Maemo Diablo Source code for the asynchronous GConf example

Training Material

February 9, 2009
## Contents

1 Source code for the asynchronous GConf example  
   1.1 gconf-listener/gconf-key-watch.c ........................................ 2  
   1.2 gconf-listener/Makefile ..................................................... 7
Chapter 1

Source code for the asynchronous GConf example

1.1  gconf-listener/gconf-key-watch.c

/*
 * A simple CLI program that uses the GObject-ified GConf library to
 * wait for GConf key changes (from within GMainLoop). Will run
 * until terminated manually.
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 * Depends on glib and gconf libraries (when building).
 * One thing to note about this example as that even if it is used to
 * track GConf key/value changes, it might not be the optimal in all
 * cases. When you are following many keys, using a single
 * notification callback will force you to use strcmp() to find out
 * exactly which key changed and act accordingly.
 * The strcmp-model is used in this example, as we only have two
 * keys to track. The other option would be to have separate
 * callbacks for _each_ key that is tracked, and that way you could
 * forego the strcmp within each callback.
 * The drawback in the latter approach is that it might increase the
 * size of the generated code (since each callback would have to be
 * a separate function). The latter approach however is the one
 * that is more commonly used in real programs, since they have
 * multiple key/values to track.
 * Please use gconftool-2 to change the values so that you
 * will see some output from this program (see below for examples).
 * Also, you will need to create the entries before-hand, as the
 * program is very simplistic, and will abort if the values are not
 * present when it's run. A proper application would populate the
 * missing keys with sane defaults. A demonstration of this can be
 * found in the GConf example in "maemo Application Development".
 */
To set the values from CLI, you can use the following commands:

```bash
$ gconftool-2 -s --type string /apps/Maemo/platdev_ex/connection /btcomm0
$ gconftool-2 -s --type string /apps/Maemo/platdev_ex/connectionparams 9600,8,N,1
```

You can use the provided Makefile with target 'primekeys' to achieve the same effect as above (there are also other targets useful for testing in the Makefile).

You can also use the same commands to change the values (while running this program).

To check the existing values from CLI:

```bash
$ gconftool-2 -R /apps/Maemo/platdev_ex
```

```c
#include <glib.h>
#include <gconf/gconf-client.h>
#include <string.h>
/* strcmp */

As per maemo Coding Style and Guidelines document, we use the `/apps/Maemo/-prefix`.

NOTE: There is no central registry (as of this moment) that you could check that your application name doesn’t collide with other application names, so caution is advised! */

#define SERVICE_GCONF_ROOT `/apps/Maemo/platdev_ex`

/* We define the names of the keys symbolically so that we may change them later if necessary, and so that the GConf "root directory" for our application will be automatically prefixed to the paths. */

#define SERVICE_KEY_CONNECTION \ SERVICE_GCONF_ROOT `/connection`
#define SERVICE_KEY_CONNECTIONPARAMS \ SERVICE_GCONF_ROOT `/connectionparams`

/**
 * Callback called when a key in watched directory changes.
 * Prototype for the callback must be compatible with GConfClientNotifyFunc (for ref).
 *
 * It will find out which key changed (using strcmp, since the same callback is used to track both keys) and the display the new value of the key.
 *
 * The changed key/value pair will be communicated in the entry parameter. userData will be NULL (can be set on notify_add [in main]). Normally the application state would be carried within the userData parameter, so that this callback could then modify the view based on the change. Since this program does not have a state, there is little that we can do within the function (it will abort the program on errors though).
 */
static void keyChangeCallback(GConfClient* client, guint cnxn_id, GConfEntry* entry, gpointer userData) {
    /* This will hold the pointer to the value. */
    const GConfValue* value = NULL;
    /* This will hold a pointer to the name of the key that changed. */
```
const gchar* keyname = NULL;
/* This will hold a dynamically allocated human-readable representation of the changed value. */
gchar* strValue = NULL;

g_print(PROGNAME": keyChangeCallback invoked.
");

/* Get a pointer to the key (this is not a copy). */
keyname = gconf_entry_get_key(entry);

/* It will be quite fatal if after change we cannot retrieve even the name for the gconf entry, so we error out here. */
if (keyname == NULL) {
    g_error(PROGNAME": Couldn’t get the key name!
");
    /* Application terminates. */
}

/* Get a pointer to the value from changed entry. */
value = gconf_entry_get_value(entry);

/* If we get a NULL as the value, it means that the value either has not been set, or is at default. As a precaution we assume that this cannot ever happen, and will abort if it does.
NOTE: A real program should be more resilient in this case, but the problem is: what is the correct action in this case? This is not always simple to decide.
NOTE: You can trip this assert with 'make primekeys', since that will first remove all the keys (which causes the CB to be invoked, and abort here). */
g_assert(value != NULL);

/* Check that it looks like a valid type for the value. */
if (!GCONF_VALUE_TYPE_VALID(value->type)) {
    g_error(PROGNAME": Invalid type for gconfvalue!
");
}

/* Create a human readable representation of the value. Since this will be a new string created just for us, we’ll need to be careful and free it later. */
strValue = gconf_value_to_string(value);

/* Print out a message (depending on which of the tracked keys change. */
if (strcmp(keyname, SERVICE_KEY_CONNECTION) == 0) {
    g_print(PROGNAME": Connection type setting changed: [%s]
", strValue);
} else if (strcmp(keyname, SERVICE_KEY_CONNECTIONPARAMS) == 0) {
    g_print(PROGNAME": Connection params setting changed: [%s]
", strValue);
} else {
    g_print(PROGNAME": Unknown key: %s (value: [%s])
", keyname, strValue);
}

/* Free the string representation of the value. */
g_free(strValue);

/* Utility to retrieve a string key and display it.

4
/* (Just as a small refresher on the API.) */

class dispStringKey(GConfClient* client,
const gchar* keyname) {

/* This will hold the string value of the key. It will be
dynamically allocated for us, so we need to release it ourselves
when done (before returning). */
gchar* valueStr = NULL;

/* We're not interested in the errors themselves (the last
parameter), but the function will return NULL if there is one,
so we just end in that case. */
valueStr = gconf_client_get_string(client, keyname, NULL);
if (valueStr == NULL) {
  g_error(PROGNAME " : No string value for \%s. Quitting\n", keyname);
  /* Application terminates. */
}

valueStr = gconf_client_get_string(client, keyname, NULL);

/* Normally one would want to use the value for something beyond
just displaying it, but since this code doesn't, we release the
allocated value string. */
g_free(valueStr);
}

int main (int argc, char** argv) {
  /* Will hold reference to the GConfClient object. */
  GConfClient* client = NULL;
  / * Initialize this to NULL so that we'll know whether an error
     occurred or not (and we don't have an existing GError object
     anyway at this point). */
  GError* error = NULL;
  /* This will hold a reference to the mainloop object. */
  GMainLoop* mainloop = NULL;
g_print(PROGNAME "::main Starting.
");  
/* Must be called to initialize GType system. The API reference for  
gconf_client_get_default() insists.  
NOTE: Using gconf_init() is deprecated! */  
g_type_init();  
/* Create a new mainloop structure that we'll use. Use default  
context (NULL) and set the 'running' flag to FALSE. */  
mainloop = g_main_loop_new(NULL, FALSE);  
if (mainloop == NULL) {  
g_error(PROGNAME ":: Failed to create mainloop!\n");  
}  
/* Create a new GConfClient object using the default settings. */  
client = gconf_client_get_default();  
if (client == NULL) {  
g_error(PROGNAME ":: Failed to create GConfClient!\n");  
}  
g_print(PROGNAME "::main GType and GConfClient initialized.\n");  
/* Display the starting values for the two keys. */  
dispStringKey(client, SERVICE_KEY_CONNECTION);  
dispStringKey(client, SERVICE_KEY_CONNECTIONPARAMS);  
/* Register directory to watch for changes. This will then tell  
GConf to watch for changes in this namespace, and cause the  
'value-changed' signal to be emitted. We won't be using that  
mechanism, but will opt to a more modern (and potentially more  
scalable solution). The directory needs to be added to the  
watch list in either case.  
*  
* When adding directories, you can sometimes optimize your program  
* performance by asking GConfClient to preload some (or all) keys  
* under a specific directory. This is done via the preload_type  
* parameter (we use GCONF_CLIENT_PRELOAD_NONE below). Since our  
* program will only listen for changes, we don't want to use extra  
* memory to keep the keys cached.  
*  
* Parameters:  
* - client: GConf-client object  
* - SERVICEPATH: the name of the GConf namespace to follow  
* - GCONF_CLIENT_PRELOAD_NONE: do not preload any of contents  
* - error: where to store the pointer to allocated GError on  
*   errors. */  
gconf_client_add_dir(client,  
SERVICE_GCONF_ROOT,  
GCONF_CLIENT_PRELOAD_NONE,  
&error);  
if (error != NULL) {  
g_error(PROGNAME ":: Failed to add a watch to GCCliet: %s\n",  
error->message);  
/* Normally we'd also release the allocated GError, but since  
this program will terminate on g_error, we won't do that.  
Hence the next line is commented. */  
/* g_error_free(error); */  
/* When you want to release the error if it has been allocated,
or just continue if not, use g_clear_error(&error); /*

*/

}  
g_print(PROGNAME " :main Added " SERVICE_GCONF_ROOT ",\n");

/* Register our interest (in the form of a callback function) for 
any changes under the namespace that we just added.

Parameters:
- client: GConfClient object.
- SERVICEPATH: namespace under which we can get notified for 
changes.
- gconf_notify_func: callback that will be called on changes.
- NULL: user-data pointer (not used here).
- NULL: function to call on user-data when notify is removed or 
GConfClient destroyed. NULL for none (since we don’t 
have user-data anyway).
- error: return location for an allocated GError.

Returns:
guint: an ID for this notification so that we could remove it 
later with gconf_client_notify_remove(). We’re not going 
to use it so we don’t store it anywhere. */

gconf_client_notify_add(client, SERVICE_GCONF_ROOT, 
    keyChangeCallback, NULL, NULL, &error);

if (error != NULL) {
    g_error(PROGNAME " : Failed to add register the callback: %s\n", 
        error->message);
    /* Program terminates. */
}

g_print(PROGNAME " :main CB registered & starting main loop\n");

/* Start the main loop. */

if (error != NULL) {
    g_error(PROGNAME " : Failed to add register the callback: %s\n", 
        error->message);
    /* Program terminates. */
}

/* Release the mainloop object. */

/* Release the gconfclient since we're done now. This would also 
run the freeing function that we could have passed on notify_add-
registration above, but since we didn't have a need for one, 
no free'ers will be run here (for us at least). */

return 0;

Listing 1.1: gconf-listener/gconf-key-watch.c

1.2 gconf-listener/Makefile
# This is the Makefile for the GConf key watch example.  
# The default target will build the application.  
# There are also two special targets:  
# primekeys : populate the GConf parameters with default values  
# clearkeys : remote all of the application GConf keys (recursively)  
# dumpkeys : list all keys for this application (recursively)  
# In order to use the targets, the GConf daemon will need to be  
# running in your system (af-sb-init.sh). The daemon is running  
# at all times on the device.  
# Define a variable for this so that the GConf root may be changed  
gconf_root := /apps/Maemo/platdev_ex  
# pkg-config packages that we'll need  
pkg_packages := glib-2.0 gconf-2.0  
PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))  
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))  
# Additional flags for the compiler:  
# -g : Add debugging symbols  
# -Wall : Enable most gcc warnings  
ADD_CFLAGS := -g -Wall  
# Combine user supplied, additional, and pkg-config flags  
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)  
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)  
# Default targets  
targets := gconf-key-watch  
.all: $(targets)  
# We define a define (PROGNAME) so that we can (or rename the program  
# later if necessary).  
gconf-key-watch: gconf-key-watch.c  
	${CC} $(CFLAGS) -DPROGNAME="$@" $^ -o $@ $(LDFLAGS)  
# This will setup the keys into default values.  
# It will first do a clear to remove any existing keys.  
primekeys: clearkeys  
gconftool-2 --set --type string \  
$(gconf_root)/connection btcomm0  
gconftool-2 --set --type string \  
$(gconf_root)/connectionparams 9600,8,N,1  
# Remove all application keys  

clearkeys:  
	@gconftool-2 --recursive-unset $(gconf_root)  
# Dump all application keys  
dumpkeys:  
	@echo Keys under $(gconf_root):  
	@gconftool-2 --recursive-list $(gconf_root)  
clean:  
	$(RM) $(targets)