Rekall Unbound

Mike Richardson

2.0.0

Abstract

Table of Contents

Chapter 1. Introduction

Welcome to *Rekall*. *Rekall* is a database front end [1] to a range of SQL database servers.

This manual applies to verion 1.0.5 of *Rekall*, but *Rekall* is in continuous development, so if there is a feture which you would like to see included, please let us know! The current version includes:

- Access to *MySQL* and *PostgreSQL* databases. It can also access *XBase* files with a restricted set of SQL. Further SQL database servers will be added in the future. A single *Rekall* "database" can access one or more SQL databases.

- Access to the underlying database types. As of version 1.0.5, *Rekall* provides access to almost all the types which the SQL database servers provide (*XBase* is rather restricitve in this respect).

- Table design, and changes in design, plus the ability to view and update table data. Common functionality can be accessed, such as single column indexing (although this area needs extending, for instance to provide multiple-column indexes).

- Form and report design and execution. Forms and reports can access data directly from database tables, via arbitrary SQL (subject to the restriction that *Rekall* can parse it) or from queries which are defined in *Rekall*; both can contain arbitrarily nested sub-forms (or sub-reports).

- Scripting using the *python* language. An interface to *Rekall* itself and to the SQL database servers is provided (so that the scripts can control form and report execution, and can access data in the database). *Rekall* also contains an interactive *python* debugger. Of course, you also have full access to whatever *python* modules are available on your system.

- Import and export of table data in a range of formats. In fact, *Rekall* generalises this as a copy operation, where the copy source and destinations can include database tables, SQL queries and text files.

The next chapter provides an overview of *Rekall*, and takes you on a tour of *Rekall*'s facilities for accessing tables, forms and reports. The examples are drawn from the sample demonstration databases which are available from the same places as *Rekall* itself.

All the screenshots in this manual are taken from *Rekall* running under KDE. The QT3-only version is identical, except for things like control styles. The Zaurus version is also similar, except that the dialogs are generally more compact, and various other tricks are employed to reduce the amount of screen space that is needed.

Please be aware that this manual is *not* intended as a tutorial or manual for SQL databases, but assumes at least a basic knowledge of such databases. In places, there are descriptions of how SQL databases work, but these descriptions are intended to provide context rather than information. There are any number of books available on database technology ans the SQL query language, and lots of resources available on the Web.

We hope that you will find *Rekall* a useful and powerful tool. We welcome all feedback, both complimentary and critical. Complimentary feedback is nice, obviously, but critical feedback points us in directions that users would like to see us going in, rather than directions that we only *think* users would like.

---

[1] In the manner of Microsoft Access®

Chapter 2. Overview and Tour

Table of Contents

This chapter introduces *Rekall*, and shows how *Rekall* accesses tables, forms and reports. The examples in this

chapter are drawn from the demonstration *Orders* database. This is a simple database containing information about clients, products and clients' orders for products.

The sections in this chapter on tables, forms, and so forth, present an overview of their use to manipulate and display data in the server database; *Rekall*'s corresponding design functions are described in the appropriate chapter, and are not covered here.

Accessing a Database Server

Rekall itself does not contain a database. Rather, it can access SQL databases such as *MySQL* and *PostgreSQL* via drivers. Since *Rekall* is really intended as a general-purpose front-end to these, it does not handle functions such as database creation and access control (although plugin modules to support specific SQL databases may become available).

Because of this, it is neccessary to set up a database which *Rekall* can use. The *Orders* database can be run on *MySQL*, *PostgreSQL* and *XBase*; there are instructions included which describe how to set up the database. If you use the *XBase* version you do not need access to any shared resources, but to run the *MySQL* or *PostgreSQL* versions you will need access to a corresponding server to which you have access.

A few words are needed to avoid confusion over the use of the word *database*. We will use it in two ways. Firstly, it is used to refer to a RDBMSs (Relational Database Management Systems) such as *MySQL*; the term *server database* will be used where needed to avoid ambiguity. Secondly, it is used to refer to the thing that a *Rekall* user will think of as a logically single database; here the term *Rekall database* will be used. Note that a *Rekall* database can access more than one server database.

The Rekall Database File

Rekall uses a single file to contain information about a *Rekall* database. This file contains information about *where* the database resides (for instance, the location, username and password for a *MySQL* database), but it *does not* store any actual data, nor does it contain information such as form or report definitions. Data is always stored in the server database, while forms and reports (and suchlike) are stored either in the server database or in files in the same directory as the information file. This file will normally have the extension *.rkl*, but for historical reasons the extension *.kdb* can be used.

Content-wise, the files have the same format, and are interchangable. Up to *Rekall* version 1.0.4 the data is stored in bar-separated format; from 1.0.5 is it stored in XML. In either case, you can look at it (or edit it, if you feel confident and want to move databases about). By the way, *all Rekall* objects, such as forms and reports, are stored as text files, usually XML, and can be viewed and edited; you could, for instance, write a program which dynamically generates a report definition.

As noted above, forms and reports, etc., can be stored either in a server database, or in the same directory as the information file. In the latter case, the files have names like *myform.xxx.frm* where *xxx* is the same as the *Rekall* information file extension [2]

SDI or MDI: That is the Question

Users coming from the Windows® world, particularly *Access*, will be familiar with the MDI model, where an application opens multiple document windows as children of a main application window. Those from the Unix® world will likely be more familiar with the SDI model, where an application opens separate documents in multiple top-level windows. [3]

Rekall can run in either MDI or SDI mode, although are some limitations with both, mainly relating to window

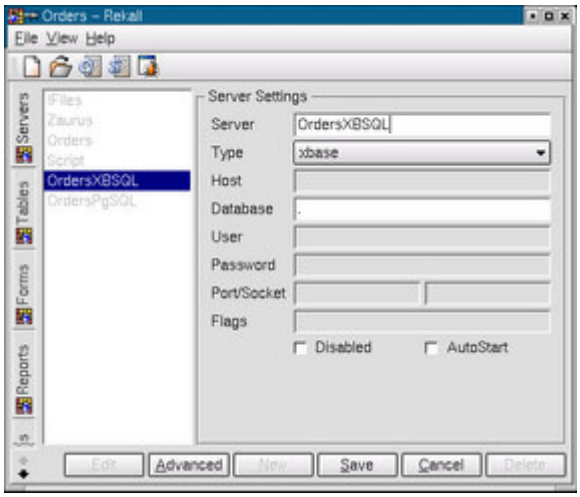
placement. *Rekall* can be started in MDI mode with the *--useMDI* argument, or *--useSDI* for SDI. Alternatively, it can be switched via the View/Options menu.

The Rekall Server Dialog

Normally, running *Rekall* and opening a database will display the server dialog. This lists the server databases which the *Rekall* database accesses, and allows you to set the information need to access them, such as location, usernames and passwords. Depending on the actual server database the exact interpretation of the values may vary a bit.

In MDI mode, the dialog window appears within the main MDI window; in SDI mode the top-level window is filled by the same dialog. In this manual, all the screenshots are takes from *Rekall* running in SDI mode (simply because it is slightly easier to generate them in this way).

The screenshow shows this dialog for the *Orders* database. The list on the left has entries for *MySQL*, *PostgreSQL* and *XBase* databases (called *Orders*, *OrdersPgSQL* and *OrdersXBSQL* respectively); plus an additional entry *!Files* which is used when forms and reports are stored in files rather than in a server database (and which is always present). The user has just selected the *OrdersXBSQL* entry and clicked the *Edit* button, to make changes to that entry. Note that entries that are not relevant to the particular server database are greyed out.



As well as the actual server database settings, there are two additional options. If *Disabled* is set, then *Rekall* will not access that entry; this can be useful, for instance, if a remote database is currently not available. The *AutoStart* option can be selected in order that a form (called *MainForm*) will automatically be opened when *Rekall* opens this database.

Finally, the tabs on the left-hand side provides access server database tables, forms, reports and so forth. The first, *Servers* shows this, the server dialog; the second shows tables in the server databases.

Advanced Settings

The *Advanced* button displays a pop-up dialog which contains more advanced settings. This popup will show one tab (the left-most) which contains common settings which apply to *Rekall* itself, and possibly other tabs which contain settings specific to the server database. The common settings are listed below; see the *database drivers* appendix for details of the server database specific settings.

Do not create Rekall-specific Tables

Rekall makes use of some private tables to store inforamation. By default, if it opens a server database and these tables to not exist, it will prompt to ask whether they should be created. Setting this option stops this check, and is

useful, for instance, if you have read-only access to a server database, or only wish to view tables.

Show all tables

Unless this option is set, *Rekall* will not list any tables which it creates for its own use.

Cache Table Information

In the normal course of operation, *Rekall* can quite often need information about a table (column names, types and so forth). By default, it will retrieve this information from the server database whenever it it needed. This has the advantage that such information is always up to date, so that if someone else changes a table, *Rekall* will see the changes next time it needs the inforamtion.

However, this may take some time on a heavily loaded or remote server database. Also, if the database design is fixed, then this table information will not change. Setting this option will tell *Rekall* to assume that table details do not change, and the information is cached.

Note this if you are the *only* person who is changing table details (adding or removing columns, changing column types, and such like) then it is safe to set this option, since *Rekall* keeps track of when it itself makes such a change.

Accept empty Username/Password

If you leave the undername or password field empty, *Rekall* will prompt for them when they are required to connect to a server database. However, in some cases, thay may not be needed. In this case, setting this option stops *Rekall* from displaying an unneccessary (and irritating) dialog.

Primary keys are Read-Only

Some server databases allow the user to update primary key values, while others do not. Where possible, the *Rekall* database drivers detect columns in tables, including primary key columns, which cannot be updated. However, it is sometimes useful to be able to prevent update of primary keys even where the server database does allow this. Setting this option will force *Rekall* to treat *all* primary columns as read-only, ie., as not updatable.

Fake Unique keys for Insertion

Appendix A discusses issues related to primary and unique key columns in tables and their affect on updating and inserting table data. Please see this appendix for details of this option.

Data Encoding

By default, *Rekall* assumes that text data is stored in the server database using *UTF-8* (*unicode*) encoding. It will retrieve and interpret textual data assuming this encoding, and will similarly store data. However, if you know or wish *Rekall* to use another encoding, then it can be set here. The available encodings are those handed by QT.

Note that specific server database drivers may detect the encoding from the server database where the server database itself supports encodings. This setting will override the server database setting.

Object Encoding

This setting similarly controls the encoding used when storing objects, such as forms and reports, in the database.

SSH Tunneling

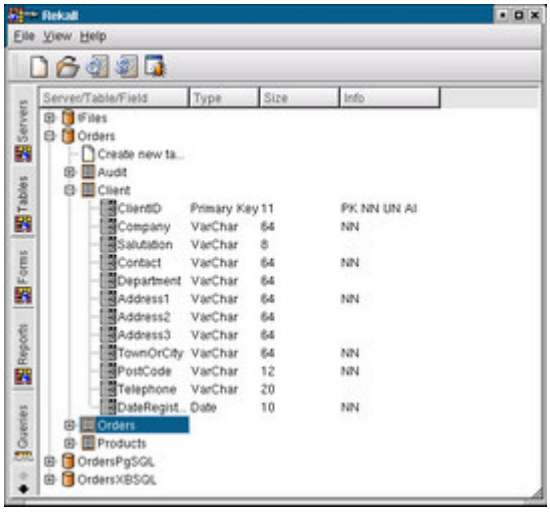
This setting (which is only available in the Linux versions of *Rekall*) allows the use of SSH tunneling where the server database is accessed over a TCP/IP connection. See chapter 14 for details.

Viewing Tables

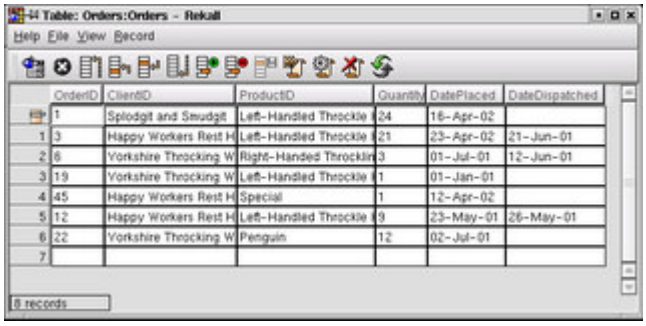
Selecting the *Tables* tab presents a view the tables in each server database. The view shows one entry for each server database, and possibly one for the *!Files* entry; expanding the tree for a server database shows all the tables in that database.

The first time you expand the tree for a particular server database (assuming that *Rekall* has not accessed the server database before), you will be prompted to create an objects table and a design dictionary. The first is used to store forms, reports, and suchlike, in the database, rather than in the file system. The second is used to store additional table information (for instance, default field values). You do not *need* to create these for *Rekall* to function, but if you do not then forms, etc., will have will not be available. This overview assumes that both are available.

Expanding the table itself shows a summary of the columns in a table. This cannot be used to modify the table structure, but can be useful if you need to remind yourself of the details of a table; the column names ans sizes, plus some basic properties, are shown. The next screenshot shows this; the branch for the *Orders MySQL* server database has been expanded, and within that the *Client* table. One thing to note is the *Primary Key* type; *Rekall* has an idea of a preferred column type to use for a primary key, and if a column fits its idea, then it is shown as such. Of course, this may not be your preferred type, and you can create primary key columns as *you* see fit.



Double-clicking on a table will show the table in data view, that is, it will show all the data in the table. You can also get to data view by right-clicking on the table and selecting the appropriate entry. The screenshow below shows table data view for the *Orders* table.



This should mostly be fairly obvious. Table data is shown in a grid, where columns correspond to columns in the table (and are labelled with the column name), and rows to rows of data from the table. The toolbar has buttons for record navigation (first record, previous record, ...), insertion and deletion, and searching. You can change column

widths, and alter the order (this is the display order, and does not change the table itself).

Values can be changed as you'd expect by editing the fields (or changing the selection in a combobox). Navigation between fields within a row can be done from the keyboard using the Enter and Tab keys (and Shift-Tab to move backwards); navigation between rows can similarly be accomplished with the Up and Down Cursor keys (and Ctrl-Up and Ctrl-Down move to the first or last record). Changes are saved whenever you change rows (or by using Ctrl-Enter).

A few things are worth pointing out here. All values are shown as simple text fields, but you can specify (via the design dictionary) that a column logically links to some other table, and that some value from the other table should be displayed in that column. In the *Orders* table, the *ClientID* and *ProductID* columns are linked to the *Client* and *Product* tables respectively, and show the client name or product description. If you click in such a column then the text field is replaced by a combobox which shows the possible values.

By default, text is displayed as it is returned from the server database, but specific formats can be given. Here, the date fields are formated as *dd-mmm-yy*. [4]

The left-most column shows row numbers, and will also display a markers for the current row. If you click in the left-column then the entire row is selected; you can also use the standard ctrl-click and shift-click methods to select multiple rows, which can then all be deleted at once.

And if you have a table with lots of columns, you can show the columns in lexical order by selecting *Order Columns* from the *View* menu (and switch back to table column order by repeating).

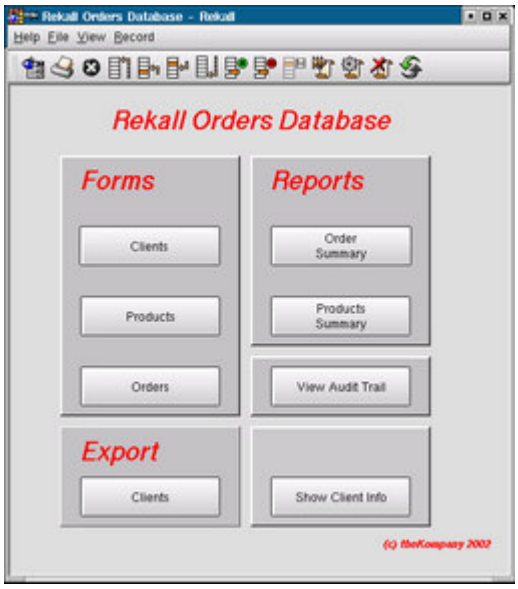
Forms

Although data can be entered, viewed and updated directly using table data view, it is much easier for users to do this using a suitably designed form.

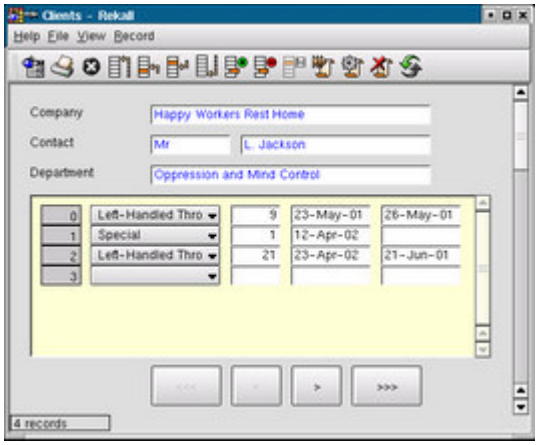
Forms are listed under the *Forms* tab. Here you will see an entry for each server database plus one more labeled *! Files*. If you expand the server database branches for each server database you will see the forms that are stored in that database (in the *Rekall* objects table); if you expand the files branch, you will see the forms that are stored in the file system. As mentioned earlier, forms are defined in XML, so it is quite easy look at a form definition store in the file system with your favourite editor. You can also look at those that are stored in a database, but you will need to export the definition to a file (right-click on the form and select *Save to File*).

One thing to remember about forms (and reports, etc.) is to be consistent. *Rekall* has the ability to open one form from another (typically when a button is clicked), but currently the form which is to be opened must be in the same place as the form from which it is opened (ie., both must be in the same directory in the file system, or in the same server database.)

The next screenshot shows the form *MainForm* from the *Orders* database. This is an example of a form that does not actually show any data from a server database. Instead, it just contains buttons that open other forms or reports, or do similar things. The form shows some of the visual effects that can be achieved - coloured text in different fonts and font sizes, and highlighted regions. The range of such effects that are available in *Rekall* is currently fairly limited, but you should be able to produce sufficiently attractive and usable forms.



Clicking on the *Orders* button on this form brings up the *Orders* form, which is shown in the next screehshot. This form displays orders, grouped up by client. It is an example of a form-subform structure, where the outer form shows some client information, and the inner form shows orders for that client.



This form shows some other features that are available in forms, such as scroll bars for rapid location of records and buttons that perform functions such as record navigation. Note that the buttons are in no way special, they simply invoke some *python* code that performs the required operations. Also, although you cannot see it from the screenshot, the client fields in the outer form are read-only, and cannot be updated by the user; the form has been designed to show the text in these fields in blue, so give some indication to the user.

The toolbar contains much the same controls as the table data view. In fact, the table data view display is actually just a form, abliet one that *Rekall* constructs on the fly to display exactly the data in the table.

Rekall Queries

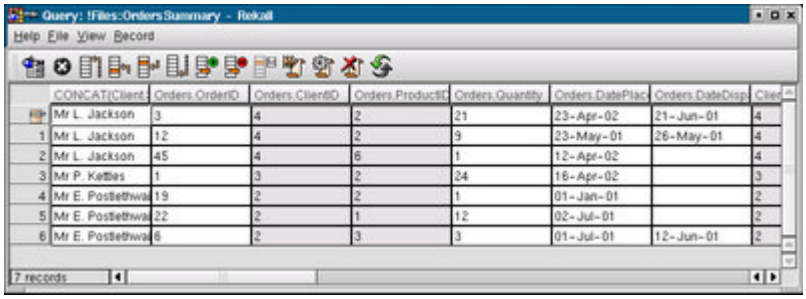
In the context of *Rekall*, the term *query* is ambiguous; there are at least tho ways it can be used. The most general used is as in an *SQL query*, ie., something like *select a, b from A, B where A.i = B.a*. *Rekall* constructs these internally whenever it needs to retrieve data from a table, or to update data in a table. It is also possible to enter free-text SQL queries to retrieve data.

The second use is *Rekall* specific, and is the subject of this section. In this usage, a *query* is an object that contains information about how data it to be retrieved from the server database. Generally, *query* will be used in this sense, and *SQL query* otherwise.

Rekall provides a GUI query designer, which allows you to select tables, to specify relationships between the tables, and to specify columns (or, more generally, expressions) which are to be retrieved from the tables. From this information, *Rekall* constructs an SQL query which is used to actually retrieve data. We will pass over the details of the designed here, and return to it in a later chaper.

A query can be displayed in data view, in which case the SQL query is executed and the data displayed, much as for a table. The screenshot below shows the results of running the *OrdersSummary* query in the *Orders* database. Columns which involved in explicit relationships between the tables are shown in grey, and cannot be changed; this is so that the user cannot make a change which would alter the combinations of rows returned. Apart from this, values can be updated in the usual way.

In this version of *Rekall* there are also restrictions on querys in relation to inserting and deleting rows. These are described in the chapter on queries.



Reports

Reports in *Rekall* are basically just the same as reports in any other database front end, and provide a way to display information from the database, possibly summarized in some way, and typically in a format that is suitable for printing.

The screenshot below shows a simple report, which is actually derived from the query shown in the previous section. This report in analagous to the orders form described earlier, in that it groups up orders by client, with information for each client output on a separate page. The report shows a few features of *Rekall* reports, such display of the date when the report was generated and the report page number, and summary values (in this case column totals).

Bear in mind that you will be able to generate output values using *python* scripts, so it is possible to produce a wide range of outputs. In this report, the date and the page number are actually produced by snippets of such code.

---

[2] If you need to, you can rename the main information file and any form or report files, to consistently change the extension.

[3] Actually, as well as the MDI-versus-SDI religious debate, there is a fair degree of disagreement over what does or does not constitute MDI or SDI. The usage here is not intended to be authoratative in any way.

[4] *Rekall* can display dates and times in any format supported by the C library *strftime* routine, and can also decode dates and times in most formats.

Chapter 3. Connecting to Database Servers

Table of Contents

Rekall does not itelf incorporate a database, as, for example does Microsoft Access with the Jet database engine. Although *Rekall* comes with an *XBase* library (which comprises the actual *XBase* library and a wrapper library which provides an SQL interface), this is not part of *Rekall*, and more that a *MySQL* or *PostgreSQL* server is.

Hence, *Rekall* must be told how to connect to a server database before it can do anything useful.

From *Rekall*'s point of view, everything starts at a single file which by has the extension *.rlk* [5]. This file is a simple text file, which, as of *Rekall* 1.0.5 is in XML format (prior versions uses a bar-delimited file, but this is automatically converted). The file only contains information about connections to server databases; in this respect it is nothing like the *.mdb* files created bu Access. You can copy the file or examine it with a text editor; you can even edit it by hand.

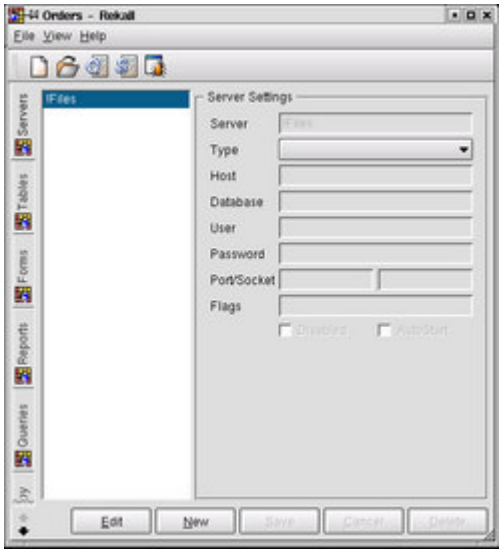
Rekall takes the view that this file is what the end-user thinks of as the database. Just as a user can open a word processing document in a word processor, or a speadsheet document in a spreadsheet program, this file can be thought of as a database document that is opened by a database program. Of course, the real server database may be being accessed from lots of places (over a network, through a web server, whatever). Actually, the *Rekall* database file could be opened by several instances of *Rekall* (for instance, if it is stored on a file server), but all issues relating to multiple concurrent access to data is handled by the server database.

The database file contains information about one or more server databases. Usually, it will contain information about a single server database, but there is no limit. In fact, if you have more than one server database, it is even possible to design forms that access data from more than one server database (but more of this later).

The Server Dialog

The first time you run *Rekall* you will get an empty window. Click the *New Database* tool (or select *File/New Database* from the menu), then select a directory and a file name, just as you would do for a new work processing or spreadsheet document; this gives the location and name of the database file. The one difference is that you should probably create a new directory and locate the database file there - why you will want to do this is explained a bit further on.

If you create a database file called *Orders*, then a dialog will appear in the window, as illustrated in the screenshot below. The tabs down the left-hand side correspond to the various components of *Rekall*; tables, forms, etc. The first tab, *Servers* displays the information about server databases, which is what is in the database file.



The !Files Entry

As was mentioned earlier, *Rekall* can store forms and reports and so forth either in a server database, or in the file system (or both). If they are stored in the file system then each form, report, etc., will be stored in a separate file, and these files will be located in the same directory as the database file (this is why you will probably want to place

the database file in a new directory). The dialog will always show an entry called *!Files*, which corresponds to forms, etc., that are stored in the file system. However, we need to specify at least one server database, otherwise there will not be any tables that *Rekall* can access!

At this point you have two choices. If you decide to store forms, etc., in the file system, then you can add a server database to the *!Files* entry. To do this, highlight the *!Files entry* and click *Edit* (or double-click the entry); this enables the fields on the right hand side. First, select the required type of database server, and then enter the details for your server database. The exact details depend on the actual database server (and fields which are not relevant to the selected type of server database will be greyed out), but generally they are as follows:

- *Host* is the machine where the server database is running. If this is the same machine as you are running *Rekall* on, then this entry can probably be left empty or set to *localhost*, otherwise it needs to be the name or IP address of the server.

- *Database* is the database name within the server database. Servers like *MySQL* and *PostgreSQL* support multiple logical databases, so you need to name the one you want.

- *User* and *Password* need to be set if you have to give a these to access the database.

- *Port*, *Socket* and *Flags* are dependant on the server database. The first is typically used for TCP/IP connections when the server database is listening on a non-standard port; the second is similarly used for local connections. Usually, you won't need to set any of these.

The *Disabled* checkbox can be set if you need to temporarly prevent *Rekall* from trying to access the server database. This may be useful if the database file specifies more than one server database and one of them is unavailable. Note that *Rekall* will set this automatically if it cannot connect to the server. The *AutoStart* option is used to arrange that a form is automatically opened when *Rekall* started up, and is explained in the chapter on forms.

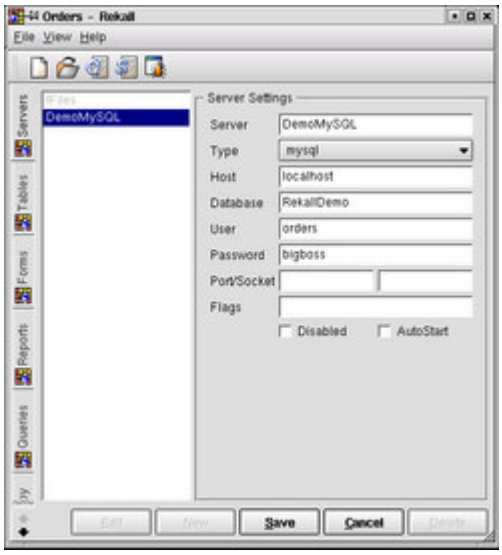
A special case is the *XBase* server database, which does not have a separate server process, and has no notion of users or passwords. In this case, *Database* is the name of the directory in which the *XBase* table files will be stored. As a special case, a period (.) can be used, meaning the same directory as the database file itself is located in.

If you leave the *user* or *password* settings empty (and the server database requires these values) then a dialog box will appear when *Rekall* first tries to connect to the server database. This is useful where you want to be sure that the *user* or *password* values cannot be discovered by someone else, since the values you enter are stored in plain text

One you are happy with the settings, click *Save*.

The second option is to store the forms, reports, etc., in a server database, in which case you need to click *New*. This also enables the right-hand fields, with the difference that this time the *Server* name field is also enabled. The server name is any name you care to use to identify the entry; you might for instance set it to be the same as the database name, but any value will do, *Rekall* isn't fussy. This time, when you save the settings, a new entry will appear with the server name.

The next screenshot shows the dialog with two entries, one for *!Files* and a second named *DemoMySQL* which accesses a database named *RekallDemo* with user name *orders* and password *bigboss* on a *MySQL* server database. The latter is being edited.



The effect of these choices will become more apparent when you move to one of the other tabs, such as Forms or Reports. Looking ahead slightly, this will show a tree view (in the manner of a file browser) with two levels; the top level comprises the servers defined here, and the next level shows the object - forms or reports or whatever - in that server.

There are two points to bear in mind. Firstly, *Rekall* cannot create databases not users with *MySQL* and *PostgreSQL* database servers, so you will need to do this yourself. The details of how you do this is outside the scope of this manual. Secondly, for security reasons, some server databases may not be installed out-of-the-box to accept all types of connections. For instance, *PostgreSQL* is generally not installed to accept TCP/IP connections, so if you want to connect *Rekall* to a *PostgreSQL* database on a remote machine, you may well need to change some settings on that machine (clue: have a look at the */var/lib/pgsql/data/pg_hba.conf* file).

The Rekall Objects and Design Tables

Before going any further, two special tables that *Rekall* creates and uses should be mentioned. *Rekall* will attempt to create these when it access a server database and finds that they do not exist (subject to checking with you that they should be created). The first table, *__RekallObjects*, is used to store objects such as forms and reports when they are stored in a server database. The second table, *__RekallDesign* is used to store additional table-related information, such as validation expressions and *Rekall*-level default values [6].

Normally, you should allow *Rekall* to create these when it asks. One time that you might not is if you are using *Rekall* simply to look at (and maybe modify) data that is stored in an existing tables, and where you will not be creating any forms or suchlike. In the event that *Rekall* cannot create the tables (maybe you have read-only access to the server database), it will warn you but continue [7].

And Now, the Real Thing ....

And with that, you should be connected to a server database, in which case it is time to get on to the real stuff, starting with ..... tables.

---

[5] For historical reasons, the extension *.kdb* is also available, but preferably should not be used.

[6] *Rekall* does not create the *__RekallObjects* table for the *!Files* entry, since in this case is does not store objects in

the database.

[7] There is a minor irritation that *Rekall* will ask you each time you start. There should be an option to stop *Rekall* from bothering at all about these tables.

Chapter 4. Accessing Tables with Rekall

Table of Contents

This chapter describes how to work with server database tables using *Rekall*; how to view and modify the data they contain, and how to create and modify (and delete) them. To start with, we'll look at how *Rekall* interacts with the types of data stored in server databases.

Data Types in Rekall and Servers

Relational databases (what most people would think of as an SQL database) store data in tables, where the a table contains columns each of which has a *type* (such as *integer* or *varchar*). *Rekall* maps these onto a set of internal types when data is read from a table, and back again when the data in a a table is updated. The table below is a list of these types:

Rekall Type	Usage
Integer	Used for whole numbers, like *42* or *1066*. This is represented by the hardware's natural integer value, so will almost certainly be a signed 32-bit number, which has a range of around plus-or-minus 2 billion (2000000000).
Float	Used for numbers with decimal points. Again, this is handled using the hardware's natural long floating point representation; almost certainly 64 bits. Note that *Rekall* does not currently have any specific internal support for *fixed* point numbers.
Date	This type holds a date, ie., a year, a month and a day. Valid dates start with the introduction of the Gregorian calendar in England (in 1752), and far as *Rekall* is concerned, the universe ends around 8000AD. If you database ceases to work after this date, you are welcome to notify us!
Time	Contains a time, ie., hours, minutes and seconds.
DateTime	Combines date and time.
String	This type is used to handle values that are returned from the database as strings of text characters. Typically these are things that can be thought of a printable, although there is no specific requirement for this. Also, strings are not required to be null-terminated (although life is often easier if they are). There are no length restrictions.
Binary	The binary type is rather like the string type, but is used for data which is typically not thought of as printable, for instance images. Again, there are no length restrictions (other than the usual things like

	availability of memory).
Boolean	True or false. This is an explicit truth value, although other types can interpreted as true or false (for instance non-zero numbers are true, and strings are converted to numbers).

When *Rekall* accesses a table in a server database, it identifies the type of each column retrieved (or, in general, each *expression* that it retrieves, this being relevant in forms and reports) and maps the server database type to one of the above internal types. The exact mapping depends on the particular server database (and the *database driver* that interfaces *Rekall* to the server database), but the table below shows typical mappings for common SQL types (both SQL92/SQL3 types and some server database specific types):

Rekall Type	**SQL Types**
Integer	smallint, int, integer
Float	float(p), real, double precision
Date	date
Time	time
DateTime	timestamp
String	char(n), varchar(n)
Binary	blob
Boolean	boolean, logical

In addition, *Rekall* defines two pseudo-types, *Primary Key* and *Foreign Key*. The first is the type that *Rekall* thinks is the most appropriate type for a primary key column, for instance for *MySQL* this is a (32 bit) integer column which is marked with the *MySQL* primary key and auto-increment attributes. The second is similarly for a foreign key, and is typically a (32 bit) non-null integer. This type is useful if you are not especially bothered about the type used for primary keys, and are happy to let *Rekall* make the decision for you [8]

Some server databases offer server-specific types. *PostgreSQL* is a notable example, having types for such things as geometric objects (points, line segements, and so forth) and computer networking (such as internet addresses). *Rekall* will generally treat these as *String* values, passing back values retrieved from the server database exactly as they are returned by the database select query. Similarly, when one of these values is updated in a table, the text string is passed as part of the update query.

Rekall will do its best to check values internally, for instance if it knows that a column contains an integer, it will ensure that only digits can be entered. This means that errors are detected and reported quite early on. However, in the case of any server-specific types, such checking is not done [9] so errors will only be reported as and when the server database reports and error when a query is presented to it. For instance, *PostgreSQL* will accept a point value in the format *(10,20)*, but *Rekall* will not prevent you from entering *(10.20)*, and an error will not be reported until *Rekall* tries to update the table. There is a way around this since input fields in forms can have arbitrary validation expressions associated with them, and you can also associate validation expressions with table columns (as explained below), but you will have to do this yourself.

Designing and Altering Tables

Selecting the Tables tab in the main database window presents a view of the tables in each server database; there should be a top-level entry for each server database (including one for the *!Files* entry). When the database contains any tables, these will be viewable by expanding the tree. However, if you have just created a new database on the server, the tree will expand to a single item, *Create new table*. Double click this to bring up the table designer.

The upper half of the table designer contains fields into which can be entered the table column *name*, the column

type, whether it is the *primary key* column, and a comment. The lower half contains fields which show information about the column selected in the upper half. Of these, the *length*, *precision*, *null OK*. *indexed* and *unique* fields are properties of the actual column in the table. The remainder are additional information which *Rekall* stores about the column (in the *__RekallDesign* table).

The standard sort of keystrokes can be used to move around the design form. Up and Down-Cursor move between rows; Enter, Tab and Shift-Tab move between fields (and between rows when at the last or first field in a row). The gray coloured bar to the left of the upper half indicates the status of each row (which in this case corresponds to a table column) as below; you can also right click in the bar to gain access to insert and delete operations.

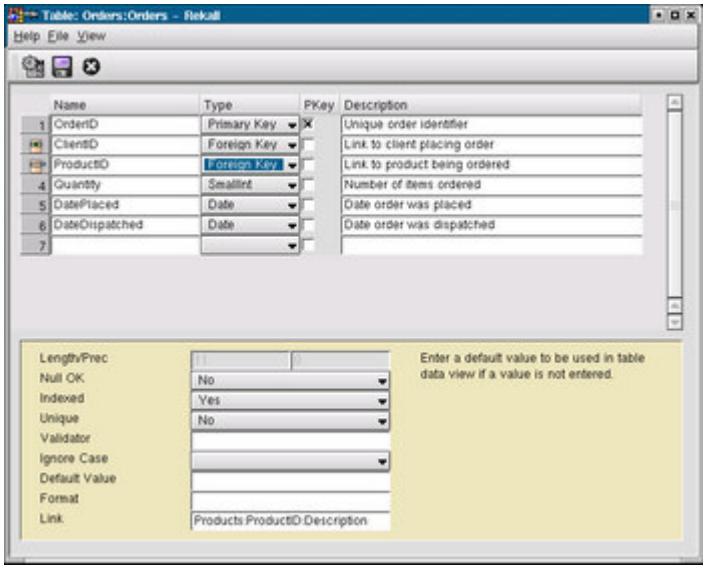
Marker	Meaning
	Current row
	Data changed
	Row marked for deletion
	Row inserted
	Current row but focus in another block

The table below shows the three database tables that are used in the *Orders* database. *Client* contains details about each client; *Products* contains information about each product; and *Orders* list clients' product orders.

Table Name	Column Name	Type	Length	Null OK?
Client	ClientID	Primary Key		
	Company	VarChar	64	No
	Salutation	VarChar	8	Yes
	Contact	VarChar	64	No
	Department	VarChar	64	Yes
	Address1	VarChar	64	No
	Address2	VarChar	64	Yes
	Address3	VarChar	64	Yes
	TownOrCity	VarChar	64	No
	PostCode	VarChar	12	No
	Telephone	VarChar	20	No
	DateRegistered	Date		No
Products	ProductID	Primary Key		
	Description	VarChar	80	No
	UnitCost	Double		No
	Stock	Integer		No
	DeliveryDate	Date		Yes
	Image	Blob		Yes
	Notes	Blob		Yes
Orders	OrderID	Primary Key		
	ClientID	Foreign Key		No
	ProductID	Foreign Key		No
	Quantity	SmallInt		No

	DatePlaced	Date		No
	DateDispatched	Date		Yes

The image below shows the table design screen for the *Orders* table, just before the table design is saved (by clicking the floppy-disk tool). You can see the use of the *Primary Key* and *Foreign Key* pseudo-types. Once the table design has been saved, the table will appear under the *Tables* tab; expanding the table there shows a summary of the table columns.



If you open an existing table in design view (or if you have just saved a table design) then *Rekall* examines the columns to see if any match with its notion of a primary key, and if so shows the type as *Primary Key*. However, this is not true of *Foreign Key*, so if you create a column with this pseudo-type, it will typically reappear as an integer. Note that, at present, *Foreign Key* is purely a design convenience, and no foreign key information is actually passed to the server database even if it does support this notion.

You can make changes to the table design and save them. *Rekall* will attempt to preserve table contents. Clearly, any data stored in a column that is dropped will be lost, otherwise values are converted (if possible) if the column type is changed [10] .

You may also have noticed some text in the field labelled *Link* at the bottom of the previous image. We will return to this later. Meanwhile, the left-hand most tool on the tool-bar is used to switch from table design view to table data view; when in data view, this is replaced by a small set-square tool, which switches back to design view.

Viewing and Updating Data in Tables

For table design view, clicking the table data view tool switches to data view (or go via the *View* menu). If you are not in design view, you can go straight to data view by double-clicking on the the table under the Tables tab, or by right-clicking on the table and selecting *Data View*. In data view you can view data, and usually update and insert (see the first appendix for a discussion of the issues involved); suffice to say here that if you created one of the column with the pseudo-type *Primary Key* then update and insert will be possible.

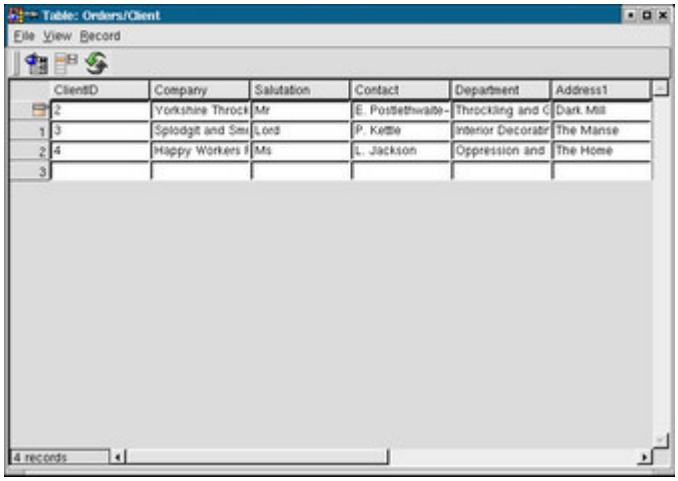
Data can be entered in the usual sort of way. Table data view is set up so that whenever you move between rows, any changes you have made to the row you are leaving are saved to the database. The contents of a field is checked for validity when you leave the field (but remember the comments about server-specific types make above); the contents of all the fields in a row are checked when you leave the row. There are tools on the tool-bar for the various record navigation operations; first, previous, left, last and so forth.

Tool Icon	Function
	Go to first record
	Go to previous record
	Go to next record
	Go to last record
	Insert new record
	Delete record
	Save current record
	Start search
	Execute search
	Cancel search
	Reload table

Clicking the *Start search* button will clear the current row. You can then enter search criteria, and click the *Execute search* button, whence *Rekall* will find and display those rows which match the criteria. As well as exact values (like *42* or *Fred*) you can enter expressions like *>12* into numeric columns and *F%* into text columns [11]

The table below shows the three database tables that are used in the *Orders* database. *Client* contains details about each client; *Products* contains information about each product; and *Orders* list clients' product orders.

The illustration below shows the *Client* table after a few rows of data have been added. As on the table design form, there is a scroll-bar which marks the current row and so forth.



Note that since the first column was specified as *Primary Key*, you do not need to enter a value; when the record is inserted, a value will be generated (in this example by *MySQL* via the *auto-increment* column type.

A row can be deleted by right-clicking in the left-hand most column and selecting delete, or by clicking in any field in the row and then clicking the delete tool in the toolbar. You can also select multiple rows for deletion using the normal Ctrl-Click and Shift-Click methods.

Other Table Design Settings

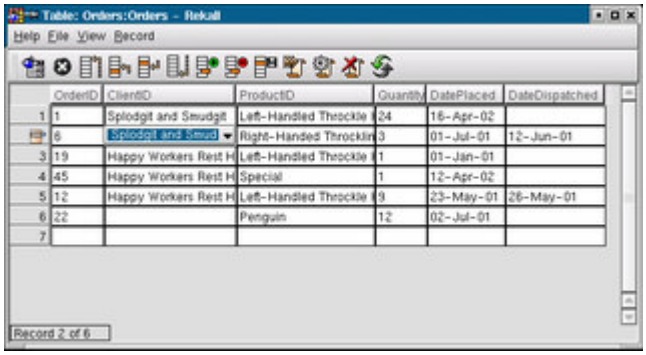
As well as the actual table as stored in the server database, *Rekall* can also store table design information such as default values and validation expressions. These are stored in objects with the *.inf* extension [12]. Note that these apply to data display and entry in table view; they are not passed to the server database and are not used when a form is created which accesses the table. The table below lists these:

Legend	
Validator	Validation expression used when entering data. This is a unanchored regular expression.
Ignore case	Case insensitive input validation
Default value	Value to be used if none given
Format	Data formatting for display. See below.
Link	Display data from a linked table. See below.

If you click in the *Format* or *Link* fields, then a small button will appear; clicking this will show a helper dialog which assists in setting an approproate value. Format setting is fairly straightforward, with the dialog showing appropriate types and sample formats. Numbers are formatted using C *fprintf* style format strings, while dates and times are formatted using *strftime* format strings. Please refer to an appropriate manual for further details.

The *link* setting needs more explanation. Suppose you have a column, say *ClientID*, which contains (foreign) key values which refer to clients stored in a table *Clients*, and you would like to display the client name rather than the key value.

To do this, the *link* setting would be something like *Client:ClientID:Name*, meaning to display the *Name* column from the *Client* table, where the *ClientID* value in that table is the same as the column value in this table. To make this easier, the helper dialog allows you to select the table to which to link, the column in that table which is matched to the (foreign) key column, and an expression to display. The screenshot below shows the *Orders* table; the *ClientID* and *ProductID* columns have been linked to the *Client* and *Product* tables, and the date columns have been formatted as *dd-mmm-yy*. Focus is in the *ClientID* column, so one row is showing a combobox.



One thing to note about linked table fields is that they appear as combo boxes when focus is in the field, but as plain text otherwise. This is an example of a *morphed* control. In fact, all controls in the table data view are morphed, although the difference is much less apparent for simple text fields. This is primarily done in order to make screen update sufficiently quick when there are a large number of columns in a table, or a large number of rows on display, however, the switch to and from a combobox can usefuly save space on the display. Looking ahead once again, you can also morph some types of control in any forms which you design; in fact, table data view is actually a perfectly standard *Rekall* form, although it is constructed on the fly to match the table.

Table Quick Filters

Ultimately, you can customise what you or your users see using forms, which are described in the next chapter. But, there are often cases where you would like to see just some subset of the columns in a table (the table may have

many, many columns), some subset of the rows (the table might have thousands, or even millions of records); and you might like the rows to be ordered in some particular way. Of these, the second can be achieved by using the search function, and the third by clicking on the column headers (provided that you only want to order on a single column). However, table quick filters provides you with a way to set of such requirements in advance, and to conventiently switch between them.

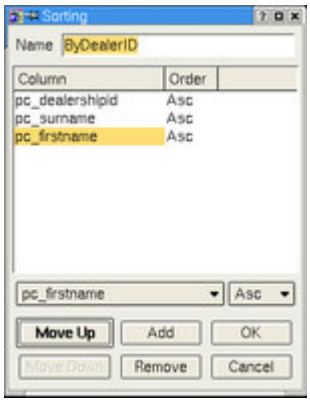
Rekall provides three types of quick filters, listed below. The three types of quick filter are independant of one-another, so you can select one of each type at the same time.

- Sorting filters: these are used to specify the order in which rows are displayed.
- Select filters: these specify which rows are displayed, by specifying conditions on values in the table.
- Column filters: these specify which columns are to be displayed, and the order in which the columns appear.

Quick filters can be set up when the table is showing in data view. The *Filter* menu contains four entries; the first opens up the quick filter configuration dialog; the other three are used to select which filters to apply. The configuration dialog is shown in the screenshot below. This is divided into three sections, corresponding the each of the filter types. All three are basically the same, showing a list of filters, plus buttons to edit an existing filter, to create a new one, or to delete an existing one.

Row sorting

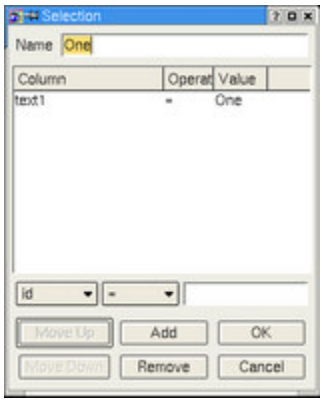
The sorting dialog is show below. The dialog shows the filter name at the top, underneath which the current condiguration is shown. This specifies one or more columns, and for each column, either ascending or descending ordering. Columns can be added by selecting a column and order in the comboboxes and clicking *Add*; or removed by selecting on an entry and then clicking the *Remove* button. In addition the entries can be reordered using the *Move Up* and *Move Down* buttons.



In this screenshot, the rows are sorted first by the *pc_dealershipid* columnmn, then, where *pc_dealershipid* are the same, by the *pc_surname* column, and finally by the *pc_firstname* column; all three columns are sorted in ascending order [13].

Row selection

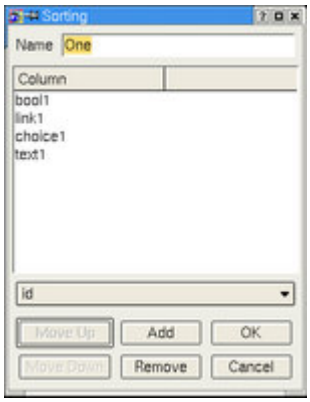
Next is the row selection dialog. This is much the same as the sorting dialog, except that there are three values to set, the column name, an operator and a value (except that no value is needed for the *is null* and *is not null* operators.. The entries are combined together such that *all* must be true for a row to be selected for display.



Note that the order in which the expressions occur is not important, unless you know that the order effects the efficiency with which the server database can retrieve matching rows.

Columns

Finally, the columns dialog is again muct the same, but with only a single value in each entry, the column. The chosen columns will be displayed in table data view in the order (left-to-right) as they are listed (top-to-bottom) in this dialog.



Using Quick Filters

Quick filters are applied by selecting from the *Filter* menu. There are three submenus, one for each type of filter. Each submenu shows each of filter of the corresponding type; in addition there is a *Default* item which removes the filter which has been applied. As noted above, the three types of filter can be applied independantly of one-another.

You should also bear in mind that each time you change the applied sorting or selection filter, *Rekall* will requery the server database for the data. Changing the column filter does not however requery the server database; in fact, the any columns which are not visible are really still there, and are only hidden.

Some Miscellanea

This chapter finishes with some miscellaneous functions that are available in table data view.

The rows can be ordered by the values in a column by clicking in the header; repeated clicking switches between ascending and descending ordering. Note that this ordering is done within *Rekall* itself and not by issuing a new query to the server database; this has the advantage of speed but may result in a different ordering (due, perhaps, to a difference between alphabetic and lexical ordering).

Column widths can be changed by dragging on the boundaries between column headers, and column order by dragging entire headers to the left and right. These changes are preserved the next time you open the table in data view. In addition, the *View* menu has a *Order Columns* item, which can be used to switch the column order between the order present in the table and ordered lexically on the column names. This is useful if you have a table with a large number of columns, are are having difficulty locating a particular column! Or, if this is a regular problem, define a column filter.

---

[8] These types are particularly useful with *XBase*, which has no notion of primary keys. The *Rekall XBase* driver handles primary keys by creating a 22 character column, and generates values that are almost certain to be unique. A foreign key also becomes a 22 character column. Note that for this to work, the primary key column *must* be the first column in the table.

[9] In fact, such checking could be done, since *Rekall* contains mechanisms to pass driver-specific type information around with data values. However, this is not currently used for type checking.

[10] Currently, *Rekall* has no knowledge of the server database's abilities to directly change a table using the SQL *ALTER* command. Data is therefore preserved by copying, so be aware that changing the design of a table which contains a large number of rows of data may not be a good idea.

[11] This is the underlying standard SQL notation for a partial string match. An option to use the Unix-like * wildcard will be added at a later date.

[12] Prior to *Rekall* version 2.0.0, this information was stored in a table in the server database called *__RekallDesign*. The change allows for much more information to be stored; as with everything else in *Rekall* it is formatted as XML. If you have used a version of *Rekall* prior to 2.0.0, the settings will be automatically transfered to the *.inf* files. The *__RekallDesign* table is not deleted, in case you need to go back ot it, but it will not be updated with any further design changes.

[13] This would generate add to the SQL query used to retrieve the data the term *order by pc_dealershipid asc, pc_surname asc, pc_firstname asc*.

Chapter 5. Designing and Using Forms

Table of Contents

Although data can be entered, viewed and updated directly using table data view, it is much easier for users to do this using a suitably designed form. In addition, there are lots of additional things that you can do with forms, such as view data from more than one table in the same form, or add functionality using *python* scripts.

This chapter describes the machanics of constructing a form, how you can structure them and what sorts of data controls are available for use in them. It does not explicitley cover *python* scripting, although it does describe some things, such as navigation buttons, which do make use of *Rekall*'s scripting capabilities. Scripting is returned to in detail in a later chapter.

Creating a Form

Forms are listed under the *Forms* tab. Here you will see a subtree for each server database plus, as usual, the *!Files* entry. As noted earlier, *Rekall* allows you to store forms either in a server database (inside the *__RekallObjects* table), or in the file system. Entries for each for appear in the appropriate subtree. The subtree also has a *Create new form* entry which can be double-clicked to create a new form.

One thing to remember is to be consistent. *Rekall* has the ability to open one form from another (typically when a button is clicked), and also to execute reports and copiers from a form, but currently the object which is to be opened must be in the same place as the form from which it is opened (ie., both must be in the same directory in the file system, or in the same server database.)

This section describes the creation of the *Clients* from from the *Orders* demonstration database. Bear in mind that, currently, *Rekall* lacks anything like the form creation wizards that are available in, for example, MicroSoft Access.

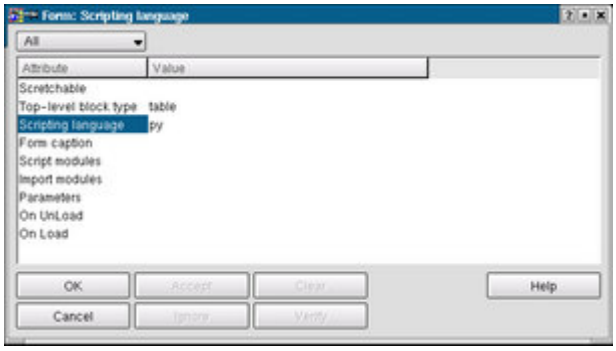
They are planned, but are not here yet.

Creating a New Form: The Form Dialog

Double-clicking the appropriate *Create new form* item will bring up a form properties dialog. This dialog is typical of the properties dialogs for all form (and report) controls. To edit a particular property, double click the property name (the left-hand column); this will replace the right-hand column with a dialog area appropriate to the property being edited. You can accept changes to a property with the *Accept* button, or ignore them with *Ignore*; however, until you click *OK*, no changes are make to the form (so *Cancel* cancels any individual property changes [14] . The small list box at the top can be used to limit the display to subsets of the properties; this is useful for things that have large numbers of properties.

Depending on which property is being editied, the *Clear* or *Verify* buttons may be enabled. The former is used to clear the property to its default state (that is, the state it would be in when the object was first created). The latter is used when appropriate to check that a property is valid, for instance to check that some *python* script compiles correctly, or that a validation expression is itself a valid regular expression. In addition, *Rekall* will make what checks it can to make sure that properties are only set to sensible values. Lastly, when running KDE versions of *Rekall* the *Help* button may be enabled; when it is, clicking it will bring up a context help popup (note that this popup is not a modal dialog, so you do not need to close it to return to the properties dialog).

The screenshot below shows the property dialog part way through setting various properties; the table which follows lists the properties, their significance and their values. This dialog essentially asks about properties of the form itself. For brevity, properties which are not relevant here and are left blank are not listed.



Property	Significance	Setting
Scripting language	Scripting language used in the form. This should always be *py* for *python*.	py
Caption	Caption for form title bar.	Clients
Top-level block type	Forms are constructed of nested blocks; this is the type of the top-level block. This is explained in more detail later.	table
Modal	Execute the form modally, in the same way as a modal dialog. See below.	table

The property dialogs for all objects will show a *Notes* setting. This is ignored by *Rekall* but can be used for arbitrary notes, for instance for documentation.

If the modal property is set, then the form will be executed modally, like a modal dialog; that is, once opened, the user can only do things in the form, and not in any window which was already open. This is useful, for instance, if you want to make sure that the user cannot make changes in other forms while this form is open. Note, however, that changing the modal setting does not take effect until the form is closed and reopened,

Making a form execute modally is particularly useful when using *tkcRekall*. On the Sharp *Zaurus* the Qtopia

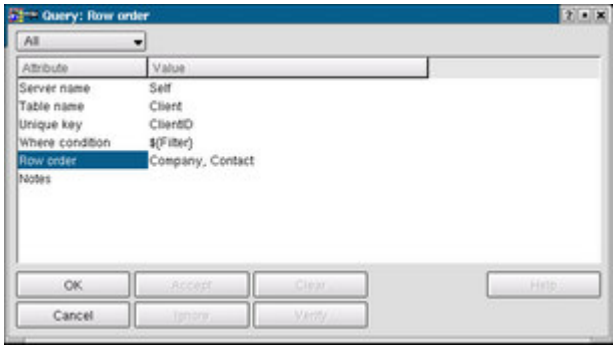
environment does not handle applications which open multiple windows very well. If you execute a form which is not modal, then switch to another application and back, you will find that the form is no longer visible, and is now "behind" *Rekall*'s main database window. It can be returned to the top using the *View/Show Window* menu, but this is inconvenient at best.

If you visit the *View/Options* menu item on the main database window, and select the *Modal* tab, there are settings to create new forms with the modal setting initially on. This option is disabled by default on the desktop versions of *Rekall*, but is enabled by default for *tkcRekall*.

There is one further caveat. Later, the manual explains how you can open one form from another (typically by clicking a button). Suppose you have two forms, *A* and *B*, and that form *B* can be opened by clicking a button on form *A*. Then, if *A* is modal, then *B* should be modal as well. If not, then form *B* will open, but will be immediately placed behind form *A*, and will not be accessible!

Creating a New Form: The Query Dialog

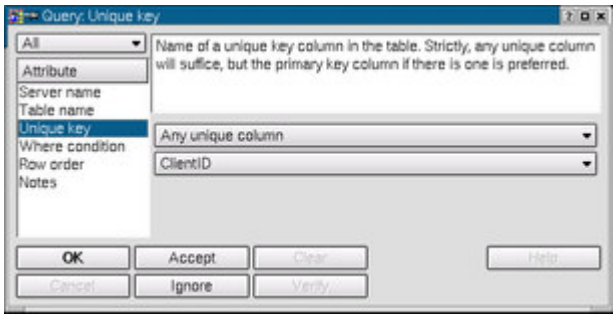
On clicking *OK*, a second similar dialog appears. This requests information about where the form should fetch its data from, in this case which table.



Property	Significance	Setting
Server name	Name of the server database which contains the table.	Self (see below)
Table name	Name of the table in the server database.	ClientID
Unique key	Name of a table column which provides a unique key. This is preferrably the primary key column.	ClientID
Row order	SQL expression giving the order in which the generated query will return rows. later.	Company, Contact
Where condition	Expression which is valid as the *where* part of an SQL select query. See below.	${Filter}

Setting the *Server Name* field to *Self* is interpreted to mean that the server database to be used is the same server database as the form is stored in (if the form is stored in the file system then for this to work the *!Files* entry in the server dialog must identify a server database. This setting is useful, since it means that you can copy the form to another server database, and provided that the second server database has compatible tables, etc., the form will work unchanged [15].

The next image shows the dialog when the *Unique Key* property is being edited. This is probably the most complicated setting, and it is complicated because *Rekall* is designed firstly to work with as wide a range of server databases as possible, and secondly to work as well as possible with server databases that already exists (ie., not just ones that have been set up using *Rekall*). If you are not sure about this section, then selecting the *Auto* option is the best course, and you can skip on to the next section.



The upper combobox (showing *Any unique column* in the image) is used to select the sort of column that is of interest, where the options are listed below. Below this may appear a second combo box.

Auto	If this option is chosen, *Rekall* will select what it sees as the best column to use, either a *primary key* column or a *unqiue* column.
Primary Key	In this case *Rekall* will only allow a *primary key* column to be used. A second combobox will appear containing just the *primary key* column, if any. When the form is run, *Rekall* will verify that the column is a *primary key*.
Unique key	In this case, *Rekall* allows any *unique* column to be used. A second combobox will show all *unique* columns in the table, and will include the *primary key* column if any. When the form is run, *Rekall* will verify that the column is indeed *unique*.
Any single column	Choosing this option will display a second combobox which shows *all* columns in the table. *Rekall* will then assume that the selected column is unique, whether or not it appears to be so in the database. It will also assume that an inserted or updated value is not changed by any server database triggers.

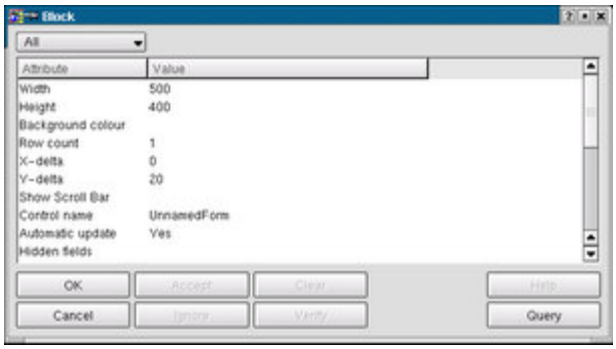
The *any single column* setting is useful if you *know* that a particular column will always be unique. However, be warned that *Rekall* will use the value from this column when updating or deleting rows in the table; if the column turns out not to be unique then unexpected results may occur [16].

Note: For any database designed prior to release *1.1.0-beta3*, the specified unique column will be used with the same behaviour as in earlier releases. It will be used as specified if it provides sufficient information that rows cna be inserted, updated and deleted. If not, then *Rekall* will use the *Auto* mode and choose the best column.

The value of the *Where condition* which has been set to *${Filter}* is clearly not valid as part of an SQL select query! This is shown here since it as an example of a *parameterized* property, such as are used in several places the demonstration *Orders* database. Briefly, when a form (or report) is executed, any properties what contain text of the form *${name}* have that text replaced with the value of the paramater *name*, if it is defined. In this example, it gives a way of executinf the form with a filter to select only clients that match some criteria. The whole area of paramaters will be returned to later in the manual.

Creating a New Form: The Block Dialog

The third and final dialog requests information about the the way data is displayed and handled; this is the *block* dialog. Briefy, forms are constructed of nested blocks, where eack block displays data which has been retrieved from somewhere in the server database. For instance, the classic form/subform arrangement is represented as an outer and an inner block. The form itself is a block, the block dialog appears.



Property	Significance	Setting
Width	Form (block) width in pixels	500
Height	Form (block) height in pixels	400
Row count	The number of rows of data to be displayed; zero will show multiple rows depending in the block size.	1
X-delta	X-offset between data controls when more than one row is displayed. Not relevant if only one row is displayed.	0
Y-delta	Y-offset between data controls when more than one row is displayed. Not relevant if only one row is displayed.	20
Y-delta	Y-offset between data controls when more than one row is displayed. Not relevant if only one row is displayed.	20
Control name	Name of the block, used for scripting. Not relevant here.	UnnamedForm
Show Scroll Bar	Setting this to *Yes* will display a scrollbar which can be used for record navigation.	Yes
Automatic update	If set to *Yes* (the default) then changes are automatically saved when moving to a new row.	Yes

Up to this stage, clicking the *Cancel* button in a properties dialog will abort the form creation. Clicking *OK* for the third time will move you on to the point of a blank form appearing.

Adding Controls to the Form

At this stage, a blank design form appears, onto which controls can be placed. *Rekall* does not use a tool box; all controls are added by sweeping out an area with the mouse (ie., point to one corner, press and hold the left button, move to the opposite corner, and release) and then using the popup menu which appears, when the right button is clicked. Lets start by adding controls for each of the table columns other than the primary key (which the user does not need to see).

Later in the manual there is a complete list of the available controls and the properties that each has, but the table below is a basic list.

Control	Description
Button	Clickable button. Can be used to trigger *python* scripts to perform actions.
Label	Fixed text label (but can be changed from a script).
Field	Simple text display and entry.
Choice	Pick one value from a list, displays as a combobox.
Link	Pick on value from a list, is linked to values in another table.

Check	On/off selection, displays as a checkbox.
Pixmap	Used to display images. Displays common formats like *.bmp*, *.jpg* and *.png*.
Row Mark	Used to display row number and current row markers, as in table data view.
Memo	Multi-line text edit control.
Rich Text	Displays text in QT *rich text* format (a very-much stripped down HTML).
Spin Box	Displays a numeric spin box control. Currently limited to simple numeric display.
Graphic	This displays an image, like a *Pixmap*, but the image is static. Typically used for eye-candy.
Tab Control	Provides a way of constructing a tabbed control, as in a tabbed dialog.

If the grid tool towards the right of the tool-bar is turned on then controls will snap to the grid, which generally helps to make layout easier. You should also note that by default forms use fixed layouts. However, *Rekall* supports a limited resizing facility (which you may have noticed in the table design and display screen [17]) which is described later.

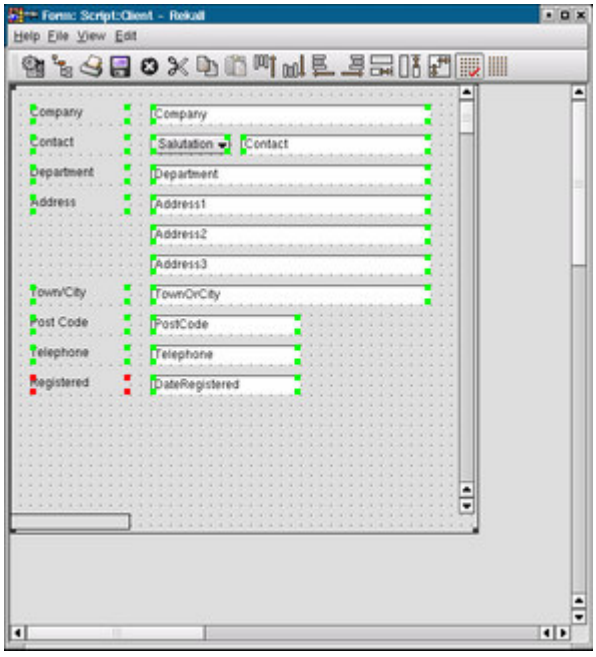
When the popup-menu appears, you can select the desired type of control. The actual contents of the menu will vary depending on exactly what sorts of controls can be added (there are some restrictions). The menu will also have a *New Block* submenu, which can be ignored for this form.

Here we will add *Fields* (which are simple line-edits controls) for each column other than *Salutation* which will be a *Choice* (a combobox). In addition, there are labels to go with the fields. The two tables below show the settings for the first *Field* control and the *Choice* control.

Property	Significance	Setting
X-position	X coordinate of the field	140
Y-position	Y coordinate of the field	20
Width	Field width in pixels	280
Height	Field height in pixels	20
Control name	A name which identifies the control in the enclosing block. Can be used it scripts.	Company
Display expression	This is the expression that is used in the SQL query which retrieves data for this field.	Company
Tab order	Tab order value when tabbing round fields.	1
Text alignment	Text alignment in the field.	left

Property	Significance	Setting
X-position	X coordinate of the field	140
Y-position	Y coordinate of the field	50
Width	Choice control width in pixels	100
Height	Choice control height in pixels	20
Control name	A name which identifies the control in the enclosing block. Can be used it scripts.	Salutation
Field name	The name of the column in the table whose contents are displayed.	Salutation
Tab order	Tab order value when tabbing round fields.	2
Values	The set of values that are offered.	Mr|Mrs|Miss|Dr|Sir|Lord

The following screenshot shows the form after all these controls have been added. The blobs at the corners of each control can be used to move or resize the control (or groups of controls, as explained a little further on). The scrollbars to thr right and bottom of the window can be used to pan the window, in case you want to design a form which is actually larger than the window. The vertical scrollbar in the design area appears because of setting the *Show Scroll Bar* property, and will be usable in data view (ie., when the form is actually running) to navigate through records. The rectangle to the bottom-left of the design area also appears on account of this setting, and will show a record number, line *Record 42 of 126*.



Double clicking a control brings up the properties dialog for the control [18]. Right-clicking in control (or in the block background) brings up a popup menu appropriate to the control or whatever, including options to *Cut*, *Copy* or *Delete* the control; the properties dialog can also be accessed from this menu. In the case of the background, this also includes options to insert new controls.

Positioning Controls

There are a couple of shortcuts that can be used to speed up the addition of controls. After adding a control, its blobs will be in red (*active*). If you right-click in the background and select a new control type while holding down the *shift* key, then the new control will be the same size and a little below the *active* control (without *shift* it would be to the right). The newly added control will then become *active* and the previous control will be marked in green (a *follower*). Now if you right-click and select a control without bothering with the shift key, it will be positioned with an offset which is the same as that between the two previous controls. This allows you to work down a column (or across a row) quite quickly.

Controls can be aligned and sized either by using the mouse to move or resize them, or by opening their property dialogs, and explictely editing the position and size properties. Also available on the toolbar are a pair of controls which assist positioning. The first *Enable Snap to Grid* if an on/off toggle; if on then controls will be automatically aligned and sizeed to the grid immediately after creation or after you manually move them. The second, *Snap Controls to Grid* will align and size all selected controls (ie., controls marked with the red *active* or green *follower* corner blobs) to the the grid. This is useful if automatic snapping is off, but you'd like to align some subset of the controls.

	Enable/disable automatic snapping.
	Snap selected controls to grid.

However, *Rekall* also allows you to align control to one-another, and to make two or more controls the same size. If two or more controls are selected, then the alignment and same-size controls on the toolbar become active. In each case, the *active* control (the one with the red blobs) determins the alignment or size. The tools are show below:

	Align controls to top
	Align control to bottom
	Align controls to left
	Align controls to right
	Make controls have the same width
	Make the controls have the same height
	Make the controls have the same size

Saving and Showing the Form

You can save the form by clicking on the Save tool or using the File/Save menu item. The first time you save you will be prompted for a name under which to save the form. If you are creating the form in the file system it will be stored as *name.rkl.frm* ; if created in the server database it will be stored in the objects table. By the way, forms (and reports) are stored in XML format, so if you are inquisitive and save the form to the file system, then you can have a look at it with an editor (and evey modify it; got the caption wrong? well, just fix it while you are there!)

Once saved, you can switch to data view (either via the View menu or using the left-hand most tool), in which case you should see any data that you entered into the table earlier. The various tools for record navigation appear on the tool bar.

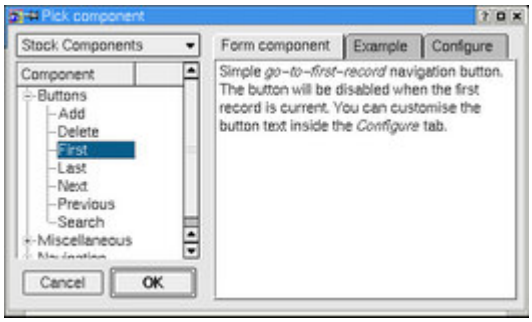
Adding Navigation Buttons

To finish the form, we'll add some buttons to allow record navigation without having to use the tool bar. Buttons are a bit like labels, in that they don't need to refer to a column in (or expression from) the table about the same, but the settings are a little more important.

There are three ways to add buttons such as those for navigation. The most general way is to add a button and to associate with it the some script code to be executed when the button is clicked, and which does the required operation, such as moving to the previous or the next record; this is described in more detail in the chapters on scripting. The second way is to use a shortcut mechanism, described in the next section but one; this is the mechanism used in *Rekall* prior to version 2.0.0 and is described for completeness. However, the third and easiest way is to use a *reusable component*.

Component Buttons

Rekall comes with a number of *reusable components*, navigation buttons included, which can be inserted into forms and reports. To insert a component, right-click at the point at which the component should be inserted, and select *Paste component* from the *Edit* menu. This will bring up the component dialog, illutrated below.



Components may be designed locally and stored in server databases or locally in you directory (these are described in a later chapter), but with the combobox at the top left showing *Stock Components*, those that come with *Rekall* are shown. Below this is a list of available components, grouped up loosely by function. Clicking on a component will show details of the component under the first tab to the right; in the illustration, the *First* button has been selected, and some descriptive text appears.

Clicking the second tab will show the component, in this case a button with the legend *First*. Clicking the third tab shows configuration options for the component. In the case of the *first* button (and, in fact, for all the buttons), the button's legend may be changed. If you change this and then go back to the second tab, the component will be displayed with the changed value, in this case the legend on the button. When you are happy, click *OK* and the button will appear in the form.

Once pasted into the form, the component is no different to any control that you constructed directly [19] , and its properties can be changed in just the same way. So, for instance, in this example you can move the button around, and change its legend. You can also see its internal workings, such as the *python* script that actually makes the button work (this may be useful if you get on to writing your own scripts, as an example of how things can be done).

In the demonstration database, buttons have been added to the form the the *First*, *Previous*, *Next* and *Last* record navigation functions. In addition, a *Find* button handles the querying; clicking it once will clear the form, so that search criteria can be entered, clicking a second time will perform the search.

Button Shortcuts

As noted, this section is included for completeness. You can probably skip it.

The table below shows the settings for a *Next* button using the shortcut mechanism.

Property	Significance	Setting
X-position	X coordinate of the field	200
Y-position	Y coordinate of the field	350
Width	Choice control width in pixels	50
Height	Choice control height in pixels	40
Control name	A name which identifies the control in the enclosing block. Can be used it scripts.	Next
Field name	The name of the column in the table whose contents are displayed.	>
On Click	The action to perform when the user clicks the button.	#Click

The two important settings are *On Click* and *Control name*. Without going into details, the *On Click* setting invokes a standard *Rekall python* function. The function gets the *Control name* setting and uses that to decide what to do, as listed in the next table. By the way, this mechanism is provided as a quick and convenient shortcut to add navigation buttons without having to write any *python* scripts (and there are some other similar shortcuts described

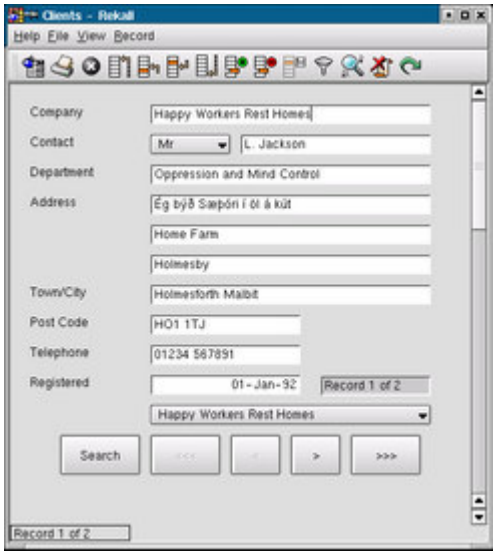
a bit further on). The sections of this manual on *python* scripting describe how you could do the same thing without using the shortcuts, which then opens the way to providing much more complicated, application specific functionality.

Control name	Action
First	Go to first record
Previous	Go to previous record
Next	Go to next record
Last	Go to last record
Add	Add (insert) a new record
Save	Save changes to current record
Delete	Delete current record
Query	Start a query (ie., search)
Execute	Execute a query
Cancel	Cancel a query

The Finished Form

Below is an image of the finished form, with everything there, as it is in the demonstration *Orders* database (the image has a bit of Icelandic in it, we trust it doesn't way anything offensive!). There is actually one extra control there (the combobox below the main fields and above the buttons), which is the *Navigation/Selector* component, with some properties altered. This can be used as a quick way to navigate to a specific record.

You'll notice that the navigation buttons are enabled and disabled as you move through the records; for instance the *First* and *Previous* buttons are disabled when you are the first record. This bit of magic is handled by the *Event/Slot* mechism described after the chapter on scripting.

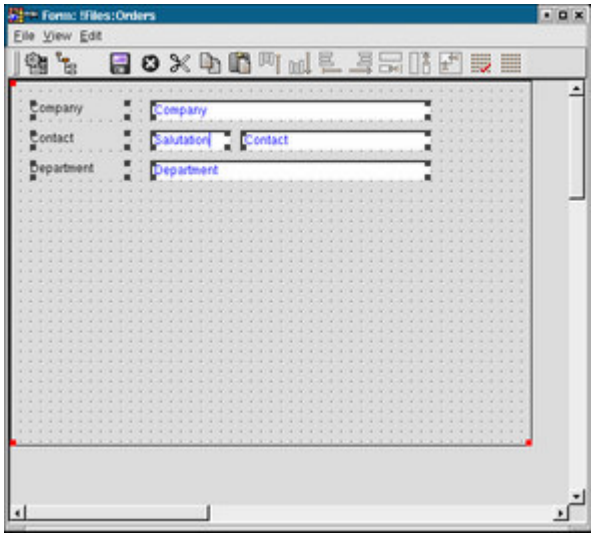


Creating a Form with a SubForm

In the demonstration *Orders* database, the form used to display products is be created in just the same way. The only extra features are the *PixMap* control, which can be used to display the *Image* column from the *Products* table, and the multiline *Memo* control.

The next form is the orders form. This is a little more complicated, since it contains a subform within the main form. The main form shows a client; the sub-form shows all orders placed by the client (analagously, there could be a form whose mainforn shows products, and whose sub-form shows all orders for that product).

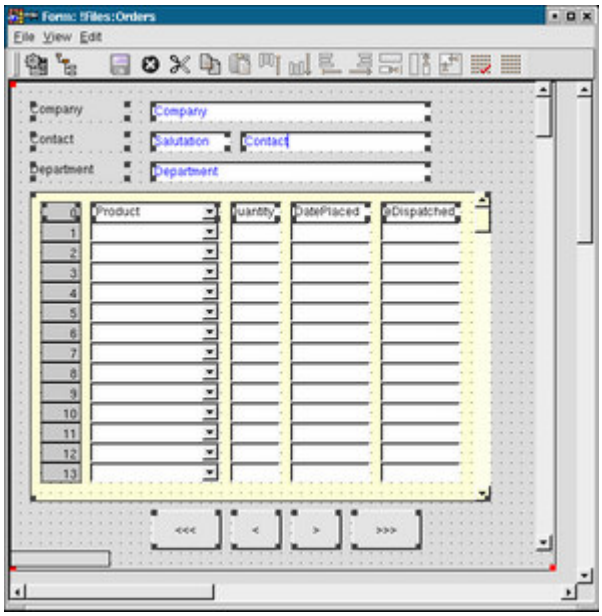
The image below shows this form (which accesses the *Client* table), called *Orders*, just after the fields for the *Client* table have been added. For a bit of variety, the fields are all marked as read-only and have the text colour set to blue (so give the user some indication that they are read-only, of course you can use whatever colour you like, or maybe change the font).



The subform can now be added. *Rekall* uses the general term *block* to refer to things like subforms (and similar objects in reports). In fact, the entire form is itself a block. Sweep out a suitably large area and select *New Block/Table Block*. This brings up a dialog for the table which is to supply the information, in this case the *Orders* table. It is followed by a dialog for the block properties; the position and size will be set according to the area that you swept out; the other settings are shown below:

Property	Significance	Setting
Row count	The number of rows of data to be displayed; zero will show multiple rows depending in the block size. later.	0
X-delta	X-offset between data controls when more than one row is displayed. Not relevant if only one row is displayed.	0
Y-delta	Y-offset between data controls when more than one row is displayed. Not relevant if only one row is displayed.	20
Parent field	This is the name of an item of data retrieved in the query for the outer block, which is used to link data displayed in the inner block.	ClientID
Child field	Similarly, a column in the query used for the inner block, which links to the outer block.	ClientID

The next image shows the form after controls have been added to the inner block. The left-hand most control is a *Row Mark*, as seen in the table design form; the right-hand three controls and simple *Fields*. The second control is a *Link*, which displays a value from some other table according to a value in the form. When creating a link, two property dialogs will appear; their settings are shown below the image.



The tables below show the important properties for the inner block and its associated querty.

Property	Significance	Setting
Server name	server database name	Orders
Table name	Name of the table from which a value will be displayed.	Products
Unique key	A unique key column in the table.	ProductID
Row order	Ordering express for the SQL query used to retrieve displayed in the inner block.	Description

Property	Significance	Setting
Control name	Name used when accessing the control from a script	Product
Parent Field	Name of the column in the block which identifies a row in the *Link* control's table	ProductID
Child field	Name of the column in the *Link* controls table used to link to the block	ProductID
Display expression	SQL expression displayed in the *Link* control.	Description

Containers and Stretchable Forms

Rekall contains some basic support for automatically resizing and repositioning of controls when the size of a form is changed by the user. Although this is not as sophisticated as (say) that provided by the QT toolkit in which *Rekall* is built, is is hopefully sufficient to provide a useful level of functionality.

Firstly, all form controls are embedded in a *container*, and containers may be nested inside one another. The form itself is a container, and so it a sub-form (and sub-sub-form ...). The position and size of a control depends on the properties of that control, and possibly on the position and size of the container in which it is embedded; that same applies to nested containers. Secondly, forms have a *Stretchable*, which can be set to *Yes* to enable resizing (if this property is not set then the form window can still be resized, but the controls remain fixed within it, and scroll bars appear as neccessary which can be used to pan the window).

Associated with each control and each container (except for the outermost form-level container), are a pair of properties, *X mode* and *Y mode*. These determin the behaviour of the control (or container) as its parent container is resized (in the X and Y directions respectively). The default value is *Fixed*, in which case resizing a container has no effect on embedded controls and containers.

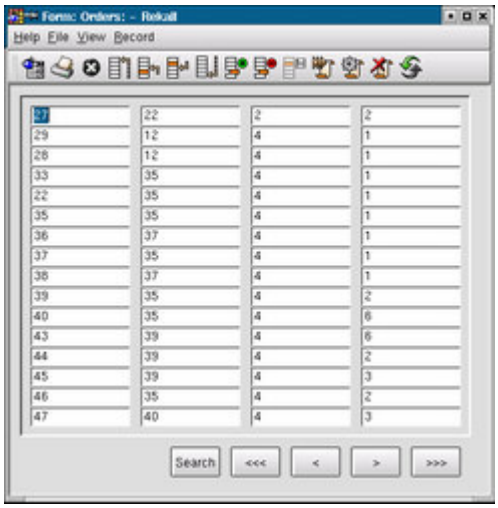
However, these properties can also be set to either *Float* or *Stretch*. If the *X mode* of a control is set to *Stretch* and the container in which it is embedded changes width, then the width of the control changes to match; if it is set to *Float* then the control stays the same width, but the whole control moves right or left with the right-hand edge of the form. *Y mode* similarly controls the behaviour as the height of the form changes. The table below gives the exact meanings and behaviour for *X mode* (*Y mode* is analagous).

Setting	Property	Value	Behaviour
Fixed	X	Distance of left edge of control from left edge of container	Position stays fixed
	W	Width of control	Width stays fixed
Float	X	Distance of left edge of control from *right* edge of container	Control position tracks right edge of container
	W	Width of control	Width stays fixed
Stretch	X	Distance of left edge of control from left left edge of container	Position stays fixed
	W	Distance of *right* edge of control from *right* edge of container.	Width position tracks right edge of container

In the demonstration database, the subform is set to stretch in both directions, while the buttons are set to float in the Y direction. You will also notice that, since the number of rows displayed in the subform is set to be adjusted automatically according to the height of the subform (ie., *Row Count* is set to zero), then the number of rows displayed changes as the form height changes.

As noted above, forms themselves are containers, as are nested subforms (blocks). In addition, there are a few other container objects. The simplest is simply called a container, which can be added in the same way as a subform - sweep out an area, and select *New Block/Container* from the popup menu. In this usage, any control that is embedded in the container is logically part of the block in which the container is embedded, so any data that is retrieved for use in data controls (fields, memos, etc.) comes from the same place as the block retrieves data from (typically the same server database table).

Supppose that you'd like a form which shows multiple rows from a table, which adjusts the number of rows according to the size of the form, and which has a row of navigation buttons across the bottom. Without using a container this is not possible, since the number of rows will adjust to occupy space down to near the bottom of the form. However, using a container, this effect can be achieved, and is illustratd in the screenshot below. The data controls are placed inside a container which has *X mode* and *Y mode* both set to *Stretch*, while the *Y mode*s of the buttons are all set to *Float*. For effect, the container has been given an edge by setting its *Frame Style* property.

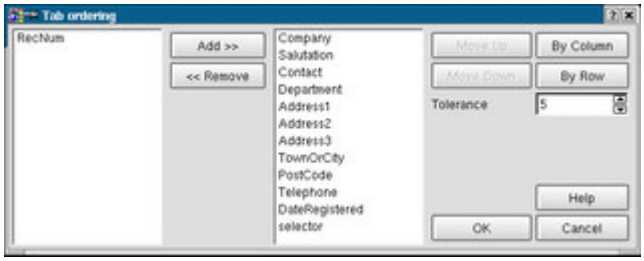


Tab controls are also containers. These comprise the tab control itself, and one or more tab pages. The geometry of the tab pages is determined by the geometry of the tab control; they are the same width, and the same height less a bit for the tabs.

Form Navigation

You can move from control to control and record to record in various ways; using the toolbar, using the keyboard, using any buttons that you have added to the form, and using the mouse.

Apart from the obvious point-and-click to move to another control, you can also use the tab key; tab on its own moves to the next control while shift-tab moves to the previous control [20] . Enter also functions the same as the tab key. The tab order is initially the same as the order in which controls are created, except that buttons and labels are not included in the tab ordering [21] . The tab order can be changed by right clicking in a block (when in design mode) and selecting the *Set tab order* item. This displays a dialog such as is show below (this actually being for the *Clients* form):



The left-hand list box shows controls which are not in the tab ordering (ie., which cannot be reached by tabbing); the right-hand list box shows those that are, with the tab ordering being the order down the list. Controls can be moved between lists; if a control is selected in the right-hand list then it can be moved up or down. In addition, controls can be automatically ordered by column (*Rekall* tries to order so that tabbing goes down successive columns) or by row (*Rekall* tries to order so that tabbing goes across rows). Automatic ordering depends on the exact layout of controls, and the *tolerance* setting defines how well aligned two controls have to be to be considered to be in the same column (or row).

Note that the tab ordering applies to controls corresponding to the same row of data in the server database. So, if you have a form which displays multiple rows of data, tabbing will work though all controls for one server database row before moving to another.

If you tab forward from the last control in the tab order, then *Rekall* moves on to the first control in the next row of data from the server database, while back-tabbing (shift-tab) from the first control will move to the last control for the previous row of data from the server database.

Movement between server database rows is accomplished using the up and down cursor keys. If the form is displaying multiple rows of data then it will scroll as appropriate. Also, assuming that the block *auto-sync* property is set [22] updated rows are automatically saved whenever focus moves to a control in a different row. In addition, the Ctrl-Return (or Ctrl-Enter) key combination saves the current row.

The toolbar has controls which can be used to navigate between rows. These are actually the same as appear for a table in data view, but are shown again below.

	Go to first record
	Go to previous record
	Go to previous record

	Go to last record
	Insert record
	Delete record
	Save record
	Start query
	Execute query
	Cancel query
	Reload form data

With the exception of the *save record* tool, all the tools operate on the block which contains the control which currently has focus, or the outermost block if no control has focus.

The *insert record* tool opens up an empty row in front of the current row. This is simply a row into which data can be entered; its position does not imply anything about where the record will be saved in the server database (and, if you have a *order by* expression in the query which retrieves data, then if you requery the server database the record will be subject to that ordering).

The *save record* tool operates on the entire form. In essence, it does a save at the outermost block level; the block will in turn do a save on any nested blocks, and so forth.

The *start query* tool is used to start a query. Clicking this tool will clear all the control in the current row. You can enter data into some of the controls, and then click the *execute query* tool. *Rekall* will then search for all server database records which match the entered data. Controls can be left blank, in which case they play no part in the query; otherwise, the data in the server database record must match exactly, with the exception of text entry controls. In this case, the % character can be used as a wildcard [23] (so *M%* would match *Mike* and *Michelle* but not *Adam Miles*). The query can be cancelled with the *cancel query* tool. Query terms like *>100* are not yet implemented.

In addition, you can enter query terms like *< 10*, using the operators < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) and *!=* (not equal).

Note that when entering data for a query, control verification and not-null checks are not applied. This does mean that you can enter somthing like *>silly* in a numerical value.

The full set of keyboard navigation keys is:

Tab, Enter	Go to next field, or next row if at last field
Shift-Tab	Go to previous field, or previous row if at first field
Cursor-Up	Go to previous record
Cursor-down	Go to next record
Ctrl-Cursor-Up	Go to first record
Ctrl-Cursor-Down	Go to last record
Ctrl-Enter/Ctrl-Return	Save record
Escape	Cancel changes to row

Menu-Only Forms

The demonstration *Orders* database has a main form which does not show any data, and only has buttons which open up the other forms and reports in the database. This is an example of a *Menu-Only* form. To get a menu-only form, set the *Top level block type* property in the form to *Menu block*. The resulting form can have buttons and labels, but does not allow any data controls.

Below is the main form from the demonstration *Orders* database. In this example, the area which contain the buttons are nested menu-only blocks, in order to be able to add frames and coloured areas to the form. This effect could equally well be achieved with containers.

---

[14] Also, the form as a whole is not actually saved in the server database or file system until you explicitely save it.

[15] *Self* will appear in a number of places when using *Rekall*. Typically, there will be a list which includes each server database that you have defined, plus a *!Files* entry when *!Files* has a server database associated with it. Unless you really want a form which is stored in one server database to access a table in another server database, then *Self* is the option of choice.

[16] For instance, suppose you tell *Rekall* that column *ClientCode* in table *ClientTable* is unique, and then in a form you delete a row that has *UNKNOWN* as the value of the *ClientCode* column. To do this, *Rekall* will execute the SQL query *delete from ClientTable where ClientCode = 'UNKNOWN'*. This will delete as many rows as match, which might not just be the row you intended!

[17] The table design and data display screens are really forms. The one for table design is embedded in the code, and the one for data display is generated at run time to match the table.

[18] Double-clicking for the properties dialog currently does not work on the QT3/KDE3 builds, due to internal QT changes. This should be fixed in the next release.

[19] A feature planned for a future version of *Rekall* is *linked* components. In this case changing the definition of a component will effect all forms that use the component. But, for the moment, using a component is rather like copying and pasteing text in a word processor; changing the copied text doesn't affect the pasted text, nor vice-versa.

[20] Tab and shift-tab do not work in *memo* controls, since you might want to actually enter a tab as data.

[21] You can however create a button with text like *&Click Me!*, in which case the *C* will be underlined, and the key combination *Alt-C* will be equivalent to clicking the button.

[22] *Auto-sync* is the default. Currently, there are a number of problems associated with clearing the *auto-sync* property; for instance, data may be lost if an updated row scrolls out of view. It is recommended that you leave this property set.

[23] This will probably be changed to * (or an option provided to select which) in a future release of *Rekall*.

Chapter 6. Queries

Table of Contents

When a form or report is designed, it can retrieve data from one of three places; directly from a table (as demonstrated by the forms in the previous chapter), from a free-text SQL query (which will be described later), or from a *Rekall* query. It is probably true that while forms are most likely to access tables directly, reports are more likely to use free-text SQL queries or *Rekall* queries. For this reason, this chapter concentrates on latter, before we move on to reports.

To reiterate what has been said earlier, there are two ways that the term *query* can be used, either in the context of an SQL query (*select ... from ...*) or in the context of a *Rekall* query. This chapter talks about the latter (although, untimately, a *Rekall* query is used to generate an SQL query which is executed by the server database).

A *Rekall* query essentially sepecifies a set of tables (possibly only a single table), a set of relationships between them, and a set of SQL expressions; the latter may be used to specify data to be retrieved from the tables, or for functions such as ordering or filtering. These component parts are combined to generate the SQL query. In addition, however, when *Rekall* queries are used in the design of forms and reports, *Rekall* can use the relationships between the tables to arrange the data in form-subform (or, analagously, report-subreport) structures.

Rekall does not have anything equivelent to the update, insert or delete queries provided by Microsoft Access. These will probably be added at a later date, but you will be able to achieve the same funcionality directly via *python* scripts and the interface between *python* and the server databases.

This section starts by describing the construction of a query which retrieves data from the demonstration *Orders* database.

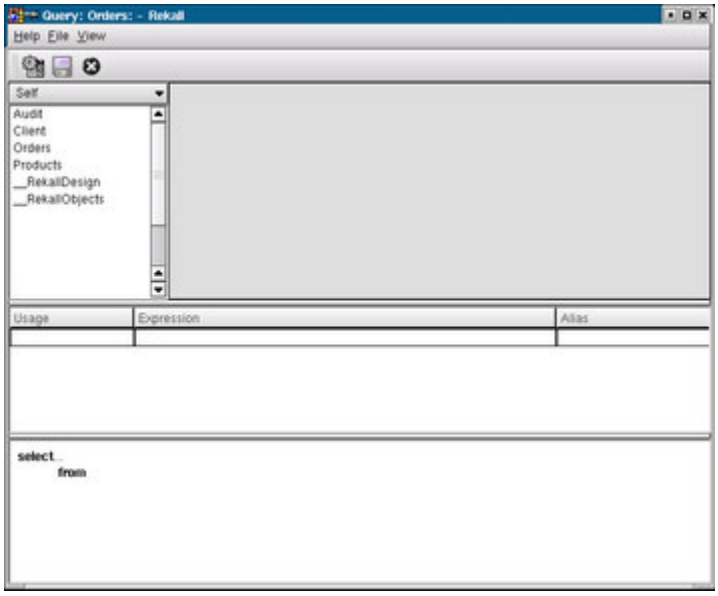
Creating Queries

The query which is designed in this section retrieves data simultaneously from the *Orders*, *Clients* and *Products* tables. Since the contents of these tables is logically linked using the products and client keys, the basic *SQL query* will look something like (where ... is replaced by whatever fields are needed).

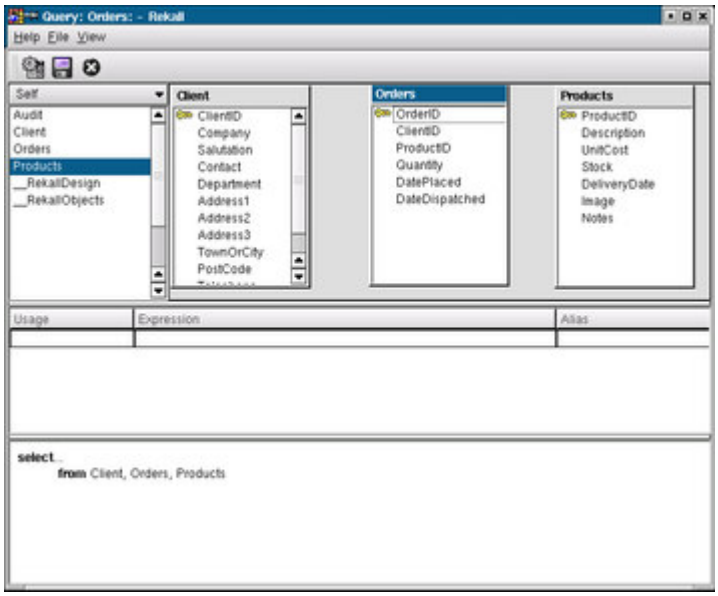
```
select ....
  from  Client, Orders, Products
  where Orders.ClientID = Client.ClientID
```

```
and   Orders.ProductID = Products.ProductID
```

To create a new query, go to the *Query* tab, open the appropriate branch, and double click the *Create new query* item. This brings up a new window; select a server database from the top-left combobox [24] , and the window will appear as shown below:



The left-hand listbox shows all tables for the selected server database. The top-right area will in due course show the tables used in the query and their relationship. The middle area is used to add expressions and critria such as filtering (SQL *where* terms) and ordering, while the lower area will show a skeleton of the *SQL query* that will be generated. Since this query requires all three tables, double-click in turn on each *Client*, *Orders* and *Products* (in that order). You can see that as changes are made, the SQL query text changes to match. With a bit of repositioning, the window should now look like:



If you need to set a table alias (so that the SQL will look something like *select .... from Client C, ...* then right-click in the table field list, and select *Set Alias* from the popup menu. This popup also contains *Delete* entry which can be use to remove a table.

The next stage is to add links between the table which specify the *SQL query* join conditions. First. two things to

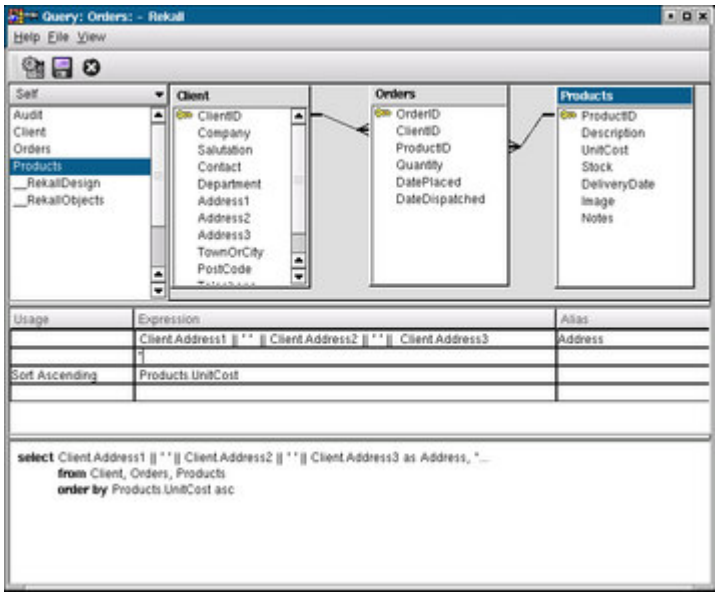
note. Currently, *Rekall* does not store any table relationship as other database front ends do, not does it access any key/foreign key information that may be stored in the server database, so you need to add the links each time; a relationship editor will be added to a future release of *Rekall*. Secondly, *Rekall* currently only allows links where one end of the link is a primary key (you can see which these are, as *Rekall* marks them with a key icon).

Links are added simply by dragging and dropping, so drag and drop from *Client.ClientID* to *Orders.ClientID*, and from *Orders.ProductID* to *Products.ProductID*.

We can also add some filtering or ordering criteria. The *Expression* column should contain a valid SQL expression. The *Usage* column in the middle area can be set to one of the values below (if you click in it then it will change to a combobox):

- Sort ascending: the expression is used to sort rows in ascending order.
- Sort descending: the expression is used to sort rows in ascending order.
- Where: the expression must be valid for use in an SQL *where* term.
- Group by: the expression must be valid for use in an SQL *group by* term.
- Having: the expression must be valid for use in an SQL *having* term.
- Blank: this is simply a way of defining an expression which will can be used when the query is accessed from a form or a table.

You can quickly enter columns into expressions (in the form *table.column* bu dragging from a column in a table to an expression area; expressions can be deleted by right-clicking and selecting *Delete*. Again, the text in the lower panel will change to reflect changes made to the ordering and filtering criteria. The next screenshot shows the query with a few expessions added.

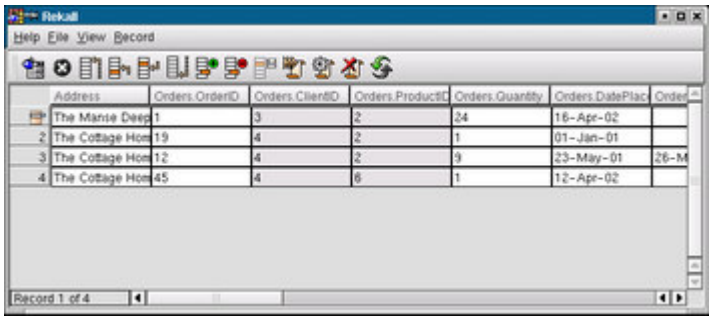


By default, when you display a query in data view, it will show a column of data for each column in each of the tables in the query. If, however, you have one or more data expressions (the last case in the list above) then in data view you will see only those expressions. If you want to see some specific expressions *and* also see all the columns, then add the expression *; this is analagous to the SQL *select * from ...* notation.

Unlike some other databases, it is not necessary to specify which columns (or, more generally, expressions) the

query returns. When you use the query when designing a form or report, you will be able to select any column from any of the tables in the query (or any expression that uses them). *Rekall* will always construct the apprpriate *SQL query*. However, as above, you can add arbitrary expressions which will can be selected when designing a form or report; this may sometimes be convenient.

A query can be viewed in data mode just as a table. Although a query can be used in a form to retrieve data in a structured way (much like the linked tables in the *Orders* form - more on this later), the query viewer will "flatten" out the data, so that one row will be displayed for each row retrieved from the database. By the way, although you can switch a form between design view and data view without saving the form, you cannot do this with a report; if you try to switch to data view and the form as been modified, you will get a warning.

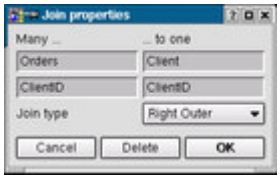


There are two caveats. Firstly, *Rekall* does not allow new rows to be inserted where the query contains more than a single table. This may be relaxed in a future release, but the semantics of doing this are not obvious (because of the join conditions between the tables). Secondly, it is not sensible to update values in columns which are used to relate the tables, since this might break the linkage displayed by the query viewer; these columns are displayed with gray backgrounds. [25]

One final note. You should not make any assumptions about the order that the server database will return rows if you do not specify any ordering. In the example, you should not assume that all rows corresponding to a client will be returned contiguously.

Joins: Inner, Outer and none

By default, *Rekall* creates queries over multiple tables using inner joins. However, you can change the join type between two tables to either *right outer* or *lft outer* by right-clicking on the link. This brings up a dialog box as shown below:



This dialog can also be used to delete a join. Note that if you attempt to save a query which has tables that are not connected, then *Rekall* will warn you and ask if the query should be saved anyway. You might want to do this if you really do want the query to return all row combinations from two (or more) queries, or it may be that the join condition is much more complicated and is defined as an explicit *where* expression (but if this is the case, *Rekall* cannot deduce any structure and queries will always be executed "flat" [26]).

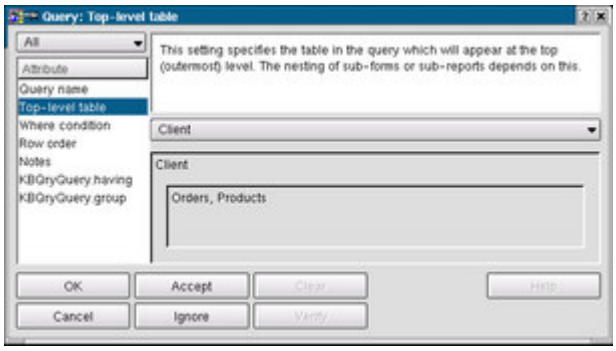
Using a Query in Forms and Reports

When the query is actually used (in a report described in the next section), then it is used in a context which provides some additional structure at the *Rekall* level. To do this, *Rekall* allows the selection of a "top" table and

then, on the basis of the relationships between the tables, will work out a sensible grouping. For instance, the *Clients* table is chosen as the "top" table, then the report would be constructed as a report-subreport with clients in the report, and orders and products in the subreport (ie., the report will group orders up by client). Conversely, if the *Products* table is chosen, then the report will group orders up by product. Finally, if you chose the *Orders* table then a simple flat report would result.

Bear in mind that this grouping is indepenent of any *group by* or *having* expressions. Essentially, *Rekall* uses the relationship information to decide on a grouping, and then groups up the data as it is retrieved from the server database. For instance, if the *Clients* table is chosen as the "top" table there the data will be grouped up on the basis of the primary key value from the *Clients* table.

The query properties dialog that appears when you use a *Rekall* query in a form or report has a property *Top-level table* which is used to select the top table. The dialog will show a illustration of the effect of choosing a particular table; in the screenshot below, the *Client* table has been so chosen:



Lastly, when you use a *Rekall* query in a form or report, there are options to set additional *where* and *order by* expressions. These modify the query for just that form or report.

Free-Text Queries

Rekall also includes support for free-text queries. These are accessed by setting the top-level block type to *SQL block*, or by inserting an *SQL block* into a form or report. The free-text query is specified by a server database and the text of the SQL query itself.

The query must start with *select ... from*. *Rekall* parses the query in order to extract its component parts. The parser is fairly loose, for instance it will accept anything other than a keyword between the *select* and *from*, so a query that *Rekall* thinks is valid may be considered invalid by the server database [27]. The property dialog for a free-text query has a *Verify* button that can be used to check the query.

Since *Rekall* parses the text of the query, when it is used in a form, *Rekall* will generally be able to save changes, that is, perform appropriate server database updates. However, it will only be possible to insert or delete rows if the query accesses a single table. Also, since *Rekall* cannot see any relationships between the tables in the query [28] there is no structure information which allows the selection for a "top" table, and data will be accessed simply as a set of rows.

---

[24] As for forms, if you select *Self* as the server then the query will access tables which are in the same server database as the query is stored in.

[25] Actually, *Rekall* will not allow you to update columns which show primary key values, however you can try to

update the related key values (ie., in the above example, you cannot update the *Client.ClientID* column, but you can try to update *Orders.ClientID*.

[26] A future feature is to allow an arbitrary expression to be associated with a link; this will give the best of both worlds.

[27] Conversely, it is just about certain that there are queries that *Rekall* does not parse but which are valid. Please let us know if you experience problems in this respect, and we will extend the parser appropriately.

[28] This might be possible in a later version of *Rekall*, once there is either a relationship editor, or access is available to any relationship information stored in the server database.

Chapter 7. Designing and using Reports

Table of Contents

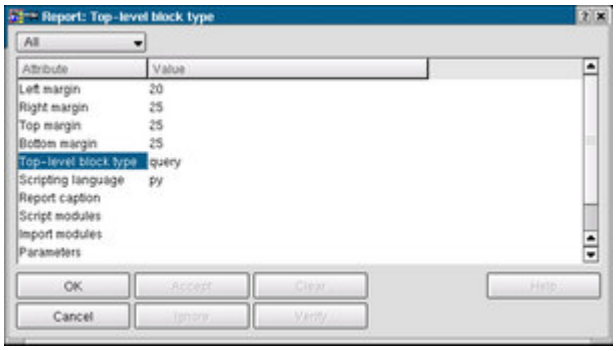
Rekall allows you to design and run reports in much the same way as you design and run forms. Reports are like forms, and can take data directly from tables, or from *Rekall* queries (or free-text queries), and like forms, display data in controls what are embedded in the report. Again, analagously to forms, reports can contain nested blocks.

Reports can be designed to produce output to various sizes, so you can output to different paper sizes. Also, if you are using KDE version 2.2 or later, then you have all the functionality provided by the KDE print system, so output can be sent to files in PDF or PostScript format, as well as being physically printed.

Creating a Report

The report that is designed in this section shows a summary of outstanding orders. This is rather like the *Orders* form, and contains an outer block (which shows the *Client* table) and an inner block (into which details of each order appear). This is the *OrdersSummary* report in the *Orders* demonstration database.

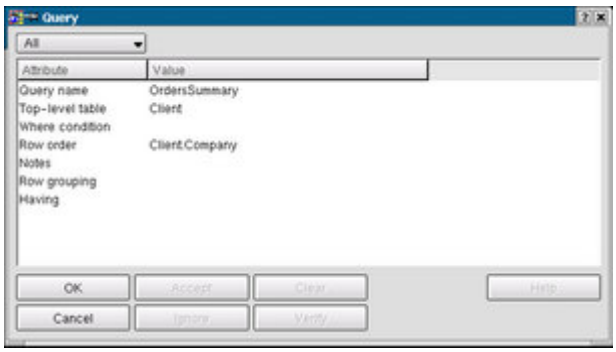
A report is created using the *New Report* entries under the However, unlike the *Orders* form, this report gets its data via the query from the previous section; select *Query Block* as the *Top-level block type* in the first report dialog, which is shown below. This dialog also allows you to set print margins. The values are initially set to defaults, which can be changed via the *View/Options* menu on the main database dialog.



Having clicked *OK* on the report dialog, a dialog for the query appears. The settings for this are listed below, and the dialog is shown in the following screenshot. Note that there are a set of properties *Where condition*, *Row order*, *Row grouping* and *Having*; these are appended to the *Rekall* query and can be used to customise the query for the

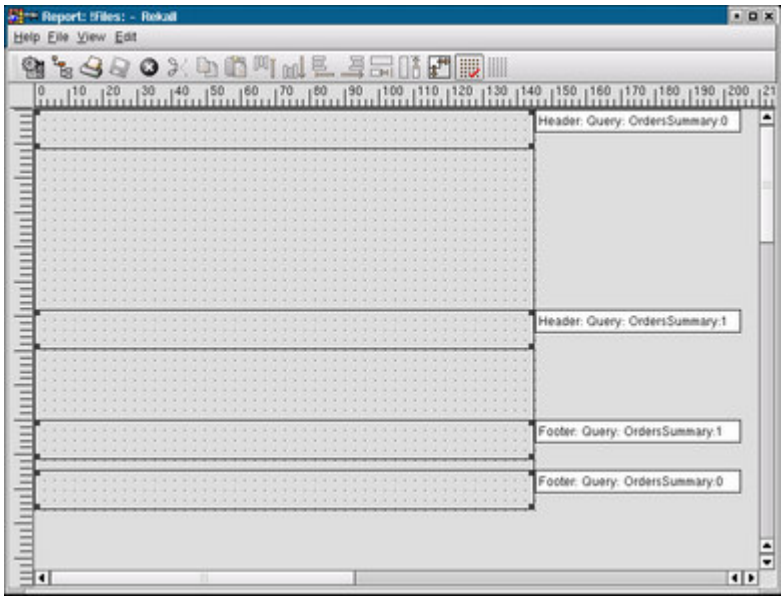
purposes of the report (although, beware that it would be possible to create invalid queries).

Property	Significance	Setting
Query name	*Rekall* query to be used to supply data	OrdersSummary
Top-level table	The table in the query which is used to supply data for the outermost report block.	Clients
Row Order	Additional SQL query ordering expression.	Client.Company



Note also the *Row order* setting. Even if you were not really bothered about the order, you would still need something which orders the clients, such as comany name or *ClientID*. This is because there is no gaurantee that the server database would otherwise return data with rows at least grouped together by company (so the report might show a page for some orders for company A, then some for B, and then another for A). In this example an order has been added here, but the order could equally well be set in the *OrdersSummary* query itself. The advantage to setting it here is that *OrdersSummary* could then be used in another report which shows client orders for each product (as opposed to product orders for each client), with the *Row order* property set to *Product.Description*.

Clicking *OK* in this dialog will lead to the third dialog, the *block* dialog. This is similar to a form, although some properties are not present, for instance there is no row count and no control spacings. The former is not needed since a report will always generate as much output as is needed for the data, and the latter since spacings are controlled by the layout in the design. Clicking *OK* once more leads to a blank report, shown in the following screenshot.



Some explanation is in order here. Because of the choice of a *Rekall* query as the top-level block type, and the choice of the *Client* table as the top table within that query, *Rekall* has created a report with a sub-report, and has

created the blank report with a nested block (the sub-report) already in place. *Rekall* has also added headers and footers at both the report (*Client* table) level and at the sub-report (*Orders* and *Product* table) level. These are tagged to the right, with the number at the end indicating the blocking level within the *OrdersSummary Rekall* query. Had you set the top-level block type to access a table or a free-text SQL query, then there would have not been any sub-report (and the tags would change appropriately).

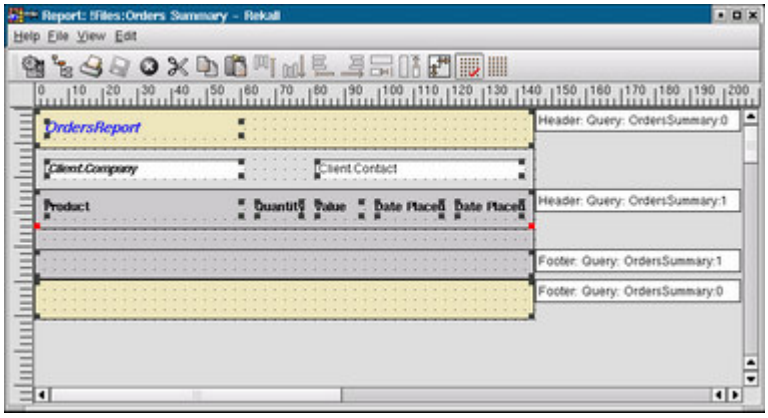
There are two things to notice about blocks in reports. Firstly, they all have the same width, and are all aligned to the left. This is because of the way that reports are generated; output is produced row-by-row, advancing down the page and throwing pages where needed. Secondly, they do not have the *dx* nor *dy*properties; movement between rows is always down the page, and the distance is controlled by the height of the block less the height of the header and the footer (since the block is generated once for ecah row).

Essentially, a report is executed by processing each row that is retrieved from the server database, generating output as needed. In this report, since the data comes from a *Rekall* query, and the *Client* table has been set as the top table, execution can be thought of as iterating over each client (in the top level block), and for each client iterating over each order (in the nested block). Page throws are controlled by the *Page throw* block properties, according to the table below:

None	Page throws only occur when a page is full
Group	A page thown occurs after the last record
Record	A page throw occurs after each record

In the *Orders* demonstration database, the *Page throw* property of the inner block is set to *Group*. Since *last record* is interpreted as meaning the last order record for the current client, there will be a page throw between clients (plus, of course, page throws if a page becomes full). Each time a page is thrown, footers are output for the block that is processing records, and all enclosing blocks, and headers are output similarly. *Rekall* keeps track of the amount of space needed for the footers and headers [29].

The image below shows the report at an early stage of development. A few basic controls have been added, and the sizes of the blocks, and headers and footers have been changed a little. In this state, executing the report would place the title *Orders Report* and the top of each page, followed by the company and contact names for the client, and a set of column headings.

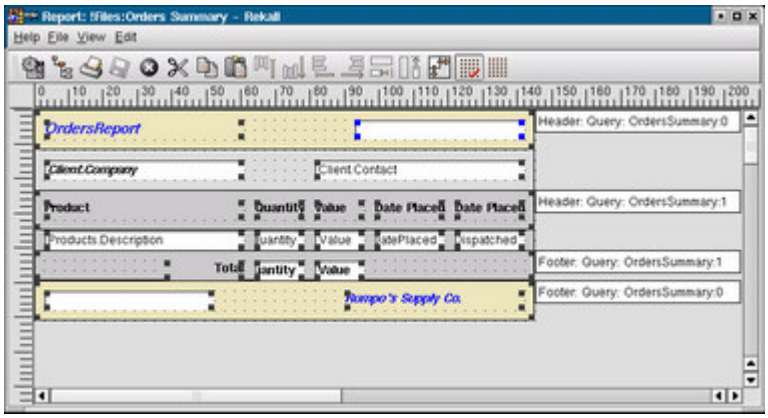


The remainder of the controls can are now added, as shown in the screenshot a little further on. Various of the controls are noteworth. The *Quantity* and *Value* controls top the right of the *Total* label are *Summary* controls. These are like normal fields, but accumulate information, and have a property *Summary function* which controls their behaviour; currently, *total*, *minimum* and *maximum* are supported. Summary controls are always reset when the block in which that are embedded is finished (note that headers and footers are two other examples of *containers*, so controls that are embedded in them are associated with the block the the header or footer is embedded in). They also have a property which controls whether they are reset each time a page is thrown (so you

can do per-page summaries or running summaries).

The bottom-left control is a field, however its *Display expression* (ie., the expression which is retrieved as part of the server database *select* query) is actually set to *'Page %{pageno} of %{pagecount}'*. This value retrieved will be exactly this string for all rows retrieved, but just prior to it being output, the *%{.....}* parts are substituted with the page number and the total page count.

The top-right control is similar, but with its *Display expression* set to *=time.strftime("%d-%b-%y", time.localtime (time.time()))*. This is actually a very small piece of *python* script, specifically it is a *python* expression and, rather than forming part of the *select* query, the expression is evaluated each time a value is needed, and will return a formatted date-and-time string [30]



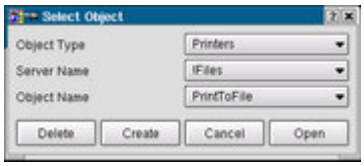
From here, you can switch to data view to display and print the report.

Printers and Printing

In a simple situation, you may have a single printer which only ever has a single type of stationery in it, and this printer is the default for all your applications. On the other hand, a the situation might be more complicated, perhaps there is a one printer loaded with pre-printed A4 paper, a second with A5 paper, and a third containing labels. In this case it would be anoying (and maybe error prone) for the user to have to select the appropriate printer whenever they print a report. To make life more difficult still, suppose that the three-printer *Rekall* application is to be installed at more than one site, and that the printers are named differently at each site (maybe different models of printers are used, or at one site printers are local while at another they are networked).

Rekall provides a mechanism to handle this gracefully, in which you can define *logical* printers, and then specify which logical printer a particular report. All that is needed when installing a *Rekall* application at a particular site is to configure the logical printers appropriately. You can think of this as a more sophisticated equivelent of the *Print Setup ...* functionality found in other applications. Of course, for the simple single printer situation, you can ignore all this, and *Rekall* will behave much like most other applications; when you press the *Print* button, the standard print dialog will appear.

Logical printer configuration is accessed via the *View/Show objects* menu. This brings up a dialog which can be used to quickly access objects such as tables and forms, as well as printers. To create a new logical printer, select *Printers* as the object type and choose the server where the definition will be stored; as usual the *!Files* entry means that the definition will be stored in the file system [31] . Then click *Create*. This will bring up the standard print dialog [32]



The required settings for the logical printer (printer name and properties) can be set as normal, but when the *Print* button (or *OK* button) is pressed, rather than any printing taking place, a save dialog will appear into which you can enter a *logical* printer name. This name is completely separate from the real printer name, but if you were printing in greyscale and in landscape on an A5 printer then you might use the logical name *LandscapeA5Grey*.

Logical printer settings can later be changed, or deleted, again via the *View/Show Objects* menu. There is no mechanism to rename a logical printer, but you can edit it (leaving the settings unchanged), save it under a different name, then delete the original.

A logical printer can be specified as one of the properties of a report. When a report is printed, the printer is determined as below. There is also a report option used to specify that the print dialog should always be shown.

- If no logical printer is specified but there is a logical printer named *Default* (note that the name is case sensitive) then the settings associated with *Default* will be used.
- If no logical printer is specified and there is no logical printer called *Default* then a standard print dialog will be shown.
- If a logical printer is specified and one exists with that name, then the settings associated with that logical printer are used.
- If a logical printer is specified but one with that name does not exist, then a warning is displayed and the standard printer dialog is shown.

Design View, Data View, Print and Preview

A report can be displayed in design view or data view. Design view, which has been described above, is where you design the report.

Data view is essentially a preview mode, where the report output is displayed in a window in the screen, one page at a time. *Rekall* takes note of printer settings to derive the page size, and converts these to screen sizes. *Rekall* does its best to generate output that is the same physical size as would be physically printed, but this may not be completely accurate, and you should not rely on the size that appears on the screen.

You can toggle back-and-forth between design and data view. In both views, the toolbar shows a printer tool. In data view this will print the report; in design view it will print the report design [33]. By the way, you may have noticesd that printing is available for forms, in both design and data view; in design view then the form design is printed, while in data view the form and its current content are printed.

Right-clicking on a report under the reports tab of the main database dialog wll bring up a popup menu which has, in addition to the *Data View* and *Design View* options, a *Print Report* option. This can be used to print a report directly without going via data view.

In addition, unded KDE from release 2.2 onwards (ie., the releases which have the KDE print dialog rather than the QT print dialog), report printing can be previewed by using the preview option in the print dialog. In this case, *Rekall* generates print output exactly as is would if the preview option were not selected, and the preview function is then handled by KDE's print system.

---

[29] The situation where the space required for the headers and footers is such that there is no space left on the output page is detected, and treated as an error!

[30] To get this to work, the *Import modules* property of the report must be set to include *time*. This is explained in the chapter on scripting.

[31] In keeping with most other *Rekall* objects, the definition is stored as XML, so you can look at it or even edit it by hand.

[32] The appearance of the dialog, and the control which it provides will depend on which version of *Rekall* and which version of KDE you are running. For the QT-only version of *Rekall*, and for KDE versions of *Rekall* on KDE 2.1.x, you will see the QT printer dialog. From KDE 2.2.x onwards, you will see the KDE printer dialog. *Rekall* stores most settings that the print dialog provides.

[33] This is currently rather basic, and will be improved in a later release.

Chapter 8. The Structure of Forms and Reports

Table of Contents

To go further with designing forms and reports, it is useful to know something about the way *Rekall* structures these internally, what types of objects can be embedded into forms and reports, and the settings that apply to each type of object. This is covered in this chapter.

The first two sections in this chapter outline describes the types of objects.

Form Controls

The various types of control than can be embedded into a form are listed below. Note that the data control (*Field*, *Memo*, ...) are not available in a *Menu* (*null*) block.

Field

A *field* is a simple one-line text entry control. You can set various properties such as font and colour.

Normally, a field is implemented as a QT line edit control, but fields can be *morphed*, that is, when input focus is not in the field then it is handled directly by *Rekall*. This is provided mostly for use in table data views where there are a large number of columns (and hence fields) displayed at the same time, whence morphing makes screen update much faster.

Memo

A *memo* is a multi-line edit control. In the current release of *Rekall* it does not do anything clever like word wrapping. Note that when in a *memo* control, *tab* insertes a tab character, rather than moving focus to the next control.

Note that *Rekall* currently lacks proper multi-line control for use in reports. This will be addressed in a future release.

Choice

A *choice* control is a combo-box which displays a defined set of options (stored as one of the properties of the combo-box).

Choice controls can be *morphed*, as for *field* controls; when input focus is not in the choice control, then it displays as simple text. This can be useful if space is limited, since the combobox drop-down arrow does not usually display.

Link

A *link* control is also a combo-box, but rather than have a defined set of options, it displays values from another table (or a *Rekall* query or SQL query). You specify a target table and a column in the target table contains values that are stored in the table or query to which the *link* control refers, and an expression based on columns in the target table; the expressions are displayed as the options in to combo-box.

For instance, suppose you what to store a person's title (for instance, *Mr*, *Mrs*, and so forth), but that you are not sure in advance that you know all possible titles. So, you create a *Title* table that has two columns, one a primary key and the other the title. Then, in say a *Client* table, there is a *title* column whicn stores primary key values from the *Title* table. In a form you then use a link control which matches the *Client.title* column to the primary key in the *Title* table, and displays the title text.

This has the advantage that to add a new title, you just add a new entry to the *Title* table. Indeed, by changing the title text in the *Title* table, you could change all the titles from, say, English to German.

Link controls can be *morphed* exactly as choice controls.

A link control can show more than one column of information from the target table. For instance, if the *display expression* is set to *col1, col2, col3* then the link control will show three columns of information, respectively *col1*, *col2* and *col3* from the target table (or query). This can be useful if you want the user to see more information about

the linked record that they are selecting. By default, all columns are shown both in the control itself and in the drop-down. However, the number of columns shown in the control itself can be limited by setting the *Show Columns* property to a non-zero number.

Pixmap

A *pixmap* control can display an image. *Rekall* knows about a reasonable selection of image types [34]

Check

A *check* is a simple yes/no checkbox. Note that the checkbox itself does *not* include a label (unlike the underlying QT checkbox control), so in the illustration below, the label is a separate object.

Rich Text

This control is displayed using the *QTextView* widget provided by the *QT* toolkit. It is mainly included for future use in *Rekall* itself, to display help information and such like. Text to be displayed in this type of control should be formatted as QT Rich Text (see Troll Techs documentation for details), which is basically a very much stripped down HTML.

Row Mark

Rowmarks do not display actual database data. Rather, they are used as a per-row marker, and show icons to indicate the current record and whether the record has changed. In addition, they can be set to show a row number.

Label

This is a text label. Label text can be formatted as QT rich text, for instance the illustration below shows the text underlined. The text is fixed except that it can be changed from a script.

Button

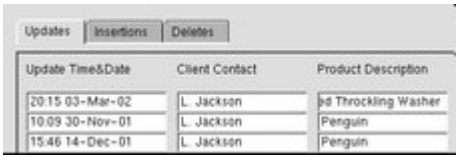
A *button* is a standard button control. For it to do anything it needs to invoke a script, although there are the

shortcuts like *#Click* described in a previous chapter to handle simple operations.

Tab Control

A *tab control* is a container object which can contain one or more pages, each of which is associated with a named tab. Once you have created a tab control, you can add pages; the pages themselves are in effect containers, into which other controls can be placed.

All data controls in all pages of a tab control are in effect embedded in the block into which the tab control itself is embedded. So, for instance, if the block takes data from a table which has a large number of columns, you can use a tab control to show different sets of columns under different tabs.



Container

As well as the *tab control*, a form can also embed a simple container object. There are two main uses for this.

Firstly, if you wanted some control to appear in an area of the form which has a different background colour, or perhaps shows as a raised or sunken panel, you can create a container. The container is given the required colour or effect, and the controls placed into the container. The main menu form of the *RekallDemo* database uses this technique for the bottom-right area (the other three areas are actually *menu-only* blocks, but the effect is the same in this situation).

Secondly, if you have a form which has the property of being *Stretchable*, then you can use a container with suitable *X-mode* and *Y-mode* stretch properties to get various resizing effects.

Report Controls

Unlike a form, a report can "display" an arbitrarily large number of rows of data, depending only on the number of rows returned from the server database. The basic mechanism is that the report retrieves data for its outer block, and then writes one set of values for each row. If the outer block contains a nested inner block, then it will perform the same for the nested block, repeatedly for each row fetched by the outer block (and so on if there are further nested blocks).

The report also output headers and footers at the beginning and end of a block, and whenever it starts a new page. It will start a new page either when there is insufficient space left on the current page for another row *and* the footers of the current and outer blocks. You can also set a block to start a new page (irrespective of whether there is sufficient space left) for every record, or to start a new page immediately after the last row is output.

For each row, the report advances down the output page by a distance equal to the height of the block in the report design, less the height of the block header and footer.

Field

A *field* control is a simple one-line text control. You can set various properties such as font and colour.

Link

This operates like a *link* control in a *form*, except that the linked value is output in text.

Pixmap

A *pixmap* control can display an image. The same set if image formats will be available as for a *form pixmap*.

Summary

A *summary* control is a simple one-line text control like a *field*. However, rather than display individual values, it calculates a summary (currently, *min*, *max* and *total* are supported).

A typical use is in the footer of a report block, where it can be used to generate a summary from each row which is output as part of the block.

The control can be set to reset on every page throw (for per-page summaries); otherwise, it effectively resets at the end of the block in which it is embedded.

Label

This is a simple text label. Unlike a form label, the text is *not* displayed as QT ricj text. The text is fixed except that it can be changed from a script.

Headers and Footers

The report designer automatically adds headers and footers to blocks which it considers to be large enough [35]. Headers are footers can contain any of the above controls.

Forms and Reports are Trees

To save text, in this section we'll refer to forms, but unless explictely mentioned, this applies to reports as well.

A *form* is organised as a tree of objects. The top-most level object is the form itself, and contains information which is global to the form (for instance, the form caption). A form is also a special case of a *block*.

A block is the most important sort of object. It is aa object in which other objects are displayed, and it is the point at which data is retrieved from the server database, displayed and possibly updated. A block can contain controls such as *fields* (simple text entry), *images*, and so forth.

A block can also contain subblocks; this provies the form-subform structure (and similarly report-subreport). Note that this can in principal be nested to any depth, and a block can contain more than one subblock. Blocks also handle some special controls, such as *rowmarks* .

A block in a form can also contain explicit *containers*. These can be used for layout and presentation (for instance, a block might have a grey background, but include a container with a yellow background). Note that so far as retrieving data from the server database is concerned, control that appear inside a container is logincally within the block that encloses the container.

A block in a report can have a *header* and a *footer*, as noted above. These are special cases of containers.

Objects are Classes

Rekall objects themselves are structured like classes in an object-oriented language; for instance, as mentioned above, a *form* is a special type of *block*. The complete structure is shown below; hence *KBItem*, *KBButton* and *KBLabel* are special cases of *KBObject*. Those marked with an asterisk never exist in their own right, but only as part of some more specialised object (in object-oriented terms, they are abstract base classes).

The names are all prefixed by *KB* for historical reasons.

- KBNode*
  - KBObject*
    - KBItem*
      - KBBlock*
        - KBFormBlock
          - KBForm
          - KBFormSubBlock
        - KBReportBlock
          - KBReport
          - KBReportSubBlock
      - KBField
      - KBChoice
      - KBCheck
      - KBLink
      - KBPixmap
      - KBMemo
      - KBRowMark
      - KBHidden
    - KBButton
    - KBLabel
    - KBFramer*
      - KBHeader
      - KBFooter
      - KBContainer
      - KBTabber
      - KBTabberPage

KBNodes, KBObjects and KBItems

KBNodes, *KBObjects* [36] and *KBItem* never exist in their own right, rather they only exist as part of some other real object such as a *KBButton*. However, they contain information which is mostly common to the real objects.

KBNode

A *KBNode* is at the bottom of the hierarchy of classes. Nothing of it is visible to the outside world except for the *Notes* property (which can be used to annotate a node, for instance for documentation, but is otherwise ignored by *Rekall*).

KBObject

A *KBObject* occupies some area of the display, and hence has properties like position and size. It can also have a name which can be used in scripts to identify and maniplulate a particular object. In practice, names should be unique amongst objects at a given level in the form or report hierarchy (for instance, amongst all the objects which are children of a particular block), although *Rekall* makes no attempt to enforce this.

KBItem

A *KBItem* (think *data item*) is an object that occupies display area, and contains data which generally (though not neccessarily) comes from the server database. As such it has properties like an expression which is used to specify the required data; it also has *event* properties which are trigged when, for example, a data value is set.

Note that a *KBItem* may actually hold several date values. This occurs when the *KBItem* is embedded in a block which is displaying more than one row from the server database (so in item may be instantiated as one or more data controls).

KBBlock and Friends

As alluded to before, the *KBBlock* object is the most important type of object in *Rekall*. First, however, note that a *KBBlock* never exists in its own right: in a form is will appear as a *KBFormblock* (or *KBFormsubblock*, see below) or in a report as a *KBReportblock* (or *KBReportsubblock*, see below). However, we will use *KBBlock* as shorthand.

Unless the *KBBlock* is the outermost, it itself holds a value from the enclosing *KBBlock*, in exactly the same way as a *KBField* or any other control which displays data. The difference compared to these controls is that the value is not actually displayed. Instead, the value is used when forming the SQL query that retrieves data for controls which are enclosed in the nested *KBBlock*. In the form-subform examples earier, the inner block (ie., the subform) was set to retrieve *ClientID* from the outer block (the *Parent field* setting), and link this to *ClientID* in its own query (the *Child field* setting). Hence, the query generated for the inner block would be like:

```
select ...
  from  Orders
  where Orders.ClientID = value
```

where *value* is the *ClientID* value from the *Clients* table which is shown in the outer *KBBlock*. As you step through records in the outer block, the query for the nested block is repeated to get the correct set of orders for the client now being displayed. Confused? Fair enough. I have to think about it each time I revisit the code. But it works! If you bring up the query log window (via the scroll-and-Q toolbar icon, you can see the text of the queries as they are executed.

A block can display a single row of data at a time, or it can display more than one; one of the properties of a block

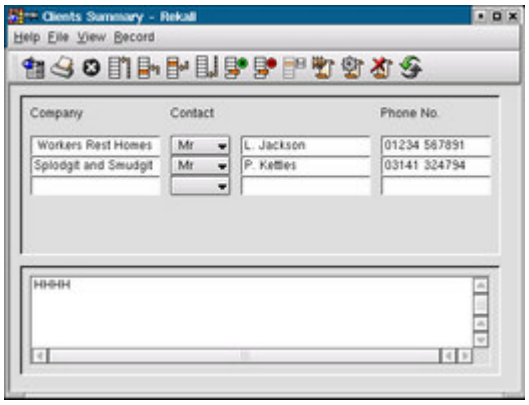
is the number that should be displayed. A special case is if this is set to zero, in which case the number of rows is calculated to just fit the space available.

There is one further subtlety. Normally, in a form-subform arrangement, the outer block (the form) would show exactly one row of data (such as a client), and the inner block (the subform) would show multiple rows (such as orders for the client). However, in *Rekall*, the outer block *can*, if you wish, display more tha one row. In this case the inner block displays rows of data related to the *current* row in the other block - if you think about it, the outer-one-row situation is just a special case of this, where the current outer row is always the one displayed.

Why would you want to do this? Suppose you have a form which shows several rows of clients information, but you would also like to display a notes memo control for just the client corresponding to the row which is current at any given time (because this field will take up quite a lot of space). You would add a nested block which also retrieves data from the clients table, with the inner and outer blocks linked on the client identifier; this means the select query for the inner block is

```
select Notes
  from  Client
  where Client.ClientID = value
```

with *value* being the current client identifier from the outer block. The inner block need only display a single row. This is shown in the screenshot below. Note that the outer block has a container whose *Y-mode* is set to *stretch*, and the inner block has *Y-mode* set to *float*; this means the form height can be changed by the user with sensible results.



Data Controls

The data controls are things like *KBField*, *KBMemo*, etc., which (usually [37]) take data from the database and display it on screen (or in a report).

In a report, data is output from the database row by row, so a block will end up showing as many rows of data as are forthcoming. However, in a form a *KBBlock* may display more than one row of data at a time. In this situation, the data control holds as many values as there are rows of data on display, and there will be multiple instances of the control shown (ie., a *KBField* may correspond to several line edit controls).

All data controls have a common set of operations and settings which specify how they relate to the server database. Individual types have their own specific settings. For instance, a *KBPixmap* has a setting for a frame to be drawn round the image, which a *KBChoice* has a set of possible values.

The *KBHidden* control is special, in that is does not actually display. Rather, it can be used if you need to retrieve a value from the server database for use in a script, but which does not need to be displayed to the user.

Containers: KBHeader, KBFooter, KBContainer, KBTabberPage

These objects are *containers* for other objects. The *KBHeader* and *KBFooter* objects are used as headers and footers respectivly in reports, and do not appear in forms. A *KBContainer* is used as a container in a *KBFormblock* and does not appear in reports. A *KBTabberPage* is a page within a tabbed control (actually, the tabbed control it itself a container, but the *only* object that it can contain are pages).

The essential purpose of a container is to provide a means of grouping together some controls. The controls really belong to the *KBBlock* in which the container is embedded, but their position is controlled relative to the container and not the *KBBlock* itself.

One use of a container is for report headers and footers. The header is a container which is set to have the same width as the block in which it is embedded; the footer similarly has the same width, but is locked to the bottom of the block.

Another instance of a *KBContainer* is the upper part of the table design form. The lower half of this form is a subblock, which set to have fixed height but to stick to the bottom of the form. The upper half is a *KBContainer*, and is set to change in height as the form is resized. As it does so it adjusts the number of rows of data (in this case, information on table columns) that are displayed. Note that, when a block with a zero rowcount calculates how many rows to display, it calculates the values for itself and for any embedded containers, and uses the minumum number.

Notice that the containers are special cases of *KBObject*, rather than *KBItem*, since the containers themselves display no data.

Forms and Reports

Lastly, at the top of the pile, as it were, are the *KBForm* and *KBReport* objects. These contain settings that are global to the form or report, for instance what scripting language to use (currently only Python is supported), and what script modules to load.

Properties

Properties are what control the behaviour of objects in *Rekall* forms and reports. Loosely, these can be divided into two groups, *event* properties and non-event properties. The former are used when you add scripting to forms and reports, and are covered in the chapter on scripting; the remainder of this chapter describes, in more detail, the non-event properties that are more frequently used. The first section covers properties that are more or less common to a number of objects; following sections cover those that are specific to individual objects. A summary of all properties can be found in an appendix.

The names in brackets are the actual property names; these are used by the *getAttr* method described in the next chapter.

Common Properties

The properties listed in the section are more-or-less common to all objects, although there are exceptions (for instance, *KBForm* and *KBReport* do not have an X-position nor Y-position).

Notes

This property appears in all objects. *Rekall* ignores it, but preserves its value. It is mainly intended for

documentation. You might use it to store information which a script can retrieve [38]

X-Position, Y-Position, Width and Height (x, y, w, h)

These properties specify the location of the object on screen, relative to the object that that they are embedded in. The default is absolute position of the top-left corner, and absolue size, but note the *X-mode* and *Y-mode* properties below.

The X-position and Y-position properties do not appear for the top-level block (ie., for what is really a form or report object).

X-mode and Y-mode (xmode, ymode)

The X-mode setting controls the way that the object responds to changes in the width of its parent; the three posibilities are:

- *Fixed*: the object's position and width are not affected by changes in the parent. The *X-position* and *Width* settings are the offset of the left-hand edge of the control from the left-hand edge of the object in which it is embedded, and its width respectively.

- *Float*: the object's width stays constant but it remains a constant distance from its parent's right-hand edge. In this case the *Width* setting is the objects width, but the *X-position* value is the distance from the right-hand edge of the control to the right-hand edge of the object in which it is embedded.

- *Stretch*: the object's position stays constant but its width changes to match that of its parent. The *X-position* value is the offset of the left-hand edge of the control from the left-hand edge of the object in which it is embedded; the *Width* setting is distance from the right-hand edge of the control to the right-hand edge of the object in which it is embedded.

The Y-mode setting correspondingly applies to vertical position and height.

Whenever the *X-mode* or *Y-mode* value is changed (when in design view), the position and size values are adjusted so that the object remains at the same place on the screen (hence, you can lay out objects leaving *X-mode* and *Y-mode* set to the default *Fixed* value, then change them as required later on.

Note: If you have a block which displays more than one row of data, then while it makes sense to set a data controls X-mode (or Y-mode) to *float*, setting it to *stretch* may make it look very strange if the block changes size.

Control name (name)

Objects can be given names, the main use of which is in scripts in order to identify and access controls (for instance, a script can locate a control by name, then update its value). Although *Rekall* does not enforce this, you should probably not give the same name to two or more objects which are embedded in the same container (for instance, two fields in the same block), since accessing a control by name from a *python* script could refer to any of the objects. Using the same name for objects in different containers would not be a problem.

The standard *Rekall python* library supports the feature whereby setting a *KBButton On Click* even to *#Click* can be used to create simple record navigation buttons (First, Previous, Next, etc) by using the *KBButton* name to specify the operation; similarly *#GoForm* and *#GoReport* will invoke the form or report whose name is the *KBButton* name.

Background Colour (bgcolor)

English spelling, folks! This setting specifies the background color to be used in applicable objects, such as

KBButton and *KBField*. It also applies to blocks.

Frame Style (frame)

Some control - such as blocks, labels and pixmaps - can have a frame. There are three components to the frame, a *shadow* effect (plain, sunken or raised), a *shape* (various options such as panel and winpanel) and a *width*. The property dialogs for the objects allow these to be controlled individually, and show an image of the general effect.

Text Colour (fgcolor)

This setting specifies the text color to be used in applicable objects, such as *KBButton* and *KBField*.

In form labels and in the rich text control, text colour (as well as other properties) can also be set if the text that is displayed is formatted using the QT Rich Text format. See Troll Techs's QT documentation for details.

Display Expression (expr)

This setting appears for objects that actually display data, such as *KBField* and *KBChoice*, and is the expression used to retrieve data from the server database. It must therefore be a valid SQL expression (for the type of SQL server; *Rekall* does not provide server independance at this level).

This expression may be empty, in which case the control does not interact with the server database. Values may be set and retrieved by scripts.

A special case is an expression of the form = *expr* where *expr* is a valid *python* expression, in which case the expression is evaluated when a value from a server database would otherwise be displayed. For example, = *time.strftime("%d-%b-%y", time.localtime (time.time()))* will show the current date (provided that the *python time* module is imported).

Data-Related Properties

The next set of properties generally apply to data controls (such as *KBField* and *KBChoice*), though again not all apply in all cases.

Row Count (rowcount)

This is not actually a property of a data control, but rather of the block which contains the data control, and specifies the number of rows of data which are displayed. Hence, it gives the number of instances of the data control which will appear.

If the value is zero, then the block will calculate the number of rows according to the block size and the block X- and Y-delta properties. As noted above, this check also includes any containers that are embedded in the block; the minimum value is chosen. If the value is still calculated as zero, then a single row will be displayed (so you might get a scrambled display, but at least something will appear!).

X and Y Spacing (dx, dy)

These are also properties of the block which contains a data control. If the block has the *rowcount* property set to zero, then these values are used for the spacing between control, and hence affect the number of rows of data that are displayed in the block.

Where the *rowcount* is zero, and the block contains embedded containers, the same spacing values are used in the

containers.

Default Value (defval)

If a row of data is saved to the server database and a data control has not been set, then the *Default* value will be used, if any. This is typically useful in a form to save the user having repeatedly enter the a common value. Note that this is completely independant of any column default value that may be provided by the server database itself.

Null OK (nullok)

This property should be set of it is OK for the data value to be empty when data is saved to the server database. If the property is not set then the user must enter some data.

This is also independant of any column not-null setting provided by the server database itself. However, where *Rekall* detects that a column is marked not-null in the server database it will check the control value whether this property is set or not.

Validator (evalid)

If this is not blank, then it is used as a regular expression against which user-entered data is checked. The regular expressions are those supported by Troll Techs QT library [39]. Note that as of this release, the expression is not anchored at either end.

Ignore Case (igncase)

Setting this property causes user-entered data validation to be case insensitive.

Read Only (rdonly)

If set then the user cannot alter the value, ie., it is for display only (although it can still be changed by a script). If you do set this option, then you might want to do something like change the text colour to give the user a hint.

Format (format)

The *Format* property is used to specify how the raw data from the server database is formatted for display.

The properties dialog will display a set of options that allow you to construct a valid format specification, although you can edit the format by hand. The format specification must match the type of data coming from the server database; again, the properties dialog will select the appropriate type for you. If you do run a form and the specified format is not applicable to the value (for instance, if a table column type has been changed) then the control will display something like *Format?Date*.

The simplest way to specify advanced formating (for example, of dates and times, which uses the *strftime* style of description) is to use the properties dialog to generate the nearest and then to edit that. The table below lists the main format types.

Date	Formats dates using *strftime* formatting. The time formats associated with *strftime* should not be used.
Time	Formats times using *strftime* formatting. The date formats associated with *strftime* should not be used.
DateTime	Again formats using *strftime* formatting; both date and time parts are value.
Fixed	Used for fixed (integer) values. The format string uses *printf* formatting; the single argument to this is the integer value.

Float	Used for floting point values. The format string uses *printf* formatting; the single argument to this is the double value.
Number	The format string is similar to that used for formatting in another well-known database product, and is described below.
Currency	This format attempts to format the value as a currency value. The format string is interpreted as a locale (eg., *gb* for the UK, *de* for Germany, and so forth), and is used to retrieve locale-specific formatting. If it cannot be found [a] then a default is used. An empty format string means the current locale.
[a] Which locales are available depends on which are supported and installed on your system.	

In the case of *Number* formatting, the format string can contain the following:

.	Decimal point. Number is separated into whole and fractional parts.
0	Outputs a digit. This format character converts leading spaces to the digit zero, hence *134* formatted with *00000* would output as *00134*.
#	Outputs a digit, but supresses leading zeros. Hence *134* formatted with *####0* would output as *134*.
$	Outputs the currency symbol for the current locale, falling back on *$* itself if if cannot be determined.
\	Escapes the following character, which is output as is, hence *\$* will output the dollar character and not the currency character for the current locale.
E	Starts exponent part of number. Lower-case *e* can also be used.
"	Starts quoted string; quoted string is output as-is.

In addition, a number format can comprise up to four sections, separated by the semi-colon *;* character. If the value is null then the fourth section is used if present; if the value is zero the the third section is used if present; if the value is negative then the second section is used if present; otherwise, the first section is used. Null is treated as zero in this context.

Text Alignment (align)

This specifies text alignment in a simple text fields and in labels. The default is left, the alternatives are right and centered.

In addition, labels have the setting *rich text*, which uses the QT rich text formatting, essentially simple *HTML*. This allows effects like bolding and underlining; also, text will be wrapped to fit if neccessary.

Input Mask (mask)

The *Input Mask* property gives some control over text entry. The mask is a text string where the following characters are significant (note thay any other character stands for itself):

A	An upper-case character. Lower-case will be converted
a	A lower-case character. Upper-case will be converted
0	A digit
_	Any single character

This area of *Rekall* is under development, so expect more mask functionality in future releases. This may require incomaptible changes.

Block Properties

The next set of properties apply to blocks, although some only appear in form or report blocks and not both. The *rowcount* and *X/Y delta* form block properties have been described above under the data control properties.

Show Scroll Bar (showbar)

If this property is set then a block will show a vertical scroll bar at the right-hand side, which can be used to scroll through rows that are displayed in the block.

Note that some user confusion may occur if the form is resized and this property is set, since it is then possible to have a right-hand scrollbar which moves through records, and a horizontal scrollbar which scrolls the form left-right in the window.

Parent/Child (master, child)

These two properties are used to link data in a nested block to the data in the block which contains it. *Parent* is a column (or, in general, an expression) which will be retrieved with the data for the outer block; *Child* is a column (or expression) which applies to the nested block. The data which appears in the inner block comprises rows which have the same value for *Child* as the value of *Parent* in the outer block.

For instance, if the outer block retrieves data from the table *Clients*, and the nested block from *Orders*, with *Parent* and *Child* both set to *ClientID*, then the effect is much as if you had used the SQL query:

```
select .....
from   Clients, Orders
where  Clients.ClientID = Orders.ClientID
```

Form Properties

The properties in this section apply to form documents. A form object is actually a special case of a block, which has a number of additional properties which are global to the form.

Streatchable (stretch)

Forms may be fixed size or stretchable. If they are fixed size then resizing the window which displays a form below the size of the form itself (strictly, the size of the top-level block) will result in scroll-bars appearing. If, however, the form is *stretchable* then the size of the form will be adjusted to match the size of the window.

This setting can be used in conjunction with the *X-* and *Y-mode* control settings to provide basic geometry management. For instance, if the number of rows in the top level block is set to zero, then resizing the form and hence the block will alter the number of rows displayed.

Scripting Language (language)

This setting controls the scripting language to be used within the form. Currently, only *py* (for *python*) is supported.

Form Caption (caption)

This property gives the caption which appears in the title bar of the window which shows the form. Note that you can control the caption which appears when the form is started by embedding a parameter into the caption value, for

instance *Orders For ${date}*; see the later description of parameters.

Script Modules

When a form is executed, any script modules listed under this property are loaded into the script interpreter (in addition to the standard *Rekall* modules). These should be script modules which appear under a *Scripts* tab in the database window.

Note that although this appears as a property, each specified module is stored in the XML form definition as a separate node. For this reason, the property cannot be manipulated from a script (not that this would make much sense anyway).

Scripting is described in much more detail later.

Import Modules

This property lists modules which should be implicitely imported into any script code which is attached to an event (such as *OnAction* or *PostSync*) or expression (such as the *defval* default valuen property).

This is needed since the way in which script code is specified for events and expressions precludes module import. So, for instance, if an event uses a *python* function such as *time.strftime* when you should include *time* in the import module list.

As for script modules, the each import module is actually specified via a separate node in the form's XML definition.

Report Properties

The properties in this section apply to report documents. A report object is actually a special case of a block, which has a number of additional properties which are global to the report. The *Script Modules* and *Import Modules* properties are the same as for a form.

Margins (lmargin, rmargin, tmargin, bmargin)

The values, all expressed in millimeters, give the left, right, top and bottom margins used when printing a report. When a report is created they are set to default values (which can be changed via the *View/Options* menu selecting the *Report* tab), but can then be changed independantly of the defaults.

Printer (printer)

This property is used to specify which *logical* printer is to be used to output the report. See the end of the reports chapter for a description of exactly how this is used.

Show Print Dialog (printdlg)

Normally, if a logical printer is specified and is correctly defined, then the standard print dialog is not displayed. Setting this property overrides this, and the print dialog is always shown.

---

[34] Actually, this depends on the image types which the *QT* library supports, but will probably include *bmp*

(Windows bitmap), *gif*, *jpg/jpeg* (Joint Photographic Group), *pbm* (Portable bitmap), *pgm*, *png* (Portable Network bitmap), *ppm*, *xbm* (X bitmap) and *xpm*.

[35] Currently, there is no way to add headers and footers to a block which lacks them, nor to remove them from a block where thay are not required (though you can collapse them to say a single pixel height).

[36] The use of the name *KBObject* is rather unfortunate, but its much too late to do anything about it.

[37] Only usually. A data control can be unbound, that it, have no expression associated with it. In this case it is not involved in any interaction with the server database. However, values can be set by scripts, and maybe retrieved by them,

[38] A planned future feature of *Rekall* is *user properties*, whereby you can add arbitrary name/value pairs to an object. The main purpose of this would be to store information for use by scripts.

[39] *Rekall* uses QT3.x regular expressions, even in versions which are built on top of QT2.x (that is, on KDE or Qtopia).

Chapter 9. Scripting with Python

Table of Contents

Rekall uses Python as its scripting language, to allow you to provide functionality over and above *Rekall*'s basic data display and update facilities. As well as giving access to the both the data that is retrieved from the server database and control over how it is displayed, you can have access to all the *python* libraries that are available. This chapter describes how to use *python* scripting in *Rekall* and assumes at least a basic knowledge of *python*.

Scripts can be located in three places. First, they can be embedded in form and report objects, whence they are

executed in response to various events. Secondly, they can be embedded in form and report objects as parts of expressions. And thirdy, they are stored in script modules which are in effect libraries available for import, much as the standard Python libraries.

Since, in this release at least, scripts execution always starts with an event, the next section deals with events.

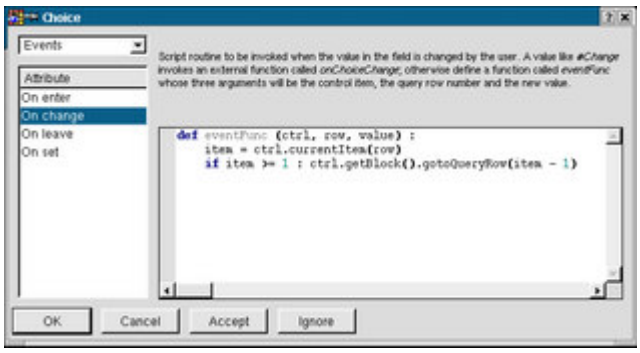
Rekall uses a number of scripts itself, both to provide some common script functions (for instance the code which allows record navigation buttons to be easily defined), and to support internal functionality, such as table design and data display. These can be seen in *$PREFIX/share/apps/rekall/script/py/*.

Introduction to Scripting

The first section of this chapter gives a basic overview of scripting, in the sense of what and where. Later sections go into more detail.

Events

The screenshot below shows the property dialog for the choice control in the *Client* form of the *RekallDemo* database. The choice control has an entry corresponding to each row displayed in a form, and can be used as a quick navigation tool [40] . The code shown is associated with the *On Change* event, which occurs when the user changes the selection.



The first thing to note is that the code defines a function called *eventFunc*. This is true of all events, since this is the name that *Rekall* will use. If you don't define *eventFunc*, then the result is undefined; most likely some other event function will get executed, but don't rely on any apparent consistency!

In the case of a choice control *On Change* event function, it is called with three arguments; the control itself, the number of the row in the query whose data is being displayed in the control, and the value now displayed by the control. The arguments will vary with the type of control and the particular event, except that the control itself is always the first argument to the event function. The control argument is actually an instance of a *python* class which corresponds to the control (and has the much the same inheritance structure as was described in the previous chapter), although you cannot actually instantiate such a class yourself. Also, you *should not* save a copy of this anywhere, since it will cease to be valid when control exits from the event function.

In the example here, the code retrieves the index of the selected value (*item = ctrl.currentItem(row)*), and uses this to navigate to the corresponding query row (*ctrl.getBlock().gotoQueryRow(item - 1)*). The additional test and the adjustment by one are there since choice controls always show a null value first.

Later in this chapter there is a full description of the methods which can be applied to control, but in this case, *ctrl.getBlock()* gets the *KBBlock* in which the choice control is embedded, and the *gotoQueryRow(item - 1)* does the navigation.

Some event functions (but not this one) should return a true or false result, where a false result will cause further operations to be abandoned. For instance, there is a *KBBlock On Action* event which is invoked just before an action such as *Next Record* is performed; if an event function is defined and returns false, then the action does not actually occur. If you are not sure whether or not you need to return a value, it does no harm to return a true result.

As a general comment about events, there are two ways of defining an action in *python*. The more general is to write a *python* function called *eventFunc*; this will be invoked with arguments that are specific to the event, and provide information about the object to which the event occured. The alternative is to write *#Foobar*, whence a function named something like *onBlockFoobar* is invoked. This is elaborated on in the scripting section.

*Warning:*When saving an event, *Rekall* checks to see if the first non-whitespace character is #, and if the # is followed by a letter (A-Z, either case). If so, it will trim any leading whitespace, and also everything after the name (so, *#Foobar rubbish* becomes *#Foobar*). Since *python* uses # as a comment character, you shouldn't enter an event function starting with a *python* comment unless the # is followed by whitespace, or it will be zapped [41]

Expressions

Data controls such as text fields have a *Expression* property which specifies the value to be displayed. Normally this would be an SQL expression used to retrieve a value from the server database. However, the expression may also be blank, in which case no value is fetched and the control is only accessed from scripts; or, as a special case, the expression may be of the form *=expr* where *expr* is a valid *python* expression.

In the latter case, whenever a server database derived value would otherwise be displayed, the expression is evaluated and the result displayed. The main use for this is to display information such as time and date, for instance *=time.strftime("%d-%b-%y", time.localtime (time.time()))* . Note that this particular expression requires the Python *time* module to be imported; see further on.

Modules

The third place to store Python scripts is in modules. These are accessed under the *Scripts: py* tag of the main database window. Just as for forms and reports, script modules can either be stored in the file system or in a server database

In most respect, *Rekall* script modules are just the same as standard *python* modules, and can be used once they have been imported. However, since the *python* import mechanism does not know about *Rekall* and where it stores scripts [42] it is necessary to explicitely instruct *Rekall* to import them. This is the purpose of the form and report *Script modules* and *Import modules* properties.

The *Script modules* property lists those script modules which should be imported for general use when the form or report executes. The effect is to preload the script modules into the *python* interpreter. For instance, suppose you have script modules *moduleA* and *moduleB*, and that the latter needs to be needs to import the former. Just writing *import moduleA* in *moduleB* will not work, since the *python* interpreter will not be able to locate it; however adding *moduleA* to the *Script module* property will preload it, whence *moduleB* can successfully import it.

The *Import modules* property fulfils a similar function with respect to event scripts and expressions. Scripts listed here are available within event scripts and expressions without the need for an *python* import statement; this is definitely neccessary with expressions where it would not be possible to write a Python import statement. Also, this list provides a means whereby standard *python* libraries can be loaded for use by event scripts and expressions.

An Aside: Query Rows

In the rest of this chapter, repeated reference is made to *query rows*. What is meant by this is the rows of data which a *KBBlock* retrieves from the server database; it may be data retrieved directly from a table (*select ... from*

tablename) or it may be retrieved via a *Rekall* query.

Whenever there is a need to identify a particular row, it is always in terms of an index into this data, irrespective of what rows are on display. *Rekall* will handle this; if you try to access the value in a data control for a row which is not currently displayed, *Rekall* will either ignore the operation or return a null value.

Examples

Before going though all the events, all the operations that can be performed on objects, and the other *python* functionality that *Rekall* provides, this section has what is hopefully a representative set of examples of things that can be done with scripting.

Record Navigation the Proper Way

In earlier examples, the shorcut mechanism was used to create record navigation buttons, where the *On Click* property was set to *#Click*. This calls some standard code in the *RekallMain.py* modules. However, the same can be done directly, as shown in the code below, which implements next record functionality:

```
def eventFunc (button) :
    button.getBlock().doAction(3)
```

When the event function is invoked, the first argument is the button. From this the enclosing block is retrieved, and the code then invokes action *3*, which is next record. This could be a little better done as in the next piece of code, the only extra requirement being that the form *Import Modules* property included *RekallMain@*

```
def eventFunc (button) :
    button.getBlock().doAction(RekallMain.actNext)
```

Doing things this way does not have any advantages over the shortcut, unless you want to do something else at the same time. Suppose, for instance, that the form has a checkbox (named *confirm*) which must be checked before the user can move on to a different record. Then:

```
def eventFunc (button) :
    block = button.getBlock()
    check = block.getNamedCtrl("confirm")
    if check.getValue(block.getQueryRow()) != "1" :
        RekallMain.messageBox ("Please confirm first!")
        return
    block.doAction(RekallMain.actNext)
```

This code assumes that the button and the checkbox control are in the same block. The code gets the block, then locates the checkbox control, and then checks the value. There is currently no explicit *isChecked* method, but the checkbox control will return values *0* or *1*. Whether or not the checkbox control is associated with a column from the table that the block retrieves data from (that is, whether it has an empty *Expression* property or not), it is still necessary to specify the query row, which is also retrieved from the block.

Now, there are two problems with this check in this code. Firstly, it would need to be replicated for all navigation buttons, although we could partially get around this by moving most of the code into a separate module, and just calling it from the event function. The second problem is more important, however; this code would not prevent the used clicking the toolbar next record button, or using a keyboard shortcut. However, we can get round both problems at once by moving the check to the block *On Action* event, which is invoked whenever an action (such as next record) is about to take place. The button event code can revert to the version without the test, and the block

On Action code is then (where the ellipsis are replaced by the other relevant actions):

```
def eventFunc (block, action) :
    if action == RekallMain.actFirst or ... :
        check = block.getNamedCtrl("confirm")
        if check.getValue(block.getQueryRow()) != "1" :
            RekallMain.messageBox ("Please confirm first!")
            return 0
    return 1
```

If the *On Action* event returns false, then the action is aborted, so this code has the desired affect. More importantly, it handles any navigation buttons you might add, it works if the user uses the toolbar buttons, and it works if the user uses a keyboard shortcut.

Locking Fields

This example shows how to lock fields against update depending on some criteria. Suppose that we have a form which shows information about products (actually, this could be the *Products* form from the *Orders* demonstration database), and that we wish to stop the user from updating some fields for particular products. For the example, we'll use the product code to control this; the fields are locked if the product code is equal to one.

The event function code below is attached to the block *On Display* event, which is executed each time a row of data is displayed (and will be executed several times in succession if the block displays more than one row). The block should contain a control named *Product* which retrieves the product code from the server database; this might well be a hidden field. Essentially, this code executes each time the set of values in a row are displayed, and enables or disables the *Quantity*, *DatePlaced* and *DateDispatched* fields.

```
def eventFunc (block, qrow) :
    ordinary = block.getNamedCtrl("Product").getValue(qrow) != "1"
    block.getNamedCtrl("Quantity"      ).setEnabled (qrow, ordinary)
    block.getNamedCtrl("DatePlaced"    ).setEnabled (qrow, ordinary)
    block.getNamedCtrl("DateDispatched").setEnabled (qrow, ordinary)
```

Much in the same way as the previous example, since the code is attached to a block event, it works correctly however the user navigates through the data.

Roll Your Own Form

As has been remarked earlier, forms and reports definitons are stored in XML, which you can view and, if you wish, edit yourself. Another feature which follows on from this is the ability to write scripts which construct entire forms or reports that are customised for specific situations; in this example the script is embedded inside the *On Click* action of a button.

When executed below, the code prompts the used to select a field from the *Client* table of the *Orders* demonstration database (this is the *RekallMain.choiceBox (....)* call. It then constructs a form which shows a small form which displays then client company name plus the selected field, along with a pair of navigation buttons. For added spice, if the user selects *Address* then the form concatenates the address fields and displays them (the code here assumes that the underlying server database is *MySQL*). The last line of the script, *button.getForm().openTextForm(form)*, opens the form that has been created.

```
def eventFunc (button) :
    name = RekallMain.choiceBox \
           (    "A client field, please:",
                [       "Telephone",
```

```
                        "Contact",
                        "Department",
                        "Address",
                        "PostCode"
                ]
            )
    if name == None :
        return
    legend = name
    if name == "Address" :
        name = "CONCAT(address1, ', ', address2, ', ', " + \
               "address3,', ',TownOrCity)"
    form = '<?xml version="1.0"?>' + \
           '<!DOCTYPE KBaseForm SYSTEM "kbaseform.dtd">' + \
            '<KBForm x="0" y="0" w="400" h="120" xmode="0" ymode="0"' + \
            '        name="UnnamedForm"' + \
            '        autosync="Yes"' + \
            '        rowcount="1" dx="0" dy="20" language="py"' + \
            '        caption="Client field: ' + legend + '" stretch="Yes">' + \
            '  <KBQryTable server="Self" table="Client" primary="ClientID"' + \
            '              order="Company"/>' + \
            '  <KBField  x="20" y="20" w="370" h="20" name="Company"' + \
            '            expr="Company" taborder="1" align="1"/>' + \
            '  <KBField  x="20" y="50" w="370" h="20" name="theField"' + \
            '            expr="' + name + '" taborder="1" align="1"/>' + \
            '  <KBButton x="20" y="80" w="70" h="30" name="Previous"' + \
            '            text="<" onclick="#Click"/>' + \
            '  <KBButton x="110" y="80" w="60" h="30" name="Next"' + \
            '            text=">" onclick="#Click"/>' + \
            '</KBForm>'
    button.getForm().openTextForm(form)
```

This is clearly not a trivial thing to do, and requires a fairly detailed knowledge of the XML that defines a form, but it illustrates one of the advanced things that *Rekall* can do. If you do want to do this, one way is to design a form in the formal way in order to get the basic layout, etc., correct, then use the XML for that form as the basis of the script.

There is work in progress to develop a set of *python* classes which can be used to do this more easily, for instance you would create a *python*form object, then add objects such as fields and buttons. This is essentially the XML DOM model.

Object Events

Button Events

On Click

This event is triggered when the button is clicked. The single argument is the button.

```
def eventFunc (button) :
    name = button.getName()
    RekallMain.messageBox ("You've clicked the '" + name + "' button!")
```

Item Events

On Set

This event is triggered when the value if the data control is set from the server database. The arguments are the control, the query row for which the control displays data, and the new value.

```
def eventFunc (ctrl, qrow, value) :
    if int(value) > 1000 :
        RekallMain.messageBox ("That's a very silly value!")
        ctrl.setValue (qrow, "0")
```

On Change

This event is triggered when the value if the data control is changed by the user. The arguments are the control, the query row for which the control displays data, and the new value.

```
def eventFunc (ctrl, qrow, value) :
    if int(value) > 0 :
        RekallMain.messageBox ("Don't set that checkbox!!")
        ctrl.setValue (qrow, "0")
```

There are two special cases. Firstly, this event is not available on fields, and secondly, on pixmaps, the value passed to the event function is undefined. If you need to process values from field controls, the field *On Leave* event, and the block *Pre-Insert* and *Pre-Update* events will probably suffice.

On Enter

This event is triggered when focus enters a control. The arguments are the control itself and the current query row number.

```
def eventFunc (ctrl, qrow) :
    if ctrl.getValue (qrow) == "" :
        ctrl.setValue (qrow, "42.00")
```

On Leave

This event is triggered when focus leaves a control. The arguments are the control itself and the current query row number. If the function returns a false result then focus remains in the control.

```
def eventFunc (ctrl, qrow) :
    if (ctrl.getValue(qrow) == None) or (ctrl.getValue(qrow) == "") :
        phone = RekallMain.promptBox \
                (   "Telephone",
                    "",
                    "Really no phone?")
        if phone != "" :
           ctrl.setValue (qrow, phone)
    return 1
```

Block Events

On Action

This event is called immediately before an action such as *Next Record* is called, the two arguments being the block and the action. If the event function returns false then the action is aborted. Action codes are defined in the *RekallMain* module; note that these values are also used as arguments to the block *doAction* method. The complete

set is listed below.

Code	Value	Meaning
actNull	0	No action
actFirst	1	Go to first record
actPrevious	2	Go to previous record
actNext	3	Go to next record
actLast	4	Go to last record
actAdd	5	Add a new record
actSave	6	Save record
actDelete	7	Delete record
actQuery	8	Start a query (search)
actExecute	9	Execute a query (search)
actCancel	10	Cancel query
actInsert	11	Insert a new record
actReset	14	Reset changes to row
actGotoQRow	15	Go to row by query number
actSyncAll	16	Save all updated rows

On UnCurrent

The *On UnCurrent* event is invoked when focus leaves a row in a block (ie., the focus moves to a control in a different row or in a different block); the arguments are the block and the query row number of the row being left.

Note that this event is *not* invoked when a row is deleted.

On Current

The *On Current* event is invoked when focus arrives in a row in the block (ie., the focus moves to a control in a different new row or block); the arguments are the block and the query row number of the row being entered.

The example below is taken from the *Client* form in the *RekallDemo* database. This has a combobox which can be used to navigate between records; the code here updates the combobox whenever the current record changes.

```
def eventFunc (block, qrow) :
    Orders.onBlockCurrent (block, qrow)
    selector = block.getNamedCtrl ("selector")
    if qrow >= block.getNumRows() :
           selector.setCurrentItem (qrow, 0)
    else : selector.setCurrentItem (qrow, qrow + 1)
```

On Display

The event is invoked whan a row is displayed, and is called with the block and the query row number as arguments. An example of using this event was shown earlier.

Pre Insert

This event is invoked immediately before a row which has been inserted into a form is actually committed to the server database; the arguments are the block and the query row number of the row. If the event function returns false then the insert is aborted (but the row remains inserted in the form).

```
def eventFunc (block, qrow) :
    Orders.ordersPreCommit (block, qrow, RekallMain.actInsert)
    return 1
```

Note that the *Pre Insert* event is different to the *On Action* event when called with the *actInsert* action. The latter is when the user opens up a new row (by right-clicking in a rowmark and selecting *Insert)*; at this stage *Rekall* simply makes space for a new row of data to be entered.

Pre Update

This event is invoked immediately before a row which has been changed in a form is actually committed to the server database; the arguments are the block and the query row number of the row. If the event function returns false then the update is aborted (but the row remains changed in the form).

```
def eventFunc (block, qrow) :
    Orders.ordersPreCommit (block, qrow, RekallMain.actSave)
    return 1
```

Pre Delete

This event is invoked immediately before a row which has been marked as deleted in a form is actually deleted from the server database; the arguments are the block and the query row number of the row. If the event function returns false then the update is aborted (but the row remains marked as deleted in the form).

Note that if the block *autosync* option is set, then a row will be deleted immediately after it is marked for deletion.

```
def eventFunc (block, qrow) :
    if not RekallMain.queryBox ("Are you sure?") :
        return 0
    Orders.ordersPreCommit (block, qrow, RekallMain.actDelete)
    return 1
```

Post Query

The *Post Query* event is triggered immediately after an SQL *select* query has been executed but before the data is displayed. The single argument is the block; any value returned is ignored.

The example below updates a combobox control to have on entry for each record retrieved from the server database See the *Client* form in the *RekallDemo* database.

```
def eventFunc (block) :
    selector = block.getNamedCtrl("selector")
    list = []
    for rowno in (range(block.getNumRows())) :
        list.append (block.getRowValue("Company", rowno))
    selector.setValues (list)
```

Post Sync

This event is invoked immediately after the data displayed in a form has been synchronized with the server database (ie., after an SQL insert, update, or delete). The arguments are the block, the current query row number, the synchronisation operation just performed, and the primary key value for the row which was operated on. Values for the operation are defined in the *RekallMain* module, and are listed above under the *On Action* event.

The example below is just the same as the *Port Query* example above, and keeps the combobox up to date when records are added, deleted or altered.

```
def eventFunc (block, qrow, act, value) :
    selector = block.getNamedCtrl("selector")
    list = []
    for rowno in (range(block.getNumRows())) :
        list.append (block.getRowValue("Company", rowno))
    selector.setValues (list)
```

Form Events

On Load

This event is executed when a form is loaded for execution. At the point of execution, the form is ready for display, but no server database queries have been issued. The single argument is the form,

```
def eventFunc (form) :
    RekallMain.messageBox ("Hello and Welcome") ;
```

On UnLoad

This event is executed immediately before a form closes. The single argument is the form.

```
def eventFunc (form) :
    RekallMain.messageBox ("Toodle-pip, old chap!") ;
```

On Close

This event is executed if a form is about to be closed. If there is an event function and the function returns zero then the form is not closed. Beware: if this routine *always* returns zero then the form cannot be closed except by switching to design view [43]

```
def eventFunc (form) :
    return RekallMain.queryBox ("Really close?") ;
```

Manipulating Objects

The chapter so far has described where scripts are stored, and when scripts are executed. We now move on to describing how scripts can manipulate objects in forms and reports.

In line with the object orientation of Python as a language, all *Rekall* objects - *KBForm*, *KBBlock*, etc. - are represented as Python objects. Hence, when an event function is invoked with is associated object as the first argument to the event function, that first argument is a Python object which represents the *Rekall* object. And, just as Python provides object inheritance, the Python objects which represent *Rekall* objects have an exactly corresponding inheritance. Hence, since a *KBField* object is a special case of a *KBItem*, so the *KBField* Python

object inherits all the methods applicable to the *KBItem* Python object.

The remainder of this section should be read with the *Rekall* object structure described earler. Each of the following sections lists the methods applicable to each *Rekall* object.

KBObject Methods

The following methods apply to *KBObject*s. Note that some methods, for instance the enable and visibility methods, are redefined for *KBItem*s since an *KBItem* may display multiple controls.

setEnabled(bool)

This method enables or disables the object, in the normal sense in which buttons and such are enabled or disabled.

The code below, which is a *KBBlock onCurrent* event, disables a button for the first record.

```
def eventFunc (block, qrow) :
    prevButton = block.getNamedCtrl("PrevButton")
    prevButton.setEnabled (qrow > 0)
```

isEnabled()

This method returns true if the control is enabled.

setVisible(bool)

This method shows or hides the object, in the normal sense in which buttons and such are shown or hidden.

```
def eventFunc (block, qrow) :
    prevButton = block.getNamedCtrl("PrevButton")
    prevButton.setVisible (qrow > 0)
```

isVisible()

This method returns true if the object is visible.

getName()

This method returns the name of the object, as set in the object's properties.

getAttr(attrName)

This method returns the value of a named property (attribute) of the object. The name is a name as specified in the previous chapter (eg., the *X-position* property is named *x*). *getName()* is actually equivalent to *getAttr("name")*.

width()

Returns the object's width in pixels.

height()

Returns the object's height in pixels.

resize(width, height)

This method resizes the object to *width x height* pixels. Note that if the object is a *KBItem* then all the controls displayed by the *KBItem* will be resized.

getBlock()

Returns the *KBBlock* in which the object is embedded. Note that if the embedding block is actually a *KBForm* or *KBReport* then the result is a *KBForm* or *KBReport* respectively.

getNamedCtrl(name, errorOK)

This method is the key to locating controls; given the name of a control it locates the control relative to the object on which it is invoked. The most common usage is to locate a control inside a block; the example below could be used on a block *On Display* event to clear a field.

```
def eventFunc (block, qrow) :
    ctrl = block.getNamedCtrl ("Password")
    control.setValue (qrow, "")
```

However, the *name* argument to *getNamedCtrl* can be an arbitrary path, with the */* character as separator, in which case the object tree is traversed. For instance, *getNamedCtrl("block1/control12")* would locate an object named *block1* inside the object to which the method is applied, and then locate *control12* within that object. When used this way, each step other than the last should return a block or container object.

In addition, you can use .. to move up the object tree. For instance, the following event code could be associated with a button *On Click* event, in which case it will disable the button named *another* in the same block (or container) as this button:

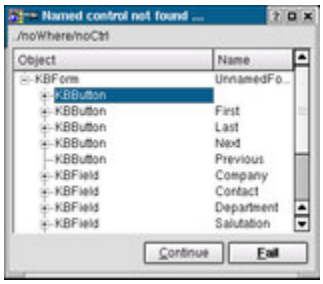
```
def eventFunc (button) :
    button.getNamedCtrl("../another").setEnabled(0)
```

If the name starts with the */* character then rather than starting at the object to which the method is applied, the location operation will start with the topmost object, that is, the form or block. But, beware, if you use a name like *block2//control34* then the *//* will go to the topmost object; although the use of .. and a leading */* is analagous to file system names, the *//* usage differs.

To make debugging easier, if the second argument is false (this argument is optional and defaults to false), and the location operation fails at some point, then a dialog is shown. This shows a tree of all objects in the form or report, along with the *name* argument. The object tree is expanded as far as the object on which the *getNamedCtrl* method was invoked. You can then either fail the operation (the method returns with the result *None* or select an object and continue with that object being returned.

The screenshot below shows the dialog, and the following code (which can be attached to a button *On Click* event) will display the name of the selected object.

```
def eventFunc (button) :
    ctrl = button.getNamedCtrl ("../noWhere/noCtrl")
    if ctrl != None :
        RekallMain.messageBox (ctrl.getName())
```



getForm()

Returns the *KBForm* in which the object is embedded, or *None* if the object is actually in a *KBReport*. Note that this is distinct from *getBlock()* which will only return a *KBForm* if the object is embedded in the top-most *KBBlock* of a form.

lastError()

This method returns a string describing the last error which occured on the object. It can be used after specific methods (such as the *KBForm executeCopier* method) which set an error message.

KBItem Methods

The following methods apply to *KBItem*s. Note that the *row* argument identifies a query row number, that is it is a row index into the data which is displayed in the block in which the *KBItem* is embedded.

setValue(row,value)

This method sets the data control which currently displays the *row* query row to the specified value. For instance, if a block is displaying 5 rows of data, which are the 11th through 15th rows of the query, then *setValue(12,"Hello")* will set the second displayed row.

If the specified row is not currently displayed, then nothing is updated.

Please note that in this release of *Rekall*, the value must be a string.

```
def eventFunc (block, qrow) :
    qty  = block.getNamedCtrl("Quantity").getValue(qrow) ;
    cost = block.getNamedCtrl("Cost"    ).getValue(qrow) ;
    block.getNamedCtrl("Amount").setValue(qrow, `int(qty) * int(cost)`)
```

getValue(row)

This method retrieves the value currently displayed in the control corresponding to the *row* query row. For instance, if a block is displaying 5 rows of data, which are the 11th through 15th rows of the query, then *getValue(12)* will get the value from the second displayed row.

If the specified row is not currently displayed, then the result null.

setEnabled(row,bool)

This method enables or disables the control corresponding to the *row* query row, in the normal sense in which text fields and such are enabled or disabled.

The code below, which is a *KBBlock onDisplay* event, disables a salary field if it contains the boss's salary, so that the wages department cannot change it. Ha! typical.

```
def eventFunc (block, qrow) :
    minion = block.getNamedCtrl("Name").getValue(qrow) != "TheBoss"
    block.getNamedCtrl("Salary").setEnabled (qrow, minion)
    if not minion :
        RekallMain.messageBox ("The Boss's salary is fixed!")
```

isEnabled(row)

This method returns true if the control corresponding to the *row* query row is enabled.

setVisible(row,bool)

This method shows or hides the control corresponding to the *row* query row, in the normal sense in which buttons and such are shown or hidden.

isVisible(row)

This method returns true if the control corresponding to the *row* query row is visible.

Containers Methods

The following methods apply to *KBBlock*s and *KBContainer*s. Note that when used on a *KBContainer*, the method in effect operates on the *KBBlock* in which the *KBContainer* is embedded. As for *KBItem*s, the *row* argument identifies a query row number.

getNumRows()

This method returns the number of rows or data which the block as retrieved from the server database. For instance, if the block gets data directly from a table, and there were no SQL *where* conditions, then the value will be equal to the number of rows in the table.

The following example, an event function for the *KBBlock postSync* event, totals up and displays stock quantity.

```
def eventFunc (block) :
    total = 0
    for row in range (0, block.getNumRows()) :
        value = block.getRowValue("Stock", row)
        if value != None : total = total + int(value)
    RekallMain.messageBox \
    (   'There are " + `total` + " items in total",
        Total Quantity of All Products"
    )
```

getQueryRow()

This method returns the current query row number.

gotoQueryRow(row)

Focus is moved to a control which is displaying data from the specified query row. If necessary, the block will scroll through its data to bring such a row into view.

getRowValue(name, row)

Name should be the name of a data control which is embedded in the block. Provided that such a control exists, then the result is the data value from the *row* query row corresponding to the control.

Note that this is not neccessarily the value displayed. Either the specified row may not be displayed at all (ie., it is outside the range of rows currently displayed by the block), or the user may have edited the value displayed but not yet saved it. If it is neccessary to ensure that the displayed value is kept correct then the script must also update the control.

See the example above under the *getNumRows()* method.

KBButton Methods

setText(text)

This method sets the button text. For example, to change the button text when a button is clicked, using the *onClick* button event:

```
def eventFunc (button) :
    button.setText ("Button Clicked")
```

KBLabel Methods

setText(text)

This method sets the label text.

Tabber and Tabber Page Methods

The following two methods apply to tabber pages, although they can also affect the tabber in which the page exists.

setEnabled(bool)

This method can be used to enable or disable a page. If the page is disabled then all control inside the page are also disabled. The tab which is associated with the page is also enabled or disabled.

setCurrent()

This method makes the page to which it is applied, that is, it is made visible (and hides all other pages), and the associated tab becomes the current tab. This is equivalent to the user clicking the tab.

Note that this method acts independantly of the *setEnabled* method, so a page can be made current even if it is not enabled.

KBForm Methods

openForm(name, params)

This method can be used to open a named form. The first argument is the name of a form; the second (which is optional) should be a dictionary of (name, value) pairs, which are passed as parameters to the form. See the chapter on executing forms and reports with paramaters for more details.

The example below is attached to the *Clients* button of *MainForm* in the *RekallDemo* database. It prompts the user for a filter to select clients to display. See chapter 6 for more details.

```
def eventFunc (button) :
    text = RekallMain.promptBox \
    (   "Enter pattern or leave empty for all",
        "",
        "Select companies"
    )
    if text == None : return
    if text != "" : text = "Company like '" + text + "'"
    button.getForm().openForm
    (   'Client',
        {'Filter' : text}
    )
```

There are three possible return conditions from *openForm*. If there was an error, then the error will be displayed before the call returns, and the result will be zero. If the form was closed normally, then the results is also zero.

The third case arrises if the form was executed modally, and was closed via the *form.closeForm(rc)* (see blow) method, with a non-zero argument. In this case the return value from *openForm* is a *python* dictionary, where the keys are the names of the data controls in the form (actually, the path name from the top of the form down to the control; so, if a control named *Name* is embedded in a block named *Person*, then the key will be *Person.Name*), and the values are the values shown in the controls when the form was closed. This allows the use of modal forms as modal dialog boxes.

openReport(name, params)

This method can be used to open a named report. The first argument is the name of a report; the second (which is optional) should be a dictionary of (name, value) pairs, which are passed as parameters to the report. See the chapter on executing forms and reports with paramaters for more details.

executeCopier(name)

This method can be used to execute a named copied, the name of which is passed as the argument. The return value is the number of rows copied, or negative on an error (in which case the *lastError* method can be used to get an error message).

```
def eventFunc (button) :
    form = button.getForm()
    rows = form.executeCopier('ClientsAsXML')
    if rows < 0 :
           RekallMain.messageBox (form.lastError())
    else : RekallMain.messageBox ('Exported ' + `rows` + ' client records to "/tmp/clients.xm
```

openTextForm(text, params)

This method opens a form whose XML definition is passed as the first argument.

See the example earlier in this chapter.

openTextReport(text, params)

This method opens a report whose XML definition is passed as the first argument.

closeForm(rc)

Calling this method closes the form. The argument should be a number. See the *openForm* method above for a description of the use of this value.

Python Scripting Help

Rekall provides automatic prompting when you are editing a *python* script. For instance, if you enter something like:

```
button.setEnabled (true)
```

Immediately after entering the opening parenthesis, *Rekall* will show summary information for possible *setEnabled* method; in this case there are two, for an *KBObject* (such as a button) and an *KBItem* (such as a text field). Because *python* is a dynamically typed language, *Rekall* cannot make any assumptions about what sort of object *button* in the above example is [44] and hence shows both methods

The help information remains visible until you either type a closing parenthesis, or move to another line in the script. It can also be dismissed using the *escape* key.

An additional way of getting help is to enter *ctrl-H*. In this case *Rekall* looks for a method name immediately in front of the current cursor position, and shows all methods which start with the string so found. For instance. with the cursor positioned immediately after *button.set* in the above example, *Rekall* will show all methods which start with *set* (*setEnabled*, *setValue*, and so forth).

---

[40] Actually, this would be pretty stupid if there were lots of records, but if there are only a few, its quite good. Anyway, this is just an example, not a statement of good database design.

[41] At some stage, the # character may be changed, for instance to !, to avoid this problem. In truth, # was a bad choice in the first place. I admit it.

[42] Actually, *python* import can be extended, so some thought may be given to direct import from *Rekall* in a later release.

[43] And even this is not possible in the runtime version of *Rekall*, since all design functionality is removed. You have been warned!

[44] You may call the object variable *button*, but that is entirely up to you; it has no special meaning to *python*.

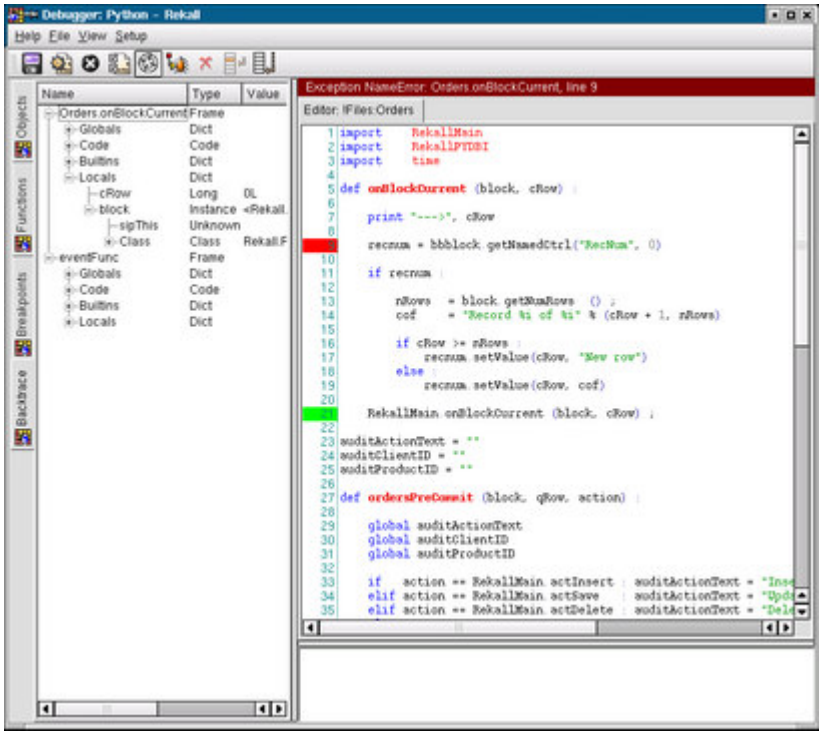
Chapter 10. The Python Debugger

Table of Contents

Rekall has a built in Python debugger. This is still under development, but currently it does the standard sort of debugger things, like trapping errors, handling breakpoints, and displaying variables. The debugger window is

brought up using the *Show Debugger* button on the main database window toolbar (the one with the bug on it!)

The screenshot below shows the debugger in action. The left-hand side contains a tabbed area, with tabs which display (from top to bottom) a view onto all python objects (which starts with the module dictionary); a view of all python functions (grouped up by module); current break and watchpoints; and a backtrace. The right-hand side shows the code of a module; the green marker shows the point at which a breakpoint or error occured, while the read markers denote breakpoints. In this case, the code has an error; *block* has been mistyped as *blocK* so the Python interpeter has been unable to locate a *getNumRows()* method.



Usually, the debugger is just a normal window, just like a form or copier window, which the exception that the debugger is always a separate top-level window, whether *Rekall* is running in SDI mode or MDI mode. However, if a *python* exception is raised or a breakpoint is hit, the window becomes modal. Effectively, this freezes *Rekall* at that point until *python* execution is continued. While you can edit and compile code at this point, you cannot continue execution with the modified code [45]

Also, you cannot directly load code into the debugger edit window. Code automatically appears if an exception occurs and *Rekall* can determin its location. Otherwise, once the debugger window is on view, go to the list of scripts in the main database window, and right-click on a script; the popup menu will now include an entry to load the script into the debugger. Alternatively, once a script module has been loaded into the Python interpreter, locate something (such as a function) which is in that module in one of the left-hand windows, and right-click; the popup menu will have an option to dusplay the code.

Tool Icon	Use
	Set exception skip list
	Abort execution (raises an exception)
	Single step execution
	Continue execution
	Toggle breakpoint at current line
	Enable/disable exception trapping

Breakpoints

Breakpoints can be set and cleared by clicking in the required line, and then clicking the breakpoint tool. Alternatively, use any of the left-hand windows to find a function, and right-click; the popup menu will contain options to set a breakpoint (execution freezes when a breakpoint is hit) or set a watchpoint (watchpoints simply count the number of times thay have been hit).

Once a breakpoint is encountered, you can continue execution (execution continues until either another breakpoint is encountered, an exception is trapped, or control exits from the *python* script, or you can single step, in which case execution continues to until control arrives on a new line. Strictly, *new line* means when the *python* interpreter reports that control as reached a new line. In the case of a statement like *while i < 10 : i = i + j*, the same line will be repeatedly executed while *i* is less than ten, so a breakpoint on this line may be trapped several times in succession.

You can also abort execution after a breakpoint. Actually, this raises a *python* exception, so if the *python* script catches the exception, execution will continue from that point.

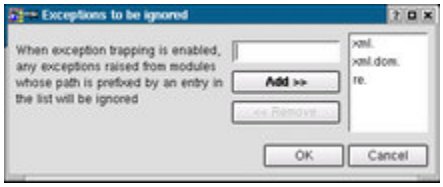
Exceptions

The debugger has two exception related controls. Firstly, you can enable or disable exception trapping. Some *python* modules generate a large number of exceptions in the normal course of execution (the *4Suite XML* library is a case in point), which means that debugging any code which makes use of such libraries can be near-impossible if exceptions are enabled.

On the other hand, it may well be the case that exceptions which occur in a *Rekall python* script really do represent errors. The screenshot above showed an example of this. The name *bblock* is a mistyping of *block*, and when the code is executed *python* raises an error because no variable called *bblock* exists. If this exception is not caught, then *Rekall* will detect the error (since control exits from an event function due to the exception) but is limited in what information it can report. Also, you would probably like the debugger to halt execution at the offending line.

The *exception skip list* provides a means to handle this situation. The skip list consists of a list of exception name prefixs; if an exception occurs but one of the entries in the skip list is a prefix of the exception name (hence *xml.dom.* is a prefix of *xml.dom.scopeError*, but not of *xml.sax.scopeError*) then the debugger does not trap the exception, and *python* processes the exception as normal.

The exception skip list dialog is shown below. The second entry, *xml.dom.* is actually superfluous, since *xml.* will includes all these. The *re.* prefix skips exceptions from the *re* package.



---

[45] Unlike, for instance Access Basic. This is just not possible with *python*; *python* is a much more powerful language than Access Basic, but nothing comes for free.

Chapter 11. Executing SQL from Python Scripts

Table of Contents

Although *Rekall* automatically access the server database when it gets data for a form or report, or when the user makes changes to data in a form, *Rekall* also allows direct access to the server database. It does this by providing a *python* class *RekallPYDBI* which is a (partial) implementation of the *python DBI2* specification.

Essentially, *RekallPYDBI* allows you to write and execute SQL queries that interact directly with the server database [46] . Currently, just the *select*, *insert*, *update* and *delete* SQL commands are supported (so you cannot, for instance, create or drop a table).

Connecting to the server database

The first stage in using *RekallPYDBI* is to connect to the server database. This is shown in the example below, and uses the *connect* function. The first argument can be any object in the form or report. For instance, if the code is in the *onClick* event function of a button then use the button as the argument [47] . The second argument is the server name used to identify the server database.

```
import RekallPYDBI
connection = RekallPYDBI.connect (object, server)
```

You now have a connection to the same server database. The next step is to create a *cursor*, which is can object which can execute SQL queries.

```
import RekallPYDBI
connection = RekallPYDBI.connect (object, server)
cursor = connection.cursor ()
```

Using a cursor

The most common use of a cursor is probably to execute an SQL *select* command. The example below (which can be attached to the *onLoad* event of a form), displays the number of products in the *Products* table.

```
def eventFunc (form) :
    connect = RekallPYDBI.connect (form, "Orders")
    cursor  = connect.cursor ()
    cursor.execute ("select count(*) from Products", [])
    RekallMain.messageBox("You have " + cursor.fetchone()[0] + " products")
```

The *cursor.execute(....)* line executes the SQL count query. The second argument is a list of values which will be substituted into the query (in this case, there aren't any); see the next example below.

The extended example below is taken from the *RekallDemo* database. The orders form has some scripting which records changes to the *Orders* table in another table called *Audit*. Basically, the *preInsert*, *preUpdate* and *preDelete* block events record infomation about what is about to happen in some *python* global variables. The *postSync* event then invokes the code shown below, which inserts a record into *Audit*.

The main thing to note is the use of *?* as a *placeholder* in the SQL query; when the queery is executed, the values in

the list argument to *cursor.execute(...)* are substituted. This example also shows the use of python exceptions.

```
def onBlockOrdersPostSync (block, qRow, action, key) :

    global auditActionText
    global auditClientID
    global auditProductID

    if auditActionText == None : return

    entered = time.strftime("%Y-%m-%d %T", time.localtime (time.time()))

    try :
        connect = RekallPYDBI.connect (block, "Orders")
        cursor = connect.cursor ()

        cursor.execute \
        (   "insert into Audit (OrderID, ClientID, ProductID, " + \
            "                   Action, Entered) values (?, ?, ?, ?, ?)",
            [   key,
                auditClientID,
                auditProductID,
                auditActionText,
                entered
            ]
        )

    except RekallPYDBI.DatabaseError, message :
        RekallMain.messageBox (message.args[0])

    auditActionText = None
```

The RekallPYDBI Code

The code used to implement *RekallPYDBI* can be found in */opt/kde3.1/share/apps/rekall/script/py/RekallPYDBI.py*. At present, this is a partial implementation of the *python DBI2* specification, but will be filled out in a future release.

Please note *RekallPYDBI* uses a lower level interface to *Rekall* itself (methods such as *qrySelect* and *getNumFields*). These may change in future releases of *Rekall*, so you should not use them yourself.

---

[46] You can, of course, import and other *python DBI* module and use that to access a server database, however you will need to code in details such as the username and password for the server database.

[47] The first argument is needed in case you have opened multiple database in the same instance of *Rekall*.

Chapter 12. Import and Export: The Copier

Table of Contents

This chapter describes *Rekall*'s import and export functionality. However, *Rekall* extends this to a more general copy mechanism; read on!

The Copier

Database front ends generally provide a means to import data and to export data. Import basically takes data from a file and loads it into a table; export basically takes data from a table and writes it out to a file. So, import can be thought of as copying from a file to a table, and export as copying from a table to a file; the important operation here is *copy*. It is the copying operation that *Rekall* extends to provide import and export.

A copy operation has two main components, namely a *source* and a *destination*. *Rekall* provides three sources and three destinations. Logically, the sources take an input, split it into rows, and splits each row into fields. They are:

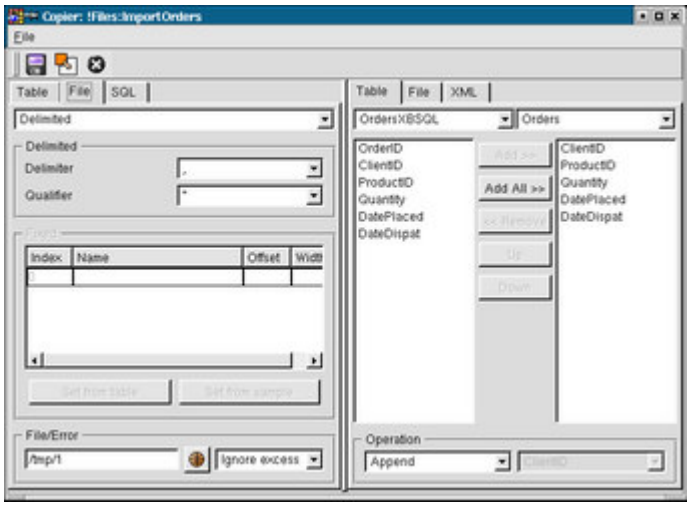
- A file (either fixed width or delimited fields)
- A table
- An arbitrary SQL select statement

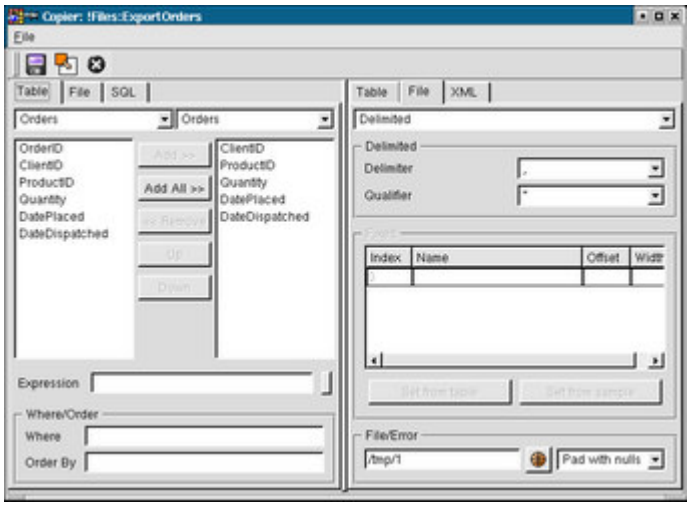
Similarly, destinations logically take rows of fields, and combine them row-by-row to generate output. They are:

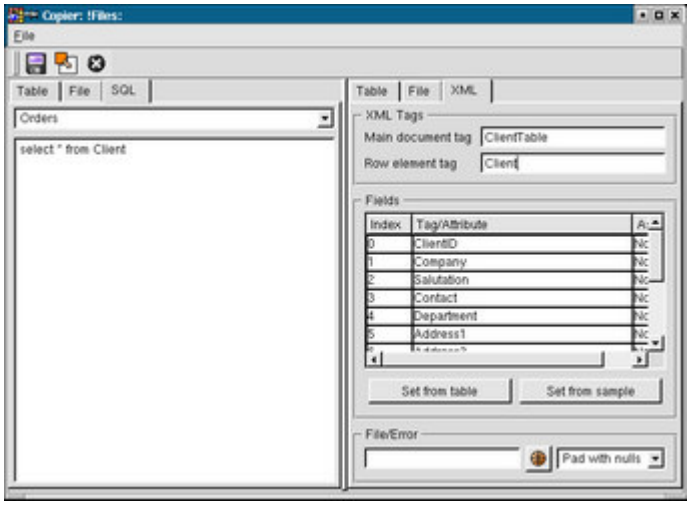
- A file (either fixed width or delimited fields)
- A table
- An XML format file

A *copier* is then a specified source and a specified destination, and data is simply copied from one to the other, effectively row by row. There are a few extra considerations, such as what to do if the source produces more or less items than the destination expects, but this is the basic operation.

The screenshots below show the copies; the first copies a file to a table (a classic *import*), the second copies a table to a file (a classic *export*) and the third copies from an SQL select query to an XML formatted file. In each casem the source is the left-hand side, and the destination the right-hand.







Copier Sources

File

A source file may be contain either fixed width or delimited; which to use is selected from the combobox at the top.

A delimited file has a delimiter, and optionally a qualifier; the delimiter separates input fields, the qualified surrounds fields. Hence, a line like *one|two|three* has the | character as delimiter bit no qualifier; *"one"|"two"|"three"* has the same delimiter but also " as the qualifier.

In a fixed with file, fields occupy sepecied columns (eg., field1 occupies columns 0 through 12, field2 occupies 13 through 20 and so on). Fields are specified in the appropriate area of the dialog; each can be given a name, which is just used as a comment. Fields do not have to occupy contiguous ranges of columns, and need not include all the columns in the file (indeed, fields may overlap, though you will be warned about his). The *Set from table* button can be used to choose a server database and table on which to base the set of fields.

The other setting (at the bottom, next to the file) controls error behaviour. *Ignore excess* means that extraneous fields are ignored (this only applies to delimited files); *Skip* means that any line with too few or too many fields is skipped; and *Abort* causes the copy to be aborted if there are too many or too few fields.

Table

A table source specifies a server database, a table in that server database and one or more fields from the table; additionally, arbitrary SQL expressions can be added.

Optionally, SQL *where* and *order by* expressions can be specified, to select only certain rows, and to order the rows.

Arbitrary SQL

The allows an arbitrary SQL *select* query to be given; the only other setting being the server database.

Copier Destinations

File

The settings for a file destination are pretty well the same as for a file as source. The only difference is the error options, where *Ignore excess* is replaced by *Pad with nulls*, whereby if the source does not provide enough values, then nulls (empty strings) are used.

Table

A table as a destination has similar settings as a table as source. However, there are a set of options that control how rows are imported:

- *Append*: Rows are simply appended to the table.
- *Replace*: All existing records in the table are deleted before any new rows are added.
- *Update*: Existing rows are updated where the value of a particular column (which is set in the dialog) matches the import row. If there is no match then the import row is ignored. [48]
- *Replace/Insert*: This is the same as *Replace*, except that if no rows are updated then the import row is appended to the table.

XML

XML destination writes ouput to a file, but in XML format.

The root document element is named according to the *Main document tag* setting; each row is then a child element of the root element, and is named as the *Row element tag* settings.

Values are ouput either as attributes of the row elements, or as text in vakue elements which are children on the row elements. The attribute name or value element names respectively are set in the *Fields* area of the dialog, along with the choice of attribute or element.

The *Set from table* button can be used to choose a server database and table on which to base the set of fields.

---

[48] *Rekall* simply generates an SQL *update* statement and executes for the values in the import row. Hence, the question of whether zero, one or more rows in the table are changed is just a function of the server database

Chapter 13. Executing Forms and Report with Parameters

Table of Contents

It is sometimes neccessary to run a form or a report with one or more parameters. For instance, it may be useful to be able to run a report which only output information between a pair of dates.

Rekall's mechanism for this is *parameters*. Parameters may be embedded inside object attributes (for instance, inside the *where* attribute associated with a table or query), and the user can be prompted for values when the form or report is run.

Values for parameters can also be supplied when a form or report is run. This does not occur when the user directly runs a form or report (by double-clicking in forms tab of the database dialog), but can be used when a form or report is started by a script.

Using Parameters

A parameter can be inserted into any text attribute. For example, the following is an *where* expression which accept records whose field *EntryDate* is between two dates:

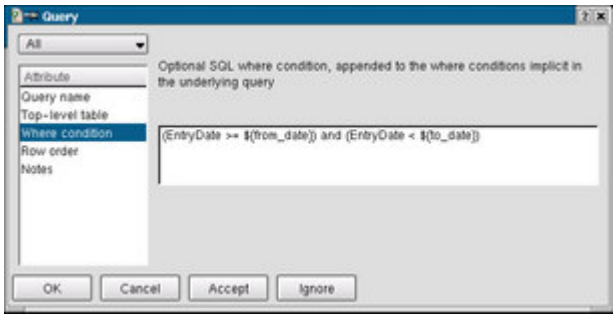
```
(EntryDate >= ${from_date}) and (EntryDate < ${to_date})
```

The text *${from_date}* will be replaced by whatever the value of *from_date* is. The value is supplied either by the user via a parameter dialog, or via a script. Additionally, the form *${name:defval}* may be used; in this case, if *name* is not otherwise defined, then *defval* will be used. This is useful in *where* expressions like:

```
Company like '${Company:%}'
```

In this case, a pattern can be supplied which filters by company name; if no pattern is supplied then % matches all companies.

An example in the dialog below, which is accessed by going to *Block Properties* and clicking the *Query* button.

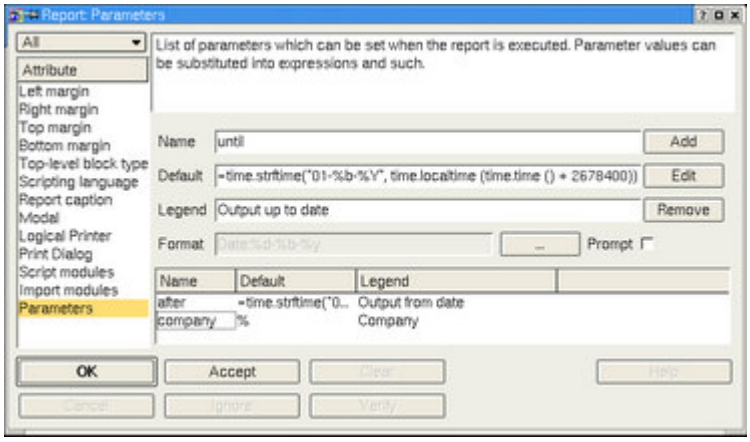


Setting up for User Entry

The parameter dialog mechanism provides a way to prompt the user for parameter values, and to supply default values.

First, it is neccessary to set up a set of parameter prompts. There should be one for each parameter that is used in the form or report, each containing the parameter name (as in ${*name*}), the default value, and some prompt text. To set these up, open the document in design view, and go to the document properties dialog; here, locate and double-click the paramaters item.

This will show the properties dialog in the form shown below; the screenshot shows one parameter *from_date* already set up, and the other *to_date* being edited. Also shown here is the use of scripting to provide a default range of dates from the first of the current month to the first of the next month [49] ; the usage is just the same as for using an expression as the value of a field in a report or a form. Note that for this example to work, the report must *import* the *python time* module.



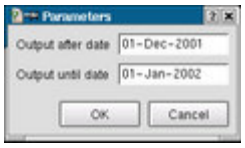
There are two further settings associated with each paramater. The first, *Format* serves two purposes, to verify the entered parameter, and to format the entered value. Formats are set using the button to the right of the format field, which brings up the same dialog as is used when setting formats on (for examples) fields in forms and reports. In this screenshot, the format is for a date as *dd-mmm-yy* (like *12-Apr-03*).

The other setting is *Prompt*. This is only relevant when the form or report is opened using the script function *openForm()* (or *openReport()*), as described below. by default, if a value for the paramater is passed via the *openForm()* function, then that parameter is not shown in the dialog to the user; however, if *Prompt* is set, then the user is always prompted for the paramater (and the dialog shows the value passed via *openForm()* rather than the default set in the paramater dialog).

User Input

Finally, when the form or report is executed, a parameter dialog will appear into which the user can enter values:

[50]



Passing Parameters via Scripts

The *KBForm openForm* and *openReport* methods take an optional argument which is a dictionary of parameter (name, value) pairs, for instance [51]

```
def eventFunc (button) :
    button.getBlock().getForm().openReport \
    (    "Clients",
         { Company : "Big%" }
    )
```

If the *Clients* report has not defined any user entry paramaters, then it will execute immediately. Assuming that you have done something sensible with the *Company* parameter such as having a *where* expression like *Company like '${Company:%}'*, then the report should just show big companies.

This mechanism can be taken to extremes. For instance, you could have a *where* expression which is just *${Filter}*, and then pass the *Filter* parameter as *Company like 'Big%'*. This way, you can construct completely arbitrary filters in a script. Below is an example from the *RekallDemo* database:

```
# This is comment and not a shortcut, since the leading # is followed
# by whitespace. OK, agreed, its a hack!
#
def eventFunc (button) :
    text = RekallMain.promptBox \
           (    "Enter pattern or leave empty for all",
                "",
                "Select companies"
           )
    if text == None : return
    if text != "" : text = "Company like '" + text + "'"
    button.getBlock().getForm().openForm ('Client', {'Filter' : text})
```

For reference, here is the *KBForm onLoad* event for the *Client* form. This displays a subtlety of inline events. Because the text is just a property like any other property (width, font, etc.), they are subject to parameter substitution, so the script here displays a welcome message, which also shows the filter, if there is one.

```
def eventFunc (form) :
    message = "Welcome to the Clients Form!"
    if "${Filter}" != "" :
        message = message + "\nFilter: " + "${Filter}"
    RekallMain.messageBox(message)
```

Opening Forms and Reports

By default, the *openForm* method will open a form in data view, and *openReport* will open a report such that it is printed. These are most likely what is usually wanted, however *Rekall* does have a mechanism to open forms and

reports differently to this, for example to open a report in data view.

This is accomplished by passing a value for a parameter *__showAs* (thats two underscore characters). The list of possible values is given below, note that not all cases apply to both forms and reports.

ShowAsData	From or report is opened in data view. This is the default for forms.
ShowAsDesign	From or report is opened in design view
ShowAsReport	Opens a report for printing. This is the default for reports.

By the way, these also work for *openTextForm* and *openTextReport*, though its doubtful that it is much use in these cases.

Parameter Passing: An End-Note

Parameter substitution takes place on all *properties* of all objects that a form or report is constructed from. In principal, there is no reason why something like a form width cannot be parameterised.

However, at present it is effectively impossible to parameterise any property that is actually used in design mode (such as form width). If you really do want to play this trick, you should be able to do so, but you will need to hand-edit the XML form or report definition (and it will be lost if you make further changes via the form or report designer).

To be honest, we have not actually tried this, but it ought to work :)

---

[49] Actually, the code isn't quite right. It gets the first of the next month by adding 31 days (in seconds) to the current time, which might of course be the month *after* next. Completely correct code is left as an exercise for the reader.

[50] There is currently a bug in *Rekall*; if you switch back and forth between design view and data view, then although the user entry parameter dialog will appear, the values may not be used. If in doubt, close the form or report, and then open it in data view.

[51] Hopefully, in a future release, it will be possible to support *python* argument passing like *openReport("Clients", Company = "Big%")*.

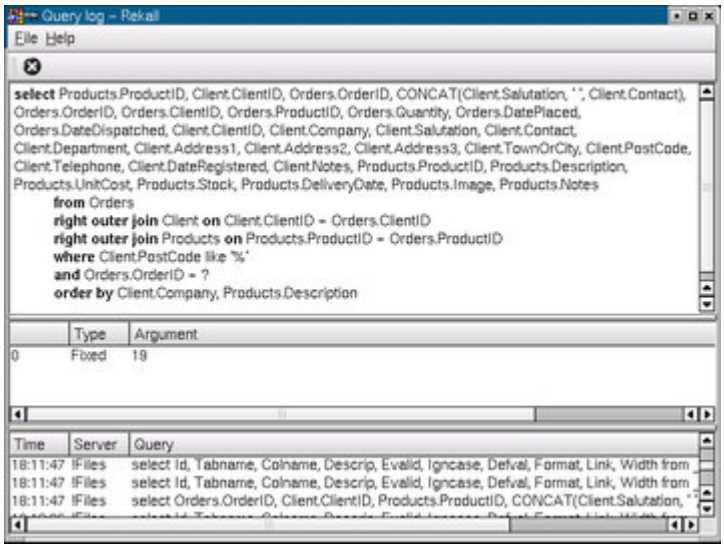
Chapter 14. Wherin Be Diverfe And Afforted Mifcellanea

Table of Contents

This chapter contains a miscellaneous collection of things which do not have, or have yet to find, a home elsewhere in this manual.

Query Logger

The query logger window is opened from the main database window, either via the *View* menu or using the appropriate tool on the tool bar (the sheet of paper overlaid with the letter *Q*). When displayed, it will record the most recently executed queries [52] and any arguments to those queries. This is shown below.



The window is split into three sections. The lower part shows the list of queries; for each is shown the time at which the query was executed, which server database it was associated with, and the text of the query. Note that the query may contain embedded *place holders*, where arguments are substituted; the character or characters used for the *place holder* will depend on the server database, but for *MySQL*, *PostgreSQL* and *XBase* will be question marks (*?*).

Double-clicking on a query will display it in the upper part of the window, with any arguments (that is, values which are used to substitute for *place holders* appearing in the middle part. Where *Rekall* can parse the query, it will be displayed suitably formatted. If it cannot be parsed then it will be displayed in plain text (and, mainly for debugging purposes, an error message will also appear). Note that in this release of *Rekall*, only *select* queries are parsed.

Event Logger

The event logger window is opened from the main database window, either via the *View* menu or using the appropriate tool on the tool bar (the sheet of paper overlaid with the letter *Q*). This window is much like the query logger, but shows events and slots.

SSH Tunneling

SSH tunnelling is only available under the Linux builds of *Rekall*, and is currently supported for *MySQL* and *PostgreSQL*

SSH tunneling, in general, is a mechanism for making TCP/IP connections possible via an (secure and encrypted) SSH connection. Please refer to the *ssh* and *sshd* documentation for details. However, the basic mechanism is that, geven that you can establish an SSH login to a remote machine, then the local instance of *ssh* listens on a sepcified port number; any connection made to this port is routed via the SSH connection to the remove machine, whence a further connection is made to a specified port on a specified machine. Data is then passed between the local port and the latter port/machine. This allows, for instance, connections though firewalls where you have an SSH login to a machine inside the firewall, but do not have access through the firewall to some server.

To use SSH tunneling with *Rekall*, the *SSH Tunneling*> setting in the advanced properties dialog for a server database should be set to a string like *name@address:port*. Here, *address* is the name or address of the machine to which you have SSH access, *name* is your login name on that machine, and *port* is a port number on the local

machine (ie., the one running *Rekall*). Note that, since *Rekall* cannot prompt for an SSH password, you must be able to establish the SSH login without requiring a password (ie., by storing your public key for the local machine in the appropriate location on the server, again see the SSH documentation).

When *Rekall* opens a database connection, if first executes the *ssh* command, passing *name@address* as the address and user name for the connection. It specifies the port from the *name@address:port* string as the local port on which *ssh* should listen, and the *host* and *port* settings (from the database settings; here *port* defaults to the standard port for the server database) as the destination to which the remote machine should connect to the server database. The server database connection is then made via the local port.

Suppose that the *SSH tunneling* setting is *mike@thekompany.com:3001*, and that the *host* and *port* settings are *www.microsoft.com* and *5432* respectively. Then, the *ssh* command will connect to *mike@thekompany.com*, and will listen on port *3001*. The server database connection is then pade to port *3001* on the local machine; the server end of the SSH connection, on *thekompany.com* will then connect to port *5432* on *www.microsoft.com* [53] (and will most likely fail, this *5432* is the default *PostgreSQL* port, and we can assume that *PostgreSQL* will not be running there:)

Please note the SSH command (or the *sshd* server process on the remote maching) may close the connection if there is no traffic for some period of time. You should consult your *ssh* and *sshd* documentation for timeout-related settings.

---

[52] Currently the most recent 64 queries, and up to 8 arguments for each. In addition, arguments are truncated to 64 characters.

[53] Specifically, the *ssh* command will be *ssh -C -N -L 3001:www.microsoft.com:5432 mike@thekomnpany.com*. The *-C* connection means that data is compressed, and *-N* that no shell is actually run on *thekompany.com*.

Chapter 15. Components and Event/Slot Scripting

Table of Contents

Earlier, this manual described how to incorporate preconfigured components, such as buttons and navigation tools, into forms and reports. These components were taken from those that are distributed as part of *Rekall*. However, *Rekall* also provides the means to construct your own components, which you can then use as often as you need.

This chapter is divided into three main parts. The first part describes the basics of building your own components. This does not involve any scripting; although this limits the level of functionality that you can achieve, it can still be used to provide a useful set of components. If you have no intention of writing any scripts of your own, you can skip the second and third parts of this chapter.

The second part describes the *Event/Slot* mechanism that *Rekall* provides. Although the use of this mechanism is not restricted to reusable components (it can be used anywhere in a form or report), its major use is with reusable components, since it provides a way of bundling a set of objects which will be used in a form or report together

with the code that is needed to make use of them.

The third part then gives some examples of constructing resuable components which make use of the event/slot mechanism.

Before going any futher, it is worth describing where *Rekall* can store components. There are three places. Firstly, there is the stock component set that come with the *Rekall* distribution; secondly, components can be stored inside databases (just like forms and reports); and thirdly, they can be stored in a local directory (on Linux this is *$HOME/.rekall/components*. Normally, any components that you create yourself would be stored in a database or locally; storing in the database would be appropriate if you want to share them between all users of the database, while locally would be appropriate for you own personal components [54].

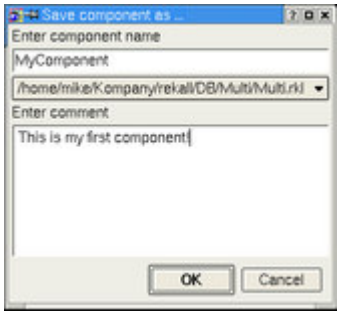
If you want you can store your components with the stock components, since stock components are just stored in files in a known directory, but (on Linux) you would normally need root access to do this, and they might be lost when upgrading - be warned.

Components: Rolling Your Own

There are two ways to go about rolling your own reusable components [55], either by saving one or more objects from an existing form or report, or by designing from scratch. Since creation of forms and reports has been described in earlier chapters, the former is described first.

Creating Components from Existing Forms and Reports

At its simplest, creating a component from a form or report is as simple as right-clicking on an object and selecting *Save as component*, or selecting several objects, and then using the *Save as component* entry on thr *Edit* menu. In either case, you will the dialog show below:



The top text box is used to enter a name for the component, while the combobox selects the place to store it. This will show each server database (including the *!Files* entry), plus *Save to file*. Selecting the latter can clicking *OK* will bring up a standard file dialog [56]. The comment area can be used to enter any comment, which will actually be saved as the *notes* propertly of the component. This text will appear under the *Description* tab of the component selection dialog that appears when you paste a component.

One thing to note is that all components are categorised as either *form* components or *report* components. A component that is created from a form is categorised as the former and can only be pasted into a form, and similarly for reports. This is necessary since, although some objects (such as labels) can appear in both forms and reports, there are many (possibly most) cases where creating a component from one and pasting it into the other would not be a good idea.

When you save an object as a component, the object is positioned in the component at position (10,10) and the component is sized to be just 10 pixels wider, and 10 pixels higher, than is strictly needed. If you save mutiple objects, then they maintain their relative positions, but the top-left hand one is similarly moved to (10,10), and the

component made (10,10) pixels larger. There is no particular reason for these values, other than for convenience if you later edit the component.

Once you have a created a component this way, then unless it was saved to a local file, it will appear under the *Components* tab of the main database window, just as do forms and reports. From there you can edit it further.

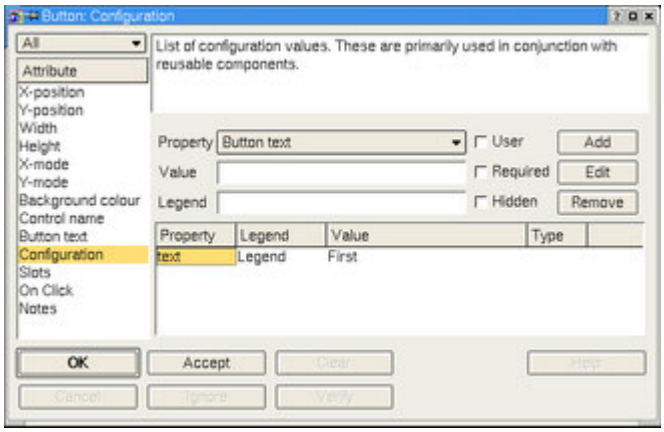
The advantage of creating a component from objects in an existing form or report is that they (presumably) already function in the way that you want. The one possible downside is that they might only function properly because thay are part of the form or report. Of course, it is perfectly possible to create components that will only work when used in a specific environment, but as a general principle, it is better to make components as independant of their environment as possible.

Creating and Editing Components

A completely new component can be created in much the same way as a form or report. Go to the *Components* tab of the main database window, and double click on the *Create new component* item. This brings up a dialog similar to that which appears when you create a new form or report (except rather simpler). The important setting is the type, either *Form component* or *Report component*. Note that this cannot be changed once set.

Once through the initial properties dialog, objects (labels, fields, etc.) can be added to the component almost exactly as thay can to a form a report. The most noticable difference is that a component never has an associated query. This means that if you add a data control, such as a *field*, then there are no table columns shown when you come to set the display expression. If you know that you will only use the component in a situation where (for instance) a column called *user_name* exists and is to be displayed, then you might set the display expression to *user_name*, but this rather limits the places you can use the component [57]. To help with this situation, however, you can make use of *configuration* properties.

When setting properties on objects, you may have noticed a propertly *Configuration*. This appears for all objects, but is mostly used in conjunction with components, where it provides a way of asking for and setting properties when the component is pasted. These are the settings that appear under the *Configure* tab of the component selection dialog. The screenshot below shows the properties dialog for the button which the stock first-record button provides, with the *Configuration* property selected. There is one configuration value, which provides the button text.



Normally, a configuration is associated with a property of the object, and the value in the configuration sets the value of the property when the component is pasted into a form or report. In the example above this provides the button text; the settings for this are:

Property	The property that this configuration value sets.	*text* is the property that specifies the text that the button displays.

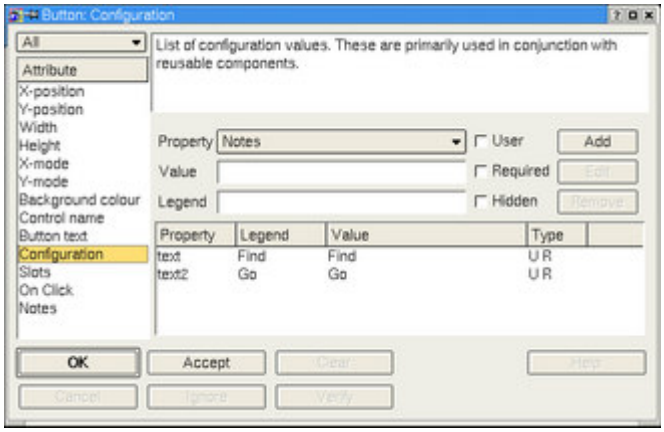
Legend	This is the text that appears under the *Configure* tab of the component selection dialog.	This should be set to some suitable prompt.
Value	Default value to be used.	The default text for the first-record button is *First*.
Type	Shows *User*, *Required* and *Hidden* settings; see below.	Shows *U,R* and/or *H* respectively.

New configurations can be added by selecting a property, specifying a value and a legend, and clicking the *Add* button; configurations can be edited by clicking the *Edit* button (or by double- clicking the required entry), making changes and then clicking *Add*.

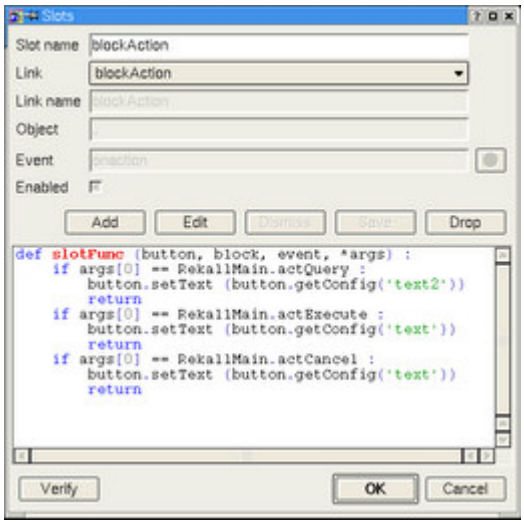
There are three additional settings for each configuration, *User*, *Required* and *Hidden*. When set, these show as *U*, *R* and *H* in the *Type* column. If *Required* is set, then a value must be specified when the component is selected and pasted into a form or report. If *Hidden* is then then the configuration does not appear under the *Configure* tab of the component selection dialog. On its own, *Hidden* is not of much use, but is is useful in conjunction with the *User* setting, described below.

When the component is pasted into a form a report, the configuration values are used to set the corresponding properties. Then (unless *User* is set) the configuration values themselves are removed, and do not appear in the form or report definition.

The next screenshot shows the configuration settings for the stock search button. By default, the button initially shows the text *Find*. When clicked, the text changes to *Go*, and the user can enter search values into form fields (just as if the toolbar start-query button had been clicked). When clicked a second time, the query is executed and the button text returns to *Find*. To manage this, the component needs to incorporate two text values for the two button states.



In this example, the two configurations are both marked as *User* configurations. The immediate effect of this is that they *are not* removed when the component is pasted (so, if you paste the button into a form, and then look at the button properties, you will see that any *User* configurations are retained). Now, as they are retained in the form or report, they can be accessed by scripts. Getting a little ahead of ourselves (*slots* are described below), selecting the *Slots* property for this button, and then the *blockAction* slot, the dialog shown below would appear. In the script code, the bits like *button.getConfig('text2')* retrieve the configuration values (which are then used to set the button text).



Hence, in this example, the *text* configuration sets the initial button text, and then *text* and *text2* are used as the query execution state switches back and forth.

Note that when *User* is set, the *Property* combobox is replaced by a text field, so that *User* configurations can be set to anything you like. In this example, the first is set to *text* since this is the property that sets the button text.

Advanced Scripting: Events and Slots

In everything described in the earlier chapter on scripting, script code is always executed in response to an event occuring. This is true, even if the code is stored in a script module, because is must be invoked from the code associated with an event. This is fine in principal, but has a major shortcoming.

Suppose that we want to provide the user with a combo-box control that can be used to quickly find records. The control should contain one entry for each record that has been fetched from the server database, and each entry should derived from some combination of values - for instance, in a form showing client information, the control might show name and department. Whenever the user updates a record, the control should be updated to match, for example when the user deletes a record, or changes a client name. Also, as the user moves amongst the records, the control should change to show the correct entry for the current record.

This can be implemented using a *KBChoice* control, plus some script code associated with some of the events of the form block. Specifically, code needs to be associated with:

- The block's *On Current* event: this is needed so that if the user moves to a different record (maybe using the cursor keys or the toolbar) then the control can be changed to show the appropriate record.
- The block's *Post Query* event: this is needed so that the control can be loaded with the right values after the used executes a query. This also handles the initial database query which fetches all records from the server database.
- The block's *Post Update* event: this is needed so that the control can be updated when the user changes a value which is displayed in the control (in the above example, the client name or department).
- The control's *On Change* event, so that when the user makes a selection, the form can move to the required record.
- The control's *On Action* event, so that the control can be disabled while the user is executing a query.

The problem is that, although the code is all logically associated with the control, most of it is actually written into

the form block; and, even worse, it is split up between several block events. So, if you decide to remove the control, you have to go and clean up the code associated with the block events; also, copying and pasting the control into another form will not work at all, since the block event code is not copied. And, lastly, you may have other code associated with the block events, so there is a real opportunity for confusion.

The *Event/Slot* mechanism provides a way around these problems, and allows you to associate *all* the script code with the control; in the above example, using this mechanism, no code has to be installed into the block events. The basic mechanism is to provide a way of telling *Rekall* that code associated with one object should be executed when an event occurs in some other object [58]; so, for instance, the occurance of the block *On Current* event would trigger the execution of the approriate code in the selection control.

An object in *Rekall* can have one or more *slot*s, each of which contains script code; and a *slot* can be connected to one or more *events* in some other object. When the event occurs, the slot's code is executed. The slot code is defined much as for an event, except that where the code in a event is defined in a function called *eventFunc*, for a slot the function should be called *slotFunc*. This function *always* takes the following arguments:

- The first argument is the control itself, just as for event code.
- The second argument is the object to which the slot is connected, and for which the triggering event occured. This is useful since you can connect a slot to more than one object and event, and allows you to determin which object event occured.
- The third argument is the name of the event. This is again useful if you connect a slot to more than one object or event.
- The remaining arguments are the arguments to the event which originated the event (that is, they are the second and subsequent arguments to the corresponding event function).

As an example, here is the slot function for a button which we want to disable whenever the user is executing a query. This slot would be connected to the *On Action* event of the block which contains the button. The first argument is the button itself, and the second the block. If this code is only connected to the *On Action* event then the third argument will always be *onaction*. The fourth argument gathers up all remaining arguments as a list [59]; the first item in the list is the specific action. By the way, this button may be the one that causes the query execution, that is, it might be the one that does *block.doAction(RekallMain.actQuery)* as part of its *On Click* event; but making the code below part of a slot and connecting it to the block's *On Action* event, it works however the query is started.

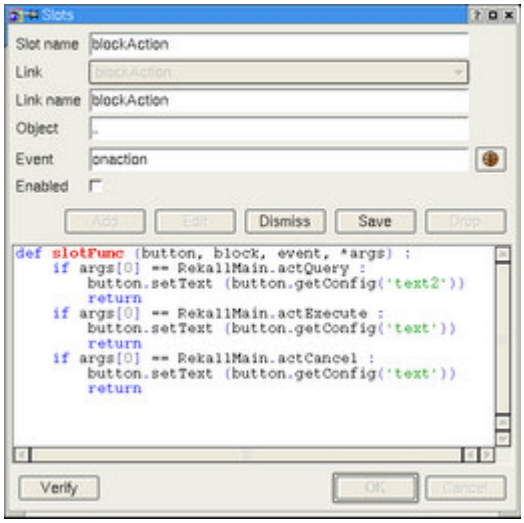
```
def slotFunc (button, block, event, *args) :
    if args[0] == RekallMain.actQuery :
        button.setEnabled (0)
        return
    if args[0] == RekallMain.actExecute :
        button.setEnabled (1)
        return
    if args[0] == RekallMain.actCancel :
        button.setEnabled (1)
        return
```

To access the slots of an object in a form or report (or in a component), open the properties dialog for the object in the normal way, then double-click the *Slots* property. This brings up a list of all slots associated with the object, from which you can add new slots, edit existing ones, and delete unwanted ones.

The earlier screenshot showed a slot immediately after the *Edit* button han been clicked, bringing up the slot dialog. At the top is the slot name; *Rekall* does not actually use this at present, and it can be anything you like. Since a slot can be connected to more than one event, the next control is a combobox which shows all current connections. Each

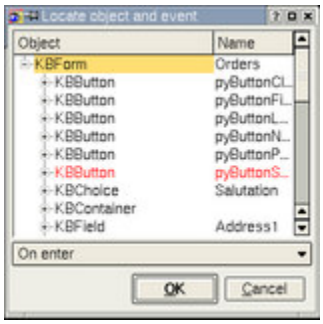
connection has a name, again *Rekall* does not use it and it can be anything you like.

The next three controls are disabled unless a link is being edited and show, in order, the name given to the link, the object to which the connection is made, and the particular event. The next screenshot shows the same dialog, but after clicking the *Edit* button in that dialog to edit the *blockAction* link (the fact that the slot and the link have the same name is not significant).



The object to which the slot is connected is identifier by a path in just the same way as used in the *object.getNamedCtrl()* script method, ie., it is the path from this object to the target object. In this example, the object path is .., which means the object's parent, which is the block that the button is embedded in. If we thought that the button might be embedded in a container, itself embedded in the block, then a more general path would be *getBlock()* [60].

A convenient way to set the object path and event controls is to use the helper button to the right. This shows a small dialog, illustrated below, which shows the structure of the form or report. The object whose slot you are editing is shown in red. You can then navigate to the object to which you wish to connect; the combobox at the bottom then shows the events for that object, and you can select the required one.



Reusable components and the Event/Slot mechanism

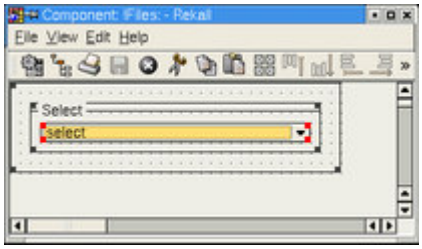
The final section of this chapter presents some examples of reusable components which make use of the event/slot mechanism. The examples are taken from the stock components supplied with *Rekall*.

A Record Selection Tool

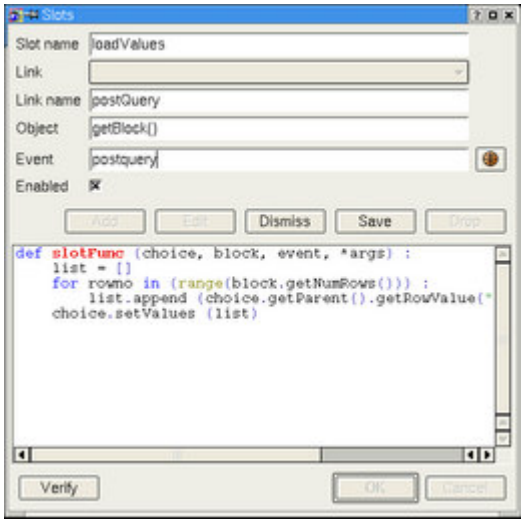
This is the combo-box control described at the start of the previous section on events and slots. To recap, it is a drop-down control that shows one entry for each record returned by the query, and can be used to navigate from one record to another. It is updated as appropriate whever the user makes a change to a record, or if the user moves

between records using, say, the tool bar or menu bar.

The first screenshot below shows the basic component, which has been created as a *form* component. The combobox is a *choice* control and has been embedded inside a container, so that we can show it inside a box; the container has the *Frame style* property set to *Plain,Box width 1* (there is a second reason as well, described later). In addition, the *X-mode* of the choice control has been set to *stretch* so that once the component has been pasted into a form, changing the width of the container will change the width of the choice control.



The first slot to consider is the one which loads values into the choice control (note that although the choice control has been given the name *select*, the *Display expression* is empty, so that it is not loaded directly from the server database). To set up this slot, open the properties dialog for the choice control, double-click the *Slots* property, then click the *Add* button. This brings up an empty slot dialog, which is shown in the following screenshot with value filled in. The code shown further on.



The *Slot name* is set (arbitrarily) to *loadValues*. The slot is being linked to the blocks *postquery* event, so that the code is executed just after the block has fetched data from the server database and the *Link name* is (again arbitrarily) set to *postQuery*. Importantly, the *Object* to which the slot is connected is set to *getBlock()* and the event to *postquery*. Of these, *getBlock()* will find the first block which encloses the choice control, so that we need not worry about having the choice control embedded in a container rather than directly inside a block. Note that since we are designing a component from scratch, the choice control is not actually inside a block at present, so the helper button is not of any use, and the *Event* setting must be set to *postquery* manually [61]

Clicking the *Save* button saves the link. Now, this slot must also be executed after the user makes any changes to displayed data, since this might affect the values displayed in the choice control. For this reason, the slot must also be connected to the *postsync* block event, which is triggered after a record is inserted, updated or deleted. Clicking the *Add* button allows a new link to be added, this time with name *postSync*, object *getBlock()* and event *postsync*. Clicking *Save* again saves the new link (the combobox near the top will now contain two entries, and is used to switch between multiple links). Finally, the *OK* button saves the slot.

The code used in this slot is shown below. Essentially, it builds a list of text strings, one for each record fetched from the block from server database and loads them into the choice control. Note that the choice control (ie., the

object which this slot is part of) is the first argument to the slot function, and that the block (the form object in which the event occured that triggered the execution of this slot) is the second. The third argument is the event which occured, and will be either *postquery* or *postsync*; in this case we are not actually interested, but the argument must still be given in the code. Similarly, any arguments to the event are gathered together as a *python* list in the fourth argument, but are also ignored here.

```
def slotFunc (choice, block, event, *args) :
    list = []
    for rowno in (range(block.getNumRows())) :
        list.append (choice.getParent().getRowValue("selector_display", rowno))
    choice.setValues (list)
```

The question arrises, how will the code pick up the actual values that are to be displayed in the choice control? To do this, the container has a *hidden* control which is given the name *selector_display*. In addition, the hidden control has a configuration setting which provides a prompt for the *Display expression* property of the hidden control. So, when the component is pasted into a form, the user will have to provide an SQL expression which specifies what value the choice control will display for each record retrieved from the server database [62] ; this is then substituted into the pasted component. When the form executes, the SQL query used to retrieve from the server database will include the SQL expression, and the values so retrieved are "displayed" in the hidden control (the user does not see them since the control is hidden, but they are still there). The slot code then picks up these values (*getRowValue ("selector_display", rowno)* and loads them into the choice control. Note that the hidden control has the *No Update* property set so that the form does not use it in a update or insert query.

By the way, this is the other reason for using a container. This means that if the user pastes the component into a form, and then later removes is, removing the container removes both the choice control *and* the hidden control, so all trace of the component is eradicated. Had a container not been used, then the choice control and the hidden control would be placed directly into the component; after pasting, removing the choice control would leave the hidden control behind in the form.

We now add a second slot, which is connected to the block's *On Current* event. This is used so that when the user moves between records, for instance by using the navigation buttons on the toolbar, or the cursor keys, the choice control is changed to show the corresponding value. In this case, the slot is named *onCurrent*, while the (single) link is named *onCurrent*, is connected to object *getBlock()* and to the *oncurrent* event. The code is shown below.

```
def slotFunc (choice, block, event, *args) :
    row = block.getQueryRow()
    if row >= block.getNumRows() :
           choice.setCurrentItem (row, 0)
    else : choice.setCurrentItem (row, row + 1)
```

When executed, the code gets the current query row number from the block. If this is greater than the total number of records in the block then the user must be adding a new record, and the choice control is set to show item zero (which will be blank), otherwise (and normally) it shows the corresponding entry (the addition of one is needed since the block numbers records from zero, but these appear in the choice control starting at item one).

Next, we need to arrange that when the user changes the current selection in the choice control, the block moves to the corresponding record. This is simply done with the *On change* property of the choice control:

```
def eventFunc (ctrl, row, value) :
    item = ctrl.currentItem(row)
    if item >= 1 :
        ctrl.getBlock().gotoQueryRow(item - 1)
```

This component will work, but it has one shortcoming; when the used starts a query (say, by using the start database query tool on the toolbar), it would be nice if the selection component was disabled, and was re-enabled when the user either executes the query or cancels the query. The first stage to doing this is to add a slot to the container itself (rather than the choice control), connected to the *On Action* event of the block. The *On Action* event is triggered whenever a block-level event such as *First Record Previous Record*, etc., action occurs. In this case we are interested in the query actions. The code is shown below:

```
def slotFunc (ctrl, block, event, *args) :
    row = block.getQueryRow()
    if int(args[0]) == block.actQuery   :
        ctrl.setEnabled(0)
        return
    if int(args[0]) == block.actExecute :
        ctrl.setEnabled(1)
        return
    if int(args[0]) == block.actCancel  :
        ctrl.setEnabled(1)
        return
```

The code checks on the action, which is the first of the arguments to the event (and hence appears in *args[0]*; it is passed as a string rather than a number to the code converts it), and looks for *actQuery* (user starts a query), *actExecute* (user executes the query), and *actCancel* (user cancels the query). In the first of these, the container (which is passed as the *ctrl* argument) is disabled; in the other two cases it is enabled. Note that the action codes can be accessed as attributes of the block, which is passed as the second argument [63]

If this is executed, you would notice that after the user cancels a query, the choice control is left blank, rather than showing the correct record. This is because there is no *On Current* event following the cancel. To get round this, we need to add one further slot to the choice control, this time connected to the block *On Action* event. The code for this is shown below, and is basically the same as the *On Current* slot, except that it only does anything if the action is cancel.

```
def slotFunc (ctrl, block, event, *args) :
    if int(args[0]) == block.actCancel  :
        row = block.getQueryRow()
        if row >= block.getNumRows() :
               ctrl.setCurrentItem (row, 0)
        else : ctrl.setCurrentItem (row, row + 1)
```

Note that this is an example where two slots in different objects are connected to the same event on the same object.

---

[54] Storing locally makes it easier to use the components in several *Rekall* databases, since they are accessible whatever *Rekall* database you have open. On the other hand, you *could* create a server database purely to hold components, and access it from all your *Rekall* databases.

[55] OK, three. Since components, like forms and reports, are defined in XML, they can be created using a text editor.

[56] This is one way to create a locally saved component.

[57] You could change the display expression when you paste the component into a form or report, but in that case you might as well leave it blank in the component.

[58] If you are familiar with *Qt*'s *signal/slot* mechanism, this will sound familiar; an *event* in *Rekall* is equivalent to a *signal* in *Qt*. Indeed, internally *Rekall* uses this as part of the implementation of its *event/slot* mechanism. But if you are not familiar, don't worry!

[59] This is a *python*ism. You definitely need to use this if the slot is to be connected to two or more events that have a different number of arguments.

[60] Using *getBlock()* works even if the button is inside a container which is inside another container, which is in turn inside a block - or any such nesting. But note that *getBlock()* must be written exactly that way, is is not implemented as a function call, so if you (say) add some spaces, it won't work!

[61] The events names are the real names of the events. The names that appear when the helper is used are the descriptive names which appear in the property dialogs.

[62] What would be really nice would be a way for the component selection dialog to show the user a list of table columns that are valid in the block into which the selector is being pasted. We are thinking about this!

[63] Prior to version 2.0.0 of *Rekall*, these codes were available via the *RekallMain* module. This is still true of version 2.0.0, but they can now be accessed via a *block* object.

Appendix A. Primary and Unique Key Columns

Table of Contents

This appendix describes the issues involved in primary and unique key columns in tables, and how this relates to tables created by *Rekall*, compared to tables which already exists when *Rekall* is used to access an extant database.

Identifying Rows in Tables

When *Rekall* displays table data in a form (or when displaying table data directly, which is effectively a form), then in order to update the table, *Rekall* must be able to uniquely identify each row in the table. This is because *Rekall* must issue an SQL query of the form *update tablename set .... where colname = uniquekey* where *colname* is the name of a column whose contents are unique (ie., is different in every row of the table) and *uniquekey* is the unique key value for the row to be updated.

In addition, for *Rekall* to be able to insert a new row into the table, it must be able to ascertain the unique key value for the row which is inserted.

For some databases, this is always possible. For instance, in *PostgreSQL* every row has associated with it a unique indeitifier called the *oid*, which can be retrieved along with the row data; after a new row is inserteed, the *oid* of that row can be ascertained. Similarly, *Oracle* has a *rownum*.

In other cases, however, this is not always possible. *MySQL* can mark a column as *auto-increment*, in which case an incrementing value is automatically generated for each row inserted; this value can be ascertained immediately after

the insertion. However, if there is no *auto-increment* column, this is not possible.

Tables Created by Rekall

Rekall defines a column pseudo-type *Primary Key*. The interpretation of this varies from server database to server database, but in all cases it is mapped to a column which provides a unique key which satisfies the requirements listed above for row insertion. For instance, using *MySQL* as the server database it will map to a 4-byte integer column which is marked as *primary key* and *auto-increment*.

If you are creating a new table from within *Rekall*, and you have no particular reason to do otherwise, the best course is to use the *Primary Key* pseudo-column type. Note that this is distinct to the checkboxes in table design view which mark columns that the server database itself calls primary columns.

Note that if you explicitly create a column which matches the *Primary Key* pseudo-column, then it will appear in the table designer as type *Primary Key*.

Accessing Extant Tables

When you use *Rekall* to access a table in a database, then there are three possibilities.

- If *Rekall* can identify a column as providing a suitable unique key, which can also be retrieved after row insertion, then row update, deletion and insertion will work. In *MySQL* for instance, any *auto-increment* column will suffice.
- If *Rekall* can identify a column as providing a unique key, but cannot retrieve the column value after a row insertion, then *Rekall* will be able to update and delete columns, but not insert [64] them.
- If *Rekall* cannot identify any unique key column, then it can display table data, but it cannot update or delete rows, nor insert new rows.

Specifiying Unique Key Columns

When a form (or report) is being designed, and which retrieves data from a table (eg., whete the top-level type is *table*) *Rekall* asks for the *Unique Key* column for the table which is being accessed by the form. If the *Auto* option is chosen, then *Rekall* will try to pick a column using the following criteria:

- If a *Rekall* can determin a column containing a unique key which can be determined after row insertion, then it will use that column. Update, deletion and insertion are possible.
- If the above is not possible, but a unique column can be found, then it will use that column. Update and deletion are possible, but insertion is not possible.
- If neither of the above are possible, no default it used. Data can be displayed but not updated.

Alternatively, you can specify a column to be used via the *Primary Key Only* and *Any Unique Column*. These force *Rekall* to use the specified column (and to check that the column is primary or unique respectively). In either case, similar criteria are used to decide what sort of updates are possible.

The last possibility is *Any Single Column*. This option should be treated with caution; it tells *Rekall* to treat the specified column as unique (whether or not it really is), and to assume that the value does not change on row insertion or update (even if there are triggers that do so). This is useful where you have a table which you *know* is in practice unique. With this option set you can always insert and update data; the only retriction is that the user must

enter values for it [65].

Fake Unique Keys for Insertion

Under the server database advanced properties dialog is an option *Fake unique keys for insertion*. This applies to table data view. If *Rekall* finds that it cannot manage row insertion using the default methods (searching for columns whose values are available immediately after an insert, and so forth) and this option is set, then it will treat a unique column in the table (if one exists) as for the *Any Single Column* setting described above, so that insertion becomes possible.

Key Generator Functions

A future release of *Rekall* may include *key generator functions*, which can be used to generate unique keys for newly insterted rows. This will help, for instance, the situation where *Rekall* is accessing an extant database which is itself already accessed by, say, scripts which contain code to generate unique key values.

---

[64] *Rekall* could not insert and then display, even if it then prevented subsequent update or deletion of the row. Consider the case where row insertion triggers a server database event which updates another column which is displayed in the form. There would be no way of retrieving the column value.

[65] Unlike, for instance, a *MySQL* auto-inrement column, where *Rekall* can leave the server database to generate a value.

Appendix B. Database Drivers

Table of Contents

This appendix describes the limitations associated with each server database supported by *Rekall*.

MySQL

The driver can only (automatically) retrieve an inserted row if that row contains an *auto-increment* column.

The advanced settings for the server dialog are described below.

Ignore MySQL character set

The *MySQL* server can be configured for a range of character encodings. If this option is not set, and a character encoding has not been set in the common advanced settings, then *Rekall* will attempt to use the database character encoding. If this option is set, then the *MySQL* character encoding is ignored.

PostgreSQL

Although *PostgreSQL* supports *large* objects, these require explicit programming support in the driver, and are not supported.

It is always possible to retrieve the last inserted row, and hence any unique key in that row even if the key is generated by the *PostgreSQL* server (for instance, a *serial* column).

The advanced settings for the server dialog are described below.

Use Serial Type for Primary Key

The default behaviour of the *PostgreSQL* driver, when creating a table with a primary key, is to create an associated sequence and to use this to generate unique primary key values. The sequence is used in such a way that if the table is renamed, then the sequence can also be renamed. This means that the sequence *create table A*, *rename table A as B*, *create table A* works as might be expected.

However, if this option is set, then a primary key column is created using the *PostgreSQL serial* type. This actually causes *PostgreSQL* to create an integer column and a related sequence, and to assign a default value expression to the the column which uses the sequence (hence, inserting a row into the table without specifying a value for the column will use the next value from the sequence).

While this is convenient, and accords with the purpose of *serial*, the create table/rename tabel/create tabel sequence described above will fail, since the sequence is not renamed, and the second create table fails.

Show Tables Irrespective of User

Normalls, *Rekall* only shows *PostgreSQL* tables (and views) which are owned by the logged-in user. If this option is set then all tables (and views) in the *PostgreSQL* server database to which *Rekall* is connected will be shown.

Show PostgreSQL Objects

Setting this option will show *all PostgreSQL* objects (ie., the tables and views whose names start with *pg_*).

Log internal driver queries

The *PostgreSQL* driver issues a number of queries in order to retrive information (such as the columns in a table) which are not specified by the user. Normally, these queries are not passed to the query logger, however this option can be set to enable such logging.

Requires SSL Connection

Setting this option will cause a connection to the *PostgreSQL* server database if the *PostgreSQL* client library supports SSL connections to the server, but an SSL connection cannot be established. The RedHat8.0 client library is known to support SSL; the Windows client library does not.

XBase

XBase itself does not provide an SQL interface. The *Rekall* driver accesses *XBase* files via the *XBSQL* library, which implements a limited subset of SQL.

XBase has a restricted set of types:

- Logical.
- Numeric: up to 16 digits with a sign, or 17 without.
- Floating point: up to 17 digits, less space for a sign or decimal point.
- Character: up to 254 characters, no null characters.
- Date: eight characters, in the format YYYYMMDD.
- Memo: up to 32760 characters, including nulls.

There are two other major restrictions. Firstly, *XBase* has no notion of *not null* columns, and does not distinguish an empty field from a null field.

Secondly, there is no notion of a *primary key* column. The driver works round this by mapping the pseudo-type *Primary Key* to a character field of length 22, provided that this is the first column in the table. It generates key values by concatenating the time at which the driver was started (expressed as seconds since 1970) with an index that increments for each record inserted. This is very likely (but not guaranteed) to be unique.

The driver can always retrieve information about a row which has just been inserted.

The advanced settings for the server dialog are described below.

Pack database files on close

By default, when records are deleted from an XBase table, the record is not actually freed up, rather it is simply marked as deleted. This means that the files in which the tables are stored grow in size as records are deleted and added, and also has side effects related to indexes.

Setting this option will case tables to be packed when *Rekall* exits, releasing unused space. Note that this is only done for tables for which one or more records have been deleted.

Note: The underlying XBase library has a *real delete* feature, however this does not appear to function correctly at present, and is not used.

Case sensitive matching

By default, string matching for the (in)equality and *like* operators is case-insensitive. Setting this option makes the operation case sensitive.

Wrap names with [....]

This option cases *Rekall* to wrap names in expressions with the [...] characters, hence the expression *Client.ClientID = 12* becomes *[Client].[ClientID] = 12*. This notation, which is the same as used by Microsoft Access® allows for names which contain the space character, and which sometimes appear as column names in *XBase* data files.

However, it does not help with other problematic names, for instance a column name that contains a period (.) character. Such names cannot be handled bu *Rekall*, since there is no general way to decide whether such a character, appearing in an SQL expression, is part of a name or is a character in an operator.

Minimise memory usage

Setting this option turns on a mode in the *XBase* wrapper library which reduces memory usage. Specifically, when a *select* query is executed, initially only information about selected rows is stored; data values are not actually retrieved until requested. This decreses memory usage at the expense of execution speed.

ODBC

ODBC is a generic server database interface which connects to a range of server databases via ODBC drivers. Using the *Rekall* ODBC driver allows *Rekall* to connect to any server database for which here is an ODBC driver.

The ODBC interface allows *Rekall* to determin various information about the underlying server database, such as column types. However, it has a number of shortcomings, the major one of which is that it does not provide a means to identify a row that has just been inserted into the database. Because of this, the *Rekall* ODBC driver comprises a main driver plus *helper* modules for certain server databases; if the driver determines that the server database is one of these, then the corresponding helper moduld is used.

Currently, the driver has helper modules for *MySQL* and for *Jet*, which is the Microsoft Access database engine (for files with the *.mdb* extension). [66]

The advanced settings for the server dialog are described below.

Map CR/LF in strings

If set, the CR/LF (DOS/Windows style newline) character sequences in text data are mapped to LF when data is retrieved from the server database

Show system tables

By default, the ODBC driver does not display tables which the underlying ODBC interface classifies as system tables. Setting this option will cause such tables to be listed.

Wrap names with [....]

This option cases *Rekall* to wrap names in expressions with the [...] characters, hence the expression *Client.ClientID = 12* becomes *[Client].[ClientID] = 12*. This notation, which is the same as used by Microsoft Access® allows for names which contain the space character, and which sometimes appear as column names in *XBase* data files.

---

[66] Since *Rekall* has its own *MySQL* driver, this is the recommended way to access a *MySQL* server database.

Appendix C. The XBase interface

Rekall provides access to XBase format files using an SQL wrapper *XBSQL* which implements a limited subset of SQL.

Currently, the XBSQL driver which accesses XBase format files supports the following:

- select *e1*, ... from *t1*, where *c1* ... order by *o1*, ...
- insert into *t* values (*e1*, ...)
- insert into *t* (*c1*, ...) select ...
- insert into *t* select ...
- insert into *t* (*c1*, ...) values (*e1*, ...)
- update *t* set *c1* = *e1*, ... where *c1*, ...
- delete from *t* where *c1*, ...
- create table *t* (*colspec*, ...)
- drop table *t*

Expressions *e1* are currently fairly limited, just some basic arithmetic and string concatenations, plus equality/inequality, greater/less greater-or-equal/less-or-equal.

Available column types for create column specifications are:

- *int*
- *double*
- *char*
- *blob*
- *date*

XBase files all contain fixed-width columns, with the exception of the *blob* type, which maps to a *memo* column. The first three cases can therefore have *(width)* appended. Individual columns can be indexed, for example, *(...., ident int(10) index, ...)*.

Note the XBase has no notion of a *not-null* column, nor of a primary key. The latter is handled by *Rekall* using a 22-character wide *char* column, when it is the first column in the table; *Rekall* generates key values which are almost guaranted to be unique [67] .

Note that in the table designer, *int* appears as *decimal*.

Unsupported SQL includes

- sub-selects
- create .... as
- alter table ...
- inner, etc., joins

---

[67] The key value is generated by concatenating the system time (in seconds since the epoch) at which the driver is started, with a serial value which is incremented by one for each key generated. Hence, an example key is *1008341402.000000023*

Appendix D. Object Properties

Table of Contents

This appendix lists the properties associated with each type of object. The tables are formatted as below, showing the internal property name, the legend which appears in the property dialogs, and the description (which are the text that appears in the property dialogs when a property is being edited).

Note that there is a fair degree of duplication below. The properties that are specific to reports and forms, rather than than the blocks from which they are derived, have been separated out. However, for data controls the properties are listed in full.

Name	Property dialog legend
	Description

Form Properties

language	Scripting language
	Scripting language to be used for script modules in this form
caption	Form caption
	Caption text to be displayed when form is active
stretch	Stretchable
	If set the form can be stretched (resized) when it is displayed; otherwise, the form layout is fixed.
onload	On Load
	Script routine to be invoked when the form is loaded. A value like #Init invokes an external function called onFormInit; otherwise define a function called eventFunc whose single argument is the form.
onunload	On UnLoad
	Script routine to be invoked when the form is closed. A value like #Cleanup invokes an external function called onFormCleanup; otherwise define a function called eventFunc whose single argument is the form.
modlist	Script modules
	List of script modules to be used by this form
implist	Import modules
	List of script modules to be imported by inline scripts in this form.
paramlist	Parameters
	List of parameters which can be set when the form is executed. Parameter values can be substituted into expressions and such.
blktype	Top-level block type
	The top-level block may contain a menu (no query), or acess a query or a table

Remaining properties are as for for blocks.

Form Block Properties

notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the block area in pixels
h	Height
	Height of the block area in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the

	interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
name	Control name
	Control name, used to access control from scripts
master	Parent field
	Field in parent query used to link to child field in this block's query
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
child	Child field
	Field in this block's query used to link to parent expression
bgcolor	Background colour
	Background colour
autosync	Automatic update
	Enabling this option will case field changes to be stored in the database whenever focus moves to a different row or block
frame	Frame style
	Specifies the frame style and width for the block.
showbar	Show Scroll Bar
	Setting this option will show a vertical scroll bar which indicates the range of rows displayed, and allows scrolling through them.
rowcount	Row count
	Number of rows of fields to be shown in this block. If set to zero then the number of calculated based on block size and row spacing.
dx	X-delta
	X-offset in pixels between fields if the rowcount is greater than one
dy	Y-delta
	Y-offset in pixels between fields if the rowcount is greater than one
onaction	On action
	Script routine to be invoked when a block-level action is about to take place. A value like #Action invokes an external function called onBlockAction; otherwise define a function called eventFunc whose two arguments will be the block and the action code.
onuncurrent	On uncurrent
	Script routine to be invoked when a record ceases to be current. A value like #UnCurrent invokes an external function called onBlockUnCurrent; otherwise define a function called eventFunc whose two arguments will be the block and the query row being left.
oncurrent	On current
	Script routine to be invoked when a record becomes current. A value like #Current invokes an external function called onBlockCurrent; otherwise define a function called eventFunc whose two arguments will be the button and the query row number.

ondisplay	On display
	Script routine to be invoked when a record is displayed. A value like #UnCurrent invokes an external function called onBlockUnCurrent; otherwise define a function called eventFunc whose two arguments will be the block and the query row being left.
preinsert	Pre-Insert
	Script routine to be invoked just before a new row is inserted into a table. A value like #Insert invokes an external function called onBlockInsert; otherwise define a function called eventFunc whose two arguments will be the block and the query row number. Insert is aborted unless the function returns true.
preupdate	Pre-Update
	Script routine to be invoked just before a row is updated in a table. A value like #Update invokes an external function called onBlockUpdate; otherwise define a function called eventFunc whose two arguments will be the block and the query row number. Update is aborted unless the function returns true.
predelete	Pre-Delete
	Script routine to be invoked jusst before a row is deleted from a table. A value like #Delete invokes an external function called onBlockDelete; otherwise define a function called eventFunc whose two arguments will be the block and the query row number. Deletion is aborted unless the function returns true.
postquery	Post-Query
	Script routine to be invoked just after a select query has been issued, but before any data is displayed. A value like #PostQuery invokes an external function called onBlockPostQuery; otherwise define a function called eventFunc whose argument will be the block.
postsync	Post-Sync
	Script routine to be invoked just after an insert, update or delete query has been issued A value like #PostSync invokes an external function called onBlockPostSync; otherwise define a function called eventFunc whose four arguments will be the block, the query row, the action and the primary key of the affected row.
hidden	Hidden fields
	Hidden fields are used to retrieve values for use in scripts. The name is the control name by which they are accessed; the expression is that used in the database query.

Report Properties

lmargin	Left margin
	This specifies the left-hand page pargin in millimeters.
rmargin	Right margin
	This specifies the right-hand page pargin in millimeters.
tmargin	Top margin
	This specifies the top-of-page page pargin in millimeters.
bmargin	Bottom margin
	This specifies the bottom-of-page page pargin in millimeters.
blktype	Top-level block type
	The top-level block may contain a menu (no query), or access a query or a table
language	Scripting language

	Scripting language to be used for script modules in this report
caption	Report caption
	Caption text to be displayed when report is active
modlist	Script modules
	List of script modules to be used by this report
implist	Import modules
	List of script modules to be imported by inline scripts in this report.
printer	KBReport.printer
	KBReport.printer
printdlg	KBReport.printdlg
	KBReport.printdlg
paramlist	Parameters
	List of parameters which can be set when the report is executed. Parameter values can be substituted into expressions and such.

Remaining properties are as for for blocks.

Report Block Properties

y	Y-position
	Y coordinate of the control relative to its parent
h	Height
	Height of the control in pixels
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
hidden	Hidden fields
	Hidden fields are used to retrieve values for use in scripts. The name is the control name by which they are accessed; the expression is that used in the database query.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
oncurrent	On current
	Script routine to be invoked when a record becomes current. A value like #Current invokes an external function called onBlockCurrent; otherwise define a function called eventFunc whose two arguments will be the button and the query row number.
postquery	Post-Query
	Script routine to be invoked just after a select query has been issued, but before any data is displayed. A value like #PostQuery invokes an external function called onBlockPostQuery; otherwise define a function called eventFunc whose argument will be the block.
pthrow	Page throw
	Set this to record to throw a page after each record, to group to throw a page after the last record, or

	none to disable page throws.

Block Header

h	Height
	Height of the control in pixels
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

Block Footer

h	Height
	Height of the control in pixels
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

Tabber

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of

	its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
bgcolor	Background colour
	Background colour
frame	KBTabber.frame
	KBTabber.frame
name	Control name
	Control name, used to access control from scripts
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

TabberPage

y	Y-position
	Y coordinate of the control relative to its parent
bgcolor	Background colour
	Background colour
frame	KBTabberPage.frame
	KBTabberPage.frame
tabtext	Tab Text
	This setting specifies the text that appears in the tab.
name	Control name
	Control name, used to access control from scripts
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

Button Properties

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.

ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
text	Button text
	Text displayed in the button
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
onclick	On Click
	Script routine to be invoked when the button is clicked. A value like #Click invokes an external function called onButtonClick; otherwise define a function called eventFunc whose single argument will be the button.

Label

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour
bgcolor	Background colour
	Background colour
font	Font
	Specify font
name	Control name

	Control name, used to access control from scripts
text	Label text
	Text to appear in the label
align	Text alignment
	Specify whether text should be horizontally aligned to the left (default), centred or to the right
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

CheckBox

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
expr	Display expression
	Expression for value to be displayed in the control. If empty, then the control is not set by the query.
default	Default value
	Default value to use if field is not entered
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the

	control cannot be entered by tabbing.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.
onchange	On change
	Script routine to be invoked when the value in the field is changed by the user. A value like #Change invokes an external function called onCheckChange; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the new value.

Choice

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour

	Text colour
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
master	Field name
	The name of the column in the table or query which is displayed in the choice control. If left empty then the choice control is not set from the table or query
default	Default value
	Default value to use if field is not entered
values	Values
	This is the list of values which cab be selected. Values should be separated by the | character
nullval	Null value
	Value to show when field contains null
nullok	Null OK
	If this option is set then then contents of the file may be null, ie., empty
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the control cannot be entered by tabbing.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
morph	Morph control
	If morphing is enabled, then the control is drawn as a simple text value when it does not have focous. This can be used to remove the dropdown on Choice and Link controls.
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.

onchange	On change
	Script routine to be invoked when the value in the field is changed by the user. A value like #Change invokes an external function called onChoiceChange; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the new value.

Link

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour
bgcolor	Background colour
	Background colour
name	Control name
	Control name, used to access control from scripts
master	Parent field
	Field in parent query used to link to child field in the associated query
child	Child field
	Field in the associated query used to link to parent expression
default	Default value
	Default value to use if field is not entered
nullval	Null value
	Value to show when parent field contains null
nullok	Null OK
	If this option is set then then contents of the file may be null, ie., empty
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.

taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the control cannot be entered by tabbing.
show	Display expression
	Expression for the associated query to be displayed in the link control
dynamic	Dynamic
	Set this option to get retrieve the list of possibe values each time focus enters the control; otherwise, the list of generated when the form or report which contains the link is started.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
morph	Morph control
	If morphing is enabled, then the control is drawn as a simple text value when it does not have focous. This can be used to remove the dropdown on Choice and Link controls.
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.
onchange	On change
	Script routine to be invoked when the value in the field is changed by the user. A value like #Change invokes an external function called onLinkChange; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the new value.

Field

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height

	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour
bgcolor	Background colour
	Background colour
font	Font
	Specify font
name	Control name
	Control name, used to access control from scripts
expr	Display expression
	Expression for value to be displayed in the control. If empty, then the control is not set by the query.
default	Default value
	Default value to use if field is not entered
nullok	Null OK
	If this option is set then then contents of the file may be null, ie., empty
evalid	Validator
	If set, this specifies a regular expression used to validate the field contents. Note that the expression is not anchored at either end.
igncase	Ignore case
	Character case is ignored when validating field contents
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
format	Format
	This specifies the display format. Note that a format specification implies a particular data type; number, date, etc.
taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the control cannot be entered by tabbing.
align	Text alignment
	Specify whether text should be horizontally aligned to the left (default), centred or to the right
mask	Input mask
	The input mask controls input to the field, providing a degree of input formatting
helper	Helper name
	If set, then when focus enters the field a helper button appears, which can be used to aid data entry. The

	setting specifies the helper; currently only date is defined.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
morph	Morph control
	If morphing is enabled, then the control is drawn as a simple text value when it does not have focous. This can be used to remove the dropdown on Choice and Link controls.
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.

Memo

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour

font	Font
	Specify font
frame	Frame style
	Specifies the frame style and width for the area in which the memo control appears.
name	Control name
	Control name, used to access control from scripts
expr	Display expression
	Expression for value to be displayed in the control. If empty, then the control is not set by the query.
default	Default value
	Default value to use if field is not entered
nullok	Null OK
	If this option is set then then contents of the file may be null, ie., empty
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the control cannot be entered by tabbing.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.
onchange	On change
	Script routine to be invoked when the value in the mem is changed by the user. A value like #Change invokes an external function called onMemoChange; otherwise define a function called eventFunc whose first two arguments will be the control item and the query row number; the third argument is undefined.
hilite	KBMemo.hilite
	KBMemo.hilite

Pixmap

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
frame	Frame style
	Specifies the frame style and width for the area into which the image is displayed.
name	Control name
	Control name, used to access control from scripts
expr	Display expression
	Expression for value to be displayed in the image. If empty, then the image is not set by the query. In practice, the expression should yield the data for an image.
default	Default value
	Default value to use if field is not entered
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the control cannot be entered by tabbing.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.

onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.
onchange	On change
	Script routine to be invoked when the value in the field is changed by the user. A value like #Change invokes an external function called onPixmapChange; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the new value.

Summary

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour
bgcolor	Background colour
	Background colour
font	Font
	Specify font
name	Control name
	Control name, used to access control from scripts
expr	Summary expression
	Expression for value to be summarised in the field
format	Format
	This specifies the display format. Note that a format specification implies a particular data type;

	number, date, etc.
align	Text alignment
	Specify whether text should be horizontally aligned to the left (default), centred or to the right
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.
summary	Summary function
	Specify the function used to summaries data values
reset	Page Reset
	Reset summary value on each page

RowMark

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
bgcolor	Background colour
	Background colour
frame	KBRowMark.frame
	KBRowMark.frame
showrow	Show row number
	Set this option to display the query row number
name	Control name
	Control name, used to access control from scripts
default	Default value

	Default value to use if field is not entered
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.

RichText

x	X-position
	X coordinate of the control relative to its parent
y	Y-position
	Y coordinate of the control relative to its parent
w	Width
	Width of the control in pixels
h	Height
	Height of the control in pixels
xmode	X-mode
	This setting specified whether the control width is fixed width, whether it floats relative to the right-hand side of its block, or whether it stretches as its block width changes. The setting affects the interpretation of the width value.
ymode	Y-mode
	This setting specified whether the control height is fixed width, whether it floats relative to the bottom of its block, or whether it stretches as its block height changes. The setting affects the interpretation of the height value.
fgcolor	Text colour
	Text colour

bgcolor	Background colour
	Background colour
font	Font
	Specify font
name	Control name
	Control name, used to access control from scripts
expr	Display expression
	Expression for value to be displayed in the control. If empty, then the control is not set by the query.
default	Default value
	Default value to use if field is not entered
rdonly	Read Only
	Set this option to prevent update of the displayed value by the user. Note that the control can still be updated from a script.
taborder	Tab order
	Tab and shift-tab cycle through controls in increasing tab order. A tab order of zero means that the control cannot be entered by tabbing.
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
noupdate	No Update
	If this option is set, the database is never updated from the control (even if the contents are changed). This is useful if you wish to display a value in more that one control, and all but done are marked as read only
onenter	On enter
	Script routine to be invoked when focus enters a control. A value like #Enter invokes an external function called onItemEnter; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onleave	On leave
	Script routine to be invoked when focus leaves a control. A value like #Enter invokes an external function called onItemLeave; otherwise define a function called eventFunc whose two arguments will be the control item and the query row number.
onset	On set
	Script routine to be invoked when the value in the control is set from the database. A value like #Set invokes an external function called onItemSet; otherwise define a function called eventFunc whose three arguments will be the control item, the query row number and the value.

Table Query

server	Server name
	Name of a server in the database servers list
table	Table name
	Name of a table in the database
primary	Unique key

	Name of a unique key column in the table. Strictly, any unique column will suffice, but the primary key column if there is one is preferred.
where	Where condition
	Optional SQL where expression to filter rows
order	Row order
	Optional SQL order expression to specify order or rows
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

Rekall Query

query	Query name
	Name of the query associated with this block
toptable	Top-level table
	This setting specifies the table in the query which will appear at the top (outermost) level. The nesting of sub-forms or sub-reports depends on this.
where	Where condition
	Optional SQL where condition, appended to the where conditions implicit in the underlying query
order	Row order
	Option SQL order expression, appended to any ordering conditions in the underlying query
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.
group	Row grouping
	Option SQL group by expression, appended to any ordering conditions in the underlying query
having	Having
	Option SQL having expression, appended to any ordering conditions in the underlying query

Free-text SQL Query

query	Query text
	Text of the SQL query
server	Server name
	Name of a server in the database servers list
notes	Notes
	The Notes setting may be used for any arbitrary notes, for instance for documentation. Rekall does nothing with the value, other than to preserve it.

Appendix E. Object Methods

Table of Contents

Block Methods

getNumRows			Number of rows in associated query
Return	number		Number of rows
getQueryRow			Get current query row number
Return	number		Row number
totoQueryRow			Move to specified row in query
Return	bool		Success
rowNum	number	No	Row number
getRowValue			Get value from associated query
Return	string		Value
ctrlName	string	No	Column required, as control name
rowNum	number	No	Row number
setRowValue			Set value in associated query
Return	void		
ctrlName	string	No	Column required, as control name
rowNum	number	No	Row number
value	string	No	Value
doAction			Perform action in block
Return	bool		Success
action	number	No	Action code, see RekallMain
isInQuery			See if query (find) is in progress
Return	bool		Query (find) in progress
firstRecord			Go to first record
Return	bool		Success
previousRecord			Go to previous record
Return	bool		Success
nextRecord			Go to next record
Return	bool		Success
lastRecord			Go to last record
Return	bool		Success
addRecord			Add a new record

Return	bool		Success
saveRecord			Save current record
Return	bool		Success
deleteRecord			Delete current record
Return	bool		Success
startQuery			Start query
Return	bool		Success
executeQuery			Execute query
Return	bool		Success
cancelQuery			Cancel query
Return	bool		Success

Button Methods

setText			Set button text
Return	void		
text	string	No	Button text

Choice (ComboBox) Methods

currentItem			Get current item
Return	number		Currenly selected item number
row	number		Row number
setCurrentItem			Update current item
Return	void		
row	number		Row number
item	number	No	New current item number
setValues			Load set of choices
Return	void		
values	list	No	List of values as strings

Form Methods

openForm			Open a named form
Return	bool		Success
formName	string	No	Name of form to be opened
parameters	dictionary	Yes	Parameter dictionary
openReport			Open a named report
Return	bool		Success
reportName	string	No	Name of report to be opened
parameters	dictionary	Yes	Parameter dictionary

openTextForm			Open an XML form definition
Return	bool		Success
xmlDefn	string	No	XML definition
parameters	dictionary	Yes	Parameter dictionary
openTextReport			Open an XML report definition
Return	bool		Success
xmlDefn	string	No	XML definition
parameters	dictionary	Yes	Parameter dictionary
openTable			Open a named table
Return	bool		Success
formName	string	No	Name of table to be opened
parameters	dictionary	Yes	Parameter dictionary
openQuery			Open a named query
Return	bool		Success
formName	string	No	Name of query to be opened
parameters	dictionary	Yes	Parameter dictionary
openServer			Get low-level connection to server database
Return	pydbi		Connection object
serverName	string	No	Server name
executeCopier			Execute a copier
Return	number		Number of rows copied
copierName	string	No	Copier name
parameters	dictionary	Yes	Parameter dictionary
getServerList			Get list of servers
Return	list		List of server names
getObjectList			Get list of objects on server
Return	list		List of object names
serverName	string	No	Server name
objectType	string	No	Object type
getObjecttext			Get XML definition of object
Return	string		XML definition text
serverName	string	No	Server name
objectName	string	No	Object name
close			Close the form
Return	void		
findOpenForm			Locate an open form
Return	form		From or None if not open
formName	string	No	Name of form to be found
close			Close the form
Return	void		

Container Methods

getNumRows			Number of rows in associated query
Return	number		Number of rows
getQueryRow			Get current query row number
Return	number		Row number
getRowValue			Get value from associated query
Return	string		Value
ctrlName	string	No	Column required, as control name
rowNum	number	No	Row number
setRowValue			Set value in associated query
Return	void		
ctrlName	string	No	Column required, as control name
rowNum	number	No	Row number
value	string	No	Value

Item Methods

setValue			Update value in control
Return	void		
rowNum	number	No	Query row number
value	string	No	Value
getValue			Get value from control
Return	string		Value
rowNum	number	No	Query row number
setTabOrder			Set control tab ordering
Return	void		
order	number	No	Tab order number
setEnabled			Enable or disable control
Return	void		
rowNum	number	No	Query row number
enable	bool	No	True to enable
setVisible			Show or hide control
Return	void		
rowNum	number	No	Query row number
show	bool	No	True to show
isEnabled			Test if control is enabled
Return	bool		True if enabled
rowNum	number	No	Query row number
isVisible			Test if control is visible
Return	bool		True if visible

rowNum	number	No	Query row number

Label Methods

setText			Set label text
Return	void		
text	string	No	Label text

Object Methods

setEnabled			Enable or disable control
Return	void		
enable	bool	No	True to enable
setVisible			Show or hide control
Return	void		
show	bool	No	True to show
isEnabled			Test if control is enabled
Return	bool		True if enabled
isVisible			Test if control is visible
Return	bool		True if visible
getName			Get control name
Return	string		Name
setAttr			Set attribute (property)
Return	void		
name	string	No	Attribute name
value	string	No	Value to set
getAttr			Get attribute (property)
Return	string		Attribute value
name	string	No	Attribute name
width			Get control width
Return	number		Width
height			Get control height
Return	number		Height
resize			Resize control
Return	void		
width	number	No	New width
height	number	No	New height
getParent			Get parent object if any
Return	object		Parent or None
getBlock			Get enclosing block if any
Return	block		Enclosing block or None

getForm			Get enclosing form if any
Return	form		Enclosing form or None
lastError			Get last error
Return	string		Text message for last error
getNamedCtrl			Locate control by name
Return	object		Control or None if not found
name	string	No	Control name relative to this object
bool	errorOK	Yes	Show error dialog if no control found
getControls			Get list of data controls
Return	list		List of data controls

Tabber Page Methods

setCurrent			Make this page current
Return	void		

RekallMain functions

messageBox			Show a simple popup message box
Return	void		
	string	No	Message to display
	string	Yes	Popup box caption
queryBox			Show a simple Yes/No query box
Return	void		
	string	No	Message to display
	string	Yes	Popup box caption
promptBox			Show a simple prompt box
Return	void		
	string	No	Message to display
	string	Yes	Default value
	string	Yes	Popup box caption
choiceBox			Show a popup with selection combobox
Return	void		
	string	No	Message to display
	list	Yes	List of entries for combobox
	string	Yes	Popup box caption
logscript			Write message to script log window
Return	void		
	string	No	Message to be written

Appendix F. tkcRekall: Rekall on the Sharp Zaurus

Table of Contents

tkcRekall is a port of *Rekall* to the Sharp Zaurus handheld device.

First, this is *not* a stripped down version which has but a small fraction of the functionality of the desktop version. Rather, it has almost all of the desktop functionality. The limitations are primarily those of the Zaurus itself:

- The amount of memory available on the Zaurus is small compared to many desktop systems, and *tkcRekall* itself occupies around 7M of memory.
- The Zaurus screen is much smaller than a desktop or portable display, so the amount of information than can be displayed is proportionally less.
- The Zaurus processor is significantly slower than most desktops and portables now in use.

The differences between *tkcRekall* and the desktop version are primarily in the layout of menus and toolbars, and the layout of dialogs. The remainder of this appendix summaries the differences,.

Right-Click Operation

A lot of *Rekall* functionality is accessed using the right-hand mouse button. In *tkcRekall*, this is achieved by pressing and holding with the stylus. In some places, for instance inside form controls this is not available at all, however, all functions can be accessed via double-clicks, or the menu or tool bar.

Menus and Toolbars

None of the windows in *tkcRekall* have menus, in order to save the space they would otherwise occupy. However, the menus found in the desktop version can be accessed via the left-hand most button that appears on each toolbar.

Dialog Layouts

Generally, dialogs have been altered to remove the space between controls, and to make better use of the landscape shape of the Zaurus screen. In a few cases some dialog controls have been removedbut you should always be able to achieve the same end result.

For instance, the tab order dialog lacks the buttons which allow automatic tab ordering, but you can still set any required ordering.

Table Design

The table design window shows the columns and the additional column details under two separate tabs, rather than above one another. As a general point, tabbed controls are used more often in *tkcRekall* to componsate for the small screen.

Query Design

The query design window has been changed. Instead of showing the table, expression and SQL windows at the same time (with the sizes adjustable), they appear under three separate tabs. The popup menu which allows you to set table aliases (and to remove tables from the query) is accessed by "right-clicking" in the table title bar, rather than in the field list.

Copier Design

The copier design window has been changed so that, rather than the source and destination panels appearing side-by-side, they are overlaid, and are selected via additional buttons on the toolbar.

Appendix G. RekallRT: The Rekall Runtime

Table of Contents

RekallRT is the runtime version of *Rekall*. It is intended for use where you have developed a *Rekall* database and wish to deploy it to users without giving them access to the design functions (so that they cannot change forms and reports and so forth), nor direct access to tables and queries.

RekallRT is not a system which "compiles" the database application, rather it is a version of *Rekall* with all the design functions stripped out, and with various other restrictions. These are described in this appendix.

Startup Form

When *RekallRT* opens a *Rekall* database, the database *must* have a form which executes automatically (ie., a form named *MainForm* in a server database for which the *AutoStart* options is set). If the *Rekall* database does not fulfill these requirements, then *RekallRT* will not run.

Open Last Database

The option to open the last database when *Rekall* is started up (and is not given a database on the command line) is disabled in *RekallRT*. This is done since it would otherwise be possible to get in a situation where the last database no longer existed; *RekallRT* would try to open this, fail, and immediately exit. If *RekallRT* is started without specifying a database on the command line it will show the file-open dialog (and exit if no database file is selected).

Database Window

The database window cannot be accessed. This in turn means that forms, etc., can only be accessed directly or indirectly from the startup form, and through any scripts that they run.

Debugging

The *python* debugger is not enabled in *RekallRT*. Any scripting error will be reported to the user; it will either be ignored or the form (or report, etc.) will close.