't one
'tabnext'	to mean the first enabled/visible tab after the current one, or self if there isn't one

No match throws an error. The id itself may not contain a '.' character. If there are multiple controls with the same id, which one is found is undefined.

PropertyName is the name of a property to be accessed. If the name is unknown an error is thrown.

Value is the value to set or returned. Its type is implied by the property.

If this function is called when the page is not being shown, an error is thrown. If the page is being shown, property changes are reflected in the page as well as in the control table(s).

When setting properties, the page is assumed to be the current Clarion TARGET (i.e. make sure you haven't opened a monitor in the callback!).

See the Callback Interface description above for the properties available.

2.4 How do I test a user interface?

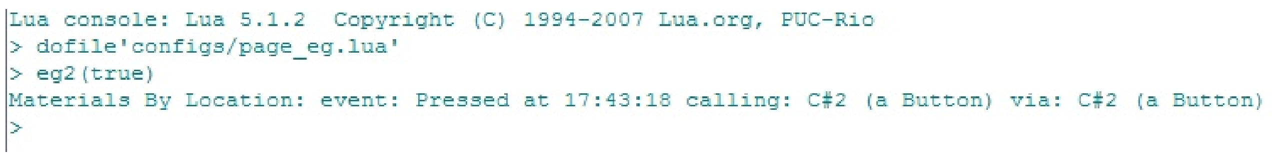
The m.page() function can take a second optional parameter that is a boolean. If this is 'true' then the page is drawn but its behaviour is modified as follows:

1. No callbacks are performed, instead a message is sent to the console describing what callback would have been called (so you can verify it will be called).
2. The 'hide' attribute is ignored, so all controls are visible (to allow size/position verification).
3. Any context where a callback would return a result (e.g. filling a calculated column), returns a NIL instead.

Using this 'preview' parameter allows you to edit the window definition, test it and edit again very quickly. To use this technique you must include the following line in your script before you call `m.page()`:

```
require'console'
```

With the example window given in What is a User Interface? the console session would look like this after you pressed the **`Select`** button:



```
Lua console: Lua 5.1.2  Copyright (C) 1994-2007 Lua.org, PUC-Rio
> dofile'configs/page_eg.lua'
> eg2(true)
Materials By Location: event: Pressed at 17:43:18 calling: C#2 (a Button) via: C#2 (a Button)
>
```

A useful editor to use is *Notepad++*, this is free to download from http://notepad-plus.sourceforge.net/uk/site.htm and has the useful feature of being able to 'fold' a Lua program file as well as syntax highlight it.

3 Accounting Software Interfaces

This topic describes how to write an interface to an accounting system. The accounts system interface is written as a *Lua* script so you need to be be familiar with *Lua* and how it is integrated into Match-IT (see the *Lua* topic). The description here is only concerned with the interaction between your script and Match-IT, it does not cover the specifics of any particular accounting system. Refer to your accounting system documentation for that.

The accounts system interface is responsible for the management of the information transfer from Match-IT to the accounts system and vice versa. Information that can be sent to your accounts system is new customer and supplier accounts, new stock codes, sales invoices, purchase invoices and stock movements. Information that can be extracted from your accounts system is customer and supplier address changes and their current credit position.

3.1 How is the interface organised?

There are four parts to the interface: the accounts centre record, a generic management script, the script you will write to this specification and your accounting system.

The accounts centre record

This record is used to specify that you are going to use a scripted interface as described in this document. It is also a 'holder' for other configuration information about the interface.

The list of available centres is accessable from the standard menu via `Functions | Standing Data | Accounts | Accounts Centres`.

Press **`New`** to create a new centre, or **`Detail`** to edit an existing one.

To use the interface described in this document the accounts centre record must have the `Generic` accounts system option checked (note: this is true even if the accounts system you are intending to use appears as one of the other options - they are specific versions of the interface that you cannot change).

On the `options` tab, you must also specify the name of the script you are about to write. It's best to use a name that describes the accounts system. Just enter a name with no path and no extension. Match-IT will automatically add the appropriate path and extension as needed.

The Generic Management Script

This script works in the background and provides facilities to ease the writing of your interface, particularly when the interface uses CSV files. Those facilities will be described elsewhere. The script is actually the file `accounts_interface.lua` in your `configs` folder.

Your Script

This is the script you will write, using the *Lua* language, according to the rules described in this topic. The script must be placed in your `configs` folder with the name you used in the accounts centre record and with a `.lua` extension. You can use any text editor to create it (e.g. notepad or wordpad, but not a word processor).

This script is the 'glue' between Match-IT and your accounts system. It is the only place where knowledge of your accounts system resides.

Your Accounting System

This is any accounting system that provides a means to import/export information. The simplest, and most common, mechanism is CSV files; but anything is possible, including COM, direct DLL calls, direct database access, etc.

3.2 How is the script organised?

Your script must implement whatever logic is necessary to export information to and import information from your accounting system. The specifics of that are beyond this topic. To achieve the interface your script must provide a number of standard functions with standard names. These functions are called by Match-IT when an import or export action is required. The action required is implied by the function called. If your interface does not provide an action, just do not implement that function and do not publish the capability (see *GetCapability* below).

The full set of standard functions is given below. You must define the functions with **EXACTLY** the name given below, <u>including</u> the case (e.g. *Setup* is <u>not</u> the same as *setup*). All these functions take as their first parameter an object that provides access to the generic management facilities. This parameter is conventionally named: `Interface`. Depending on the function, there may be other parameters as well. The functions must return a numeric result code that is non-zero to indicate success, or 0 to indicate failure. Alternatively, failure can be indicated by raising an error (by using the *Lua* `error` or `assert` functions). See the Reference section for a description of the helper facilities available to these functions. Depending on the function, a second string result can be returned.

function setup(Interface)

This is called when the user requests to setup the interface. It typically asks questions about the configuration and saves the results as qualifiers attached to the accounts centre record. Such qualifiers are available as properties of the interface (see *Interface:GetProperties*).

function exportCUS(Interface,RecNoQ)

This is called when the user has requested that new customer accounts be sent to your accounting system. The `RecNoQ` parameter will contain a list of customer records to be exported. It is an object of type `RecNoQ` containing entries of type `Customer`.

The function's responsibility is to traverse this list (using `m.members`), extract the required information and export it. On a success return from this function all the entries in the `RecNoQ` are automatically tagged within Match-IT as having been exported. If you do not want a particular record to be so tagged, remove it from the list before returning (using `m.exclude`).

function exportSUP(Interface,RecNoQ)

This is the same as `exportCUS` except it is dealing with suppliers (Q entry type is `Supplier`).

function exportSI(Interface,RecNoQ)

This is called when the user has requested that new sales invoices be sent to your accounting system. The `RecNoQ` parameter will contain a list of invoices to be exported. It's an object of type `RecNoQ` containing entries of type `SInvoice`.

The function's responsibility is to traverse this list, collate all the invoice lines as appropriate (use `m.sdGetInvoiceLines` to get the lines associated with any invoice), and export them.

On a success return from this function all the entries in the `RecNoQ` are automatically tagged within Match-IT as having been exported. If you do not want a particular record to be so tagged, remove it from the list before returning (using `m.exclude`).

function exportPI(Interface,RecNoQ)

This is the same as `exportSI` except it is dealing with purchase invoices (Q entry type is `PInvoice`, use `m.poGetInvoiceLines` to get purchase invoice lines).

function importCUS(Interface)

This is called when the user has requested that customer account details be imported into Match-IT.

The function's responsibility is to extract all account information from your accounting system and either create new records or update existing records in Match-IT as appropriate.

function importSUP(Interface)

This is the same as `importCUS` except the user has requested supplier account details.

function importCBAL(Interface)

This is called when the user has requested that customer credit balances be imported into Match-IT.

The function's responsibility is to extract all account credit information from your accounting system and update records in Match-IT as appropriate.

function importSBAL(Interface)

This is the same as `importCBAL` except the user has requested supplier credit balances.

function exportMAT(Interface,RecNoQ)

This function is called when the user has requested that new stock codes be sent to your accounting system. The `RecNoQ` parameter will contain a list of stock codes to be exported. It's an object of type `RecNoQ` containing entries of type `Material`.

The function's responsibility is to traverse this list (using `m.members`), extract the required information and export it. On a success return from this function all the entries in the `RecNoQ` are automatically tagged within Match-IT as having been exported. If you do not want a particular record to be so tagged, remove it from the list before returning (using `m.exclude`).

function exportMOVE(Interface,RecNoQ)

This function is called when the user has requested that new stock movements be sent to your accounting system. The `RecNoQ` parameter will contain a list of stock movements to be exported. It's an object of type `RecNoQ` containing entries of type `Movement`.

The function's responsibility is to traverse this list, extract the appropriate information from Match-IT and send it to your accounting system. On a success return from this function all the entries in the `RecNoQ` are automatically tagged within Match-IT as having been exported. If you do not want a particular record to be so tagged, remove it from the list before returning (using `m.exclude`).

function MakeAccountRef(Interface,Ignore,CusSup)

This function is called when either Match-IT itself, or the user, wants to create a new account reference for a customer or supplier record. The second parameter should be ignored. The third parameter may be missing or it may be an empty string. If present and not empty, it will be a string that should be used to create an initial code if possible. In either case the new account reference should be created to be consistent with the current contents of the `CSH` file buffer (i.e. do not open the file, just reference its fields as `m.csh.<field>`).

The function's responsibility is to populate the `csh.AccountRef` field with a code that is unique and conforms to the conventions of your accounting system.

function GetCapability(Interface)

This function is called when Match-IT needs to know what functions are implemented by the interface. The function must return two results, a non-zero number (to indicate success) and a string of concatenated capability codes. The capability codes have mnemonics of the form `csAccCan…`; they are all listed in the Reference section under Constants. E.g. if your interface just implements the exporting of customer accounts and sales invoices then the function return statement would look like this:

```
        return 1,m.csAccCanExportCustomers..m.csAccCanExportSalesInvoices
```

function GetDescription(Interface)

This function is called when you ask to see the description of an interface from the `Accounts Centre` form. The function must return two results, a non-zero number (to indicate success) and a string describing the interface sufficiently for anyone probing the interfaces to recognise they have selected the one they want. A typical response would be this:

```
return 1,’This interface exports information to the ’..
       ‘XXXX accounts package’
```

3.3 Interface Reference

Below is described all the facilities available through the Interface object passed to the functions listed in How is the interface organised.

Useful Constants

```
m.yes = m.flag('Yes')
m.no = m.flag('No')
m.emptystr = m.str('')
m.null = m.void()
m.zero = m.real(0)
```

Interface:LoadDefaults(CusSup)

Load the customer or supplier specific defaults for the customer or supplier defined by `CusSup`.

Interface:UnLoadDefaults()

Un-load the last set of customer/supplier defaults loaded. This is automatic if the script terminates or returns.

v1,v2,…vN = Interface:GetProperties(p1,p2,…pN)

Get one or more properties associated with the accounts centre involved. The parameters `p1,p2,…pN` is an arbitrary list of property names to get. The results `v1,v2,…vN` will be the value of those properties, they are returned in the same order as they were requested.

Properties are either predefined as part of the accounts centre record (i.e. `CSA` fields) or user defined as qualfiers. Attempting to get a property that is not defined will cause a user dialog to be shown to define it.

Each property can be specified in one of three ways:

- just as their name, e.g. '`version`',
- as a table with a field=value pair, e.g. `{version=6}`,
- as a table with name=field,value=expr, e.g. `{name='version',value=6}`

When a table is used, in either form, it can also have description=, validate= and ask= entries. If a value is given it defines a default value to use if the property has not got a value or is undefined. If a 'description' entry is given, it is used to describe the property if it is created due to it not being defined. If a 'validate' entry is given it is a function to call to validate the property. The function is given the current value as a parameter and must return nil if it is valid or a message describing what is wrong. If 'ask' is given and true, then the user is given a dialog to set the properties. The user is always asked to define property values that do not exist or are invalid.

f1,f2,…,fN = Interface:Open(mode,fn1,fn2,…fnN)

Open a set of files, either in isolation or in a transaction. mode is either '`r`' (read) or '`w`' (write), if mode is '`w`' the files are opened in a transaction.

The file buffers are cleared and populated with defaults, so the caller can be sloppy.

`fn1,fn2,…fnN` specifies the files to be opened (e.g. `m.csh, m.csc` to open the customer/supplier file and the contacts file).

`f1,f2,…fN` are open file handles for the files requested. They are returned in the same order as requested.

Interface:Load(file,object,watch)

Like `m.load` but aborts on error (so you don't have to worry about catching errors).

CusSup = Interface:Add(type)

Add the CSH currently in the file buffer. The `type` specifies a customer (`m.csTypeCustomer`) or a supplier (`m.csTypeSupplier`) is being added. If no type is given, then the same as last time is used. If `csh.CusSup` in the file buffer clashes with an existing record, it will be changed to a non-clashing value.

It returns the CusSup created on success or aborts on failure.

Contact = Interface:AddCon(type,CusSup,rank,nodef)

Add the CSC currently in the file buffer and create a default contact unless directed not to (by setting `nodef` to true). If no `type`, `CusSup` or `rank` is given, then that of the last add/get/put is used.

It returns the Contact created on success, nil if the contact already exists or aborts on failure.

CusSup = Interface:Get(type,AccountRef)

Attempt to get a cus/sup account via the given `AccountRef`. Set `type` to `m.csTypeCustomer` to get a customer or `m.csTypeSupplier` to get a supplier.

It returns the CusSup found if it exists or nil if it doesn't.

Contact = Interface:GetCon(type,CusSup,rank)

Attempt to get a contact for the given `CusSup` via the given `rank`. Set `type` to `m.csTypeCustomer` to get a customer contact or `m.csTypeSupplier` to get a supplier contact. If no `type`, `CusSup` or `rank` is given, then that of the last add/get/put is used.

It returns the Contact if exists or nil if it doesn't.

CusSup = Interface:Put()

Put the current CSH file buffer into the file.It returns the CusSup on success or aborts on failure.

Contact = Interface:PutCon()

Put the current CSC file buffer into the file.

It returns the Contact on success or aborts on failure.

Interface:Close()

Close all files and transactions previously opened with `Interface:Open`.

If a transaction is closed and it fails, the script aborts.

String = Interface:Join(lines)

Joins strings together as lines with trailing spaces clipped.

`lines` is a table containing the strings to be joined. Each entry is concatenated with the previous with a newline sequence. Trailing spaces from each entry are removed.

ExportTable = Interface:NewExportTable(Name,Description,Separator)

Create a new table to be exported. This creates an object with name '`Name`' to encapsulate the information to be exported. The '`Description`' is purely for documentation purposes. The ‘`Separator`’ specifies the field separator character to be used in the output CSV file. If omitted, the default is a comma. It should be a single character.

The object created here implements various helper functions for use by the interface specific code. A property in the interface of the table name must exist, and be populated, that is a `PathNam` unless the given table name starts with '`@`' in which case it's interpreted as the file name itself.

If the interface table name property does not exist, the user is asked to define it (see `Interface:GetProperties`).

See Export Table Reference, for the functions available through this object.

ImportTable = Interface:NewImportTable(Name,Description)

Create a new table to be imported. This creates an object with name '`Name`' to encapsulate the control of the import process. The '`Description`' is purely for documentation purposes.

The object created here implements various helper functions for use by the interface specific code. A property in the interface of the table name must exist, and be populated, that is a `PathNam` unless the given table name starts with '`@`' in which case it's interpreted as the file name itself.

If the interface table name property does not exist, the user is asked to defined it (see `Interface:GetProperties`).

See Import Table Reference, for the functions available through this object.

3.4 Export Table Reference

The `Interface:NewExportTable` function creates an object that provides facilities to assist in the export of information to CSV files. These facilities are described below.

ExportTable:field(Name,Format,Default,Description)

Define a field in the export table. Fields should be defined in the order they must appear in the output file.

`Name` is the name of the field. `Format` is how to render it, options are anything acceptable to `string.format()`. `Default` is the value to export when not explicitly set in the record, it's subjected to the formatting defined above. `Description` is purely for documentation purposes.

A field `Name` can be in any of the following forms:

- `group.name[index]`
- `group.name`
- `name[index]`
- `name`

Where `index` is numeric. This is useful when the output record consists of sub-records and arrays.

If the `Format` is nil or '', it means this field should be skipped in the output. This is useful in conjunction with variant records (see `fieldis` below) when each variant has a different number of output fields. Just add 'skipped' fields to make each variant have the same number of field definitions.

Each field must be defined before the first call to `ExportTable:add()`

FieldNumber = ExportTable: fieldis(number)

Set and/or get the last field number. This is used to create field definition variants, like this:

```
mark = fieldis()     --get field number of last field
field('f1')
field{'f2')
fieldis(mark) --reset to prior to 'f1' field#
field('f3')          --'f3' is now an alternative definition for the same field# as 'f1'
field('f4')          --'f4' is now an alternative definition for the same field# as 'f2'
```

It's the caller's responsibility to ensure everything lines up appropriately.

ExportTable:add(Fields[,Raw])

Add the given field buffer to the output record list for the table. `Fields` must be a table of field name/value pairs, where each field name must have been defined by a prior call to `ExportTable:field()`. Name/value pairs in `Fields` but not defined by a `:field()` call are ignored. A field defined but not present is populated with its default value.

If `Raw` is present and `true` the given fields are added to the output as is, otherwise they will be quoted if they contain embedded quotes or separators.

3.5 Import Table Reference

The `Interface:NewImportTable` function creates an object that provides facilities to assist in the import of information from CSV files. These facilities are described below.

ImportTable:field(Name,Format,Default,Description)

Define a field in the import table. Fields should be defined in the order they must appear in the input file. `Name` is the name of the field. `Format` is how to render default values, options are anything acceptable to `string.format()`. `Default` is the value to assume when not explicitly set in the record, it's subjected to the formatting defined above. `Description` is purely for documentation purposes. Each field must be defined before the first call to `ImportTable:records()`

ImportTable:records()

This is an iterator for the input CSV file. Use it in a for loop like this:

```
for record in table:records() do
       …  whatever
end
```

3.6 Example

A complete and real example of a script that conforms to the specifications described in this topic can be found in your `configs` folder as `SageL50.lua`. This script implements an interface to the *Sage Line 50* accounting system using CSV files.

4 Extending Report Files

The term *Report File* in this context is referring to the files that contain a relevant snapshot of the Match-IT databases for the purposes of printing something. Such files are also referred to as '`xTx`' (pronounced ex-tee-ex) files because their three letter acronym has a 'T' as the middle character. Whenever a document is needed to be printed a set of `xTx` records is created containing relevant information, for example for a sales order confirmation an `STH` record is created for the header information and an `STL` record for each line in the order. These `xTx` names are what you can see in the variables in the document designer, and expanding them shows all the fields available.

Although the set of fields available in the `xTx` records is extensive, it does not cover everything possible. However, should you find you want to print something that is not available in the `xTx` record, there is a mechanism that allows you to add it.

This section describes that mechanism.

Qualifiers

The mechanism exploits the qualifier system within Match-IT. This provides for the ability to add arbitrary fields of your own to almost any Match-IT database. In this context the fields are going to be added to the `xTx` records.

Steps

To use the mechanism there are four steps that must be completed:

1. Add the qualifier names to the `xTx` records
2. Write a *Lua* script to create qualifiers
3. Tell Match-IT where that script is
4. Modify the report paper design to use your new fields

Step 1 - add qualifier names

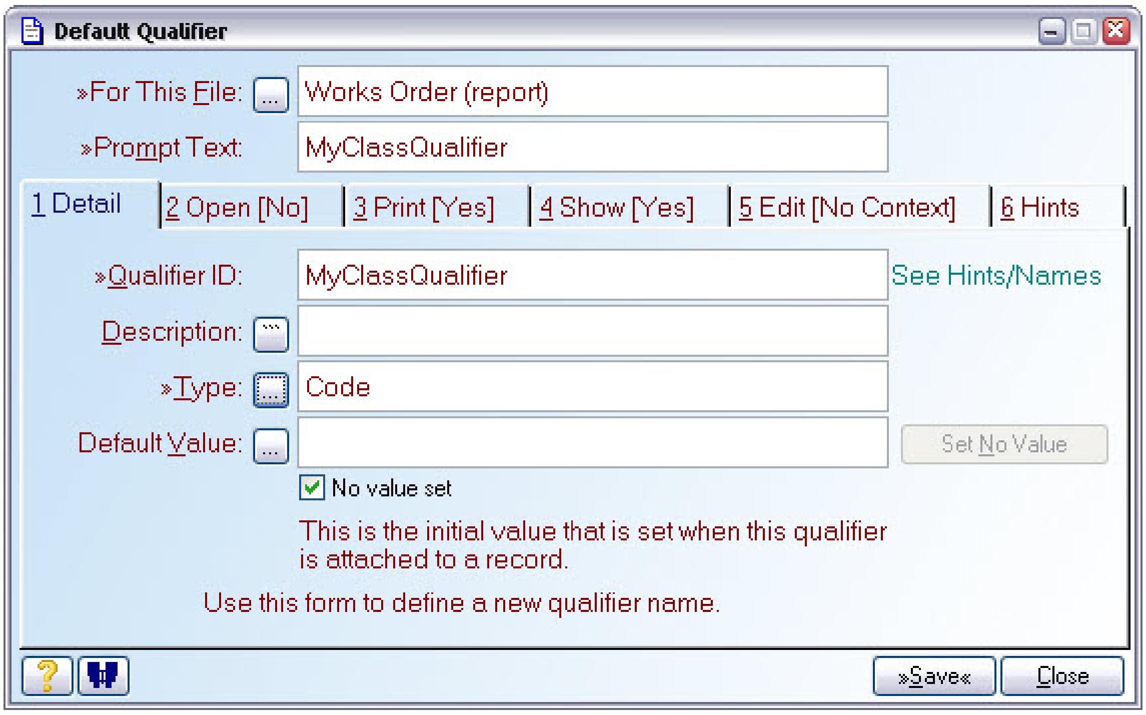
In order that your extra fields are visible to the document designer they must be added as a vocabulary to the appropriate `xTx` record. This can either be done using the `qaMakeField` 'drop' in some setup script, or manually via the `Qualifier Maintenance` form. The file the qualifier is to be attached to, the name of the qualifier and the type of information it will hold must be specified. Giving it a default value is not necessary.

The following is an example of creating the vocabulary using the `qaMakeField` 'drop' within a script:

```
m.qaMakeField(m.wth,'MyClassQualifier',m.Kode ,m.pack(m.void()))
```

This creates a field called `MyClassQualiifer` that will contain a `Kode` and attaches it to the `WTH` file.

This is what the form would look like to enter the same qualifier using the manual method:



Default Qualifier

»For This File:	Works Order (report)	
»Prompt Text:	MyClassQualifier	

1 Detail | 2 Open [No] | 3 Print [Yes] | 4 Show [Yes] | 5 Edit [No Context] | 6 Hints

»Qualifier ID:	MyClassQualifier	See Hints/Names
Description:		
»Type:	Code	
Default Value:		Set No Value

✓ No value set

This is the initial value that is set when this qualifier is attached to a record.

Use this form to define a new qualifier name.

Step 2 - write the *Lua* script

The *Lua* script to perform the extension must consist of functions with names of the form `qualifyTLA`, where `TLA` is the label of the report file that is to be extended, e.g. `qualifyWTH` to extend the `WTH` file. The existence of the function enables the facility for that file. The function is passed eight parameters when it is called, in order they are:

`dFile`	The file number of the destination file. This will be the same as the TLA in the functions name. The type is a `FileNo`.
`dRec`	The record number in that file. This is the actual xTx record that is being extended. This is of type `RecNo`.
`sFile`	The file number of the file that is providing the source for qualifiers being added. The type is a `FileNo`.
`sRec`	The record of the source file for qualifiers being added. This will be an object of the type implied by `sFile`. This means fields in the record can be accessed as `sRec.Field`
`cFile`	The 'context' file for the source. The context file is dependant on the source. For example, the context for the SOL (Sales Order Line) is the MCH (Material Catalogue Header) of the product being ordered in that line. This parameter is of type `FileNo`.
`cRec`	The record of the context file if there is one, else it's 0. This will be an object of the type implied by `cFile`. This means fields in the record can be accessed as `cRec.Field`
`PrintOnly`	A boolean that will be `true` if the qualifier set is being produced for printing purposes.
`VariantPrefix`	This is the variant prefix that is prevailing for the qualifier set being created. This is normally blank when printing.

The function is called whenever qualifiers are being set for the 'TLA' file.

The function must behave as a *Lua* coroutine, where it 'yields' back to Match-IT for each qualifier it wants to add and then terminates. To add a qualifier, the function populates the `QAT` file buffer with a record, then yields. It must just populate the `QAT` file buffer directly, it must <u>not</u> open it. To add fields to the file buffer use assign statements of the form:

```
m.qat.field = value
```

Where `field` is a valid field name in the `QAT` file, and `value` is some expression to assign to that field. As a minimum, the `Qualifier`, `Type`, `Value` and `IsPrinted` fields should be populated.

To yield back to Match-IT use the *Lua* `coroutine.yield` function with no parameters. To terminate, just `return` from the function. The proforma loop for the function is:

```
function qualifyTLA(dFile,dRec,sFile,sRec,cFile,cRec,PrintOnly,VariantPrefix)
    --initialise
    while NotDone do
        --build record in the QAT
        coroutine.yield()
    end
end
```

Step 3 - set the script name default

To tell Match-IT where the script resides you must set the `Qualifier extender Lua script` default (in the `QA` class) to the file you created in step 2.

Step 4 - modify the paper design

Your new fields will appear in the designer as qualifiers attached to the `xTx` record. Using them is the same as any other field and is fully described in the Document Design Tutorial manual.

Example

There is a very simple (and silly) example script installed with Match-IT that adds a few fields to the `WTH` record (works order header). It's located in `configs\qualify.lua`

5 Product Configurators

This topic describes how to set-up and use *Product Configurators* in Match-IT. A *Product Configurator* is a set of 'rules' that describe a product family. They can be useful in situations where you make a range of very similar products that can be tailored to suit a particular requirement. You provide the 'specification', or options, required by answering questions, then the *Product Configurator* creates a product structure according to the rules and options selected.

The questions asked and the rules are all defined by you when you design the *Product Configurator*. *Product Configurator* design is a complex task that requires in depth knowledge of the product family and the capabilities of Match-IT. The execution of the *Product Configurator*, i.e. actually creating a product structure, is a very simple task that can be performed by anyone.

5.1 What is a Product Configurator?

A *Product Configurator* is a set of 'rules' that describe a product family. The rules can include questions that are asked to get options and specification elements. The answers to the questions are used to construct product structures that are appropriate to the answers. For example, if a question asks *'What colour?'* then the product structure could be constructed to call up the appropriate colour paint.

Product Configurators are just a short-cut to creating product structures. The end result is the same as if you did it by hand.

5.2 Why would I use a Product Configurator?

Product Configurators are useful if you make a range of products that are made to order with the options and specifications selected or defined by your customers. In this situation a product configurator could construct the product structure for you much faster, and with less effort, than if you did it yourself. The product configurators can also 'capture' any specialist production knowledge. This means the production knowledge is not needed by the person creating the product structure through a product configurator.

There is significant thought and planning required to design a product configurator. This means they are not useful for creating product structures that are fixed with no variants.

5.3 What happens when I run a product configurator?

Running a product configurator is rather like answering a few questions from an expert. The expert then takes your answers and creates what you want from them. A typical session has three parts:

In part 1, you are prompted to supply answers to some questions.

In part 2, the product configurator analyses your answers and, in conjunction with the 'rules' built into the product configurator, creates all the information needed for the product structure. If you wish, you can review this information prior to creating the product structure.

In part 3, the information created in part 2 is translated into an equivalent product structure. Once this is done, the product structure appears in your catalogue just as if you had entered it manually. The manufacturing cost of the structure would also have been calculated for you.

5.4 How do I run a product configurator?

There are a number of ways to get to the product configurator system. For example, the **`Design`** button on the list of lines for an enquiry or sales order will take you there. There is also a **`Design`** button on the materials catalogue list, and in many other places too.

To run a product configurator: from the standard menu select `Favourites | Functions | Standing Data | Scripts | Run Any Wizard`. This will present you with a form that allows you to select the product configurator you wish to run. Select the product configurator by pressing the `Select Wizard` ... then press

`Select` to run it.

5.5 How do I design a product configurator?

Considerable thought and planning is required to design a product configurator. There are no hard and fast rules about how to go about it. Treat the following as an initial guideline and topic list. If you wish, you can commission your Match-IT supplier to help you. These are the major planning topics that should be considered before you actually start defining the product configurator.

Identify the need

The first task is to realise that a product configurator will help you. A product configurator is likely to be useful if you find yourself creating very similar product structures on a regular basis.

Identify product families

The next task is to identify how many product configurators you need. You could have one large product configurator that covered all your requirement or a number of more specialised ones. Generally, creating several specialised ones is more convenient because they will be smaller, simpler and run faster.

Identify the options

The only reason for using a product configurator is if you wish to design specific variants of something. This implies choices. All the choices should be identified. These will becomes the questions that are asked when the product configurator is run.

Identify the form of the product structure to be created

The result of running the product configurator will be a product structure. The form of the product structure will dictate the form of the associated product configurator. You will find it helpful to draw the product structure you intend to create for easy reference when defining the product configurator.

Manually create one to verify how it's done

You will also find it helps to create an example of the intended product manually first, noting what you did as you go. Essentially, all a product configurator does is generate the entries you just made in the various forms associated with creating a new product.

Identify the structure elements

Identify all the processes, resources, inspections, parts and services that are required for your product structure. Identify where they all fit in on your diagram of the product structure.

Identify any codes required

Identify any new codes and code classes you require for your product configurator. You could use codes to present the user with a limited set of named options in response to a question. For example, you could define a colour code that only allows a response of a standard colour name.

Identify any qualifiers required

Identify any qualifiers you wish to associate with the elements of your product structure. Qualifiers are useful to specialise something. For example, to define the surface finish of some material you might need.

Identify the structure records that must be created

All the elements of the product structure you are going to create with the product configurator are defined by records in files. For example, a component that goes into a 'widget' is defined to the system by a record in the `mcb:Assembly Structure` file. The product configurator will need to create all these records. Annotate you diagram with the records to be created.

Identify the field values that must be created

All the responses you made when you manually created an example of the product are defined to the system by fields in records. For example, the class of the product is defined by the `mch:Class` field. The product configurator will need to create all these fields. The files reference section in the on-line help system lists fields that must be considered. Annotate you diagram with the fields to be created.

Identify any look-up tables required

You may find that some elements of your product structure are dependent on choices made elsewhere. For example, which lathe you use to turn something may be dependent on the diameter of the 'widget' being turned. In these kinds of situations you will find that using a look-up table might be helpful (see How do I define a look-up table?). Note all these situations and what the dependencies are.

Identify the expressions required to create the field values

Many of the fields you identified earlier will need values that are calculated in some way from the choices selected. For example, the amount of time it takes to drill the holes in a printed circuit board is dependent (mostly) on how many holes have to be drilled. These calculations are defined by expressions. Note all the expressions required to define all your field values.

Define symbolic names for the records and fields to be created

Many of the records and fields you identified earlier will need to be given names for the purposes of referencing them within the product configurator. Inventing names will be one of the hardest parts of the overall design task. It helps if you define a 'convention' that allows you to invent names quickly. One possibility is to use a convention that mimics the hierarchy of the product and the fields involved. For example, if making a 'widget' requires a quantity of 'gizmo' then the symbolic name for that field could be: WidgetGizmoQuantity (joining capitalised words like this is a common programming convention).

5.6 How do I define a look-up table?

To define a new look-up table: from the standard menu select `Favourites | Functions | Standing Data | Scripts | Look-up Tables`.

This will present you with a list of all the tables that you have already defined, if any. The *First Dimension* and *Second Dimension* columns define the axes of the table, the actual table entries are defined by the *Qualifiers*. To create a new table press the **New** button.

This will present you with a form that allows you to define the dimensions of the table. Each dimension is defined by the value of something within the system. For example, a material type, a resource name, a code, etc. You first define the type of the value. For dimension 1 you do this by pressing the `Dimension 1 type` ... and selecting the type from the list, then you define the value by pressing the `Dimension 1 value` ... and selecting the value you want. The process is the same for dimension 2.

If you wish to define a look-up table that only requires one dimension, then just set both dimension 1 and dimension 2 to the same type and value.

Press **Save** to make your selection permanent. You define the value(s) of the table entry as qualifiers in the usual way on the `Items` tab.

Note: You can also define look-up tables 'on-the-fly'. If you reference a table entry in your product configurator that does not exist, then you will be given the opportunity to define it as it comes across it.

Example

Lets imagine you have an operation that involves wrapping something; and how long that operation takes is dependent on which machine you use and what type of wrapping material you use.

To set-up a table to do this, you would define dimension 1 as the machine, dimension 2 as the material (or the other way round) and an entry that defines how long it takes for that combination of machine and material. You would define similar entries for every combination of machine and material you wish to use.

You can either pre-set every combination, or you can do it 'on-the-fly'. If a product configurator references a combination that you haven't defined you will be given the opportunity to define as it is encountered.

5.7 How do I define a product configurator?

You do this using a text editor to create a Lua script and installing it. One simple sentance for this author, one big task for you!