Developing widgets on Maemo 5

Juha Järvi

Software designer

Foreca

ForecaWeather widget for Maemo 5

Introduction

- The talk is based on experiences from one Maemo 5 widget
  - Likely to apply to other Fremantle widgets
  - Our widget shows rarely changing location-based data
- Nothing presented is the official opinion of Foreca or Nokia
  - No guarantees offered for correctness!
- Everything will be on these slides, available for download later

Graphic Design 1/4

- Keep the widget small
  - Fit 3 widgets on one desktop, comfortably spaced
- A plain design fits different themes better
  - Dark gray translucent background is used in many widgets
- Use colors from the theme for text and other elements
  - Single-color symbols can be colored programmatically

Graphic Design 2/4

Show only essential info

Use large graphics and fonts with short text

- User should understand everything important at a glance

Leave space for translations in different languages!

- Avoid multiple text columns, words may be very long
- Text could expand 3 times from English version

Graphic Design 3/4

If the widget has buttons, make them large

- Best if only thumbs are needed for use
- Accidental clicks shouldn’t have annoying results

Avoid complicated interactive elements

- Put controls in a separate dialog or fullscreen program
- Whole widget can be like a button to open more content

Graphic Design 4/4

- Separate fullscreen mode has many benefits
  - More controls can easily fit on the screen
  - Tabs and scrolling can be used to flip through information
  - Widget content can be shown even bigger for car use etc.
  - Allows more creative freedom for users and developers
  - Widget settings are easier to find than in desktop menu

Architecture 1/5

- Widgets are libraries loaded by the home screen application
  - They need to be written more carefully than other apps
  - Crashes are a worse experience than with full programs
  - Memory leaks can accumulate over many days
  - Draining the battery must be avoided
- Moving complicated operations to a separate program is safer

Architecture 2/5

- Create a separate stand-alone fullscreen program
  - Crashes and hangs are easier for users to recover from
  - Complex features and processing are safer to implement
  - Debugging on a PC is simpler
  - Software is useful even if desktop is already full of widgets!
- Problem is making the parts communicate

Architecture 3/5

- Having a widget and a fullscreen part introduces problems
  - Settings changes in either one should affect the other
    - D-BUS message sent to the full program launches it
  - System events are easier to receive in a stand-alone app
  - Network transfers must work with just one part present
    - Both parts together also need to be synchronized

Architecture 4/5

- Messaging issues are solved by a third, background program
  - Let’s call it “controller”
- Widget and fullscreen program message it when started
  - Controller is then started by D-BUS if not running
- Keeps track of which of the other parts are running
- Propagates messages from one to the other
- Handles network transfers and system events

Architecture 5/5

- Design the architecture for stability and low power usage
- Pay special attention to the widget and controller
  - Keep their source code simple
    - Small enough to read through before important releases
  - Carefully verify they can’t hang or crash
  - As a general rule they should always be idle
- Fullscreen program is less critical

Widget 1/1

Widget on its own can simply stay in GTK main loop

Only reacts to mouse events or D-BUS messages

- Everything is handled in small simple subroutines

No timers or anything that runs a long time

Have the controller do any online data retrieval

Foreca’s widget is under 2k total lines of C

- Only uses D-BUS, GTK and Hildon libs

Controller 1/4

Controller waits for messages in GLib main loop

Listens to system sleep events to stop all activity

Possibly tracks network connection and GPS status changes

- Use ACWP instead of GPS or turn it on only briefly!

One timer can be running to fetch data updates or similar

- Activating only every 30 minutes or less frequently
- Turned off if the device goes to sleep

Controller 2/4

Automatically updates data from the network only if:

- Device woke up, got online or other event occurred
- AND the current data is invalid
  - Over 30 minutes old or user chose to view new data
  - If updating based on location, user moved by several km
- AND the screen is on and program is visible on screen

Controller 3/4

Splitting activities into helper programs can go further

Simple to write and test a program to do one HTTP download

- Handling many simultaneously takes more effort
- Also third-party libraries can get unstable over time

Same goes for any input/output that isn’t a constant stream

- Just wait forever until all data has arrived, blocking IO

Controller simply needs to start and manage the helpers

Controller 4/4

Controller forks and executes helper programs, uses pipes

A list is kept of all running helpers

Special timer fires every second while any are running

- If one has been running too long, kill it
- If device goes to sleep, kill them all!
- When a helper finishes, send results over D-BUS
- Timer stops when all have finished

Benefits 1/2

- The multiple processes may sound like wasting resources
  - However this isn’t an application program or web browser
  - Widgets should transfer little data very rarely
  - Most of the time helpers are not running
  - The widget is less complex and uses less resources!
    - Foreca’s controller is under 1k lines of C

Benefits 2/2

- User experience is also improved by the modular design
  - Device memory usage is lower
  - Less problems from hangs or crashes
    - The user interface isn’t affected
    - Code for the small modules is verifiable by reading
    - Few interactions between parallel processes help testing

Implementation 1/2

The current API requires using C on some level

For rapid development of many widgets there are better tools

Lua is a good recommendable scripting language

- We have written bindings for Diablo widgets
- Straightforward to use, little extra C code needed (2k lines)
- Binary is about 100k, even widget graphics take more space

Implementation 2/2

More controversially, avoid Autoconf

- 5k lines of C and 30k lines of scripts to compile it = wrong

Write a makefile for the widget project manually

- It’s only a couple dozen lines and only needs to call gcc

Keep it simple: if you want to port, plan for a partial rewrite

- A widget is mainly UI and every device has a new UI style
- Widget sources don’t need to compile for several devices

Testing 1/4

- Test constantly on the device itself if possible
  - Gives more reliable results of real-world behavior
- A script on the device can download new binary files
  - Two key presses after every build, or automated with SSH
  - Force widget reload by deleting and restoring config entry
    - It’s inside ~/.config/hildon-desktop/home.plugins
    - (Done in the controller also works for widget resizing)

Testing 2/4

Types of tests to use regularly:

- Smoke test after every build (does it still run on the device)
- Automatic test cases for controller and its helpers
- Ask next person you see to try out the user interface
- Remember to switch locales
  - For fullscreen program setting LC_* in xterm is enough
  - Get a draft German translation early, long words there!

Testing 3/4

- Special for a mobile/embedded device, test power usage!
  - Along with not crashing, this is top priority for a widget
  - Run “powertop -t60” on device
    - Need to be root, install rootsh package first
  - Execute a planned 60-second test
  - Check total wakeups

Testing 4/4

Test and compare different power usage scenarios

- Widget on desktop or not in use
- Device online using WLAN or 3G
- GPS enabled or disabled if relevant
- Flip the device lock side switch to sleep for some seconds
- Other widgets or apps present using the same services

Widget must not cause extra wakeups without good reason!

Summary

Keep widget small and text easy to read

Put complex features in a separate program

Split code into short independent processes

Carefully test stability and power usage

Thanks!

Questions or comments?