QddbScript in a Nutshell

Herrin Software Development, Inc.

March, 2001

Contents

Chapter 1

Introduction

Qddb is a database tool that provides a very fast and powerful graphical user interface as well as the programming interfaces *Qtcl* and the graphical *Fx Toolkit*. Qtcl provides the underlying database engine using the Tcl command language [1], while Fx provides the link between the user and the data. Although Fx is very simple for programmers, the screen layout/design process is somewhat cumbersome and monotonous.

After much experience designing applications with Fx, we decided that a higher-level interface for the screen layout was necessary to reduce the amount of time that a programmer spends designing screens and maintaining applications. A 30 attribute database might require a couple hundred lines of Fx code and several hours of a programmer's time to get it "just right". Most Fx screens are quite similar, so it seemed that we could design a very simple language (like HTML) that anyone could manipulate. Qddb *Schemas* can be designed so quickly, it is a real shame to force programmers to spend so much time designing the interface.

1.1 The Idea

QddbScript separates the design of the database schema from the design of the user interface. Custom database functions are easily inserted without obscuring the layout as is common with Fx scripts. You also don't have to deal with the level of detail required by Tk; QddbScript handles all the levels of frames and entries for you. The steps required to build a custom database and user interface with QddbScript are:

1. Design the schema
2. Generate a user interface template from the schema
3. Customize the template
4. Add custom database features

After the form is designed, custom database features can be inserted and linked to the form with simple Fx toolkit commands (and Tcl/Tk). Many aspects of the user interface can be dynamically modified by the user as with all Fx-based applications[1].

1.2 Overview

QddbScript is extremely simple and even somewhat language independent. Its simplicity is best demonstrated with a simple example, so let's work our way through one. Suppose we have the following simple Qddb Schema for business invoices:

```
Number verbosename "Invoice Number" type string
Date type date format "%m/%d/%y %H:%M:%S"
DayOfWeek verbosename "Day of Week"
SalesPerson verbosename "Sales Person"
Items (
    Qty type real format "%.2f"
    Code
    Desc verbosename "Description"
    Taxable exclude
    Price type real format "%.2f"
    Total type real format "%.2f"
)*
Subtotal type real format "%.2f"
Discounts type real format "%.2f"
Tax type real format "%.2f"
Total type real format "%.2f"
PaymentType verbosename "Payment Type" type real format "%.2f"
PaymentAmount verbosename "Payment Amount" type real format "%.2f"
AmountDue verbosename "Amount Due" type real format "%.2f"
```

The database described by this Schema contains a number of records, where each record contains a unique invoice number, any number of items, and some other informational fields. Using Qddb's *nxqddb(1)*, we can have this simple database up and running in a few minutes. However, if we want to design a custom form, we have to use the Fx toolkit or $Qtcl + Tk$. As of Qddb 1.43R5, QddbScript gives us the ability to display the database fields with an automatically generated form-style interface. If the automatically generated form doesn't satisfy the designer, they can quickly modify the QddbScript by moving fields around, adding notebook tabs, changing field types and

[1] Fx-based applications allow the user to change field widths and heights, radiobutton values, fonts, colors, and so forth.

sizes, and so forth. The entire interface definition is written in QddbScript and does not require knowledge of Tck/Tk or any other scripting language. Custom functions, however, must be written in a standard (and more complicated) scripting language such as Tcl (Qtcl).

To build the initial database and interface template, you can run the following commands:

```
1) $ qnewdb MyDB
Remember to edit the Schema for relation MyDB.
2) $ vi MyDB/Schema
(type in Schema description)
3) $ qddbscript autogen MyDB
4) $ vi MyDB/QddbScript
(change the interface definition)
```

After this sequence of commands, the contents of `MyDB/QddbScript` is as shown in Figure 1.1. Notice the marked (and intentional) similarity between the original Schema and the QddbScript. While this convenient default (see Figure 1.2 for a screenshot) may work fine for some situations, most applications need a more customized look and feel. By adding and moving some lines in our QddbScript and increasing the widths of various fields (Figure 1.3), we get a reasonably nice-looking form as seen in Figure 1.4. Typing has been significantly reduced from the equivalent code in *Fx* and the resulting script is significantly easier to manage (add new fields, etc.) Of course, all the standard database functions are automatically in place and ready to use after the QddbScript is modified.

```
[prologue]
[menubar]
[options autoalign]
[entry attr=Number width=10]
[entry attr=Date width=10]
[entry attr=DayOfWeek width=10]
[entry attr=SalesPerson width=10]
[break]
[frame attr=Items border=3 relief=groove]
    [entry attr=Items.Qty width=10]
    [entry attr=Items.Code width=10]
    [entry attr=Items.Desc width=10]
    [entry attr=Items.Taxable width=10]
    [break]
    [entry attr=Items.Price width=10]
    [entry attr=Items.Total width=10]
[end frame]
[break]
[entry attr=Subtotal width=10]
[entry attr=Discounts width=10]
[entry attr=Tax width=10]
[entry attr=Total width=10]
[break]
[entry attr=PaymentType width=10]
[entry attr=PaymentAmount width=10]
[entry attr=AmountDue width=10]
[epilogue]
```

Figure 1.1: Automatically generated QddbScript

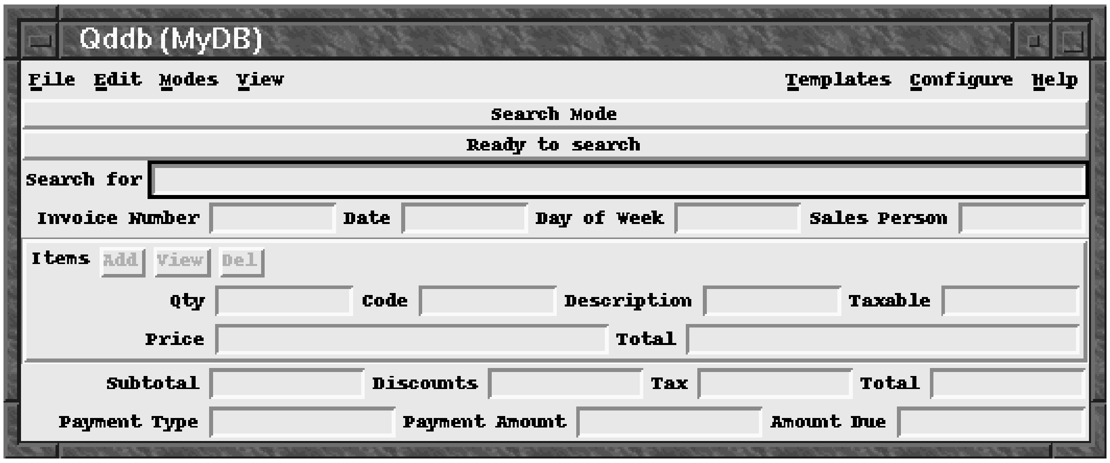


Figure 1.2: nxqddb screen **before** modifying QddbScript

```
[prologue]
[menubar]
[options autoalign]
[entry attr=Number width=15]
[entry attr=Date width=10]
[entry attr=DayOfWeek width=10]
[break]
[entry attr=SalesPerson width=40]
[break]
[frame attr=Items border=3 relief=groove]
    [entry attr=Items.Qty width=10]
    [entry attr=Items.Code width=10]
    [break]
    [entry attr=Items.Desc width=40]
    [break]
    [entry attr=Items.Taxable width=2 noexpand]
    [entry attr=Items.Price width=10]
    [entry attr=Items.Total width=10]
[end frame]
[break]
[entry attr=Subtotal width=10 side=right noexpand]
[break]
[entry attr=Discounts width=10 side=right noexpand]
[break]
[entry attr=Tax width=10 side=right noexpand]
[break]
[entry attr=Total width=10 side=right noexpand]
[break]
[entry attr=PaymentAmount width=10 side=right noexpand]
[radiobutton attr=PaymentType width=10 side=right noexpand
    value="Mastercard" value="Visa" value="AmEx" value={Discover}
]
[break]
[entry attr=AmountDue width=10 side=right noexpand]
[epilogue]
```

Figure 1.3: Customized QddbScript

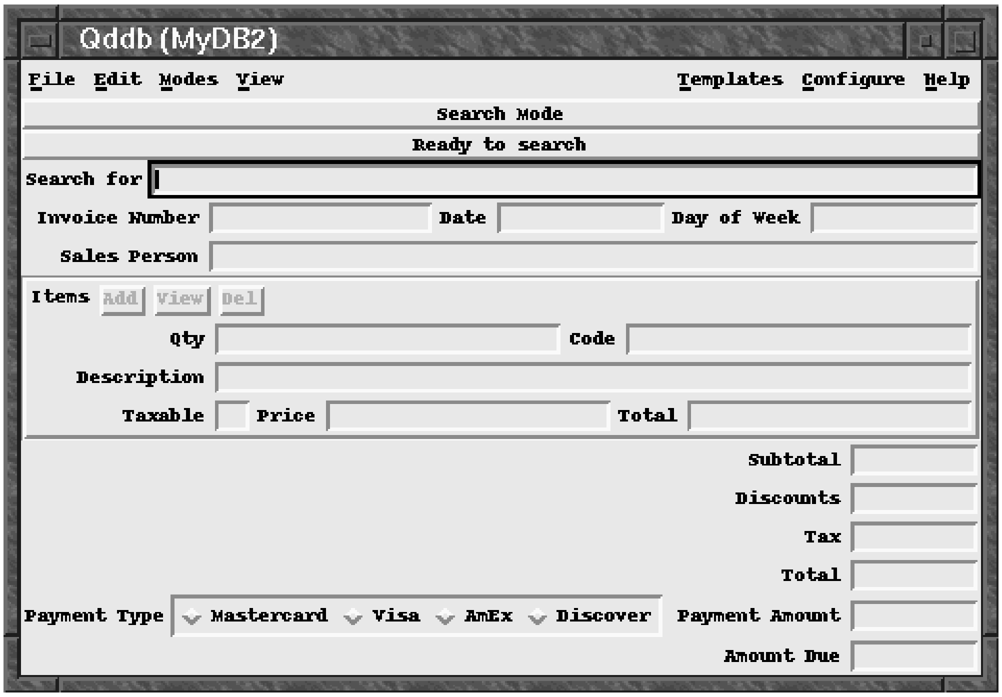


Figure 1.4: nxqddb screen **after** modifying QddbScript

Chapter 2

The Language

QddbScript has a format similar to HTML[2], but it uses square brackets (`[]`) instead of `<>` to denote the beginning and end of a command. All command names are case insensitive. Some QddbScript commands such as *frame* require end markers:

```
[frame]
[end frame]
```

Other commands, such as *entry*, require certain options for proper operation of the script. Unrecognized or invalid options are always ignored, as well as any part of a line after a pound sign (#) not within a string. Semicolons that are not within a string are also ignored, so you can freely place them anywhere you like to add clarity to your script.

The current general commands and their options are listed in Table 2.1. Strings may be one of the following forms:

`A[A-Za-z0-9\.]+`	Alphanumeric strings (periods ok) starting with alpha
"any valid string \" goes here"	Double quoted string
{any valid string \} goes here}	String within curly braces

The options *attr*, *relief*, and *side* require certain values. *attr* requires a valid attribute name for the Qddb schema (such as **Items.Qty**). *relief* must be one of **flat**, **raised**, **sunken**, or **groove**. The default relief is **flat**. *side* must be one of **left**, **right**, **top**, or **bottom**. The default side is **top** for frames and **left** for entries, text, labels, and radiobuttons.

2.1 Prologue, menubar, and epilogue

Every QddbScript requires a prologue, menubar, and epilogue statement. You may insert custom code before or after any of these with the *code* statement. A typical customized QddbScript looks like this:

Option	Commands							
	frame	entry	text	radiobutton	label	notebook	tab	code
attr=STRING (attr name)	opt	req	req	req	no	no	no	no
value=STRING	no	no	no	req	no	no	no	req
label=STRING	opt	opt	opt	opt	opt	no	opt	no
labelwidth=INT (chars)	opt	opt	opt	opt	opt	no	no	no
width=INT (chars)	no	opt	opt	opt	opt	no	no	no
height=INT (chars)	no	no	opt	opt	opt	no	no	no
relief=STRING	opt	opt	opt	opt	opt	no	no	no
border=INT (pixels)	opt	opt	opt	opt	opt	no	no	no
side=STRING	opt	opt	opt	opt	opt	no	no	no
padx=INT (mm)	opt	opt	opt	opt	opt	no	no	no
pady=INT (mm)	opt	opt	opt	opt	opt	no	no	no
ipadx=INT (mm)	opt	opt	opt	opt	opt	no	no	no
ipady=INT (mm)	opt	opt	opt	opt	opt	no	no	no
underline=INT (char #)	no	no	no	no	no	no	opt	no
expand	opt	opt	opt	opt	opt	no	no	no
noexpand	opt	opt	opt	opt	opt	no	no	no
end marker	req	no	no	no	no	req	req	no

Table 2.1: QddbScript commands and options

```
[code value={
    lappend auto_path /usr/local/lib/myapp
    MyApp:Init        ;# Load up some initial variables
}]
[prologue             ;# sets up Fx library and banner
    config="myapp"    ;# default configuration dir ~/.myapp_config
    bannergif="/usr/local/lib/myapp/myapp.gif"
    bannerxbm="/usr/local/lib/myapp/myapp.xbm"
    bannertitle="MyApp Version $myapp_version" ;# myapp_version set by MyApp:Init
]
[menubar]  ;# sets up the default menubar
# this is a comment
# Frames, entries, text, and radiobuttons go here
[epilogue] ;# finishes setup and displays the screen
```

Notice that the *code* statement sets up the Tcl *auto_path* variable for autoloading application-specific Tcl procedures such as *MyApp:Init*. The value of a *code* statement is inserted verbatim into the compiled QddbScript's Tcl code. See Section 3.1 for more information on inserting your own Tcl functions, menu items, and other special effects.

If the *prologue* statement does not contain the options listed above, they will default to the same values that the standard nxqddb(1) application uses.

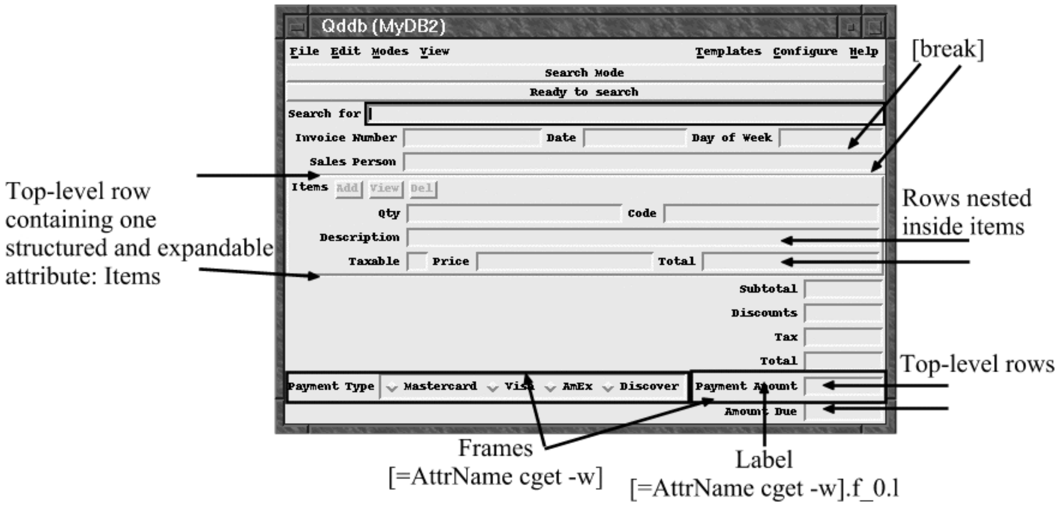


Figure 2.1: Breakdown of GUI produced with the modified QddbScript

2.2 Fx Widgets

QddbScript requires that all Fx-based widgets reside between `[prologue]` and `[epilogue]` statements. Automatically generated QddbScript uses fixed-width entries (four per row). Structured and expandable attributes are indented so the structure is easily seen by users.

The basic structure of a QddbScript consists of a set of rows (frames), each of which may contain a set of columns or nested rows. Each column in a row may contain entries, text boxes, or radiobuttons bound directly to attributes. For example, the GUI produced by our modified QddbScript consists of rows and columns as shown in Figure 2.1.

A single cell is represented by an *Fx_Entry*. An Fx_Entry is a Tk frame (say $w) consisting of a subframe (*$w.f_0*), a label (*$w.f_0.l*), buttons if the attribute is expandable (*$w.f_0.b_add*, *$w.f_0.b_view*, and *$w.f_0.b_del*), and an entry (*$w.e*)[1].

2.2.1 Frames

Frames are typically used with a corresponding structured attribute to provide a logical appearance for the user, although you can build frames simply for their structuring properties. The typical use of a frame is:

```
[frame attr=Items border=3 relief=groove side=top]
    [entry attr=Items.Qty width=10]
```

[1]Radiobuttons are contained inside a text entry (*$w.e.t*) and are numbered from left to right 0..*n* like *$w.e.t.r0* *$w.e.t.r1* etc.

```
    [entry attr=Items.Code width=10]
    [entry attr=Items.Desc width=10]
    [entry attr=Items.Taxable width=10]
    [break]
    [entry attr=Items.Price width=10]
    [entry attr=Items.Total width=10]
[end frame]
```

A frame command forces QddbScript to start a new frame at the current level. If the frame is attributed (that is, it has an **attr=** option), it is associated with a structured attribute from the Qddb Schema and will automatically contain Add, View, and Del buttons if it is also expandable.

Inside the frame, the subattributes are separated by `[break]` statements. Each `[break]` statement forces QddbScript to begin a new row (an inner frame) inside the frame. For example, if we want to see the above frame's contents in three rows instead of two, we simply insert another break statement:

```
[options autoalign]
[frame attr=Items border=3 relief=groove side=top]
    [entry attr=Items.Qty width=10]
    [entry attr=Items.Code width=10]
    [break]
    [frame ipadx=10 ipady=10 noexpand] ;# do some odd padding
        [entry attr=Items.Desc width=10]
        [entry attr=Items.Taxable width=10]
    [end frame]
    [break]
    [entry attr=Items.Price width=10]
    [entry attr=Items.Total width=10]
[end frame]
```

By default, the frames and entries will expand to fill the width of the screen. If the option *autoalign* is on (via an `[options autoalign]` statement) the first entry in each frame will have a fixed-width label ($w.f_0.l) so the entries will line up on the form. If for some reason you do not want a frame to expand to fill the width of its parent, add the option *noexpand* to the frame statement as shown above.

Each attributed frame creates a command based on the name of the attribute and an '=' prefix. For example, the frame above creates a Tcl command `=Items` that may be used to address the special features and options of the Fx_Frame (see Fx_Frame(n)).

The *side* option specifies where the frame will be physically located within its parent frame. By default, frames are packed at the top of their parent, effectively stacking them one after the other. If you need two side-by-side frames, you might do something like this:

```
[options autoalign]
# ...

[frame attr=Items border=3 relief=groove side=top]
    [options noautoalign] ;# turn off autoalign
    [frame side=left]
        [entry attr=Items.Qty labelwidth=12 side=top]
        [entry attr=Items.Code labelwidth=12 side=top]
    [end frame]
    [frame side=left]
        [entry attr=Items.Desc labelwidth=12 side=top]
        [entry attr=Items.Taxable labelwidth=12 side=top]
    [end frame]
    [options autoalign] ;# turn autoalign back on
    [break]
    [entry attr=Items.Price width=10]
    [entry attr=Items.Total width=10]
[end frame]
```

2.2.2 Entry types

There are three basic types of entry fields: entry, text, and radiobutton. The *entry* type is a single line entry field and accepts the *width=INTEGER* option to set its initial width. The *text* type is a multi-line entry field with a scrollbar that accepts the *width=INTEGER* and *height=INTEGER* options. Since the *radiobutton* type is contained in a text box, it also accepts both the width and height options, as well as an arbitrary number of *value=STRING* options to set the values of the radiobuttons' labels. All entry types **require** an *attr=* option.

Examples:

```
#...
[break]
[text attr=Items.Desc width=40 height=4 side=top]
[break]
[entry attr=Items.Price width=10 noexpand]
#...
[radiobutton attr=PaymentType height=1
    value="Visa" value="M/C" value="AmEx"
    value="Cash" value="Check"
]
```

2.2.3 Notebooks and tabs

Qddb Schemas are often quite large and sometimes contain more attributes than we can fit on the screen at once. Notebooks are very convenient for organizing the entries into logical sections. Each notebook has a set of tabs. Each tab contains a frame and a set of entries. You may have multiple notebooks in a single application, but the notebooks cannot be nested.

The basic structure of a notebook is as follows:

```
[notebook]
    [tab label=Tab1 underline=3]
    [end tab]
    [tab label=Tab2 underline=3]
    [end tab]
    [tab label=Tab3 underline=3]
    [end tab]
[end notebook]
```

You may place arbitrary statements inside each pair of `[tab]` and `[end tab]` statements, but you may not place statements inside a notebook and outside the tabs. For example, the following QddbScript excerpt is invalid.

```
[notebook]
    [tab label=Tab1 underline=3]
        ;# more entries
    [end tab]
    [entry attr=Items.Price]   ;# ERROR!!
    [tab label=Tab2 underline=3]
        [entry attr=Items.Qty] ;# OK!
        ;# more entries
    [end tab]
    [tab label=Tab3 underline=3]
        ;# more entries
    [end tab]
[end notebook]
```

Qtime (`http://www.hsdi.com/qddb/applications`) is a good example of how notebook tabs can be used to condense the size of an application's screen. Figure 2.2 shows a screenshot of Qtime. If you have purchased a Qddb binary or source license, the full source code to this application is available to you free of charge.

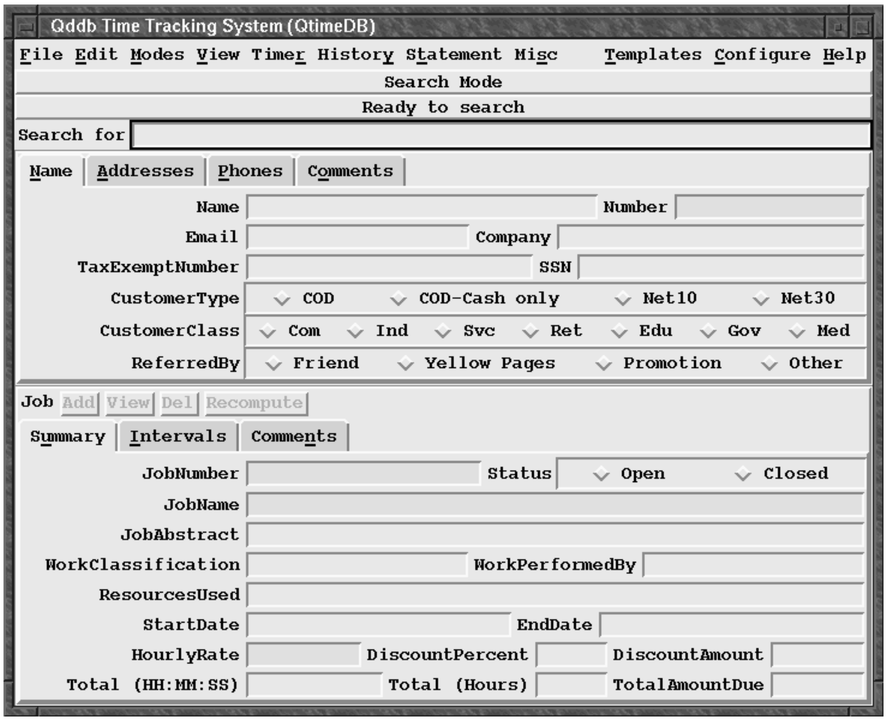


Figure 2.2: The Qtime application's screen

Chapter 3

Custom Programming

Designing an Fx application's form used to consume a significant amount of development time. Now that designing the form is relatively simple, a developer's time can be spent adding custom functions to the application. This chapter details the interface between QddbScript and Qtcl/Fx.

3.1 The *code* statement

In its simplest form, a QddbScript is self-contained and requires no external code. However, if you want to add custom features to your interface or access the full capabilities of Fx, you must insert some links to Tcl code inside the QddbScript.

As we saw in Section 2.1, you can add your own library to the Tcl *auto_path* variable as follows:

```
[code value={
    lappend auto_path /usr/local/lib/myapp
}]
[prologue]
# ...
```

After appending your application's library directory to the auto_path, you can freely call Tcl procedures indexed in that directory:

```
[code value={
    MyAppFunction1 arg1 arg2
    MyAppFunction2
}]
```

The code values are inserted verbatim into the compiled Fx script at the global level and are clearly surrounded by comments:

```
;# [USER-INSERTED CODE]

    MyAppFunction1 arg1 arg2
    MyAppFunction2

;# [END USER-INSERTED CODE]
```

3.2 Accessing Fx Widgets and Tk frames

QddbScript uses the following Tk window hierarchy for its main toplevel window:

Fx Widget	Access Command	Tk path
Fx_Menubar	=menubar	.menubar
Fx_Frame	=AttrName	All under the frame .fx
Fx_Entry	=AttrName	All under the frame .fx

You can configure settings or retrieve the current settings for any Fx widget using *configure* and *cget* commands:

```
if {[menubar cget -auto_clear] == 0} {
    ;# auto-clear is turned off
}
=Items.Qty configure -labelfg red -afterview {
}
bind [=Items.Qty GetEntry] <FocusOut> {
    ;# recalculate, etc.
}
```

3.2.1 Fx_Frame options

Most Fx_Frame options are automatically set by QddbScript, so you probably don't want to modify them. There are some options, however, that applications need to change on a regular basis. These include:

-beforeadd list	evaluate *list* before adding an instance
-afteradd list	evaluate *list* after adding an instance
-beforedelete list	evaluate *list* before deleting an instance
-afterdelete list	evaluate *list* after deleting an instance
-beforechange list	evaluate *list* before bringing a new instance into view
-afterchange list	evaluate *list* after bringing a new instance into view
-addtoend 0|1	add all new instances to the end of the instance list
-copy_instances list	when a new instance is created, copy the last one in view
-showbutton_add 0|1	show "Add" button if attribute is expandable
-showbutton_view 0|1	show "View" button if attribute is expandable
-showbutton_del 0|1	show "Del" button if attribute is expandable
-anchor `e|w|s|n|ne|se|sw|nw`	anchor for sub-frame $w.f_0
-labelfg color	label foreground color

The *before* and *after* options are important because they allow the programmer to insert code (the "list") around the execution of any standard Fx action. For example, you may want to recalculate an invoice total after an item is deleted:

```
=Items configure -afterdelete {
    MyApp:RecalculateTotal
} -addtoend 1
```

You may (re)configure any of these options at any time[1].

3.2.2 Fx_Entry options

Fx_Entry widgets are highly configurable and allow all the same options as the Fx_Frame widget[2] plus:

-entryfg color	entry foreground color
-read_only `0|1`	if set to 1, the field cannot be modified by user
-mandatory `0|1`	if set to 1, field must be entered
-unique `0|1`	if set to 1, field must be a unique value
-userconfig `0|1`	if set to 0, user cannot reconfigure field options
-regexp_search `0|1`	if set to 0, do not parse searches as regular expressions
-numeric_search `0|1`	if set to 0, do not parse searches as numeric values
-date_search `0|1`	if set to 0, do not parse searches as date values
-range_search `0|1`	if set to 0, do not parse searches as ranges

The `*_search` options are effective only during simple searching; expert searches can still perform any type of search on any field. You typically want to disable regular expressions, dates, ranges, and numeric searches on fields that contain data such as "123-1343-1234" and have separators set to "" in the Schema (inventory item numbers, etc.)

[1]Some options listed in the Fx_Frame manual page, such as *-w* and *-attr*, should probably never be reconfigured after creation by QddbScript.

[2]Attributes bound to an Fx_Entry can also be expandable and require Add/View/Del buttons.

Example:

You might want to prevent a sales person from modifying line totals in your invoices.

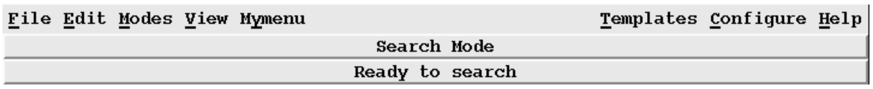
```
=Items.Price configure -mandatory 1 -userconfig 0
=Items.Total configure -read_only 1 -mandatory 1 -userconfig 0
```

3.3 Custom menubuttons and actions

The Fx_Menubar widget occurs exactly once per application. When you specify the `[menubar]` command, QddbScript creates a new Tcl command called "menubar" and the menubar is packed into a Tk frame called ".menubar". You can insert Tk code directly after the `[menubar]` command to create a new menubutton and associated Tk menu:

```
# ...
[menubar]
[code value={
    menubutton .menubar.mymenu -text Mymenu \
        -underline 1 -menu .menubar.mymenu.menu
    menubar configure -auxbuttons .menubar.mymenu
    menu .menubar.mymenu.menu -tearoff 0
    .menubar.mymenu.menu add command -label "Hello" \
        -underline 0 -command {
        tk_dialog .dialog "Hello World" "Hello World" \
                info 0 Ok
    }
}]
# ...
```

Of course, the code is much cleaner if you put it into a procedure that can be called from your application's library. Using the code above, our Invoices database now has a new menubutton called "Mymenu" with a <Alt-y> accelerator:



Mymenu consists of one menu item called *Hello*. You can add as many menubuttons and menuitems as you like. Note that your menubar customizing procedure should contain a call similar to this:

```
menubar configure -auxbuttons {
    .menubar.menu1
    .menubar.menu2
    .menubar.menu3
}
```

The argument to *-auxbuttons* must contain all auxilliary menubuttons or they will not be packed properly in the menubar.

3.3.1 Standard menus

When adding your own menubuttons, you must take care not to add a Tk name that conflicts with one of the standard menubuttons. The standard Fx menubuttons are:

Standard Fx Menubuttons			
Label	Accelerator	Tk menubutton	Tk menu
File	`<Alt-f>`	.menubar.file	.menubar.file.menu
Edit	`<Alt-e>`	.menubar.edit	.menubar.edit.menu
Modes	`<Alt-m>`	.menubar.modes	.menubar.modes.menu
View	`<Alt-v>`	.menubar.view	.menubar.view.menu
Templates	`<Alt-t>`	.menubar.templates	.menubar.templates.menu
Configure	`<Alt-c>`	.menubar.configure	.menubar.configure.menu
Help	`<Alt-h>`	.menubar.help	.menubar.help.menu

You may add menubuttons using any Tk path under `.menubar` other than those listed above.

3.3.2 QddbScript variables

QddbScript sets several variables you can use in your `[code]` statements or procedures:

qddbscript(schema)	Currently open Qddb schema descriptor
qddbscript(trans_name)	Current restricted directory for saving templates, etc.
qddbscript(search_entry)	Tk entry widget for the "search for:" box
qddbscript_frame	Current Tk frame widget (row)

The `qddbscript` array is for convenience and internal use within QddbScript. The most useful variable for application developers is `qddbscript_frame`. `qddbscript_frame` refers to the current innermost non-Fx frame (i.e. row). For example, you may want to set special options on a row's frame:

```
[entry attr=Items.Qty]
[entry attr=Items.Code]
```

```
[code value={
    $qddbscript_frame configure -bd 10 -bg blue
}]
[break]
```

This code sequence will put a 10-pixel blue outline around the Items.Qty and Items.Code entries.

3.4 Using Tcl libraries

You may append any Tcl libraries you like to the current list "`auto_path`", however, you must append the library path before calling any procedures in that library. Typically you should do this before the `[prologue]`, but you can append to the `auto_path` anywhere you like.

```
[code value={
    lappend auto_path /usr/local/lib/mylibrary1 \
        /usr/local/lib/mylibrary2
}]
[prologue]
```

If you are building your own Tcl libraries for use with QddbScript, you must remember to build a *tclIndex* file for each library before use:

```
$ echo 'auto_mkindex /usr/local/lib/mylibrary1 *.tcl' \
    | /usr/local/bin/qtclsh
$ echo 'auto_mkindex /usr/local/lib/mylibrary2 *.tcl' \
    | /usr/local/bin/qtclsh
```

You can always build a *Makefile* for your application and insert commands like this for easy installation (see Section 3.6 for a detailed example).

3.5 Common Fx settings and functions

Most Fx settings and functions are built into Fx_Menubar. Almost all the menu buttons, menu items, and modes can be modified for any particular application at runtime. Table 3.1 lists the common settings and Table 3.2 lists the common Fx_Menubar functions used by most applications. For a complete list, see the Fx_Menubar(n) manual page.

Options can be set or retrieved with *configure* and *cget*, respectively. For example:

```
menubar configure -auto_clear 1 -force_readonly_mode 1
puts "autoclear: [menubar cget -auto_clear]"
puts "readonly:  [menubar cget -force_readonly_mode]"
```

Fx_Menubar functions are called using the “menubar” command:

```
proc MyApp:GetCode {{code 42}} {
    global gv_attr

    menubar SearchModeProc ;# go to Search Mode
    menubar ClearProc      ;# clear array
    set gv_attr(Items.Code) $code
    menubar SearchProc     ;# search and display results
}
```

Option	Action
-auto_clear `0|1`	autoclear entries for search/add modes
-auto_save `0|1`	automatically save tuples when necessary
-force_readonly_mode `0|1`	don't allow change mode on any tuple
-menubar `0|1`	show the menubar
-searchfor `0|1`	show the generic "search for:" entry
-modebar `0|1`	show the mode bar
-statusbar `0|1`	show the status bar
-auxbuttons `list`	set extra menubuttons
-beforechangemode `list`	eval list before entering "Change Mode"
-afterchangemode `list`	eval list after entering "Change Mode"
-beforeaddmode `list`	eval list before entering "Add Mode"
-afteraddmode `list`	eval list after entering "Add Mode"
-beforesearchmode `list`	eval list before entering "Search Mode"
-aftersearchmode `list`	eval list after entering "Search Mode"
-beforereadonlymode `list`	eval list before entering "Read-only Mode"
-afterreadonlymode `list`	eval list after entering "Read-only Mode"
-beforetupledelete `list`	eval list before entering deleting a tuple
-beforesave `list`	eval list before saving a tuple
-aftersave `list`	eval list after saving a tuple
-beforequit `list`	eval list before starting exit checks
-afterquit `list`	eval list after starting exit checks
-exitproc `list`	eval list as the exit procedure
-beforeclear `list`	eval list before clearing a tuple
-afterclear `list`	eval list after clearing a tuple
-tearoffs `0|1`	use Tk tearoffs for menus
-afterpost_file `list`	eval list after posting File menu
-afterpost_edit `list`	eval list after posting Edit menu
-afterpost_modes `list`	eval list after posting Modes menu
-afterpost_view `list`	eval list after posting View menu
-afterpost_templates `list`	eval list after posting Template menu

Table 3.1: Common Fx_Menubar options

Function	Action
DisplayLastSearch	display last search results
KillLastSearch	kill last search results
GetLastSearch	returns qddb_table as returned by *qddb_rows select -format table*
SetLastSearch list	list must be a qddb_table as returned by *qddb_rows select -format table*
LastSearchWasNull	returns 1 if last search returned no entries, 0 otherwise
SearchModeProc	go into search mode
ChangeModeProc index	go into change mode using tuple at *index* row of the last search results table
ReadOnlyModeProc index	go into read-only mode using tuple at *index* row of the last search results table
AddModeProc	go into add mode
SearchProc	search using contents of the global attribute array *gv_attr*
ClearProc	clear the screen in search or add mode
SaveProc	save the currently viewed tuple
QuitProc	exit the application

Table 3.2: Common Fx_Menubar functions

3.6 Putting it all together

After you have a feeling for the Fx interfaces, it's easy to put together nice applications in a jiffy. As an example, let's add some custom features to our little Invoices application.

One useful function for invoices might be an auto-increment invoice number with a prefix, say "HPC". We would like to add a unique invoice number each time we add a new record, so we might add the following to our QddbScript:

```
;# ...
[code value={
    ;# implements autoincrement by 123, starting at 1000000, with
    ;# a prefix of "HPC".
    proc MyInvoiceNumber {prefix min increment} {
        global gv_attr

        set filename [qddb_schema path [menubar cget -schema]]/.nextinv
        if {[catch {open $filename r+} fd] != 0} {
            set next $min
            if {[catch {open $filename w+} fd] != 0} {
```

```
                Fx:Dialog .dialog "Cannot open $filename"
                return
            }
            qddb_util flock $fd LOCK_EX
        } else {
            qddb_util flock $fd LOCK_EX
            set next [read $fd]
        }
        set gv_attr(Number) "$prefix$next"
        incr next $increment
        seek $fd 0 start ;# rewind to beginning of file
        puts -nonewline $fd $next
        flush $fd
        qddb_util flock $fd LOCK_UN
        close $fd
    }
    menubar configure -afteraddmode {
        MyInvoiceNumber HPC 1000000 123
    }
}]
;# ...
[epilogue]
```

Next, we might like to recalculate subtotal, tax, and total each time these fields change. For this purpose, it makes sense to perform the calculations each time the focus leaves one of the important fields:

```
;# ...
[code value={
    proc Recalculate {} {
        global fx:mode_variable myview

        if {"${fx:mode_variable}" != "Add Mode"} {
            return
        }
        set tuple [menubar cget -tuple]
        set view [qddb_view define $tuple {
            {Items.Qty myview(Items.Qty)}
            {Items.Taxable myview(Items.Taxable)}
            {Items.Price myview(Items.Price)}
```

```
            {Items.Total myview(Items.Total)}
            {Subtotal myview(Subtotal)}
            {Tax myview(Tax)}
            {AmountDue myview(AmountDue)}
            {PaymentAmount myview(PaymentAmount)}
            {Discounts myview(Discounts)}
            {Total myview(Total)}
        }]
        set max [qddb_instance maxnum $view Items]
        set myview(Tax) 0.00
        set myview(Subtotal) 0.00
        set myview(Total) 0.00
        for {set i 1} {$i <= $max} {incr i} {
            qddb_instance switch $view Items $i
            catch {
                set myview(Items.Total) [expr \
                    $myview(Items.Qty) * $myview(Items.Price) \
                ]
        set myview(Subtotal) [expr $myview(Subtotal) \
                    + $myview(Items.Total)]
                if {"$myview(Items.Taxable)" != "N"} {
                    set myview(Tax) [format "%.2f" [expr $myview(Tax) \
                        + ($myview(Items.Total) * 0.06)]]
                }
            }
        }
        catch {
            set myview(Total) [expr $myview(Subtotal) + \
                $myview(Tax) - $myview(Discounts)]
        }
        set myview(AmountDue) [expr $myview(Total) - $myview(PaymentAmount)]
        ;# force GUI to re-read variables
        qddb_view refresh [menubar cget -view]
        update idletasks
        qddb_view delete $view
        unset myview
    }
    bind Entry <FocusOut> {+Recalculate}
    bind Radiobutton <FocusOut> {+Recalculate}
}]
```

```
;# ...
[epilogue]
```

You can retrieve a full copy of this QddbScript and the associated Schema from `ftp://ftp.hsdi.com/pub/qddb/sources/QddbScript-example.tar.gz`. To try it out, simply unpack the archive in a newly created database directory.

```
qnewdb Invoices
cd Invoices
gunzip -c ~/QddbScript-example.tar.gz | tar -xvf -
nxqddb .
```

Enjoy!

The Qddb Team

Bibliography

[1] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, 1994. ISBN 0-201-63337-X.

[2] Sandia National Laboratories. *HTML Reference Manual.*