

Maemo Diablo Reference Manual for maemo 4.1

December 22, 2008

Contents

1	Introduction	11
2	Glossary	15
3	Development Environment	17
3.1	Introduction	17
3.2	Maemo Software Development Kit	17
3.3	Installing SDK	21
3.3.1	Prerequisites	21
3.3.2	Automatic Install of Scratchbox	21
3.3.3	Manual Install of Scratchbox	22
3.3.4	Automatic Install of Maemo SDK	22
3.3.5	Manual Install of Maemo SDK	23
3.3.6	Manual Install of ARMEL Target	25
3.4	Testing SDK Installation	26
3.4.1	Testing Scratchbox	26
3.4.2	Writing GUI Hello World	28
3.4.3	Running GUI Hello World	32
3.4.4	Starting Virtual X Server (Xephyr)	33
3.4.5	Directing Client to Virtual Server	33
3.4.6	Starting Application Framework	34
3.4.7	Running Hello World in Application Framework	35
3.5	Setting up USB Networking	36
3.5.1	Introduction	36
3.5.2	Setting up Environment and Device	36
3.5.3	Testing Connection	37
3.6	CPU Transparency	38
3.6.1	Setting up Host Linux PC Environment	38
3.6.2	Setting up Scratchbox ARMEL Target	39
3.6.3	Setting up Device for CPU Transparency	39
3.6.4	Testing CPU Transparency	39
3.7	Using Linux Flasher Tool	40
4	GNU Build System	42
4.1	Introduction	42
4.2	GNU Make and Makefiles	42
4.2.1	Simplest Real Example	43
4.2.2	Anatomy of Makefile	46

4.2.3	Default Goal	47
4.2.4	On Names of Makefiles	47
4.2.5	Questions	48
4.2.6	Adding Make Goals	48
4.2.7	Making One Target at a Time	49
4.2.8	PHONY Keyword	49
4.2.9	Specifying Default Goal	50
4.2.10	Other Common Phony Goals	51
4.2.11	Variables in Makefiles	51
4.2.12	Variable Flavors	51
4.2.13	Recursive Variables	52
4.2.14	Simple Variables	53
4.2.15	Automatic Variables	54
4.2.16	Integrating with Pkg-Config	55
4.3	GNU Autotools	56
4.3.1	Brief History of Managing Portability	57
4.3.2	GNU Autoconf	58
4.3.3	Substitutions	62
4.3.4	Introducing Automake	64
4.3.5	Checking for Distribution Sanity	69
4.3.6	Cleaning up	69
4.3.7	Integration with Pkg-Config	70
5	Architecture	72
5.1	Introduction	72
5.1.1	Key Components	72
5.1.2	Development Environment	73
5.2	Software Layers	73
5.2.1	Operating System Layer and Bootup	73
5.2.2	System Libraries	74
5.2.3	System Services	74
5.2.4	Hildon Framework	74
5.2.5	Applications	74
5.3	Software Decomposition View	75
5.3.1	Software Components	75
5.3.2	Kernel	82
5.3.3	Flash Partitioning	83
5.3.4	Application Framework	83
5.3.5	Base Distribution	84
5.4	Run Time View	86
5.4.1	Overview	86
5.4.2	Components	86
5.4.3	Application Activation	87
5.4.4	Application Termination	89
5.4.5	State Saving and Background Killing	89
5.4.6	X Window System	91
5.4.7	Window Management	93
5.4.8	Misbehaved Applications	94
5.5	Major APIs	95
5.6	Maemo Compared to Desktop Linux Distributions	95

6	Application Development	98
6.1	Introduction	98
6.2	Typical Maemo GUI Application	99
6.3	Hildon Desktop	101
6.4	Writing Maemo GUI Applications	101
6.4.1	Overview of Maemo GUI Applications	101
6.4.2	Basic Hildon Layouts	101
6.4.3	Windows	103
6.4.4	Menus	105
6.4.5	Toolbars	107
6.4.6	HildonFindToolbar	109
6.5	Other Hildon Widgets	113
6.5.1	HildonFileChooserDialog	113
6.5.2	HildonColorChooser	115
6.5.3	HildonFontSelectionDialog	116
6.5.4	HildonFileDetailsDialog	117
6.5.5	HildonBanner	119
6.6	Using Maemo Input Mechanism	121
6.6.1	Touchscreen (Mouse)	121
6.6.2	Context Sensitive Menu	122
6.6.3	Hardware Keys	122
6.6.4	Maemo's Gtk Accelerator and Mnemonics	124
6.6.5	Maemo Text Input Methods	126
6.6.6	Application Layout Considerations	126
6.7	Writing Control Panel Applets	126
6.7.1	Functions	127
6.7.2	Building Applet	128
6.7.3	The .desktop File	128
6.8	Writing Hildon Desktop Plug-ins	128
6.8.1	Introduction	128
6.8.2	Task Navigator Plug-ins	129
6.8.3	Home Plug-ins	133
6.8.4	Status Bar Plug-ins	135
6.8.5	Control Panel Plug-ins	138
6.8.6	Making Plug-in Icons Visible	139
6.8.7	Creating Makefiles and Package for Applet	139
6.9	Integrating Applications to Maemo Framework	144
6.9.1	Getting Application to Task Navigator Menu	144
6.9.2	LibOSSO Library	146
6.9.3	Application Settings	150
6.10	MIME Types Mapping	151
6.10.1	New MIME Type with OSSO Category Extension	152
6.10.2	What is OSSO Category	152
6.10.3	Updating Platform Databases	153
6.10.4	Registering MIME Type with Package	153
6.11	Writing Application from Scratch	154
6.11.1	Introduction	154
6.11.2	Creating Application File Structure	154
6.11.3	Coding Application Main	156
6.11.4	User Interface	158

6.11.5	Localization	161
6.11.6	Adding Application to Menu	163
6.11.7	Link to Maemo Menu	163
6.11.8	Adding Help	163
6.11.9	Packaging Application	164
6.12	Help Framework	165
6.12.1	Creating Help File	165
6.12.2	Adding Help Context Support into Application	167
6.12.3	Distributing Example Application	169
7	Using Generic Platform Components	170
7.1	Introduction	170
7.2	File System - GnomeVFS	171
7.3	Message Bus System - D-Bus	179
7.3.1	D-Bus Basics	179
7.3.2	LibOSSO	191
7.3.3	Using GLib Wrappers for D-Bus	211
7.3.4	Implementing and Using D-Bus Signals	233
7.3.5	Asynchronous GLib/D-Bus	248
7.3.6	D-Bus Server Design Issues	258
7.4	Application Preferences - GConf	265
7.4.1	Using GConf	265
7.4.2	Using GConf to Read and Write Preferences	267
7.4.3	Asynchronous GConf	274
7.5	Alarm Framework	281
7.5.1	Alarm Events	282
7.5.2	Managing Alarm Events	286
7.5.3	Checking for Errors	287
7.5.4	Localized Strings	288
7.6	Usage of Back-up Application	290
7.6.1	Custom Back-up Locations	290
7.6.2	After Restore Run Scripts	291
7.7	Using Maemo Address Book API	292
7.7.1	Using Library	292
7.7.2	Accessing Evolution Data Server (EDS)	294
7.7.3	Creating User Interface	299
7.7.4	Using Autoconf	303
7.8	Clipboard Usage	304
7.8.1	GtkClipboard API Changes	304
7.8.2	GtkTextBuffer API Changes	304
7.9	Global Search Usage	305
7.9.1	Global Search Plug-ins	306
7.10	Writing "Send Via" Functionality	308
7.11	Using HAL	309
7.11.1	Background	310
7.11.2	C API	312
7.12	Certificate Storage Guide	314
7.12.1	Digital Certificates	314
7.12.2	Certificates in Maemo Platform	316
7.12.3	Creating Own Certificates with OpenSSL	316

7.12.4	Maemo Certificate Databases	317
7.12.5	Creating Databases	317
7.12.6	Importing Certificates and Keys	318
7.12.7	Sample Program for Searching and Listing Certificates	320
7.12.8	Deleting Certificates	321
7.12.9	Validating Certificate Files	322
7.12.10	Exporting Certificates	322
7.13	Extending Hildon Input Methods	322
7.13.1	Overview	322
7.13.2	Plug-in Features	323
7.13.3	Interaction with Main User Interface	331
7.13.4	Component Dependencies	334
7.13.5	Language Codes	334
8	Using Multimedia Components	335
8.1	Introduction	335
8.2	Getting Started with Multimedia	337
8.2.1	Simple GStreamer Example	337
8.2.2	Plug-in Development	338
8.2.3	Installing OGG Vorbis	338
8.2.4	Deployment	339
8.3	Camera API Usage	340
8.3.1	Camera Hardware and Linux	340
8.3.2	Camera Manipulation in C Language	341
8.4	Using Games Start-up Screen	345
8.4.1	Application Functionality	346
8.4.2	Integration	346
8.4.3	Creating Game-Specific Plug-in	348
9	Using Connectivity Components	356
9.1	Introduction	356
9.2	Maemo Connectivity	357
9.2.1	Connectivity Subsystem	358
9.2.2	Internet Connectivity Daemon	364
9.2.3	LibConIC Library	366
9.2.4	Bluetooth Libraries	374
9.2.5	Connectivity UI	377
9.2.6	Samba Network Shares	386
9.2.7	Location	387
9.3	Implementing Custom Connection Managers	389
9.3.1	Connection Manager Implementation Examples	389
9.3.2	Connection Manager and Connections	389
9.3.3	Channels and Channel Types	390
9.3.4	Additional Connection Interfaces	391
9.4	Using STUN in Applications	392
9.4.1	Network Address Translation (NAT)	393
9.4.2	NAT Problems	394
9.4.3	NAT Peer-to-Peer Circumvention Techniques	394
9.4.4	STUN, TURN and ICE Protocols	395
9.4.5	NAT Transversal API in Maemo	396

9.4.6	Example: P2P Client	396
9.4.7	Signaling Server	404
9.4.8	STUN and Relay Servers	405
10	Customizing User Interface	407
10.1	Introduction	407
10.2	Theme	408
10.2.1	Quick Start with Hildon Theme Tools	410
10.3	How to Create Debian Package for Custom Data Files	412
10.4	Customizing Icons	412
10.5	Editing Theme Layouts Manually	413
10.6	Adding New Fonts	415
10.7	How to Customize Device Sounds	416
10.8	Creating Custom Background Image (Wallpaper)	416
10.9	Third Party Theme Packages and Theme Tools	418
10.10	References	418
11	Kernel Guide	419
11.1	Prerequisites	419
11.2	Getting Kernel Sources	420
11.3	Configuring Source Tree and Compiling Kernel	420
11.4	Changing Default Kernel Configuration	421
11.5	Configuring and Compiling Kernel Modules	421
11.6	Flashing Kernel	422
12	Porting Software	423
12.1	Introduction	423
12.2	Porting Existing Applications to Maemo 4.x	424
12.2.1	Introduction	424
12.2.2	Application File Structure	424
12.2.3	Requirements and Configure Changes	425
12.2.4	Basic Porting	426
12.2.5	User Interface Changes	427
12.2.6	State Saving	431
12.2.7	Network Changes	438
12.2.8	Integration to Menu	439
12.2.9	Application Packaging	440
12.3	Maemo Localization	442
12.3.1	Overview	442
12.3.2	Localization	442
12.3.3	Localizing Application	443
12.3.4	Easing Extraction of Strings	443
13	Packaging, Deploying and Distributing	446
13.1	Creating Debian Packages	447
13.2	Making Application Packages	449
13.2.1	Prerequisites	449
13.2.2	Application Manager	449
13.2.3	Packaging	449
13.2.4	General	450

13.2.5	Dependencies	450
13.2.6	Sections	450
13.2.7	Icons	451
13.2.8	Installation and Removal Policy	451
13.2.9	Feedback from Maintainer Scripts	451
13.2.10	Removing or Upgrading Running Applications	452
13.2.11	Utilities to Use in Maintainer Scripts	452
13.2.12	Controlling Installation	453
13.2.13	Signing Package	456
13.3	Deploying Packages	456
13.3.1	Installing Application Packages to SDK	456
13.3.2	Installing Application Packages to Device	456
13.3.3	Red Pill Mode	456
14	Debugging	458
14.1	Introduction	458
14.2	Maemo Debugging Guide	458
14.2.1	Pre-Requisites	459
14.2.2	General Notes on Debugging	459
14.2.3	Using Gdb Debugger	460
14.2.4	Debugging Hildon Desktop Plug-ins	472
14.2.5	Running Out of Memory During Debugging in Device	475
14.2.6	Valgrind Debugger	476
14.2.7	Other Debugging Tools	480
14.3	Making a Debian Debug Package	481
14.3.1	Creating DBG Packages	481
14.3.2	Using and Installing DBG Packages	484
14.3.3	For Further Reading	484
15	Quality Considerations	485
15.1	Quality Awareness	486
15.2	Secure Software Design	495
15.2.1	Overview	495
15.2.2	Elements of Secure Software Development	495
15.2.3	Threat Analysis	496
15.2.4	Security Testing	500
15.2.5	Conflicts of Interest?	501
15.2.6	Tools and Resources	502
15.2.7	Maemo Security Policies	503
15.3	Maemo Coding Style and Programming Guidelines	504
15.3.1	Maemo Coding Practices	505
15.3.2	Basics of Good Programming	506
15.3.3	Maemo Conventions	508
15.3.4	Power Saving	515
16	maemo.org	517
16.1	Documentation	517
16.2	Support	517
16.3	Bug Reporting	517
16.4	Project Hosting	518

16.5 Roadmap and News	518
---------------------------------	-----

Preface

Legal notice

Copyright ©2007-2008 Nokia Corporation. All rights reserved.

Nokia and maemo are trademarks or registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. This document is provided for informational purposes only. Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights. Nokia Corporation retains the right to make changes to this material at any time, without notice.

Licenses

This document is licensed under the [GNU Free Documentation License, version 1.2](#). Permission is granted to copy, distribute and/or modify documentation under the terms of the GNU Free Documentation License, version 1.2 or any later version published by the Free Software Foundation. Please refer to the copyright notice in each piece of documentation for Invariant Sections, Front-Cover Texts or Back-Cover Texts, if any, as defined in the License.

The code examples copyrighted by Nokia Corporation and included to this document are licensed to you under MIT-style license. License text for that MIT-style license is as follows:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Chapter 1

Introduction

This document strives to give an overall picture of the maemo platform to developers willing to bring their applications to the Nokia Internet Tablets. These mobile devices run a Linux-based operating system on ARM architecture. The maemo software development kit (SDK) is currently provided natively for desktop GNU/Linux distributions, such as Debian and Ubuntu. The SDK can also be used in other operating systems through a virtual environment.

The main focus is to explain the basics necessary to create and port maemo-compatible software. It is simple to get started, even if concepts such as cross-compilation or touchscreen user interface design are not familiar. Those who are familiar with GNU/Linux, GTK+ toolkit and C programming language will feel right at home.

This document is also a good starting point for software development for any platform or device related to the GNOME Mobile family. As well as giving an introduction to the new frameworks, libraries and tools provided by maemo, it explains the overall design concepts for this new generation of Internet-capable devices.

History and Philosophy

Maemo uses known open desktop frameworks to enable easy software portability and familiarity. This strategy is quite different compared to many Linux-based media players and phones that are closed, or require special development tools. About 90% of the maemo platform is open, and the majority of that comes from upstream open source projects. The rest that is closed consists of e.g. parts of the user interface and device drivers, owned either by Nokia or third-party providers.

One important philosophy is to enable easy development and hacking. Developers can make modifications to the platform, e.g. to introduce their own kernel modules. Nokia also hosts an active open source maemo community around the platform (maemo.org).

Internet Tablet Overview

The devices are smaller than a laptop, larger than a PDA, and quite lightweight. Some of them (e.g. Nokia N810) have a small keyboard, and all of them have a stylus and a touch-sensitive screen. The stylus-driven GUI will cause some design challenges later on, since software will need to be designed with this in mind. There is also a possibility of using an on-screen keyboard with the stylus, including handwriting recognition and a predictive input system to aid the user. In all devices, there is a limited set of hardware buttons available for applications.

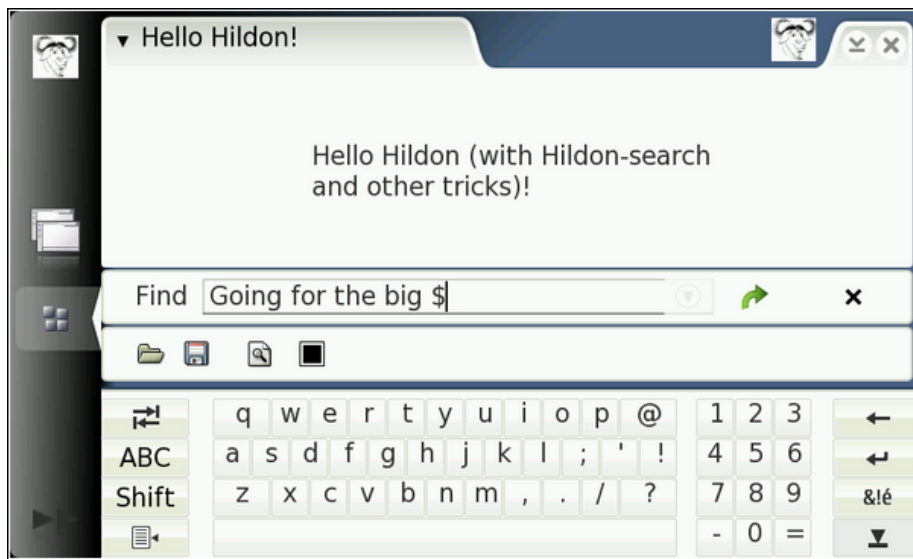


Figure 1.1: The Virtual Keyboard (VKB)

As programmers appreciate knowing a bit about the fundamentals of the devices for which they program, table 1.1 presents a short list of the most important components.

N800	N810	N810 WiMax Edition
an 800x480 pixel, 225 pixels-per-inch (PPI) wide-screen touch screen display with 16-bits per pixel color depth		
Hardware buttons with a layout optimized for Web surfing		
Virtual Keyboard	Small slide-out keyboard (& VKB)	
Wi-Fi (802.11b/g)		WiMAX 802.16e / 2.5GHz
External GPS via Bluetooth	Integrated GPS (external also supported)	
1500 mAh battery		
3.5 mm stereo audio out socket (works also as mic input on N800/N810)		
Built-in VGA resolution webcam		
USB 2.0 port (in target mode by default)		
128 MiB of RAM		
256 MiB flash memory with JFFS2 file system		
Two memory card slots, SD, MicroSD, MiniSD, MMC, and RS-MMC (some types with extender).	One memory card slot, compatible with MiniSD and MicroSD (with extender).	
Bluetooth 2.0		
TI OMAP 2420 multi-core processor with maximum clock frequency of 400 MHz, with:		
<ul style="list-style-type: none">• TMS320C55x DSP logic (Backward compatibility with the 54x-series)• ARM1136 core ("ARMv6") with an MMU (Backward compatibility with ARM926)		

Table 1.1: Internet Tablet components

The USB port normally acts as a USB target, although the direction can be reversed, and the device can be the USB host (i.e. initiator). The port is not capable of providing USB power, so an external power feed is necessary. This allows various usage scenarios, when the R&D mode is enabled on a device. The default version of Internet Tablet software runs in target mode only.

Some noteworthy points about the hardware and software:

- There is not a lot of RAM (compared to a "modern PC"), and the memory is shared between all the applications that are running at any given time.
- The system runs a modified Linux kernel 2.6 (omap-port).
- The system library is GNU libc 2, meaning that most software can be ported without too much effort (even networking software).
- To conserve battery power, one needs to be careful with application core logic (loops, delays, timeouts, threads, etc.)

- There is no hardware acceleration for graphics operations (2D or 3D).
- The built-in flash contains approximately 64 MiB of shipped software. 192 MiB available to be shared between applications covers also data those applications use. Using tablet as memory stick is not so common use case.
- The built-in flash uses a file system specifically designed for flash memory, and contains transparent compression and decompression. This means that sometimes optimizing for space requirements is not sensible. Compressing an image as a **.gif** is not very good idea, as it would have been compressed anyhow. However, the RS-MMC card uses FAT/VFAT file system. The compression rates may vary, and if space conservation is important for an application, it is advisable to test the specific use scenario properly.
- There is some support for Java acceleration in the ARM core, but this is not utilized, since there is no supported JVM to execute Java code.

N.B. The above feature list holds for the "end user" version of the software that is shipped with Internet Tablets.

Chapter 2

Glossary

ABI Application Binary Interface, providing object code level compatibility.

API Application Programming Interface, providing source code level compatibility.

applet A small application that integrates to *Hildon Desktop*.

ARMEL A name that e.g. Debian uses for the little endian ARM EABI (*ABI* for the ARM architecture).

devkit Part of the *maemo SDK* that contains software development tools. The SDK contains multiple devkits, e.g. doctools devkit.

Hildon Application framework used in the *maemo platform*. Developed by Nokia and based on GNOME/GTK+ technologies, currently in the process of becoming an upstream project in gnome.org.

Hildon Desktop The main user interface component of the maemo release Chinook, rewrite of *maemo desktop*.

Internet Tablet Product category for Internet-optimized mobile devices with touchscreen. The term was coined by Nokia, but is nowadays used more widely to include other devices.

initfs Initial file system used as the root file system during Linux kernel boot e.g. for hardware initialization (contains kernel modules and utilities for initializing them). Mounted after boot to `/mnt/initfs`.

maemo Software platform for mobile devices developed by Nokia, based on GNU/Linux and GNOME/GTK+ technologies. It includes proprietary components to make it work on the Nokia Internet Tablets.

maemo.org Developer community web site maintained by Nokia, main point of reference for open source and third-party developers in general.

maemo SDK Software Development Kit to create and port applications to the maemo platform using a PC.

Nokia Internet Tablet OS *maemo platform* + proprietary applications packaged to an official device image provided by Nokia.

OSSO Open Source Software Operations, Nokia organization developing and integrating software for Internet Tablets.

rootfs Root file system on the device.

rootstrap Part of the SDK that contains selected software components from rootfs. Rootstrap is the root file system of a target inside Scratchbox.

Sardine An experimental distribution based on Hildon for maemo, primarily of interest for developers wishing to test "bleeding edge" features that are being developed for future releases of maemo.

toolchain Part of the SDK that contains ARM cross compilation tools, such as compiler and linker.

Maemo SDK Releases

Mistral maemo 2.0 release. Corresponds to the Nokia Internet Tablet SE 2006 version 2.01.2006.26-8.

Scirocco maemo 2.1 release, including mainly bugfixes and some other enhancements. Corresponds to Nokia Internet Tablet SE 2006 version 2.2006.39- 14.

Gregale maemo 2.2 release (bugfixes and enhancements)

Bora maemo 3.x releases. corresponds to Internet Tablet OS releases "1.2006.47-20", "2.2006.51-6" (maemo 3.0), "3.2007.10-7" (maemo 3.1) and "4.2007.26-8"+"4.2007.38-2" (maemo 3.2)

Chinook maemo 4.0.x releases, corresponds to Internet Tablet OS releases "1.2007.44-4" (maemo 4.0) and "2.2007.51-3" (maemo 4.0.1)

Diablo maemo 4.1.x releases, corresponds to Internet Tablet OS releases 4.2008.23-14 (maemo 4.1) and 4.2008.36-5 (maemo 4.1.1)

Fremantle maemo 5.x releases, corresponds to Internet Tablet OS release "x.2009.xx-x" (maemo 5.x)

Chapter 3

Development Environment

3.1 Introduction

The following code examples are used in this chapter:

- [helloworld.c](#)
- [gtk_helloworld-1.c](#)

The development environment for maemo running on the desktop is called *maemo SDK*[72]. It will only install and run on a Linux operating system. Supported Linux distributions for maemo SDK are currently Debian and Ubuntu, but installing maemo SDK is also possible for other distributions. On other operating systems such as Windows, a VMWare image[74] can be used to provide a working Linux environment.

3.2 Maemo Software Development Kit

The maemo SDK creates a sandboxed maemo development environment on a GNU/Linux desktop system largely built on a tool called *Scratchbox*[90]. In most ways, this environment behaves like the operating system on the device, but with added development tools. This means that the development process is very similar to a normal desktop GNU/Linux, and the kinks of embedded development, such as cross-compiling, are handled transparently by Scratchbox.

Scratchbox is a specially packaged "sandbox" environment, providing the necessary tools and also isolating the development efforts from the real Linux system. Scratchbox also makes it easy to perform cross-compiling, which means building the software into a binary format that is executable in the target device.

The name "Scratchbox" comes from "Linux from scratch" + "chroot jail" (sandbox). This also tells something about its implementation and intended use. While working inside Scratchbox, programs will be running in a changed root environment (chroot). In Linux systems, it is possible to change the part of file paths that a process will see. Scratchbox uses this mechanism on start to switch its root directory (/) to something other than the real root. This is part of the isolation technique used. Because of this, the environment is called

a sandbox, a private area where it is possible to play around without disturbing the environment, and without all the mess that real sand would cause. The other parts of the isolation technique are library call diversions (using LD_PRELOAD), wrapping of compiler executables and other commands.

Scratchbox:

- Is a software package to implement development sandboxes (for isolation)
- Contains easy-to-use tools to assist cross-compilation
- Supports multiple developers using the same development system
- Supports multiple configurations for each developer
- Supports executing target executables on the hardware target, via a mechanism called sbrsh
- Supports running non-native binaries on the host system via instruction set emulators ([Qemu](#) is used [87]).

Besides these main features, it is possible to develop own software packages that can be installed and used inside a Scratchbox environment. Scratchbox also includes some integration for Debian package management, so that once the source files are set up correctly and a couple of configuration files have been written, binary distribution packages can be created for various architectures (similar to .msi-files in Windows, or .rpm-files in Fedora Core, RHEL and SUSE). These tools are also used to provide the environment with a packaging database, so that other development packages can be installed over the Internet when needed (by using standard Debian package management tools).

The Internet Tablet also uses a similar packaging system, and this means that packages built using Scratchbox and the SDK can be installed on the real device.

Scratchbox is licensed under the GPL and it is open for outside contributions. For an in-depth coverage on Scratchbox capabilities please see [90].

This material discusses only the Scratchbox capabilities that are necessary to use the maemo SDK.

Hardware architectures

Maemo SDK supports two architectures and has an environment for them both. *x86* is used for native performance and better tool support through native execution without the need for emulation. *ARMEL* is used for working with the actual device's architecture. Both have their advantages and roles in maemo development. It is important to understand that maemo SDK actually provides these two environments as preconfigured *targets* inside a working Scratchbox installation. This is explained in detail later.

Generally the *x86* environment is used in active development, because it provides practically the same performance as normal GNU/Linux applications. Also, although the underlying architecture is different from the actual device, programs usually behave exactly as they would, when compiled and run on ARMEL. As stated before, many tools are available only for the *x86* environment.

Working means that application also functions properly and not just run, the next step is to compile it for the ARMEL architecture. The process for

compilation and packaging is exactly the same as in x86, albeit a bit slower, because some of the required software is emulated. The developer needs not to concern themselves with cross-compilation. This is the main reason maemo SDK use Scratchbox in the first place.

The applications compiled in ARMEL environment can be run straight on the device. Additionally, it is possible to run some of the applications inside the ARMEL environment in maemo SDK. This is possible, because of the automatic emulation maemo SDK provides. The emulation is not complete, and things like multithreading will cause problems, so actual testing must be performed on the device.

Using emulation for the whole development process may not sound ideal because of the effect on performance. That is why even when in ARMEL environment, native performance is achieved with Scratchbox by internally using the host computer's tools without emulation, when possible. For example, compilation on ARMEL environment is actually performed by an x86-ARMEL cross-compiler, but Scratchbox hides the details so that the developer can execute GCC like on any GNU/Linux system.

Scratchbox In-Depth

Scratchbox is maemo SDK's cross-compiling environment. The default Scratchbox installation works as-is under most conditions, but some details are good to know for more specialized usage.

The *target* inside Scratchbox contains a root file system that is being worked on. When a new target inside Scratchbox is created, a *toolchain* must be specified for it. Using this toolchain, applications are built for the target. Examples of a *target* are X86 and ARMEL, which are provided by the maemo SDK on top of Scratchbox.

Host tools are native to the host provided for convenience and speed. They are always preferred over target tools and transparently for example cross-compile applications to the target architecture. Host tools consist of devkits and *toolchains*.

A toolchain provides the minimal set of tools for compiling binaries for the target. One and only one toolchain must be selected for every Scratchbox target.

CPU transparency methods take care of running the applications on an emulator, target device or directly on the host transparently to the user. The available CPU transparency methods come from a special *devkit* called *cpustransp*. For each of maemo SDK's pre-defined targets, a CPU transparency method is selected and defined.

A *toolchain* is a collection of tools used to produce binaries for the target environment. In addition to a compiler (*gcc*), it contains a linker (*ld*) and other *binutils*, such as *strip*, *objdump* and *strings*.

A *devkit* is a collection of tools native to the host. A toolkit can be selected or disabled for a target. An example of a devkit is doctools devkit, which provides tools (like doxygen) for building documentation.

A *rootstrap* is a root file system for the target device. Maemo SDK provides root file systems for both targets (X86 and ARMEL) inside Scratchbox. Note that the user's home directory is shared for all targets. The */tmp* directory is shared for all targets and also with the host.

From Scratchbox's point of view, maemo SDK is a set of preconfigured

targets and *root file systems*. One set is provided for both X86 and ARMEL architectures on top of a working Scratchbox installation.

More Scratchbox information can be found at Scratchbox's web site[90].

Development on Maemo SDK

The maemo SDK provides all of its development tools inside Scratchbox. Also the Hildon Desktop is started with a single command *af-sb-init* start inside Scratchbox. However, the Hildon Desktop needs a secondary X server of proper size and bitdepth to be displayed on.

As an exception to the rule, the X server such as *Xephyr*[105] must be started on the host Linux environment, instead of being started inside Scratchbox. The use of Xephyr is described in section 3.4.

Development Tools on Scratchbox

As the Scratchbox environment is practically a full GNU/Linux system, it includes the standard GNU/Linux development tools. Debugging is performed with tools like *GDB*[24], *valgrind*[99], *ltrace* and *strace*[93]. Performance profiling can be performed with tools like *htop*[50], *oprofile*[82] and *time*, and compiling with the GCC toolchain. Some of these tools offer graphical user interfaces, which can also be used. Naturally, this is not a comprehensive list of the tools, and if the tools shipped with the SDK do not suit to needs or personal preferences, most utilities can be easily run practically unchanged in Scratchbox.

Chapter *Application Development* of Maemo Diablo Reference Manual gives good introductions to development using Scratchbox, and the debugging tools are described in depth in chapter *Debugging*14.

Testing and Debugging on Device

Even though Scratchbox is quite accurate in emulating a full target environment on the device, it isn't 100% identical. Especially applications that make use of the device's special hardware can behave differently on the device than on Scratchbox. They even may not work at all. Fortunately, testing the software on device is quite straightforward using either SSH or a tool called *sbrsh* to run target binaries on the device transparently from Scratchbox.

The CPU Transparency section 3.6 has instructions for getting started with *sbrsh*. SSH server and client for maemo can be downloaded from maemo web site[65].

Other Programming Languages

Currently C is the only official programming language for maemo. But thanks to the community, unofficial support exist for several other languages. To name a few, the SDK itself compiles C++, and by adding hildonmm bindings[45], Hildon applications can be created the C++ way. Python scripting language also has a good support in form of *pymaemo*[86], and Ruby bindings are in the works[71], not to forget Mono[78]. For Java support, JaliMo[52] is an interesting project to track.

3.3 Installing SDK

3.3.1 Prerequisites

Before continuing, the installation instructions of the maemo SDK should be reviewed.

There is a special feature that the kernel needs to support in order for the instruction emulator in sbox to work properly. This is the `binfmt_misc`-feature. It is normally built as a module, so verify that it is loaded in Linux (no root access needed for this):

```
user@system:~$ lsmod | grep binfmt
binfmt_misc 12936 0
```

If you do not see a line of output, attempt to do a `modprobe binfmt_misc` as root (or with `sudo`). If this still does not work, you will have to find the module somewhere, or even recompile the kernel. On most Debian-based systems (Debian, Ubuntu), the module is included, so there should not be any problems, unless you have built your own kernel. It is also possible that the feature has been built inside the kernel directly, instead of a module.

Also a pseudo X server should be installed to act as an X client to the real system. It will be necessary to run the applications that are developed, after installing the SDK.

There are a few options for this purpose, but this material will cover the usage of Xephyr. Xephyr is a Kdrive-based X server/client that can emulate 16-color depth for its clients even if it is acting as a client to an 24-bit depth real X server. It also implements modern X protocol extensions.

The concept of having a program that is both X server and a client may seem weird. However, there is no reason to worry, as it is a tested technology and works quite well. If, on the other hand, it does not make any sense, revisit the X Window System introduction in the previous chapter.

To install Xephyr to Debian based Linux:

- Issue the command `sudo apt-get install xserver-xephyr` on your real Linux system.
- Verify installation status by issuing the command `'dpkg -l | grep xephyr'` (as non-root).

3.3.2 Automatic Install of Scratchbox

Up-to-date installation instructions can be found from [72] with instructions for each maemo SDK release.

The preferred way to install the Scratchbox is to use the automated installation script. Manual installation of the Scratchbox is described here for educational purposes, and for situations where the automatic installation script fails.

Quick installation of Scratchbox on a Debian system with the automated `install-script`:

```
user@system:~$ sudo sh ./maemo-scratchbox-install_X.X.sh -u user
```

The `-u user` option is used, so that Scratchbox will add the user account "user" automatically into the group that is allowed to use Scratchbox.

3.3.3 Manual Install of Scratchbox

Scratchbox can also be installed manually. The Debian packages (for the real Linux system) are located at [91]. Apophis is the release of Scratchbox that is suited to be used with maemo 4.x SDK. Please refer to the Scratchbox documentation for further instructions [92].

3.3.4 Automatic Install of Maemo SDK

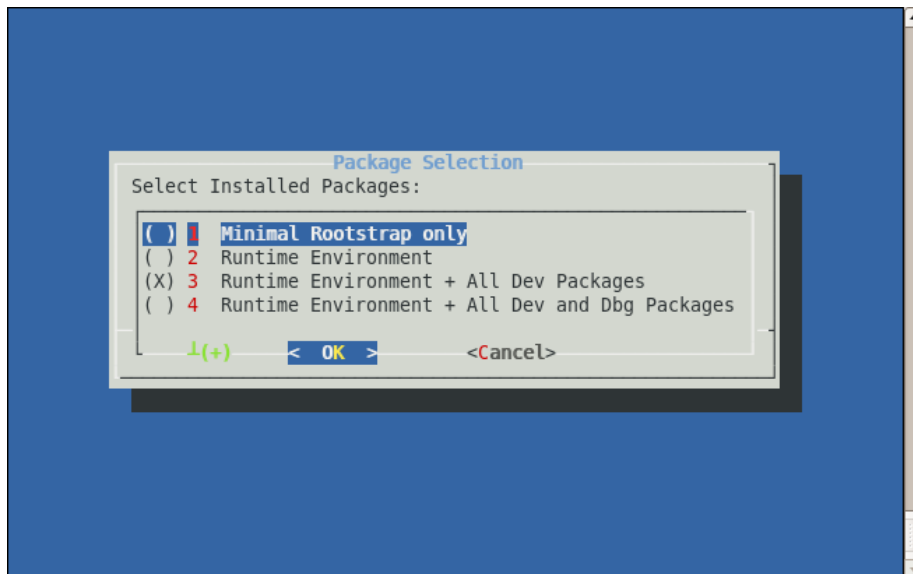
Up-to-date installation instructions can be found from [72], with instructions for each maemo SDK Release.

The preferred way to install the the maemo SDK is to use the automated installation script. In some cases, using a manual process is more suitable; this is covered later. Installing the SDK in an offline environment is officially unsupported, but possible as well.

Quick installation with automated install-script:

```
user@system:~$ sh maemo-sdk-install_X.X.sh
```

Running this script will display the end user license agreement. Pressing Enter key to accept the license presents you with package selection dialog.



You are presented with four options for installing SDK:

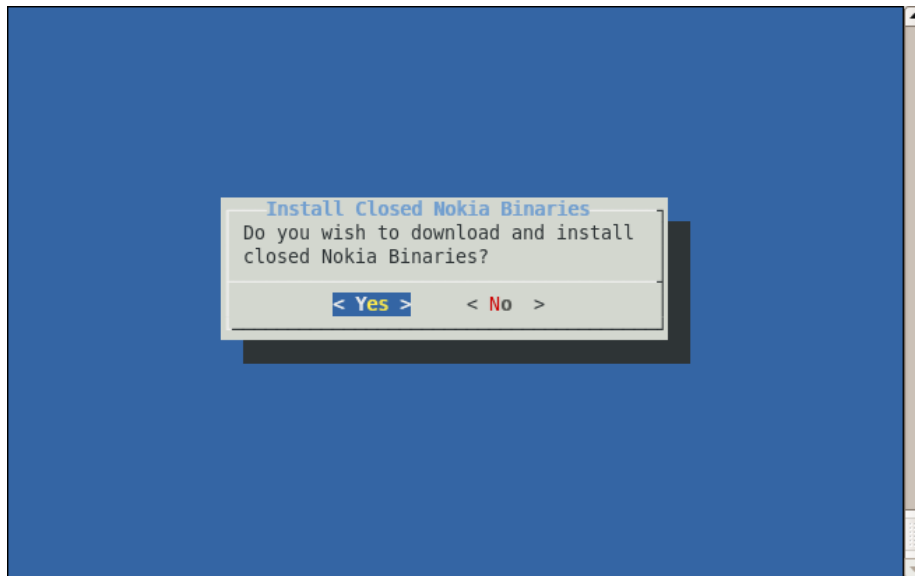
1. Minimal Rootstrap only. Choose this only if you are going to install all packages you need from repository.
2. Runtime Environment. Use this to install and run software inside Scratchbox. Cannot be used for building software.
3. Runtime Environment + All Dev Packages. Choose this to get a full development environment.

4. Runtime Environment + All Dev and Dbg Packages. You will get a full development environment plus debug symbols for many system components.

By default, option 3 is selected.

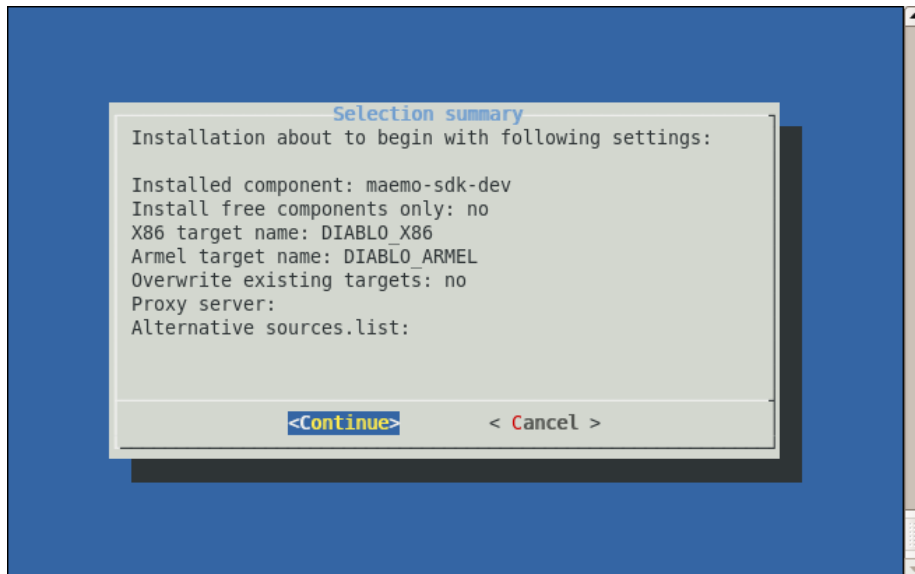
N.B.: The SDK installer will always download and install the minimal rootstrap, but will install additional packages using apt-get based on your choice.

In the next dialog, you can choose to install closed Nokia binaries or not.



Selecting 'yes' will run the Nokia binaries installer script which will display the EUSA(End User Software Agreement). If you accept the agreement, the installer script will extract the Nokia binaries into a folder under the user's home directory inside scratchbox. It will also configure the /etc/apt/sources.list file in the scratchbox targets to make this 'local repository' visible to the Debian apt tools.

In the next dialog, a summary of your selections so far and the default settings are listed.



Selecting 'Continue' will initiate the SDK installation process. If the selection summary is not OK, you can cancel the process and re-start the SDK installation script.

After it's successful execution, you will have 2 scratchbox targets ready for use:

- DIABLO_X86: Suitable for software development and testing.
- DIABLO_ARMEL: Suitable for building software for the ARM architecture.

The Nokia binaries are not installed by default but just made available. If you wish to install all of them, then execute the following command inside the scratchbox targets:

```
[sbox-DIABLO_<target>: ~] > fakeroot apt-get install maemo-explicit
```

N.B. The installer script by default will prompt the user to install the Nokia binaries, which are not open source. To disable this feature, please use `-f` command line parameter for the script. For more options, use the command line help option.

```
user@system:~$ sh maemo-sdk-install_X.X.sh --help
```

3.3.5 Manual Install of Maemo SDK

In order to install the maemo SDK manually, the first step is to download the necessary rootstrap files. There will be two: one for the X86 target, and the other one for the ARM target.

The rootstrap files are available in the same location as the automatic install scripts (for maemo 4.1 SDK they can be found at [\[61\]](#)).

It is necessary to download the minimal rootstraps for i386 and arm, so the filenames will be as follows:

- i386/maemo-sdk-rootstrap_4.1_i386.tgz for the X86 version

- armel/maemo-sdk-rootstrap_4.1_armel.tgz for the ARMEL version

For other versions of the SDK, the exact path names above will need to be adjusted (please consult the SDK installation instructions).

Do not extract the downloaded files. They have to be moved under a location where Scratchbox setup tools can find them (**/scratchbox/packages/**):

```
user@system:~$ sudo mv /tmp/download-location/maemo-sdk-rootstrap* \
/scratchbox/packages/
```

You are now ready to setup your first sbx target. Scratchbox comes with a simple menu-driven tool (sb-menu), which can be used for this. The other option would be using a command line driver tool (sb-conf), but using the menu driver tool is easier.

The first step is to log in on the Scratchbox environment:

```
user@system:~$ /scratchbox/login

You dont have active target in scratchbox chroot.
Please create one by running "sb-menu" before continuing

Welcome to Scratchbox, the cross-compilation toolkit!

Use 'sb-menu' to change your compilation target.
See /scratchbox/doc/ for documentation.

sb-conf: No current target
[sbox-: ~] >
```

By default, Scratchbox will activate the same target that was used previously, but since this is the first time Scratchbox is used, there is no target to activate. One can be built with sb-menu:

1. Type sb-menu inside Scratchbox to launch the tool.

```
[sbox-: ~] > sb-menu
```

2. Select "Setup" in order to create a new target.

Normally the tool would display all configured targets in a list, but since there are none, the dialog is empty. Select "NEW" in order to create a new target.

3. Using the same names as the automatic install script uses allows you to use the Nokia binaries installer later. Type DIABLO_X86 as the target name.
4. Since the first target will be for X86 environment, select the i386 compiler version (cs2005q3.2-glibc2.5-i386).
5. Next, you will need to select all the devkit packages that you want to enable for the new target. You will need debian-etch, maemo3-tools and perl. Do not select cputransp for the X86 target. Select each of them in a row and then press "DONE".
6. Since the cputransp devkit was not selected in the previous step, selecting the CPU transparency becomes "none".

7. This concludes the target-specific tool choices, but there are still things to do. The next step is to select a rootstrap package to extract into the target (select "Yes").
8. And since the rootstraps were already downloaded and copied to the proper location, select "File".
9. Using TAB arrows, navigate to the proper rootstrap file (the one that ends with i386), and select it by pressing space and then press ENTER to go forward.
10. Unpacking the rootstrap will not take long, and soon after that, a dialog will come up with a question about files installation. Select "Yes" (even if it is not entirely obvious what the question means), and then select the C-library, /etc, Devkits and fakeroot. Other tools can be installed later from the maemo SDK repository (or local mirror of the repository).
11. After extracting the selected files from the rootstrap, the target is now ready. You should next opt to select the target (so that it becomes active and will be default target from now on).

Selecting the target will restart the Scratchbox session and if everything went well, you are now left with a very minimal maemo SDK environment:

```
Shell restarting...
[sbox-DIABLO_X86: ~] > arch
i686
[sbox-DIABLO_X86: ~] > dpkg -l | grep maemo-repository
ii maemo-repository      4.1-1      Configuration for maemo repository.
```

12. In order to complete the SDK installation, you will have to fetch the package list and then install the maemo-sdk-dev meta-package. The package depends on a lot of other packages, and all of them will be downloaded into the target. The number of packages is quite significant, so reserve some time for this step. This step will require a working Internet connection (or DNS redirection into a local copy of the repository).

```
[sbox-DIABLO_X86: ~] > apt-get update
```

```
[sbox-DIABLO_X86: ~] > fakeroot apt-get install maemo-sdk-dev
```

Using fakeroot is important in the above command so that the package install scripts think that they are running as the root user. Otherwise the installation phase will fail with errors. Modern Debian-style repositories are signed with GPG keys in order to prevent tampering with the repository contents. The maemo repositories, however, do not use this convention, and this makes apt-get slightly concerned. This can be ignored by accepting installation of unverified packages.

13. The closed Nokia binaries can be obtained by running the script maemo-sdk-nokia-binaries_X.X.sh. If you choose to accept the EUSA, then proceed to the following step.
14. You can install all the nokia binaries in your targets by installing the meta package 'maemo-explicit'.

After `apt-get` finishes installing all the packages, the SDK installation is ready.

When you are finished with `sbox`, you need to logout. This is done by terminating the command shell with the `exit` command, or by using `logout`.

3.3.6 Manual Install of ARMEL Target

If the X86 target was installed manually (above), it is advisable to create the ARMEL target to enable building software for the Internet Tablets. This step can also be postponed until the need to perform cross-building for Internet Tablets arises.

If the automatic install process was used, the ARMEL target is already available (as `DIABLO_ARMEL`), and the following steps are not necessary.

Creating the ARMEL target requires creating a new target in Scratchbox, using the same steps that were taken for the X86 target.

Here is how the ARMEL target install process differs from the X86 (described above):

- Stop any processes you may have running on the X86 `sbox` target (`sb-conf killall`)
- Then start `sb-menu` as you did with the X86 target:
 - Name your target `DIABLO_ARMEL` (for compatibility)
 - You will need to select the arm version of the compiler.
 - You will need to select the `cputransp` devkit and then select `qemu-arm-0.8.2-sb2` as the CPU transparency method (instead of none, as used for X86).
 - You will need to select the arm version of the maemo SDK base rootstrap.

The `apt-get` command remains exactly the same, as do all of the other steps.

You may wish to verify the target by using the steps below (see section 3.4.1), at least build the hello world program and verify the architecture of the resulting executable with `file` command.

3.4 Testing SDK Installation

3.4.1 Testing Scratchbox

The following shows how to create a small non-graphical Hello World program, to verify that the Scratchbox environment works:

```
/**
 * helloworld.c
 *
 * 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.
 *
 * Simple standard I/O (printf)-based Hello World that we can use to
```



```

/* test our toolchains.
*/

#include <stdio.h> /* printf */

/* main implementation */
int main(int argc, char** argv) {

    printf("Hello world\n");

    /* In Linux, each process upon termination must set its exit code.
Exit code 0 means success to whoever executed this program. It
is routinely used inside scripts to test whether running some
program succeeded or not. Other exit codes mean failure. Each
program is free to use different non-zero codes to signify
different kinds of failures. These are normally listed in the
manual page for the program (since there is no standard). If you
forget to set your exit code, it will be undefined. */

    return 0;
}

```

Listing 3.1: helloworld.c

First, it has to be verified that the proper directory is chosen. This can be done by using `pwd` (print working directory). At this point, the work directory should be your home directory:

```

[sbox-DIABLO_X86: ~] > pwd
/home/user

```

Then, start an editor and write the small hello world program (you may use the above listing as a template if you wish):

```

[sbox-DIABLO_X86: ~] > nano helloworld.c

```

`nano` is a GNU version of "pico" editor, which is a simple text file editor. Use `Control+character` to execute the commands listed on the bottom of the screen. `WriteOut` means "save". You may also use `vi` or an external editor to the SDK environment (see below for hints on using `vi` and `emacs`).

```

[sbox-DIABLO_X86: ~] > gcc -Wall -g helloworld.c -o helloworld
[sbox-DIABLO_X86: ~] > ls -F hello*
helloworld* helloworld.c

```

The `-g` option to `gcc` tells the compiler to add debugging symbols to the generated output file. `-Wall` will tell the compiler to enable most of the syntax and other warnings that the source code could trigger. `-o helloworld` then tells the output filename to which `gcc` will write the result binary.

The `-F` option to `ls` is mainly useful when working with a non-color terminal (e.g. paper) to indicate the type of different files. The asterisk after `helloworld` signifies that the file is an executable.

```

[sbox-DIABLO_X86: ~] > ./helloworld
Hello world

```

Running the binary should not produce any surprises.

```

[sbox-DIABLO_X86: ~] > file helloworld
helloworld: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.0, dynamically linked (uses shared libs), not stripped

```

The file tool is a generic utility that will load some bytes from the start of the given file and then use its internal database to decode what the file might "mean". In this case, it will correctly decode the file as a X86 format binary file.

```
[sbox-DIABLO_X86: ~] > ldd helloworld
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/libc.so.6 (0xb7e9f000)
/lib/ld-linux.so.2 (0xb7fd2000)
[sbox-DIABLO_X86: ~] > ls -l /lib/libc.so.6
lrwxrwxrwx 1 user user 11 Nov 12 15:52 /lib/libc.so.6 -> libc-2.5.so
[sbox-DIABLO_X86: ~] > ls -l /lib/libc-2.5.so
-rwxr-xr-x 1 user user 1213256 Sep 7 13:28 /lib/libc-2.5.so
```

The names of dynamic libraries that the executable uses will be shown on the left-hand column, and the files where the libraries live on the system if executing the program will be shown on the right-hand column. After that, use `ls` to check out the exact version of the C library that is used in the SDK by using the "long listing format" `-l` option (running these commands using the ARMEL target would yield more or less the same results). N.B. The `linux-gate.so.1` is a so-called hack to support a certain way of doing system calls on the X86 architecture, and is not always present on newer systems.

When comparing the version of `libc` used on the real system with `ls -l`, it will probably show a difference (in version numbers). This means that the executables that were built inside `sbox` use libraries that are also inside `sbox`. This also means a stable development platform, especially when working in team where each member has their own Linux, which they have customized. This might not seem very important at this stage, but when encountering all the different tools that are used in free software development, this feature of `sbox` will come in handy.

Scratchbox does not contain any logic to emulate the kernel (or to use a different kernel for running programs inside `sbox`). The only easy possibility for this is using the `sbrsh` CPU-transparency option.

vi

It is also possible to use `vi` (Visual Interactive) editor inside `sbox`. It is possible to install own favorite editors inside `sbox` (with the `debian-devkit`), but the following examples will use `nano`, since it is the easiest to start with. To learn `vi`, it is best to ask an Internet search engine for a "vi tutorial". There are lots of them to be found. To understand why `vi` can be considered to be "strange", it is useful to know its history first. Using `vi` is optional of course.

The version of `vi` that is commonly installed on Linux systems is really `vim` (VI iMproved), which is a more user friendly `vi`, including syntax high-lighting and all kinds of improvements. `sbox` has a program called `vimtutor` installed to help in learning the use of `vi` interactively.

It is also fairly simple to use existing editors. `/scratchbox/users/x/home/x/` is the home directory of user `x` when accessing it from the real Linux desktop. Ubuntu comes with `gedit`, which is a fairly good graphical editor that also supports syntax high-lighting and multiple tabs for editing multiple files at the same time.

And as a final note, also `emacs` can be used.

Here is how to do it:

- Start `emacs` outside of `sbox`

- In emacs, use M-x server-start
- Inside sbx use emacsclient filename to open the file for editing in your emacs

3.4.2 Writing GUI Hello World

The following example shows how to write the first GUI program. N.B. Only GTK+ library is used here, meaning that the platform provided widgets or coding style are not utilized. That will be discussed soon.

```
/**
 * gtk_helloworld-1.c
 *
 * 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.
 *
 * A simple GTK+ Hello World. You need to use Ctrl+C to terminate
 * this program since it doesn't implement GTK+ signals (yet).
 */

#include <stdlib.h> /* EXIT_* */
/* Introduce types and prototypes of GTK+ for the compiler. */
#include <gtk/gtk.h>

int main(int argc, char** argv) {

    /* We'll have two references to two GTK+ widgets. */
    GtkWindow* window;
    GtkLabel* label;

    /* Initialize the GTK+ library. */
    gtk_init(&argc, &argv);

    /* Create a window with window border width of 12 pixels and a
       title text. */
    window = g_object_new(GTK_TYPE_WINDOW,
        "border-width", 12,
        "title", "Hello GTK+",
        NULL);

    /* Create the label widget. */
    label = g_object_new(GTK_TYPE_LABEL,
        "label", "Hello World!",
        NULL);

    /* Pack the label into the window layout. */
    gtk_container_add(GTK_CONTAINER(window), GTK_WIDGET(label));

    /* Show all widgets that are contained by the window. */
    gtk_widget_show_all(GTK_WIDGET(window));

    /* Start the main event loop. */
    g_print("main: calling gtk_main\n");
    gtk_main();

    /* Display a message to the standard output and exit. */
    g_print("main: returned from gtk_main and exiting with success\n");
}
```

```

/* The C standard defines this condition as EXIT_SUCCESS, and this
   symbolic macro is defined in stdlib.h (which GTK+ will pull in
   in-directly). There is also a counter-part for failures:
   EXIT_FAILURE. */
return EXIT_SUCCESS;
}

```

Listing 3.2: gtk_helloworld-1.c

Build your program:

```

[sbox-DIABLO_X86: ~] > gcc -Wall -g gtk_helloworld-1.c -o gtk_helloworld-1
gtk_helloworld-1.c:15:21: gtk/gtk.h: No such file or directory
gtk_helloworld-1.c: In function 'main':
gtk_helloworld-1.c:20: error: 'GtkWindow' undeclared (first use in this function)
gtk_helloworld-1.c:20: error: (Each undeclared identifier is reported only once
gtk_helloworld-1.c:20: error: for each function it appears in.)
gtk_helloworld-1.c:20: error: 'window' undeclared (first use in this function)
gtk_helloworld-1.c:21: error: 'GtkLabel' undeclared (first use in this function)
gtk_helloworld-1.c:21: error: 'label' undeclared (first use in this function)
gtk_helloworld-1.c:24: warning: implicit declaration of function 'gtk_init'
gtk_helloworld-1.c:28: warning: implicit declaration of function 'g_object_new'
gtk_helloworld-1.c:28: error: 'GTK_TYPE_WINDOW' undeclared (first use in this function)
gtk_helloworld-1.c:34: error: 'GTK_TYPE_LABEL' undeclared (first use in this function)
gtk_helloworld-1.c:39: warning: implicit declaration of function 'gtk_container_add'
gtk_helloworld-1.c:39: warning: implicit declaration of function 'GTK_CONTAINER'
gtk_helloworld-1.c:39: warning: implicit declaration of function 'GTK_WIDGET'
gtk_helloworld-1.c:42: warning: implicit declaration of function 'gtk_widget_show_all'
gtk_helloworld-1.c:45: warning: implicit declaration of function 'g_print'
gtk_helloworld-1.c:46: warning: implicit declaration of function 'gtk_main'

```

As can be seen, this does not look at all promising. At the start of the source code, there was `#include`. The compiler needs to be told where it should look for that critical GTK+ header file. It is also quite likely that some special flags need to be passed to the compiler in order for it to use the proper compilation settings when building GTK+ software. How to decide which flags to use?

This is where a tool called `pkg-config` comes to the rescue. It is a simple program that provides a unified interface to output compiler, linker flags and library version numbers. Its utility will be discussed later, when starting the automating of the building process. For now, the `pkg-config` will be used manually.

```

[sbox-DIABLO_X86: ~] > pkg-config --list-all | sort
.. listing cut to include only relevant libraries ..
dbus-glib-1    dbus-glib - Glib integration for the free desktop message bus
gconf-2.0      gconf - GNOME Config System.
gdk-2.0        GDK - GIMP Drawing Kit (x11 target)
gdk-pixbuf-2.0 GdkPixbuf - Image loading and scaling
glib-2.0       Glib - C Utility Library
gnome-vfs-2.0  gnome-vfs - The GNOME virtual file-system libraries
gtk+-2.0       GTK+ - GIMP Tool Kit (x11 target)
hildon-1       hildon - Hildon widgets library
hildon-fm-2    hildon-fm - Hildon file management widgets
pango         Pango - Internationalized text handling
x11           X11 - X Library

```

`pkg-config` also has some other commands that will be useful:

```

[sbox-DIABLO_X86: ~] > pkg-config --modversion gtk+-2.0
2.10.12

```

```

[sbox-DIABLO_X86: ~] > pkg-config --cflags gtk+-2.0
-I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include -I/usr/include/atk-1.0
-I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/glib-2.0
-I/usr/lib/glib-2.0/include -I/usr/include/freetype2
-I/usr/include/libpng12

```

As can be seen, there are many. With this version of GTK+, all of them are -I options. They are used to tell the compiler which additional directories to check for system header files in addition to the default ones.

```
[sbox-DIABLO_X86: ~] > pkg-config --libs gtk+-2.0
-lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm
-lpangocairo-1.0 -lpango-1.0 -lcairo -lgobject-2.0 -lgmodule-2
-ldl -lglib-2.0
```

When linking the application, the linker has to be told which libraries to link against. In fact, the whole program linking phase will fail (as shown shortly) without this information.

Now it is time to try and compile the software again, this time using the compilation flags that pkg-config provides:

```
[sbox-DIABLO_X86: ~] > gcc -Wall -g gtk_helloworld-1.c \
'pkg-config --cflags gtk+-2.0' -o gtk_helloworld-1
/var/tmp/ccQ14x4c.o: In function 'main':/home/user/gtk_helloworld-1.c:24:
undefined reference to 'gtk_init'
:/home/user/gtk_helloworld-1.c:28: undefined reference to 'gtk_window_get_type'
:/home/user/gtk_helloworld-1.c:28: undefined reference to 'g_object_new'
:/home/user/gtk_helloworld-1.c:34: undefined reference to 'gtk_label_get_type'
:/home/user/gtk_helloworld-1.c:34: undefined reference to 'g_object_new'
:/home/user/gtk_helloworld-1.c:39: undefined reference to 'gtk_widget_get_type'
:/home/user/gtk_helloworld-1.c:39: undefined reference to 'g_type_check_instance_cast'
:/home/user/gtk_helloworld-1.c:39: undefined reference to 'gtk_container_get_type'
:/home/user/gtk_helloworld-1.c:39: undefined reference to 'g_type_check_instance_cast'
:/home/user/gtk_helloworld-1.c:39: undefined reference to 'gtk_container_add'
:/home/user/gtk_helloworld-1.c:42: undefined reference to 'gtk_widget_get_type'
:/home/user/gtk_helloworld-1.c:42: undefined reference to 'g_type_check_instance_cast'
:/home/user/gtk_helloworld-1.c:42: undefined reference to 'gtk_widget_show_all'
:/home/user/gtk_helloworld-1.c:45: undefined reference to 'g_print'
:/home/user/gtk_helloworld-1.c:46: undefined reference to 'gtk_main'
:/home/user/gtk_helloworld-1.c:49: undefined reference to 'g_print'
collect2: ld returned 1 exit status
```

The command above might seem somewhat strange to someone not having used UNIX command shells. What is happening here is the backtick expansion. It is an operation where the shell will start another shell to execute just the text inside the backticks. In this case, another shell is started to run "pkg-config cflags gtk+-2.0". Normal output from the commands is then read into the main shell, and this output is replaced into the location where the backticks were. N.B. It is very important to use the ' character. Not ', nor the other quote character that might be used in a Swedish keyboard layout (also used in Finland). In some keyboard layouts, it will be necessary to press space after the backtick since it is also used for character composition (try backtick and the letter 'a').

Something like \$(pkg-config ..) might also be encountered. This is the same operation as backtick. However, backtick is more portable across antique UNIX shells. Nowadays, it is a matter of taste which way to use it.

The errors printed by gcc are quite different this time. These errors come from ld, which is the binary code linker in Linux systems and it is complaining about missing symbols (the undefined references). Obviously something is still missing.

The linker needs to be told where to find the missing symbols. Since it is the linker this is all about, and not the compiler, the missing symbols are found in the library files. To fix the problem (again with the backticks), pkg-config libs can be used:

```
[sbox-DIABLO_X86: ~] > gcc -Wall -g gtk_helloworld-1.c \
'pkg-config --cflags gtk+-2.0' -o gtk_helloworld-1 \
'pkg-config --libs gtk+-2.0'
[sbox-DIABLO_X86: ~] >
```

The order and placement of the pkg-configs above is important: the cflags need to be placed as early as feasible, but the libs must come last (this does matter in some problematic linking scenarios).

The next step is to repeat the basic commands that were used before with the non-GUI hello world:

```
[sbox-DIABLO_X86: ~] > ls -l gtk_helloworld-1
-rwxrwxr-x 1 user user 16278 Nov 20 00:22 gtk_helloworld-1
[sbox-DIABLO_X86: ~] > file gtk_helloworld-1
gtk_helloworld-1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.0, dynamically linked (uses shared libs), not stripped
[sbox-DIABLO_X86: ~] > ldd gtk_helloworld-1
linux-gate.so.1 => (0xffffe000)
libgtk-x11-2.0.so.0 => /usr/lib/libgtk-x11-2.0.so.0 (0xb7c4c000)
libgdk-x11-2.0.so.0 => /usr/lib/libgdk-x11-2.0.so.0 (0xb7bc8000)
libatk-1.0.so.0 => /usr/lib/libatk-1.0.so.0 (0xb7bad000)
libgdk_pixbuf-2.0.so.0 => /usr/lib/libgdk_pixbuf-2.0.so.0 (0xb7b97000)
libm.so.6 => /lib/libm.so.6 (0xb7b71000)
libpangocairo-1.0.so.0 => /usr/lib/libpangocairo-1.0.so.0 (0xb7b68000)
libpango-1.0.so.0 => /usr/lib/libpango-1.0.so.0 (0xb7b2b000)
libcairo.so.2 => /usr/lib/libcairo.so.2 (0xb7ab5000)
libgobject-2.0.so.0 => /usr/lib/libgobject-2.0.so.0 (0xb7a7a000)
libgmodule-2.0.so.0 => /usr/lib/libgmodule-2.0.so.0 (0xb7a76000)
libdl.so.2 => /lib/libdl.so.2 (0xb7a71000)
libglib-2.0.so.0 => /usr/lib/libglib-2.0.so.0 (0xb79dd000)
libc.so.6 => /lib/libc.so.6 (0xb78b2000)
libX11.so.6 => /usr/lib/libX11.so.6 (0xb77bd000)
libXfixes.so.3 => /usr/lib/libXfixes.so.3 (0xb77b8000)
libXtst.so.6 => /usr/lib/libXtst.so.6 (0xb77b3000)
libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0xb7788000)
libXext.so.6 => /usr/lib/libXext.so.6 (0xb777a000)
libXrender.so.1 => /usr/lib/libXrender.so.1 (0xb7771000)
libXi.so.6 => /usr/lib/libXi.so.6 (0xb7769000)
libXrandr.so.2 => /usr/lib/libXrandr.so.2 (0xb7762000)
libXcursor.so.1 => /usr/lib/libXcursor.so.1 (0xb7759000)
/lib/ld-linux.so.2 (0xb7fc3000)
libpangoft2-1.0.so.0 => /usr/lib/libpangoft2-1.0.so.0 (0xb772b000)
libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0xb76c6000)
libz.so.1 => /usr/lib/libz.so.1 (0xb76b7000)
libpng12.so.0 => /usr/lib/libpng12.so.0 (0xb7692000)
libXau.so.6 => /usr/lib/libXau.so.6 (0xb768f000)
libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0xb7689000)
libexpat.so.1 => /usr/lib/libexpat.so.1 (0xb7669000)
```

As can be seen from the last ldd listing, this simple Hello World manages to require quite a number of other libraries to run. The program directly only requires GTK+, but GTK+ needs GDK (and all the other libraries that were covered in the introduction). Those libraries in turn need other libraries and so on.

So, what is seen here is almost the full list of all required libraries to run. Almost, because modern UNIX systems (and Linux) can also load libraries on demand (called runtime dynamic module loading).

This might make one wonder, why writing simple Hello World is this painful. It is actually much simpler in real life. The reason why this chapter introduces the various errors is that they will be encountered in actual situations quite early on. This chapter serves as a reference to some possible errors, and (hopefully) also show a solution.

All of these tools will be needed later on, when starting the packaging of the software, and they will not be covered at this level of detail there.

3.4.3 Running GUI Hello World

Let's try to execute our nice Hello World (inside sbx):

```
[sbx-DIABLO_X86: ~] > ./gtk_helloworld-1
gtk_helloworld-1[4759]: GLIB WARNING **: Gtk - cannot open display:
[sbx-DIABLO_X86: ~] > echo $DISPLAY
```

Seems that GTK+ is having problems opening the connection to the X server. This can be verified by displaying the contents of the DISPLAY environmental variable, and indeed, it comes out empty. If the DISPLAY variable contains :0.0, it means that the value has been copied from the real graphical session into sbx, and clients will try to connect to the real X server (and probably fail during authentication).

Xephyr was installed in the previous chapter, so now it has to be started so that it can be used as the server for all clients running inside the Scratchbox session.

3.4.4 Starting Virtual X Server (Xephyr)

Open another terminal emulator (do not close your sbx session).

Start up the server with:

```
user@system:~$ Xephyr :2 -host-cursor -screen 800x480x16 -dpi 96 -ac \
-extension Composite
```

The first parameter is the Display number (:2) that X server should start on (and provide to clients). :2 is used here, since it is normally unused in regular Linux desktop environments.

The screen parameters tells Xephyr how large the screen should be (in pixels), and how many bits to use for color-information (16). This is the resolution in pixels of Internet Tablets. -dpi 96 tells the server to tell its clients that the logical to physical size mapping should be done with 96 dots-per-inch setting (should the client request that information). The DPI setting is mainly important when dealing with fonts and text.

-ac tells Xephyr that any client may connect to it. This means that the networking environment should be treated with extreme caution, so that rogue users will not target your Xephyr with their own clients.

The last parameter (-extension Composite) disables the Composite extension.

When Xephyr starts, it will connect to the X server given in the DISPLAY environmental variable that it sees. Do not modify or touch your real DISPLAY variable that Xephyr sees.

By default, the server will have one screen, and it will be filled by the default X server background pattern (a tight braid made out of white pixels on black).

N.B. The terminal emulator (more specifically, the shell that the emulator started) is waiting for Xephyr to end. If all the graphical applications running in the SDK ever need to be killed, it can be done easily by just closing Xephyr. This will leave the daemons running inside sbx (D-Bus and friends). Normally this is not a good idea. To ask the foreground process to terminate itself, use Ctrl+c. Inside sbx this same technique can be used to terminate a graphical client that was started from the command line (as will be done shortly).

3.4.5 Directing Client to Virtual Server

Now that there is an X server running, it is time to switch back to sbbox.

The first step is to set the environmental variable so that X server knows to use a local domain socket, and tell all X clients to connect to Display number 2, since that is where the Xephyr was just started:

```
[sbox-DIABLO_X86: ~] > export DISPLAY=:2
[sbox-DIABLO_X86: ~] > ./gtk_helloworld-1
main: calling gtk_main
[[Ctrl+c]]
```

The program has to be terminated with Ctrl+c, since it does not implement any graphical methods for closing. N.B. The DISPLAY needs to be set correctly on each sbbox login (or target switch).

Since window manager is missing (there were none started for the X server), the Hello World cannot be controlled with the mouse. Kill it with Ctrl+c for now (It will be started again in a moment).

To quickly detach the foreground process from the shell and continue running it in the background, use the following key-combination Ctrl+z and then use the shell command bg. Between those two steps the process will be "paused".

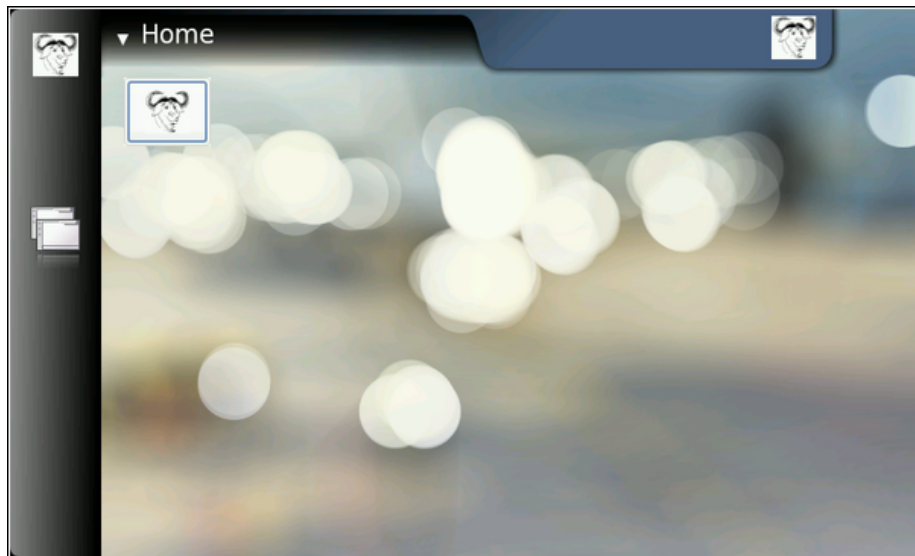
3.4.6 Starting Application Framework

The next step is to have a nice graphical environment that will implement a nicer graphical screen. For this, a series of clients will be started, each of which have a specific role. These were introduced before.

To start the Application Framework (referred to as AF from now on), it is possible to use a handy script that comes with the SDK:

```
[sbox-DIABLO_X86: ~] > af-sb-init.sh start
Sample files present.
Starting DBUS system bus
Starting D-BUS session bus daemon
Starting Maemo Launcher: maemo-launcher
maemo-launcher: error rising the oom shield for pid=4847 status=5632.
Starting Sapwood image server
Starting Matchbox window manager
sapwood-server[4858]: GLIB INFO default - server started
Starting clipboard-manager
Starting Keyboard
maemo-launcher: invoking '/usr/bin/hildon-input-method.launch'
Starting Hildon Desktop
maemo-launcher: invoking '/usr/bin/hildon-desktop.launch'
.. listing cut for brevity ..
[sbox-DIABLO_X86: ~] >
```

The start and stop parameters are used to start and stop the graphical environment. If everything works, it should show a screen resembling this one:



3.4.7 Running Hello World in Application Framework

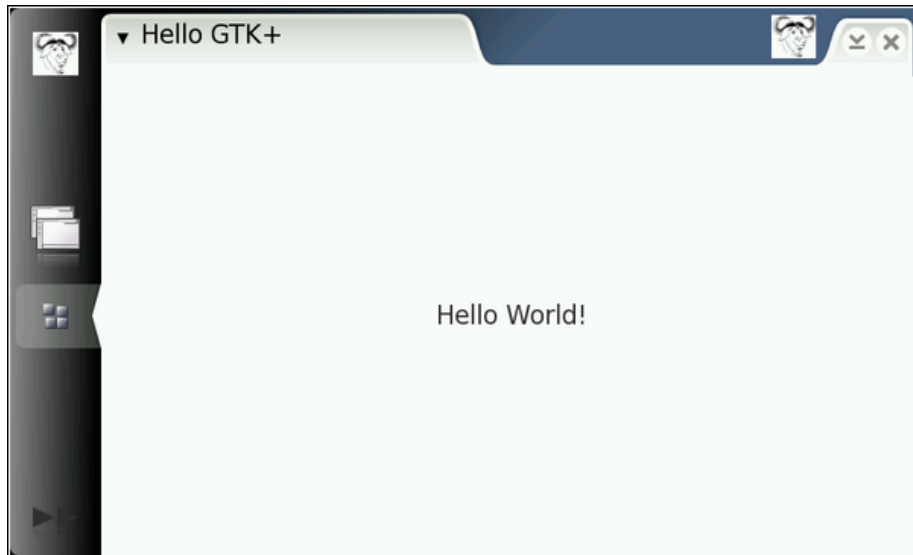
While AF is running, the Hello World can be started again (`./gtk_helloworld-1` in `sbox`):



Since there now is a window manager running, the client will get much larger window to draw in. GTK+ will scale the widget accordingly (there is only one widget in the program). N.B. The screen still looks a bit off. If the application is closed by pressing the X in the top-right corner, it can be seen that the Hello World will disappear from the screen. Since no signals have been implemented yet and thus there is nothing to handle window destruction, the Hello World application will only be hidden. It is still running (as can be seen in the `sbox` terminal emulator, since the shell does not display its prompt). Stop the Hello World with `Ctrl+c`.

Next step shows how to use an SDK utility script called `run-standalone.sh`. Its job is to set up correct environmental variables for themes and communication for the command that is given to it as its command line parameter.

Use `run-standalone.sh ./gtk_helloworld-1` to start your X client again:



The screen is still a bit off (there are no borders around the main `GtkLabel` widget), but looks already better. The text is scaled to be more in sync with the other text sizes and also the color is in sync with the platform color (it is not gray anymore).

3.5 Setting up USB Networking

3.5.1 Introduction

This section describes how USB networking can be set up between the host Linux machine and Internet Tablet.

USB networking connects the device and Linux laptop with IP protocol over the USB cable. This provides fast and effortless access to the device and enables e.g. file copying, remote login over SSH, using of NFS and remote debugging.

3.5.2 Setting up Environment and Device

In order to use USB networking on the device, some extra packages must be installed, such as `openssh`.

Linux host side instructions are for Debian and its derivatives, and might not work on other distributions.

A root shell access on the device is needed for setting up the USB networking on the device side. For this, `xterm` has to be started. There is a script for root access that should be run on terminal.

```
$ sudo gainroot
```

When using USB networking, device must be booted from flash, meaning that MMC dualboot is an unavailable option. The reason for this is that the device can only act as either a mass storage or network device at any given time.

Check whether the device has been preconfigured for USB networking with IP **192.168.2.15**. If not, please add the following lines to your `/etc/network/interfaces`:

```
auto usb0
iface usb0 inet static
    address 192.168.2.15
    netmask 255.255.255.0
    gateway 192.168.2.14
```

The IP address of the host machine in the USB network has to be **192.168.2.14** for the `usb0` interface.

The following steps should prepare the device and host machine for USB networking:

1. Make sure the USB cable is disconnected.
2. Gain root shell access on the device, and perform the following:

```
# insmod /mnt/initfs/lib/modules/2.6.21-omap1/g_ether.ko
# ifup usb0
```

N.B. You may need to run `insmod` twice.

3. Make sure the kernel of the host machine has `usbnet` enabled.
4. Add the following entry to the `/etc/network/interfaces` file of the host machine:

```
mapping hotplug
    script grep
    map usb0

auto usb0
iface usb0 inet static
    address 192.168.2.14
    netmask 255.255.255.0
    network 192.168.2.0
    broadcast 192.168.2.255
    up iptables -t nat -A POSTROUTING -o eth0 -s 192.168.2.15 -j MASQUERADE
    up echo 1 > /proc/sys/net/ipv4/ip_forward
    down iptables -t nat -D POSTROUTING -o eth0 -s 192.168.2.15 -j MASQUERADE
    down echo 0 > /proc/sys/net/ipv4/ip_forward
```

5. Now both the device and the host machine should be ready for initiating a USB connection

3.5.3 Testing Connection

After making the preparations, the USB network should automatically be set up when connecting the device to the host machine with a USB cable.

For basic testing, ping can be performed from the host machine.

```
$ ping 192.168.2.15
PING 192.168.2.15 (192.168.2.15) 56(84) bytes of data.
64 bytes from 192.168.2.15: icmp_seq=1 ttl=245 time=79.8 ms
...
```

Use Application Manager to install openssh from [Diablo extras](#). Now, an ssh root shell over the USB network from host to tablet can be initiated.

```
$ ssh root@192.168.2.15
```

The password of the root is rootme.

3.6 CPU Transparency

This document provides instructions for setting up sbrsh CPU transparency for the Scratchbox ARMEL target. N.B. The use of sbrsh is insecure. It should only be run on trusted networks.

Prerequisites for setting up CPU transparency are as follows:

1. Scratchbox installed by maemo Scratchbox installer
2. Scratchbox ARMEL target installed by maemo SDK installer with sbrsh set as the CPU transparency method.
3. NFS support compiled and transferred onto the device. For instructions on how to do this, see [11](#).
4. Networking set-up using static IP addresses 192.168.2.15 on the device, and 192.168.2.14 on the host Linux PC computer, as described in section [3.5](#).

3.6.1 Setting up Host Linux PC Environment

To set up the host environment, the following commands have to be run with root access.

1. Install NFS kernel server in the Linux PC. In Debian, this is provided by the package nfs-kernel-server. Fetch the required software by running the command:

```
$ apt-get install nfs-kernel-server
```

2. Edit the /etc/exports file in the Linux PC. This file lists the NFS file systems that will be exported to the device. Add the following line (replace sbbox_username, sbbox_user_uid and sbbox_user_gid with correct values, i.e. use your own user names and IDs):

```
/scratchbox/users/<sbbox_username>/targets/ 192.168.2.15(rw,all_squash,anonuid=
<sbbox_user_uid>,anongid=<sbbox_user_gid>,sync)
/scratchbox/users/<sbbox_username>/home 192.168.2.15(rw,all_squash,anonuid=
<sbbox_user_uid>,anongid=<sbbox_user_gid>,sync)
```

Make sure that the user IDs used in the /etc/exports file in Linux PC match the user IDs that are effective inside the Scratchbox.

3. Restart NFS kernel server to apply new exports:

```
root@host:~ # /etc/init.d/nfs-kernel-server restart
* Stopping NFS kernel daemon... [ ok ]
* Unexporting directories for NFS kernel daemon... [ ok ]
* Exporting directories for NFS kernel daemon... [ ok ]
* Starting NFS kernel daemon: [ ok ]
root@host:~ #
```

Now the Linux PC host system should be set up.

3.6.2 Setting up Scratchbox ARMEL Target

Now the ARMEL target needs to be set up inside Scratchbox. To do this, first start Scratchbox and change to the ARMEL target, if necessary.

Edit the `~/sbrsh` in the home directory of the Scratchbox user, adding the following lines, replacing `<username>` and `<target_name>` with real names using the Scratchbox:

```
ARMEL 192.168.2.15
nfs    192.168.2.14:/scratchbox/users/<username>/targets/<target_name> / rw,nolock,noac
nfs    192.168.2.14:/scratchbox/users/<username>/home /home rw,nolock,noac
bind   /dev          /dev
bind   /dev/pts      /dev/pts
bind   /proc         /proc
bind   /tmp          /tmp
```

If you are using the `sb-menu` command in Scratchbox, configuring the `sbrshd` settings with the tool, make sure that you do not use the term `localhost`, but instead the full IP address of your Linux PC.

N.B. If the network connection between the Tablet and Linux PC breaks, and the Scratchbox is trying to push the required configurations to the device, it will not give any warning of failure. Check always from the device that the configurations are in order. In a later section, there are instructions to set up device configuration manually. It can also be used for checking purposes.

With these changes, the `_ARMEL` target should be set up accordingly.

3.6.3 Setting up Device for CPU Transparency

1. Copy `sbrshd` and its init-script onto the device using a memory card or `scp`. These can be found under `/scratchbox/device_tools/sbrsh-7.4/cs2005q3.2-glibc2.5-arm`, in the current stable release of Scratchbox. N.B. The `initscript` searches binary from `/usr/sbin` by default, and quietly exits if not found there.
2. Edit `/etc/sbrshd.conf` and add entry for the desktop PC:

```
192.168.2.14 *
```

N.B. There are additional configurable options for `sbrshd`, and these are covered in the Scratchbox documentation.

3. Make sure NFS modules are loaded into the device's kernel.
4. Start `sbrshd`

```
# /etc/init.d/sbrshd start
```

The `sbrshd` daemon should now be up and running, ready to accept host PC connections for CPU transparency.

3.6.4 Testing CPU Transparency

Now the CPU transparency should be set up. To test it, log into Scratchbox and change to the ARMEL target.

After this, extract the `hello-world.tar.gz` package and go the `hello-world` directory. Run `./autogen.sh` and `make`, followed by `./hello`. This should print

the text hello world on the console.

```
[sbox-ARMEL: ~] > tar xzf /scratchbox/packages/hello-world.tar.gz
[sbox-ARMEL: ~] > cd hello-world
[sbox-ARMEL: ~/hello-world] > ./autogen.sh
+ aclocal
[... output from aclocal ...]
+ autoconf
+ autoheader
+ automake --add-missing --foreign
[... output from automake ...]
+ ./configure
[... output from ./configure ...]
[sbox-ARMEL: ~/hello-world] > make
make all-am
make[1]: Entering directory '/home/username/hello-world'
if gcc -DHAVE_CONFIG_H -I. -I/home/username/hello-world -I.      -g -O2 -MT main.o -MD -MP
-MF ".deps/main.Tpo" -c -o main.o main.c; \
then mv -f ".deps/main.Tpo" ".deps/main.Po"; else rm -f ".deps/main.Tpo"; exit 1; fi
gcc -g -O2      -o hello main.o
make[1]: Leaving directory '/home/username/hello-world'
[sbox-ARMEL: ~/hello-world] > ./hello
Hello World!
[sbox-ARMEL: ~/hello-world] >
```

The program is run transparently on the device. It can be verified by the log written to /tmp/cputransp_username.log

```
[sbox-ARMEL: ~/hello-world] > cat /tmp/cputransp_username.log
[2005-10-13 17:37:26 26032] method: "sbrsh"  pwd: "/home/username/hello-world"  cmd: "/home/username/hello-world/a.out"

[2005-10-13 17:37:30 26032] rc: 0  time: 3.944084
[2005-10-13 17:37:30 26272] method: "sbrsh"  pwd: "/home/username/hello-world"  cmd: "/home/username/hello-world/conftest"
[2005-10-13 17:37:31 26272] rc: 0  time: 0.196024
[2005-10-13 17:37:42 26596] method: "sbrsh"  pwd: "/home/username/hello-world"  cmd: "/home/username/hello-world/hello"

[2005-10-13 17:37:42 26596] rc: 0  time: 0.192171
[sbox-ARMEL: ~/hello-world] >
```

As expected, "rc" contains the result of the remote running program, and "time" contains the total running time on the device. Any sbrsh errors are also output to this file.

3.7 Using Linux Flasher Tool

This section describes how to use the Linux flasher tool, and how to flash a rootfs to the Internet Tablet device.

The flasher is a command line tool that can be used to flash items to the Nokia Internet Tablet. The utility can be downloaded from this location. **N.B.** Attention should be paid to selecting the correct flasher for the product in question, as different Internet Tablet products may require a specific flasher tool.

The flasher tool should be saved to the directory that is to be used as the working directory. After saving the tool, the file should be changed with chmod to be executable.

```
$ chmod +x flasher
```

To run the flasher, root-user rights are needed. As a root user, type:

```
$ ./flasher
```

This command will display the following information on how to use the flasher:

```
Usage: flasher [OPTIONS]
Options:
--fiasco, -F          Location of a FIASCO image
--kernel, -k          Location of kernel image
--initfs, -n          Location of initfs image
--rootfs, -r          Location of root JFFS2 image
--xloader, -x         Location of X-Loader image
--secondary, -s       Location of NOL0 secondary bootloader image
--2nd, -2             Location of NOL0 cold flasher ("2nd") image
--unpack, -u [arg]    Unpack a FIASCO image
--flash, -f           Load and flash all supplied images
--load, -l            Only load all supplied images
--boot, -b [arg]      Boot the kernel with optional cmdline
--reboot, -R          Reboot the board (e.g. after flashing NOL0)
--read-board-id, -i   Print out the board type
--serial-port, -S     Serial port used for cold flashing
--initialize-port, -I Only initialize the serial port
--cold-flash, -c       "Cold flash" the device
--enable-rd-mode      Enable R&D mode on the device
--disable-rd-mode     Disable R&D mode on the device
--set-rd-flags [arg]  Set R&D mode flags on the device
--clear-rd-flags [arg] Clear R&D mode flags on the device
--query-rd-mode       Query the device R&D mode status and flags
--set-root-device     Set the default root device
--query-root-device   Query the default root device
--enable-usb-host-mode Set the device into USB host mode
--disable-usb-host-mode Set the device into USB peripheral mode
--flash-only          Flash only certain components
```

This shows all the necessary parameters and information that are enabled for use in the flasher.

To try out the flasher, first retrieve a flash image for your specific product. For example, software for N800 is available in tablets-dev.nokia.com.

Save the image to the same directory where the flasher tool is located, and complete the following steps:

- Make sure you are the root user on your PC
- Turn off the Nokia Internet Tablet
- Make sure your device is connected to your PC via USB

After these steps, the device is ready to receive the root file system. Enter the following command (N.B. You must be the root user or use the sudo command):

```
$ ./flasher -F RX-XX_20XXSE_X.20XX.XX-X_PR_COMBINED_MR0_ARM.bin -f -R
```

After this command, turn on the device, and the flashing will start.

If everything goes well, a progress bar should appear at the bottom of the device screen.

The command above will flash the new image to the device. When the flashing is complete, the device will reboot.

Chapter 4

GNU Build System

4.1 Introduction

The following code examples are used in this chapter:

- [simple-make-files](#)
- [autoconf-automake](#)

4.2 GNU Make and Makefiles

The *make* program from the GNU project is a powerful tool to aid implementing automation in the software building process. Beside this, it can be used to automate any task that uses files and in which these files are transformed into some other form. Make by itself does not know what the files contain or what they represent, but using a simple syntax it can be taught how to handle them.

When developing software with gcc (and other tools), gcc will often be invoked repeatedly with the same parameters and flags. After changing one source file, it will be noticed that other output files need to be rebuilt, and even the whole application if some interface has changed between the functions. This might happen whenever declarations change, new parameters are added to function prototypes etc.

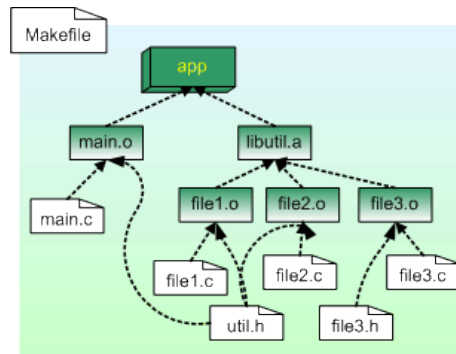
These tasks could, of course, be always performed manually, but after a while a nicer way is going to be more desirable.

GNU make is a software building automation tool that will execute repetitive tasks. It is controlled via a **Makefile** that contains lists of dependencies between different source files and output files. It also contains lists of commands that should be executed to satisfy these dependencies. Make uses the **timestamps** of files and the information of the files' existence to automate the rebuilding of applications (targets in make), as well as the rules that are specified in the Makefile.

Make can be used for other purposes as well. A target can easily be created for installing the built software on a destination computer, a target for generating documentation by using some automatic documentation generation tool, etc. Some people use make to keep multiple Linux systems up to date with the

newest software and various system configuration changes. In short, make is flexible enough to be generally useful.

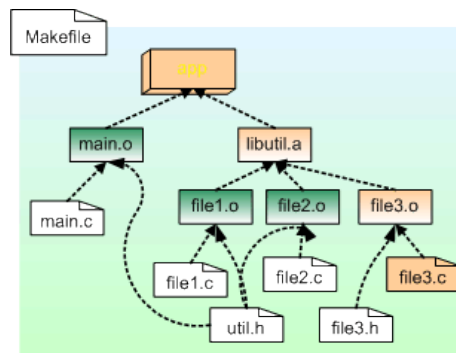
The dependencies between different files making up a software project:



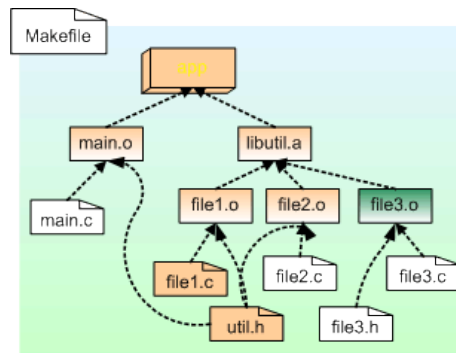
The aim of make is to satisfy the target. Each target has its own dependencies. A user will generally select a target for make to satisfy by supplying the target name on the command line. Make will start by checking, whether all of the dependencies exist and have an older timestamp than the target. If so, make will do nothing, as nothing has changed. However, since a header file (that an application is not dependent on directly) might change, make will propagate the changes to the 'root' of the target as shown in the picture above.

Make will rebuild all of the necessary dependencies and targets on the way towards the target. This way, make will only rebuild those dependencies that actually affect something, and thus, will save time and effort. In big projects, the amount of time saved is significant.

To illustrate this, suppose that **file3.c** in the above picture is modified. After that, make is run, and it will automatically rebuild the necessary targets (**file3.o**, **libutil.a** and **app**):



Now suppose that another function is added to **file1.c**. Then also **util.h** would need to be modified accordingly. The picture shows that quite many objects and targets depend on this header file, so a sizable number of objects need to be rebuilt (but not all):



N.B. In the example pictures, there is a project with a custom static library, which will be linked against the test application.

4.2.1 Simplest Real Example

Before delving too deeply into the syntax of makefiles, it is instructive to first see make in action. For this, a simple project will be used, written in the C language.

In C, it is customary to write "header" files (conventional suffix for them is `.h`) and regular source files (`.c`). The header files describe calling conventions, APIs and structures that are to be made usable for the outside world. The `.c` files contain the implementation of these interfaces.

The first rule in this is: if something is changed in the interface file, then the binary file containing the code implementation (and other files that use the same interface) should be regenerated. Regeneration in this case means invoking gcc to create the binary file out of the C-language source file.

Make needs to be told two things at this point:

- If a file's content changes, which other files will be affected? Since a single interface will likely affect multiple other files, make uses a reversed ordering here. For each resulting file, it is necessary to list the files on which this one file depends on.
- What are the commands to regenerate the resulting files when the need arises?

This example deals with that as simply as possible. There is a project that consists of two source files and one header file. The contents of these files are listed below:

```

/**
 * The old faithful hello world.
 *
 * 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.
 */

#include <stdlib.h> /* EXIT_* */
#include "hello_api.h" /* sayhello */

```

```
int main(int argc, char **argv) {
    sayhello();

    return EXIT_SUCCESS;
}
```

Listing 4.1: simple-make-files/hello.c

```
/**
 * Implementation of sayhello.
 *
 * 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.
 */

#include <stdio.h>      /* printf */
#include "hello_api.h" /* sayhello declaration */

void sayhello(void) {
    printf("Hello world!\n");
}
```

Listing 4.2: simple-make-files/hello_func.c

```
/**
 * Interface description for the hello_func module.
 *
 * 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.
 */

#ifndef INCLUDE_HELLO_API_H
#define INCLUDE_HELLO_API_H
/* The above is protection against circular header inclusion. */

/* Function to print out "Hello world!\n". */
extern void sayhello(void);

#endif
/* ifndef INCLUDE_HELLO_API_H */
```

Listing 4.3: simple-make-files/hello_api.h

So, in effect, there is the main application in **hello.c**, which uses a function that is implemented in **hello_func.c** and declared in **hello_api.h**. Building an application out of these files could be performed manually like this:

```
gcc -Wall hello.c hello_func.c -o hello
```

Or, it could be done in three stages:

```
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

In both cases, the resulting application (**hello**) will be the same.

In the first case, gcc is instructed to process all source files in one go, link the resulting object code and store that code (program in this case) into **hello**.

In the second case, gcc is instructed to create a binary object file for each of the source files. After that, gcc is instructed to link these output files (**hello.o** and **hello_func.o**) together, and store the linked code into **hello**.

N.B. When gcc reads through the C source files, it will also read in the header files, since the C code uses the #include -preprocessor directive. This is because gcc will internally run all files ending with .c through cpp (the preprocessor) first.

The file describing this situation to make is as follows:

```
# define default target (first target = default)
# it depends on 'hello.o' (which will be created if necessary)
# and hello_func.o (same as hello.o)
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from it)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o
#
```

Listing 4.4: simple-make-files/Makefile

This file is in the simplest form without using any variables or relying on built-in magic in GNU make. Later it will be shown that the same rules could be written in a much shorter way.

This makefile can be tested by running make in the directory that contains the makefile and the source files:

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

The resulting files after running make:

```
user@system:~$ ls -la
total 58
drwxr-xr-x 3 user user 456 Aug 11 15:04 .
drwxr-xr-x 13 user user 576 Aug 11 14:56 ..
-rw-r--r-- 1 user user 699 Jun 1 14:48 Makefile
-rwxr-xr-x 1 user user 4822 Aug 11 15:04 hello
-rw-r--r-- 1 user user 150 Jun 1 14:48 hello.c
-rw-r--r-- 1 user user 824 Aug 11 15:04 hello.o
-rw-r--r-- 1 user user 220 Jun 1 14:48 hello_api.h
-rw-r--r-- 1 user user 130 Jun 1 14:48 hello_func.c
-rw-r--r-- 1 user user 912 Aug 11 15:04 hello_func.o
```

Executing the binary:

```
user@system:~$ ./hello
Hello world!
```

4.2.2 Anatomy of Makefile

From the simple example above, some of make's syntax can be deduced.

Here are the rules that can be learned:

- Comments are lines that start with the #-character. To be more precise, when make reads the makefile, it will ignore the #-character and any characters after it up to the end of the line. This means that comments can also be put at the end of lines, and make will ignore them, but this is considered bad practice as it would lead to subtle problems later on.
- The backslash character (\) can be used to escape the special meaning of the next character. The most important special character in makefiles is the dollar character (\$), which is used to access the contents of variables. There are also other special characters. To continue a line that is too long, the newline character can be escaped on that line. When the backslash is put at the end of the line, make will ignore the newline when reading input.
- Empty lines by themselves are ignored.
- A line that starts at column 0 (start of the line) and contains a colon character (:) is considered a rule. The name on the left side of the colon is created by the commands. This name is called a target. Any filenames specified after the colon are the files that the target depends on. They are called prerequisites (i.e. they are required to exist, before make decides to create the target).
- Lines starting with the tabulation character (tab) are commands that make will run to achieve the target.

In the printed material, the tabs are represented with whitespace, so be careful when reading the example makefiles. Note also that in reality, the command lines are considered as part of the rule.

Using these rules, it can now be deduced that:

- make will regenerate **hello** when either or both of its prerequisites change. So, if either **hello.o** or **hello_func.o** change, make will regenerate **hello** by using the command: `gcc -Wall hello.o hello_func.o -o hello`
- If either **hello.c** or **hello_api.h** change, make will regenerate **hello.o**.
- If either **hello_func.c** or **hello_api.h** change, make will regenerate **hello_func.o**.

4.2.3 Default Goal

It should be noted that make was not explicitly told what it should do by default when run without any command line parameters (as was done above). How does it know that creating **hello** is the target for the run?

The first target given in a makefile is the default target. A target that achieves some higher-level goal (like linking all the components into the final application) is sometimes called a goal in GNU make documentation.

So, the first target in the makefile is the default goal when make is run without command line parameters.

N.B. In a magic-like way, make will automatically learn the dependencies between the various components and deduce that in order to create **hello**, it will also need to create **hello.o** and **hello_func.o**. Make will regenerate the missing prerequisites when necessary. In fact, this is a quality that causes make do its magic.

Since the prerequisites for hello (hello.o and hello_func.o) do not exist, and hello is the default goal, make will first create the files that the hello target needs. This can be evidenced from the screen capture of make running without command line parameters. It shows the execution of each command in the order that make decides is necessary to satisfy the default goal's prerequisites, and finally create the hello.

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

4.2.4 On Names of Makefiles

The recommended name for the makefile is **Makefile**. This is not the first filename that GNU make will try to open, but it is the most portable one. In fact, the order of filenames that make attempts to open is: **GNUMakefile**, **Makefile** and finally **makefile**.

Unless one is sure that one's makefile will not work on other make systems (not GNU), one should refrain from using GNUMakefile. Makefile will be used here for the most of this material. The idea behind using Makefile instead of makefile is related to how shells and commands sort filenames, when contents of directories are requested. Since in ASCII the uppercase letters come before the lowercase letters, the important files are listed first. Makefiles are considered important, since they are the basis for building the project. (The collation order might be different in your locale and your environment.)

Make can explicitly be told which file to read by using the -f command line option. This option will be used in the next example.

4.2.5 Questions

Based on common sense, what should make do when it is run after:

- Deleting the hello file?
- Deleting the hello.o file?
- Modifying hello_func.c?
- Modifying hello.c?
- Modifying hello_api.h?
- Deleting the Makefile?

What would be done when writing the commands manually?

4.2.6 Adding Make Goals

Sometimes it is useful to add targets, whose execution does not result in a file, but instead causes some commands to be run. This is commonly used in makefiles of software projects to get rid of the binary files, so that building can be attempted from a clean state. These kinds of targets can also be justifiably called goals. GNU documentation uses the name "phony target" for these kinds of targets, since they do not result in creating a file, like normal targets do.

The next step is to create a copy of the original makefile, and add a goal that will remove the binary object files and the application. N.B. Other parts of the makefile do not change.

```
# add a cleaning target (to get rid of the binaries)

# define default target (first target = default)
# it depends on 'hello.o' (which must exist)
# and hello_func.o
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

# This is the definition of the target 'clean'
# Here we'll remove all the built binaries and
# all the object files that we might have generated
# Notice the -f flag for rm, it means "force" which
# in turn means that rm will try to remove the given
# files, and if there are none, then that's ok. Also
# it means that we have no writing permission to that
# file and have writing permission to the directory
# holding the file, rm will not then ask for permission
# interactively.
clean:
    rm -f hello hello.o hello_func.o
```

Listing 4.5: simple-make-files/Makefile.2

In order for make to use this file instead of the default Makefile, the `-f` command line parameter needs to be used as follows:

```
user@system:~$ make -f Makefile.2
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

To control which target make will process (instead of the default goal), target name needs to be given as command line parameter like this:

```
user@system:~$ make -f Makefile.2 clean
rm -f hello hello.o hello_func.o
```

No binary files remain in the directory:

```
user@system:~$ ls -la
total 42
drwxr-xr-x 3 user user 376 Aug 11 15:08 .
drwxr-xr-x 13 user user 576 Aug 11 14:56 ..
-rw-r--r-- 1 user user 699 Jun 1 14:48 Makefile
-rw-r--r-- 1 user user 1279 Jun 1 14:48 Makefile.2
-rw-r--r-- 1 user user 150 Jun 1 14:48 hello.c
-rw-r--r-- 1 user user 220 Jun 1 14:48 hello_api.h
-rw-r--r-- 1 user user 130 Jun 1 14:48 hello_func.c
```

4.2.7 Making One Target at a Time

Sometimes it is useful to ask make to process only one target. Similar to the clean case, it just needs the target name to be given on the command line:

```
user@system:~$ make -f Makefile.2 hello.o
gcc -Wall -c hello.c -o hello.o
```

Multiple target names can also be supplied as individual command line parameters:

```
user@system:~$ make -f Makefile.2 hello_func.o hello
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

This can be done with any number of targets, even phony ones. Make will try and complete all of them in the order that they are on the command line. Should any of the commands within the targets fail, make will abort at that point, and will not pursue the rest of the targets.

4.2.8 PHONY Keyword

Suppose that for some reason or another, a file called **clean** appears in the directory in which make is run with clean as the target. In this case, make would probably decide that since clean already exists, there is no need to run the commands leading to the target, and in this case, make would not run the rm command at all. Clearly this is something that needs to be avoided.

For these cases, GNU make provides a special target called **.PHONY** (note the leading dot). The real phony targets (clean) will be listed as a dependency to .PHONY. This will signal to make that it should never consider a file called clean to be the result of the phony target.

In fact, this is what most open source projects that use make do.

This leads in the following makefile (comments omitted for brevity):

```
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
```



```
rm -f hello hello.o hello_func.o
```

Listing 4.6: simple-make-files/Makefile.3

4.2.9 Specifying Default Goal

Make will use the first target in a makefile as its default goal. What if there is a need to explicitly set the default goal instead? Since it is not possible to actually change the "first target is the default goal" setting in make, this needs to be taken into account. So, a new target has to be added, and made sure that it will be processed as the first target in a makefile.

In order to achieve this, a new phony target should be created, and the wanted targets should be listed as the phony target's prerequisites. Any name can be used for the target, but all is a very common name for this use. The only thing that needs to be remembered is that this target needs to be the first one in the makefile.

```
.PHONY: all
all: hello

hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
    rm -f hello hello.o hello_func.o
```

Listing 4.7: simple-make-files/Makefile.4

Something peculiar can be seen in the listing above. Since the first target is the default goal for make, would that not make .PHONY now to be the default target? Since .PHONY is a special target in GNU make, this is safe. Also, because of compatibility reasons, GNU make will ignore any targets that start with a leading dot (.) when looking for the default goal.

4.2.10 Other Common Phony Goals

Many other phony goals will be encountered in makefiles that projects use.

Some of the more common ones include:

- **install:** Prerequisites for install is the software application binary (or binaries). The commands (normally *install* is used on Linux) will specify which binary file to place under which directory on the system (/usr/bin, /usr/local/bin, etc).
- **distclean:** Similar to clean; removes object files and such, but removes other files as well (sounds scary). Normally this is used to implement the removal of Makefile in the case that it was created by some automatic tool (*configure* for example).

- package: Prerequisites are as in install, but instead of installing, creates an archive file (*tar*) with the files in question. This archive can then be used to distribute the software.

Other common phony targets can be found in the [GNU make manual](#) [32].

4.2.11 Variables in Makefiles

So far these files have been listing filenames explicitly. Writing makefiles this way can get rather tedious, if not error prone.

This is why all make programs (not just the GNU variant) provide variables. Some other make programs call them macros.

Variables work almost as they do inside regular UNIX command shells. They are a simple mechanism, by which a piece of text can be associated with a name that can be referenced later on in multiple places in the makefiles. Make variables are based on text replacement, just like shell variables are.

4.2.12 Variable Flavors

The variables that can be used in makefiles come in two basic styles or flavors.

The default flavor is where referencing a variable will cause make to expand the variable contents at the point of reference. This means that if the value of the variable is some text in which other variables are referenced, their contents will also be replaced automatically. This flavor is called recursive.

The other flavor is simple, meaning that the content of a variable is evaluated only once, when the variable is defined.

Deciding on which flavor to use might be important, when the evaluation of variable contents needs to be repeatable and fast. In these cases, simple variables are often the correct solution.

4.2.13 Recursive Variables

Here are the rules for creating recursive variables:

- Names must contain only ASCII alphanumerics and underscores (to preserve portability).
- Only lowercase letters should be used in the names of the variables. Make is case sensitive, and variable names written in capital letters are used when it is necessary to communicate the variables outside make, or their origin is from outside (environment variables). This is, however, only a convention, and there will also be variables that are local to the makefile, but still written in uppercase.
- Values may contain any text. The text will be evaluated by make when the variable is used, not when it is defined.
- Long lines may be broken up with a backslash character (\) and newline combination. This same mechanism can be used to continue other long lines as well (not just variables). Do not put any whitespace after the backslash.

- A variable that is being defined should not be referenced in its text portion. This would result in an endless loop whenever this variable is used. If this happens GNU make will stop to an error.

Now the previously used makefile will be reused, but some variables will be introduced. This also shows the syntax of defining the variables, and how to use them (reference them):

```
# define the command that we use for compilation
CC = gcc -Wall

# which targets do we want to build by default?
# note that 'targets' is just a variable, its name
# does not have any special meaning to make
targets = hello

# first target defined will be the default target
# we use the variable to hold the dependencies
.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
    $(CC) hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    $(CC) -c hello_func.c -o hello_func.o

# we'll make our cleaning target more powerful
# we remove the targets that we build by default and also
# remove all object files
.PHONY: clean
clean:
    rm -f $(targets) *.o
```

Listing 4.8: simple-make-files/Makefile.5

The CC variable is the standard variable to hold the name of the C compiler executable. GNU make comes preloaded with a list of tools that can be accessed in similar way (as will be shown with \$(RM) shortly, but there are others). Most UNIX tools related to building software already have similar pre-defined variables in GNU make. Here one of them will be overridden for no other reason than to demonstrate how it is done. Overriding variables like this is not recommended, since the user running make later might want to use some other compiler, and would have to edit the makefile to do so.

4.2.14 Simple Variables

Suppose you have a makefile like this:

```
CC = gcc -Wall

# we want to add something to the end of the variable
CC = $(CC) -g

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o
```

Listing 4.9: simple-make-files/Makefile.6

What might seem quite logical to a human reader, will not seem very logical to make.

Since the contents of recursive variables are evaluated when they are referenced, it can be seen that the above fragment will lead to an infinite loop.

How can this be corrected? Make actually provides two mechanisms for this. This is the solution with simple variables:

```
CC := gcc -Wall
# we want to add something to the end of the variable
CC := $(CC) -g
hello.o: hello.c hello_api.h
        $(CC) -c hello.c -o hello.o
```

Listing 4.10: simple-make-files/Makefile.7

Notice that the equals character (=) has been changed into := .

This is how simple variables are created. Whenever a simple variable is referenced, make will just replace the text that is contained in the variable without evaluating it. It will do the evaluation only when the variable is defined. In the above example, CC is created with the content of gcc -Wall (which is evaluated, but is just plain text), and when the CC variable is defined next time, make will evaluate \$(CC) -g which will be replaced with gcc -Wall -g, as one might expect.

So, the only two differences between the variable flavors are:

- When defining simple variables, := is used.
- make evaluates the contents, when the simple variable is defined, not when it is referenced later.

Most of the time it is advisable to use the recursive variable flavor, since it does what is wanted.

There was a mention about two ways of appending text to an existing variable. The other mechanism is the += operation as follows:

```
CC = gcc -Wall
# we want to add something to the end of the variable
CC += -g
hello.o: hello.c hello_api.h
        $(CC) -c hello.c -o hello.o
```

Listing 4.11: simple-make-files/Makefile.8

The prepending operation can also be used with simple variables. Make will not change the type of variable on the left side of the += operator.

4.2.15 Automatic Variables

There is a pre-defined set of variables inside make that can be used to avoid repetitive typing, when writing out the commands in a rule.

This is a list of the most useful ones:

- `$<` : replaced by the first prerequisite of the rule.
- `^` : replaced by the list of all prerequisites of the rule.
- `$@` : replaced by the target name.
- `$?` : list of prerequisites that are newer than the target is (if target does not exist, they are all considered newer).

By rewriting the makefile using automatic variables, the result is:

```
# add the warning flag to CFLAGS-variable
CFLAGS += -Wall

targets = hello

.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
      $(CC) $^ -o $@

hello.o: hello.c hello_api.h
      $(CC) $(CFLAGS) -c $< -o $@

hello_func.o: hello_func.c hello_api.h
      $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
      $(RM) $(targets) *.o
```

Listing 4.12: simple-make-files/Makefile.9

Notice the cases when `^` is used instead of `<` in the above snippet. It is not desired to pass the header files for compilation to the compiler, since the source file already includes the header files. For this reason, `<` is used. On the other hand, when linking programs and there are multiple files to put into the executable, `^` would normally be used.

This relies on a couple of things:

- `$(RM)` and `$(CC)` will be replaced with paths to the system compiler and removal commands. `$(CFLAGS)` is a variable that contains a list of options to pass whenever make will invoke the C compiler.

It can also be noticed that all these variable names are in uppercase. This signifies that they have been communicated from the system environment to make. In fact, if creating an environment variable called `CFLAGS`, make will create it internally for any makefile that it will process. This is the mechanism to communicate variables into makefiles from outside.

Writing variables in uppercase is a convention signaling external variables, or environmental variables, so this is the reason why lowercase should be used in own private variables within a Makefile.

4.2.16 Integrating with Pkg-Config

The above has provided the knowledge to write a simple Makefile for the Hello World example. In order to get the the result of `pkg-config`, the GNU make

`$(shell command params)` function will be used here. Its function is similar to the backtick operation of the shell.

```
# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

# get the necessary flags for compiling
PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
# get the necessary flags for linking
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

# additional flags
# -Wall: warnings
# -g: debugging
ADD_CFLAGS := -Wall -g

# combine the flags (so that CFLAGS/LDFLAGS from the command line
# still work).
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

hildon_helloworld-1: hildon_helloworld-1.o
    $(CC) $^ -o $@ $(LDFLAGS)

hildon_helloworld-1.o: hildon_helloworld-1.c
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
    $(RM) $(targets) *.o
```

Listing 4.13: simple-make-files/Makefile.10

One might wonder, where the CC and RM variables come from. They certainly were not declared anywhere in the Makefile. GNU make comes with a list of pre-defined variables for many programs and their arguments. GNU make also contains a preset list of pattern rules (which will not be dealt here), but basically these are pre-defined rules that are used when (for example) one needs an .o file from an .c file. The rules that were manually specified above are actually the same rules that GNU make already contains, so the Makefile can be made even more compact by only specifying the dependencies between files.

This leads to the following, slightly simpler version:

```
# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

ADD_CFLAGS := -Wall -g

# combine the flags
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)
```

```

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

# we can omit the rules and just specify the dependencies
# for our simple project this doesn't seem like a big win
# but for larger projects where you have multiple object
# files, this will save considerable time.
hildon_helloworld-1: hildon_helloworld-1.o
hildon_helloworld-1.o: hildon_helloworld-1.c

.PHONY: clean
clean:
    $(RM) $(targets) *.o

```

Listing 4.14: simple-make-files/Makefile.11

4.3 GNU Autotools

Creating portable software written in the C language has historically been challenging. The portability issues are not restricted just to the differences between binary architectures, but also encompass differences in system libraries and different implementations of UNIX APIs in different systems. This section introduces GNU autotools as one solution for managing this complexity. There are also other tools available, but the most likely to be encountered are autotoolized projects and also Debian packaging supports autotools.

4.3.1 Brief History of Managing Portability

When discussing portability challenges with C code, various issues will crop up:

- Same library function has differing prototypes in different UNIX-style systems.
- Same library function has differing implementations between systems (this is a hard problem).
- Same function can be found in different libraries in different UNIX systems, and because of this, the linker parameters need to be modified when making the final linking of the project.
- Besides these not so obvious problems, there is also of course the problem of potentially different architecture, and hence different sizes for the standard C data types. This problem is best fixed by using GLib (as in this case), so it will not be covered here.
- GLib also provides the necessary macros to do endianness detection and conversion.

The historical solutions to solve these problems can be roughly grouped as follows:

- The whole software is built with one big shell script, which will try to pass the right options to different compilers and tools. Most of these scripts have luckily remained in-house, and never seen the light of the world. They are very hard to maintain.
- Makefiles (GNU or other kind) allow some automation in building, but do not directly solve any of the portability issues. The classical solution was to ship various makefiles already tailored for different UNIX systems, and it was the end user's responsibility to select the appropriate Makefile to use for their system. This quite often also required editing the Makefile, so the end user needed to be knowledgeable in UNIX programming tools. A good example of this style is found in the game nethack (if looking at a suitably old version).
- Some parts of the Makefile selection above can be automated by a suitable script that tries to guess the system on which it is running and select a suitably prepared Makefile. Maintaining such scripts is quite hard, as there were historically so many different kinds of systems (different library versions, different compilers, etc.).
- Even the creation of Makefiles can be automated from such guessing script. Such scripts are normally called **configure** or **Configure**, and are marked executable for the sake of convenience. The name of the script does not automatically mean that the script is related to GNU autotools.
- Two major branches of such master configure scripts evolved, each operating a bit differently and suiting differing needs.
- These two scripts and a third guessing script were then combined into GNU autoconf. Since this happened many years ago, most of the historical code has already been purged from GNU autoconf.

Besides autoconf, it became evident that a more general Makefile generation tool could be useful as part of the whole process. This is how GNU automake was born. It can also be used outside autoconf. Automake will be covered a bit later, but first there will be a look at a simple autoconf configuration file (historically called a driver, but this is an obsolete term now).

4.3.2 GNU Autoconf

Autoconf is a tool that will use the GNU m4 macro preprocessor to process the configuration file and output a shell script based on the macros used in the file. Anything that m4 does not recognize as a macro, will be passed verbatim to the output script. This means that almost any wanted shell script fragments can be included into the **configure.ac** (the modern name for the default configuration file for autoconf).

The first subject here is a simple example to see how the basic configuration file works. Then some limitations of m4 syntax will be covered, and hints on how to avoid problems with the syntax will be given.

```
# Specify the "application name" and application version
AC_INIT(hello, version-0.1)
```



```

# Since autoconf will pass through anything that it does not recognize
# into the final script ('configure'), we can use any valid shell
# statements here. Note that you should restrict your shell to
# standard features that are available in all UNIX shells, but in our
# case, we are content with the most used shell on Linux systems
# (bash).
echo -e "\n\nHello from configure (using echo)!\n\n"

# We can use a macro for this messages. This is much preferred as it
# is more portable.
AC_MSG_NOTICE([Hello from configure using msg-notice!])

# Check that the C Compiler works.
AC_PROG_CC
# Check what is the AWK-program on our system (and that one exists).
AC_PROG_AWK
# Check whether the 'cos' function can be found in library 'm'
# (standard C math library).
AC_CHECK_LIB(m, cos)
# Check for presence of system header 'unistd.h'.
# This will also test a lot of other system include files (it is
# semi-intelligent in determining which ones are required).
AC_CHECK_HEADER(unistd.h)
# You can also check for multiple system headers at the same time,
# but notice the different name of the test macro for this (plural).
AC_CHECK_HEADERS([math.h stdio.h])
# A way to implement conditional logic based on header file presence
# (we do not have a b0rk.h in our system).
AC_CHECK_HEADER(b0rk.h, [echo "b0rk.h present in system"], \
    [echo "b0rk.h not present in system"])

echo "But that does not stop us from continuing!"

echo "Directory to install binaries in is '$bindir'"
echo "Directory under which data files go is '$datadir'"
echo "For more variables, check 'config.log' after running configure"
echo "CFLAGS is '$CFLAGS'"
echo "LDFLAGS is '$LDFLAGS'"
echo "LIBS is '$LIBS'"
echo "CC is '$CC'"
echo "AWK is '$AWK'"

```

Listing 4.15: autoconf-automake/example1/configure.ac

The listing is verbosely commented, so it should be pretty self-evident what the different macros do. The macros that test for a feature or an include file will normally cause the generated configure script to generate small C code test programs that will be run as part of the configuration process. If these programs run successfully, the relevant test will succeed, and the configuration process will continue to the next test.

The following convention holds true in respect to the names of macros that are commonly used and available:

- **AC_***: A macro that is included in autoconf or is meant for it.
- **AM_***: A macro that is meant for automake.
- **Others**: The autoconf system can be expanded by writing own macros which can be stored in one's own directory. Also some development packages install new macros to use. One example of this will be covered later on.

The next step is to run *autoconf*. Without any parameters, it will read **configure.ac** by default. If **configure.ac** does not exist, it will try to read **configure.in** instead. N.B. Using the name **configure.in** is considered obsolete; the reason for this will be explained later.

```
[sbox-DIABLO_X86: ~/example1] > autoconf
[sbox-DIABLO_X86: ~/example1] > ls -l
total 112
drwxr-xr-x 2 user user 4096 Sep 16 05:14 autom4te.cache
-rwxrwxr-x 1 user user 98683 Sep 16 05:14 configure
-rw-r--r-- 1 user user 1825 Sep 15 17:23 configure.ac
[sbox-DIABLO_X86: ~/example1] > ./configure

Hello from configure (using echo)!

configure: Hello from configure using msg-notice!
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for gawk... gawk
checking for cos in -lm... yes
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking math.h usability... yes
checking math.h presence... yes
checking for math.h... yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
checking b0rk.h usability... no
checking b0rk.h presence... no
checking for b0rk.h... no
b0rk.h not present in system
But that doesn't stop us from continuing!
Directory to install binaries in is '${exec_prefix}/bin'
Directory under which data files go is '${prefix}/share'
For more variables, check 'config.log' after running configure
CFLAGS is '-g -O2'
LDFLAGS is ''
LIBS is '-lm'
CC is 'gcc'
AWK is 'gawk'
```

Autoconf will not output information about its progress. Normally, only errors are output on stdout. Instead, it will create a new script called **configure** in the same directory, and set it as executable.

The **configure** script is the result of all of the macro processing. If taking a look at the generated file with an editor or by using *less*, it can be seen that it contains a lot of shell code.

Unless one is experienced in reading convoluted shell code, it is best not

to try to understand what is attempted at the various stages. Normally the generated file is not fixed or modified manually, since it would be overwritten anyway the next time that someone runs autoconf.

When the generated script is executed, the output of various parts can be seen in the order that they were used in the configuration file.

N.B. Autoconf (because of m4) is an imperative language, which means that it will execute the commands one by one when it detects them. This is in contrast to declarative languages, like Makefiles.

M4-syntax notes:

- Public autoconf M4 macros all start with A[CST]_* .
- Private macros start with an underscore, but they should not be used, since they will change from one autoconf version to the other (using undocumented features is bad style anyway).
- m4 uses [and] to quote arguments to functions, but in most cases quoting is not needed. It is best to avoid using brackets, unless the macro does not seem to work properly otherwise. When writing new macros, using brackets will become more important, but this material does not cover creating custom macros.
- If it is absolutely necessary to pass brackets to the generated script, there are three choices:
 - @<:@ is same as [, and @>:@ is same as]
 - [] will expand into [] (most of time)
 - avoid [and] (most command line tools and shell do not really require them)
- Since m4 will use brackets for its own needs, the [command cannot be used to test things in scripts, but instead test has to be used (which is more clear anyway). This is why the brackets are escaped with the rules given above, if they really are needed to be output into the generated shell script.

If bad shell syntax is introduced into the configuration file, the bad syntax will cause errors only when the generated script file is run (not when autoconf generates the script). In general, autoconf will almost always succeed, but the generated script might not. It is not always simple to know which error in the shell script corresponds to which line in the original configure.ac, but experience will teach that.

There can be noticed a line that reports the result of testing for **unistd.h**. It will appear twice: the first time because it is the default test to run whenever testing for headers, and the second time because it was explicitly tested for. The second test output contains text (cached), which means that the test has been already run, and the result has been saved into a cache (the mysterious **autom4te.cache** directory). This means that for large projects, which might do the same tests over and over, the tests are only run once and this will make running the script quite a bit faster.

The last line's output above contains the values of variables. When the configure script runs, it will automatically create shell variables that can be

used in shell code fragments. The macros for checking what programs are available for compiling should illustrate that point. Here `awk` was used as an example.

The `configure` script will take the initial values for variables from the environment, but also contains a lot of options that can be given to the script, and using those will introduce new variables that can also be used. Anyone compiling modern open source projects will be familiar with options like `prefix` and similar. Both of these cases are illustrated below:

```
[sbox-DIABLO_X86: ~/example1] > CFLAGS='-O2 -Wall' ./configure --prefix=/usr
...
Directory to install binaries in is '${exec_prefix}/bin'
Directory under which data files go is '${prefix}/share'
For more variables, check 'config.log' after running configure
CFLAGS is '-O2 -Wall'
LDFLAGS is ''
LIBS is '-lm'
CC is 'gcc'
AWK is 'gawk'
```

It might seem that giving the `prefix` does not change anything, but this is because the shell does not expand the value in this particular case. It would expand the value later on, if the variable was used in the script for doing something. In order to see some effect, one can try passing the `datadir`-option (because that is printed out explicitly).

If interested in the other that variables are available for `configure`, one can read the generated **config.log** file, since the variables are listed at the end of that file.

4.3.3 Substitutions

Besides creating the `configure` script, `autoconf` can do other useful things as well. Some people say that `autoconf` is at least as powerful as `emacs`, if not more so. Unfortunately, with all this power comes also a lot of complexity. Sometimes it may be difficult to find out why things do not quite work.

Sometimes it is useful to use the variables inside text files that are not directly related to the **configure.ac**. These might be configuration files or files that will be used in some part of the building process later on. For this, `autoconf` provides a mechanism called *substitution*. There is a special macro that will read in an external file, replace all instances of variable names in it, and then store the resulting file as a new file. The convention in naming the input files is to add a suffix `.in` to the names. The name of generated output file will be the same, but without the suffix. N.B. The substitution will be done when the generated `configure` script is run, not when `autoconf` is run.

The generated `configure` script will replace all occurrences of the variable name surrounded with `'at'` (`@`) characters with the variable value when it reads through each of the input files.

This is best illustrated with a small example. The input file contents are listed after the `autoconf` configuration file. In this example, the substitution will only be made for one file, but it is also possible to process multiple files using the substitution mechanism.

```
# An example showing how to use a variable-based substitution.
AC_INIT(hello, version-0.1)
```

```

AC_PROG_CC
AC_PROG_AWK
AC_CHECK_HEADER(unistd.h)
AC_CHECK_HEADERS([math.h stdio.h])
AC_CHECK_HEADER(b0rk.h, [echo "b0rk.h present in system"], \
                        [echo "b0rk.h not present in system"])

echo "But that doesn't stop us from continuing!"

# Run the test-output.txt(.in) through autoconf substitution logic.
AC_OUTPUT(test-output.txt)

```

Listing 4.16: autoconf-automake/example2/configure.ac

```

This file will go through autoconf variable substitution.

The output file will be named as 'test-output.txt' (the '.in'-suffix
is stripped).

All the names surrounded by '@'-characters will be replaced by the
values
of the variables (if present) by the configure script, when it is run
by the end user.

Prefix: @prefix@
C compiler: @CC@

This is a name for which there is no variable.
Stuff: @stuff@

```

Listing 4.17: autoconf-automake/example2/test-output.txt.in

We then run autoconf and configure:

```

[sbox-DIABLO_X86: ~/example2] > autoconf
[sbox-DIABLO_X86: ~/example2] > ./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for gawk... gawk
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking math.h usability... yes
checking math.h presence... yes
checking for math.h... yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
checking b0rk.h usability... no
checking b0rk.h presence... no
checking for b0rk.h... no
b0rk.h not present in system
But that doesn't stop us from continuing!
configure: creating ./config.status
config.status: creating test-output.txt
[sbox-DIABLO_X86: ~/example2] > cat test-output.txt
This file will go through autoconf variable substitution.

The output file will be named as 'test-output.txt' (the '.in'-suffix
is stripped).

All names surrounded by '@'-characters will be replaced by the values
of the variables (if present) by the configure script when it is run
by end user.

Prefix: /usr/local
C compiler: gcc

This is a name for which there is no variable.
Stuff: @stuff@

```

This feature will be used later on. When running one's own version of the file, one might notice the creation of file called **config.status**. It is the file that actually does the substitution for external files, so if the configuration is otherwise complex, and one only wants to re-run the substitution of the output files, one can run the config.status script.

4.3.4 Introducing Automake

The next step is to create a small project that consists of yet another hello world. This time, there will be a file implementing the application (main), a header file describing the API (printHello()) and the implementation of the function.

As it happens, GNU automake is designed so that it can be easily integrated into autoconf. This will be utilized in the next example, so that the Makefile does not have to be written by hand anymore. Instead, the Makefile will be

generated by the configure script, and it will contain the necessary settings for the system on which configure is run.

In order for this to work, two things are necessary:

- A configuration file for automake (conventionally Makefile.am).
- Telling the autoconf that it should create Makefile.in based on Makefile.am by using automake.

The example starts with the autoconf configuration file:

```
# Any source file name related to our project is ok here.
AC_INIT(helloapp, 0.1)

# We're using automake, so we init it next. The name of the macro
# starts with 'AM' which means that it is related to automake ('AC'
# is related to autoconf).
# Initiating automake means more or less generating the .in file from
# the .am file although it can also be generated at other steps.
AM_INIT_AUTOMAKE

# Compiler check.
AC_PROG_CC
# Check for 'install' program.
AC_PROG_INSTALL
# Generate the Makefile from Makefile.in (using substitution logic).
AC_OUTPUT(Makefile)
```

Listing 4.18: autoconf-automake/example3/configure.ac

Then the configuration file is presented for automake:

```
# Automake rule primer:
# 1) Left side_ tells what kind of target this will be.
# 2) _right side tells what kind of dependencies are listed.
#
# As an example, below:
# 1) bin = binaries
# 2) PROGRAMS lists the programs to generate Makefile.ins for.
bin_PROGRAMS = helloapp

# Listing source dependencies:
#
# The left side_ gives the name of the application to which the
# dependencies are related to.
# _right side gives again the type of dependencies.
#
# Here we then list the source files that are necessary to build the
# helloapp -binary.
helloapp_SOURCES = helloapp.c hello.c hello.h

# For other files that cannot be automatically deduced by automake,
# you need to use the EXTRA_DIST rule which should list the files
# that should be included. Files can also be in other directories or
# even whole directories can be included this way (not recommended).
#
# EXTRA_DIST = some.service.file.service some.desktop.file.desktop
```

Listing 4.19: autoconf-automake/example3/Makefile.am

This material tries to introduce just enough of automake for it to be useful for small projects. Because of this, the detailed syntax of Makefile.am is not

explained. Based on the above description, automake will know what kind of Makefile.in to create, and autoconf will take it over from there and fill in the missing pieces.

The source files for the project are as simple as possible, but notice the implementation of printHello.

```
/**
 * 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.
 */

#include <stdlib.h>
#include "hello.h"

int main(int argc, char** argv) {

    printHello();

    return EXIT_SUCCESS;
}
```

Listing 4.20: autoconf-automake/example3/helloapp.c

```
/**
 * 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.
 */

#ifndef INCLUDE_HELLO_H
#define INCLUDE_HELLO_H

extern void printHello(void);

#endif
```

Listing 4.21: autoconf-automake/example3/hello.h

```
/**
 * 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.
 */

#include <stdio.h>
#include "hello.h"

void printHello(void) {
    /* Note that these are available as defines now. */
    printf("( PACKAGE " " VERSION ")\n");
    printf("Hello world!\n");
}
```

Listing 4.22: autoconf-automake/example3/hello.c

The PACKAGE and VERSION defines will be passed to the build process automatically, and their use will be shown later.


```
[sbox-DIABLO_X86: ~/example3] > autoconf
configure.ac:10: error: possibly undefined macro: AM_INIT_AUTOMAKE
If this token and others are legitimate, please use m4_pattern_allow.
See the Autoconf documentation.
```

Autoconf is complaining about a macro, for which it cannot find a definition (actually m4 is the program that does the complaining). The problem is that by default, autoconf only knows about built-in macros. When using a macro for integration, or a macro that comes with another package, autoconf needs to be told about it. Luckily, this process is quite painless.

A local **aclocal.m4** file needs to be created into the same directory with the **configure.ac**. When starting, autoconf will read this file, and it is enough to put the necessary macros there.

For this, a utility program called **aclocal** will be used, which will scan the **configure.ac** and copy the relevant macros into the local **aclocal.m4**. The directory for all the extension macros is usually **/usr/share/aclocal/**, which should be checked out at some point.

Now it is time to run **aclocal**, and then try and run **autoconf** again. N.B.: The messages that are introduced into the process from now on are inevitable, since some macros use obsolete features, or have incomplete syntax, and thus trigger warnings. There is no easy solution to this, other than to fix the macros themselves.

```
[sbox-DIABLO_X86: ~/example3] > aclocal
/scratchbox/tools/share/aclocal/pkg.m4:5: warning:
underquoted definition of PKG_CHECK_MODULES
run info '(automake)Extending aclocal' or see
http://sources.redhat.com/automake/automake.html#Extending%20aclocal
/usr/share/aclocal/pkg.m4:5: warning:
underquoted definition of PKG_CHECK_MODULES
/usr/share/aclocal/gconf-2.m4:8: warning:
underquoted definition of AM_GCONF_SOURCE_2
/usr/share/aclocal/audiofile.m4:12: warning:
underquoted definition of AM_PATH_AUDIOFILE
[sbox-DIABLO_X86: ~/example3] > autoconf
[sbox-DIABLO_X86: ~/example3] > ./configure
configure: error: cannot find install-sh or install.sh in
~/example3 ~/ ~/..
```

If one now lists the contents of the directory, one might be wondering, where the **Makefile.in** is. Automake needs to be run manually, so that the file will be created. At the same time, the missing files need to be introduced to the directory (such as the **install.sh** that **configure** seems to complain about).

This is done by executing **automake -ac**, which will create the **Makefile.in** and also copy the missing files into their proper places. *This step will also copy a file called **COPYING** into the directory, which by default will contain the GPL.* So, if the software is going to be distributed under some other license, this might be the correct moment to replace the license file with the appropriate one.

```
[sbox-DIABLO_X86: ~/example3] > automake -ac
configure.ac: installing './install-sh'
configure.ac: installing './missing'
Makefile.am: installing './depcomp'
[sbox-DIABLO_X86: ~/example3] > ./configure
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
```

Notice the second to last line of the output, telling that autoconf just created a Makefile (based on the Makefile.in that *automake* created).

It can prove to be tedious to perform all these steps manually each time when wanting to make sure that all generated files are really generated. Most developers create a script called **autogen.sh**, which will implement the necessary bootstrap procedures for them. Below is a file that is suitable for this example. Real projects might have more steps because of localization and other requirements.

```
#!/bin/sh
#
# A utility script to setup the autoconf environment for the first
# time. Normally this script would be run when checking out a
# development version of the software from SVN/version control.
# Regular users expect to download .tar.gz/tar.bz2 source code
# instead, and those should come with with 'configure' script so that
# users do not require the autoconf/automake tools.
#
# Scan configure.ac and copy the necessary macros into aclocal.m4.
aclocal

# Generate Makefile.in from Makefile.am (and copy necessary support
# files, because of -ac).
automake -ac

# This step is not normally necessary, but documented here for your
# convenience. The files listed below need to be present to stop
# automake from complaining during various phases of operation.
#
# You also should consider maintaining these files separately once
# you release your project into the wild.
#
# touch NEWS README AUTHORS ChangeLog

# Run autoconf (will create the 'configure'-script).
autoconf

echo 'Ready to go (run configure)'
```

Listing 4.23: autoconf-automake/example3/autogen.sh

In the above code, the line with `touch` is commented. This might raise a question. There is a target called `distcheck` that automake creates in the Makefile, and this target checks whether the distribution tarball contains all the necessary files. The files listed on the `touch` line are necessary (even if empty), so they need to be created at some point. Without these files, the penultimate Makefile will complain, when running the `distcheck`-target.

Build the project now and test it out:

```
[sbox-DIABLO_X86: ~/example3] > make
if gcc -DPACKAGE_NAME=\"helloapp\" -DPACKAGE_TARNAME=\"helloapp\"
-DPACKAGE_VERSION=\"0.1\" -DPACKAGE_STRING=\"helloapp 0.1\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"helloapp\" -DVERSION=\"0.1\"
-I. -Iexample3 -g -O2 -MT helloapp.o -MD -MP -MF ".deps/helloapp.Tpo"
-c -o helloapp.o helloapp.c; \ then \
mv -f ".deps/helloapp.Tpo" ".deps/helloapp.Po";
else rm -f ".deps/helloapp.Tpo"; exit 1;
fi
if gcc -DPACKAGE_NAME=\"helloapp\" -DPACKAGE_TARNAME=\"helloapp\"
-DPACKAGE_VERSION=\"0.1\" -DPACKAGE_STRING=\"helloapp 0.1\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"helloapp\" -DVERSION=\"0.1\"
-I. -Iexample3 -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo"
-c -o hello.o hello.c; \ then \
mv -f ".deps/hello.Tpo" ".deps/hello.Po";
else rm -f ".deps/hello.Tpo"; exit 1;
fi
gcc -g -O2 -o helloapp helloapp.o hello.o
[sbox-DIABLO_X86: ~/example3] > ./helloapp
(helloapp 0.1)
Hello world!
[sbox-DIABLO_X86: ~/example3] > make clean
test -z "helloapp" || rm -f helloapp
rm -f *.o
```

4.3.5 Checking for Distribution Sanity

The generated Makefile contains various targets that can be used, when creating distribution tarballs (tar files containing the source code and necessary files to build the software). The most important of these is the `dist`-target, which will by default create a `.tar.gz`-file out of the source, including the configure script and other necessary files (which are specified in `Makefile.am`).

To test whether it is possible to build the software from such distribution tarball, execute the `distcheck`-target. It will first create a distribution tarball, then extract it in a new subdirectory, run `configure` there and try and build the software with default make target. If it fails, the relevant error will be given.

It is recommended to make the `distcheck` target each time before making a `dist` target, so that one can be sure that the distribution tarball can be used outside the source tree. This step is especially critical later, when making Debian packages.

4.3.6 Cleaning up

For the sake of convenience, the `example3` directory also includes a script called `antigen.sh`, which will try its best to get rid of all generated files (it will be necessary to `autogen.sh` the project afterwards).

Having a clean-up script is not very common in open source projects, but it is especially useful when starting autotools development, as it allows testing the toolchain easily from scratch.

The contents of `antigen.sh` that is suitable for simple projects:

```
#!/bin/sh
#
# A utility script to remove all generated files.
#
# Running autogen.sh will be required after running this script since
# the 'configure' script will also be removed.
#
# This script is mainly useful when testing autoconf/automake changes
# and as a part of their development process.
#
# If there's a Makefile, then run the 'distclean' target first (which
# will also remove the Makefile).
if test -f Makefile; then
    make distclean
fi

# Remove all tar-files (assuming there are some packages).
rm -f *.tar.* *.tgz

# Also remove the autotools cache directory.
rm -Rf autom4te.cache

# Remove rest of the generated files.
rm -f Makefile.in aclocal.m4 configure depcomp install-sh missing
```

Listing 4.24: `autoconf-automake/example3/antigen.sh`

4.3.7 Integration with Pkg-Config

The last part that is covered for autoconf is how to integrate pkg-config support into the projects when using `configure.ac`.

Pkg-config comes with a macro package (**pkg.m4**), which contains some useful macros for integration. The best documentation for these can be found in the pkg-config manual pages.

Here only one will be used, `PKG_CHECK_MODULES`, which should be used like this:

```
# Check whether the necessary pkg-config packages are present. The
# PKG_CHECK_MODULES macro is supplied by pkg-config
# (/usr/share/aclocal/).
#
# The first parameter will be the variable name prefix that will be
# used to create two variables: one to hold the CFLAGS required by
# the packages, and one to hold the LDFLAGS (LIBS) required by the
# packages. The variable name prefix (HHW) can be chosen freely.
PKG_CHECK_MODULES(HHW, gtk+-2.0 hildon-1 hildon-fm-2 gnome-vfs-2.0 \
                  gconf-2.0 libosso)
# At this point HHW_CFLAGS will contain the necessary compiler flags
# and HHW_LIBS will contain the linker options necessary for all the
# packages listed above.
#
# Add the pkg-config supplied values to the ones that are used by
# Makefile or supplied by the user running ./configure.
CFLAGS="$HHW_CFLAGS $CFLAGS"
```

```
LIBS="$HHW_LIBS $LIBS"
```

Listing 4.25: Part of configure.ac for pkg-config integration

The proper placing for this code is after all the AC_PROG_* checks and before the AC_OUTPUT macros (so that the build flags may affect the substituted files).

One should also check the validity of the package names by using:

```
pkg-config --list-all
```

to make sure one does not try to get the wrong library information.

Chapter 5

Architecture

This chapter describes the high-level architecture of the maemo platform from an application developer's point of view (see section 5.3 for a more detailed view). There is also a comparison of maemo and popular Ubuntu and Debian desktop Linux distributions in the chapter.

5.1 Introduction

Maemo is based on Debian GNU/Linux operating system, which itself inherits its architecture from the Unix operating system. Linux, GNU and Debian are open source projects (or project umbrellas for multiple separate projects), which embrace sharing of source code, collaboration and open development model. Maemo promotes these values as well by providing an integrated open source based platform for mobile devices, by sharing source code, and by contributing code directly to the upstream projects.

5.1.1 Key Components

Maemo is based on the Linux operating system kernel. Linux is a monolithic kernel that supports multiple hardware platforms and is scalable to be able to support wide range of different kinds of devices from wrist watches to large server systems. Currently all maemo-based devices have OMAP chipsets, which contain a general-purpose ARM processor and a DSP unit. Maemo devices run recent 2.6 kernels.

The user space software links with the standard GNU C library interface. The GNU C library comes with all major Linux distributions except some minimalistic embedded systems which use some other size-optimized C library (like uClibc). The GNU C library aims at POSIX compatibility, and maemo aims at being compatible with mainstream Linux systems as much as possible to minimize the effort of porting applications and application engines, and to bring in the best open source solutions for mobile computers.

The package management framework, the file system hierarchy, and general design policies come from the Debian distribution. Maemo aims at following Debian policies as much as possible. The Debian distribution has by far the largest development community; in September 2007 Debian supported about

a dozen different hardware architectures, and it had more than 18,000 software components available.

The user interface architecture is based on GNOME framework, especially the GTK+ widget set. GNOME is a leading application framework for desktop Linux systems. From the GNOME project, maemo has inherited many central components, including for instance GTK+, GStreamer multimedia framework, GConf configuration management, and XML library. Maemo extends GTK+/GNOME by providing the Hildon extensions for a mobile desktop.

5.1.2 Development Environment

The cross-compilation environment of maemo is based on Scratchbox. Cross-compilation is a problematic issue in Linux systems, since build scripts typically utilize Autotools, which have not been designed well for cross-compilation. Thus, many Linux distributors solve the cross-compilation problem by avoiding it, and use dedicated hardware to run native compilations. This is a big limitation, since sometimes native hardware build environment is difficult to arrange. And even if it was available, builds run typically multiple times slower than in cross-compilation. Scratchbox solves this problem by totally isolating the target and host environments. Autotools-based build scripts can be run on Scratchbox without modifications to build target platform binaries on a host system having a hardware platform other than the target.

Official maemo APIs are provided for the C language. Maemo also has C++ and Python bindings for its core APIs. Other unofficial bindings for other languages and environments exist.

5.2 Software Layers

This section shortly describes the software layers. A more detailed picture is given in the next section (Software Decomposition View).

5.2.1 Operating System Layer and Bootup

The bootloader takes care of some hardware-specific initializations, and then loads the operating system kernel during the early stages of the boot up process. Maemo systems are based on Linux 2.6 operating system. At the last state of the kernel boot process, the InitFS is mounted (a small JFFS2 file system image). It is used during the boot time as the root file system, and in the normal device operation it is mounted into /mnt/initfs. The final root file system is on a JFFS2 image, which is mounted after the InitFS boot scripts are done.

Applications on the InitFS are linked with uClibc, a minimal version of the C library in order to reduce space. The Device State Management Entity (dsme) is also located in the InitFS. The existence of the dsme daemon is vital, so it is used to ping the hardware watchdog as well.

The Linux kernel is the central software component of the system. It provides the hardware abstraction layer for the system devices, memory management, process management, networking services including link and transport layer protocols like TCP/IP, file management including file systems and various other services.

Parts of the kernel functionality can be implemented as dynamically-loadable kernel modules. A kernel module can be loaded or removed during runtime. Kernel functionality, such as device drivers, network protocols, or file systems can be implemented as kernel modules.

The ARM/OMAP-based Linux kernel on maemo devices implements several hardware-specific device drivers and bus drivers on top of the core kernel's virtual services. The device drivers include USB, LCD, WLAN, Camera, and Audio, for instance. The bus drivers include Flash bus, SPI, I2C, and serial bus, for instance.

5.2.2 System Libraries

Maemo is based on the standard GNU C library. Also the Standard C++ library is implemented. For networking security, there is OpenSSL library that provides cryptography, and libcurl that provides HTTP access for applications.

For hardware abstraction, maemo provides HAL (Hardware Abstraction Layer). It provides a shared library that has an API for device objects. A device object has properties, and it is up to an actual device driver, which properties it supports. HAL is thus capable of loading the right device driver, when a new device is detected, creating and maintaining /dev files, tracking the status of devices and providing a means to uses each device.

5.2.3 System Services

The primary communication channel between applications is DBUS. DBUS also provides a channel for interaction between the system and applications. DBUS is also used to invoke all the applications by sending messages.

The system provides an SQL database, SQLite 3, that can be used to store user application data. SQLite database is accessed through a library interface; there is no centralized server process to connect.

5.2.4 Hildon Framework

The user interface is based on X Window System with Matchbox window manager. The application programming API on top of X is a GTK+ widget toolkit with Hildon extensions. GTK+ is the UI framework of the GNOME project as well. Other GNOME components have also been included in maemo, like GConf application configuration management, XML library, GnomeVFS, Evolution Data Server for address book and calendar management, GSF structured file streaming, and SVG (Scalable Vector Graphics). The multimedia framework is also the same, GStreamer-based.

Hildon framework provides components on top of the GNOME components to support control panel, status bar, task navigator, and home applets. Hildon framework also provides backup/restore framework, help framework, and an application manager.

5.2.5 Applications

The applications are built on top of the Hildon framework. Simple applications link just with Hildon libraries, GTK+, Glib, and libosso in order to use the

graphical user interface elements. More complex applications use other services according to their needs, for example, they link with GStreamer for multimedia access and libcurl for HTTP access.

5.3 Software Decomposition View

5.3.1 Software Components

The main components of maemo are presented in table below. More details can be viewed by clicking on the desired component. The component division does not map one-on-one on the actual package division, since some abstraction has been added. In addition, the layers in the picture do not mean actual package dependencies from an upper layer to a lower layer, although some correlation to the actual dependencies does exist (e.g. none of the components below GTK+ depend on it).

Applications						
Fonts		Sounds			Icons	
5.3.1 Connectivity		5.3.1 System UI	5.3.1 Search	5.3.1 Text Input		5.3.1 MIME Types
5.3.1 Home Applets		5.3.1 Control Panel		5.3.1 Task Navigator		5.3.1 Status Bar
5.3.1 Backup		5.3.1 Installer	5.3.1 Alarm	5.3.1 Help		5.3.1 Launcher
XML	E-D-S		Telepathy		GConf	
GStreamer		GnomeVFS				GSF
5.3.1 Sapwood		5.3.1 Hildon Widgets		5.3.1 Hildon File UI		HTML Widget
GTK+						
GDK				GdkPixbuf		
Pango		Cairo			Atk	
GLib				GObject		
Samba	GPS	Obex	5.3.1 ConIC	UPnP	JPEG PNG TIFF SVG	Matchbox
D-BUS		HAL	SQLite	curl HTTP	5.3.1 Clipboard	
SSL	5.3.1 System SW		5.3.1 Cert. mgnt	5.3.1 libosso	X	
Libstd C++		5.3.1 Compression	dpkg	apt	Freetype	Fontconfig
Sysvinit	5.3.1 Base Files	Busybox	GNU C Library	Core Libs	Core Utils	5.3.1 Core Daemons
BlueZ		5.3.1 Power mgnt		WLAN security	ALSA	Video4-Linux
Bootloader		Linux kernel including JFFS2, TCP/IP				InitFS including uClibc dsme

The components marked with yellow color are in general provided as binary only (some subcomponents in them may be provided with source code, though), others come with source code as well. Some specific applications, however, are provided with source code. Some maemo-specific components are described in more detail below.

Maemo Connectivity Subsystem

Components of the maemo Connectivity Architecture include:

- Maemo connectivity UI - User Interfaces parts of the connectivity. This includes Connection manager, Control panel applets and several different dialogs.
- Maemo connectivity daemon (ICd) - LibConIC API works together with ICd that handles all Internet Access Points (IAPs). IC daemon handles both WLAN and Bluetooth connections.
- OBEX wrapper - Interface to OBEX services. The primary target user of this library is the OBEX gnome-vfs module.
- OpenOBEX - Open source implementation of the Object Exchange (OBEX) protocol. See more information on OpenOBEX from: <http://triq.net/obex/>
- BlueZ Bluetooth stack - The de-facto implementation of Bluetooth for Linux. See more information on BlueZ from: <http://www.bluez.org>
- BlueZ D-Bus API - BlueZ accepts commands via D-Bus.
- WLAN connectivity daemon - The daemon that controls WLAN connections.
- WLAN device driver - Device driver for Wireless LAN (IEEE 802.11g). Kernel driver is composed of two parts: a binary part (closed source) and an open source wrapper, which binds the binary to the Linux kernel.

Maemo System User Interface

System UI consists of two parts. One of the parts is the System UI itself, and the other consists of System UI plug-ins.

- System UI provides the main application for system UI plug-ins, including responsibility of plug-in loading and unloading.
- System UI plug-ins provide the user interface to power key menu, splash screen, alarm, device lock, touch screen, keypad lock and mode change.

Maemo Global Search

Maemo Global Search component provides the search framework for maemo.

Maemo Text Input Methods

Embedded devices have special needs for text input. This framework provides the text input methods.

Hildon Home Applets

Home applets (also referred to as plug-ins) are small applications on the main window. They can provide e.g. online weather information, or a view to the latest news items.

Hildon Control Panel

Hildon control panel is the standardized place to put end-user changeable settings for applications, servers etc. in the system.

Hildon Task Navigator

Hildon Task Navigator provides the menu for switching between applications. To make an application visible in the Hildon Task Navigator, a Desktop file for the application is needed. This file contains all the essential information needed to show the application entry in the menu; such as name, binary and D-BUS service name. Name of the file should be [application].desktop, and the location in filesystem `"/usr/share/applications/hildon/"`.

Hildon Status Bar

Status bar is a UI component, displaying the status of various system tasks with a small icon on the main screen. The maemo status bar can contain user-defined items as well. Normally there is a place for two of these additional items. These two slots are by default used by the usb connection indicator and alarm indicator, but can be used by any plug-in. Although plug-ins can specify a priority, the current version of the status bar does not handle the plug-in priorities, so only two newest plug-ins are visible.

MIME Type Registry

This component provides the [MIME type](#) registry.

Maemo Backup

The maemo backup application saves and restores user data stored in `/My-Docs` (by default) and setting directories/files `/etc/osso-af-init/gconf-dir` (a link to GConf database `/var/lib/gconf`), `/etc/osso-af-init/locale`, and `/etc/bluetooth/-name`. It can be configured to back up other locations and files as well with the help of custom configuration files.

The backup application must not to be disrupted by other applications writing or reading during a backup or restore operation. For restore process, backup will therefore, if the user approves, ask the application killer to close all applications, and then wait until it has been done. For backing up, the `backup_start` and `backup_finish` D-BUS signals will be emitted on the session bus, indicating to applications that they should not write to disk.

Maemo Application Manager

Maemo application manager is an application that is capable of managing application installations and upgrading. The architecture underneath is based on `dpkg` and `apt`.

Maemo Alarm Framework

The alarm framework provides a mechanism to manage timed events. Timed event functionality is provided by the `alarmd` daemon. It makes possible for applications to get D-Bus messages or `exec` calls at certain times.

Packages included in the subsystem are:

- `libalarm`
- `alarmd`

Maemo Help Framework

The Help Framework is a centralized way to offer the user help services for a program. Maemo platform has an inbuilt help system that handles all help documentation for the programs, using the Help Framework. For this, there are libraries to register a program to the Help Framework, so that only the content of the actual help documentation needs to be written. An ID tag will be given to the help file, which will be in [XML format](#). This way it is easy control, which help file will be loaded, when the user asks for help, just by calling the right help content ID. When using the Help Framework, the help documentation for a program will also be available, when using maemo Platform Help application.

Maemo Launcher

Maemo launcher launches most applications on the maemo platform. It is there to speed up the application start-up by sharing some of the initialization data of an application start-up. Maemo Launcher is composed of two parts: (I) the `maemo-invoker`, which is executed by D-BUS daemon or scripts to start the given (application) service, and (II) `maemo-launcher`, a server that has initialized most of the data used by the applications.

The `maemo-invoker` asks the `maemo-launcher` to start the actual application. Use of `maemo launcher` requires that the application is compiled as a shared library. There is a set of helper Debian package rules that make an application to "automatically" use `maemo-invoker`, when given suitable build options. As a result, the application binary name is linked to `maemo-invoker` and the application (library) binary name has `.launch` extension. By default, the invoker will wait until the `maemo-launcher` tells it that the application has exited, so that it can return the correct return value for the caller.

The `maemo-launcher` is a server process that has initialized most of the data used by the applications, such as Glib types, Gtk theme and some Gtk widget classes. When it is asked by the `maemo-invoker` to start an application, instead of executing the application binary, it will `dl-load` that as a shared library, fork and call `main()`. With fork, the initialized data is handled as copy-on-write, i.e. shared until it is modified. If the application exits abnormally, the `maemo-launcher` notifies the Desktop, so that the Desktop can inform the user about it.

Because prelinking does not work with `dl-loaded` libraries, the `maemo Launcher` cannot speed up starting of applications, where library linking has a larger effect on the start-up time than AF library initializations. It can still save memory, though.

Sapwood

Sapwood provides a server and client library for accessing theme images. The server is responsible of loading the theme-related images, and distributing them to clients. Sapwood saves memory compared to Pixbuf engine, which does not share the [bitmaps](#) between applications. Sapwood is also much faster, because it tiles the 16-bit images using X server, whereas the pixbuf engine scales the 24/32-bit on the client side and then converts them to 16-bit for X server for blitting.

Hildon Widgets

Hildon widgets provide many GUI extensions over the GTK+ standard widget set. These include application widgets (such as HildonApp, HildonWindow, and HildonProgram), selectors (such as HildonCalenderPopup), editors (such as HildonRange), notifications (such as HildonBanner) etc.

Hildon File UI

This package provides the graphical user interface widget for accessing the file system.

LibOSSO

LibOSSO is a shared library, containing required and helpful services for maemo applications to integrate them better with the platform.

System Software

The System Software subsystem provides system-wide services to applications and users. The services include device state management (dsm), mode control (mce), battery management (bme) and a few graphical user interface elements to manage the behavior of the services.

More detailed description of the components:

- **Device State Management:** Responsible for managing the states of the device, including shutdown and start-up. In addition, dsm is responsible for keeping the device running and operational. This is achieved by monitoring the status of critical processes, such as D-Bus, X11 and Window Manager. Finally, dsm is responsible for tracking inactivity and, based on that, initiating power-saving operations (e.g. turn off the screen).
- **Mode Control:** Provides interfaces for controlling of device modes, such as offline mode (disabling of Bluetooth and WLAN), and various system level user interfaces, such as device lock, touch screen and keypad lock, LEDs, etc.
- **Battery Management:** Responsible for battery voltage monitoring and recognition, battery charging, and charger recognition.

Power Management

The Power Management (PM) framework revolves around the concepts of [dynamic tick](#), [OS idle](#), clock framework and [DVFS](#). The PM framework can be divided into two independent mechanisms: OS idle and DVFS.

- OS idle is based on the operating system [scheduler](#). Whenever the scheduler has no tasks to perform, it calls the idle function. The idle function can then choose to shutdown all or parts of the hardware, and thus save power. The level of power savings depends on the clock and voltage resources in use.
- DVFS allows to scale down the SoC's frequency and voltage at runtime to reduce leakage currents, and hence save power. The decision to scale the frequency (and consequently voltage) of the ARM and DSP processors is based on the load on them.

OS-idle and DVFS are triggered independently of each other. DVFS is triggered based on any increase or decrease of required processing power, while OS-idle is triggered by the Linux scheduler.

Currently the [suspend](#) and resume functionality provided by Linux Driver Model is not used. Also, the kernel level power management does not get any guidance from the user space.

Clipboard

In maemo, there are a number of clipboard enhancements to the X clipboard and Gtk+, in order to enable

- Supporting retaining the clipboard data, when applications owning the clipboard exit.
- Copying and pasting rich text data between Gtk+ text views in different applications.
- Providing a generally more pleasant user experience; making it easy for application developers to gray out "Paste" menu items when the clipboard data format is not supported by the application.

Compression

This subsystem provides various libraries and utilities for general purpose data compression and uncompression. The supported compression algorithms include the Lempel-Ziv (gzip) used in zip and PKZIP as well. Supported compression formats include:

- [zlib](#)
- [deflate](#)
- [gzip](#)

ConIC

ConIC provides an interface for manipulating and using Internet access points (IAPs) and IAP connections.

Certificate Manager

The maemo platform offers an API to deal with certificate manager storage and handling. This enables every piece of software to have access to all certificates so that, for example, installing a new CA certificate takes immediate effect in all the relevant programs (such as Web browser, e-mail, VPN and wireless connection). This saves effort and disk space.

Core Daemons

Daemons are server processes to perform specific tasks. Most of the daemons, like dbus-daemon, are described separately so this subsystem presents the miscellaneous daemons.

Packages include

- syslogd ([system message log](#))

Base Files

This subsystem delivers the basic filesystem hierarchy of a Debian system as well as the master copies of user database files (/etc/passwd and /etc/group) containing the user and group IDs.

Packages include

- base-files
- base-passwd

5.3.2 Kernel

Maemo uses a Linux 2.6 operating system kernel. Linux is an open source operating system, developed by thousands of volunteers and companies that share their work under GNU GPL license. Architecturally Linux has a monolithic kernel. All kernel code is run under supervisor mode. The kernel can be extended at runtime by dynamically loadable kernel modules. Various APIs exist for device driver, file system and networking protocol modules. Developers can add new kernel modules.

The maemo kernel is based on the ARM kernel branch and can be modified, recompiled and flashed by a developer. Chapter *Kernel Guide*[11](#) gives the details for these procedures. Some of the modules, such as WLAN, come as binary only, which means that the module APIs should remain unchanged, if the kernel is changed by a developer.

5.3.3 Flash Partitioning

There are four separate flash partitions on a maemo-based device. The partitions are

- Bootloader partition
- Kernel partition
- Init file system (on a small [JFFS2](#) partition)
- Root file system (on a JFFS2 partition)

The root file system thus contains all the components of the component decomposition table, other than the lowest layer.

5.3.4 Application Framework

The purpose of an application framework is to help application development by providing a standard structure for an application. Applications that have a graphical user interface tend to have a similar structure, e.g. the event-driven runtime model. The events are triggered by the user, for example, by touching a button on the touchscreen. Events can also be triggered by the application engine itself. An example is when new data is received from the network. The application framework of maemo is called *Hildon*. It is partially based on the same technologies that the *GNOME* framework is built on, most notably the *GTK+* components.

Hildon has several additions and enhancements to GNOME/GTK+, including Hildon widget set, *Sapwood* theme engine and image server, task navigator, Hildon control panel and status bar. Some of the changes to standard GNOME, like Sapwood, are made to reduce memory requirements and to improve speed on a small hand-held device. In addition, Hildon framework has many features to support mobility, such as automatic state saving, touchscreen input methods, and window management on a physically small device.

The programming APIs are familiar to GNOME and GTK+ developers. The framework has *GLib* and *GObject* object management system underneath. The GTK+ widget set is provided with Hildon extensions. The interprocess communication is performed using *D-BUS* messages. The user files are accessed through *GNOME-VFS*, and multimedia applications can use *GStreamer* to get accelerated support for various codecs. User configurations are stored via GConf and an XML parser API is available.

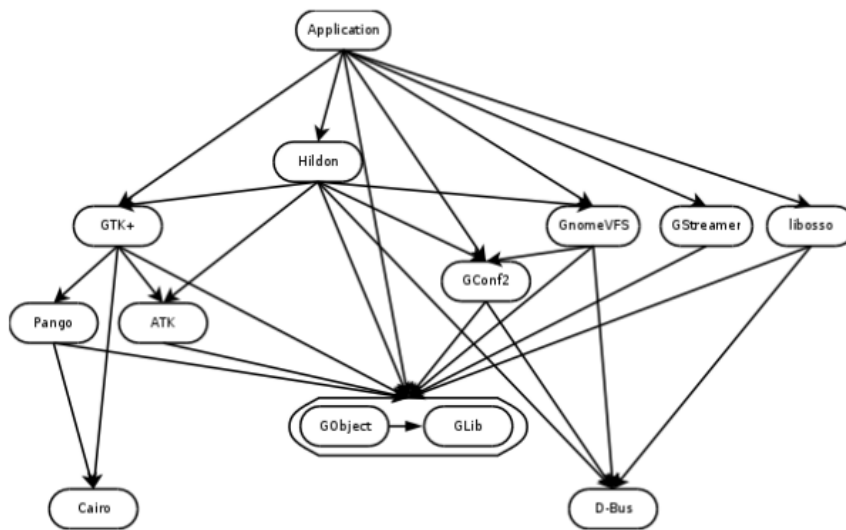


Figure 5.1: Major Application Framework Components

The figure 5.1 illustrates the most crucial components and their dependencies that a maemo application developer must deal with. These components are explained in more detail in the next sections.

5.3.5 Base Distribution

Maemo is based to a large extent on the same open-source components as found in the Debian[9] Linux distribution. Maemo builds on GNU/Linux for operating system core and GNOME/GTK+ for user interface architecture.

Maemo uses the same component packaging system as Debian - *dpkg*-tool's binary packages. New packages can be installed, old ones can be removed and the whole system can be upgraded by the package management framework. The file system structure also comes from Debian.

In order to run the software on an Internet Tablet, various optimizations and enhancements have been made. These include power management related issues, touchscreen input, performance and size optimizations. In order to reduce space, maemo uses shell and command line tools from *Busybox*[5].

Also notably the maemo platform adheres closely to the GNOME Mobile Platform[30]. At the time of writing, maemo uses all the same components as this platform, except for service discovery.

Essential Packages and Delta to Standard Debian Systems

The core distribution is Debian-based, having tools like *dpkg* and *apt* for installation package management. Also *sysvinit*, *base-files* and *base-passwd* are included from the list of Debian Etch essential packages. A couple of differences to standard Debian exist, however. One is the replacement of *Bash* and *coreutils* with *Busybox*, an optimized command line tool set and shell for embedded devices. There is no *bsdutils* package as such, but *Busybox* provides *logger* and *renice* utilities (*script*, *wall* and *scriptreplay* are missing). *Diff* has

also been left out, as well as e2fsprogs and ncurses packages. The size reduction just from the utilities and libraries is about 2.5MB and an additional 1.5MB from replacing Bash.

Package	Maemo	Debian
base-files	included	included
base-passwd	included	included
dpkg	included	included
sysvinit	included	included
awk	in Busybox	included
bash	in Busybox	included
coreutils	in Busybox	included
debianutils	in Busybox	included
findutils	in Busybox	included
grep	in Busybox	included
gzip	in Busybox	included
hostname	in Busybox	included
ifupdown	in Busybox	included
login	in Busybox	included
module-init-tools	in Busybox	included
mount	in Busybox	included
net-tools	in Busybox	included
procps	in Busybox	included
sed	in Busybox	included
tar	in Busybox	included
util-linux	in Busybox	included
vi	in Busybox	included
bsdutils	logger and renice in Busybox, script, wall, scriptreplay missing	included
diff	–	included
e2fsprogs	–	included
mktemp	in Busybox	included
ncurses-base	–	included
ncurses-bin	–	included
perl-base	included	included
sysvinit-utils	last, mesg, pidof in Busybox, lastb, killall5, sulogin missing	included

N.B. Busybox may not support all the command line options of the standard tool, even though the tool is included. Check out [Busybox manual](#) for more details.

5.4 Run Time View

5.4.1 Overview

The user starts applications primarily from the Task Navigator, but they can be started also from the Status Bar (e.g. connection manager), from File Manager to view a file, or from other applications (e.g. for "Send as E-mail" functionality).

To conserve memory, only single instance of an application can be running at any given time. If the application is already running, it will only receive a message about the new invocation, such as "open file 'foo'" and top itself. The user can switch to another, already running application either by using the Task Navigator UI, or by closing the topmost application.

Because the device does not have enough memory to run all the applications at the same time, the system may kill an application in the background. Background killing is done only if the application has indicated itself to be killable. Applications exit when the user closes them from the application UI, or when the system requests for it.

5.4.2 Components

This section describes the components involved in the application life cycle management and switching.

Task Navigator

Task Navigator (TN) is used to:

- List applications in the Others menu, so that the user can easily launch them
- Launch applications. See the section on launching applications
- Background kill applications that have set themselves as killable when the system indicates that it is low on memory
- List all the running and background-killed applications. The latter appear to the user as if they were still running
- Switch between already running applications, or to a background-killed application. This is done either by:
 - requesting the window manager to top the application window, or
 - sending the application a message requesting it to top a particular window view, or
 - restarting the application in case it was background killed

D-BUS Session Bus

Each application in the device has a well-known name, e.g. "Browser" or "Email". The application name uniquely identifies the application. There is a D-BUS service for each application, derived from the application name.

Applications are executed (i.e. activated) by the D-BUS session-bus daemon. If not running, an application is implicitly activated, when a message with the auto-activation flag is sent to the corresponding service. D-BUS gets the binary name to execute from the corresponding D-BUS .service file. The activation message may also contain parameters for the application, e.g. the name of a file to open in the application. D-Bus activation guarantees that at most one instance of the application is running at any given time.

If the service does not register within a given timeout, D-BUS assumes that the service (application) process start-up failed, and will kill the started process.

Maemo Launcher

The maemo launcher exists to speed up the application start-up, and to enable the sharing of some of the data initialized at the application start-up. The complete maemo launcher is composed of two parts. The first part, maemo-invoker, is executed by the D-BUS daemon or a script to start the given application service. The maemo-invoker asks the second part, maemo-launcher, to start the actual application. Use of the maemo launcher requires that the application is compiled as a shared library. There is a set of helper Debian package rules that make an application to "automatically" use maemo-invoker when given suitable build options. As a result, the application binary name is linked to maemo-invoker and application (library) binary name has .launch extension. By default, the invoker will wait until the maemo-launcher tells it that the application has exited, so that it can return the correct return value for the caller.

The maemo-launcher is a server that has initialized most of the data used by the applications, such as Glib types, Gtk [theme](#) and some Gtk widget classes. When it is asked by the maemo-invoker to start an application, instead of executing the application binary, it will dl-load that as a shared library, fork and call main(). With fork, the initialized data is handled as [copy-on-write](#), i.e. shared until it is modified. If the application exits abnormally, the maemo-launcher notifies the Desktop, so that the Desktop can inform the user about it.

Because [prelinking](#) does not work with [dl-loaded](#) libraries, the maemo launcher cannot speed up the starting of applications where library linking has a larger effect on the start-up time than AF library initializations. It can still save memory, though.

Window Manager

Matchbox window manager takes care of handling the window switching and window stacking. See section [5.4.7](#) for more information.

5.4.3 Application Activation

The user starts applications from the Task Navigator. The Task Navigator starts the applications by sending a D-BUS message to the application service with the D-BUS auto-activation flag set.

Applications can also be started implicitly, when other applications send them D-BUS messages, e.g. to open a mime-type that application has registered to the MIME database.

Places where different applications get the (application) D-BUS service names:

- Task Navigator: Service name is specified in the application .desktop file along with the (localized) application name and its icon
- File Manager and Browser: service name is retrieved from the gnome-vfs mime-type-handler application registry through libosso-mime library
- Other applications use the service application API libraries. The libraries know which service the service application implements or registers to D-BUS

D-BUS daemon looks into application .service file to know how to execute the application before delivering the message. Applications launched by the D-BUS daemon will only have one instance of them running, because D-BUS does not allow the same service name to be registered by more than one process. Applications should not care whether the recipient of their message on D-BUS is running, they should just send all messages with the auto-activation flag to cause automatic activation of the recipient when it is not running.

N.B. Using the auto-activation flag is not always desirable, e.g. when asking the application to shut down; thus the application framework does not force using the flag. An activated application will automatically appear on the foreground.

The Task Navigator supports also executing the application directly, if the application D-BUS service name is missing from its .desktop file. This way, the launched application does not need to know about D-BUS or Libosso at all, so any unmodified Open Source program that has a .desktop file can be launched this way. A single instance is not guaranteed for applications started like this.

An application can use the maemo launcher to speed up its start-up. Figure 5.2 shows how this happens in practice.

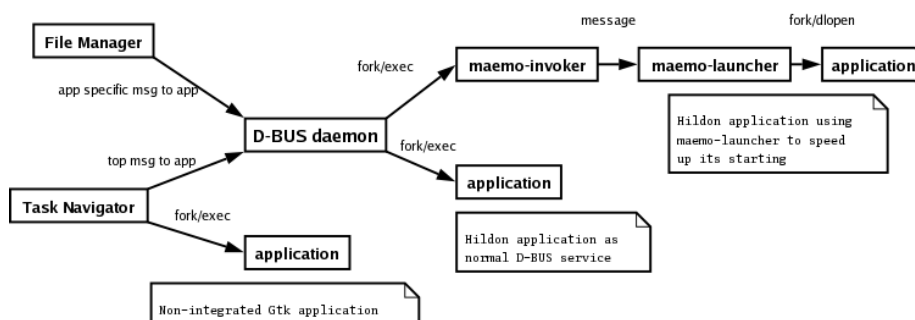


Figure 5.2: Application activation methods

Passing Environment Variables

An application launched by the D-BUS daemon will inherit [environment variables](#) that were defined at the time when the D-BUS session bus was started. D-BUS does not provide a way to change the environment variables passed to an application to be activated.

An application launched by the Task Navigator could inherit any environment variables, if such a feature is implemented to it. But when the Task Navigator uses D-BUS to launch the application, the situation is the same as directly launching the application using D-BUS.

An application launched by the maemo-launcher will inherit environment variables that were defined at the time when the maemo-launcher was started. To be able to give the applications correct (preloaded) theme, the maemo-launcher will listen on the theme changes.

Environment variables should not be used for dynamic configuration changes, since they require program restart, and D-BUS does not support that.

Locale and language changes are communicated through environment variables. As applications and their libraries also cache locale state (messages etc), changing this to use e.g. Gconf would not help. Therefore, changing the device locale or language requires restarting all applications and processes involved in the application invocation.

5.4.4 Application Termination

Applications exit when the user closes them from the application UI, or when the system requests. The system will request and force applications to exit for example when:

- The device battery charge drops low enough
- User switches off the device
- User changes the device language
- User resets the device back to factory settings
- User tries to close, or to switch to an application that does not respond, and the user accepts a system request asking whether the application should be forcefully closed

In case a use-case gets the system to run low in memory, the system can request applications to be background killed; see below. If there is not enough memory in the system to satisfy the application request, the application requesting more memory will be out-of-memory (OOM) killed by the kernel.

5.4.5 State Saving and Background Killing

Application state saving is needed so that the application UI state and user data can be restored in the case application crashes, or application is background killed. Applications are required to save their changed state when they go to background.

Background Killing

Application framework has a mechanism for shutting down GUI applications on the background to save memory, so that other applications can be run. This is called background killing.

Background killing is implemented by the Task Navigator to transparently close an application, when the user does not see it and to restart the application when the user needs it again. This is possible, because applications are required to be able to save their user interface (UI) states, and the Task Navigator knows all running UI applications. An application is required to save its UI state when it goes to background, i.e. behind some other application's window.

Unfortunately, saving the UI state may not be feasible for the application in every situation (e.g. when there is a download in progress), that is why the application notifies the Task Navigator when it has saved the state and can be killed. Task Navigator will kill all the killable applications when the system notifies that it is low on memory. When the application is started again, it is required to rebuild the UI according to the saved state, if such exists. The Task Navigator will not try to (re-)start an application, if there is not enough memory in the system for that. Problems with background killing in maemo:

- The killed application could be immediately below the top application in the window stack, and in case the user closes the top application (from the application's own menu), the user should see the application that was just closed, but will not, until it has restarted and initialized itself.
- Currently the Task Navigator checks whether a new application can be started by comparing the amount of system free memory with a build-time configured value. This might be less than a specific application would require, or in some cases more than would be required. An application-specific minimum required free memory value would be more optimal.

UI State Saving

The following picture describes a scenario, where application A saves its state, exits, restarts and reads the saved state. The libosso library only creates the state file and provides the file descriptor to the application. Libosso will make sure that the real state file will not be updated, until the state file write has completed and file closed. The application uses the standard POSIX file system API for writing to and reading from the file descriptor.

If a device is restarted, the UI states will be discarded, so that the applications will start from their default state. Each application with different version number will have its own UI state file. This means that if application is updated (its version number given in libosso initialization is changed), it will start from the default state.

Autosaving User Data

Some applications are required to save unsaved user data periodically when on the foreground (top), so that as little as possible is lost in case of a battery failure. Applications should register a libosso callback function for this operation, and

tell libosso when they are in a so-called dirty state (i.e. user data has changed), libosso will then tell applications when they should perform the autosave. (In the current implementation this is just a timer in libosso, but the reason for the existence of this API is that later on the saves can be synchronized with the device power management.)

N.B. The applications should call "forced autosave" libosso function, when they go to background (libosso does not know when this happens).

5.4.6 X Window System

This is a short and simplified introduction to the X Window System. It is covered here, because it is the underlying system by which graphics and user interaction are implemented in both the maemo platform and the Internet Tablets.

The X Window System is an architecture-independent client/server system that allows multiple programs to interact with a user via a graphical (pixel-based) screen, keyboard and a pointing device (traditionally a mouse).

The program that wants to display something to the user, and read input from the user is called the **X client**. Each X client connects to one X server, which will perform the requested graphics operations and relay keyboard and pointer events back to the client.

When speaking about clients and servers, it is easy to make the mistake of reversing the meaning of client and server. It helps to think about the roles from the standpoint of the application, not the user. When the client starts, it will connect to an X server to create a window. A window is a rectangular area, into which the client can draw. N.B. The client can ask the server to position the window at a specific screen location, but normally does not. There is a special kind of client that will handle the placement of the windows of all other clients. This client is called the **window manager**. The window manager usually draws some graphical elements around each client's window, so that the user can more easily tell the boundaries between the windows. It also handles all HID-events in the window decoration areas, implements window minimizing, closing, etc. The HID-events that occur within the client area of the window are passed to the client.

There are a lot of different window managers, but most work in a similar way. The "Desktop" (whatever the word means inside a computer) is normally implemented by yet another client. Also the taskbar that might be visible is yet another client. Even the screen saver is a separate client. In real world, there are some exceptions to the above arrangement, but having separate clients for all the elements is the most common case.

The protocol that clients use with the server is called X11. It is stream-based and bi-directional (for obvious reasons).

Clients can commonly connect to the server in two ways:

- By connecting to an IP address / TCP port on which the server is listening.
- By connecting locally using a UNIX domain socket. A UNIX domain socket is similar to TCP, but without the network in between, and the client will find the server using a name in the filesystem (N.B. This name does not correspond to a "regular" file).

How does the client know where to connect? By using an environmental variable called `DISPLAY`. There are only a handful of applications that know how to implement the X11-protocol, because it is quite complicated to encode and decode. Normally clients will use a library called `Xlib`, which was developed for this purpose. `Xlib` also contains the logic to read the `DISPLAY`-variable and will get the address to connect to from the contents of the variable. It is also possible to tell the client to use a specific display via a command line parameter (`display=`). The parameter will be processed internally by `Xlib` and override the environmental variable (if any).

The content of the `DISPLAY`-variable consists of two parts:

- **Hostname:** a text field that contains a name that will go through a `gethostbyname` library call. In practice, this is most often a DNS name or an IP address of the server, but depending on local NSS (Name Service Switch), it can be something else as well. This material will assume that DNS or IP will be used.
- **Display/Screen-pair:** number of the X server instance, and a number of a screen within that instance. Normally 0 is the only X server instance available, and it will number its screens starting from 0. The screen number can also be omitted, and 0 will be used by default.

So, for example: `DISPLAY=remote.machine.com:0.0` would mean the first screen on the first X server running on `remote.machine.com`. When starting an X server, it can normally be told which screen to create and control.

What about the "similar to TCP but not quite" UNIX domain socket? `Xlib` will connect using a system-specific file system path when the hostname portion is empty. To try it out on the Linux desktop, type `echo $DISPLAY` in a terminal emulator. The graphical terminal emulator will connect to the X server knowing the `DISPLAY` variable. It most probably is `:0.0`, unless there is a more complicated system (e.g. split dual-head).

To instruct an X client to connect to another X server, it is then necessary to modify the environmental variable: `export DISPLAY=:2.0` for example. Then start the X client, and it will at least try to connect to the X server specified. In the example above, the server is running on the same system as the client (the hostname part is empty). Since the screen part is optional, also `export DISPLAY=:2` can be used.

This all may not seem to be related to maemo, but the connection will be explained in the following part on installing the environment and testing the applications.

It should be remembered that X11 is architecture-independent. This means that applications running on Internet Tablets (ARM-binaries) can connect to an X server running on a x86/PC Linux (or even to an X server running on Apple OS X, Windows or other operating systems).

As a side note, the Internet Tablet has an X server as well. It runs as `:0.0`. It is a special version of an X server that requires less memory, and has been configured to support most of the extensions used on the Linux desktop. The version on the Internet Tablet is based on the Kdrive version of the X.Org X server. However, a Linux desktop is running a regular X.Org server.

N.B. By default, most modern Linux distributions ship with the X server not listening for network connections. They will only accept local connections

through the UNIX domain socket (`/tmp/.X11-unix/X0`, where 0 is server screen number).

For more information on X, please see the [X.Org-project](#) pages and [X Window System WWW links](#). Also, ssh can be used to tunnel X11 connections securely over networks. Please see [X Over SSH2 Tutorial](#) for examples.

5.4.7 Window Management

There are several components participating in window management. The X server performs all the graphics and window handling operations. The Matchbox window manager implements the window management policy. It takes care of:

- Themed drawing of titlebars and dialog borders.
- That dialogs being transient to an application window (or another dialog) are moved in the window stack together. For example when an application is topped, its dialogs are also topped.
- When an application switches to fullscreen mode, titlebar and panels (such as Task Navigator and Status Bar) are hidden.
- System dialogs and windows always keep above any application windows.
- When input method opens, application window is resized and dialog is moved so that it does not overlap with the input method. If dialogs would go out of screen, they are resized so that they are fully visible.
- When a window is opened, giving focus to new window and when a window is closed, giving focus to next window.

The third component of window management is the Task Navigator. It keeps a list of all windows and views that applications have open, to enable the user to switch to them. If an application is closed through state saving and background killing, the Task Navigator will still show it in this list, so that it seems to the user as the application was still running.

Application Views

In maemo, applications implement views using HildonWindow widgets which each have their own X window, all of these belonging to the same HildonProgram window group. In the maemo UI style, all dialogs are modal. Earlier the dialog blocking an application could not have been raised above the window(s) that it was blocking. The HildonWindow supports standard window properties like window specific icons etc.

Application Topping

An application can top itself, or the Task Navigator can top the application. When an application is topped by the Task Navigator and it is already running, the topping is performed by sending a standard X message.

When another application sends a message to the application requesting some functionality (e.g. the File Manager tells the image viewer to show a file), it is automatically started (but not yet topped) by the D-BUS daemon (if it is not already running, and auto-activation is desired). When the application receives the message, it can raise its window, if the operation requires user interaction.

Registering Windows to Task Navigator

For the Task Navigator to be able to top application windows, it needs information telling which windows belong to which application. This mapping is done with the application .desktop file StartupWMClass field.

When an application is launched by the D-BUS daemon, the Task Navigator compares the WM_CLASS property of the opened application window to the StartupWMClass field in the application .desktop files to see for which application it belongs to. WM_CLASS window property is automatically set to the application binary name by Gtk. Application icons in the Task Navigator application launcher and switcher menus are also taken from the .desktop file.

If the application WM_CLASS does not match StartupWMClass of any .desktop file, there will be no icon for the application in the Task Navigator application switcher, and the user cannot switch back to the application. This is a limitation of the current Task Navigator, and hopefully fixed in later products.

Closing Application Windows

Application windows are in a stack. When topping an application, it comes on top of the stack. When the window is closed, the previously shown window will be shown. The application window stack is separate from the system (dialog) window stack.

System-UI Window Layer

All System-UI windows are on the prioritized system modal "dialogs" layer, i.e. windows with the KEEP_ABOVE property. The Matchbox window manager keeps them on top of all other system and application windows and dialogs.

At any given time, there can be visible none or any number of System-UI windows.

The topmost (active) window always performs pointer and keyboard grabs. Normal application menus do also these grabs; in OSSO they are required (modified) to close and release grabs when another window comes above them.

All of the System-UI windows are taken care of by the same process, which has a certain internal stacking order for them. I.e. same window comes always in the same position, but all of them might not be visible or even exist at the same time.

There is also a 1x1 window outside the screen, always mapped to catch Gtk theme change messages.

5.4.8 Misbehaved Applications

The application framework assumes that applications behave well. However, in a few situations it checks that an application has not jammed by sending a

standard [EWMH] `_NET_WM_PING` message to it, and expecting a reply. Gtk will automatically reply to the ping in the Gtk event/main loop if an application runs it.

This pinging is performed by the Matchbox window manager when:

- The application window close button is tapped, i.e. the user tries to close the application
- The Task Navigator asks it to open an application window, i.e. the user tries to switch back to the application by selecting it from the Task Navigator application switcher. This is needed because the window close button is not visible for fullscreen applications (in fullscreen mode the user can use the Home key to switch between applications).

If the application does not answer timely, the window manager informs the Task Navigator, which will then show the user a dialog requesting whether the application should be terminated. If the user accepts this, first a `TERM` and then a `KILL` signal are sent to the application process (process ID is taken from the window `_NET_WM_PID` property), and the killing is logged. The dialog will go away automatically, if the application answers the ping while the dialog is open.

If a maemo-launched application terminates abnormally, maemo-launcher will send a message to the Desktop, which will inform user about it.

5.5 Major APIs

Major APIs are listed in the table below. APIs like the X APIs are not listed here, since they are intended to be for internal use only, and not for application programmers.

5.6 Maemo Compared to Desktop Linux Distributions

Ubuntu[95] is a popular Debian-based Linux distribution for the ordinary desktop. The Ubuntu wiki[96] lists the key differences between Ubuntu and Debian. As one of the design principles of maemo has been to be as close as a traditional desktop Linux as feasible, here the differences between Debian/Ubuntu and maemo are explained in detail.

CPU Architecture Differences

Whereas Debian supports multiple CPU architectures, Ubuntu's set is a bit more restricted. Compared to that, maemo is an embedded ARM EABI[1] distribution. Maemo is also cross-compiled, instead of natively compiled, like Debian or Ubuntu. Maemo also uses different versions of toolchains (GCC, glibc[28] etc.) than Ubuntu or Debian for ARM feature support and maturity differences between architectures.

Security Model Differences

glibc	The core C library API
Glib	Utility library, basic types, memory management, dynamic strings, linked lists, etc.
GObject	The object model API of GTK+ & GNOME.
GTK+	GTK+ graphical user interface widget set library.
gdk-pixbuf	GTK+ Bitmap image handling.
libosso	maemo base library, application initialization etc.
Hildon APIs	maemo Hildon APIs.
GStreamer	GStreamer multimedia framework.
libxml2	XML and HTML parser and tools.
libpango	Text rendering framework.
libatk	Accessibility framework.
D-BUS	Inter-process communication framework.
GConf	Configuration management framework.
libcurl	HTTP access library.
SQLite	SQL database library.
LibC++	Standard library for C++ language.
GnomeVFS	Filesystem Abstraction library.
HAL	Hardware Abstraction Layer Specification.
OpenSSL	OpenSSL Documents.
libconic	Internet Connectivity library.
libebook	Evolution Addressbook Library.

Instead of a multi-user system, such as a traditional Linux desktop, maemo is considered a single-user desktop system. The security model in maemo is focused on protecting the user from remote attacks and from themselves, not from other users. Maemo also uses *suid* root binaries and */etc/passwd*, whereas Ubuntu enforces the use of *sudo* and *shadow passwords*.

Unlike Ubuntu, maemo makes use of a root account like Debian does, but has a trivial default password. The user should really change the root password before installing e.g. *OpenSSH* to the device with root login.

Base System Differences

The greatest difference in the base system is that maemo uses a lightweight BusyBox [5] replacement for the essential GNU utilities[23] on the device e.g. *ls* and *sh*. In maemo, kernel and initfs reside in separate partitions and cannot be updated with a package manager like with a common desktop Linux. Programs in initfs use uClibc[97] instead of glibc. Ubuntu has Perl and Python languages as essential packages, Debian has only Perl and maemo has neither. Maemo has no *debconf*. Ubuntu uses Upstart for device start-up instead of SYSV init scripts used in maemo.

Chapter 6

Application Development

6.1 Introduction

The following code examples are used in this chapter:

- [example_hildonprogram.c](#)
- [example_menu.c](#)
- [example_toolbar.c](#)
- [example_findtoolbar.c](#)
- [example_file_chooser.c](#)
- [example_color_chooser.c](#)
- [example_font_selector.c](#)
- [example_file_details.c](#)
- [example_banner.c](#)
- [example_context.c](#)
- [example_hard_keys.c](#)
- [example_libosso.desktop](#)
- [example_libosso.service](#)
- [example_libosso.c](#)
- [example_message.c](#)
- [example_gconf.c](#)
- [libapplet.c](#)
- [Hildon Desktop Plug-ins sample](#)
- [example_message.c](#)

- [example_gconf.c](#)
- [MaemoPad](#)
- [Help File](#)
- [example_help_framework.c](#)

The maemo platform's basic starting point for graphical user interface programming is the *Hildon*[44], an application framework comprising of a lightweight desktop, a set of widgets optimized for handheld devices, a set of theming tools and other complementary libraries and applications.

Hildon is based on GNOME[29] technologies to a great extent. Compared to a GNOME desktop from a user interface point of view, Hildon is designed to provide a new desktop for mobile embedded devices. Therefore it for example uses a lighter window manager called Matchbox[76].

6.2 Typical Maemo GUI Application

The following shows the components making up a typical GUI application developed for maemo (starting from the bottom):

- **C library:** Implements wrappers around the system calls to the kernel and a lot of other useful things. However, the libraries presented below also provide their own APIs to similar functions, so it should always be checked whether they can be used directly, and avoid doing POSIX-level and system-level calls when possible. This will make the application easier to debug, and in some cases easier understand. This library is used (indirectly at least) by every application running on any Linux-based system.
- **Xlib:** A library that allows an application to send graphics-related commands to the X server, and receive HID events from the server. Normally an application would not use Xlib API directly, but would instead use some easier toolkit which in turn will use Xlib.
- **GLib:** A utility library that provides portable types, an object-oriented framework (GObject/GType), a general event mechanism (sometimes referred to as GSignal), common abstract data structure types like hash tables, linked lists, etc.
- **GDK:** A library that abstracts the Xlib and provides a lot of convenience code to implement most common graphical operations. Used by an application wanting to implement drawing directly, for example in order to implement custom widgets. In theory, GDK is meant to be independent of graphics systems, which it mostly is. Complete abstraction, however, is not yet complete, but for now, the original Xlib target will be enough. GDK uses GLib internally.
- **Pango:** A portable library designed to implement correct and flexible text layout for various cultures around the world. This is necessary to support the different ways that people read and write text, since it is not always

left-to-right, top-to-bottom. Uses GLib and GDK. Used by GTK+ for all displayed text.

- **ATK:** The Accessibility ToolKit. Provides generic methods by which an application can support people with special needs with respect to using computers.
- **GTK+:** A library that provides a portable set of graphical elements, graphical area layout capabilities and interaction functions for applications. Graphical elements in GTK+ are called widgets. GTK+ also supports the notion of themes, which are user-switchable sets of graphics and behavior models. These are called skins in some other systems. Uses GLib, GDK, Pango and ATK.
- **Hildon:** A library containing widgets and themes designed specifically for maemo. This is necessary since the screen has very high PPI (compared to PCs), and applications are sometimes controlled via a stylus. Uses all of the libraries above.
- Other support libraries of interest:
 - **GConf:** A library from the GNOME project that allows applications to store and retrieve their settings in a consistent manner (in a way, similar to the registry in Windows). Uses GLib.
 - **GnomeVFS:** A library that provides a coherent set of file access functions, and implements those functions using plug-in libraries for different kinds of files and devices. Allows the application to ignore the semantics of implementations between different kinds of devices and services. By using GnomeVFS, an application does not need to care whether it will read a file coming from a web server (URLs are supported), or from within an compressed file archive (.zip, .rpm, .tar.gz, etc.) or a memory card. Modeled to follow POSIX-style file and directory operations.
 - **GDK-Pixbuf:** A library that implements various graphical bitmap formats and also alpha-channeled blending operations using 32-bit pixels (RGBA). The Application Framework uses pixbufs to implement the shadows and background picture scaling when necessary. Uses GLib and GDK.
 - **LibOSSO:** A library specific to the maemo platform, allowing an application to connect to D-Bus in a simple and consistent manner. Also provides an application state serialization mechanism. This mechanism can be used by an application to store its state, so that it can continue from the exact point in time, when the user switched to another application. Useful to conserve battery life on portable devices.

There are some caveats related to API changes between major GTK+ versions, which will be mentioned in the text, so existing codes should not be copy-pasted blindly.

6.3 Hildon Desktop

The end user's desktop experience on Nokia Internet Tablets is provided by the *Hildon Desktop* that is based on **Hildon**. The latter provides several libraries to interface with the desktop environment UI elements and services.

Hildon framework's main UI elements are separated to four categories and all of them can be extended with plug-ins:

- **Hildon Home** is the root desktop, which can be customized by Hildon home applets.
- **Hildon Status bar** provides area for information and quick-access items used mainly to communicate device status changes.
- **Hildon Task Navigator** plug-ins implement the top-level desktop menus.
- **Hildon Control Panel** is a general interface for application configuration and is extended by control panel plug-ins.

GUI applications in maemo usually have one or more *HildonWindows*, which are the top-level application windows. They can include a standardized menubar and a toolbar.

The Hildon framework also includes other auxiliary widgets and services specialized for the Hildon environment. Examples of these are the *Hildon-FM* library for file system dialogs and widgets, *HildonBanner* for displaying notifications to the user and *HildonWizardDialog* for creating wizard user interfaces. For information about Hildon's widgets, see Maemo API References[57].

Another important element of the Hildon framework is the *Hildon Input Method* API, which is an interface for creating new user input systems in addition to the included virtual keyboard and handwriting recognition.

6.4 Writing Maemo GUI Applications

6.4.1 Overview of Maemo GUI Applications

Hildon is a graphical user interface designed for small mobile devices. This section aims to assist developers in getting started with the development of Hildon-compatible applications.

Most GTK+ widgets and methods work unaltered in the Hildon environment, but to make applications fit in the maemo GUI environment, some source code changes for plain GTK+ applications are required.

The following sections introduce the most important widgets in Hildon; all GTK+ widgets can be used in addition of these. For more information, see the GTK+ Reference Manual at [41].

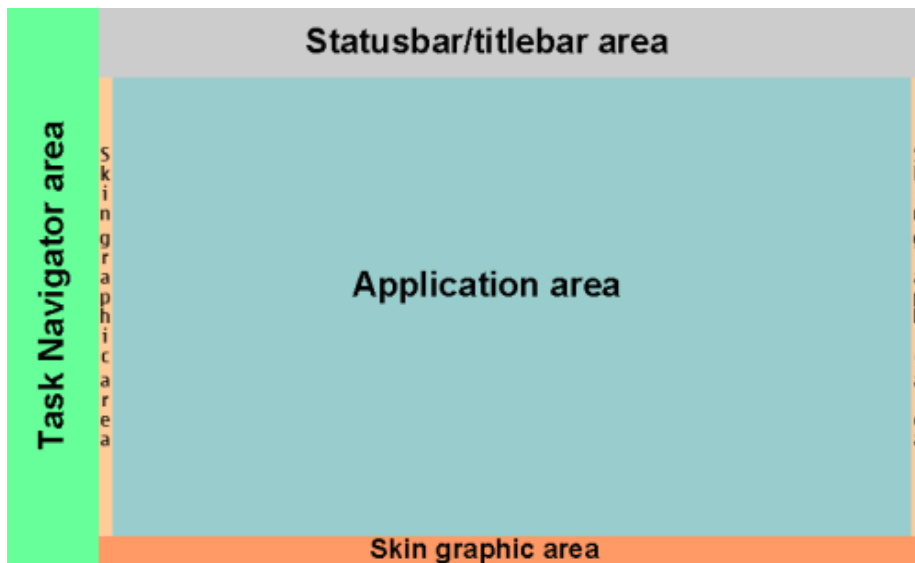
6.4.2 Basic Hildon Layouts

This section shows basic layout modes of the Hildon user interface. Applications can use all of these and switch dynamically between different views. In all the views, application area can, of course, be shared in any possible way using GTK+ containers (e.g. GtkHBox, GtkVBox and GtkTable).

Normal View

The figures below describe the basic layout of the Hildon user interface in the normal view. This view consists of the Task Navigator area, statusbar/titlebar area, application area and three areas of inactive skin graphics.

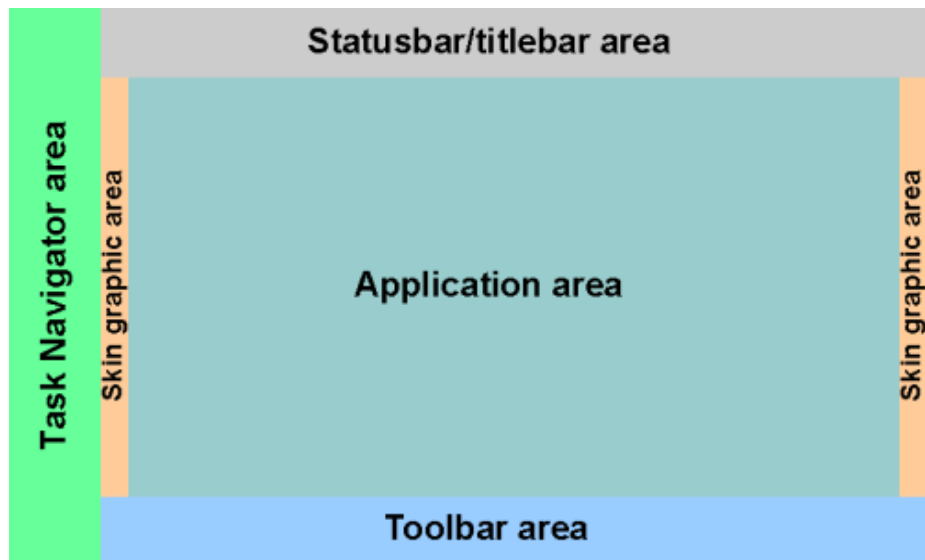
- Task Navigator area on the left is 80 pixels wide
- Statusbar/titlebar area is 720x60 pixels
- Application area in the middle is 696x396 pixels



Normal View with Toolbar

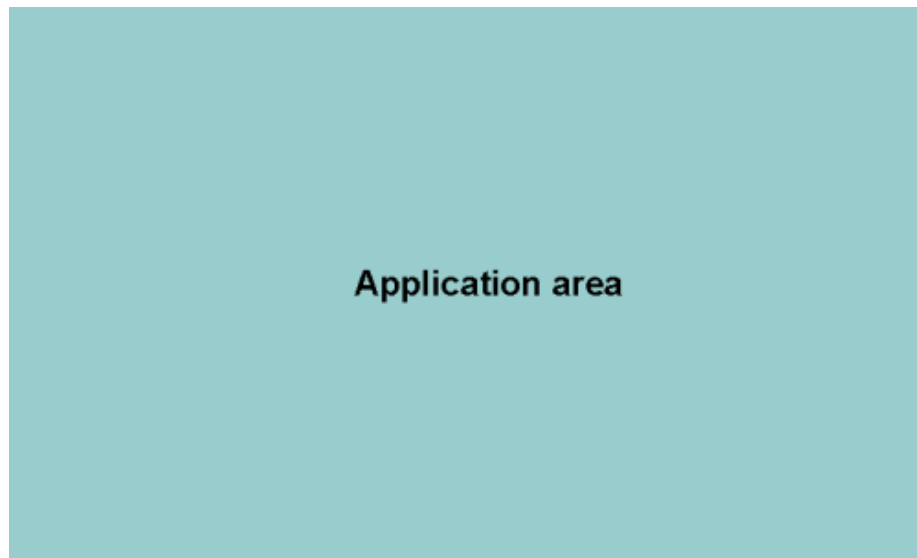
This basic layout contains all types of functional areas. The layout is basically the same as the Normal View, but there is a toolbar area at the bottom, replacing the inactive skin graphic area. The toolbar area width is 720 pixels and the height varies depending on the toolbar type:

- 360 pixels with a single toolbar
- 310 pixels with both application and Find toolbar



Full Screen View

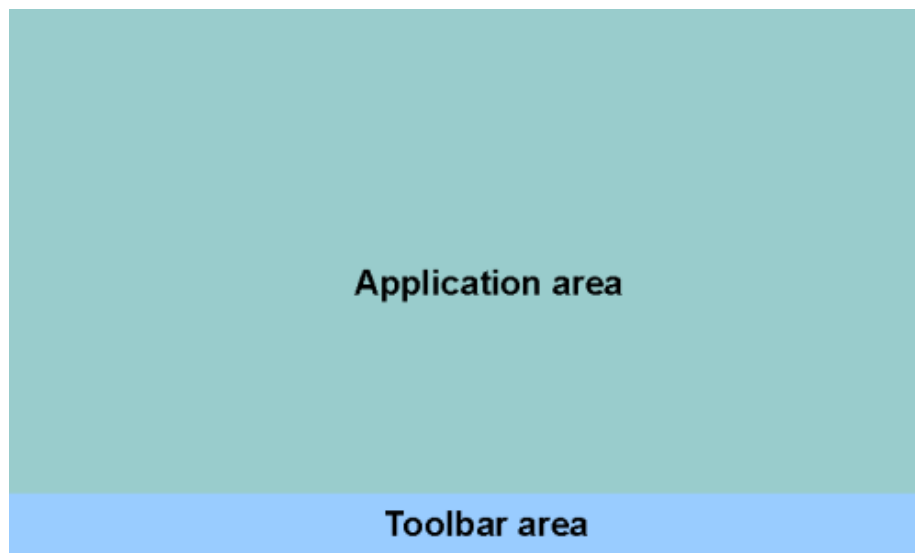
In this view, the whole screen (800x480 pixels) is reserved for the application area. The full screen mode can be activated and deactivated by a hardware button, or in the application code. All applications should provide full screen mode to enable maximized screen usage when needed.



Full Screen View with Toolbar

This is a variation of the full screen view. There is a toolbar area at the bottom of the screen, reducing the space available for the application area. The toolbar can be visible in both normal and full screen modes, so it should be made scalable.

- With a single toolbar, the application area height is 422 pixels.
- Multiple toolbars can be used (e.g. the Find toolbar) simultaneously. All of them appear at the bottom of screen, on top of each other, reducing application area. With two toolbars, the application area height is 370 pixels.



6.4.3 Windows

HildonProgram

```
#include <hildon/hildon-program.h>
#include <gtk/gtkmain.h>
```

A HildonProgram is the base of any Hildon application. It is inherited from GObject, and represents an application running in the Hildon framework. The HildonProgram tracks the top-most status of an application, and informs the Task Navigator when the application can be hibernated. It also provides tools to get/set menus and toolbars common for all application windows.

The HildonProgram widget is created by calling the function **`hildon_program_get_instance()`**, which does not take any parameters, and returns the newly-created HildonProgram. For this tutorial, the following API functions are of interest:

```
HildonProgram *hildon_program_get_instance (void)
```

- Creates new HildonProgram widget.

```
void hildon_program_add_window (HildonProgram *self,
                               HildonWindow *window)
```

- Adds HildonWindow to HildonProgram.

```
hildon_program_set_can_hibernate (HildonProgram *self,  
                                  gboolean killable)
```

- Informs Task Navigator that it can hibernate the HildonProgram.

```
void hildon_program_set_common_menu (HildonProgram *self,  
                                     GtkMenu *menu)
```

- Sets common menu for all application windows.

```
void hildon_program_set_common_toolbar (HildonProgram *self,  
                                        GtkToolbar *toolbar)
```

- Sets common toolbar for all application windows.

This is an example ([example_hildonprogram.c](#)) application, which creates new HildonProgram and HildonWindow. It sets a title for the application and creates a sample label.

```
int main(int argc, char *argv[])  
{  
    /* Create needed variables */  
    HildonProgram *program;  
    HildonWindow *window;  
  
    /* Initialize the GTK. */  
    gtk_init(&argc, &argv);  
  
    /* Create the Hildon program and setup the title */  
    program = HILDON_PROGRAM(hildon_program_get_instance());  
    g_set_application_name("App Title");  
  
    /* Create HildonWindow and set it to HildonProgram */  
    window = HILDON_WINDOW(hildon_window_new());  
    hildon_program_add_window(program, window);  
  
    /* Add example label to window */  
    gtk_container_add(GTK_CONTAINER(window),  
                      GTK_WIDGET(gtk_label_new("HildonProgram Example")  
                                ));  
  
    /* Begin the main application */  
    gtk_widget_show_all(GTK_WIDGET(window));  
  
    /* Connect signal to X in the upper corner */  
    g_signal_connect(G_OBJECT(window), "delete_event",  
                     G_CALLBACK(gtk_main_quit), NULL);  
  
    gtk_main();  
  
    /* Exit */  
    return 0;  
}
```

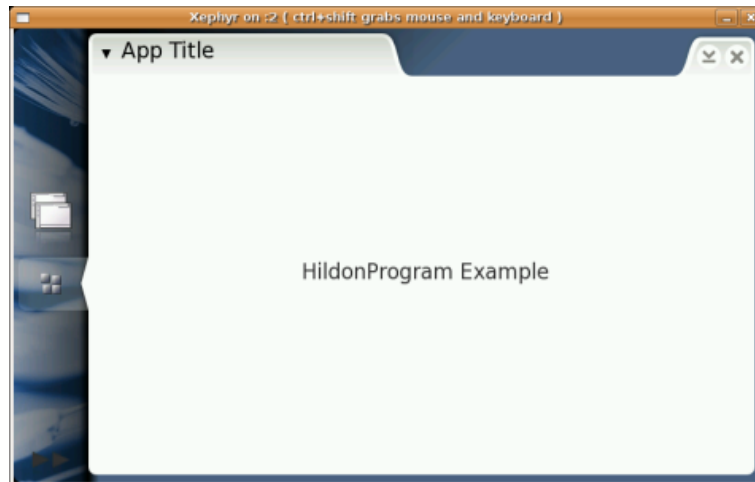
To compile the program, type

```
$ gcc -o example_hildonprogram example_hildonprogram.c `pkg-config gtk+-2.0 hildon-1  
--cflags --libs` -Wall
```

To run this sample from inside Scratchbox, start the Xephyr server and

```
$ run-standalone.sh ./example_hildonprogram
```

And you will be able to see



HildonWindow

```
#include <hildon/hildon-window.h>
```

The HildonWindow is inherited from the GtkWindow. It represents a top-level window of an application running in the Hildon framework. It provides facilities to manage window menus and toolbars. GtkWindow API applies to it as well (`gtk_window_fullscreen`, `gtk_window_set_urgency_hint`, ...).

Each HildonWindow can have its own menu and toolbars. Applications can also have common menu and toolbar, shared between all application windows. To get more information, check HildonProgram, Menus and Toolbars sections.

The following HildonWindow functions are the most important ones:

```
GtkWidget *hildon_window_new (void)
```

- Creates a new HildonWindow widget.

```
void hildon_window_set_menu (HildonWindow *self,  
                             GtkMenu *menu)
```

- Sets a menu for the HildonWindow.

```
void hildon_window_add_toolbar (HildonWindow *self,  
                                GtkToolbar *toolbar)
```

- Adds a toolbar for the HildonWindow.

See the previous section for a sample code.

6.4.4 Menus

```
#include <hildon/hildon-program.h>
#include <gtk/gtk.h>
```

The menus in maemo use GTK+ menu functions, with the exception that the menu is attached to the HildonProgram/Window, and appears in the title bar. A GtkMenu can be created and attached to the HildonProgram to be used as a common menu for all HildonWindows that do not have their own menu. Another way is to use a GtkMenu in a HildonWindow. This way, every application window can have different menu.

GTK+ menus can be made both manually and using GtkUIManager. The GtkUIManager is a new action-based method to create menus and toolbars using an XML description. This tutorial introduces only manual creation. More about GtkUIManager can be read at [\[42\]](#).

The following example ([example_menu.c](#)) creates a menu with a submenu. This submenu contains radio menu items and a check menu item, all of which can be toggled. The last item in the menu (Close) is attached to a callback function, so that the application can be closed also from the menu. The starting point here is the function creating this menu. In this example, the menu is attached to the one and only HildonWindow.

```
/* Create the menu items needed for the main view */
static void create_menu(HildonWindow * main_window)
{
    /* Create needed variables */
    GtkWidget *main_menu;
    GtkWidget *menu_others;
    GtkWidget *item_others;
    GtkWidget *item_radio1;
    GtkWidget *item_radio2;
    GtkWidget *item_check;
    GtkWidget *item_close;
    GtkWidget *item_separator;

    /* Create new main menu */
    main_menu = gtk_menu_new();

    /* Create new submenu for "Others" */
    menu_others = gtk_menu_new();

    /* Create menu items */
    item_others = gtk_menu_item_new_with_label("Others");
    item_radio1 = gtk_radio_menu_item_new_with_label(NULL, "Radio1");
    item_radio2 =
        gtk_radio_menu_item_new_with_label_from_widget(
            GTK_RADIO_MENU_ITEM
                (item_radio1),
                "Radio2");
    item_check = gtk_check_menu_item_new_with_label("Check");
    item_close = gtk_menu_item_new_with_label("Close");
    item_separator = gtk_separator_menu_item_new();

    /* Add menu items to right menus */
    gtk_menu_append(main_menu, item_others);
    gtk_menu_append(menu_others, item_radio1);
    gtk_menu_append(menu_others, item_radio2);
    gtk_menu_append(menu_others, item_separator);
}
```

```

gtk_menu_append(menu_others, item_check);
gtk_menu_append(main_menu, item_close);

/* Add others submenu to the "Others" item */
hildon_window_set_menu(HILDON_WINDOW(main_window), GTK_MENU(
    main_menu));
gtk_menu_item_set_submenu(GTK_MENU_ITEM(item_others), menu_others);

/* Attach the callback functions to the activate signal */
g_signal_connect(G_OBJECT(item_close), "activate",
    GTK_SIGNAL_FUNC(item_close_cb), NULL);

/* Make all menu widgets visible */
gtk_widget_show_all(GTK_WIDGET(main_menu));
}

```

The menu entry "Close" is attached with the function "g_signal_connect" to the callback function "item_close_cb". This function, which is presented next, ends the application by executing "gtk_main_quit()".

```

/* Callback for "Close" menu entry */
void item_close_cb()
{
    g_print("Closing application...\n");
    gtk_main_quit();
}

```

The main function in this application is similar to others presented earlier. The only new thing here is executing the function "create_menu()" with HildonWindow as its parameter. This function creates the menu and attaches it to the window.

```

/* Main application */
int main(int argc, char *argv[])
{
    /* Create needed variables */
    HildonProgram *program;
    HildonWindow *window;

    /* Initialize the GTK. */
    gtk_init(&argc, &argv);

    /* Create the hildon program and setup the title */
    program = HILDON_PROGRAM(hildon_program_get_instance());
    g_set_application_name("Menu Example");

    /* Create HildonWindow and set it to HildonProgram */
    window = HILDON_WINDOW(hildon_window_new());
    hildon_program_add_window(program, window);

    /* Add example label to HildonWindow */
    gtk_container_add(GTK_CONTAINER(window),
        gtk_label_new("Menu Example"));

    /* Create menu for HildonWindow */
    create_menu(window);

    /* Connect signal to X in the upper corner */
    g_signal_connect(G_OBJECT(window), "delete_event",
        G_CALLBACK(item_close_cb), NULL);

    /* Begin the main application */
}

```

```

gtk_widget_show_all(GTK_WIDGET(window));
gtk_main();

/* Exit */
return 0;
}

```

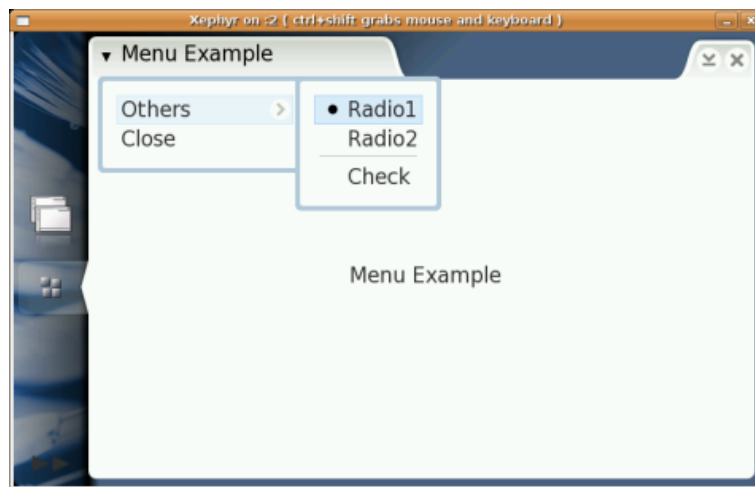
To compile and run, start Xephyr server and

```

$ gcc -o example_menu example_menu.c `pkg-config gtk+-2.0 hildon-1 --cflags --libs` -Wall
$ run-standalone.sh ./example_menu

```

Screenshot of the application is presented below, with the submenu opened.



As the maemo menus are GTK+ menus, they are not explained here in more detail. For full API documentation regarding GTK+ menu widgets, see [40].

6.4.5 Toolbars

```

#include <hildon/hildon-program.h>
#include <gtk/gtk.h>

```

To create an application that has a toolbar, create a normal HildonWindow and then a normal GtkToolbar. The toolbar is used as any GtkToolbar is used, and it can contain normal GtkToolItems, like:

- **GtkToolButton** - A normal toolbar button that can be clicked.
- **GtkToggleToolButton** - A toolbar button that can be toggled between two different states. An example of this kind of usage is the 'Bold' button in text editors: it can be either enabled or disabled.
- **GtkRadioToolButton** - A toolbar button that can be toggled between two different states, with the addition that only one of the buttons in a group can be selected at a time. Examples of this kind of usage are tool buttons (line, box, fill) in drawing applications: only one tool can be selected at any given time.

- `GtkSeparatorToolItem` - A toolbar item that is used as a separator between real items. It can be a visible separation line, or just an empty space.
- `GtkToolItem` - The base class of widgets that can be added to `GtkToolbar`. By using this as a parent widget, all kinds of widgets can be inserted inside a `GtkToolbar`. An example of this kind of usage can be seen in the following sample code, where the `GtkComboBox` is attached to a toolbar.

The main difference compared to a normal GTK+ usage is that toolbars can be common to all `HildonWindows` in an application, or they can be used only in a particular `HildonWindow`.

The function below creates a toolbar and adds it to the `HildonWindow`. In this example ([example_toolbar.c](#)), the defined function is called in the main function in the same way as in the previous menu example. The callback function for the 'Close' button (`tb_close_cb`) is similar to the earlier example, also.

```
/* Create the toolbar needed for the main view */
static void create_toolbar(HildonWindow * main_window)
{
    /* Create needed variables */
    GtkWidget *main_toolbar;
    GtkToolItem *tb_new;
    GtkToolItem *tb_open;
    GtkToolItem *tb_save;
    GtkToolItem *tb_close;
    GtkToolItem *tb_separator;
    GtkToolItem *tb_comboitem;
    GtkComboBox *tb_combo;

    /* Create toolbar */
    main_toolbar = gtk_toolbar_new();

    /* Create toolbar button items */
    tb_new = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);
    tb_open = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
    tb_save = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);
    tb_close = gtk_tool_button_new_from_stock(GTK_STOCK_CLOSE);

    /* Create toolbar combobox item */
    tb_comboitem = gtk_tool_item_new();
    tb_combo = GTK_COMBO_BOX(gtk_combo_box_new_text());
    gtk_combo_box_append_text(tb_combo, "Entry 1");
    gtk_combo_box_append_text(tb_combo, "Entry 2");
    gtk_combo_box_append_text(tb_combo, "Entry 3");
    /* Select second item as default */
    gtk_combo_box_set_active(GTK_COMBO_BOX(tb_combo), 1);
    /* Make combobox to use all available toolbar space */
    gtk_tool_item_set_expand(tb_comboitem, TRUE);
    /* Add combobox inside toolitem */
    gtk_container_add(GTK_CONTAINER(tb_comboitem), GTK_WIDGET(tb_combo)
    );

    /* Create separator */
    tb_separator = gtk_separator_tool_item_new();

    /* Add all items to toolbar */
    gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_new, -1);
    gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_separator, -1);
    gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_open, -1);
```

```

gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_save, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_comboitem, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_close, -1);

/* Add signal listener to "Close" button */
g_signal_connect(G_OBJECT(tb_close), "clicked",
                 G_CALLBACK(tb_close_cb), NULL);

/* Add toolbar HildonWindow */
hildon_window_add_toolbar(main_window, GTK_TOOLBAR(main_toolbar));
}

```

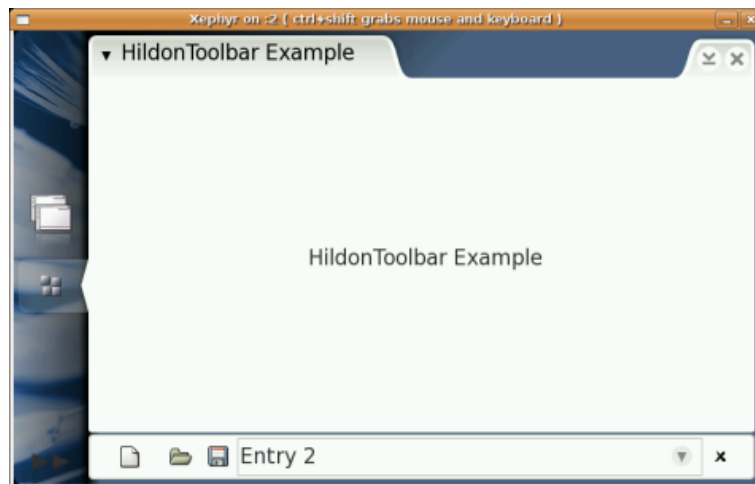
The screenshot presents the created toolbar with several buttons and combobox.

To compile and run, start Xephyr server and

```

$ gcc -o example_toolbar example_toolbar.c `pkg-config gtk+-2.0 hildon-1 --cflags --libs`
-Wall
$ run-standalone.sh ./example_toolbar

```



6.4.6 HildonFindToolbar

```

#include <hildon/hildon-program.h>
#include <hildon/hildon-find-toolbar.h>
#include <gtk/gtk.h>

```

A HildonFindToolbar is a special toolbar for a find feature. Maemo applications may have several toolbars, which are then attached on top of each other. The Find toolbar is generally placed on the top of the main toolbar, as in this example.

Now a bit more complete application ([example_findtoolbar.c](#)), for which the Find toolbar needs to be opened and hidden. A structure is created to contain the main UI widget pointers. This is the way a real maemo application should be built to make accessing generated widgets easier. The code is reviewed here piece by piece, at first the data structure, holding important data needed to be accessed by other functions.

```

/* Application UI data struct */

```

```
typedef struct _AppData AppData;
struct _AppData {

    /* View items */
    HildonProgram *program;
    HildonWindow *window;

    /* Toolbar */
    GtkWidget *main_toolbar;

    /* Find toolbar */
    HildonFindToolbar *find_toolbar;

    /* Is Find toolbar visible or not */
    gboolean find_visible;

    /* Result label */
    GtkWidget *label;
};
```

The next part presents all the callback functions required: "Close" and "Find" for the main toolbar, and "Close" and "Search" for the Find toolbar. The actual search is not implemented, but it should be included in the *find_tb_search()* function.

```
/* Callback for "Close" toolbar button */
void tb_close_cb(GtkToolButton * widget, AppData * view)
{
    g_print("Closing application...\n");
    gtk_main_quit();
}

/* Callback for "Find" toolbar button */
void tb_find_cb(GtkToolButton * widget, AppData * view)
{
    /* Show or hide find toolbar */
    if (view->find_visible) {
        gtk_widget_hide_all(GTK_WIDGET(view->find_toolbar));
        view->find_visible = FALSE;
    } else {
        gtk_widget_show_all(GTK_WIDGET(view->find_toolbar));
        view->find_visible = TRUE;
    }
}

/* Callback for "Close" find toolbar button */
void find_tb_close(GtkWidget * widget, AppData * view)
{
    gtk_widget_hide_all(GTK_WIDGET(view->find_toolbar));
    view->find_visible = FALSE;
}

/* Callback for "Search" find toolbar button */
void find_tb_search(GtkWidget * widget, AppData * view)
{
    gchar *find_text = NULL;
    gchar *label_text = NULL;

    g_object_get(G_OBJECT(widget), "prefix", &find_text, NULL);
    /* Implement the searching here... */
    label_text = g_strdup_printf ("HildonFindToolbar Example.\n\nSearch
    for:\n\n%s.", find_text);
}
```

```

gtk_label_set_text (GTK_LABEL (view->label), label_text);
g_free(label_text);

g_print("Search for: %s\n", find_text);
}

```

Next, the HildonFindToolbar is created in a separate function, and connected to the callback listeners presented before. **N.B.** There is no *gtk_widget_show()* function for the toolbar here, as it is designed to be kept hidden when application starts.

```

/* Create the find toolbar */
void create_find_toolbar(AppData * view)
{
    HildonFindToolbar *find_toolbar;
    find_toolbar = HILDON_FIND_TOOLBAR
        (hildon_find_toolbar_new("Find String: "));

    /* Add signal listeners to "Search" and "Close" buttons */
    g_signal_connect(G_OBJECT(find_toolbar), "search",
        G_CALLBACK(find_tb_search), view);
    g_signal_connect(G_OBJECT(find_toolbar), "close",
        G_CALLBACK(find_tb_close), view);
    hildon_window_add_toolbar(view->window, GTK_TOOLBAR(find_toolbar));

    /* Set variables to AppData */
    view->find_toolbar = find_toolbar;
    view->find_visible = FALSE;
}

```

The function *create_toolbar()* is very similar to the earlier example, only a callback is added for the Find toolbar button. Also, instead of finding, the find button in the main toolbar just shows or hides the *find_toolbar*

```

/* Create the toolbar for the main view */
static void create_toolbar(AppData * appdata)
{
    /* Create needed variables */
    GtkWidget *main_toolbar;
    GtkToolItem *tb_new;
    GtkToolItem *tb_open;
    GtkToolItem *tb_save;
    GtkToolItem *tb_find;
    GtkToolItem *tb_close;
    GtkToolItem *tb_separator;
    GtkToolItem *tb_comboitem;
    GtkComboBox *tb_combo;

    /* Create toolbar */
    main_toolbar = gtk_toolbar_new();

    /* Create toolbar button items */
    tb_new = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);
    tb_open = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
    tb_save = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);
    tb_find = gtk_tool_button_new_from_stock(GTK_STOCK_FIND);
    tb_close = gtk_tool_button_new_from_stock(GTK_STOCK_CLOSE);

    /* Create toolbar combobox item */
    tb_comboitem = gtk_tool_item_new();
    tb_combo = GTK_COMBO_BOX(gtk_combo_box_new_text());
    gtk_combo_box_append_text(tb_combo, "Entry 1");
}

```

```

gtk_combo_box_append_text(tb_combo, "Entry 2");
gtk_combo_box_append_text(tb_combo, "Entry 3");
/* Select second item as default */
gtk_combo_box_set_active(GTK_COMBO_BOX(tb_combo), 1);
/* Make combobox to use all available toolbar space */
gtk_tool_item_set_expand(tb_comboitem, TRUE);
/* Add combobox inside toolitem */
gtk_container_add(GTK_CONTAINER(tb_comboitem), GTK_WIDGET(tb_combo)
);

/* Create separator */
tb_separator = gtk_separator_tool_item_new();

/* Add all items to toolbar */
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_new, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_separator, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_open, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_save, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_comboitem, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_find, -1);
gtk_toolbar_insert(GTK_TOOLBAR(main_toolbar), tb_close, -1);

/* Add signal listener to "Close" button */
g_signal_connect(G_OBJECT(tb_close), "clicked",
                 G_CALLBACK(tb_close_cb), NULL);

/* Add signal listener to "Find" button */
g_signal_connect(G_OBJECT(tb_find), "clicked",
                 G_CALLBACK(tb_find_cb), appdata);

/* Add toolbar HildonWindow */
hildon_window_add_toolbar(appdata->window, GTK_TOOLBAR(main_toolbar)
);

gtk_widget_show_all(main_toolbar);
appdata->main_toolbar = main_toolbar;
}

```

Then the main application loop creates a new *AppData* structure and stores the newly-created *HildonProgram* and *HildonWindow* into it. Both toolbars are created, but only the main toolbar is set visible.

```

/* Main application */
int main(int argc, char *argv[])
{
    /* Create needed variables */
    HildonProgram *program;
    HildonWindow *window;
    AppData *appdata;
    GtkWidget *label;

    /* Initialize the GTK. */
    gtk_init(&argc, &argv);

    /* Create the Hildon program and setup the title */
    program = HILDON_PROGRAM(hildon_program_get_instance());
    g_set_application_name("HildonFindToolbar Example");

    /* Create HildonWindow and set it to HildonProgram */
    window = HILDON_WINDOW(hildon_window_new());
    hildon_program_add_window(program, window);
}

```



```

/* Add example label to window */
label = gtk_label_new("HildonFindToolbar Example");
gtk_container_add(GTK_CONTAINER(window), label);

/* Create AppData */
appdata = g_new0(AppData, 1);
appdata->program = program;
appdata->window = window;
appdata->label = label;

/* Create toolbar for view */
create_toolbar(appdata);

/* Create find toolbar, but keep it hidden */
create_find_toolbar(appdata);

/* Connect signal to X in the upper corner */
g_signal_connect(G_OBJECT(appdata->window), "delete_event",
    G_CALLBACK(item_close_cb), NULL);

/* Show all other widgets except the find toolbar */
gtk_widget_show_all(GTK_WIDGET(appdata->window));
gtk_widget_hide_all(GTK_WIDGET(appdata->find_toolbar));

/* Begin the main application */
gtk_main();

/* Exit */
return 0;
}

```

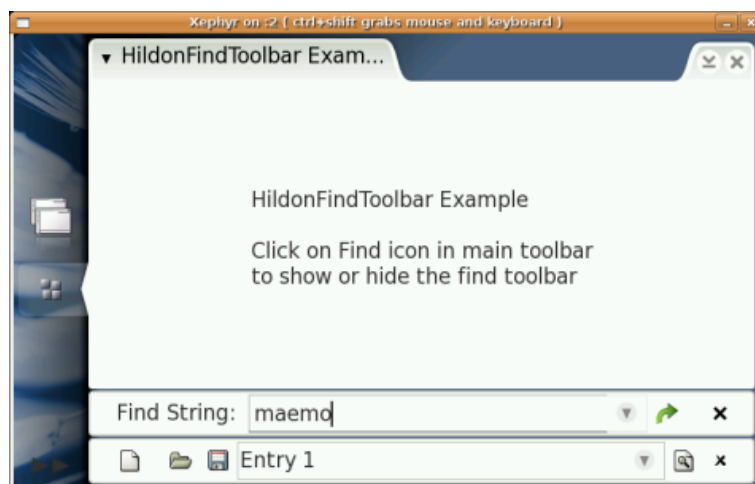
To compile and run, start Xephyr server and

```

$ gcc -o example_findtoolbar example_findtoolbar.c `pkg-config gtk+-2.0 hildon-1 --cflags
--libs` -Wall
$ run-standalone.sh ./example_findtoolbar

```

Now the application is ready; the screenshot below presents the result. The Find toolbar can be opened and closed by pressing the 'Find' toolbar button (second from right). As no real functionality was created for the search, pressing "Find" only prints the search string into the command line.



6.5 Other Hildon Widgets

This section introduces general Hildon widgets, which can and should be used in different maemo applications. Only the most common ones are introduced; see Hildon API documentation [57] for the full list of available widgets. Remember that all normal GTK+ widgets can also be used; see GTK+ Reference Manual at [41].

6.5.1 HildonFileChooserDialog

```
#include <hildon/hildon-program.h>
#include <hildon/hildon-file-chooser-dialog.h>

#include <gtk/gtk.h>

/* Application UI data struct */
typedef struct _AppData AppData;
struct _AppData {
    HildonProgram *program;
    HildonWindow *window;

    GtkWidget * main_vbox;
    GtkWidget * label;

    /* Menu stuff */
    GtkMenu * main_menu;
    GtkWidget * menu_item_file_open;
    GtkWidget * menu_item_file_save;
    GtkWidget * menu_item_quit;
};
```

HildonFileChooserDialog is a dialog used to save and open the user files. It is based on the GtkFileChooser, so the API is similar to the one used in GTK+.

HildonFileChooserDialog is usually opened by event callback of "Open" or "Save" menu entries, or toolbar buttons. To use the file dialog, these callbacks should include similar code as below ([example_file_chooser.c](#)).

```
void cb_example_file_open (GtkWidget * w, AppData * data)
{
    gchar *filename = NULL;

    filename = interface_file_chooser (data,
        GTK_FILE_CHOOSER_ACTION_OPEN);

    if (filename == NULL) {
        filename = "NULL";
    }

    example_show_test_result (data, "Open File", filename);
}

void cb_example_file_save (GtkWidget * w, AppData * data)
{
    gchar *filename = NULL;
    filename = interface_file_chooser (data,
        GTK_FILE_CHOOSER_ACTION_SAVE);

    if (filename == NULL) {
        filename = "NULL";
    }
}
```

```

    }
    else {
        FILE * f = fopen (filename, "w");
        fprintf (f, "This file was generated by Hildon File Chooser
        example.");
        fclose (f);
    }

    example_show_test_result (data, "File saved as", filename);
}

```

The *interface_file_chooser()* function creates a dialog, using the given GtkFileChooserAction:

```

gchar * interface_file_chooser (AppData * appdata, GtkFileChooserAction
    action)
{
    GtkWidget *dialog;
    gchar *filename = NULL;

    dialog = hildon_file_chooser_dialog_new (GTK_WINDOW (appdata->
        window), action);
    gtk_widget_show_all (GTK_WIDGET (dialog));

    if (gtk_dialog_run (GTK_DIALOG (dialog)) == GTK_RESPONSE_OK) {
        filename = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (
            dialog));
    }

    gtk_widget_destroy(dialog);
    return filename;
}

```

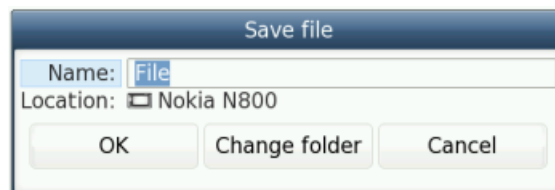
To compile and run this program, include additional hildon-fm-2 to the pkg-config path, as follows:

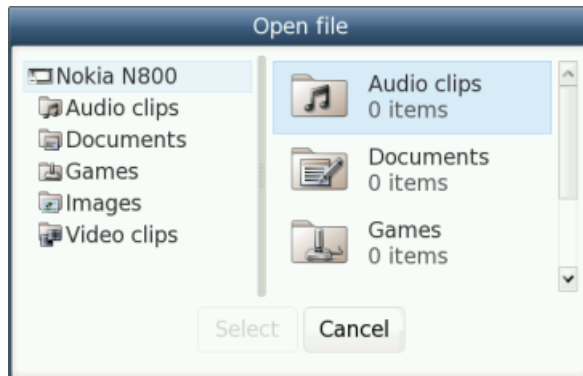
```

$ gcc -o example_file_chooser example_file_chooser.c `pkg-config gtk+-2.0 hildon-1
hildon-fm-2 --cflags --libs` -Wall
$ run-standalone.sh ./example_file_chooser

```

Below are screenshots of both Save and Open dialogs.





For more information about `HildonFileChooserDialog`, see Hildon FM Reference Manual [57] or in section 6.11.

6.5.2 HildonColorChooser

N.B. In Bora, it was named `HildonColorSelector`, and from Chinook onwards, the name is changed to `HildonColorChooser`

```
#include <hildon/hildon-program.h>
#include <hildon/hildon-color-chooser-dialog.h>
#include <hildon/hildon-color-button.h>
#include <gtk/gtk.h>
```

`HildonColorChooser` is a dialog for selecting colors. It is quite similar to `GtkColorSelectionDialog`, but more suitable for embedded devices.

There are two different ways to use `HildonColorChooser`: either by using `HildonColorButton`, showing the selected color in the button area and launching `HildonColorChooser` dialog automatically when clicked, or by creating the dialog manually.

First, the `HildonColorButton` should be created ([example_color_chooser.c](#)). It can be created by `hildon_color_button_new()`, and attached to wherever wanted in the UI.

```
color_button = HILDON_COLOR_BUTTON(hildon_color_button_new());
g_signal_connect(G_OBJECT(color_button), "clicked",
                 G_CALLBACK(color_button_clicked), NULL);
```

In the `color_button_clicked` callback function, the selected color is retrieved from the `HildonColorButton` using the "color" property of the widget.

```
void color_button_clicked(GtkWidget * widget, gpointer data)
{
    GdkColor *new_color = NULL;
    g_object_get(widget, "color", &new_color, NULL);
}
```

If `HildonColorButton` is not needed, `HildonColorChooser` can be created manually:

```
void ui_show_color_chooser(GtkWidget * widget, AppData * appdata)
{
    GdkColor color = {0, 0, 0, 0};
    GtkWidget *selector;
```

```

gint result;

selector = hildon_color_chooser_dialog_new();
/* Set the current selected color to selector */
hildon_color_chooser_dialog_set_color(
    HILDON_COLOR_CHOOSER_DIALOG(selector),
    &color);

/* Show dialog */
result = gtk_dialog_run(GTK_DIALOG(selector));

/* Wait for user to select OK or Cancel */
switch (result) {
case GTK_RESPONSE_OK:
    /* Get the current selected color from selector */
    hildon_color_chooser_dialog_get_color(
        (HILDON_COLOR_CHOOSER_DIALOG(selector)), &color);

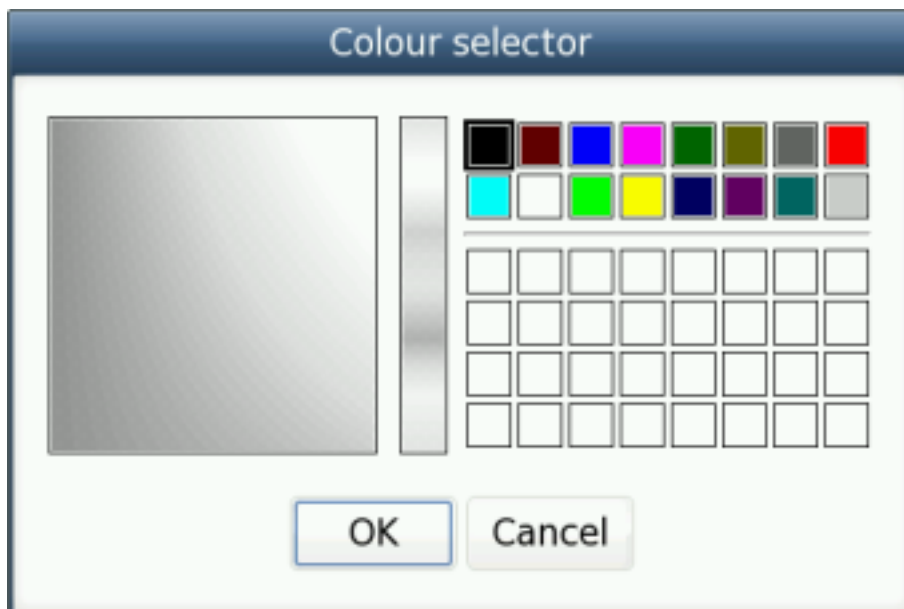
    /* Now the new color is in 'color' variable */
    /* Use it however suitable for the application */

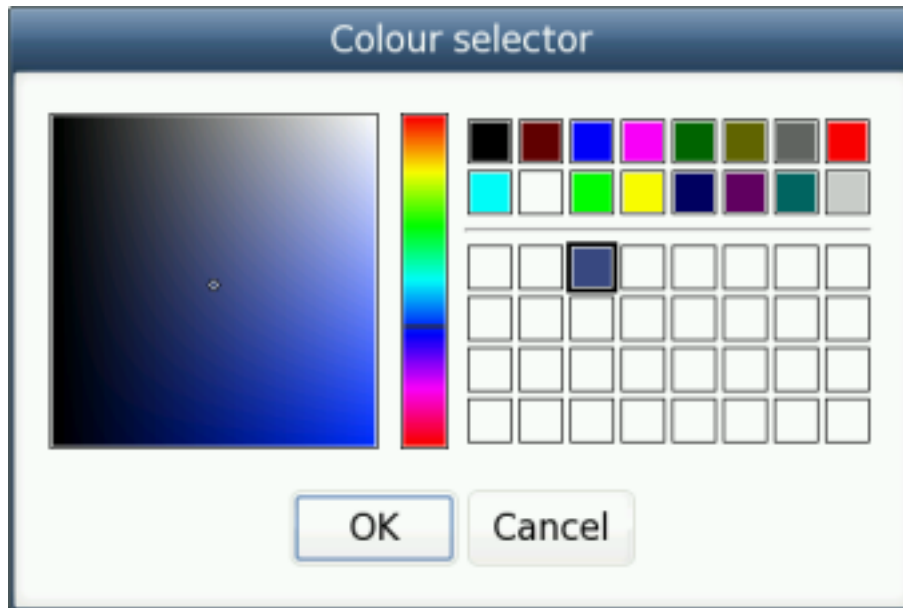
    gtk_widget_destroy(selector);
    break;

default:
    /* If dialog didn't return OK then it was canceled */
    gtk_widget_destroy(selector);
    break;
}
}

```

In HildonColorChooser, a pre-defined color can be selected, or a new color can be generated. Both are presented in the screenshots below.





To compile and run this program:

```
$ gcc -o example_color_chooser example_color_chooser.c `pkg-config gtk+-2.0 hildon-1`
--cflags --libs' -Wall
$ run-standalone.sh ./example_color_chooser
```

6.5.3 HildonFontSelectionDialog

```
#include <hildon/hildon-program.h>
#include <hildon/hildon-font-selection-dialog.h>
#include <gtk/gtk.h>
```

Where `HildonColorChooser` provides methods to query color from the user, `HildonFontSelectionDialog` is used to define font. It contains three tabs with the different font formatting information, and is ideally used in text editors and other applications, where the font style can be selected ([example_font_selector.c](#)).

To be able to use `HildonFontSelectionDialog`, some knowledge of Pango text formatting is required, as the returned variable is of type `PangoAttrList`. For more information about Pango, see [Pango Reference Manual](#) [83].

```
void callback_font_selector(GtkWidget * widget, gpointer data)
{
    HildonFontSelectionDialog *dialog;
    gint result;
    AppData *appdata = (AppData *) data;

    /* Create dialog */
    dialog = HILDON_FONT_SELECTION_DIALOG
        (hildon_font_selection_dialog_new(GTK_WINDOW(appdata->window),
            "Font selector"));

    /* Show the dialog */
```

```

gtk_widget_show_all(GTK_WIDGET(dialog));

/* Wait for user to select OK or Cancel */
result = gtk_dialog_run(GTK_DIALOG(dialog));

if (result == GTK_RESPONSE_OK) {
    /* Get selected font from dialog */
    /* Use it however suitable for the application */

}

/* Close the dialog */
gtk_widget_destroy(GTK_WIDGET(dialog));
}

```

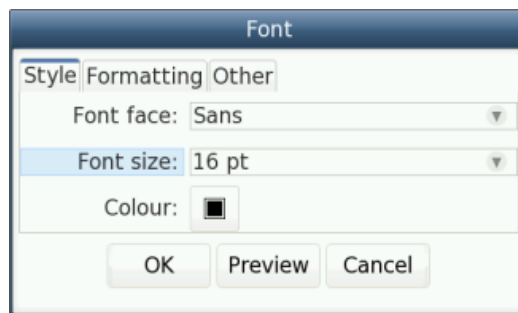
To compile and run this program:

```

$ gcc -o example_font_selector example_font_selector.c `pkg-config gtk+-2.0 hildon-1
hildon-fm-2 --cflags --libs` -Wall
$ run-standalone.sh ./example_font_selector

```

The screenshot below shows the HildonFontSelectionDialog with the Style tab open.



6.5.4 HildonFileDetailsDialog

```

#include <hildon/hildon-program.h>
#include <hildon/hildon-file-details-dialog.h>
#include <gtk/gtk.h>

/* Application UI data struct */
typedef struct _AppData AppData;
struct _AppData {
    HildonProgram *program;
    HildonWindow *window;
};

```

HildonFileDetailsDialog is a dialog to show file information. It has two tabs: the first one containing file name, size, last modification date, and some general information; and the second one being empty for the use of applications. These kinds of dialogs are used not only in a file manager application, but also in all kinds of applications with user files, such as editors and viewers.

The following source code ([example_file_details.c](#)) describes how to create HildonFileDetailsDialog. Remember to make sure the file exists first. A HildonFileSystemModel is needed.

```

/* This variable will get filled with the fullpath of the
 * filename that was called to run this program
 * and it will be inspected by the file details dialog*/
static char *global_running_file;

#define TEXT_FILE "/MyDocs/.example/foo.txt"

/* Create a Hildon File System Model to be used in the File details
dialog */
static HildonFileSystemModel* get_file_system_model(GtkWidget *
ref_widget)
{
    return HILDON_FILE_SYSTEM_MODEL(g_object_new(
        HILDON_TYPE_FILE_SYSTEM_MODEL,
                                "ref_widget", ref_widget,
                                NULL));
}

```

This is how the file details dialog is created. A specific file `TEXT_FILE` (whose fullpath is `global_running_file`) is attached to the `HildonFileSystemModel` object which is attached to the file details dialog.

```

void callback_file_details(GtkWidget * widget, AppData * data)
{
    HildonFileDetailsDialog *dialog;
    GtkWidget *label = NULL;
    HildonFileSystemModel *model = NULL;
    gint result;
    GtkTreeIter iter;

    /* Create a hildon file system model */
    if( (model = get_file_system_model(GTK_WIDGET(data->window)) ) ==
        NULL)
    {
        g_print("could not get file system model\n\n");
        return;
    }
    /* Load file uri */
    if( !hildon_file_system_model_load_uri( model,
                                            TEXT_FILE,
                                            &iter ) )
    {
        g_print("couldnot load uri %s\n", TEXT_FILE);
        return;
    }
    /* Create dialog */
    dialog = HILDON_FILE_DETAILS_DIALOG
        (hildon_file_details_dialog_new_with_model(GTK_WINDOW(data->
            window), model));

    hildon_file_details_dialog_set_file_iter
        (HILDON_FILE_DETAILS_DIALOG(dialog), &iter);
    /*.....*/
}

```

The file details dialog can also have an additional application-specific tab. In this example, the extra tab only reshows the file name; but in a real application, it could contain line count, preview of a document, length of the sound, etc. The type of the object "additional tab" is `GTK_LABEL`, so all kind of widgets can be inserted inside it.


```

/* This variable will get filled with the fullpath of the
 * filename that was called to run this program
 * and it will be inspected by the file details dialog*/
static char *global_running_file;

#define TEXT_FILE "/MyDocs/.example/foo.txt"

void callback_file_details(GtkWidget * widget, AppData * data)
{
    /*.....*/
    /* Set the dialog to show tabs */
    g_object_set(dialog, "show-tabs", TRUE, NULL);

    /* Create some content to additional tab */
    label =
        gtk_label_new(g_strdup_printf
            ("Name of this \nfile really is:\n%s",
             global_running_file)
        );

    /* Add content to additional tab label */
    g_object_set(dialog, "additional-tab", label, NULL);

    /* Change tab label */
    g_object_set(dialog, "additional-tab-label", "More Info", NULL);

    /*.....*/
}

```

Then the dialog is run just like any other GTK dialog.

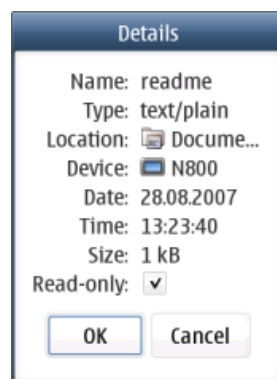
```

/* Show the dialog */
gtk_widget_show_all(GTK_WIDGET(dialog));

/* Wait for user to select OK or CANCEL */
result = gtk_dialog_run(GTK_DIALOG(dialog));

```

Here is an example screenshot of the File Details dialog.



To compile and run, start Xephyr server and:

```

$ gcc -o example_file_details example_file_details.c `pkg-config gtk+-2.0 hildon-1
hildon-fm-2 --cflags --libs` -Wall
$ run-standalone.sh ./example_file_details

```

6.5.5 HildonBanner

```
#include <hildon/hildon-program.h>
#include <hildon/hildon-banner.h>
#include <gtk/gtk.h>
```

HildonBanners are used in many places in maemo applications. They show a small information banner in the top right-hand corner of the application view for a few seconds, and then disappear automatically. They are used for all kinds of notifications, such as "File Saved", "Unable to make connection" or "Please choose only one option".

There are multiple functions to choose from, depending on the information that is wanted to be shown. The banner can show plain text, custom icon, or progress bar to the user. The next example ([example_banner.c](#)) presents all these alternatives. First the function which shows the different HildonBanners one by one.

```
static gint banner_type = 1;
GtkWidget *banner = NULL;

/* Callback to show information banners */
void show_banner(GtkButton * widget, HildonWindow * window)
{
    switch (banner_type) {
    case 1:
        /* Show normal information banner and this automatically goes
           away */
        hildon_banner_show_information(GTK_WIDGET(window), NULL, "Hi
            there!");
        break;

    case 2:
        /* Information banner with animation icon.
           * This banner does not automatically disappear. */
        banner = hildon_banner_show_animation(GTK_WIDGET(window), NULL,
            "This is animation icon");
        ;
        break;

    case 3:
        /* Remove current information banner */
        gtk_widget_destroy(GTK_WIDGET(banner));
        break;

    case 4:
        /* Information banner with progressbar */
        banner = hildon_banner_show_progress(GTK_WIDGET(window), NULL,
            "Info with progress bar");
        /* Set bar to be 20% full */
        hildon_banner_set_fraction(HILDON_BANNER(banner), 0.2);
        break;

    case 5:
        /* Set bar to be 80% full */
        hildon_banner_set_fraction(HILDON_BANNER(banner), 0.8);
        break;

    case 6:
        /* With sixth click, end the application */
        gtk_main_quit();
    }
```

```

    }

    /* Increase the counter */
    banner_type++;
}

```

The main function introduces how to pack widgets inside the HildonWindow using GtkVBox, which is a vertical box container in GTK+. Inside the vbox, it places one button, which is connected to the show_banner callback function.

```

/* Main application */
int main(int argc, char *argv[])
{
    /* Create needed variables */
    HildonProgram *program;
    HildonWindow *window;
    GtkWidget *main_vbox;
    GtkWidget *button1;

    /* Initialize the GTK. */
    gtk_init(&argc, &argv);

    /* Create the hildon program and setup the title */
    program = HILDON_PROGRAM(hildon_program_get_instance());
    g_set_application_name("App Title");

    /* Create HildonWindow and set it to HildonProgram */
    window = HILDON_WINDOW(hildon_window_new());
    hildon_program_add_window(program, window);

    /* Add vbox to appview */
    main_vbox = gtk_vbox_new(FALSE, 0);
    gtk_container_add(GTK_CONTAINER(window), main_vbox);

    /* Add button to vbox */
    button1 = gtk_button_new_with_label("Show Info");
    gtk_box_pack_start(GTK_BOX(main_vbox), button1, FALSE, TRUE, 0);

    /* Add signal listener to button */
    g_signal_connect(G_OBJECT(button1), "clicked",
                     G_CALLBACK(show_banner), window);

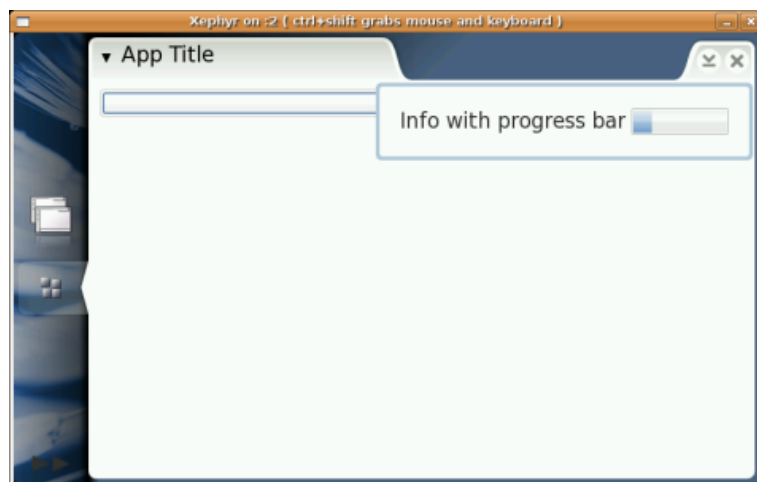
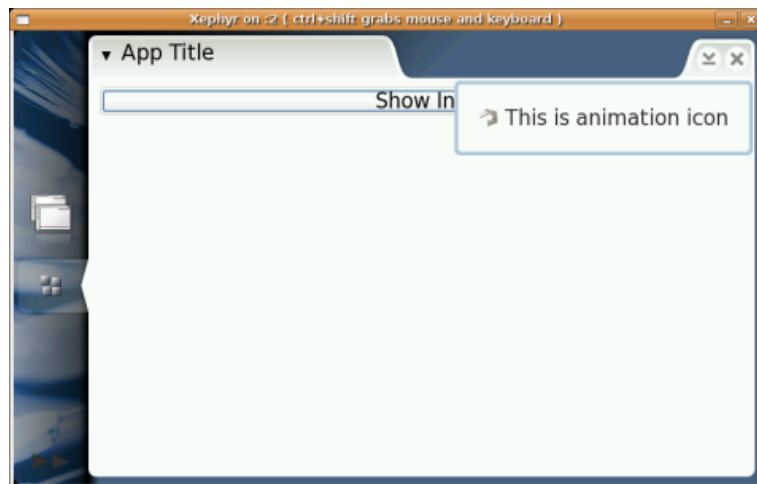
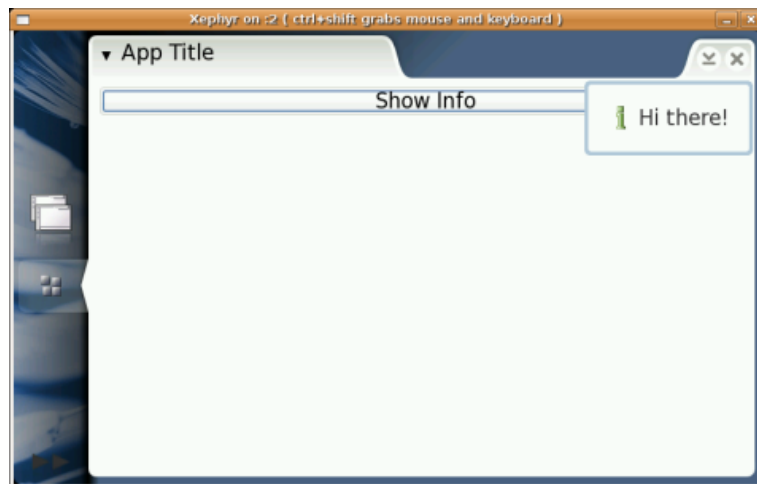
    /* Connect signal to X in the upper corner */
    g_signal_connect(G_OBJECT(window), "delete_event",
                     G_CALLBACK(gtk_main_quit), NULL);

    /* Begin the main application */
    gtk_widget_show_all(GTK_WIDGET(window));
    gtk_main();

    /* Exit */
    return 0;
}

```

Using HildonBanner widgets is very simple, as the HildonWindow takes care of showing them. The output of the application can be seen in the screenshots below. When clicking the button, a normal banner appears first, then one with the custom icon, and at last the progress bar. The last click closes the application.



There are a lot more Hildon widgets available; to get more information about them, see the Hildon API documentation. To compile and run, start

Xephyr server and:

```
$ gcc -o example_banner example_banner.c `pkg-config gtk+-2.0 hildon-1 --cflags --libs`  
-Wall  
$ run-standalone.sh ./example_banner
```

6.6 Using Maemo Input Mechanism

6.6.1 Touchscreen (Mouse)

The maemo platform is designed to be used with a touchscreen, and the SDK with a mouse to emulate the touchscreen functionality. Some aspects to remember:

- The main functional difference between the use of mouse and touchscreen needs to be remembered: It is not possible to move cursor position on the touchscreen without "pressing" the button all the time. This difference may be noticed e.g. in a drawing application or game, where the listening events need to be designed so that user can click left and right side of the screen without ever "moving" the mouse pointer from left to right.
- Only the left mouse button should be used. No functionality should be added for the right mouse button. Touchscreen can only recognize one kind of a 'click'.
- When menus, usually opened by pressing the right mouse button, are wanted to be created, it is possible to implement them using a slightly longer click of the left mouse button. These kinds of context-sensitive menus are supported by the platform; see GtkWidget tap-and-hold setup function in the Hildon API documentation.
- GtkWidget in maemo has been modified to pass a special `insensitive_press` signal, when the user presses on an insensitive widget. This was added because of the need for actions such as showing a banner, when disabled widgets are pressed in menu.

With this information, it is possible to fully emulate touchscreen behavior with the mouse in the maemo SDK, so that there should not be any problem when using the application in the actual device.

6.6.2 Context Sensitive Menu












Context sensitive menus are implemented in the GtkWidget class. An application may register these menus for its widgets, accessed by tap-and-hold over the widget.

An example on how to add this functionality can be found in [example_context.c](#).

6.6.3 Hardware Keys

```
#include <hildon/hildon-program.h>  
#include <hildon/hildon-banner.h>  
#include <gtk/gtk.h>  
#include <gdk/gdkkeysyms.h>
```

The maemo device has hardware keys, which are also supported in the SDK using different keyboard events. The actual number and type of keys depends on the device, but the common ones are presented in the table below.

Button	Description	Keyboard	Key Event
	Move up	Arrow key up	HILDON_HARDKEY_UP
	Move down	Arrow key down	HILDON_HARDKEY_DOWN
	Move left	Arrow key left	HILDON_HARDKEY_LEFT
	Move right	Arrow key right	HILDON_HARDKEY_RIGHT
	Select, Confirm	Return	HILDON_HARDKEY_SELECT
	Cancel, Close	Esc	HILDON_HARDKEY_ESC
	Open menu	F4	HILDON_HARDKEY_MENU
	Show Home	F5	HILDON_HARDKEY_HOME
	Full screen	F6	HILDON_HARDKEY_FULLSCREEN
	Increase / Zoom in / Volume up	F7	HILDON_HARDKEY_INCREASE
	Decrease / Zoom out / Volume down	F8	HILDON_HARDKEY_DECREASE

Adding support for a wider number of keys is easy, as the actual key pressing signals can be implemented with GDK key events. An example ([example_hard_keys.c](#)) of this mapping is presented below, first the callback function that is called whenever keys are pressed.

```
/* Callback for hardware keys */
gboolean key_press_cb(GtkWidget * widget, GdkEventKey * event,
                      HildonWindow * window)
{
    switch (event->keyval) {
        case HILDON_HARDKEY_UP:
            hildon_banner_show_information(GTK_WIDGET(window), NULL, "
                Navigation Key Up");
            return TRUE;

        case HILDON_HARDKEY_DOWN:
            hildon_banner_show_information(GTK_WIDGET(window), NULL, "
                Navigation Key Down");
            return TRUE;

        case HILDON_HARDKEY_LEFT:
            hildon_banner_show_information(GTK_WIDGET(window), NULL, "
                Navigation Key Left");
            return TRUE;
    }
}
```

```

case HILDON_HARDKEY_RIGHT:
    hildon_banner_show_information(GTK_WIDGET(window), NULL, "
        Navigation Key Right");
    return TRUE;

case HILDON_HARDKEY_SELECT:
    hildon_banner_show_information(GTK_WIDGET(window), NULL, "
        Navigation Key select");
    return TRUE;

case HILDON_HARDKEY_FULLSCREEN:
    hildon_banner_show_information(GTK_WIDGET(window), NULL, "Full
        screen");
    return TRUE;

case HILDON_HARDKEY_INCREASE:
    hildon_banner_show_information(GTK_WIDGET(window), NULL, "
        Increase (zoom in)");
    return TRUE;

case HILDON_HARDKEY_DECREASE:
    hildon_banner_show_information(GTK_WIDGET(window), NULL, "
        Decrease (zoom out)");
    return TRUE;

case HILDON_HARDKEY_ESC:
    hildon_banner_show_information(GTK_WIDGET(window), NULL, "
        Cancel/Close");
    return TRUE;
}

return FALSE;
}

```

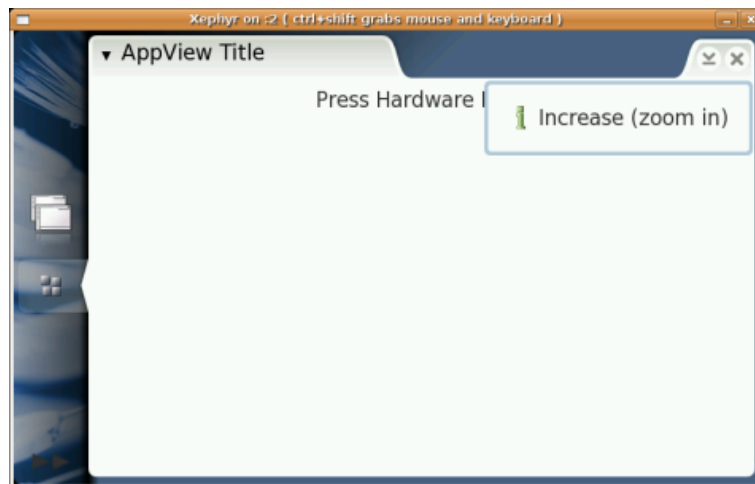
In the main function, the callback function is connected with `g_signal_connect` to listen to "key_press_event" signal. Whenever a hardware key is pressed, `key_press_cb` function is called to process the event.

```

int main( int argc, char* argv[] )
{
    ...
    /* Add hardware button listener to application */
    g_signal_connect(G_OBJECT(window),
        "key_press_event", G_CALLBACK(key_press_cb), window);
    ...
}

```

The output of the hardware key example application is shown below in the case zoom in (F7) is pressed.



6.6.4 Maemo's Gtk Accelerator and Mnemonics

GtkAccelerator

Just like the standard GTK+ lib, Hildon can also recognize accelerator keys. For example, to recognize and handle when Ctrl+g or Ctrl+G is pressed.

```
guint modifiers;
modifiers = gtk_accelerator_get_default_mod_mask ();

if ((event->state & modifiers) == GDK_CONTROL_MASK)
    && (event->keyval == GDK_g || event->keyval == GDK_G))
{
    hildon_banner_show_information(GTK_WIDGET(window), NULL,
                                  "Ctrl+g or Ctrl+G key combination"
                                  );
}
```

More information on the GTK+ key events can be found at [GDK Key Values \[25\]](#).

Maemo 4.x also supports GtkAcceleratorGroup, which is a group of keyboard accelerators, typically attached to a toplevel Gtk Window. To learn more about GtkAccelerator, look at [GTK+ Accelerator Groups \[39\]](#). As an example, usually Gtk Applications have "Ctrl + Q" as a shortcut for menu item "Quit" to quit the application. Following is how it is implemented in maemo 4.x.

1. First of all, create a GtkAccelGroup and add it to the main window

```
GtkAccelGroup* accel_group;
accel_group = gtk_accel_group_new ();
gtk_window_add_accel_group(GTK_WINDOW(main_window),
                           accel_group);
```

2. Create a sample menu item "Quit" window

```
#define ACCEL_PATH_ROOT "/menu"
#define ACCEL_PATH_QUIT ACCEL_PATH_ROOT"/item_quit"

/* Create new main menu */
```



```

main_menu = gtk_menu_new();
item_quit = gtk_menu_item_new_with_label("Quit");

/* Attach the menu item to the Gtk Accelerator defined above
*/
gtk_menu_set_accel_group( GTK_MENU(main_menu),
                           accel_group);
gtk_menu_item_set_accel_path( GTK_MENU_ITEM(item_quit),
                              ACCEL_PATH_QUIT);

gtk_menu_append(main_menu, item_quit);

```

The `accel_group` is attached to the main menu, which contains the "Quit" menu item. Also, the `item_quit` is attached with an accelerator path, which is a constant string "<example>/menu/item_quit". Basically, this string is a unique string that identifies the menu item "Quit" of the current application. For example, you can also have "<example>/menu/item_new" as a valid accelerator path for "Open new file" menu item. Please read on for more information on accelerator path.

More information on how to attach an Accelerator with a Menu item can be found in GTK's documentation.

3. Define the accelerator path

```

gtk_accel_map_add_entry(ACCEL_PATH_QUIT, GDK_q, GDK_CONTROL_MASK
);

```

The above function defines, on the application's level, that a "Ctrl+q" key accelerator is attached to the accelerator path "<example>/menu/item_quit". How to define a proper accelerator path and get it mapped is further explained in [GTK's documentation](#).

Now "Ctrl+q" will activate the menu item "Quit".
Additionally, Gtk-Accel must be enabled by a gconf value:

```
$ gconftool-2 -s /apps/osso/gtk/enable-accel -t bool true
```

Gtk Mnemonics

[GTK Mnemonics](#) can be enabled by:

```
$ gconftool-2 -s /apps/osso/gtk/enable-mnemonics -t bool true
```

6.6.5 Maemo Text Input Methods

The maemo platform includes text input methods used with touchscreen. There are two different input types: virtual keyboard and handwriting recognition. Only the virtual keyboard is provided in maemo SDK, but as the input is transparent for the application, all applications will automatically support also handwriting recognition. When using the device, it is possible to select between two keyboard layouts. When touching the screen using a finger, a thumb keyboard is provided.

The input methods work automatically, so that whenever text widgets (entry, text view) are pressed, or text inside them selected, the keyboard pops up. In the same way, the keyboard is hidden automatically, when the user selects another widget.

For the development use, it is also possible to use the PC keyboard to input text. It should work without any changes, but if any problems are experienced, it can also be enabled in SDK by clicking the right mouse button on top of a text widget, and then selecting "Input Methods -> X Input Method" from the menu.

6.6.6 Application Layout Considerations

The layout of maemo applications can be described as wide and not very tall, because of the screen dimensions (800x480 pixels). This wideness is emphasized when a text input method is popped up. Therefore, the design of the application UI should follow some rules.

- Make the application UI area scalable by using the GTK+ layout containers, such as GtkVBox, and do not use any fixed layout, such as GtkFixed. In case the application area does not fit into the screen once the keyboard pops up, it should be scrollable. This can be achieved by adding application area widgets inside GtkScrolledWindow.
- Use only one visible toolbar at any given time in one application. An exception for this is the HildonFindToolbar, which should pop up on top of the main toolbar when a search is performed. It is also a good design principle to allow the user to hide toolbars through the menu to maximize the application area.
- Enable full screen mode in the application. The mode can be toggled with full screen hardware button. This is easy to implement with `_window_fullscreen()` and `gtk_window_unfullscreen()` functions.
- Avoid large dialogs taking up a large part of the application area height. This is especially important if the dialog needs text input, because the virtual keyboard is then needed to be visible at the same time.

6.7 Writing Control Panel Applets

The control panel is designed to be extendable, and items can be added to it with dynamic libraries with certain functions and desktop files describing the library.

6.7.1 Functions

There are two functions needed for a control panel applet ([libapplet.c](#)). These functions are defined in

```
#include <hildon-cp-plugin/hildon-cp-plugin-interface.h>
```

The first and more important one, `execute`, is called when the applet is activated from Control Panel. Usually this function creates a dialog and waits until

it is done. Any dialog created should be modal to the window in parameter. **N.B.** The library might be unloaded when execute returns, so no `g_timeouts`, `gconf_notify`s or such should be left when done. Gtk or osso initialization is not needed, as they are already done at this point.

```
#include <hildon-cp-plugin/hildon-cp-plugin-interface.h>
#include <gtk/gtk.h>

osso_return_t execute(osso_context_t *osso, gpointer data, gboolean
    user_activated)
{
    GtkWidget *dialog;
    gint response;

    /* Create dialog with OK and Cancel buttons. Leave the separator out,
     * as we do not have any content. */
    dialog = gtk_dialog_new_with_buttons(
        "Hello control panel",
        GTK_WINDOW(data),
        GTK_DIALOG_MODAL | GTK_DIALOG_NO_SEPARATOR,
        GTK_STOCK_OK,
        GTK_RESPONSE_OK,
        GTK_STOCK_CANCEL,
        GTK_RESPONSE_CANCEL,
        NULL);

    /* ... add something to the dialog ... */

    if (!user_activated)
    {
        /* ... load state ... */
    }

    /* Wait until user finishes the dialog. */
    response = gtk_dialog_run(GTK_DIALOG(dialog));

    if (response == GTK_RESPONSE_OK)
    {
        /* ... do something with the dialog stuff ... */
    }

    /* Free the dialog (and it's children) */
    gtk_widget_destroy(GTK_WIDGET(dialog));

    return OSSO_OK;
}
```

The other function is called `save_state`. It is called when application using the applet is saving state. Usually this does nothing, but if support for state saving is needed, this should be used.

```
osso_return_t save_state(osso_context_t *osso, gpointer data)
{
    /* ... save state ... */

    return OSSO_OK;
}
```

6.7.2 Building Applet

To use the applet, it needs to be built into a dynamic library aka shared object. This can be accomplished by giving flag `-shared` to `gcc`.

```
[sbox-DIABLO_X86: ~] > gcc -shared `pkg-config gtk+-2.0 libosso --libs --cflags` libapplet.c  
-o libapplet.so
```

The binary produced by `gcc` should be installed to path specified in `hildon-control-panel` `pkg-config` entry. This path can be obtained with `pkg-config hildon-control-panel --variable=pluginlibdir`. By default this is `/usr/lib/hildon-control-panel`.

6.7.3 The .desktop File

Any control panel applet needs a desktop file describing it. The file contains metadata like name, icon name and library of the applet. The applet desktop file is much like the desktop file for any other application.

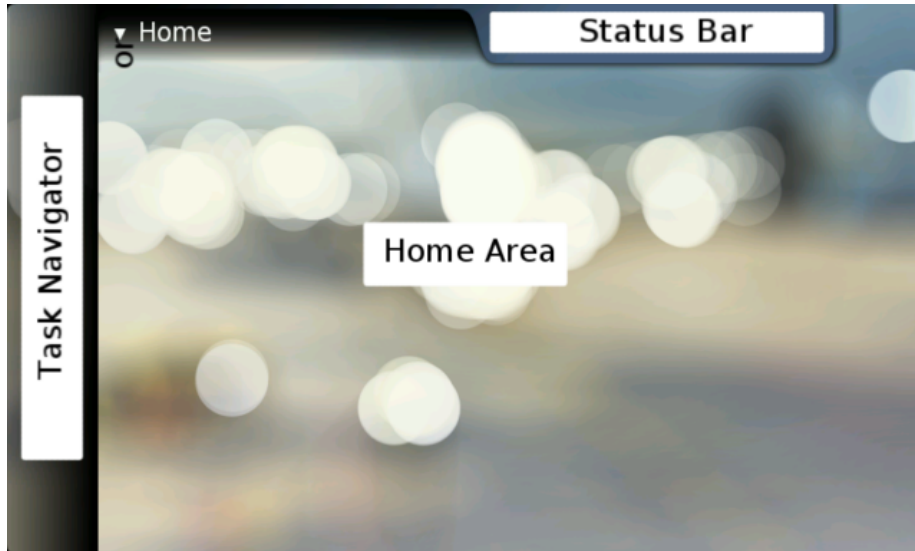
Here is an example desktop file ([applet.desktop](#)) for the applet created above. Maemo 4.x brings new "Categories" field.

```
[Desktop Entry]  
Encoding=UTF-8  
Version=1.0  
Name=Control Panel Hello World  
Comment=A control panel example applet  
Type=HildonControlPanelPlugin  
Icon=qgn_list_cp_isetup  
X-control-panel-plugin=libapplet.so  
Categories=general
```

The desktop file must be installed to directory specified in `pkg-config` file for `hildon-control-panel`. This directory can be obtained by `pkg-config hildon-control-panel --variable=plugindesktopentrydir`. By default, this is `/usr/share/applications/hildon-control-panel`.

6.8 Writing Hildon Desktop Plug-ins

6.8.1 Introduction



Hildon Desktop consists of several subcomponents: Task Navigator, Home Area and Status Bar. They support additional plug-ins that are displayed on each these components, extending the functionality of the host application. This section will explain how to write plug-ins for the desktop components using the new GTypeModule-based API. The section also explains the API for writing applets for Control Panel.

6.8.2 Task Navigator Plug-ins

The Task Navigator implements a modular plug-in architecture, allowing binary plug-ins to be loaded. This extends the functionality of the Task Navigator, and is achieved with task buttons. The license for the Task Navigator plug-in can be open source or closed source; in this way, the Task Navigator is versatile and ideal for use in many kinds of devices and applications.

Task Navigator Plug-in API

Task Navigator plug-ins can have any widget as their representation on the Task Navigator plug-in area. The plug-in representation, statically visible in the task navigator, can be e.g. `GtkButton`. Task navigator loads the library and displays the main widget of the plug-in in the plug-in area of the Task Navigator.

The plug-ins are responsible for their own signals, e.g. `clicked`, `toggle` and related events, and implementing the callback functions for them. Task navigator only provides a container for the plug-ins; the plug-ins themselves are responsible for implementing their own UI.

Desktop File Location

Task Navigator plug-ins must provide a desktop file. It is placed in the directory given by:

```
pkg-config osso-af-settings --variable=tasknavigatordesktopentrydir
```

and is by default: /usr/share/applications/hildon-navigator/

Desktop File Contents

The following describes the required contents of the .desktop file.

The Task Navigator Configuration Control Panel applet is the only one involved with the contents of these .desktop files. Thus, the Task Navigator itself does not read them, and the absence of the file does not prevent the plug-in from working.

The following is an example .desktop file for the Task Navigator applet:

```
[Desktop Entry]
Name=Hello World
Type=default
X-Path=/usr/lib/hildon-navigator/libhelloworld-tn.so
```

- Name of the plug-in in Control Panel.
- Type is always default.
- X-Path tells where to find the plug-in.

Public Interface

Should look like a usual GObject-based class/object declation. The following is an example implementation:

```
#ifndef HELLO_NAVIGATOR_PLUGIN_H
#define HELLO_NAVIGATOR_PLUGIN_H

#include <glib-object.h>

/* For Task Navigator plugins */
#include <libhildondesktop/tasknavigator-item.h>

G_BEGIN_DECLS

/* Common struct types declarations */
typedef struct _HelloNavigatorPlugin HelloNavigatorPlugin;
typedef struct _HelloNavigatorPluginClass HelloNavigatorPluginClass;
typedef struct _HelloNavigatorPluginPrivate HelloNavigatorPluginPrivate;
;

/* Common macros */
#define HELLO_TYPE_NAVIGATOR_PLUGIN ( \
    hello_navigator_plugin_get_type () )
#define HELLO_NAVIGATOR_PLUGIN(obj) ( \
    G_TYPE_CHECK_INSTANCE_CAST ((obj), \
    HELLO_TYPE_NAVIGATOR_PLUGIN, HelloNavigatorPlugin))
#define HELLO_NAVIGATOR_PLUGIN_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST \
    (klass), \
    HELLO_TYPE_NAVIGATOR_PLUGIN, HelloNavigatorPluginClass))
```

```

#define HELLO_IS_NAVIGATOR_PLUGIN(obj) (
    G_TYPE_CHECK_INSTANCE_TYPE ((obj), \
    HELLO_TYPE_NAVIGATOR_PLUGIN))
#define HELLO_IS_NAVIGATOR_PLUGIN_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE
    ((klass), \
    HELLO_TYPE_NAVIGATOR_PLUGIN))
#define HELLO_NAVIGATOR_PLUGIN_GET_CLASS(obj) (
    G_TYPE_INSTANCE_GET_CLASS ((obj), \
    HELLO_TYPE_NAVIGATOR_PLUGIN, HelloNavigatorPluginClass))

/* Instance struct */
struct _HelloNavigatorPlugin
{
    TaskNavigatorItem titem;

    HelloNavigatorPluginPrivate *priv;
    GtkWidget *button;
    GtkWidget *menu;
};

/* Class struct */
struct _HelloNavigatorPluginClass
{
    TaskNavigatorItemClass parent_class;
};

GType hello_navigator_plugin_get_type (void);
GtkWidget *hello_world_button_new (int padding);

G_END_DECLS

#endif /* HELLO_NAVIGATOR_PLUGIN_H */

```

hello_navigator_plugin_init

The function is called upon initialization. It is declared with `HD_DEFINE_PLUGIN`. It creates the widget that is displayed on the Task Navigator. As mentioned earlier, this is typically a `GtkButton`. The Task Navigator automatically sizes the button correctly. It also initializes the pop-up menu.

If the button needs to have proper skin, its name should be set as "hildon-navigator-button-one" with `gtk_widget_set_name` function.

The following is an example implementation:

```

HD_DEFINE_PLUGIN (HelloNavigatorPlugin, hello_navigator_plugin,
    TASKNAVIGATOR_TYPE_ITEM);

static void
hello_navigator_plugin_init (HelloNavigatorPlugin *navigator_plugin)
{
    GtkWidget *button;

    button = hello_world_button_new (10);
    navigator_plugin->button=button;
    navigator_plugin->menu=create_menu();
    g_signal_connect (G_OBJECT (button), "clicked",
        G_CALLBACK (popup_menu), navigator_plugin);

    gtk_widget_set_size_request (button, 80, 80);
}

```

```

gtk_widget_set_name (button, "hildon-navigator-button-one");

gtk_widget_show_all (button);

gtk_container_add (GTK_CONTAINER (navigator_plugin), button);
gtk_widget_show_all (navigator_plugin);
}

static void
hello_navigator_plugin_class_init (HelloNavigatorPluginClass *class)
{
}

```

It is advisable to set a pointer to the button in the data that is returned, since it is used later on (navigator_plugin->button).

Initializing Menu

The menu is created, and proper pop-up, position and action (show_dialog) functions are defined.

The following is an example implementation:

```

static void
show_dialog (GtkWidget *item, HelloNavigatorPlugin *thw)
{
    GtkWidget *dialog;
    dialog = gtk_message_dialog_new (NULL,
                                     GTK_DIALOG_MODAL |
                                     GTK_DIALOG_DESTROY_WITH_PARENT,
                                     GTK_MESSAGE_INFO,
                                     GTK_BUTTONS_CLOSE,
                                     "Hello world!");

    gtk_window_set_title (GTK_WINDOW(dialog), "TN Plugin Example");
    gtk_dialog_run (GTK_DIALOG (dialog));
    gtk_widget_destroy (dialog);
}

static void
menu_position (GtkMenu *menu,
               gint *x,
               gint *y,
               gboolean *push_in,
               HelloNavigatorPlugin *thw)
{
    g_return_if_fail (thw->button);
    *push_in = TRUE;

    *x = thw->button->allocation.x + thw->button->allocation.width;
    *y = thw->button->allocation.y;
}

static void
popup_menu (GtkWidget *button, HelloNavigatorPlugin *plugin)
{
    if (!plugin->menu)
        return;

    gtk_menu_popup (GTK_MENU (plugin->menu),
                    NULL,
                    NULL,

```



```

        (GtkMenuPositionFunc)menu_position,
        plugin,
        0,
        gtk_get_current_event_time());
}

static GtkWidget*
create_menu (void)
{
    GtkWidget *menu;
    GtkWidget *menu_item;

    menu = gtk_menu_new ();

    menu_item = gtk_menu_item_new_with_label ("Hello World!");
    g_signal_connect (G_OBJECT (menu_item), "activate",
                      G_CALLBACK (show_dialog), NULL);

    gtk_menu_append (menu, menu_item);

    /* Name the menu to get the appropriate theming */
    gtk_widget_set_name (menu, "menu_from_navigator");

    gtk_widget_show_all (menu);

    return menu;
}

```

Installing Plug-ins

The plug-in install path can be received using command:

```
pkg-config osso-af-settings --variable=hildondesktoplibdir
```

6.8.3 Home Plug-ins

Home plug-ins are located in the Home Area of the desktop. They can be resized, if the resizable flag X-home-applet-resizable is mentioned in the .desktop file of the plug-in.

Desktop File Location

Each home applet needs to provide a .desktop file. The .desktop file location is determined with:

```
pkg-config osso-af-settings --variable=homedesktopentrydir
```

and it defaults to: /usr/share/applications/hildon-home

Desktop File Contents

The following is an example hello-world-home.desktop:

```

[Desktop Entry]
Name=Hello, World!
Comment=Example Home plugin
Type=default
X-Path=libhelloworld-home.so

```

The mandatory fields are the same as for Task Navigator plug-ins.

Home Plug-in Implementation

Home plug-ins should inherit from libhildondesktop's HildonDesktopHomeItem. Here is an example header file:

```
#ifndef HELLO_HOME_PLUGIN_H
#define HELLO_HOME_PLUGIN_H

#include <glib-object.h>

#include <libhildondesktop/hildon-desktop-home-item.h>

G_BEGIN_DECLS

/* Common struct types declarations */
typedef struct _HelloHomePlugin HelloHomePlugin;
typedef struct _HelloHomePluginClass HelloHomePluginClass;
typedef struct _HelloHomePluginPrivate HelloHomePluginPrivate;

/* Common macros */
#define HELLO_TYPE_HOME_PLUGIN ( \
    hello_statusbar_plugin_get_type () \
)
#define HELLO_HOME_PLUGIN(obj) (G_TYPE_CHECK_INSTANCE_CAST \
    ((obj), \
    HELLO_TYPE_HOME_PLUGIN, HelloHomePlugin))
#define HELLO_HOME_PLUGIN_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST (( \
    klass), \
    HELLO_TYPE_HOME_PLUGIN, HelloHomePluginClass))
#define HELLO_IS_HOME_PLUGIN(obj) (G_TYPE_CHECK_INSTANCE_TYPE \
    ((obj), \
    HELLO_TYPE_HOME_PLUGIN))
#define HELLO_IS_HOME_PLUGIN_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE (( \
    klass), \
    HELLO_TYPE_HOME_PLUGIN))
#define HELLO_HOME_PLUGIN_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS (( \
    obj), \
    HELLO_TYPE_HOME_PLUGIN, HelloHomePluginClass))

/* Instance struct */
struct _HelloHomePlugin
{
    HildonDesktopHomeItem hitem;

    HelloHomePluginPrivate *priv;
};

/* Class struct */
struct _HelloHomePluginClass
{
    HildonDesktopHomeItemClass parent_class;
};

GType hello_home_plugin_get_type (void);

G_END_DECLS

#endif
```

The main functionality consists of required headers, HD_DEFINE_PLUGIN and init functions. The following example creates a button which is defined

in the libhelloworld.h header (see example files) and assigns a dialog showing function to it. Instead of a button, any other suitable GTK+ widget may be used. The widget will be shown in the Home view.

```
#include <glib.h>
#include <gtk/gtk.h>

#include <libhildondesktop/libhildondesktop.h>

#include "hello-world-home.h"
#include "libhelloworld.h"

HD_DEFINE_PLUGIN (HelloHomePlugin, hello_home_plugin,
                  HILDON_DESKTOP_TYPE_HOME_ITEM);

static void
hello_home_plugin_init (HelloHomePlugin *home_plugin)
{
    GtkWidget *button;

    button = hello_world_button_new (10);
    g_signal_connect (button, "clicked",
                      G_CALLBACK (hello_world_dialog_show),
                      NULL);

    gtk_widget_show_all (button);

    /* Set the resizing behavior */
    hildon_desktop_home_item_set_resize_type (HILDON_DESKTOP_HOME_ITEM (
        home_plugin), HILDON_DESKTOP_HOME_ITEM_RESIZE_BOTH);

    gtk_container_add (GTK_CONTAINER (home_plugin), button);
}

static void
hello_home_plugin_class_init (HelloHomePluginClass *class)
{
}
```

Installing Plug-ins

The plug-in install path can be received using command:

```
pkg-config osso-af-settings --variable=hildondesktoplibdir
```

6.8.4 Status Bar Plug-ins

The Status Bar contains status-bar plug-ins. Order of plug-ins and other properties can be set with the Task Navigator and Status Bar configuration applet found on the Control Panel ("Navigation applet").

Status Bar plug-ins are divided into three categories; namely, permanent, conditional and temporal. Permanent plug-ins are shown all the time. Conditional and temporal plug-ins are shown only when the condition is fulfilled.

Desktop File Location

The .desktop file shall be placed in the directory given by:

```
pkg-config osso-af-settings --variable=statusbardesktopentrydir
```

which defaults to: /usr/share/applications/hildon-status-bar/.

Desktop File Contents

Each plug-in has to offer a .desktop file:

```
[Desktop Entry]
Name=<logical name identifier>
Icon=<logical icon identifier>
Category=<permanent/conditional/temporal>,
        the default value is permanent if the key value read fails.
X-Path=lib<plugin name>.so
Type=default
Mandatory=<true/false>,
        if not set, the default value is false
```

Here is an example:

```
[Desktop Entry]
Name=helloworld_sb
Icon=hello-world
Category=temporal
Type=default
X-Path=libhelloworld_sb.so
```

Header File

Example header file:

```
#ifndef _HILDON_STATUS_BAR_HELLOWORLD_H_
#define _HILDON_STATUS_BAR_HELLOWORLD_H_

/* StatusBarItem */
#include <libhildondesktop/statusbar-item.h>

/* osso_context_t */
#include <libosso.h>

/* gboolean, gint, G_BEGIN_DECLS/G_END_DECLS */
#include <glib.h>

/* GtkWidget */
#include <gtk/gtk.h>

G_BEGIN_DECLS

/* Every plug-in has a constant priority */
#define HILDON_STATUS_BAR_HELLOWORLD_PRIORITY 1

#define HILDON_STATUS_BAR_HELLOWORLD_ICON_SIZE 40

typedef struct _HildonStatusBarHelloWorld HildonStatusBarHelloWorld;
typedef struct _HildonStatusBarHelloWorldClass
    HildonStatusBarHelloWorldClass;

#define HILDON_TYPE_STATUS_BAR_HELLOWORLD \
(hildon_status_bar_helloworld_get_type())
#define HILDON_STATUS_BAR_HELLOWORLD(obj) \
    G_TYPE_CHECK_INSTANCE_CAST ((obj), \
```

```

DUMMY_TYPE_STATUS_BAR_HELLOWORLD, HildonStatusBarHelloWorld))
#define HILDON_STATUS_BAR_HELLOWORLD_CLASS(klass) (
    G_TYPE_CHECK_CLASS_CAST ((klass), \
DUMMY_TYPE_STATUS_BAR_HELLOWORLD, HildonStatusBarHelloWorldClass))
#define HILDON_IS_STATUS_BAR_HELLOWORLD(obj) (
    G_TYPE_CHECK_INSTANCE_TYPE ((obj), \
DUMMY_TYPE_STATUS_BAR_HELLOWORLD))
#define HILDON_IS_STATUS_BAR_HELLOWORLD_CLASS(klass) (
    G_TYPE_CHECK_CLASS_TYPE ((klass), \
DUMMY_TYPE_STATUS_BAR_HELLOWORLD))
#define HILDON_STATUS_BAR_HELLOWORLD_GET_CLASS(obj) (
    G_TYPE_INSTANCE_GET_CLASS ((obj), \
DUMMY_TYPE_STATUS_BAR_HELLOWORLD, HildonStatusBarHelloWorldClass))

struct _HildonStatusBarHelloWorld
{
    StatusBarItem parent;
};

struct _HildonStatusBarHelloWorldClass
{
    StatusBarItemClass parent_class;
};

GType hildon_status_bar_helloworld_get_type(void);

typedef struct
{
    osso_context_t      *osso;           /* osso */
    GtkWidget           *icon;           /* icon in button */
    GtkWidget           *button;         /* button in StatusBar */
} HildonStatusBarHelloWorldPrivate;

G_END_DECLS

#endif /* _HILDON_STATUS_BAR_HELLOWORLD_H_ */

```

Status Bar Functions

The most important function is the init function, which is called when the library is opened. If any g_timeouts or such are used, the IDs of these must be stored for removal. The following example illustrates the init function:

```

static void hildon_status_bar_helloworld_init(HildonStatusBarHelloWorld
    *helloworld)
{
    HildonStatusBarHelloWorldPrivate *info =
        HILDON_STATUS_BAR_HELLOWORLD_GET_PRIVATE(helloworld);

    ULOG_OPEN("hildon-sb-helloworld");

    g_return_if_fail(info);

    info->icon = gtk_image_new_from_pixbuf(NULL);
    info->button = gtk_toggle_button_new();

    set_helloworld_icon("hello-world", info);

    gtk_container_add(GTK_CONTAINER(info->button),
        GTK_WIDGET(info->icon));
}

```

```

gtk_container_add(GTK_CONTAINER(helloworld), info->button);

/* Signal for icon (button) */
g_signal_connect(G_OBJECT(info->button), "button-press-event",
                 G_CALLBACK(hello_world_dialog_show), NULL);

/* Initialize osso */
info->osso = osso_initialize("com.nokia.hildon_sb_helloworld", "1.0",
                           FALSE, NULL);
if (!info->osso)
    ULOG_WARN("%s: error while initializing osso\n", __FUNCTION__);

gtk_widget_show_all(GTK_WIDGET(helloworld));
}

```

Destroying Plug-ins

When the item is destroyed, the finalize function is called. The function is defined in class initialization. It should free all the memory and release all the notifications that can lead to the plug-in code. The following example illustrates the finalize function:

```

static void hildon_status_bar_helloworld_class_init(
    HildonStatusBarHelloWorldClass *klass)
{
    GObjectClass *object_class = G_OBJECT_CLASS(klass);
    object_class->finalize = hildon_status_bar_helloworld_finalize;
    g_type_class_add_private(klass, sizeof(
        HildonStatusBarHelloWorldPrivate));
}

...

static void hildon_status_bar_helloworld_finalize(GObject *object)
{
    HildonStatusBarHelloWorldPrivate *info =
        HILDON_STATUS_BAR_HELLOWORLD_GET_PRIVATE(object);

    osso_deinitialize(info->osso);

    LOG_CLOSE();

    G_OBJECT_CLASS(g_type_class_peek_parent(G_OBJECT_GET_CLASS(object)))
        ->finalize(object);
}

```

There is the possibility of avoiding reloading of plug-ins by setting the plug-in as mandatory in the .desktop file, so that it is never unloaded under any circumstances. If so, the plug-in cannot be deselected in control panel navigator applet.

Building Shared Object

The shared object can be built like any normal binary, by only giving the "-shared" flag to gcc:

```

[sbox-DIABLO_X86: ~] > gcc -shared 'pkg-config gtk+-2.0 libosso hildon-1 libhildondesktop
--libs --cflags' hello-world-statusbar.c -o libhelloworld_sb.so

```

The binary produced by gcc must be installed to the path specified in the "osso-af-settings pkg-config" entry. This path can be obtained with

```
pkg-config osso-af-settings --variable=statusbardesktopentrydir
```

By default, it is /usr/share/applications/hildon-status-bar.

6.8.5 Control Panel Plug-ins

Hildon Control Panel follows the same approach with its plug-ins as the other components in the Hildon Desktop environment. Control panel is a standardized place for putting settings that are changeable by the end users, for applications, servers etc. in the system.

The Control Panel is divided into four categories: General, Connectivity, Personalization and Extras.

Desktop File Contents

Each Control Panel applet has to provide a .desktop file in the following format:

```
[Desktop Entry]
Name=<logical applet name identifier>
Comment=Task Navigator Control Panel Applet
Type=default
Icon=<logical icon name identifier>
Categories=<general/connectivity/personalisation/extras>, from which
           the extras is the default value, in case the key value
           read fails.
X-Path=lib<applet name>.so
```

Init Function

Hildon Control Panel calls the init function as defined by the HD_DEFINE_PLUGIN macro when the plug-in is launched.

Control Panel Plug-in Example

Section 6.7 contains a control panel example applet.

6.8.6 Making Plug-in Icons Visible

In order to make the plug-ins icons to show up, the following command must be run:

```
[sbox-DIABLO_X86: ~] >gtk-update-icon-cache -f /usr/share/icons/hicolor
```

where /usr/share/icons/hicolor is the directory containing the icons.

To see the Statusbar and Home plugins, you might need to restart the Desktop, for example by rebooting.

To see the Control Panel plugin, you need to restart the Control panel.

To see the Task Navigator plugin, you need to remove the .osso directory in your home directory so that the configuration files are re-created.

When making a .deb package (see section 6.8.7), the above command should be included in debian/postinst and debian/postrm files so the icon cache is automatically updated when installing or uninstalling the package.

6.8.7 Creating Makefiles and Package for Applet

Before reading this section, it is advisable to familiarize with Reference Guide chapters *GNU build system* 4 and *Packaging, Deploying and Distributing* 13.

Following the instructions in this document, simple Home applets can be made. The applet basically just contains an eventbox whose state can be saved. The building of applets makes use of autotools: `autogen.sh`, `configure.ac` and `Makefile.am`, which are created in addition to the actual applet source code.

The directory structure of the package is:

```
/hello_world_applet
/hello_world_applet/debian
/hello_world_applet/data
/hello_world_applet/src
```

First, the `autogen.sh` script (this script does not contain any applet-specific details):

```
#!/bin/sh
set -x
libtoolize --automake
aclocal-1.7 || aclocal
autoconf
autoheader
automake-1.7 --add-missing --foreign || automake --add-missing --
foreign
```

The `configure.ac` file is as follows:

```
AC_PREREQ(2.59)
% AC_INIT(maemo-hello-world-applet, 0.1, xxxx@maemo.org)

AM_INIT_AUTOMAKE
AM_CONFIG_HEADER(config.h)

dnl #####
dnl This script generates host names
dnl #####
AC_CANONICAL_HOST

dnl #####
dnl Check for installed programs that is needed
dnl #####
AC_PROG_CC
AM_PROG_CC_STDC
AC_PROG_INSTALL
AC_PROG_RANLIB
AC_PROG_INTLTOOL([0.21])
AC_PROG_LIBTOOL

AM_PATH_GLIB_2_0

AC_CHECK_PROG(HAVE_PKG_CONFIG, pkg-config, yes, no)
if test "x$HAVE_PKG_CONFIG" = "xno"; then
AC_MSG_ERROR([You need to install pkg-config tool])
fi

dnl #####
dnl Compiler flags, note that the -ansi flag is not used because
dnl in that case the rint function is not available in math.h
dnl #####
```



```

CFLAGS="$CFLAGS -g -Wall -Werror -ansi -Wmissing-prototypes -Wmissing-
declarations"

dnl #####
dnl Check needed headers, like C standard headers, GLib, GStreamer etc.
dnl #####
AC_HEADER_STDC

GLIB_REQUIRED=2.6.0
GTK_REQUIRED=2.4.0
LIBOSSO_REQUIRED=0.8.3

PKG_CHECK_MODULES(HELLO, [
glib-2.0 >= $GLIB_REQUIRED,
gtk+-2.0 >= $GTK_REQUIRED,
libhildondesktop,
])

AC_SUBST(HELLO_LIBS)
AC_SUBST(HELLO_CFLAGS)

dnl #####
dnl directories
dnl #####

localedir='pkg-config osso-af-settings --variable=localedir'
AC_SUBST(localedir)

pluginlibdir='pkg-config osso-af-settings --variable=
hildondesktoplibdir'
AC_SUBST(pluginlibdir)

AC_CONFIG_FILES([Makefile
src/Makefile
data/Makefile
])

AC_OUTPUT

```

The package config file is used to provide information about the installation directory of the compiled plug-in by querying the value of pluginlibdir variable. All version numbers for various components above should be interpreted as illustrative only.

The master Makefile.am is:

```

SUBDIRS = src data

EXTRA_DIST= \
autogen.sh \
intltool-extract.in \
intltool-merge.in \
intltool-update.in \
debian/rules \
debian/control \
debian/copyright \
debian/changelog \
debian/maemo-hello-world-applet.install

deb_dir = $(top_builddir)/debian-build

INCLUDES = $(DEPS_CFLAGS)

```

```

deb:    dist
      -mkdir $(deb_dir)
      cd $(deb_dir)      &&    tar xzf ../$(top_builddir)/$(PACKAGE)-$(VERSION).
                             tar.gz
      cd $(deb_dir)/$(PACKAGE)-$(VERSION)  &&    dpkg-buildpackage -rfakeroot
      -rm -rf $(deb_dir)/$(PACKAGE)-$(VERSION)

```

Makefile.am of the src subdirectory:

```

pluginlib_LTLIBRARIES = libhello_applet.la

common_CFLAGS = \
$(HELLO_CFLAGS) \
-DPREFIX=\"$(prefix)\" \
-DLOCALEDIR=\"$(localedir)\"

common_LDADD = \
$(HELLO_LIBS)

libhello_applet_la_LDFLAGS= -module -avoid-version

libhello_applet_la_CFLAGS = $(common_CFLAGS)

libhello_applet_la_LIBADD = $(common_LDADD)

libhello_applet_la_SOURCES = \
hello-world-home.h \
hello-world-home.c \
libhelloworld.h \
libhelloworld.c

```

Makefile.am of data directory:

```

homeapplet_desktopdir='pkg-config osso-af-settings --variable=
    homedesktopentrydir'
homeapplet_desktop_DATA=hello-world-applet.desktop

EXTRA_DIST=$(homeapplet_desktop_DATA)

```

The debian/control is as follows:

```

Source: maemo-hello-world-applet
Section: misc
Priority: optional
Maintainer: Mr Maemo <xxxx@maemo.org>
Build-Depends: libgtk2.0-dev (>=2.4.0-1), pkg-config, libhildondesktop-
    dev
Standards-Version: 3.6.1

Package: maemo-hello-world-applet
Section: user/internet
Architecture: any
Depends: ${shlibs:Depends},${launcher:Depends}
Description: Maemo Hello World home panel applet
Home panel applet for showing Hello World.
The debian/changelog is:
maemo-hello-world-applet (0.1) experimental; urgency=low

* Created package

-- Mr Maemo <xxxx@maemo.org>   Tue, 30 May 2006 10:04:45 +0200

```

The debian/rules are:

```

#!/usr/bin/make -f

# export DH_VERBOSE=1

# These are used for cross-compiling and for saving the configure
# script
# from having to guess our platform (since we know it already)
DEB_HOST_GNU_TYPE    ?= $(shell dpkg-architecture -qDEB_HOST_GNU_TYPE)
DEB_BUILD_GNU_TYPE   ?= $(shell dpkg-architecture -qDEB_BUILD_GNU_TYPE)

CFLAGS = -Wall -g
PACKAGE_NAME = maemo-hello-world-applet

ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
CFLAGS += -O0
else
CFLAGS += -O2
endif
ifeq (,$(findstring nostrip,$(DEB_BUILD_OPTIONS)))
INSTALL_PROGRAM += -s
endif

config.status:
dh_testdir
# Add here commands to configure the package.
CFLAGS="$(CFLAGS)" ./configure --host=$(DEB_HOST_GNU_TYPE) --build=$(
DEB_BUILD_GNU_TYPE) \
--prefix=/usr --mandir=\${prefix}/share/man --infodir=\${prefix}/
share/info

build: build-stamp

build-stamp: config.status
dh_testdir

# Add here commands to compile the package.
$(MAKE)

touch build-stamp

clean:
dh_testdir
dh_testroot
rm -f build-stamp configure-stamp

# Add here commands to clean up after the build process.
-$(MAKE) clean

dh_clean

install: build
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs

# Add here commands to install the package
$(MAKE) install DESTDIR=$(CURDIR)/debian/tmp

# Build architecture-independent files here.
binary-indep: build install
# We have nothing to do by default.

```

```

# Build architecture-dependent files here.
binary-arch: build install
dh_testdir
dh_testroot
# dh_installchangelogs
dh_installdocs
# dh_installexamples
dh_install -v --sourcedir=debian/build
# dh_installmenu
# dh_installdebconf
# dh_installogrotate
# dh_installemacsen
# dh_installpam
# dh_installmime
# dh_installinit
# dh_installcron
# dh_installinfo
# dh_installman
dh_link
dh_strip
dh_compress
dh_fixperms
# dh_perl
# dh_python
dh_makeshlibs
dh_installdeb
dh_shlibdeps -V
dh_gencontrol
dh_md5sums
dh_builddeb

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure

```

The debian/maemo-hello-world-applet.install is:

```

usr/lib/hildon-desktop/*.so
usr/share/applications/hildon-home/hello-word-applet.desktop

```

The data/hello-word-applet.desktop is:

```

[Desktop Entry]
Name=Hello, world
Comment=Example hello
Type=default
X-Path=libhello_applet.so

```

6.9 Integrating Applications to Maemo Framework

This section explains how to integrate applications to maemo Application Framework, including adding applications to Task Navigator menu and different features provided by the platform.

6.9.1 Getting Application to Task Navigator Menu

Maemo .desktop File

To make an application visible in maemo Task Navigator, a Desktop file is needed for the application. This file contains all the essential information

needed to show the application entry in the menu, such as name, binary and D-BUS service name. Name of the file should be [application].desktop and location in filesystem "/usr/share/applications/hildon/".

An [example_libosso.desktop](#) file for the application looks like this:

```
[Desktop Entry]
Encoding=UTF-8
Version=1.0
Type=Application
Name=Example libOSSO
Exec=/usr/bin/example_libosso
X-Osso-Service=org.maemo.example_libosso
Icon=qgn_list_gene_default_app
MimeType=application/x-example;
```

The fields in this file are:

Encoding - The character encoding of the file, should be UTF-8.

Version - Version of the application desktop file.

Type - Type of the entry, should be Application.

Name - The name of the application which will be visible to Task Navigator menu.

Exec - The application binary which will be started from the menu entry. Although D-BUS allows launching of applications by service name, this is useful in order to be able to launch also applications that are not registered as D-BUS services.

X-Osso-Service - D-BUS service name of the application. This will be defined in the [application].service file, see next section.

Icon - Name of icon that represents this application. The icon will be visible in Task Navigator before application name. Application can install its own icon, or use one of the pre-defined icons available in platform. Format of the file should be PNG and the suffix (.png) should not be defined.

MimeType - The MIME type(s) supported by this entry.

Other commonly used fields:

StartupWMClass - Name of the application binary name. This field is optional and only needed when the binary name is different from the D-BUS service name. In this case, if this property is not set, application will not be shown in the Application Switcher of the Task Navigator.

X-Window-Icon - Usually the same as Icon field.

X-Window-Icon-Dimmed - Defines whether the dimmed icon should have a different look than the normal icon.

X-Osso-Type - Type for the exec, usually application/x-executable.

Terminal - Defines whether the program runs in a terminal window.

N.B. These fields are not to be followed by a whitespace.

See the [Desktop Entry Specification](#) [14] for more details.

Desktop file itself does not include the location of the application in the Task Navigator menu. So to make an application visible there, also a symbolic link to desktop file is needed in "/etc/others-menu/". This link assigns the order of the entries as well as the possibility that it is not in the root menu, but instead inside some folder. Application should make this link when package is installed.

The link is named by beginning with four numbers defining the order of the entry (the smallest comes on top of the menu) and then underscore (_) followed by the application name. To create this link for the test application:

```
[sbox-DIABLO_X86: /etc/others-menu] > ln -s \  
/usr/share/applications/hildon/example_libosso.desktop \  
/etc/others-menu/0010_example_libosso.desktop
```

To see the new entry in Task Navigator, restart the GUI with "af-sb-init.sh restart". Example LibOSSO entry should be visible to menu.

Now to launch the application, D-BUS service file is still needed. The next section describes how to create it.

D-BUS Service File

D-BUS service file is needed to be able to launch a maemo application and connect it to D-BUS services. One special feature of D-BUS is that if the application is not running already when a D-BUS message is sent to it, D-BUS will start the application automatically. Also only one instance of a D-BUS service can be run at a time, guaranteeing that each application is running only once.

The format of the service file is simpler than desktop file, below is shown what the [example_libosso.service](#) looks like.

```
[D-BUS Service]  
Name=org.maemo.example_libosso  
Exec=/usr/bin/example_libosso
```

The fields in this file are:

Name - The D-BUS service name. This needs to be unique, and same as used in application source codes when initializing the application. Usually the name is built with the application provider URL, followed by the application name to minimize the possibility for conflicts with other service files.

Exec - The full path of the application binary to launch when service name gets called.

D-BUS service file is installed in "/usr/share/dbus-1/services/" and maemo GUI needs to be restarted with "af-sb-init.sh restart" before D-BUS daemon recognizes it.

6.9.2 LibOSSO Library

LibOSSO is the basic library, containing required and helpful functions for maemo applications. The full API documentation of LibOSSO is available in Doxygen format.

Maemo Initialization

All maemo applications need to be initialized correctly, or they will not work as expected. One symptom of missing initialization is that application starts from Task Navigator, but closes automatically after few seconds.

Initializing application is performed with `osso_initialize()` function. With this function, the application connects to D-BUS session bus and system bus. `osso_initialize()` function should only be called once in the application, and the structure of type `osso_context_t` type that is returned should be stored for later use. First parameter of the function is the application D-BUS name, used also in application D-BUS service file. Second parameter is the application version as a string. Third is the activation type, with `TRUE` the library assumes that the application binary has been launched by the D-BUS daemon, and thus will

connect to the D-BUS activation bus. Fourth parameter is the GLib main-loop context to connect to, NULL should be used for the default context.

*osso_context_t *osso_initialize(const gchar *application, const gchar *version, gboolean activation, GMainContext *context)*

When application is closing, *osso_deinitialize* function should be called to close the message bus connection and free all memory allocated by the library.

*void osso_deinitialize(osso_context_t *osso)*

Below is an example ([example_libosso.c](#)) of this; if the initialization does not succeed, the function returns NULL.

```
#define OSSO_EXAMPLE_NAME      "example_libosso"
#define OSSO_EXAMPLE_SERVICE  "org.maemo."OSSO_EXAMPLE_NAME

/* ... */

osso_context_t *osso_context;

/* ... */

/* Initialize maemo application */
osso_context = osso_initialize(OSSO_EXAMPLE_SERVICE, "0.0.1", TRUE,
    NULL);

/* Check that initialization was ok */
if (osso_context == NULL) {
    return OSSO_ERROR;
}

/* ... */

/* Deinitialize OSSO */
osso_deinitialize(osso_context);
```

Remote Process Messages

System wide messages in maemo platform are handled with D-BUS system messaging, which is a Remote Process Communication (RPC) method. LibOSSO has own wrappers to normal D-BUS functions to make usage simpler and to maintain backward compatibility. By using D-BUS, applications can send messages from one process to another.

Callback function receiving the messages can be as follows ([example_libosso.c](#)), where AppData structure contains the initialized *osso_context*:

```
#include <hildon/hildon-program.h>
#include <hildon/hildon-banner.h>

#include <gtk/gtk.h>
#include <libosso.h>

#define OSSO_EXAMPLE_NAME      "example_libosso"
#define OSSO_EXAMPLE_SERVICE  "org.maemo."OSSO_EXAMPLE_NAME
#define OSSO_EXAMPLE_OBJECT    "/org/maemo/"OSSO_EXAMPLE_NAME
#define OSSO_EXAMPLE_IFACE     "org.maemo."OSSO_EXAMPLE_NAME

/* ... */
```

```

/* Application UI data struct */
typedef struct _AppData AppData;
struct _AppData {
    HildonProgram *program;
    HildonWindow *window;
    osso_context_t *osso_context;
};

/* ... */

/* Callback for normal D-BUS messages */
gint dbus_req_handler(const gchar * interface, const gchar * method,
                     GArray * arguments, gpointer data,
                     osso_rpc_t * retval)
{
    AppData *appdata;
    appdata = (AppData *) data;

    osso_system_note_infoprint(appdata->osso_context, method, retval);
    osso_rpc_free_val(retval);

    return OSSO_OK;
}

```

To attach this callback function to receive all the normal D-BUS messages which will come to application, `osso_rpc_set_default_cb_f` function shall be used:

```

int main(int argc, char *argv[])
{
    osso_return_t result;

    /* ... */
    /* Add handler for session bus D-BUS messages */
    result = osso_rpc_set_cb_f(appdata->osso_context,
                              OSSO_EXAMPLE_SERVICE,
                              OSSO_EXAMPLE_OBJECT,
                              OSSO_EXAMPLE_IFACE,
                              dbus_req_handler, appdata);

    if (result != OSSO_OK) {
        g_print("Error setting D-BUS callback (%d)\n", result);
        return OSSO_ERROR;
    }
    /* ... */

    /* Deinitialize OSSO */
    osso_deinitialize(osso_context);
}

```

Now the application is ready to receive D-BUS messages, and whenever it receives one, `dbus_req_handler` function is called to process the message. From another test application ([example_message.c](#)), it is possible to send the "HelloWorld" that this application was designed to handle, like this:

```

/* ... */
#define OSSO_EXAMPLE_NAME      "example_libosso"
#define OSSO_EXAMPLE_SERVICE  "org.maemo."OSSO_EXAMPLE_NAME
#define OSSO_EXAMPLE_OBJECT    "/org/maemo/"OSSO_EXAMPLE_NAME
#define OSSO_EXAMPLE_IFACE     "org.maemo."OSSO_EXAMPLE_NAME
#define OSSO_EXAMPLE_MESSAGE   "HelloWorld"

```



```

/* ... */

ret = osso_rpc_run(osso_context,
                  OSSO_EXAMPLE_SERVICE,
                  OSSO_EXAMPLE_OBJECT,
                  OSSO_EXAMPLE_IFACE,
                  OSSO_EXAMPLE_MESSAGE, &retval, DBUS_TYPE_INVALID);

/* ... */

```

When `example_libosso_test` is started, it sends an "example_message" D-BUS message to `org.maemo.example_libosso` service, attached to the `example_libosso` application (See more about D-BUS service files from an earlier section). Now when `example_libosso` receives the message, it shows a banner.

One nice thing about D-BUS is that the receiving application does not even need to be started: D-BUS can automatically start the application based on its service file, and then pass the message to it!

Hardware State Messages

Maemo applications can connect to listen the system D-BUS messages, like "battery low" and "shutdown". When these messages are received, the application may want to ask the user to save files that are open, or react however wanted.

A callback function is defined like below, taking `osso_hw_state_t` and `gpointer` as parameters. The changed state can be gotten from state variable ([example_libosso.c](#)).

```

/* Callback for hardware D-BUS events */
void hw_event_handler(osso_hw_state_t *state, gpointer data)
{
    AppData *appdata;
    appdata = (AppData *) data;

    if (state->shutdown_ind)
    {
        hildon_banner_show_information(GTK_WIDGET(appdata->window),
                                      NULL,
                                      "Shutdown event!");
    }
    if (state->memory_low_ind)
    {
        hildon_banner_show_information(GTK_WIDGET(appdata->window),
                                      NULL,
                                      "Memory low event!");
    }
    if (state->save_unsaved_data_ind)
    {
        hildon_banner_show_information(GTK_WIDGET(appdata->window),
                                      NULL,
                                      "Must save unsaved data event!");
    }
    if (state->system_inactivity_ind)
    {
        hildon_banner_show_information(GTK_WIDGET(appdata->window),
                                      NULL,
                                      "Minimize application inactivity
                                      event!");
    }
}

```

To attach this handler, e.g. in application main(), use `osso_hw_set_event_cb()`.

```
/* ... */

/* Add handler for hardware D-BUS messages */
result = osso_hw_set_event_cb( appdata->osso_context,
    NULL, hw_event_handler, (gpointer) appdata );

if (result != OSSO_OK)
{
    g_print("Error setting HW state callback (%d)\n", result);
    return OSSO_ERROR;
}

/* ... */
```

N.B. These hardware events are not sent to the SDK, so testing them is only possible in maemo device.

System Exit Message

There is a separate system message apart from the hardware state messages presented earlier, sent when the applications are required to close themselves. Callback for it is like this ([example_libosso.c](#)):

```
/* Callback for exit D-BUS event */
void exit_event_handler(gboolean die_now, gpointer data)
{
    AppData *appdata;
    appdata = (AppData *) data;
    g_print("exit_event_handler called\n");
    /* Do whatever application needs to do before exiting */
    hildon_banner_show_information(GTK_WIDGET(appdata->window), NULL,
        "Exiting...");
}
```

The callback is set (e.g. in application main) as follows.

```
int main( int argc, char* argv[] )
{
    /* ... */
    /* Add handler for Exit D-BUS messages */
    result = osso_application_set_exit_cb(appdata->osso_context,
        exit_event_handler,
        (gpointer) appdata);

    if (result != OSSO_OK) {
        g_print("Error setting exit callback (%d)\n", result);
        return OSSO_ERROR;
    }
    /* ... */
}
```

Now whenever the system needs to close an application for any reason, the close will be performed gracefully.

Application State Saving and Auto Save

State saving is a special feature of the maemo platform. It means that applications save their running state from RAM to permanent memory, such as flash memory, and then end the running application process. This state saving

happens when switching out from the application and the device memory is running low. When the user then returns to a backgrounded application, it is started again, and the application state is read from the flash memory, so that the user will not even know that the application was not running all the time.

Auto saving is an addition to state saving for editor-like applications containing user data. This data/document is automatically saved at the same time with state save. The main difference is that the application state is usually just a small amount of variables describing the state where the application was, whereas the auto-saved file can be any file the user was editing. Auto saving guarantees that user files are not lost when e.g. the battery runs out.

Information on how to implement state saving can be found in LibOSSO API document.

6.9.3 Application Settings

```
#include <gconf/gconf.h>
#include <gconf/gconf-client.h>
```

Maemo uses GConf for maintaining application settings. GConf is widely used in Gnome desktop environment for this same purpose. Read more about GConf from [GConf configuration system](#) [22].

To make compiler and linker find these headers, give additional "pkg-config cflags gconf-2.0" and "pkg-config libs gconf-2.0" parameters for gcc or to Makefile.

GConf needs to be initialized first with `gconf_client_get_default()` function, returning `GconfClient` type of variable. After this, settings of different types can be stored with `gconf_client_set_` functions and restored with `gconf_client_get_` functions.

To keep GConf keys unique in maemo, they should be of type `/apps/maemo/[application]/[key]`, for example: `/apps/maemo/hello_maemo/default_font_size`.

Example ([example_gconf.c](#)) of basic usage for setting and loading GConf values is following:

```
GConfClient *client;

/* ... */

/* Initialize GTK. */
gtk_init(&argc, &argv);

/* Get the default client */
client = gconf_client_get_default();

/*Add GConf node if absent*/
gconf_client_add_dir (client, "/apps/example_prefs",
                     GCONF_CLIENT_PRELOAD_NONE, NULL);

/* ... */

gconf_client_set_string(client, GCONF_KEY, str, NULL );

/* ... */

/* Exit */
g_object_unref (G_OBJECT (client));
```

The maemo version of GConf is unaltered, so all tips and tricks available from Internet are usable also in maemo.

6.10 MIME Types Mapping

MIME types mapping specifies for the platform which application should handle a given MIME type. A mapping has to be defined in the desktop file of the application by adding to it the MimeType field.

An [example_libosso.desktop](#) file for the application looks like the following:

```
[Desktop Entry]
Encoding=UTF-8
Version=1.0
Type=Application
Name=Example libOSSO
Exec=/usr/bin/example_libosso
X-Osso-Service=org.maemo.example_libosso
Icon=qgn_list_gene_default_app
MimeType=application/x-example;
```

The last line is the most important one, and specifies that this application can handle the MIME type "application/x-example".

6.10.1 New MIME Type with OSSO Category Extension

If the application is introducing a new MIME type to the system, it is necessary to provide the mime-info XML (see more at <http://standards.freedesktop.org/shared-mime-info-spec/>) that defines it, in this case an example-mime.xml file for the application looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<mime-info xmlns="http://www.freedesktop.org/standards/shared-mime-info"
  xmlns:osso="http://nokia.com/osso/mime-categories">
  <mime-type type="application/x-example">
    <comment>Example application file</comment>
    <osso:category name="images"/>
    <magic priority="50">
      <match type="string" value="FOO" offset="0"/>
    </magic>
    <glob pattern="*.foo"/>
  </mime-type>
</mime-info>
```

This entry maps one extension and a "magic" string to the "application/x-example" MIME type.

N.B. The glob pattern should be given in lowercase.

6.10.2 What is OSSO Category

The platform has a notion of file categories for the user's data files. The available categories are:

- Bookmarks
- Contacts

- Documents
- Emails
- Images
- Audio
- Video
- Other

A mapping is set up between categories and MIME types, so that the MIME type of a file decides which category it is in. The MIME type setup is handled by the shared-mime-info infrastructure, and the category information is added to that same framework.

Adding a mapping between a category and a number of MIME types is performed much like adding or editing the supported MIME types in the system.

Each application or library that adds a category mapping should add a file in

```
/usr/share/mime/packages/
```

The file format is the same XML format used for MIME types, with an added tag "<osso:category>". See the example above, where it sets up a mapping between .foo files and the Images category.

6.10.3 Updating Platform Databases

To introduce the newly defined MIME type(s) to the platform, it is necessary to:

1. Copy the mime-information XML under /usr/share/mime/packages:

```
[sbox-DIABLO_X86: ~] > cp example-mime.xml /usr/share/mime/packages
```

2. Update the MIME and desktop database:

```
[sbox-DIABLO_X86: ~] > update-mime-database /usr/share/mime
[sbox-DIABLO_X86: ~] > update-desktop-database /usr/share/applications
```

3. Update the OSSO category database:

```
[sbox-DIABLO_X86: ~] > hildon-update-category-database /usr/share/mime
```

If the MIME type is to be removed from the platform, it is done by simply deleting the XML file under /usr/share/mime/packages/, and updating the databases as above.

6.10.4 Registering MIME Type with Package

Since most of the applications are installed on the platform via pre-compiled packages, the MIME type registration has to be performed as well.

The steps are similar to the ones shown above.

To have the MIME information XML installed under /usr/share/mime/packages, the package rules have to be edited and the files installed; in this case it would look as follows:

- in the **rules** file under install section, it is necessary to add the following lines

```
mkdir -p $(CURDIR)/debian/tmp/usr/share/mime/packages
cp $(CURDIR)/example-mime.xml $(CURDIR)/debian/tmp/usr/share/mime/packages
```

- and in .install we need to add

```
usr/share/mime/packages/example-mime.xml
```

This way, it can be assured that the mime information XML is being installed under /usr/share/mime/packages.

The following lines have to be added both into the postinst and postrm files of the package:

```
if [ -x /usr/bin/update-mime-database ]; then
    update-mime-database /usr/share/mime
fi

if [ -x /usr/bin/update-desktop-database ]; then
    update-desktop-database /usr/share/applications
fi

if [ -x /usr/bin/hildon-update-category-database ]; then
    hildon-update-category-database /usr/share/mime
fi
```

This keeps the platform mime information and OSSO category databases up-to-date.

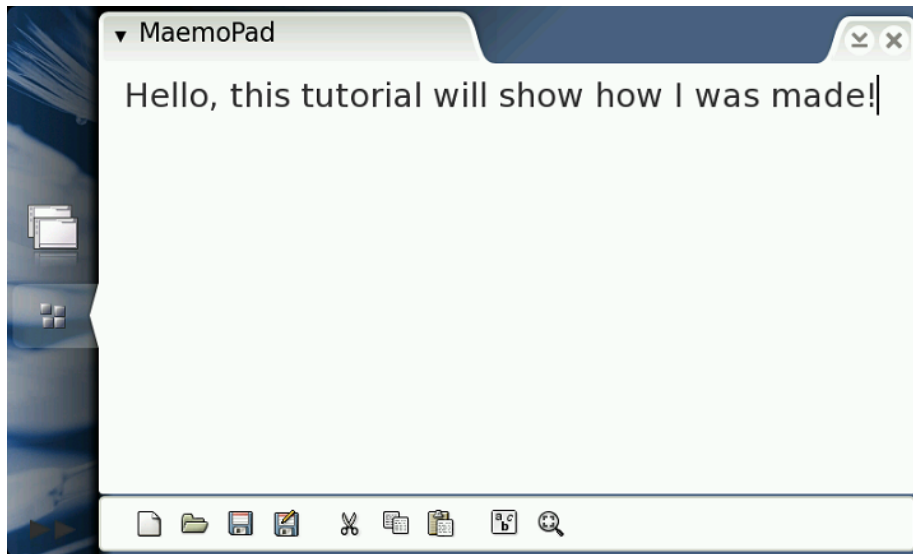
6.11 Writing Application from Scratch

6.11.1 Introduction

This section is a guide for making new applications for the maemo platform. When starting to write a new application, the first phase is to set up the development environment. The actual coding process is described in this section.

As an example, a simple plain text editor is used, with only a few essential features. Good examples for this kind of plain text editors are gtk2edit and gpe-edit. From now on, the application is called "MaemoPad".

MaemoPad will have some basic features, such as "New", "Open", "Save", "Save As...", "Cut", "Copy", "Paste", "Font", "Full Screen", "Full Screen" hardware key handling, "Send-Via email/bluetooth" and "Close". For simplicity, there will be no features like "Undo", "Redo", different fonts in one document, pictures, tables, etc.



The figure [6.11.1](#) is a screenshot from the MaemoPad. As can be seen, MaemoPad application has its own toolbar at the bottom of screen, and a drop-down menu in the upper left-hand corner. The area on the left is used to show the file.

6.11.2 Creating Application File Structure

First, the file structure for MaemoPad has to be created. The structure presented is quite general, and can be used for different kinds of applications on the maemo SDK platform.

The project directory has four subdirectories:

- **src/**: Contains the source files. In the `src/` directory, `main.c` is in the top of the hierarchy in the MaemoPad code. The file `appdata.h` has the data needed by the application. Writing the code in `main.c` and `appdata.h` is explained in the next subsection [6.11.3](#). The file `interface.c` in the `src/ui/` directory takes care of creating the graphical user interface ([6.11.4](#)). `callbacks.c` ([6.11.4](#)) contains the code taking care of the messages between MaemoPad and maemo platform.
- **debian/**: Contains the files related to debian packaging ([13.1](#)).
- **data/**: Contains `.desktop` file ([6.11.6](#)), a D-Bus `.service` file ([6.9.1](#)), and data files to be used when running the application, such as icons and help files.
- **po/**: Contains the localization files. The localization directory `po/` has a file `POTFILES.in`, listing the names of the files to be localized, and a translation file `en_GB.po` containing British English strings for the application, see section [6.11.5](#) for more information.

Apart from the four subdirectories, the project main directory includes only three script files, using GNU `autoconf` and `automake` tools to configure and compile the project:

- *autogen.sh* is a tiny script file to make other script files to build the project.
- *configure.ac* includes definitions on how to produce the configure script for the project.
- *Makefile.am* includes the files and subdirectories needed to make the application: *src/*, *po/* and *data/* and all the Makefiles in *src/*, *po/*, and *data/* directories.

The way used to compile a project is usually

```
$ ./autogen.sh && ./configure && make;
```

For more information about GNU autoconf and automake, see chapter *GNU build system 4* on Maemo Reference Manual.

The file structure of the MaemoPad application looks like this:

```
Makefile.am
autogen.sh
configure.ac
src/
    Makefile.am
    appdata.h
    main.c
    ui/
        callbacks.h
        callbacks.c
        interface.h
        interface.c
data/
    Makefile.am
    com.nokia.maemopad.service
    maemopad.desktop
    icons/
        26x26/
            maemopad.png
        40x40/
            maemopad.png
        scalable/
            maemopad.png
    help/
        en_GB/
            MaemoPad.xml
po/
    Makefile.in.in
    POTFILES.in
    en_GB.po
debian/
    changelog
    control
    copyright
    rules
```

6.11.3 Coding Application Main

In the *src/* directory of MaemoPad, there are *main.c* and *appdata.h*.

appdata.h

appdata.h defines *AppData* structure to hold the application data. Even though the data varies depending on the application, a sample *AppData* struct might look like this:

```
struct _AppData
{
```



```

AppUIData *ui; /* handle to app's UI */
HildonProgram *program; /* handle to application */
osso_context_t *osso; /* handle to osso */
AppConfData *conf; /*handle to app's Gconf data */
};

```

Where

- AppUIData is a struct containing pointers to all the UI objects, such as HildonWindow, menu items, toolbar; and UI-related data, for instance a boolean variable indicating whether the application is in fullscreen mode.
- HildonProgram is explained above (6.4.3). It can, alternatively, be declared inside AppUIData struct.
- osso_context_t is explained above (6.9.2).
- AppConfData is a struct that contains Gconf-related data of the applications.

Each application will create its own AppData variable, and this variable is usually passed around as a parameter in the functions, particularly the callback functions, so that the applications data can be accessible by the functions.

Many applications declare this AppData variable as global.

MaemoPad's AppData struct is as follows:

```

struct _AppData
{
    HildonProgram *program; /* handle to application */
    HildonWindow *window; /* handle to app's window */
    osso_context_t *osso; /* handle to osso */
};

```

N.B. AppUIData in MaemoPad points to AppData, instead of AppData pointing to AppUIData. This is not a significant difference, since the ultimate purpose is to have application data accessible by the functions.

```

typedef struct _AppUIData AppUIData;
struct _AppUIData
{
    /* Handle to app's data */
    AppData *data;

    /* Fullscreen mode is on (TRUE) or off (FALSE) */
    gboolean fullscreen;

    /* Items for menu */
    GtkWidget *file_item;
    GtkWidget *new_item;
    GtkWidget *open_item;
    GtkWidget *save_item;
    GtkWidget *saveas_item;
    GtkWidget *edit_item;
    GtkWidget *cut_item;
    GtkWidget *copy_item;
    GtkWidget *paste_item;
    /*.....more truncated .....*/
}

```

main.c

main.c usually performs, at least, the following functions:

- Initializing GTK
- Initializing localization
- Creating an instance of HildonProgram
- Calling to a function, usually defined in interface.c, to create the main view.
- Connecting the main view window to the created HildonProgram
- Running gtk_main()
- Connecting the "delete_event" of the main view to a callback function to handle a proper application exit, such as destroying the main view window, freeing used memory and saving application states before exit.
- Call gtk_main_quit()

Here is the main function of MaemoPad with comments:

```
int main( int argc, char* argv[] )
{
    AppData* data;
    HildonProgram* program;
    AppUIData* main_view;

    /* Initialize the locale stuff */
    setlocale ( LC_ALL, "" );
    bindtextdomain ( GETTEXT_PACKAGE, LOCALEDIR );
    bind_textdomain_codeset(GETTEXT_PACKAGE, "UTF-8");
    textdomain ( GETTEXT_PACKAGE );

    /* Init the gtk - must be called before any hildon stuff */
    gtk_init( &argc, &argv );

    /* Create the hildon application and setup the title */
    program = HILDON_PROGRAM ( hildon_program_get_instance ( ) );
    g_set_application_name ( _("MaemoPad") );

    if(!g_thread_supported()) {
        g_thread_init(NULL);
    }

    /* Create the data and views for our application */
    data = create_data ();
    data->program = program;
    main_view = interface_main_view_new ( data );
    hildon_program_add_window( data->program, data->window );

    g_signal_connect( G_OBJECT(data->window), "delete_event",
        gtk_main_quit, NULL );
    /* Connect key presses signals */
    g_signal_connect(G_OBJECT(data->window), "key_press_event",
        G_CALLBACK(key_press), (gpointer) main_view);
    g_signal_connect(G_OBJECT(data->window), "key_release_event",
        G_CALLBACK(key_release), (gpointer) main_view);
}
```

```

    /* Begin the main app */
    gtk_widget_show ( GTK_WIDGET ( data->window ) );
    gtk_main();

    /* Clean up */
    interface_main_view_destroy ( main_view );
    destroy_data ( data );

    return 0;
}

```

6.11.4 User Interface

The graphical user interface code is implemented in directory `./src/ui/`. There are two `.c` files: `interface.c` and `callbacks.c`

`interface.c`

This file will create the graphical user interface (GUI) and connect the signals and events to appropriate handlers defined in `callbacks.c`. See section 6.4 for information on how to create GUI in maemo. If GTK is not familiar, it is advisable to start from the GTK+ Reference Manual [57].

As a general practice, an `AppUIData` struct variable is created when creating the GUI. And then, a `HildonWindow` and smaller components are created in different functions, such as `create_menu()`, `create_toolbar()`. When creating each component, `AppUIData` should refer various necessary UI objects created along the way.

The following excerpt shows how `AppUIData` is created, and how it points to the toolbar and the button "New file" on the toolbar.

```

/* Creates and initializes a main_view */
AppUIData* interface_main_view_new( AppData *data )
{
    /* Zero memory with g_new0 */
    AppUIData* result = g_new0( AppUIData, 1 );
    /*....*/
    create_toolbar( result );
    /*....*/
}

/* Create toolbar to mainview */
static void create_toolbar ( AppUIData *main )
{
    /* Create new GTK toolbar */
    main->toolbar = gtk_toolbar_new ();
    /*....*/
    /* Create the "New file" button in the toolbar */
    main->new_tb = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);
    /*....*/
}

```

`callbacks.c`

`callbacks.c` defines all the functions that handle the signals or events that might be triggered by the UI. When creating different UI objects in `interface.c`, the

handlers are registered as follows.

```
/* Create the menu items needed for the drop down menu */
static void create_menu( AppUIData *main )
{
    /* ... */

    main->new_item = gtk_menu_item_new_with_label ( _("New") );

    /* ... */

    /* Attach the callback functions to the activate signal */
    g_signal_connect( G_OBJECT( main->new_item ), "activate",
                      G_CALLBACK ( callback_file_new), main );

    /* ... */
}
```

Function `callback_file_new` is implemented in `callbacks.c`, saving the current file if needed, and then opening a new file to edit.

```
void callback_file_new(GtkAction * action, gpointer data)
{
    gint answer;
    AppUIData *mainview = NULL;
    mainview = ( AppUIData * ) data;
    g_assert(mainview != NULL && mainview->data != NULL );

    /* save changes note if file is edited */
    if( mainview->file_edited ) {
        answer = interface_save_changes_note( mainview );
        if( answer == CONFRESP_YES ) {
            if( mainview->file_name == NULL ) {
                mainview->file_name = interface_file_chooser(mainview,
                    GTK_FILE_CHOOSER_ACTION_SAVE);
            }
            write_buffer_to_file ( mainview );
        }
    }

    /* clear buffer, filename and free buffer text */
    gtk_text_buffer_set_text ( GTK_TEXT_BUFFER (mainview->buffer), "",
        -1 );
    mainview->file_name = NULL;
    mainview->file_edited = FALSE;
}
```

N.B. The `AppUIData` struct variable `mainview` is retrieved in such a way that the handlers can have a direct effect on the UI for the users.

There are many other functions in `MaemoPad`, explained above:

- File Save/Save-As/Open: This uses `HildonFileChooserDialog` ([6.5.1](#))
- Edit Cut/Copy/Paste: This uses `Clipboard` ([7.8](#))
- Hardware keys: `MaemoPad` can recognize the "Fullscreen" button key presses (F6). It will switch application from fullscreen mode to normal mode, and vice versa. Many other hardware key events can be added to `MaemoPad`, see section [6.6.3](#).
- Font/Color Selector: These are explained in `HildonFontSelectionDialog` ([6.5.3](#)) and `HildonColorChooser` ([6.5.2](#))

- Send via Email/Bluetooth: Refer to section “Send via” functionality 7.10 on chapter *Using Generic Platform Components* of Maemo Reference Manual for instructions on how to implement Send via functionality.

More information on GTK Widgets can be found on the [GTK+ Reference Manual](#).

interface.h

In the interface header file interface.h, public functions are defined for main.c and callbacks.c. In the case of MaemoPad, confirmation responses for the save changes note, MaemopadError enum for the Hildon error note and the AppUIData are also defined here. **N.B.** AppUIData can also be defined in appdata.h in some other applications. MaemoPad’s interface.h looks like this:

```
#define MAIN_VIEW_NAME "AppUIData"

typedef enum {
    MAEMOPAD_NO_ERROR = 0,
    MAEMOPAD_ERROR_INVALID_URI,
    MAEMOPAD_ERROR_SAVE_FAILED,
    MAEMOPAD_ERROR_OPEN_FAILED
} MaemopadError;

/* Struct to include view's information */
typedef struct _AppUIData AppUIData;
struct _AppUIData
{
    /* Handle to app's data */
    AppData *data;

    /* Fullscreen mode is on (TRUE) or off (FALSE) */
    gboolean fullscreen;

    /* Items for menu */
    GtkWidget *file_item;
    GtkWidget *new_item;

    /* ... */

    GtkWidget *font_item;
    GtkWidget *fullscreen_item;

    /* Toolbar */
    GtkWidget* toolbar;
    GtkWidget* iconw;
    GtkToolItem* new_tb;
    GtkToolItem* open_tb;

    /* ... */

    /* Textview related */
    GtkWidget* scrolledwindow; /* textview is under this widget */
    GtkWidget* textview;       /* widget that shows the text */
    GtkTextBuffer* buffer;      /* buffer that contains the text */
    GtkClipboard* clipboard;    /* clipboard for copy/paste */

    PangoFontDescription* font_desc; /* font used in textview */

    gboolean file_edited; /* tells is our file on view edited */
}
```

```

gchar* file_name;          /* directory/file under editing */
};

/* Public functions: */
AppUIData* interface_main_view_new( AppData* data );
void interface_main_view_destroy( AppUIData* main );
char* interface_file_chooser( AppUIData* main, GtkFileChooserAction
    action );
PangoFontDescription* interface_font_chooser( AppUIData * main );

/* ... */

```

6.11.5 Localization

Localization means translating the application into different languages. In maemo, this is fairly easily performed by grouping all the strings needing translations into a .po file, giving them each an id, and then using the id in the code instead of hard-coded strings. The function used to generate the translated strings from an id in maemo is the standard GNU gettext().

Initialization

When the application runs, depending on the locale settings of the system, gettext() will translate the id into the correct language. The application will have to initialize the text domain as follows:

```

int main( int argc, char* argv[] )
{
    /* ... */

    /* Initialize the locale stuff */
    setlocale ( LC_ALL, "" );
    bindtextdomain ( GETTEXT_PACKAGE, LOCALEDIR );
    bind_textdomain_codeset( GETTEXT_PACKAGE, "UTF-8" );
    textdomain ( GETTEXT_PACKAGE );

    /* ... */
}

```

More information on localization can be found in [section 12.3](#).

File Structure

Localization files are stored in the po/ directory. The following files will be used for MaemoPad's localization:

```

Makefile.am
POTFILES.in
en_GB.po

```

POTFILES.in contains the list of source code files that will be localized. In MaemoPad, only main.c and interface.c contain strings that need to be localized.

```

# List of MaemoPad source files to be localized

../src/main.c
../src/ui/interface.c

```

File en_GB.po includes translated text for British English. It contains pairs of id/string as follows:

```
# ...
msgid "maemopad_yes"
msgstr "Yes"
# ...
```

N.B. The comments in .po file start with "#".

Using en_GB.po

The msgid(s) are passed to the GNU gettext() function as a parameter to generate the translated string. In maemo, the recommended way is

```
#define _(String) gettext(String)
```

Therefore, in MaemoPad, the string for Menu->File->Open menu will be created as follows:

```
main->open_item = gtk_menu_item_new_with_label ( _("Open") );
```

Creating .po Files from Source

Sometimes it is necessary to localize code for applications, where localization was not considered at the beginning. It is possible to create .po files from source code using [GNU xgettext](#) [26], by extracting all the strings from the source files into a template.po file

```
xgettext -f POTFILES.in -C -a -o template.po
```

Read the man page for xgettext for more information, in short:

- "-f POTFILES.in" uses POTFILES.in to get the files to be localized
- "-C" is for C-code type of strings
- "-a" is for ensuring that we get all strings from specified files
- "-o template.po" defines the output filename.

The next step is to copy this template.po into ./po/en_GB.po and add or edit all the strings in British English. Other languages can be handled in the same way.

6.11.6 Adding Application to Menu

See section [6.9.1](#) for information on how to perform this. In short, the maemopad.desktop and com.nokia.maemopad.service files are stored in the ./data directory, and they look like this, respectively

```
[Desktop Entry]
Encoding=UTF-8
Version=0.1
Type=Application
Name=MaemoPad
Exec=/usr/bin/maemopad
```

```
Icon=maemopad
X-Window-Icon=maemopad
X-Window-Icon-Dimmed=maemopad
X-Osso-Service=com.nokia.maemopad
X-Osso-Type=application/x-executable
```

(N.B. Whitespace is not allowed after the lines.)

```
# Service description file
[D-BUS Service]
Name=com.nokia.maemopad
Exec=/usr/bin/maemopad
```

6.11.7 Link to Maemo Menu

When the Debian package is installed to maemo platform, .desktop and .service files are used to link MaemoPad to the Task Navigator.

6.11.8 Adding Help

Applications can have their own help files. Help files are XML files located under the /usr/share/osso-help directory. Each language is located in its own subdirectory, such as /usr/share/osso-help/en_GB for British English help files. In MaemoPad, there is data/help/en_GB/MaemoPad.xml, having very simple help content. It is mandatory that the middle part of the contextUID is the same as the help file name (without suffix):

```
<?xml version="1.0" encoding="UTF-8"?>
<hildonhelpsource>
  <folder>
    <title>MaemoPad</title>
    <topic>
      <topicitle>Main Topic</topicitle>
      <context contextUID="Example_MaemoPad_Content" />
      <para>This is a help file with example content.</para>
    </topic>
  </folder>
</hildonhelpsource>
```

By using hildon_help_show() function (see hildon-help.h), this help content can be shown in the application. After creating a Help menu item, a callback function can be connected to it:

```
void callback_help( GtkAction * action, gpointer data )
{
    osso_return_t retval;

    /* connect pointer to our AppUIData struct */
    AppUIData *mainview = NULL;
    mainview = ( AppUIData * ) data;
    g_assert(mainview != NULL && mainview->data != NULL );

    retval = hildon_help_show(
        mainview->data->osso, /* osso_context */
        HELP_TOPIC_ID,       /* topic id */
        HILDON_HELP_SHOW_DIALOG);
}
```

To get more information, see section 6.12.

6.11.9 Packaging Application

A Debian package is an application packed in one file to make installing easy in Debian-based operating systems, like maemo platform. More information about creating a Debian packages can be found in section *Creating Debian packages* 13.1 of Maemo Reference Manual Chapter *Packaging, Deploying and Distributing*. Our goal in this section is to create a Debian package of MaemoPad, to be installed in the maemo platform.

If creating a package that can be installed using the Application Manager, see section *Making Application Packages* 13.2 of the same aforementioned chapter.

Creating debian/ Directory

Some files are needed to create the package. They are placed under the debian/ directory. These files will be created:

```
changelog
control
copyright
rules
... etc ...
```

The 'rules' file is the file defining how the Debian package is built. The 'rules' file tells where the files should be installed. Also a 'control' file is needed to define what kind of packages (often different language versions) are going to be created. Changelog and copyright files are also needed, or building the package will not work. The 'changelog' file consists of the version number of the package, and a short description about changes compared to older versions. The 'copyright' file includes information in plain text about the package copyrights.

Most important lines in rules file are:

```
# Add here commands to install the package into debian/<installation
directory>
$(MAKE) install DESTDIR=$(CURDIR)/debian/<installation directory>
```

These lines define, where the package files will be installed. debian/<installation directory> is used as a temporary directory for package construction.

Creating and Building Package

The package is made using the following command:

```
dpkg-buildpackage -rfakeroot -uc -us -sa -D
```

The result should be these MaemoPad files:

- maemopad_x.x.dsc
- maemopad_x.x.tar.gz
- maemopad_x.x_i386.changes
- maemopad_x.x_i386.deb

There is now a .deb file. This package can be installed using "fakeroot dpkg -i maemopad_x.x_i386.deb" command. The icon to the application should now be in the maemo Task Navigator menu, and it should be launchable from there. The package can be removed with the command "fakeroot dpkg -r maemopad".

6.12 Help Framework

Various services are available for softwares developed on the maemo platform. One of them is the Help Framework, a context-based help system that can be easily used in any application.

The Help Framework is a centralized way to offer help services to the users of the programs. Maemo platform has an built-in help system that handles all the help documentation for the programs using the Help Framework. Libraries are used to register a program to the Help Framework, and after that, the content of the actual help documentation can be written. An ID tag will be given to the help file, which will be in XML format. This way, it is possible to control which help file will be loaded when the user asks for help: it is only necessary to call the correct help content ID. When using the Help Framework, the help documentation for a program will also be available when using the maemo platform help application.

This section is a short guide for using the Help Framework. It focuses on the explanation of the various steps necessary to enable this feature in an application, by going through a small [example](#) of using the framework. The section also explains the XML format used in the actual writing of the help content.

6.12.1 Creating Help File

The help file is an XML document, with various pre-defined tags, used to aid the creation of help topics. The example below shows the format this file must obey:

```
<?xml version="1.0" encoding="UTF-8"?>
<ossohelpsource>
  <folder>
    <title>Help Framework Example</title>
    <topic>
      <topicitle>Main Topic</topicitle>
      <context contextUID="osso_example_help" />
      <para>This is a help file with example content.</para>
    </topic>
  </folder>
</ossohelpsource>
```

The `<context />` tag is used to define the topic "context", which is needed for referring the document by the help framework. The context string is defined by the attribute `contextUID` and must follow the format `xxx_filename_yyy`, where filename must be exactly the name of the XML file (without the extension).

In the example above, `contextUID osso_example_help` was created, and it will be used in the example application.

In addition to the `<context />` tag, there are various other tags. Below is a list of the tags with a short description:

- **folder** Contains help topics, one or more.
- **title** Title text for the folder to be shown in the Contents list of the Help application. Usually it is the name of the application.

- **topic** Single help content to be shown for the user. Every topic can be referred to with its unique contextUID.
- **topictitle** Title text for the topic to be shown under the main title in the folder list.
- **heading** Can be used in long help pages to separate different subsections. N.B. Not all the topics require a heading, because the **topictitle** is used as the main heading of a topic. Headings are formatted with a different color, and a larger and bolder font.
- **display_text** Used for presenting text that appears on the device display, like button labels etc. Display text is formatted with a different color in the Help application.
- **graphic** Used to show in-line images among the text. To use them, either the absolute path (with 'filename' parameter), or the plain filename has to be defined. The former is used to show third-party pictures, while the latter is used to show system images.
- **task_seq** Used for task sequence ordered lists.
- **step** Used to define a step in a task sequence ordered list.
- **list** Used for unordered lists.
- **listitem** Used for an item in an unordered list.
- **para** A general paragraph.
- **tip** Shows a "Tip:" prefix.
- **note** Shows a "Note:" prefix.
- **important** Shows an "Important:" prefix.
- **example** Shows an "Example:" prefix.
- **warning** Shows a "Warning:" prefix.
- **ref** Used to create a cross-reference: '*refid*' parameter defines the contextUID of the referred topic, and '*refdoc*' parameter is the name of the reference.
- **emphasis** Used to emphasize the text.
- **ossohelpsource** Main XML tag for the help files.
- **change_history** Change history of the help content. Used only by help content authors (see change).
- **change** Entry on change_history.
- **synonyms** Synonyms are only used by the Global search, and are not visible to the users. Synonyms can be used as extra keywords for search.

Also common HTML tags can be used, such as `` for bold text and `<i>` for italics.

The example XML file can be downloaded from [help-framework downloads](#).

6.12.2 Adding Help Context Support into Application

This section will demonstrate, how to create an application with an online help system. The application will use the XML file described in the previous section.

To use the online help system on the maemo platform, it is necessary to include the following header files into the application:

```
#include <libosso.h>
#include <hildon/hildon-help.h>
```

The header file `libosso.h` provides the environment initialization functions of the maemo platform, which are responsible for registering the context used by the application for the help framework.

The `hildon/hildon-help.h` file contains prototypes for functions in the library `hildon-help`. These are used for locating the help topics and displaying these topics to the user.

Next, after including these files into the source code, a couple of pre-compilation macros are defined to help the build in the future. It is also important to define a global structure for the application to save `HildonProgram`, `HildonWindow`, and `osso_context_t`, which are, respectively, application object, main window, and application context object of the program.

```
#define OSSO_HELP_TOPIC_EXAMPLE "osso_example_help"

/* Application UI data struct */
typedef struct {
    HildonProgram *program;
    HildonWindow *window;
    osso_context_t *osso_context;
} AppData;

AppData appdata;
```

To start the application, two steps are necessary:

- Initialize the GTK+ library by calling the function `gtk_init()`.
- Initialize the application context using `osso_initialize()`.

The following example shows in detail, how this initialization must be performed:

```
#define OSSO_EXAMPLE_NAME "example_help_framework"
#define OSSO_EXAMPLE_SERVICE "org.maemo."OSSO_EXAMPLE_NAME

/* ... */

/* Initialize the GTK. */
gtk_init(&argc, &argv);

/* Initialize maemo application */
appdata.osso_context = osso_initialize(OSSO_EXAMPLE_SERVICE, "0.0.1",
    TRUE, NULL);

/* Check that initialization was ok */
if (appdata.osso_context == NULL) {
    return OSSO_ERROR;
}
```

Now the hildon help system can be used. To show one of the topics from the help file, the function `hildon_help_show()` has to be called, as shown in the example.

In case the topic is not available, the function `hildon_help_show()` will return the error code `OSSO_ERROR`.

```
/* handler for Help button */
void help_activated(GtkWidget *win, gchar *help_id)
{
    osso_return_t retval;

    if (!help_id) {
        return;
    }

    retval = hildon_help_show(
        appdata.osso_context, /* global osso_context */
        help_id,              /* topic id */
        HILDON_HELP_SHOW_DIALOG);
}
```

To create a button to activate help, the following code should be added into the main function:

```
/* Add a Help button */
help_button = gtk_button_new_with_label("Help");
g_signal_connect(G_OBJECT(help_button), "clicked",
                 G_CALLBACK(help_activated),
                 OSSO_HELP_TOPIC_EXAMPLE);
g_signal_connect(G_OBJECT(help_button), "clicked",
                 G_CALLBACK(help_activated),
                 OSSO_HELP_TOPIC_EXAMPLE);
```

The next step is to add a help button into a dialog box (symbol '?' on the upper right-hand corner of the dialog box). The process is quite simple: the button is enabled in the dialog box with the function `hildon_help_dialog_help_enable()`. The example below shows how this is done.

```
void button_clicked(GtkWidget *widget, gpointer data) {
    GtkWidget *dialog;
    guint result;

    dialog = hildon_color_chooser_dialog_new();

    /* enable help system on dialog */
    hildon_help_dialog_help_enable(
        GTK_DIALOG(dialog),
        OSSO_HELP_TOPIC_EXAMPLE,
        appdata.osso_context
    );

    result = gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
    return;
}
```

Now the only thing remaining is to compile the example:

```
gcc 'pkg-config --cflags --libs hildon-1 libosso hildon-help' \
-o example_help_framework example_help_framework.c
```

It is possible to test the help system also in Scratchbox.

6.12.3 Distributing Example Application

To quickly test the help content, the XML file can be manually copied to the correct path (e.g. `/usr/share/osso-help/en_GB/`), where it can be found by the help library. The directories may have to be created manually, if testing in Scratchbox. If the help file is written for some other language, `en_GB` in the path should be replaced with the correct language definition. After starting the newly-created application (or Help application), the help content should be available for viewing. The easiest way to distribute the help file and the application is to create an Application Manager package. For more details, please see section *Making Application Packages* [13.2](#) in chapter *Packaging, Deploying and Distributing* of Maemo Reference Manual.

Chapter 7

Using Generic Platform Components

7.1 Introduction

The following code examples are used in this chapter:

- [hildon_helloworld-8.c](#)
- [libdbus-example](#)
- [libosso-example-sync](#)
- [libosso-example-async](#)
- [libosso-flashlight](#)
- [glib-dbus-sync](#)
- [glib-dbus-signals](#)
- [glib-dbus-async](#)
- [hildon_helloworld-9.c](#)
- [gconf-listener](#)
- [example_alarm.c](#)
- [example_abook.c](#)
- [MaemoPad](#)
- [Certificate Manager Examples](#)
- [hildon-input-method-plugins-example](#)

The underlying system services in the maemo platform differ slightly from those used in desktop Linux distributions. This chapter gives an overview of the most important system services.

7.2 File System - GnomeVFS

Maemo includes a powerful file system framework, GnomeVFS. This framework enables applications to use a vast number of different file access protocols without having to know anything about the underlying details. Some examples of the supported protocols are: local file system, HTTP, FTP and OBEX over Bluetooth.

In practice, this means that all GnomeVFS file access methods are transparently available for both developer and end user just by using the framework for file operations. The API for file handling is also much more flexible than the standard platform offerings. It features, for example, asynchronous reading and writing, MIME type support and file monitoring.

All user-file access should be done with GnomeVFS in maemo applications, because file access can be remote. In fact, many applications that come with the operating system on the Internet tablets do make use of GnomeVFS. Access to files not visible to the user should be done directly for performance reasons.

A good hands-on starting point is taking a look at the GnomeVFS example in maemo-examples package. Detailed API information can be found in the GnomeVFS API reference[\[31\]](#).

GnomeVFS Example

In maemo, GnomeVFS also provides some filename case insensitivity support, so that the end users do not have to care about the UNIX filename conventions, which are case-sensitive.

The GnomeVFS interface attempts to provide a POSIX-like interface, so that when one would use `open()` with POSIX, `gnome_vfs_open` can be used instead. Instead of `write()`, there is `gnome_vfs_write`, etc. (for most functions). The GnomeVFS function names are sometimes a bit more verbose, but otherwise they attempt to implement the basic API. Some POSIX functions, such as `mmap()`, are impossible to implement in the user space, but normally this is not a big problem. Also some functions will fail to work properly over network connections and outside the local filesystem, since they might not always make sense there.

Shortly there will follow a simple example of using the GnomeVFS interface functions.

In order to save and load data, at least the following functions are needed:

- **gnome_vfs_init()**: initializes the GnomeVFS library. Needs to be done once at an early stage at program startup.
- **gnome_vfs_shutdown()**: frees up resources inside the library and closes it down.
- **gnome_vfs_open()**: opens the given URI (explained below) and returns a file handle for that if successful.
- **gnome_vfs_get_file_info()**: get information about a file (similar to, but with broader scope than `fstat`).
- **gnome_vfs_read()**: read data from an opened file.
- **gnome_vfs_write()**: write data into an opened file.

In order to differentiate between different protocols, GnomeVFS uses Uniform Resource Location syntax when accessing resources. For example in **file:///tmp/somefile.txt**, the **file://** is the protocol to use, and the rest is the location within that protocol space for the resource or file to manipulate. Protocols can be stacked inside a single URI, and the URI also supports username and password combinations (these are best demonstrated in the GnomeVFS API documentation).

The following simple demonstration will be using local files.

A simple application will be extended in the following ways:

- Implement the "Open" command by using GnomeVFS with full error checking.
- The memory will be allocated and freed with `g_malloc0()` and `g_free()`, when loading the contents of the file that the user has selected.
- Data loaded through "Open" will replace the text in the GtkLabel that is in the center area of the HildonWindow. The label will be switched to support Pango simple text [markup](#), which looks a lot like simple HTML.
- Notification about loading success and failures will be communicated to the user by using a widget called HildonBanner, which will float a small notification dialog (with an optional icon) in the top-right corner for a while, without blocking the application.
- N.B. Saving into a file is not implemented in this code, as it is a lab exercise (and it is simpler than opening).
- File loading failures can be simulated by attempting to load an empty file. Since empty files are not wanted, the code will turn this into an error as well. If there is no empty file available, one can easily be created with the touch command (under **MyDocs**, so that the open dialog can find it). It is also possible to attempt to load a file larger than 100 KiB, since the code limits the file size (artificially), and will refuse to load large files.
- The goto statement should normally be avoided. Team coding guidelines should be checked to see, whether this is an allowed practice. Note how it is used in this example to cut down the possibility of leaked resources (and typing). Another option for this would be using variable finalizers, but not many people know how to use them, or even that they exist. They are gcc extensions into the C language, and you can find more about them by reading gcc info pages (look for variable attributes).
- Simple GnomeVFS functions are used here. They are all synchronous, which means that if loading the file takes a long time, the application will remain unresponsive during that time. For small files residing in local storage, this is a risk that is taken knowingly. Synchronous API should not be used when loading files over network, since there are more uncertainties in those cases.
- I/O in most cases will be slightly slower than using a controlled approach with POSIX I/O API (controlled meaning that one should know what to use and how). This is a price that has to be paid in order to enable easy switching to other protocols later.

N.B. Since GnomeVFS is a separate library from GLib, you will have to add the flags and library options that it requires. The pkg-config package name for the library is `gnome-vfs-2.0`.

```
/**
 * hildon_helloworld-8.c
 *
 * 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.
 *
 * We add file loading support using GnomeVFS. Saving files using
 * GnomeVFS is left as an exercise. We also add a small notification
 * widget (HildonBanner).
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
 */

#include <stdlib.h>
#include <hildon/hildon-program.h>
#include <hildon/hildon-color-button.h>
#include <hildon/hildon-find-toolbar.h>
#include <hildon/hildon-file-chooser-dialog.h>
/* A small notification window widget (NEW). */
#include <hildon/hildon-banner.h>
/* Pull in the GnomeVFS headers (NEW). */
#include <libgnomevfs/gnome-vfs.h>

/* Declare the two slant styles. */
enum {
    STYLE_SLANT_NORMAL = 0,
    STYLE_SLANT_ITALIC
};

/**
 * The application state.
 */
typedef struct {
    gboolean styleUseUnderline;
    gboolean styleSlant;

    GdkColor currentColor;

    /* Pointer to the label so that we can modify its contents when a
     * file is loaded by the user (NEW). */
    GtkWidget* textLabel;

    gboolean fullScreen;

    GtkWidget* findToolbar;
    GtkWidget* mainToolbar;
    gboolean findToolbarIsVisible;
    gboolean mainToolbarIsVisible;

    HildonProgram* program;
    HildonWindow* window;
} ApplicationState;

/*... Listing cut for brevity ...*/
/**
```

```

* Utility function to print a GnomeVFS I/O related error message to
* standard error (not seen by the user in graphical mode) (NEW).
*/
static void dbgFileError(GnomeVFSResult errCode, const gchar* uri) {
    g_printerr("Error while accessing '%s': %s\n", uri,
               gnome_vfs_result_to_string(errCode));
}

/**
 * MODIFIED (A LOT)
 *
 * We read in the file selected by the user if possible and set the
 * contents of the file as the new Label content.
 *
 * If reading the file fails, the label will be left unchanged.
 */
static void cbActionOpen(GtkWidget* widget, ApplicationState* app) {

    gchar* filename = NULL;
    /* We need to use URIs with GnomeVFS, so declare one here. */
    gchar* uri = NULL;

    g_assert(app != NULL);
    /* Bad things will happen if these two widgets don't exist. */
    g_assert(GTK_IS_LABEL(app->textLabel));
    g_assert(GTK_IS_WINDOW(app->>window));

    g_print("cbActionOpen invoked\n");

    /* Ask the user to select a file to open. */
    filename = runFileChooser(app, GTK_FILE_CHOOSER_ACTION_OPEN);
    if (filename) {
        /* This will point to loaded data buffer. */
        gchar* buffer = NULL;
        /* Pointer to a structure describing an open GnomeVFS "file". */
        GnomeVFSHandle* fileHandle = NULL;
        /* Structure to hold information about a "file", initialized to
         zero. */
        GnomeVFSFileInfo fileInfo = {};
        /* Result code from the GnomeVFS operations. */
        GnomeVFSResult result;
        /* Size of the file (in bytes) that we'll read in. */
        GnomeVFSFileSize fileSize = 0;
        /* Number of bytes that were read in successfully. */
        GnomeVFSFileSize readCount = 0;

        g_print(" you chose to load file '%s'\n", filename);

        /* Convert the filename into an GnomeVFS URI. */
        uri = gnome_vfs_get_uri_from_local_path(filename);
        /* We don't need the original filename anymore. */
        g_free(filename);
        filename = NULL;
        /* Should not happen since we got a filename before. */
        g_assert(uri != NULL);
        /* Attempt to get file size first. We need to get information
         about the file and aren't interested in other than the very
         basic information, so we'll use the INFO_DEFAULT setting. */
        result = gnome_vfs_get_file_info(uri, &fileInfo,
                                         GNOME_VFS_FILE_INFO_DEFAULT);
        if (result != GNOME_VFS_OK) {
            /* There was a failure. Print a debug error message and break

```

```

        out into error handling. */
        dbgFileError(result, uri);
        goto error;
    }

    /* We got the information (maybe). Let's check whether it
       contains the data that we need. */
    if (fileInfo.valid_fields & GNOME_VFS_FILE_INFO_FIELDS_SIZE) {
        /* Yes, we got the file size. */
        fileSize = fileInfo.size;
    } else {
        g_printerr("Couldn't get the size of file!\n");
        goto error;
    }

    /* By now we have the file size to read in. Check for some limits
       first. */
    if (fileSize > 1024*100) {
        g_printerr("Loading over 100KiB files is not supported!\n");
        goto error;
    }

    /* Refuse to load empty files. */
    if (fileSize == 0) {
        g_printerr("Refusing to load an empty file!\n");
        goto error;
    }

    /* Allocate memory for the contents and fill it with zeroes.
       NOTE:
       We leave space for the terminating zero so that we can pass
       this buffer as gchar to string functions and it is
       guaranteed to be terminated, even if the file doesn't end
       with binary zero (odds of that happening are small). */
    buffer = g_malloc0(fileSize+1);
    if (buffer == NULL) {
        g_printerr("Failed to allocate %u bytes for buffer\n",
                    (guint)fileSize);
        goto error;
    }

    /* Open the file.

       Parameters:
       - A pointer to the location where to store the address of the
         new GnomeVFS file handle (created internally in open).
       - uri: What to open (needs to be GnomeVFS URI).
       - open-flags: Flags that tell what we plan to use the handle
         for. This will affect how permissions are checked by the
         Linux kernel. */

    result = gnome_vfs_open(&fileHandle, uri, GNOME_VFS_OPEN_READ);
    if (result != GNOME_VFS_OK) {
        dbgFileError(result, uri);
        goto error;
    }

    /* File opened succesfully, read its contents in. */
    result = gnome_vfs_read(fileHandle, buffer, fileSize,
                            &readCount);
    if (result != GNOME_VFS_OK) {
        dbgFileError(result, uri);
        goto error;
    }
}

```

```

/* Verify that we got the amount of data that we requested.
NOTE:
    With URIs it won't be an error to get less bytes than you
    requested. Getting zero bytes will however signify an
    End-of-File condition. */
if (fileSize != readCount) {
    g_printerr("Failed to load the requested amount\n");
    /* We could also attempt to read the missing data until we have
    filled our buffer, but for simplicity, we'll flag this
    condition as an error. */
    goto error;
}

/* Whew, if we got this far, it means that we actually managed to
load the file into memory. Let's set the buffer contents as
the new label now. */
gtk_label_set_markup(GTK_LABEL(app->textLabel), buffer);

/* That's it! Display a message of great joy. For this we'll use
a dialog (non-modal) designed for displaying short
informational messages. It will linger around on the screen
for a while and then disappear (in parallel to our program
continuing). */
hildon_banner_show_information(GTK_WIDGET(app->window),
    NULL, /* Use the default icon (info). */
    "File loaded successfully");

/* Jump to the resource releasing phase. */
goto release;

error:
/* Display a failure message with a stock icon.
Please see http://maemo.org/api\_refs/4.0/gtk/gtk-Stock-Items.html
for a full listing of stock items. */
hildon_banner_show_information(GTK_WIDGET(app->window),
    GTK_STOCK_DIALOG_ERROR, /* Use the stock error icon. */
    "Failed to load the file");

release:
/* Close and free all resources that were allocated. */
if (fileHandle) gnome_vfs_close(fileHandle);
if (filename) g_free(filename);
if (uri) g_free(uri);
if (buffer) g_free(buffer);
/* Zero them all out to prevent stack-reuse-bugs. */
fileHandle = NULL;
filename = NULL;
uri = NULL;
buffer = NULL;

return;
} else {
    g_print(" you didn't choose any file to open\n");
}
}

/**
 * MODIFIED (kind of)
 *
 * Function to save the contents of the label (although it doesn't
 * actually save the contents, on purpose). Use gtk_label_get_label

```

```

* to get a gchar pointer into the application label contents
* (including current markup), then use gnome_vfs_create and
* gnome_vfs_write to create the file (left as an exercise).
*/
static void cbActionSave(GtkWidget* widget, ApplicationState* app) {
    gchar* filename = NULL;

    g_assert(app != NULL);

    g_print("cbActionSave invoked\n");

    filename = runFileChooser(app, GTK_FILE_CHOOSER_ACTION_SAVE);
    if (filename) {
        g_print(" you chose to save into '%s'\n", filename);
        /* Process saving .. */

        g_free(filename);
        filename = NULL;
    } else {
        g_print(" you didn't choose a filename to save to\n");
    }
}

/*... Listing cut for brevity ...*/
/**
 * MODIFIED
 *
 * Add support for GnomeVFS (it needs to be initialized before use)
 * and add support for the Pango markup feature of the GtkLabel
 * widget.
*/
int main(int argc, char** argv) {

    ApplicationState aState = {};

    GtkWidget* label = NULL;
    GtkWidget* vbox = NULL;
    GtkWidget* mainToolbar = NULL;
    GtkWidget* findToolbar = NULL;

    /* Initialize the GnomeVFS (NEW). */
    if(!gnome_vfs_init()) {
        g_error("Failed to initialize GnomeVFS-libraries, exiting\n");
    }

    /* Initialize the GTK+ */
    gtk_init(&argc, &argv);

    /* Setup the HildonProgram, HildonWindow and application name. */
    aState.program = HILDON_PROGRAM(hildon_program_get_instance());
    g_set_application_name("Hello Hildon!");
    aState.window = HILDON_WINDOW(hildon_window_new());
    hildon_program_add_window(aState.program,
                             HILDON_WINDOW(aState.window));

    /* Create the label widget, with Pango marked up content (NEW). */
    label = gtk_label_new("<b>Hello</b> <i>Hildon</i> (with Hildon"
                          "<sub>search</sub> <u>and</u> GnomeVFS "
                          "and other tricks<sup>tm</sup>)!");

    /* Allow lines to wrap (NEW). */

```

```

gtk_label_set_line_wrap(GTK_LABEL(label), TRUE);

/* Tell the GtkLabel widget to support the Pango markup (NEW). */
gtk_label_set_use_markup(GTK_LABEL(label), TRUE);

/* Store the widget pointer into the application state so that the
   contents can be replaced when a file will be loaded (NEW). */
aState.textLabel = label;

buildMenu(&aState);

vbox = gtk_vbox_new(FALSE, 0);
gtk_container_add(GTK_CONTAINER(aState.window), vbox);
gtk_box_pack_end(GTK_BOX(vbox), label, TRUE, TRUE, 0);

mainToolbar = buildToolbar(&aState);
findToolbar = buildFindToolbar(&aState);

aState.mainToolbar = mainToolbar;
aState.findToolbar = findToolbar;

/* Connect the termination signals. */
g_signal_connect(G_OBJECT(aState.window), "delete-event",
                 G_CALLBACK(cbEventDelete), &aState);
g_signal_connect(G_OBJECT(aState.window), "destroy",
                 G_CALLBACK(cbActionTopDestroy), &aState);

/* Show all widgets that are contained by the Window. */
gtk_widget_show_all(GTK_WIDGET(aState.window));

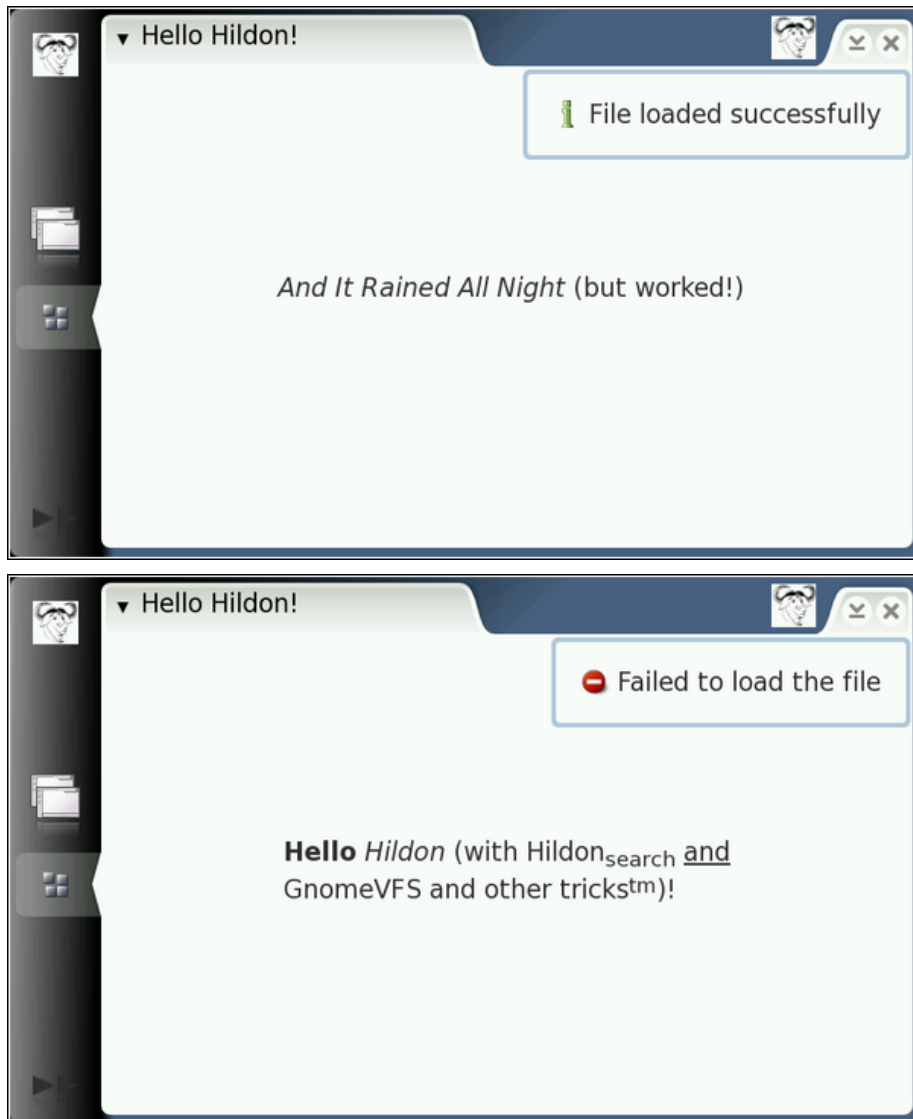
/* Add the toolbars to the Hildon Window. */
hildon_window_add_toolbar(HILDON_WINDOW(aState.window),
                         GTK_TOOLBAR(mainToolbar));
hildon_window_add_toolbar(HILDON_WINDOW(aState.window),
                         GTK_TOOLBAR(findToolbar));

/* Register a callback to handle key presses. */
g_signal_connect(G_OBJECT(aState.window), "key_press_event",
                 G_CALLBACK(cbKeyPressed), &aState);

g_print("main: calling gtk_main\n");
gtk_main();
g_print("main: returned from gtk_main and exiting with success\n");

return EXIT_SUCCESS;
}

```



In order to experiment with loading other content, a simple file can be created, containing Pango markup like this: `echo "Hello world" > MyDocs/hello.txt`, and then loading `hello.txt`.

As can be imagined, these examples have only scratched the surface of GnomeVFS that is quite a rich library, and contains a broad API and a large amount of plug-ins. Many things have been completely avoided, such as directory content iteration, the asynchronous interface, callback signaling on directory content changes etc. Please see GnomeVFS API [57] for more information. The API also contains some mini tutorials on various GnomeVFS topics, so it is well worth the time spent reading. It will also show that GnomeVFS has been overloaded with functions, which are not even file operation related (such as ZeroConf and creating TCP/IP connections etc.).

GTK+ does not have to be used in order to use GnomeVFS. One such example program is Midnight Commander (a Norton Commander clone, but

better), which is a menu-based "text" mode program. GnomeVFS uses GLib though, so if using GnomeVFS, one should think about using GLib as well, as it will be loaded anyway.

7.3 Message Bus System - D-Bus

For interprocess communications (IPC), maemo relies heavily on D-Bus. D-Bus makes it possible for programs to export their programming interfaces, so that other processes can call them in a consistent manner, without having to define a custom IPC protocol. Using these exported APIs is also language agnostic, so as long as programming language supports D-Bus, it can also access the interfaces.

A maemo-specific library called *libosso* provides helpful wrappers for D-BUS communication. It also contains the required functionality for every maemo application. Applications must be initialized using this library. With it, applications can connect to listen to system hardware state messages, such as "battery low". The library is used also for application state saving and auto-save functionality. Section *LibOSSO Library 6.9.2* of the chapter *Application Development* of Maemo Reference Manual provides a good introduction to libosso.

7.3.1 D-Bus Basics

Introduction

D-Bus (the D originally stood for "Desktop") is a relatively new interprocess communication (IPC) mechanism designed to be used as a unified middleware layer in free desktop environments. Some example projects, where D-Bus is used, are GNOME and Hildon. Compared to other middleware layers for IPC, D-Bus lacks many of the more refined (and complicated) features, and thus is faster and simpler.

D-Bus does not directly compete with low-level IPC mechanisms, such as sockets, shared memory or message queues. Each of these mechanisms have their uses, which normally do not overlap the ones in D-Bus. Instead, D-Bus aims to provide higher level functionality, such as:

- Structured name spaces
- Architecture-independent data formatting
- Support for the most common data elements in messages
- A generic remote call interface with support for exceptions (errors)
- A generic signaling interface to support "broadcast" type communication
- Clear separation of per-user and system-wide scopes, which is important when dealing with multi-user systems
- No bindings to any specific programming languages (while providing a design that readily maps to most higher level languages, via language-specific bindings)

The design of D-Bus benefits from the long experience of using other middleware IPC solutions in the desktop arena, and this has allowed the design to be optimized. Also, it does not yet suffer from "creeping featurism", e.g. having extra features just to satisfy niche use cases.

All this said, the main problem area that D-Bus aims to solve is facilitating easy IPC between related (often graphical) desktop software applications.

D-Bus has a very important role in maemo, as it is the IPC mechanism to use when using the services provided in the platform (and devices). Providing services over D-Bus is also the easiest way to assure component re-use from other applications.

D-Bus Architecture and Terminology

In D-Bus, the *bus* is a central concept. It is the channel through which applications can make the method calls, send signals and listen to signals. There are two pre-defined buses: the *session bus* and the *system bus*.

- The session bus is meant for communication between applications that are connected to the same desktop session, and normally started and run by one user (using the same user identifier, or UID).
- The system bus is meant for communication when applications (or services), running with disparate sessions, wish to communicate with each other. The most common use for this bus is sending system-wide notifications, when system-wide events occur. Adding of a new storage device, network connectivity change events and shutdown-related events are all examples of when system bus would be the more suitable bus for communication.

Normally only one system bus will exist, but there might be several session buses (one per each desktop session). Since in Internet Tablets all user applications will run with the same user id (user), there will only be one session bus as well.

A bus exists in the system in the form of a *bus daemon*, a process that specializes in passing messages from one process to another. The daemon will also forward notifications to all applications on the bus. At the lowest level, D-Bus only supports point-to-point communication, normally using the local domain sockets (AF_UNIX) between the application and the bus daemon. The point-to-point aspect of D-Bus is however abstracted by the bus daemon, which will implement addressing and message passing functionality, so that applications do not need to care about which specific process will receive each method call or notification.

The above means that sending a message using D-Bus will always involve the following steps (under normal conditions):

- Creation and sending of the message to the bus daemon. This will cause at minimum two context switches.
- Processing of the message by the bus daemon and forwarding it to the target process. This will again cause at minimum two context switches.

- The target application will receive the message. Depending on the message type, it will either need to acknowledge it, respond with a reply or ignore it. The last case is only possible with notifications (i.e., *signals* in D-Bus terminology). An acknowledgment or reply will cause further context switches.

Coupled together, the above rules mean that if planning to transfer large amounts of data between processes, D-Bus will not be the most efficient way to do it. The most efficient way would be using some kind of shared memory arrangement. However, it is often quite complex to implement correctly.

Addressing and Names in D-Bus

In order for the messages to reach the intended recipient, the IPC mechanism needs to support some form of addressing. The addressing scheme in D-Bus has been designed to be flexible, but at the same time efficient. Each bus has its private name space, which is not directly related to any other bus.

In order to send a message, a destination address is needed. It is formed in a hierarchical manner from the following elements:

- The bus on which the message is to be sent. A bus is normally opened only once per application lifetime. The bus connection will then be used for sending and receiving messages for as long as necessary. This way, the target bus will form a transparent part of the message address (i.e., it is not specified separately for each message sent).
- The *well-known name* for the service provided by the recipient. A close analogy to this would be the DNS system in Internet, where people normally use names to connect to services, instead of specific IP addresses providing the services. The idea in D-Bus well-known names is very similar, since the same service might be implemented in different ways in different applications. It should be noted, however, that currently most of the existing D-Bus services are "unique" in that each of them provides their own well-known name, and replacing one implementation with another is not common.
 - A well-known name consists of characters A-Z (lower or uppercase), dot characters, dashes and underscores. There must be at least two dot-separated elements in a well-known name. Unlike DNS, the dots do not carry any additional information about management (zones), meaning that the well-known names are NOT hierarchical.
 - In order to reduce clashes in the D-Bus name space, it is recommended that the name is formed by reversing the order of labels of a DNS domain that you own. A similar approach is used in Java for package names.
 - Examples: org.maemo.Alert and org.freedesktop.Notifications.
- Each service can contain multiple different objects, each of which provides a different (or same) service. In order to separate one object from another, *object paths* are used. A PIM information store, for example, might include separate objects to manage the contact information and synchronization.

- Object paths look like file paths (elements separated with the character '/').
- In D-Bus, it is also possible to make a "lazy binding", so that a specific function in the recipient will be called on all remote method calls, irrespective of object paths in the calls. This allows on-demand targeting of method calls, so that a user might remove a specific object in an address book service (using an object path similar to /org/maemo/AddressBook/Contacts/ShortName). Due to the limitations in characters that can be put into the object path, this is not recommended. A better way would be to supply the ShortName as a method call argument instead (as a UTF-8 formatted string).
- It is common to form the object path using the same elements as in the well-known name, but replacing the dots with slashes, and appending a specific object name to the end. For example: /org/maemo/Alert/Alerter. It is a convention, but also solves a specific problem, when a process might re-use an existing D-Bus connection without explicitly knowing about it (using a library that encapsulates D-Bus functionality). Using short names here would increase the risk of name-space collisions within that process.
- Similar to well-known names, object paths do not have inherent hierarchy, even if the path separator is used. The only place where some hierarchy might be seen because of path components is the introspection interface (which is out of the scope of this material).
- In order to support object-oriented mapping, where objects are the units providing the service, D-Bus also implements a naming unit called the *interface*. The interface specifies the legal (i.e. defined and implemented) method calls, their parameters (called *arguments* in D-Bus) and possible signals. It is then possible to re-use the same interface across multiple separate objects implementing the same service, or more commonly, a single object can implement multiple different services. An example of the latter is the implementation of the org.freedesktop.DBus.Introspectable interface, which defines the method necessary to support D-Bus introspection (more about this later on). When using the GLib/D-Bus wrappers to generate parts of the D-Bus code, the objects will automatically also support the introspection interface.
 - Interface names use the same naming rules as well-known names. This might seem somewhat confusing at start, since well-known names serve a completely different purpose, but with time, one will get used to it.
 - For simple services, it is common to repeat the well-known name in the interface name. This is the most common scenario with existing services.
- The last part of the message address is the member name. When dealing with remote procedure calls, this is also sometimes called *method name*, and when dealing with signals, *signal name*. The member name selects the procedure to call, or the signal to emit. It needs to be unique only within the interface that an object will implement.

- Member names can have letters, digits and underscores in them. For example: RetrieveQuote.

- For a more in-depth review on these, please see the [Introduction](#) to D-Bus page.

That about covers the most important rules in D-Bus addresses that one is likely to encounter. Below is an example of all four components that will also be used shortly to send a simple message (a method call) in the SDK:

```
#define SYSNOTE_NAME "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE "SystemNoteDialog"
```

Even if switching to use the LibOSSO RPC functions (which encapsulate a lot of the D-Bus machinery), operations will still be performed with all of the D-Bus naming components.

Role of D-Bus in Maemo

D-Bus has been selected as de facto IPC mechanism in maemo, to carry messages between the various software components. The main reason for this is that a lot of software developed for the GNOME environment is already exposing its functionality through D-Bus. Using a generic interface, which is not bound to any specific service, makes it also easier to deal with different software license requirements.

The SDK unfortunately does not come with a lot of software that is exposed via D-Bus, but this document will be using one component of the application framework as demonstration (it works also in the SDK).

An item of particular interest is asking the notification framework component to display a Note dialog. The dialog is modal, which means that users cannot proceed in their graphical environment, unless they first acknowledge the dialog. Normally such GUI decisions should be avoided, but later in this document it will be discussed why and when this feature can be useful. N.B. The SystemNoteDialog member is an extension to the draft org.freedesktop.Notifications specification, and as such, is not documented in that draft.

The notification server is listening for method calls on the org.freedesktop.Notifications well-known name. The object that implements the necessary interface is located at /org/freedesktop/Notifications object path. The method to display the note dialog is called SystemNoteDialog, and is defined in the org.freedesktop.Notifications D-Bus interface.

D-Bus comes with a handy tool to experiment with method calls and signals: dbus-send. The following snippet will attempt to use it to display the dialog:

```
[sbox-DIABLO_X86: ~] > run-standalone.sh dbus-send --print-reply \
--type=method_call --dest=org.freedesktop.Notifications \
/org/freedesktop/Notifications org.freedesktop.Notifications
Error org.freedesktop.DBus.Error.UnknownMethod: Method "Notifications" with
signature "" on interface "org.freedesktop" doesn't exist
```

Parameters for dbus-send:

- --session: (implicit since default) which bus to use for sending (the other option being system)

- `--print-reply`: ask the tool to wait for a reply to the method call, and print out the results (if any)
- `--type=method_call`: instead of sending a signal (which is the default), make a method call
- `--dest=org.freedesktop.Notifications`: the well-known name for the target service
- `/org/freedesktop/Notifications`: object path within the target process that implements the interface
- `org.freedesktop.Notifications`: (incorrectly specified) interface name defining the method

When using `dbus-send`, extra care needs to be taken, when specifying the interface and member names. The tool expects both of them to be combined into one parameter (without spaces in between). Thus, the command line needs to be modified a bit before a new try:

```
[sbox-DIABLO_X86: ~] > run-standalone.sh dbus-send --print-reply \
--type=method_call --dest=org.freedesktop.Notifications \
/org/freedesktop/Notifications org.freedesktop.Notifications.SystemNoteDialog
Error org.freedesktop.DBus.Error.UnknownMethod: Method "SystemNoteDialog" with
signature "" on interface "org.freedesktop.Notifications" doesn't exist
```

Seems that the RPC call is still missing something. Most RPC methods will expect a series of parameters (or arguments, as D-Bus calls them).

`SystemNoteDialog` expects these three parameters (in the following order):

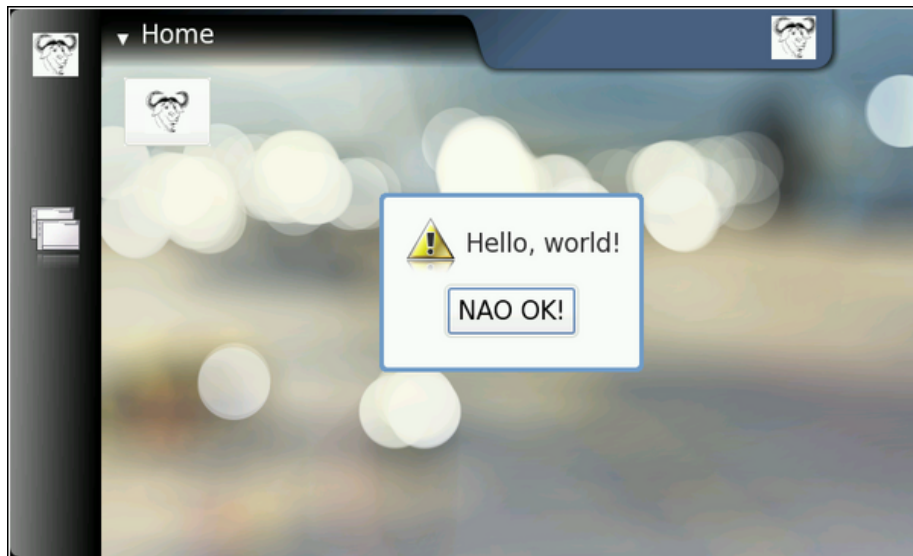
- `string`: The message to display
- `uint32`: An unsigned integer giving the style of the dialog. Styles 0-4 mean different icons, and style 5 is a special animated "progress indicator" dialog.
- `string`: Message to use for the "Ok" button that the user needs to press to dismiss the dialog. Using an empty string will cause the default text to be used (which is "Ok").

Arguments are specified by giving the argument type and its contents separated with a colon as follows:

```
[sbox-DIABLO_X86: ~] > run-standalone.sh dbus-send --print-reply \
--type=method_call --dest=org.freedesktop.Notifications \
/org/freedesktop/Notifications org.freedesktop.Notifications.SystemNoteDialog \
string:'Hello, world!' uint32:0 string:'NAO OK!'
method return sender=:1.1 -> dest=:1.15
uint32 4
```

Since `dbus-send` was asked to print replies, the reply will come out as a single unsigned integer, with value of 4. This is the unique number for this notification, and could be used with the `CloseNotification` method of the `Notifications` interface to pre-emptively close the dialog. It might be especially useful, if the software can notice that some warning condition has ended, and there is no need to bother the user with the warning anymore.

Assuming that the above command is run while the application framework is already running, the end result should more or less look like this:



If the command is repeated multiple times, one will notice that the notification service is capable of displaying only one dialog at a time. This makes sense, as the dialog is modal anyway. It can also be noticed that the method calls are queued somewhere, and not lost (i.e. the notification service will display all of the requested dialogs). The service also acknowledges the RPC method call without delay (which is not always the obvious thing to do), giving a different return value each time (incrementing by one each time).

Programming Directly with Libdbus

The lowest level library to use for D-Bus programming is libdbus. Using this library directly is discouraged, mostly because it contains a lot of specific code to integrate into various main-loop designs that the higher level language bindings use.

The libdbus API reference documentation [57] contains a helpful note:

```
/**
 * Uses the low-level libdbus which shouldn't be used directly.
 * As the D-Bus API reference puts it "If you use this low-level API
 * directly, you're signing up for some pain".
 */
```

At this point, this example will ignore the warnings, and use the library to implement a simple program that will replicate the dbus-send example that was seen before. In order to do this with the minimum amount of code, the code will not process (or expect) any responses to the method call. It will, however, demonstrate the bare minimum function calls that are needed to use to send messages on the bus.

The first step is to introduce the necessary header files.

```
#include <dbus/dbus.h> /* Pull in all of D-Bus headers. */
#include <stdio.h>      /* printf, fprintf, stderr */
#include <stdlib.h>     /* EXIT_FAILURE, EXIT_SUCCESS */
#include <assert.h>     /* assert */
```

```

/* Symbolic defines for the D-Bus well-known name, interface, object
   path and method name that we're going to use. */

#define SYSNOTE_NAME    "org.freedesktop.Notifications"
#define SYSNOTE_OPATH   "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE   "org.freedesktop.Notifications"
#define SYSNOTE_NOTE    "SystemNoteDialog"

```

Listing 7.1: libdbus-example/dbus-example.c

Unlike the rest of the code in this material, dbus-example does not use GLib or other support libraries (other than libdbus). This explains why it uses printf and other functions that would normally be replaced with GLib equivalents.

Connecting to the session bus will (hopefully) yield a DBusConnection structure:

```

/**
 * The main program that demonstrates a simple "fire & forget" RPC
 * method invocation.
 */
int main(int argc, char** argv) {

    /* Structure representing the connection to a bus. */
    DBusConnection* bus = NULL;
    /* The method call message. */
    DBusMessage* msg = NULL;

    /* D-Bus will report problems and exceptions using the DBusError
       structure. We'll allocate one in stack (so that we don't need to
       free it explicitly. */
    DBusError error;

    /* Message to display. */
    const char* dispMsg = "Hello World!";
    /* Text to use for the acknowledgement button. "" means default. */
    const char* buttonText = "";
    /* Type of icon to use in the dialog (1 = OSSO_GN_ERROR). We could
       have just used the symbolic version here as well, but that would
       have required pulling the LibOSSO-header files. And this example
       must work without LibOSSO, so this is why a number is used. */
    int iconType = 1;

    /* Clean the error state. */
    dbus_error_init(&error);

    printf("Connecting to Session D-Bus\n");
    bus = dbus_bus_get(DBUS_BUS_SESSION, &error);
    terminateOnError("Failed to open Session bus\n", &error);
    assert(bus != NULL);

```

Listing 7.2: libdbus-example/dbus-example.c

N.B. Libdbus will attempt to share existing connection structures when the same process is connecting to the same bus. This is done to avoid the somewhat costly connection set-up time. Sharing connections is beneficial, when the program is using libraries that would also open their own connections to the same buses.

In order to communicate errors, libdbus uses DBusError structures, whose contents are pretty simple. The dbus_error_init will be used to guarantee that

the error structure contains a non-error state before connecting to the bus. If there is an error, it will be handled in `terminateOnError`:

```
/**
 * Utility to terminate if given DBusError is set.
 * Will print out the message and error before terminating.
 *
 * If error is not set, will do nothing.
 *
 * NOTE: In real applications you should spend a moment or two
 *       thinking about the exit-paths from your application and
 *       whether you need to close/unreference all resources that you
 *       have allocated. In this program, we rely on the kernel to do
 *       all necessary cleanup (closing sockets, releasing memory),
 *       but in real life you need to be more careful.
 *
 *       One possible solution model to this is implemented in
 *       "flashlight", a simple program that is presented later.
 */
static void terminateOnError(const char* msg,
                           const DBusError* error) {

    assert(msg != NULL);
    assert(error != NULL);

    if (dbus_error_is_set(error)) {
        fprintf(stderr, msg);
        fprintf(stderr, "DBusError.name: %s\n", error->name);
        fprintf(stderr, "DBusError.message: %s\n", error->message);
        /* If the program wouldn't exit because of the error, freeing the
         * DBusError needs to be done (with dbus_error_free(error)).
         * NOTE:
         *       dbus_error_free(error) would only free the error if it was
         *       set, so it is safe to use even when you're unsure. */
        exit(EXIT_FAILURE);
    }
}
```

Listing 7.3: `libdbus-example/dbus-example.c`

`libdbus` also contains some utility functions, so that everything does not have to be coded manually. One such utility is `dbus_bus_name_has_owner` that checks, whether there is at least some process that owns the given well-known name at that moment:

```
/* Normally one would just do the RPC call immediately without
 * checking for name existence first. However, sometimes it is useful
 * to check whether a specific name even exists on a platform on
 * which you are planning to use D-Bus.

 * In our case it acts as a reminder to run this program using the
 * run-standalone.sh script when running in the SDK.

 * The existence check is not necessary if the recipient is
 * startable/activateable by D-Bus. In that case, if the recipient
 * is not already running, the D-Bus daemon will start the
 * recipient (a process that has been registered for that
 * well-known name) and then passes the message to it. This
 * automatic starting mechanism will avoid the race condition
 * discussed below and also makes sure that only one instance of
 * the service is running at any given time. */
printf("Checking whether the target name exists ("
```

```

        SYSNOTE_NAME "\n");
if (!dbus_bus_name_has_owner(bus, SYSNOTE_NAME, &error)) {
    fprintf(stderr, "Name has no owner on the bus!\n");
    return EXIT_FAILURE;
}
terminateOnError("Failed to check for name ownership\n", &error);
/* Someone on the Session bus owns the name. So we can proceed in
   relative safety. There is a chance of a race. If the name owner
   decides to drop out from the bus just after we check that it is
   owned, our RPC call (below) will fail anyway. */

```

Listing 7.4: libdbus-example/dbus-example.c

Creating a method call using libdbus is slightly more tedious than using the higher-level interfaces, but not very difficult. The process is separated into two steps: creating a message structure, and appending the arguments to the message:

```

/* Construct a DBusMessage that represents a method call.
   Parameters will be added later. The internal type of the message
   will be DBUS_MESSAGE_TYPE_METHOD_CALL. */
printf("Creating a message object\n");
msg = dbus_message_new_method_call(SYSNOTE_NAME, /* destination */
                                   SYSNOTE_OPATH, /* obj. path */
                                   SYSNOTE_IFACE, /* interface */
                                   SYSNOTE_NOTE); /* method str */

if (msg == NULL) {
    fprintf(stderr, "Ran out of memory when creating a message\n");
    exit(EXIT_FAILURE);
}

/*... Listing cut for brevity ...*/

/* Add the arguments to the message. For the Note dialog, we need
   three arguments:
   arg0: (STRING) "message to display, in UTF-8"
   arg1: (UINT32) type of dialog to display. We will use 1.
         (libosso.h/OSSO_GN_ERROR).
   arg2: (STRING) "text to use for the ack button". "" means
         default text (OK in our case).

   When listing the arguments, the type needs to be specified first
   (by using the libdbus constants) and then a pointer to the
   argument content needs to be given.

   NOTE: It is always a pointer to the argument value, not the value
         itself!

   We terminate the list with DBUS_TYPE_INVALID. */
printf("Appending arguments to the message\n");
if (!dbus_message_append_args(msg,
                              DBUS_TYPE_STRING, &dispMsg,
                              DBUS_TYPE_UINT32, &iconType,
                              DBUS_TYPE_STRING, &buttonText,
                              DBUS_TYPE_INVALID)) {
    fprintf(stderr, "Ran out of memory while constructing args\n");
    exit(EXIT_FAILURE);
}

```

Listing 7.5: libdbus-example/dbus-example.c

When arguments are appended to the message, their content is copied, and possibly converted into a format that will be sent over the connection to the daemon. This process is called marshaling, and is a common feature to most RPC systems. The method call will require two parameters (as before), the first being the text to display, and the second one being the style of the icon to use. Parameters passed to libdbus are always passed by address. This is different from the higher level libraries, and this will be discussed later.

The arguments are encoded, so that their type code is followed by the pointer where the marshaling functions can find the content. The argument list is terminated with DBUS_TYPE_INVALID, so that the function knows where the argument list ends (since the function prototype ends with an ellipsis, ...).

```
/* Set the "no-reply-wanted" flag into the message. This also means
   that we cannot reliably know whether the message was delivered or
   not, but since we don't have reply message handling here, it
   doesn't matter. The "no-reply" is a potential flag for the remote
   end so that they know that they don't need to respond to us.

   If the no-reply flag is set, the D-Bus daemon makes sure that the
   possible reply is discarded and not sent to us. */
dbus_message_set_no_reply(msg, TRUE);
```

Listing 7.6: libdbus-example/dbus-example.c

Setting the no-reply-flag effectively tells the bus daemon that even if there is a reply coming back for this RPC method, it is not wanted. In this case, the daemon will not send one.

Once the message is fully constructed, it can be added to the sending queue of the program. Messages are not sent immediately by libdbus. Normally this allows the message queue to accumulate to more than one message, and all of the messages will be sent at once to the daemon. This in turn cuts down the number of context switches necessary. In this case, this will be the only message that the program ever sends, so the send queue is instructed to be flushed immediately, and this will instruct the library to send all messages to the daemon without a delay:

```
printf("Adding message to client's send-queue\n");
/* We could also get a serial number (dbus_uint32_t) for the message
   so that we could correlate responses to sent messages later. In
   our case there won't be a response anyway, so we don't care about
   the serial, so we pass a NULL as the last parameter. */
if (!dbus_connection_send(bus, msg, NULL)) {
    fprintf(stderr, "Ran out of memory while queueing message\n");
    exit(EXIT_FAILURE);
}

printf("Waiting for send-queue to be sent out\n");
dbus_connection_flush(bus);

printf("Queue is now empty\n");
```

Listing 7.7: libdbus-example/dbus-example.c

After the message is sent, the reserved resources should be freed. Here, the first one to be freed is the message, and then the connection structure.

```
printf("Cleaning up\n");
```

```

/* Free up the allocated message. Most D-Bus objects have internal
   reference count and sharing possibility, so _unref() functions
   are quite common. */
dbus_message_unref(msg);
msg = NULL;

/* Free-up the connection. libdbus attempts to share existing
   connections for the same client, so instead of closing down a
   connection object, it is unreferenced. The D-Bus library will
   keep an internal reference to each shared connection, to
   prevent accidental closing of shared connections before the
   library is finalized. */
dbus_connection_unref(bus);
bus = NULL;

printf("Quitting (success)\n");

return EXIT_SUCCESS;
}

```

Listing 7.8: libdbus-example/dbus-example.c

After building the program, attempt to run it:

```

[sbox-DIABLO_X86: ~/libdbus-example] > ./dbus-example
Connecting to Session D-Bus
process 6120: D-Bus library appears to be incorrectly set up;
failed to read machine uuid:
  Failed to open "/var/lib/dbus/machine-id": No such file or directory
See the manual page for dbus-uuidgen to correct this issue.
  D-Bus not built with -rdynamic so unable to print a backtrace
Aborted (core dumped)

```

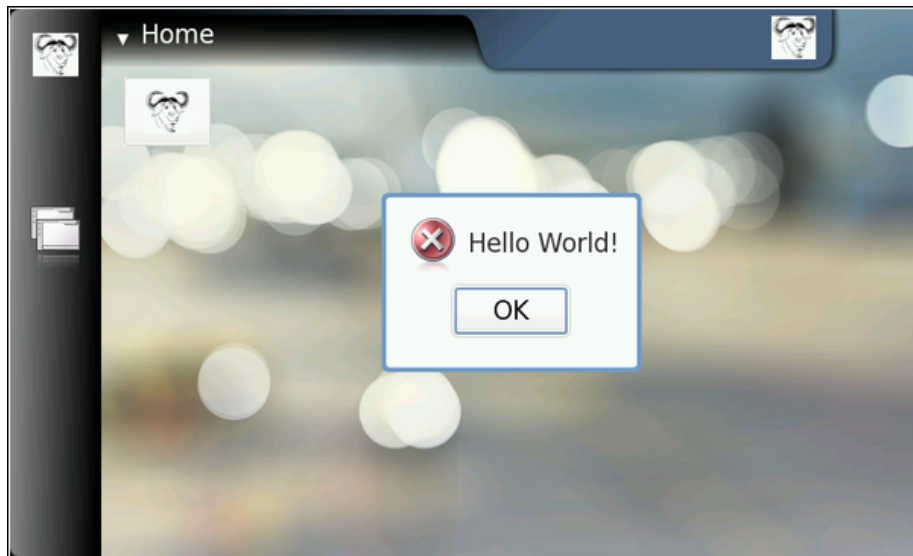
The D-Bus library needs environmental variables set correctly in order to locate the session daemon. The command was not prepended with `run-standalone.sh`, and this caused the library to internally abort the execution. Normally, `dbus_bus_get` would have returned a NULL pointer and set the error structure, but the version on the 4.1 SDK will assert internally in this condition, and programs cannot avoid the abort. After correcting this, try again:

```

[sbox-DIABLO_X86: ~/libdbus-example] > run-standalone.sh ./dbus-example
Connecting to Session D-Bus
Checking whether the target name exists (org.freedesktop.Notifications)
Creating a message object
Appending arguments to the message
Adding message to client's send-queue
Waiting for send-queue to be sent out
Queue is now empty
Cleaning up
Quitting (success)
/dev/dsp: No such file or directory

```

The error message (about `/dev/dsp`) printed to the same terminal where AF was started is normal (in SDK). Displaying the Note dialog normally also causes an "Alert" sound to be played. The sound system has not been setup in the SDK, so the notification component complains about failing to open the sound device.



The friendly error message, using low-level D-Bus

In order to get `libdbus` integrated into makefiles, `pkg-config` has to be used. One possible solution is presented below (see section *GNU Make and Makefiles 4.2* in chapter *GNU Build System*, if necessary):

```
# Define a list of pkg-config packages we want to use
pkg_packages := dbus-glib-1

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)
```

Listing 7.9: `libdbus-example/Makefile`

The above shows one possibility to integrate user-supplied variables into makefiles, so that they will still be passed along the toolchain. This allows the user to execute `make` with custom flags, overriding those that are introduced via other means. For example: `"CFLAGS='-g0' make"` would result in `-g0` being interpreted after the `-g` that is in the Makefile, and this would lead to debugging symbols being disabled. Environmental variables can be taken into account in exactly the same way.

For more complicated programs, it is likely that multiple different `CFLAGS` settings are required for different object files or multiple different programs that are being built. In that case, the combining in each target rule would be performed separately. In this material, all the example programs are self-contained and rather simple, so the above mechanism will be used in all the example makefiles.

7.3.2 LibOSSO

Introduction to LibOSSO

LibOSSO is a library that all applications designed for maemo are expected to use. Mainly because it automatically allows the application to survive the task killing process. This task killing is performed by the Desktop environment, when an application launched from the Task navigator does not register the proper D-Bus name on the bus within a certain time limit after the launch. LibOSSO also conveniently isolates the application from possible implementation changes on D-Bus level. D-Bus used to be not API stable before as well, so LibOSSO provided "version isolation" with respect D-Bus. Since D-Bus has reached maturity (1.0), no API changes are expected for the low level library, but the GLib/D-Bus wrapper might still change at some point.

Besides the protection and isolation services, LibOSSO also provides useful utility functions to handle auto saving and state saving features of the platform, process hardware state and device mode changes, and other important events happening in Internet Tablets. It also provides convenient utility wrapper functions to send RPC method calls over the D-Bus. The feature set is aimed at covering the most common GUI application needs, and as such, will not be enough in all cases. In these cases, it will be necessary to use the GLib/D-Bus wrapper functions (or libdbus directly, which is not recommended).

Using LibOSSO for D-Bus Method Calls

The first step is to re-implement the functionality from the libdbus example that was used before, but use LibOSSO functions instead of direct libdbus ones. The new version will use exactly the same D-Bus name-space components to pop up a Note dialog. LibOSSO also contains a function to do all this automatically (osso_system_note_dialog), which will be used directly later on. It is, however, instructive to see what LibOSSO provides in terms of RPC support, and using a familiar RPC method is the easiest way to achieve this.

The starting point here will be the header section of the example program:

```
#include <libosso.h>

/*... Listing cut for brevity ...*/

#define SYSNOTE_NAME "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE "SystemNoteDialog"
```

Listing 7.10: libosso-example-sync/libosso-rpc-sync.c

LibOSSO by itself only requires the **libosso.h** header file to be included. The example will also use the exact same D-Bus well-known name, object path, interface name and method name as before.

When reading other source code that implements or uses D-Bus services, one might sometimes wonder, why the D-Bus interface name is using the same symbolic constant as the well-known name (in the above example `SYSNOTE_IFACE` would be omitted, and `SYSNOTE_NAME` would be used whenever an interface name would be required). If the service in question is not easily reusable or re-implementable, it might make sense to use an interface name that is as

unique as the well-known name. This goes against the idea of defining interfaces, but is still quite common, and is the easy way out without bothering with difficult design decisions.

The following will take a look at how LibOSSO contexts are created, and how they are eventually released:

```
int main(int argc, char** argv) {

    /* The LibOSSO context that we need to do RPC. */
    osso_context_t* ossoContext = NULL;

    g_print("Initializing LibOSSO\n");
    /* The program name for registration is communicated from the
       Makefile via a -D preprocessor directive. OSSO_SERVICE has
       'com.nokia.' prefix added to the program name. */
    ossoContext = osso_initialize(OSSO_SERVICE, "1.0", FALSE, NULL);
    if (ossoContext == NULL) {
        g_error("Failed to initialize LibOSSO\n");
    }

    g_print("Invoking the method call\n");
    runRPC(ossoContext);

    g_print("Shutting down LibOSSO\n");
    /* Deinitialize LibOSSO. The function does not return status code so
       we cannot know whether it succeeded or failed. We assume that it
       always succeeds. */
    osso_deinitialize(ossoContext);
    ossoContext = NULL;

    g_print("Quitting\n");
    return EXIT_SUCCESS;
}
```

Listing 7.11: libosso-example-sync/libosso-rpc-sync.c

A LibOSSO context is a small structure, containing the necessary information for the LibOSSO functions to communicate over D-Bus (both session and system buses). When a context is created, the "application name" needs to be passed to `osso_initialize`. This name is used to register a name on the D-Bus, and this will keep the task killer from killing the process later on (assuming the application was started via the Task navigator). If the application name does not contain any dot characters in it, `com.nokia.` will be prepended to it automatically in LibOSSO. The application name is normally not visible to users, so this should not be a big problem. Application name collisions might be encountered, if some other application uses the same name (even without the dots), so it might be a good idea to provide a proper name based on a DNS domain you own or control. If planning to implement a service to clients over the D-Bus (with `osso_rpc_set_cb`-functions), it is necessary to be extra careful about the application name used here.

The version number is currently still unused, but 1.0 is recommended for the time being. The second to last parameter is obsolete, and has no effect, while the last parameter tells LibOSSO, which mainloop structure to integrate into. Using `NULL` here means that LibOSSO event processing will integrate into the default `GMainLoop` object created, as is most often desired.

Releasing the LibOSSO context will automatically close the connections to the D-Bus buses, and release all the allocated memory related to the connections

and LibOSSO state. When using LibOSSO functions after this, a context will have to be reinitialized.

The following snippet shows the RPC call using LibOSSO, and also contains code suitable for dealing with possible errors in the launch, as well as the result of the RPC. The `ossoErrorStr` function is covered shortly, as is the utility function to print out the result structure.

```
/**
 * Do the RPC call.
 *
 * Note that this function will block until the method call either
 * succeeds, or fails. If the method call would take a long time to
 * run, this would block the GUI of the program (which we do not have).
 *
 * Needs the LibOSSO state to do the launch.
 */
static void runRPC(osso_context_t* ctx) {

    /* Message to display. */
    const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/sync.";
    /* Icon type to use. */
    gint iconType = OSSO_GN_ERROR;
    /* Button label text to use, "" means leaving the defaults. */
    const char* labelText = "";

    /* Will hold the result from the RPC invocation function. */
    osso_return_t result;
    /* Will hold the result of the method call (or error). */
    osso_rpc_t methodResult = {};

    g_print("runRPC called\n");

    g_assert(ctx != NULL);

    /* Compared to the libdbus functions, LibOSSO provides conveniently
     a function that will do the dispatch and also allows us to pass
     the arguments all with one call.

     The arguments for the "SystemNoteDialog" are the same as in
     dbus-example.c (since it is the same service). You might also
     notice that even if LibOSSO provides some convenience, it does
     not completely isolate us from libdbus. We still supply the
     argument types using D-Bus constants.

     NOTE Do not pass the argument values by pointers as with libdbus,
     instead pass them by value (as below). */
    result = osso_rpc_run(ctx,
        SYSNOTE_NAME,          /* well-known name */
        SYSNOTE_OPATH,        /* object path */
        SYSNOTE_IFACE,        /* interface */
        SYSNOTE_NOTE,         /* method name */
        &methodResult, /* method return value */
        /* The arguments for the RPC. The types
         are unchanged, but instead of passing
         them via pointers, they are passed by
         "value" instead. */
        DBUS_TYPE_STRING, dispMsg,
        DBUS_TYPE_UINT32, iconType,
        DBUS_TYPE_STRING, labelText,
        DBUS_TYPE_INVALID);

    /* Check whether launching the RPC succeeded. */
```



```

if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
           ossoErrorStr(result));
    /* We also terminate right away since there is nothing to do. */
}
g_print("RPC launched successfully\n");

/* Now decode the return data from the method call.
   NOTE: If there is an error during RPC delivery, the return value
   will be a string. It is not possible to differentiate that
   condition from an RPC call that returns a string.

   If a method returns "void", the type-field in the methodResult
   will be set to DBUS_TYPE_INVALID. This is not an error. */
g_print("Method returns: ");
printOssoValue(&methodResult);
g_print("\n");

g_print("runRPC ending\n");
}

```

Listing 7.12: libosso-example-sync/libosso-rpc-sync.c

It is important to note that `osso_rpc_run` is a synchronous (blocking) call, which will wait for either the response from the method call, a timeout or an error. In this case, the method will be handled quickly, so it is not a big problem, but in many cases the methods will take some time to execute (and might require loading external resources), so this should be kept in mind. Asynchronous LibOSSO RPC functions will be covered shortly.

If the method call will return more than one return value (this is possible in D-Bus), LibOSSO currently does not provide a mechanism to return all of them (it will return the first value only).

Decoding the result code from the LibOSSO RPC functions is pretty straightforward, and is done in a separate utility:

```

/**
 * Utility to return a pointer to a statically allocated string giving
 * the textual representation of LibOSSO errors. Has no internal
 * state (safe to use from threads).
 *
 * LibOSSO does not come with a function for this, so we define one
 * ourselves.
 */
static const gchar* ossoErrorStr(osso_return_t errCode) {

    switch (errCode) {
        case OSSO_OK:
            return "No error (OSSO_OK)";
        case OSSO_ERROR:
            return "Some kind of error occurred (OSSO_ERROR)";
        case OSSO_INVALID:
            return "At least one parameter is invalid (OSSO_INVALID)";
        case OSSO_RPC_ERROR:
            return "Osso RPC method returned an error (OSSO_RPC_ERROR)";
        case OSSO_ERROR_NAME:
            return "(undocumented error) (OSSO_ERROR_NAME)";
        case OSSO_ERROR_NO_STATE:
            return "No state file found to read (OSSO_ERROR_NO_STATE)";
        case OSSO_ERROR_STATE_SIZE:
            return "Size of state file unexpected (OSSO_ERROR_STATE_SIZE)";
    }
}

```

```

        default:
            return "Unknown/Undefined";
    }
}

```

Listing 7.13: libosso-example-sync/libosso-rpc-sync.c

Decoding the RPC return value is, however, slightly more complex, as the return value is a structure containing a typed union (type is encoded in the type field of the structure):

```

/**
 * Utility to print out the type and content of given osso_rpc_t.
 * It also demonstrates the types available when using LibOSSO for
 * the RPC. Most simple types are available, but arrays are not
 * (unfortunately).
 */
static void printOssoValue(const osso_rpc_t* val) {

    g_assert(val != NULL);

    switch (val->type) {
        case DBUS_TYPE_BOOLEAN:
            g_print("boolean:%s", (val->value.b == TRUE)?"TRUE":"FALSE");
            break;
        case DBUS_TYPE_DOUBLE:
            g_print("double:%.3f", val->value.d);
            break;
        case DBUS_TYPE_INT32:
            g_print("int32:%d", val->value.i);
            break;
        case DBUS_TYPE_UINT32:
            g_print("uint32:%u", val->value.u);
            break;
        case DBUS_TYPE_STRING:
            g_print("string:'%s'", val->value.s);
            break;
        case DBUS_TYPE_INVALID:
            g_print("invalid/void");
            break;
        default:
            g_print("unknown(type=%d)", val->type);
            break;
    }
}

```

Listing 7.14: libosso-example-sync/libosso-rpc-sync.c

N.B. LibOSSO RPC functions do not support array parameters either, so there is a restriction: the used method calls can only have simple parameters.

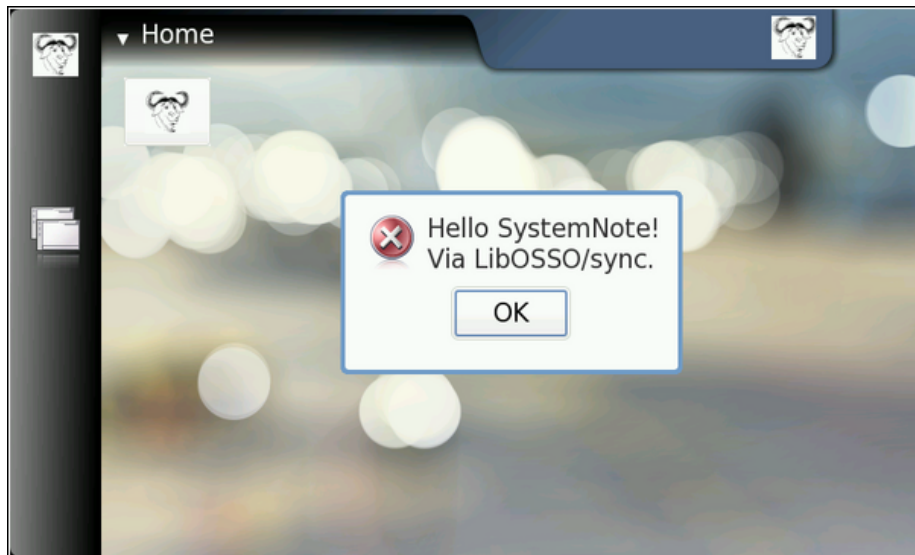
Now the example will be built and run. The end result is the now-familiar Note dialog.

```

[sbox-DIABLO_X86: ~/libosso-example-sync] > run-standalone.sh ./libosso-rpc-sync
Initializing LibOSSO
Invoking the method call
runRPC called
/dev/dsp: No such file or directory
RPC launched successfully
Method returns: uint32:8
runRPC ending
Shutting down LibOSSO
Quitting

```

The only difference is the location of the audio device error message. It will now appear before runRPC returns, since runRPC waits for RPC completion. This kind of ordering should never be relied on, because the RPC execution could also be delayed (and the message might appear at a later location when trying this program).



One point of interest in the **Makefile** (see section *GNU Make and Makefiles 4.2* in chapter *GNU Build System*) is the mechanism by which the ProgName define is set. It is often useful to separate the program name related information outside the source code, so that the code fragment may then be re-used more easily. In this case, there is control over the application name that is used when LibOSSO is initialized from the Makefile.

```
# define a list of pkg-config packages we want to use
pkg_packages := glib-2.0 libosso

# ... Listing cut for brevity ...

libosso-rpc-sync: libosso-rpc-sync.c
    $(CC) $(CFLAGS) -DProgName=\"LibOSSOExample\" \
    $< -o $@ $(LDFLAGS)
```

Listing 7.15: libosso-example-sync/Makefile

Asynchronous Method Calls with LibOSSO

Sometimes the method call will take long time to run (or one cannot be sure, whether it might take long time to run). In these cases, it is advisable to use the asynchronous RPC utility functions in LibOSSO, instead of the synchronous ones. The biggest difference is that the method call will be split into two parts: launching of the RPC, and handling its result in a callback function. The same limitations with respect to method parameter types and the number of return values still apply.

In order for the callback to use LibOSSO functions and control the mainloop object, it is necessary to create a small application state. The state will be passed to the callback, when necessary.

```
/**
 * Small application state so that we can pass both LibOSSO context
 * and the mainloop around to the callbacks.
 */
typedef struct {
    /* A mainloop object that will "drive" our example. */
    GMainLoop* mainloop;
    /* The LibOSSO context which we use to do RPC. */
    osso_context_t* ossoContext;
} ApplicationState;
```

Listing 7.16: libosso-example-async/libosso-rpc-async.c

The `osso_rpc_async_run` function is used to launch the method call, and it will normally return immediately. If it returns an error, it will be probably a client-side error (since the RPC method has not returned by then). The callback function to handle the RPC response will be registered with the function, as will the name-space related parameters and the method call arguments:

```
/**
 * We launch the RPC call from within a timer callback in order to
 * make sure that a mainloop object will be running when the RPC will
 * return (to avoid a nasty race condition).
 *
 * So, in essence this is a one-shot timer callback.
 *
 * In order to launch the RPC, it will need to get a valid LibOSSO
 * context (which is carried via the userData/application state
 * parameter).
 */
static gboolean launchRPC(gpointer userData) {

    ApplicationState* state = (ApplicationState*)userData;
    /* Message to display. */
    const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/async.";
    /* Icon type to use. */
    gint iconType = OSSO_GN_ERROR;
    /* Button label text to use. */
    const char* labelText = "Execute!";

    /* Will hold the result from the RPC launch call. */
    osso_return_t result;

    g_print("launchRPC called\n");

    g_assert(state != NULL);

    /*... Listing cut for brevity ...*/

    /* The only difference compared to the synchronous version is the
     * addition of the callback function parameter, and the user-data
     * parameter for data that will be passed to the callback. */
    result = osso_rpc_async_run(state->ossoContext,
                                SYSNOTE_NAME,      /* well-known name */
                                SYSNOTE_OPATH,      /* object path */
                                SYSNOTE_IFACE,      /* interface */
                                SYSNOTE_NOTE,       /* method name */
                                rpcCompletedCallback, /* async cb */
```

```

        state,          /* user-data for cb */
        /* The arguments for the RPC. */
        DBUS_TYPE_STRING, dispMsg,
        DBUS_TYPE_UINT32, iconType,
        DBUS_TYPE_STRING, labelText,
        DBUS_TYPE_INVALID);

/* Check whether launching the RPC succeeded (we don't know the
   result from the RPC itself). */
if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
           ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
}
g_print("RPC launched successfully\n");

g_print("launchRPC ending\n");

/* We only want to be called once, so ask the caller to remove this
   callback from the timer launch list by returning FALSE. */
return FALSE;
}

```

Listing 7.17: libosso-example-async/libosso-rpc-async.c

The return from the RPC method is handled by a simple callback function that will need to always use the same parameter prototype. It will receive the return value, as well as the interface and method names. The latter two are useful, as the same callback function can be used to handle returns from multiple different (and simultaneous) RPC method calls.

The return value structure is allocated by LibOSSO, and will be freed once the callback returns, so it does not need to be handled manually.

```

/**
 * Will be called from LibOSSO when the RPC return data is available.
 * Will print out the result, and return. Note that it must not free
 * the value, since it does not own it.
 *
 * The prototype (for reference) must be osso_rpc_async_f().
 *
 * The parameters for the callback are the D-Bus interface and method
 * names (note that object path and well-known name are NOT
 * communicated). The idea is that you can then reuse the same
 * callback to process completions from multiple simple RPC calls.
 */
static void rpcCompletedCallback(const gchar* interface,
                                const gchar* method,
                                osso_rpc_t* retVal,
                                gpointer userData) {

    ApplicationState* state = (ApplicationState*)userData;

    g_print("rpcCompletedCallback called\n");

    g_assert(interface != NULL);
    g_assert(method != NULL);
    g_assert(retVal != NULL);
    g_assert(state != NULL);

    g_print(" interface: %s\n", interface);
    g_print(" method: %s\n", method);
    /* NOTE If there is an error in the RPC delivery, the return value

```

```

will be a string. This is unfortunate if your RPC call is
supposed to return a string as well, since it is not
possible to differentiate between the two cases.

If a method returns "void", the type-field in the retVal
will be set to DBUS_TYPE_INVALID (it's not an error). */
g_print(" result: ");
printOssValue(retVal);
g_print("\n");

/* Tell the main loop to terminate. */
g_main_loop_quit(state->mainloop);

g_print("rpcCompletedCallback done\n");
}

```

Listing 7.18: libosso-example-async/libosso-rpc-async.c

In this case, receiving the response to the method call will cause the main program to be terminated.

The application set-up logic is covered next:

```

int main(int argc, char** argv) {

    /* Keep the application state in main's stack. */
    ApplicationState state = {};
    /* Keeps the results from LibOSSO functions for decoding. */
    osso_return_t result;
    /* Default timeout for RPC calls in LibOSSO. */
    gint rpcTimeout;

    g_print("Initializing LibOSSO\n");
    state.ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
    if (state.ossoContext == NULL) {
        g_error("Failed to initialize LibOSSO\n");
    }

    /* Print out the default timeout value (which we don't change, but
       could, with osso_rpc_set_timeout()). */
    result = osso_rpc_get_timeout(state.ossoContext, &rpcTimeout);
    if (result != OSSO_OK) {
        g_error("Error getting default RPC timeout (%s)\n",
                ossoErrorStr(result));
    }
    /* Interestingly the timeout seems to be -1, but is something else
       (by default). -1 probably then means that "no timeout has been
       set". */
    g_print("Default RPC timeout is %d (units)\n", rpcTimeout);

    g_print("Creating a mainloop object\n");
    /* Create a GMainLoop with default context and initial condition of
       not running (FALSE). */
    state.mainloop = g_main_loop_new(NULL, FALSE);
    if (state.mainloop == NULL) {
        g_error("Failed to create a GMainLoop\n");
    }

    g_print("Adding timeout to launch the RPC in one second\n");
    /* This could be replaced by g_idle_add(cb, &state), in order to
       guarantee that the RPC would be launched only after the mainloop
       has started. We opt for a timeout here (for no particular
       reason). */
}

```

```

g_timeout_add(1000, (GSourceFunc)launchRPC, &state);

g_print("Starting mainloop processing\n");
g_main_loop_run(state.mainloop);

g_print("Out of mainloop, shutting down LibOSSO\n");
/* Deinitialize LibOSSO. */
osso_deinitialize(state.ossoContext);
state.ossoContext = NULL;

/* Free GMainLoop as well. */
g_main_loop_unref(state.mainloop);
state.mainloop = NULL;

g_print("Quitting\n");
return EXIT_SUCCESS;
}

```

Listing 7.19: libosso-example-async/libosso-rpc-async.c

The code includes an example on how to query the method call timeout value as well; however, timeout values are left unchanged in the program.

The RPC method call is launched in a slightly unorthodox way, via a timeout call that will launch one second after the mainloop processing starts. One could just as easily use `g_idle_add`, as long as the launching itself is performed after the mainloop processing starts. Since the method return value callback will terminate the mainloop, the mainloop needs to be active at that point. The only way to guarantee this is to launch the RPC after the mainloop is active.

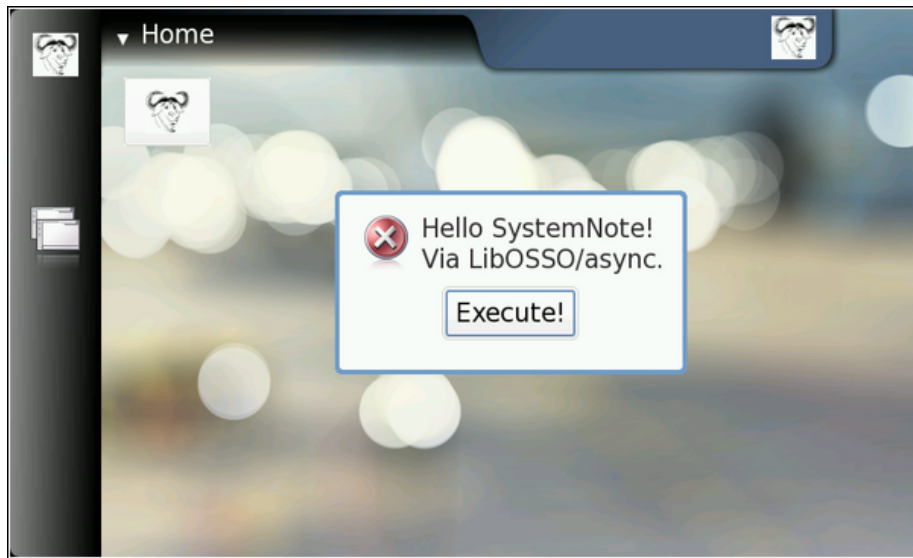
Testing the program yields few surprises (other than the default timeout value being -1):

```

[sbox-DIABLO_X86: ~/libosso-example-async] > run-standalone.sh ./libosso-rpc-async
Initializing LibOSSO
Default RPC timeout is -1 (units)
Creating a mainloop object
Adding timeout to launch the RPC in one second
Starting mainloop processing
launchRPC called
RPC launched successfully
launchRPC ending
rpcCompletedCallback called
interface: org.freedesktop.Notifications
method: SystemNoteDialog
result: uint32:10
rpcCompletedCallback done
Out of mainloop, shutting down LibOSSO
Quitting
/dev/dsp: No such file or directory

```

There is another shift to be noticed in the audio device error string. It is now displayed after all other messages (similar to the libdbus example). It seems that the audio playback is started "long after" the dialog itself is displayed, or maybe the method returns before `SystemNote` starts the dialog display. Again, one should not rely on exact timing, when dealing with D-Bus remote method calls.



End result of the async example (no surprises)

The label text was modified slightly, to test that non-default labels will work. The **Makefile** for this example does not contain anything new or special.

Device State and Mode Notifications

Since Internet Tablets are mobile devices, it is to be expected that people use them (and the software) while on the move, and also on airplanes and other places that might restrict network connectivity. If the program uses network connectivity, or needs to adapt to the conditions in the device better, it is necessary to handle changes between the different devices states. The changes between the states are normally initiated by the user of the device (when boarding an aircraft for example).

In order to demonstrate the handling of the most important device state, the next example implements a small utility program that will combine various utility functions from LibOSSO, as well as handle the changes in the device state. The state that is of particular interest is the "offline" mode. This mode is initiated by the user by switching the device into "Offline" mode.

The following application is a simple utility program that keeps the backlight of the device turned on by periodically asking the system to delay the automatic display dimming functionality. Normally the backlight is turned off after a short period of inactivity, although this setting can be changed by the user. It is the goal of the application then to request a postponement of this mechanism (by 60 seconds at a time). Here an internal timer frequency of 45 seconds is chosen, so that it can always extend the time by 60 seconds (and be sure that the opportunity is not missed due to using a lower frequency than the maximum).

The program will also track the device mode, and once the device enters the offline mode, the program will terminate. Should the program be started when the device is already in offline mode, the program will refuse to run.

Since the program has no GUI of its own, Note dialogs and the infoprint facility will be used to display status information to the user. The Note is used

to remind the user that leaving the program running will exhaust the battery. The infoprints are used when the application terminates, or if it refuses to run.

Most of the work required will be contained in the application set-up logic, which allows reducing the code in main significantly: (Please note that the 'Flight-Mode' in sample code means means 'Offline-Mode')

```
/**
 * Main program:
 *
 * 1) Setup application state
 * 2) Start mainloop
 * 3) Release application state & terminate
 */
int main(int argc, char** argv) {

    /* We'll keep one application state in our program and allocate
       space for it from the stack. */
    ApplicationState state = {};

    g_print(PROGNAME ":main Starting\n");
    /* Attempt to setup the application state and if something goes
       wrong, do cleanup here and exit. */
    if (setupAppState(&state) == FALSE) {
        g_print(PROGNAME ":main Setup failed, doing cleanup\n");
        releaseAppState(&state);
        g_print(PROGNAME ":main Terminating with failure\n");
        return EXIT_FAILURE;
    }

    g_print(PROGNAME ":main Starting mainloop processing\n");
    g_main_loop_run(state.mainloop);
    g_print(PROGNAME ":main Out of main loop (shutting down)\n");
    /* We come here when the application state has the running flag set
       to FALSE and the device state changed callback has decided to
       terminate the program. Display a message to the user about
       termination next. */
    displayExitMessage(&state, ProgName " exiting");

    /* Release the state and exit with success. */
    releaseAppState(&state);

    g_print(PROGNAME ":main Quitting\n");
    return EXIT_SUCCESS;
}
```

Listing 7.20: libosso-flashlight/flashlight.c

In order for the device state callbacks to force a quit of the application, the LibOSSO context needs to be passed to it. Also access to the mainloop object is needed, as well as utilizing a flag to tell when the timer should just quit (since timers cannot be removed externally in GLib). (Please note the 'Flight-Mode' in the example code means 'Offline-Mode')

```
/* Application state.

   Contains the necessary state to control the application lifetime
   and use LibOSSO. */
typedef struct {
    /* The GMainLoop that will run our code. */
    GMainLoop* mainloop;
    /* LibOSSO context that is necessary to use LibOSSO functions. */
```

```

osso_context_t* ossoContext;
/* Flag to tell the timer that it should stop running. Also utilized
   to tell the main program that the device is already in Flight-
   mode and the program shouldn't continue startup. */
gboolean running;
} ApplicationState;

```

Listing 7.21: libosso-flashlight/flashlight.c

All of the setup and start logic is implemented in `setupAppState`, and contains a significant number of steps that are all necessary: (Please note the 'Flight-Mode' in the example code means 'Offline-Mode')

```

/**
 * Utility to setup the application state.
 *
 * 1) Initialize LibOSSO (this will connect to D-Bus)
 * 2) Create a mainloop object
 * 3) Register the device state change callback
 *    The callback will be called once immediately on registration.
 *    The callback will reset the state->running to FALSE when the
 *    program needs to terminate so we'll know whether the program
 *    should run at all. If not, display an error dialog.
 *    (This is the case if the device will be in "Flight"-mode when
 *    the program starts.)
 * 4) Register the timer callback (which will keep the screen from
 *    blanking).
 * 5) Un-blank the screen.
 * 6) Display a dialog to the user (on the background) warning about
 *    battery drain.
 * 7) Send the first "delay backlight dimming" command.
 *
 * Returns TRUE when everything went ok, FALSE when caller should call
 * releaseAppState and terminate. The code below will print out the
 * errors if necessary.
 */
static gboolean setupAppState(ApplicationState* state) {

    osso_return_t result;

    g_assert(state != NULL);

    g_print(PROGNAME ":setupAppState starting\n");

    /* Zero out the state. Has the benefit of setting all pointers to
       NULLs and all gbooleans to FALSE. This is useful when we'll need
       to determine what to release later. */
    memset(state, 0, sizeof(ApplicationState));

    g_print(PROGNAME ":setupAppState Initializing LibOSSO\n");

    /*... Listing cut for brevity ...*/

    state->ossoContext = osso_initialize(OSSO_SERVICE, "1.0", FALSE, NULL
    );
    if (state->ossoContext == NULL) {
        g_printerr(PROGNAME ": Failed to initialize LibOSSO\n");
        return FALSE;
    }

    g_print(PROGNAME ":setupAppState Creating a GMainLoop object\n");
    /* Create a new GMainLoop object, with default context (NULL) and

```

```

        initial "running"-state set to FALSE. */
state->mainloop = g_main_loop_new(NULL, FALSE);
if (state->mainloop == NULL) {
    g_printerr(PROGNAME ": Failed to create a GMainLoop\n");
    return FALSE;
}

g_print(PROGNAME
        ":setupAppState Adding hw-state change callback.\n");
/* The callback will be called immediately with the state, so we
   need to know whether we're in offline mode to start with. If so,
   the callback will set the running-member to FALSE (and we'll
   check it below). */
state->running = TRUE;
/* In order to receive information about device state and changes
   in it, we register our callback here.

   Parameters for the osso_hw_set_event_cb():
   osso_context_t* : LibOSSO context object to use.
   osso_hw_state_t* : Pointer to a device state type that we're
                     interested in. NULL for "all states".
   osso_hw_cb_f* :   Function to call on state changes.
   gpointer :        User-data passed to callback. */
result = osso_hw_set_event_cb(state->ossoContext,
                              NULL, /* We're interested in all. */
                              deviceStateChanged,
                              state);

if (result != OSSO_OK) {
    g_printerr(PROGNAME
        ":setupAppState Failed to get state change CB\n");
    /* Since we cannot reliably know when to terminate later on
       without state information, we will refuse to run because of the
       error. */
    return FALSE;
}

/* We're in "Flight" mode? */
if (state->running == FALSE) {
    g_print(PROGNAME ":setupAppState In offline, not continuing.\n");
    displayExitMessage(state, ProgName " not available in Offline mode"
        );
    return FALSE;
}

g_print(PROGNAME ":setupAppState Adding blanking delay timer.\n");
if (g_timeout_add(45000,
                  (GSourceFunc)delayBlankingCallback,
                  state) == 0) {
    /* If g_timeout_add returns 0, it signifies an invalid event
       source id. This means that adding the timer failed. */
    g_printerr(PROGNAME ": Failed to create a new timer callback\n");
    return FALSE;
}

/* Un-blank the display (will always succeed in the SDK). */
g_print(PROGNAME ":setupAppState Unblanking the display\n");
result = osso_display_state_on(state->ossoContext);
if (result != OSSO_OK) {
    g_printerr(PROGNAME ": Failed in osso_display_state_on (%s)\n",
        ossoErrorStr(result));
    /* If the RPC call fails, odds are that nothing else will work
       either, so we decide to quit instead. */
    return FALSE;
}

```

```

}

/* Display a "Note"-dialog with a WARNING icon.
   The Dialog is MODAL, so user cannot do anything with the stylus
   until the Ok is selected, or the Back-key is pressed. */

/*... Listing cut for brevity ...*/

/* Other icons available:
   OSSO_GN_NOTICE: For general notices.
   OSSO_GN_WARNING: For warning messages.
   OSSO_GN_ERROR: For error messages.
   OSSO_GN_WAIT: For messages about "delaying" for something (an
   hourglass icon is displayed).
   5: Animated progress indicator. */

/*... Listing cut for brevity ...*/

g_print(PROGNAME ":setupAppState Displaying Note dialog\n");
result = osso_system_note_dialog(state->ossoContext,
/* UTF-8 text into the dialog */
"Started " ProgName ".\n"
"Please remember to stop it when you're done, "
"in order to conserve battery power.",
/* Icon to use */
OSSO_GN_WARNING,
/* We're not interested in the RPC
   return value. */
NULL);

if (result != OSSO_OK) {
    g_error(PROGNAME ": Error displaying Note dialog (%s)\n",
            ossoErrorStr(result));
}
g_print(PROGNAME ":setupAppState Requested for the dialog\n");

/* Then delay the blanking timeout so that our timer callback has a
   chance to run before that. */
delayDisplayBlanking(state);

g_print(PROGNAME ":setupAppState Completed\n");

/* State set up. */
return TRUE;
}

```

Listing 7.22: libosso-flashlight/flashlight.c

The callback to handle device state changes is registered with the `_hw_set_event_cb`, and it can also be seen how to force the backlight on (which is necessary so that the backlight dimming delay will accomplish something). Also the timer callback will be registered, and it will then start firing away after 45 seconds, and will keep delaying the backlight dimming and perform the first delay so that the backlight is not dimmed right away.

The callback function will always receive the new "hardware state" as well as the user data. It is also somewhat interesting to note that just by registering the callback, it will be triggered immediately. This will happen even before starting the mainloop in order to tell the application the initial state of the device, when the application starts. This can be utilized to determine whether the device is already in offline mode, and keep from starting the program if that is the case. Since it cannot always be known whether the mainloop is active (the callback

can be triggered later on as well), the program also utilizes an additional flag to communicate the timer callback that it should eventually quit. (Please note the 'Flight-Mode' in the example code means 'Offline-Mode')

```
/**
 * This callback will be called by LibOSSO whenever the device state
 * changes. It will also be called right after registration to inform
 * the current device state.
 *
 * The device state structure contains flags telling about conditions
 * that might affect applications, as well as the mode of the device
 * (for example telling whether the device is in "in-flight"-mode).
 */
static void deviceStateChanged(osso_hw_state_t* hwState,
                               gpointer data) {
    ApplicationState* state = (ApplicationState*)data;

    g_print(PROGNAME ":deviceStateChanged Starting\n");

    printDeviceState(hwState);

    /* If device is in/going into "flight-mode" (called "Offline" on
     some devices), we stop our operation automatically. Obviously
     this makes flashlight useless (as an application) if someone gets
     stuck in a dark cargo bay of a plane with snakes.. But we still
     need a way to shut down the application and react to device
     changes, and this is the easiest state to test with.

     Note that since offline mode will terminate network connections,
     you will need to test this on the device itself, not over ssh. */
    if (hwState->sig_device_mode_ind == OSSO_DEVMODE_FLIGHT) {
        g_print(PROGNAME ":deviceStateChanged In/going into offline.\n");

        /* Terminate the mainloop.
         NOTE: Since this callback is executed immediately on
         registration, the mainloop object is not yet "running",
         hence calling quit on it will be ineffective! _quit only
         works when the mainloop is running. */
        g_main_loop_quit(state->mainloop);
        /* We also set the running to correct state to fix the above
         problem. */
        state->running = FALSE;
    }
}
```

Listing 7.23: libosso-flashlight/flashlight.c

The printDeviceState is a utility function to decode the device state structure that the callback will be invoked with. The state contains the device mode, but also gboolean flags, which tell the application to adapt to the environment in other ways (like memory pressure and other indicators): (Please note the 'Flight-Mode' in the example code means 'Offline-Mode')

```
/* Small macro to return "YES" or "no" based on given parameter.
   Used in printDeviceState below. YES is in capital letters in order
   for it to "stand out" in the program output (since it's much
   rarer). */
#define BOOLSTR(p) ((p)?"YES":"no")

/**
 * Utility to decode the hwstate structure and print it out in human
 * readable format. Mainly useful for debugging on the device.
```

```

*
* The mode constants unfortunately are not documented in LibOSSO.
*/
static void printDeviceState(osso_hw_state_t* hwState) {

    gchar* modeStr = "Unknown";

    g_assert(hwState != NULL);

    switch(hwState->sig_device_mode_ind) {
        case OSSO_DEVMODE_NORMAL:
            /* Non-flight-mode. */
            modeStr = "Normal";
            break;
        case OSSO_DEVMODE_FLIGHT:
            /* Power button -> "Offline mode". */
            modeStr = "Flight";
            break;
        case OSSO_DEVMODE_OFFLINE:
            /* Unknown. Even if all connections are severed, this mode will
             not be triggered. */
            modeStr = "Offline";
            break;
        case OSSO_DEVMODE_INVALID:
            /* Unknown. */
            modeStr = "Invalid(?)";
            break;
        default:
            /* Leave at "Unknown". */
            break;
    }
    g_print(
        "Mode: %s, Shutdown: %s, Save: %s, MemLow: %s, RedAct: %s\n",
        modeStr,
        /* Set to TRUE if the device is shutting down. */
        BOOLSTR(hwState->shutdown_ind),
        /* Set to TRUE if our program has registered for autosave and
         now is the moment to save user data. */
        BOOLSTR(hwState->save_unsaved_data_ind),
        /* Set to TRUE when device is running low on memory. If possible,
         memory should be freed by our program. */
        BOOLSTR(hwState->memory_low_ind),
        /* Set to TRUE when system wants us to be less active. */
        BOOLSTR(hwState->system_inactivity_ind));
}

```

Listing 7.24: libosso-flashlight/flashlight.c

The delaying of the display blanking is achieved with a utility function of LibOSSO (`osso_display_blanking_pause`), which is implemented in a separate function since it is called from multiple places:

```

/**
 * Utility to ask the device to pause the screen blanking timeout.
 * Does not return success/status.
 */
static void delayDisplayBlanking(ApplicationState* state) {

    osso_return_t result;

    g_assert(state != NULL);

```

```

result = osso_display_blanking_pause(state->ossoContext);
if (result != OSSO_OK) {
    g_printerr(PROGNAME ":delayDisplayBlanking. Failed (%s)\n",
               ossoErrorStr(result));
    /* But continue anyway. */
} else {
    g_print(PROGNAME ":delayDisplayBlanking RPC succeeded\n");
}
}

```

Listing 7.25: libosso-flashlight/flashlight.c

The timer callback will normally just ask the blanking delay to be further extended, but will also check whether the program is shutting down (by using the running field in the application state). If the application is indeed shutting down, the timer will ask itself to be removed from the timer queue by returning FALSE:

```

/**
 * Timer callback that will be called within 45 seconds after
 * installing the callback and will ask the platform to defer any
 * display blanking for another 60 seconds.
 *
 * This will also prevent the device from going into suspend
 * (according to LibOSSO documentation).
 *
 * It will continue doing this until the program is terminated.
 *
 * NOTE: Normally timers shouldn't be abused like this since they
 * bring the CPU out from power saving state. Because the screen
 * backlight dimming might be activated again after 60 seconds
 * of receiving the "pause" message, we need to keep sending the
 * "pause" messages more often than every 60 seconds. 45 seconds
 * seems like a safe choice.
 */
static gboolean delayBlankingCallback(gpointer data) {
    ApplicationState* state = (ApplicationState*)data;

    g_print(PROGNAME ":delayBlankingCallback Starting\n");

    g_assert(state != NULL);
    /* If we're not supposed to be running anymore, return immediately
       and ask caller to remove the timeout. */
    if (state->running == FALSE) {
        g_print(PROGNAME ":delayBlankingCallback Removing\n");
        return FALSE;
    }

    /* Delay the blanking for a while still (60 seconds). */
    delayDisplayBlanking(state);

    g_print(PROGNAME ":delayBlankingCallback Done\n");

    /* We want the same callback to be invoked from the timer
       launch again, so we return TRUE. */
    return TRUE;
}

```

Listing 7.26: libosso-flashlight/flashlight.c

There is also a small utility function that will be used to display an exit message (there are two ways of exiting):

```

/**
 * Utility to display an "exiting" message to user using infoprint.
 */
static void displayExitMessage(ApplicationState* state,
                               const gchar* msg) {

    osso_return_t result;

    g_assert(state != NULL);

    g_print(PROGNAME ":displayExitMessage Displaying exit message\n");
    result = osso_system_note_infoprint(state->ossoContext, msg, NULL);
    if (result != OSSO_OK) {
        /* This is rather harsh, since we terminate the whole program if
         the infoprint RPC fails. It is used to display messages at
         program exit anyway, so this isn't a critical issue. */
        g_error(PROGNAME ": Error doing infoprint (%s)\n",
                ossoErrorStr(result));
    }
}

```

Listing 7.27: libosso-flashlight/flashlight.c

And finally comes the application state tear-down function, which will release all the resources that have been allocated by the set-up function.

```

/**
 * Release all resources allocated by setupAppState in reverse order.
 */
static void releaseAppState(ApplicationState* state) {

    g_print(PROGNAME ":releaseAppState starting\n");

    g_assert(state != NULL);

    /* First set the running state to FALSE so that if the timer will
     (for some reason) be launched, it will remove itself from the
     timer call list. This shouldn't be possible since we are running
     only with one thread. */
    state->running = FALSE;

    /* Normally we would also release the timer, but since the only way
     to do that is from the timer callback itself, there's not much we
     can do about it here. */

    /* Remove the device state change callback. It is possible that we
     run this even if the callback was never installed, but it is not
     harmful. */
    if (state->ossoContext != NULL) {
        osso_hw_unset_event_cb(state->ossoContext, NULL);
    }

    /* Release the mainloop object. */
    if (state->mainloop != NULL) {
        g_print(PROGNAME ":releaseAppState Releasing mainloop object.\n");
        g_main_loop_unref(state->mainloop);
        state->mainloop = NULL;
    }

    /* Lastly, free up the LibOSSO context. */
    if (state->ossoContext != NULL) {
        g_print(PROGNAME ":releaseAppState De-init LibOSSO.\n");
    }
}

```



```

    osso_deinitialize(state->ossoContext);
    state->ossoContext = NULL;
}
/* All resources released. */
}

```

Listing 7.28: libosso-flashlight/flashlight.c

The Makefile for this program contains no surprises, so it is not shown here. Now the program is built, and run in the SDK:

```

[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh ./flashlight
flashlight:main Starting
flashlight:setupAppState starting
flashlight:setupAppState Initializing LibOSSO
flashlight:setupAppState Creating a GMainLoop object
flashlight:setupAddState Adding hw-state change callback.
flashlight:deviceStateChanged Starting
Mode: Normal, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:setupAppState Adding blanking delay timer.
flashlight:setupAppState Unblanking the display
flashlight:setupAppState Displaying Note dialog
flashlight:setupAppState Requested for the dialog
flashlight:delayDisplayBlanking RPC succeeded
flashlight:setupAppState Completed
flashlight:main Starting mainloop processing
/dev/dsp: No such file or directory
flashlight:delayBlankingCallback Starting
flashlight:delayDisplayBlanking RPC succeeded
flashlight:delayBlankingCallback Done
...

```



The application will display a modal dialog when it starts, to remind the user of the ramifications.

Since the SDK does not contain indications about backlight operations, it will not be noticeable that the application is running in the screen. From the debugging messages, it can be seen that it is. It just takes 45 seconds between each timer callback launch (and for new debug messages to appear).

Simulating Device Mode Changes

In order to test the flashlight application without requiring a device, it is useful to know how to simulate device mode changes in the SDK. From the standpoint of LibOSSO (and programs that use it), it will feel and look exactly as it does on a device. When LibOSSO receives the D-Bus signal, it will be exactly the same signal as it would be on a device. D-Bus signals are covered more thoroughly later on.

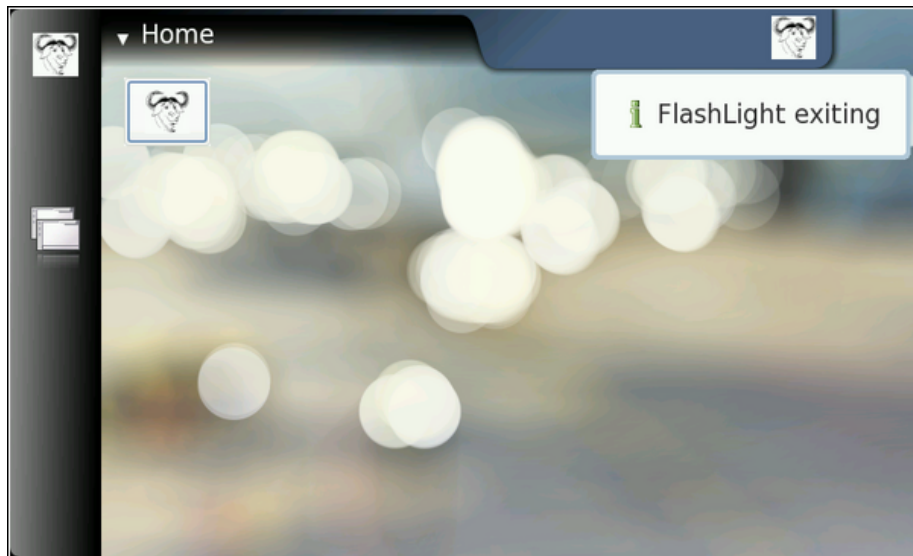
To see how this works, start flashlight in one session and leave it running (you might want to dismiss the modal dialog). Then open another session and send the signal using the system bus that signifies a device mode change (below).

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh \
dbus-send --system --type=signal /com/nokia/mce/signal \
com.nokia.mce.signal.sig_device_mode_ind string:'offline'
```

The argument for the signal is a string, stating the new device mode. They are defined (for LibOSSO) in the LibOSSO source code (src/osso-hw.c), which can be downloaded with `apt-get source libosso`. The start of that file also defines other useful D-Bus well-known names, object paths and interfaces, as well as method names relating to hardware and system-wide conditions. This example covers only switching the device mode from 'normal' to 'offline' and then back (see below).

```
flashlight:delayBlankingCallback Starting
flashlight:delayDisplayBlanking RPC succeeded
flashlight:delayBlankingCallback Done
...
flashlight:deviceStateChanged Starting
Mode: Flight, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:deviceStateChanged In/going into offline.
flashlight:main Out of main loop (shutting down)
flashlight:displayExitMessage Displaying exit message
flashlight:releaseAppState starting
flashlight:releaseAppState Releasing mainloop object.
flashlight:releaseAppState De-init LibOSSO.
flashlight:main Quitting
```

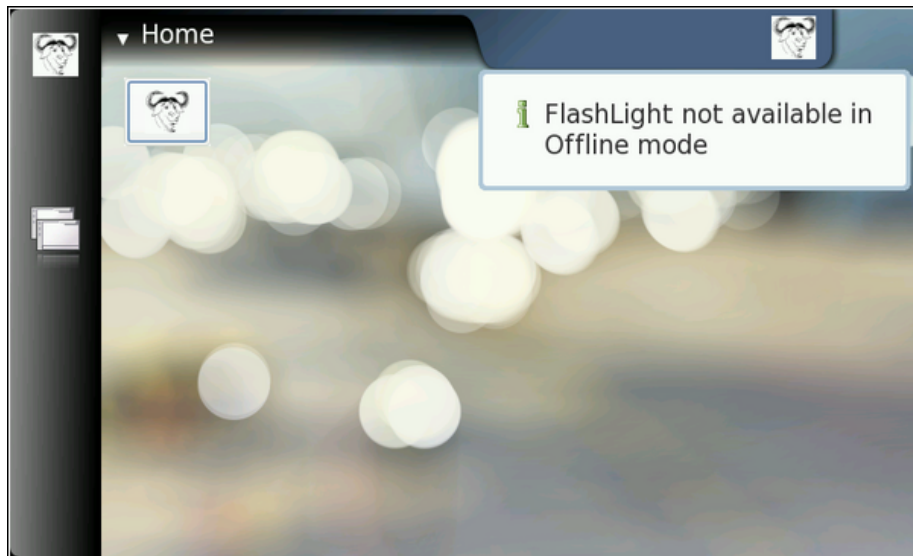
Once the signal is sent, it will eventually be converted into a callback call from LibOSSO, and the `deviceStateChanged` function gets to run. It will notice that the device mode is now `OFFLINE_MODE`, and shutdown flashlight.



If the flashlight is started again, something peculiar can be noticed. It will see that the device is still in offline mode. How convenient! This allows testing the rest of the code paths remaining in the application (when it refuses to start if the device is already in offline mode).

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh ./flashlight
flashlight:main Starting
flashlight:setupAppState starting
flashlight:setupAppState Initializing LibOSSO
flashlight:setupAppState Creating a GMainLoop object
flashlight:setupAppState Adding hw-state change callback.
flashlight:deviceStateChanged Starting
Mode: Flight, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:deviceStateChanged In/going into offline.
flashlight:setupAppState In offline, not continuing.
flashlight:displayExitMessage Displaying exit message
flashlight:main Setup failed, doing cleanup
flashlight:releaseAppState starting
flashlight:releaseAppState Releasing mainloop object.
flashlight:releaseAppState De-init LibOSSO.
flashlight:main Terminating with failure
```

In this case, the user is displayed the cause why flashlight cannot be started, since it would be quite rude not to display any feedback to the user.



In order to return the device back to normal mode, it needs to be sent the same signal as before (sig_device_mode_ind), but with the argument normal.

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh \
dbus-send --system --type=signal /com/nokia/mce/signal \
com.nokia.mce.signal.sig_device_mode_ind string:'normal'
```

7.3.3 Using GLib Wrappers for D-Bus

Introduction to GObject

In order to support runtime binding of GTK+ widgets to interpreted languages, a somewhat complicated system for implementing object-oriented machinery for C was developed. Depending on the particular viewpoint, this system is either called GObject or GType. GType is the low-level runtime type system, which is used to implement GObject, and GObject are the implementations of objects using the GType framework. For a short introduction about GObject, please see the [Wikipedia entry on GType](#). GObject/GType is part of GLib, but one might note that most of GLib is useable without the GType part of the library. In fact, the GObject/GType functionality is separated into its own library (libgobject).

The following example will use the GType in order to implement a very simple object that will be published over the D-Bus. This also means that some of the inherent complexity involved in implementing a fully fledged GObject can be forgone. For this reason, while the object will be usable over the D-Bus, it might not have all the support required to use it directly as a GObject (full dynamic type registration and properties are missing).

The implementation will be a simple non-inheriting stand-alone class, implementing an interface to access and modify two private members: value1 and value2, first of which is an 32-bit signed integer, and the second a gdouble.

It will be necessary to implement the per-class constructor as well as the per-object constructor. Both of these will be quite short for the first version of the implementation.

D-Bus Interface Definition Using XML

Since the primary objective here is to make the object available over D-Bus, the example will start by covering one of the easiest way of achieving this: the `dbus-bindings-tool`. The tool will generate a lot of the bindings code for both the client and server side. As its input, it uses an XML file describing the interface for the service that is being implemented.

The first step is to describe one method in XML. Each method is described with a separate method element, whose name attribute is the name of the method to be generated (this name will be copied into the generated stub code automatically by the tool). The first method is `setvalue1`, which will get one argument, `new_value`, which is an 32-bit signed integer:

```
<!-- setvalue1(int newValue): sets value1 -->
<method name="setvalue1">
  <arg type="i" name="new_value" direction="in"/>
</method>
```

Listing 7.29: `glib-dbus-sync/value-dbus-interface.xml`

Each argument needs to be defined explicitly with the `arg` element. The `type` attribute is required, since it will define the data type for the argument. Arguments are sometimes called parameters, when used with D-Bus methods. Each argument needs to specify the "direction" of the argument. Parameters for method calls are "going into" the service, hence the correct content for the `direction` attribute is `in`. Return values from method calls are "coming out" of the service. Hence, their `direction` will be `out`. If a method call does not return any value (returns void), then no argument with the `direction out` needs to be specified.

It should also be noted that D-Bus by itself does not limit the number of return arguments. C language supports only one return value from a function, but a lot of the higher level languages do not have this restriction.

The following argument types are supported for D-Bus methods (with respective closest types in GLib):

- `b`: boolean (`gboolean`)
- `y`: 8-bit unsigned integer (`guint8`)
- `q/n`: 16-bit unsigned/signed integer (`guint16/gint16`)
- `u/i`: 32-bit unsigned/signed integer (`guint32/gint32`)
- `t/x`: 64-bit unsigned/signed integer (`guint64/gint64`)
- `d`: IEEE 754 double precision floating point number (`gdouble`)
- `s`: UTF-8 encoded text string with NUL termination (only one NUL allowed) (`gchar*` with additional restrictions)
- `a`: Array of the following type specification (case-dependent)
- `o/g/r/(/)/v/e//`: Complex types, please see the [official D-Bus documentation on type signatures](#).

From the above list, it can be seen that `setValue1` will accept one 32-bit signed integer argument (`newValue`). The name of the argument will affect the generated stub code prototypes (not the implementation), but is quite useful for documentation, and also for D-Bus introspection (which will be covered later).

The next step is the interface specification of another method: `getValue1`, which will return the current integer value of the object. It has no method call parameters (no arguments with `direction="in"`), and only returns one 32-bit signed integer:

```
<!-- getValue1(): returns the first value (int) -->
<method name="getValue1">
  <arg type="i" name="cur_value" direction="out"/>
</method>
```

Listing 7.30: `glib-dbus-sync/value-dbus-interface.xml`

Naming of the return arguments is also supported in D-Bus (as above). This will not influence the generated stub code, but serves as additional documentation.

The methods need to be bound to a specific (D-Bus) interface, and this is achieved by placing the method elements within an interface element. The interface name attribute is optional, but very much recommended (otherwise the interface becomes unnamed, and provides less useful information on introspection).

Multiple interfaces can be implemented in the same object, and if this would be the case, the multiple interface elements would be listed within the node element. The node element is the "top-level" element in any case. In this case, only one explicit interface is implemented (the binding tools will add the introspection interfaces automatically, so specifying them is not necessary in the XML). And so, the result is the minimum required interface XML file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<node>
  <interface name="org.maemo.Value">
    <!-- getValue1(): returns the first value (int) -->
    <method name="getValue1">
      <arg type="i" name="cur_value" direction="out"/>
    </method>
    <!-- setValue1(int newValue): sets value1 -->
    <method name="setValue1">
      <arg type="i" name="new_value" direction="in"/>
    </method>
  </interface>
</node>
```

Listing 7.31: `glib-dbus-sync/value-dbus-interface.xml`

The minimal interface specification is then extended by adding the correct reference to the proper DTD. This will allow validation tools to work automatically with the XML file. Methods are also added to manipulate the second value. The full interface file will now contain comments, describing the purpose of the interface and the methods. This is highly recommended, if planning to publish the interface at some point, as the bare XML does not carry semantic information.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```

<!-- 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. --
>

<!-- If you keep the following DOCTYPE tag in your interface
      specification, xmllint can fetch the DTD over the Internet
      for validation automatically. -->
<!DOCTYPE node PUBLIC
  "-//freedesktop//DTD D-Bus Object Introspection 1.0//EN"
  "http://standards.freedesktop.org/dbus/1.0/introspect.dtd">

<!-- This file defines the D-Bus interface for a simple object, that
      will hold a simple state consisting of two values (one a 32-bit
      integer, the other a double).

      The interface name is "org.maemo.Value".
      One known reference implementation is provided for it by the
      "/GlobalValue" object found via a well-known name of
      "org.maemo.Platdev_ex". -->

<node>
  <interface name="org.maemo.Value">

    <!-- Method definitions -->

    <!-- getvalue1(): returns the first value (int) -->
    <method name="getvalue1">
      <!-- NOTE Naming arguments is not mandatory, but is recommended
           so that D-Bus introspection tools are more useful.
           Otherwise the arguments will be automatically named
           "arg0", "arg1" and so on. -->
      <arg type="i" name="cur_value" direction="out"/>
    </method>

    <!-- getvalue2(): returns the second value (double) -->
    <method name="getvalue2">
      <arg type="d" name="cur_value" direction="out"/>
    </method>

    <!-- setvalue1(int newValue): sets value1 -->
    <method name="setvalue1">
      <arg type="i" name="new_value" direction="in"/>
    </method>

    <!-- setvalue2(double newValue): sets value2 -->
    <method name="setvalue2">
      <arg type="d" name="new_value" direction="in"/>
    </method>

  </interface>
</node>

```

Listing 7.32: glib-dbus-sync/value-dbus-interface.xml

When dealing with automatic code generation, it is quite useful to also automate testing of the "source files" (in this case, XML). One important validation technique for XML is verifying for well-formedness (all XML files need to satisfy the rules in XML spec 1.0). Another is validating the structure of XML (that elements are nested correctly, that only correct elements are present and

that the element attributes and data are legal). Structural validation rules are described by a DTD (Document Type Definition) document for the XML format that the file is supposed to adhere to. The DTD is specified in the XML, within the DOCTYPE processing directive.

This is still not perfect, as DTD validation can only check for syntax and structure, but not meaning or semantics.

The next step is to add a target called `checkxml` to the **Makefile**, so that it can be run whenever the validity of the interface XML is to be checked.

```
# One extra target (which requires xmllint, from package libxml2-utils)
# is available to verify the well-formedness and the structure of the
# interface definition xml file.
#
# Use the 'checkxml' target to run the interface XML through xmllint
# verification. You will need to be connected to the Internet in order
# for xmllint to retrieve the DTD from fd.o (unless you setup local
# catalogs, which are not covered here).

# ... Listing cut for brevity ...

# Interface XML name (used in multiple targets)
interface_xml := value-dbus-interface.xml

# ... Listing cut for brevity ...

# Special target to run DTD validation on the interface XML. Not run
# automatically (since xmllint is not always available and also needs
# Internet connectivity).
checkxml: $(interface_xml)
    @xmllint --valid --noout $<
    @echo $< checks out ok
```

Listing 7.33: glib-dbus-sync/Makefile

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make checkxml
value-dbus-interface.xml checks out ok
```

Just to demonstrate what kind of error messages to expect when there are problems in the XML, the valid interface specification is modified slightly by adding one invalid element (`invalidElement`), and by removing one starting tag (`method`).

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make checkxml
value-dbus-interface.xml:36: element invalidElement: validity error :
  No declaration for element invalidElement
  </invalidElement>
  ^
value-dbus-interface.xml:53: parser error :
  Opening and ending tag mismatch: method line 39 and interface
  </interface>
  ^
value-dbus-interface.xml:54: parser error :
  Opening and ending tag mismatch: interface line 22 and node
  </node>
  ^
value-dbus-interface.xml:55: parser error :
  Premature end of data in tag node line 21
  ^
make: *** [checkxml] Error 1
```

The first error (validity error) is detected, because the file does not adhere to the DTD. The other errors (parser errors) are detected, because the file is no longer

a well-formed XML document.

If the makefile targets depend on checkxml, the validation can be integrated into the process of the build. However, as was noted before, it might not be always the best solution.

Generating Automatic Stub Code

Now the following step is to generate the "glue" code that will implement the mapping from GLib into D-Bus. The generated code will be used later on, but it is instructive to see what the dbus-binding-tool program generates.

The **Makefile** will be expanded to invoke the tool whenever the interface XML changes, and store the resulting glue code separately for both the client and server.

```
# Define a list of generated files so that they can be cleaned as well
cleanfiles := value-client-stub.h \
              value-server-stub.h

# ... Listing cut for brevity ...

# If the interface XML changes, the respective stub interfaces will be
# automatically regenerated. Normally this would also mean that your
# builds would fail after this since you would be missing
# implementation
# code.
value-server-stub.h: $(interface_xml)
    dbus-binding-tool --prefix=value_object --mode=glib-server \
    $< > $@

value-client-stub.h: $(interface_xml)
    dbus-binding-tool --prefix=value_object --mode=glib-client \
    $< > $@

# ... Listing cut for brevity ...

clean:
    $(RM) $(targets) $(cleanfiles) *.o
```

Listing 7.34: glib-dbus-sync/Makefile

Two parameters are passed for the dbus-binding-tool program. The `--prefix` parameter is used to tell what text should be prefixed to all generated structure and function names. This will help avoid namespace collisions, when pulling the generated glue files back into the programs. The `value_object` will be used, since it seems like a logical prefix for the project. It is advisable to use a prefix that is not used in the code (even in the object implementation in server). This way, there is no risk of reusing the same names that are generated with the tool.

The second parameter will select what kind of output the tool will generate. At the moment, the tool only supports generating GLib/D-Bus bindings, but this might change in the future. Also, it is necessary to select which "side" of the D-Bus the bindings will be generated for. The `-client` side is for code that wishes to use GLib to access the Value object implementation over D-Bus. The `-server` side is respectively for the implementation of the Value object.

Running the tool will result in the two header files:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make value-server-stub.h value-client-stub.h
dbus-binding-tool --prefix=value_object --mode=glib-server \
value-dbus-interface.xml > value-server-stub.h
dbus-binding-tool --prefix=value_object --mode=glib-client \
value-dbus-interface.xml > value-client-stub.h
[sbox-DIABLO_X86: ~/glib-dbus-sync] > ls -la value*stub.h
-rw-rw-r-- 1 user user 5184 Nov 21 14:02 value-client-stub.h
-rw-rw-r-- 1 user user 10603 Nov 21 14:02 value-server-stub.h
```

The object implementation will be handled in a bit, but first it should be checked what the tool produced, starting with the server stub file:

```
/* Generated by dbus-binding-tool; do not edit! */

/*... Listing cut for brevity ...*/

#include <dbus/dbus-glib.h>
static const DBusGMethodInfo dbus_glib_value_object_methods[] = {
    { (GCallback) value_object_getvalue1,
      dbus_glib_marshal_value_object_BOOLEAN__POINTER_POINTER, 0 },
    { (GCallback) value_object_getvalue2,
      dbus_glib_marshal_value_object_BOOLEAN__POINTER_POINTER, 47 },
    { (GCallback) value_object_setvalue1,
      dbus_glib_marshal_value_object_BOOLEAN__INT_POINTER, 94 },
    { (GCallback) value_object_setvalue2,
      dbus_glib_marshal_value_object_BOOLEAN__DOUBLE_POINTER, 137 },
};

const DBusGObjectInfo dbus_glib_value_object_object_info = {
    0,
    dbus_glib_value_object_methods,
    4,
    "org.maemo.Value\0getvalue1\0S\0cur_value\00\0F\0N\0i\0\0",
    "org.maemo.Value\0getvalue2\0S\0cur_value\00\0F\0N\0d\0\0",
    "org.maemo.Value\0setvalue1\0S\0new_value\0I\0i\0\0",
    "org.maemo.Value\0setvalue2\0S\0new_value\0I\0d\0\0\0",
    "\0",
    "\0",
};
```

Listing 7.35: glib-dbus-sync/value-server-stub.h

The interest here lies in the method table, mainly because it lists the names of the functions that are needed to be implemented: `value_object_getvalue1`, `_object_getvalue2`, `value_object_setvalue1` and `value_object_setvalue2`. Each entry in the table consists of a function address, and the function to use to marshal data from/to GLib/D-Bus (the functions that start with `dbus_glib_marshal_*`). The marshaling functions are defined in the same file, but were omitted from the listing above.

Marshaling in its most generic form means the conversion of parameters or arguments from one format to another, in order to make two different parameter passing conventions compatible. It is a common feature found in almost all RPC mechanisms. Since GLib has its own type system (which will be shown shortly) and D-Bus its own, it would be very tedious to write the conversion code manually. This is where the binding generation tool really helps.

The other interesting feature of the above listing is the `_object_info` structure. It will be passed to the D-Bus daemon, when the object is ready to be published on the bus (so that clients may invoke methods on it). The very long string (that contains binary zeroes) is the compact format of the interface specification. Similarities can be seen between the names in the string and the names of the

interface, methods and arguments that were declared in the XML. It is also an important part of the D-Bus introspection mechanism, which will be covered at the end of this chapter.

As the snippet says at the very first line, it should never be edited manually. This holds true while using the XML file as the source of an interface. It is also possible to use the XML only once, when starting the project, and then just start copy-pasting the generated glue code around, while discarding the XML file and dbus-binding-tool. Needless to say, this makes maintenance of the interface much more difficult and is not really recommended. The generated stub code will not be edited in this material.

The example will next continue with the server implementation for the functions that are called via the method table.

Creating Simple GObject for D-Bus

The starting point here is with the per-instance and per-class state structures for the object. The per-class structure contains only the bare minimum contents, which are required from all classes in GObject. The per-instance structure contains the required "parent object" state (GObject), but also includes the two internal values (value1 and value2), with which the rest of this example will be concerned:

```
/* This defines the per-instance state.

Each GObject must start with the 'parent' definition so that common
operations that all GObjects support can be called on it. */
typedef struct {
    /* The parent class object state. */
    GObject parent;
    /* Our first per-object state variable. */
    gint value1;
    /* Our second per-object state variable. */
    gdouble value2;
} ValueObject;

/* Per class state.

For the first Value implementation we only have the bare minimum,
that is, the common implementation for any GObject class. */
typedef struct {
    /* The parent class state. */
    GObjectClass parent;
} ValueObjectClass;
```

Listing 7.36: glib-dbus-sync/server.c

Then convenience macros will be defined in a way expected for all GTypes. The G_TYPE_-macros are defined in GType, and include the magic by which the object implementation does not need to know much about the internal specifics of GType. The GType macros are described in the GObject API reference for GType at [57].

Some of the macros will be used internally in this implementation later on.

```
/* Forward declaration of the function that will return the GType of
the Value implementation. Not used in this program since we only
need to push this over the D-Bus. */
GType value_object_get_type(void);
```

```

/* Macro for the above. It is common to define macros using the
   naming convention (seen below) for all GType implementations,
   and that is why we are going to do that here as well. */
#define VALUE_TYPE_OBJECT (value_object_get_type())

#define VALUE_OBJECT(object) \
    (G_TYPE_CHECK_INSTANCE_CAST((object), \
    VALUE_TYPE_OBJECT, ValueObject))
#define VALUE_OBJECT_CLASS(klass) \
    (G_TYPE_CHECK_CLASS_CAST((klass), \
    VALUE_TYPE_OBJECT, ValueObjectClass))
#define VALUE_IS_OBJECT(object) \
    (G_TYPE_CHECK_INSTANCE_TYPE((object), \
    VALUE_TYPE_OBJECT))
#define VALUE_IS_OBJECT_CLASS(klass) \
    (G_TYPE_CHECK_CLASS_TYPE((klass), \
    VALUE_TYPE_OBJECT))
#define VALUE_OBJECT_GET_CLASS(obj) \
    (G_TYPE_INSTANCE_GET_CLASS((obj), \
    VALUE_TYPE_OBJECT, ValueObjectClass))

/* Utility macro to define the value_object GType structure. */
G_DEFINE_TYPE(ValueObject, value_object, G_TYPE_OBJECT)

```

Listing 7.37: glib-dbus-sync/server.c

After the macros, the next phase contains the instance initialization and class initialization functions, of which the class initialization function contains the integration call into GLib/D-Bus:

```

/**
 * Since the stub generator will reference the functions from a call
 * table, the functions must be declared before the stub is included.
 */
gboolean value_object_getvalue1(ValueObject* obj, gint* value_out,
                                GError** error);
gboolean value_object_getvalue2(ValueObject* obj, gdouble* value_out,
                                GError** error);
gboolean value_object_setvalue1(ValueObject* obj, gint value_in,
                                GError** error);
gboolean value_object_setvalue2(ValueObject* obj, gdouble value_in,
                                GError** error);

/**
 * Pull in the stub for the server side.
 */
#include "value-server-stub.h"

/*... Listing cut for brevity ...*/

/**
 * Per object initializer
 *
 * Only sets up internal state (both values set to zero)
 */
static void value_object_init(ValueObject* obj) {
    dbg("Called");

    g_assert(obj != NULL);

    obj->value1 = 0;

```

```

    obj->value2 = 0.0;
}

/**
 * Per class initializer
 *
 * Registers the type into the GLib/D-Bus wrapper so that it may add
 * its own magic.
 */
static void value_object_class_init(ValueObjectClass* klass) {

    dbg("Called");

    g_assert(klass != NULL);

    dbg("Binding to GLib/D-Bus");

    /* Time to bind this GType into the GLib/D-Bus wrappers.
     NOTE: This is not yet "publishing" the object on the D-Bus, but
     since it is only allowed to do this once per class
     creation, the safest place to put it is in the class
     initializer.
     Specifically, this function adds "method introspection
     data" to the class so that methods can be called over
     the D-Bus. */
    dbus_g_object_type_install_info(VALUE_TYPE_OBJECT,
                                    &dbus_glib_value_object_object_info);

    dbg("Done");
    /* All done. Class is ready to be used for instantiating objects */
}

```

Listing 7.38: glib-dbus-sync/server.c

The `dbus_g_object_type_install_info` will take a pointer to the structure describing the D-Bus integration (`dbus_glib_value_object_object_info`), which is generated by `dbus-bindings-tool`. This function will create all the necessary runtime information for the GType, so the details can be left alone. It will also attach the introspection data to the GType, so that D-Bus introspection may return information on the interface that the object will implement.

The next functions to be implemented are the get and set functions, which allow us to inspect the interface as well. N.B. The names of the functions and their prototypes are ultimately dictated by `dbus-bindings-tool` generated stub header files. This means that if the interface XML is sufficiently changed, the code will fail to build (since the generated stubs will yield different prototypes):

```

/**
 * Function that gets called when someone tries to execute "setvalue1"
 * over the D-Bus. (Actually the marshaling code from the stubs gets
 * executed first, but they will eventually execute this function.)
 *
 * NOTE: If you change the name of this function, the generated
 * stubs will no longer find it! On the other hand, if you
 * decide to modify the interface XML, this is one of the places
 * that you will have to modify as well.
 * This applies to the next four functions (including this one).
 */
gboolean value_object_setvalue1(ValueObject* obj, gint valueIn,
                                GError** error) {

```

```

    dbg("Called (valueIn=%d)", valueIn);

    g_assert(obj != NULL);

    /* Change the value. */
    obj->value1 = valueIn;

    /* Return success to GLib/D-Bus wrappers. In this case we do not need
       to touch the supplied error pointer-pointer. */
    return TRUE;
}

/**
 * Function that gets executed on "setvalue2".
 * Other than this function operating with different type input
 * parameter (and different internal value), all the comments from
 * set_value1 apply here as well.
 */
gboolean value_object_setvalue2(ValueObject* obj, gdouble valueIn,
                                GError** error) {

    dbg("Called (valueIn=%.3f)", valueIn);

    g_assert(obj != NULL);

    obj->value2 = valueIn;

    return TRUE;
}

/**
 * Function that gets executed on "getvalue1".
 */
gboolean value_object_getvalue1(ValueObject* obj, gint* valueOut,
                                GError** error) {

    dbg("Called (internal value1 is %d)", obj->value1);

    g_assert(obj != NULL);

    /* Check that the target pointer is not NULL.
       Even is the only caller for this will be the GLib-wrapper code,
       we cannot trust the stub generated code and should handle the
       situation. We will terminate with an error in this case.

       Another option would be to create a new GError, and store
       the error condition there. */
    g_assert(valueOut != NULL);

    /* Copy the current first value to caller specified memory. */
    *valueOut = obj->value1;

    /* Return success. */
    return TRUE;
}

/**
 * Function that gets executed on "getvalue2".
 * (Again, similar to "getvalue1").
 */
gboolean value_object_getvalue2(ValueObject* obj, gdouble* valueOut,
                                GError** error) {

```

```

dbg("Called (internal value2 is %.3f)", obj->value2);

g_assert(obj != NULL);
g_assert(valueOut != NULL);

*valueOut = obj->value2;
return TRUE;
}

```

Listing 7.39: glib-dbus-sync/server.c

The GLib/D-Bus wrapper logic will implement all of the parameter conversion necessary from D-Bus into the functions, so it is only necessary to handle the GLib corresponding types (`gint` and `gdouble`). The method implementations will always receive an object reference to the object as their first parameter, and a pointer to a place where to store new `GError` objects if the method decides an error should be created. This error would then be propagated back to the caller of the D-Bus method. This simple get/set examples will never set errors, so the last parameter can be ignored here.

N.B. Returning the values is not performed via the conventional C way (by using `return someVal`), but instead return values are written via the given pointers. The return value of the method is always a `gboolean`, signifying either success or failure. If failure (`FALSE`) is returned, it is also necessary to create and setup an `GError` object, and store its address to the error location.

Publishing GType on D-Bus

Once the implementation is complete, it will be necessary to publish an instance of the class onto the D-Bus. This will be done inside the main of the server example, and involves performing a D-Bus method call on the bus.

So that both the server and client do not have to be changed, if the object or well-known names are changed later, they are put into a common header file that will be used by both:

```

/* Well-known name for this service. */
#define VALUE_SERVICE_NAME "org.maemo.Platdev_ex"
/* Object path to the provided object. */
#define VALUE_SERVICE_OBJECT_PATH "/GlobalValue"
/* And we're interested in using it through this interface.
   This must match the entry in the interface definition XML. */
#define VALUE_SERVICE_INTERFACE "org.maemo.Value"

```

Listing 7.40: glib-dbus-sync/common-defs.h

The decision to use `/GlobalValue` as the object path is based on clarity only. Most of the time something like `/org/maemo/Value` would be used instead.

Before using any of the GType functions, it is necessary to initialize the runtime system by calling `g_type_init`. This will create the built-in types and set-up all the machinery necessary for creating custom types as well. When using GTK+, the function is called automatically, when initializing GTK+. Since this example only uses GLib, it is necessary to call the function manually.

After initializing the GType system, the next step is to open a connection to the session bus, which will be used for the remainder of the publishing sequence:

```

/* Pull symbolic constants that are shared (in this example) between
   the client and the server. */
#include "common-defs.h"

/*... Listing cut for brevity ...*/

int main(int argc, char** argv) {

    /*... Listing cut for brevity ...*/

    /* Initialize the GType/GObject system. */
    g_type_init();

    /*... Listing cut for brevity ...*/

    g_print(PROGNAME ":main Connecting to the Session D-Bus.\n");
    bus = dbus_g_bus_get(DBUS_BUS_SESSION, &error);
    if (error != NULL) {
        /* Print error and terminate. */
        handleError("Could not connect to session bus", error->message,
                    TRUE);
    }
}

```

Listing 7.41: glib-dbus-sync/server.c

In order for prospective clients to find the object on the session bus, it is necessary to attach the server to a well-known name. This is done with the `RequestName` method call on the D-Bus server (over D-Bus). In order to target the server, it is necessary to create a GLib/D-Bus proxy object first:

```

g_print(PROGNAME ":main Registering the well-known name (%s)\n",
        VALUE_SERVICE_NAME);

/* In order to register a well-known name, we need to use the
   "RequestMethod" of the /org/freedesktop/DBus interface. Each
   bus provides an object that will implement this interface.

   In order to do the call, we need a proxy object first.
   DBUS_SERVICE_DBUS = "org.freedesktop.DBus"
   DBUS_PATH_DBUS = "/org/freedesktop/DBus"
   DBUS_INTERFACE_DBUS = "org.freedesktop.DBus" */
busProxy = dbus_g_proxy_new_for_name(bus,
                                     DBUS_SERVICE_DBUS,
                                     DBUS_PATH_DBUS,
                                     DBUS_INTERFACE_DBUS);

if (busProxy == NULL) {
    handleError("Failed to get a proxy for D-Bus",
               "Unknown(dbus_g_proxy_new_for_name)", TRUE);
}

/* Attempt to register the well-known name.
   The RPC call requires two parameters:
   - arg0: (D-Bus STRING): name to request
   - arg1: (D-Bus UINT32): flags for registration.
     (please see "org.freedesktop.DBus.RequestName" in
     http://dbus.freedesktop.org/doc/dbus-specification.html)
   Will return one uint32 giving the result of the RPC call.
   We are interested in 1 (we are now the primary owner of the name)
   or 4 (we were already the owner of the name, however in this
   application it would not make much sense).

```



```

The function will return FALSE if it sets the GError. */
if (!dbus_g_proxy_call(busProxy,
    /* Method name to call. */
    "RequestName",
    /* Where to store the GError. */
    &error,
    /* Parameter type of argument 0. Note that
       since we are dealing with GLib/D-Bus
       wrappers, you will need to find a suitable
       GType instead of using the "native" D-Bus
       type codes. */
    G_TYPE_STRING,
    /* Data of argument 0. In our case, this is
       the well-known name for our server
       example ("org.maemo.Platdev_ex"). */
    VALUE_SERVICE_NAME,
    /* Parameter type of argument 1. */
    G_TYPE_UINT,
    /* Data of argument 0. This is the "flags"
       argument of the "RequestName" method which
       can be use to specify what the bus service
       should do when the name already exists on
       the bus. We will go with defaults. */
    0,
    /* Input arguments are terminated with a
       special GType marker. */
    G_TYPE_INVALID,
    /* Parameter type of return value 0.
       For "RequestName" it is UINT32 so we pick
       the GType that maps into UINT32 in the
       wrappers. */
    G_TYPE_UINT,
    /* Data of return value 0. These will always
       be pointers to the locations where the
       proxy can copy the results. */
    &result,
    /* Terminate list of return values. */
    G_TYPE_INVALID)) {
    handleError("D-Bus.RequestName RPC failed", error->message,
               TRUE);
    /* Note that the whole call failed, not "just" the name
       registration (we deal with that below). This means that
       something bad probably has happened and there is not much we can
       do (hence program termination). */
}
/* Check the result code of the registration RPC. */
g_print(PROGNAME ":main RequestName returned %d.\n", result);
if (result != 1) {
    handleError("Failed to get the primary well-known name.",
               "RequestName result != 1", TRUE);
    /* In this case we could also continue instead of terminating.
       We could retry the RPC later. Or "lurk" on the bus waiting for
       someone to tell us what to do. If we would be publishing
       multiple services and/or interfaces, it even might make sense
       to continue with the rest anyway.

       In our simple program, we terminate. Not much left to do for
       this poor program if the clients will not be able to find the
       Value object using the well-known name. */
}

```

Listing 7.42: glib-dbus-sync/server.c

The `dbus_g_proxy_call` function is used to do synchronous method calls in GLib/D-Bus wrappers, and in this case, it will be used to run the two argument `RequestName` method call. The method returns one value (and `uint32`), which encodes the result of the well-known name registration.

One needs to be careful with the order and correctness of the parameters to the function call, as it is easy to get something wrong, and the C compiler cannot really check for parameter type validity here.

After the successful name registration, it is finally time to create an instance of the `ValueObject` and publish it on the D-Bus:

```
g_print(PROGNAME ":main Creating one Value object.\n");
/* The NULL at the end means that we have stopped listing the
   property names and their values that would have been used to
   set the properties to initial values. Our simple Value
   implementation does not support GObject properties, and also
   does not inherit anything interesting from GObject directly, so
   there are no properties to set. For more examples on properties
   see the first GTK+ example programs from the maemo Application
   Development material.

   NOTE: You need to keep at least one reference to the published
   object at all times, unless you want it to disappear from
   the D-Bus (implied by API reference for
   dbus_g_connection_register_g_object(). */
valueObj = g_object_new(VALUE_TYPE_OBJECT, NULL);
if (valueObj == NULL) {
    handleError("Failed to create one Value instance.",
               "Unknown(OOM?)", TRUE);
}

g_print(PROGNAME ":main Registering it on the D-Bus.\n");
/* The function does not return any status, so can not check for
   errors here. */
dbus_g_connection_register_g_object(bus,
                                   VALUE_SERVICE_OBJECT_PATH,
                                   G_OBJECT(valueObj));

g_print(PROGNAME ":main Ready to serve requests (daemonizing).\n");

/*... Listing cut for brevity ...*/
}
```

Listing 7.43: `glib-dbus-sync/server.c`

And after this, the main will enter into the main loop, and will serve client requests coming over the D-Bus, until the server is terminated. N.B. All the callback registration is performed automatically by the GLib/D-Bus wrappers on object publication, so there is no need to worry about them.

Implementing the dependencies and rules for the server and the generated stub code will give this snippet:

```
server: server.o
      $(CC) $^ -o $@ $(LDFLAGS)

# ... Listing cut for brevity ...

# The server and client depend on the respective implementation source
# files, but also on the common interface as well as the generated
# stub interfaces.
```

```
server.o: server.c common-defs.h value-server-stub.h
$(CC) $(CFLAGS) -DPROGNAME=\"$(basename $@)\" -c $< -o $@
```

Listing 7.44: glib-dbus-sync/Makefile

When implementing makefiles that separate compilation from linking, it is not possible to pass the target name (automatic variable \$@) directly as the PROGNAME-define (since that would expand into server.o, and would look slightly silly when all the messages were prefixed with the name). Instead, a GNU make function (basename) is used to strip any prefixes and suffixes out of the parameter. This way, the PROGNAME will be set to server.

The next step is to build the server and start it:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make server
dbus-binding-tool --prefix=value_object --mode=glib-server \
  value-dbus-interface.xml > value-server-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -g -Wall
-DG_DISABLE_DEPRECATED -DNO_DAEMON -DPROGNAME=\"server\"
-c server.c -o server.o
cc server.o -o server -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh ./server
server:main Connecting to the Session D-Bus.
server:main Registering the well-known name (org.maemo.Platdev_ex)
server:main RequestName returned 1.
server:main Creating one Value object.
server:value_object_class_init: Called
server:value_object_class_init: Binding to GLib/D-Bus
server:value_object_class_init: Done
server:value_object_init: Called
server:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
server: Not daemonizing (built with NO_DAEMON-build define)
```

After this, dbus-send is used to test out the implementation details from the server. This is done in the same session (for simplicity) by first suspending the server with Ctrl+z, and then continuing running it with the bg shell built-in command. This is done so that it will be easier to see the reaction of the server to each dbus-send command.

The first step here is to test the getvalue1 and setvalue1 methods:

```
[Ctrl+z]
[1]+  Stopped                  run-standalone.sh ./server
[sbox-DIABLO_X86: ~/glib-dbus-sync] > bg
[1]+  run-standalone.sh ./server &
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 0)
method return sender=:1.15 -> dest=:1.20
int32 0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.setvalue1 int32:5
server:value_object_setvalue1: Called (valueIn=5)
method return sender=:1.15 -> dest=:1.21
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 5)
method return sender=:1.15 -> dest=:1.22
int32 5
```

The next step is to test the double state variable with getvalue2 and setvalue2 methods:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue2
server:value_object_getvalue2: Called (internal value2 is 0.000)
method return sender=:1.15 -> dest=:1.23
double 0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.setvalue2 double:42.0
server:value_object_setvalue2: Called (valueIn=42.000)
method return sender=:1.15 -> dest=:1.24
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue2
server:value_object_getvalue2: Called (internal value2 is 42.000)
method return sender=:1.15 -> dest=:1.25
double 42
```

This results in a fully functional D-Bus service implementation, albeit a very simple one.

The next step is to utilize the service from a client.

Using GLib/D-Bus Wrapper from Client

By using the generated client stub file, it is now possible to write the client that will invoke the methods on the Value object. The D-Bus method calls could also be performed "manually" (either with GLib/D-Bus functions, or even by using libdbus directly, but the latter is discouraged).

The dbus-bindings-tool (when run with the `--mode=glib-client` parameter) will generate functions for each of the interface methods, and the functions will handle data marshaling operations internally.

Two generated stub functions are presented below, and they will be used shortly:

```
/* Generated by dbus-binding-tool; do not edit! */

/*... Listing cut for brevity ...*/

static
#ifdef G_HAVE_INLINE
inline
#endif
gboolean
org_maemo_Value_getvalue1 (DBusGProxy *proxy, gint* OUT_cur_value,
                          GError **error)
{
    return dbus_g_proxy_call (proxy, "getvalue1", error, G_TYPE_INVALID,
                              G_TYPE_INT, OUT_cur_value, G_TYPE_INVALID);
}

/*... Listing cut for brevity ...*/

static
#ifdef G_HAVE_INLINE
inline
#endif
gboolean
org_maemo_Value_setvalue1 (DBusGProxy *proxy, const gint IN_new_value,
                          GError **error)
{

```

```

return dbus_g_proxy_call (proxy, "setvalue1", error, G_TYPE_INT,
                           IN_new_value, G_TYPE_INVALID,
                           G_TYPE_INVALID);
}

```

Listing 7.45: glib-dbus-sync/value-client-stub.h

The two functions presented above are both **blocking**, which means that they will wait for the result to arrive over the D-Bus, and only then return to the caller. The generated stub code also includes **asynchronous** functions (their names end with `_async`), but their usage will be covered later.

For now, it is important to notice how the prototypes of the functions are named, and what are the parameters that they expect to be passed to them.

The `org_maemo_Value` prefix is taken from the interface XML file, from the name attribute of the interface element. All dots will be converted into underscores (since C reserves the dot character for other uses), but otherwise the name will be preserved (barring dashes in the name).

The rest of the function name will be the method name for each method defined in the interface XML file.

The first parameter for all the generated stub functions will always be a pointer to a `DBusProxy` object, which will be necessary to use with the GLib/D-Bus wrapper functions. After the proxy, a list of method parameters is passed. The binding tool will prefix the parameter names with either `IN_` or `OUT_` depending on the "direction" of the parameter. Rest of the parameter name is taken from the name attributed of the arg element for the method, or if not given, will be automatically generated as `arg0`, `arg1`, etc. Input parameters will be passed as values (unless they are complex or strings, in which case they will be passed as pointers). Output parameters are always passed as pointers.

The functions will always return a `gboolean`, indicating failure or success, and if they fail, they will also create and set the error pointer to an `GError`-object which can then be checked for the reason for the error (unless the caller passed a `NULL` pointer for error, in which case the error object will not be created).

```

#include <glib.h>
#include <dbus/dbus-glib.h>
#include <stdlib.h> /* exit, EXIT_FAILURE */
#include <string.h> /* strcmp */

/* Pull the common symbolic defines. */
#include "common-defs.h"

/* Pull in the client stubs that were generated with
   dbus-binding-tool */
#include "value-client-stub.h"

```

Listing 7.46: glib-dbus-sync/client.c

This will allow the client code to use the stub code directly as follows:

```

/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * The function will start with two statically initialized variables
 * (int and double) which will be incremented after each time this
 * function runs and will use the setvalue* remote methods to set the
 * new values. If the set methods fail, program is not aborted, but an

```

```

/* message will be issued to the user describing the error.
*/
static gboolean timerCallback(DBusGProxy* remoteobj) {

    /* Local values that we'll start updating to the remote object. */
    static gint localValue1 = -80;
    static gdouble localValue2 = -120.0;

    GError* error = NULL;

    /* Set the first value. */
    org_maemo_Value_setvalue1(remoteobj, localValue1, &error);
    if (error != NULL) {
        handleError("Failed to set value1", error->message, FALSE);
    } else {
        g_print(PROGNAME ":timerCallback Set value1 to %d\n", localValue1);
    }

    /* If there was an error with the first, release the error, and
    don't attempt the second time. Also, don't add to the local
    values. We assume that errors from the first set are caused by
    server going off the D-Bus, but are hopeful that it will come
    back, and hence keep trying (returning TRUE).*/
    if (error != NULL) {
        g_clear_error(&error);
        return TRUE;
    }

    /* Now try to set the second value as well. */
    org_maemo_Value_setvalue2(remoteobj, localValue2, &error);
    if (error != NULL) {
        handleError("Failed to set value2", error->message, FALSE);
        g_clear_error(&error); /* Or g_error_free in this case. */
    } else {
        g_print(PROGNAME ":timerCallback Set value2 to %.31f\n",
            localValue2);
    }

    /* Step the local values forward. */
    localValue1 += 10;
    localValue2 += 10.0;

    /* Tell the timer launcher that we want to remain on the timer
    call list in the future as well. Returning FALSE here would
    stop the launch of this timer callback. */
    return TRUE;
}

```

Listing 7.47: glib-dbus-sync/client.c

What is left is connecting to the correct D-Bus, creating a GProxy object which will be done in the test program:

```

/**
 * The test program itself.
 *
 * 1) Setup GType/GSignal
 * 2) Create GMainLoop object
 * 3) Connect to the Session D-Bus
 * 4) Create a proxy GObject for the remote Value object
 * 5) Start a timer that will launch timerCallback once per second.
 * 6) Run main-loop (forever)
 */

```

```

int main(int argc, char** argv) {
    /* The D-Bus connection object. Provided by GLib/D-Bus wrappers. */
    DBusGConnection* bus;
    /* This will represent the Value object locally (acting as a proxy
       for all method calls and signal delivery. */
    DBusGProxy* remoteValue;
    /* This will refer to the GMainLoop object */
    GMainLoop* mainloop;
    GError* error = NULL;

    /* Initialize the GType/GObject system. */
    g_type_init();

    /* Create a new GMainLoop with default context (NULL) and initial
       state of "not running" (FALSE). */
    mainloop = g_main_loop_new(NULL, FALSE);
    /* Failure to create the main loop is fatal (for us). */
    if (mainloop == NULL) {
        handleError("Failed to create the mainloop", "Unknown (OOM?)",
                    TRUE);
    }

    g_print(PROGNAME ":main Connecting to Session D-Bus.\n");
    bus = dbus_g_bus_get(DBUS_BUS_SESSION, &error);
    if (error != NULL) {
        handleError("Couldn't connect to the Session bus", error->message,
                    TRUE);
        /* Normally you'd have to also g_error_free() the error object
           but since the program will terminate within handleError,
           it is not necessary here. */
    }

    g_print(PROGNAME ":main Creating a GLib proxy object for Value.\n");

    /* Create the proxy object that we'll be using to access the object
       on the server. If you would use dbus_g_proxy_for_name_owner(),
       you would be also notified when the server that implements the
       object is removed (or rather, the interface is removed). Since
       we don't care who actually implements the interface, we'll use the
       more common function. See the API documentation at
       http://maemo.org/api\_refs/4.0/dbus/ for more details. */
    remoteValue =
        dbus_g_proxy_new_for_name(bus,
                                   VALUE_SERVICE_NAME, /* name */
                                   VALUE_SERVICE_OBJECT_PATH, /* obj path */
                                   VALUE_SERVICE_INTERFACE /* interface */);
    if (remoteValue == NULL) {
        handleError("Couldn't create the proxy object",
                    "Unknown(dbus_g_proxy_new_for_name)", TRUE);
    }

    g_print(PROGNAME ":main Starting main loop (first timer in 1s).\n");

    /* Register a timer callback that will do RPC sets on the values.
       The userdata pointer is used to pass the proxy object to the
       callback so that it can launch modifications to the object. */
    g_timeout_add(1000, (GSourceFunc)timerCallback, remoteValue);

    /* Run the program. */
    g_main_loop_run(mainloop);
    /* Since the main loop is not stopped (by this code), we shouldn't
       ever get here. The program might abort() for other reasons. */
}

```

```

    /* If it does, return failure as exit code. */
    return EXIT_FAILURE;
}

```

Listing 7.48: glib-dbus-sync/client.c

Integrating the client into the **Makefile** is done the same way as was did for the server before:

```

client: client.o
    $(CC) $^ -o $@ $(LDFLAGS)

# ... Listing cut for brevity ...

client.o: client.c common-defs.h value-client-stub.h
    $(CC) $(CFLAGS) -DPROGRAMNAME=\"$(basename $@)\" -c $< -o $@

```

Listing 7.49: glib-dbus-sync/Makefile

After building the client, will be started, and let it execute in the same terminal session where the server is still running:

```

[sbox-DIABLO_X86: ~/glib-dbus-sync] > make client
dbus-binding-tool --prefix=value_object --mode=glib-client \
  value-dbus-interface.xml > value-client-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include \
  -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -g -Wall \
  -DG_DISABLE_DEPRECATED -DNO_DAEMON -DPROGRAMNAME=\"client\" \
  -c client.c -o client.o
cc client.o -o client -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh ./client
client:main Connecting to Session D-Bus.
client:main Creating a GLib proxy object for Value.
client:main Starting main loop (first timer in 1s).
server:value_object_setvalue1: Called (valueIn=-80)
client:timerCallback Set value1 to -80
server:value_object_setvalue2: Called (valueIn=-120.000)
client:timerCallback Set value2 to -120.000
server:value_object_setvalue1: Called (valueIn=-70)
client:timerCallback Set value1 to -70
server:value_object_setvalue2: Called (valueIn=-110.000)
client:timerCallback Set value2 to -110.000
server:value_object_setvalue1: Called (valueIn=-60)
client:timerCallback Set value1 to -60
...
[Ctrl+c]
[sbox-DIABLO_X86: ~/glib-dbus-sync] > fg
run-standalone.sh ./server
[Ctrl+c]

```

Since the client will normally run forever, it will now be terminated, and then the server will be moved to the foreground, so that it can also be terminated. This concludes the first GLib/D-Bus example, but for more information about the GLib D-Bus wrappers, please consult D-BUS GLib Bindings Documentation [57].

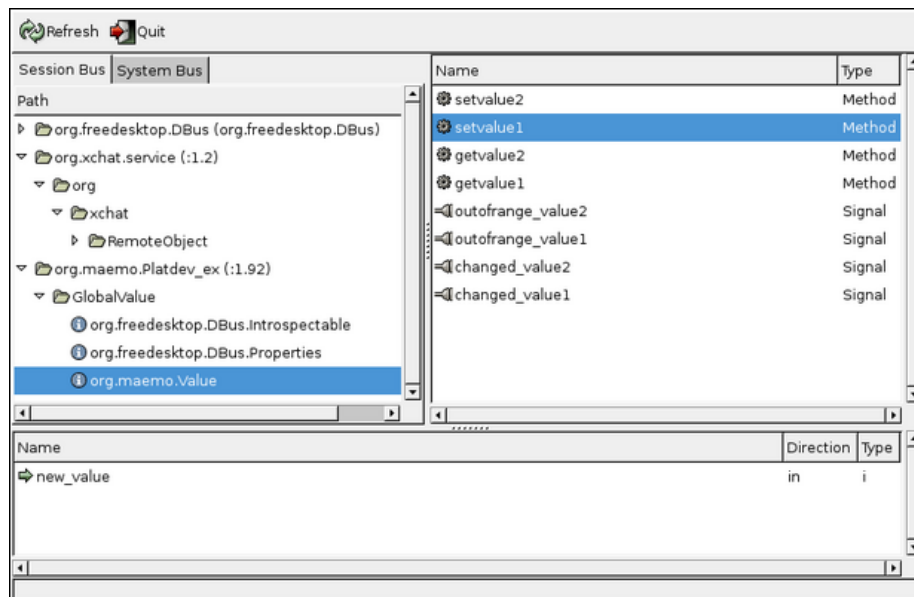
D-Bus Introspection

D-Bus supports a mechanism by which programs can interrogate the bus for existing well-known names, and then get the interfaces implemented by the objects available behind the well-known names. This mechanism is called *introspection* in D-Bus terminology.

The main goal of supporting introspection in D-Bus is allowing dynamic bindings to be made with high-level programming languages. This way, the language wrappers for D-Bus can be more intelligent automatically (assuming they utilize the introspection interface). The GLib-wrappers do not use the introspection interface.

Introspection is achieved with three D-Bus methods: ListNames, GetNameOwner and Introspect. The destination object must support the introspection interface in order to provide this information. If the dbus-bindings-tool is used, and the GObject is registered correctly, the service will automatically support introspection.

D-Bus (at this moment) does not come with introspection utilities, but some are available from other sources. One simple program is the "DBus Inspector" that is written in Python, and uses the Python D-Bus bindings and GTK+. If planning to write one's own tool, it is necessary to prepare to parse XML data, since that is the format of results that the Introspect method returns.



Using DBUS Inspector on GlobalValue on a desktop system. Note that the version of GlobalValue used here also implements signals, which will be covered next

Introspection can also be useful, when trying to find out what are the different interfaces and methods available for use on a system. It just has to be remembered that not all D-Bus services actually implement the introspection interface. Their well-known names can still be received, but their interface descriptions will come up empty when using Introspect.

7.3.4 Implementing and Using D-Bus Signals

D-Bus Signal properties

Performing remote method invocations over the D-Bus is only one half of D-Bus capabilities. As was noted before, D-Bus also supports a *broadcast* method of communication, which is also *asynchronous*. This mechanism is called a

signal (in D-Bus terminology), and is useful when it is necessary to notify a lot of receivers about a state change that could affect them. Some examples where signals could be useful are notifying a lot of receivers, when the system is being shut down, network connectivity has been lost, and similar system-wide conditions. This way, the receivers do not need to poll for the status continuously.

However, signals are not the solution to all problems. If a recipient is not processing its D-Bus messages quickly enough (or there just are too many), a signal might get lost on its way to the recipient. There might also be other complications, as with any RPC mechanism. For these reasons, if the application requires extra reliability, it will be necessary to think how to arrange it. One possibility would be to occasionally check the state that the application is interested in, assuming it can be checked over the D-Bus. It just should not be done too often; the recommended interval is once a minute or less often, and only when the application is already active for other reasons. This model will lead to reduction in battery life, so its use should be carefully weighed.

Signals in D-Bus are able to carry information. Each signal has its own name (specified in the interface XML), as well as "arguments". In signal's case, the argument list is actually just a list of information that is passed along the signal, and should not be confused with method call parameters (although both are delivered in the same manner).

Signals do not "return", meaning that when a D-Bus signal is sent, no reply will be received, nor will one be expected. If the signal emitter wants to be sure that the signal was delivered, additional mechanisms need to be constructed for this (D-Bus does not include them directly). A D-Bus signal is very similar to most datagram-based network protocols, for example UDP. Sending a signal will succeed, even if there are no receivers interested in that specific signal.

Most D-Bus language bindings will attempt to map D-Bus signals into something more natural in the target language. Since GLib already supports the notion of signals (as GLib signals), this mapping is quite natural. So in practice, the client will register for GLib signals, and then handle the signals in callback functions (a special wrapper function must be used to register for the wrapped signals: `dbus_g_proxy_connect_signal`).

Declaring Signals in Interface XML

Next, the Value object will be extended so that it contains two threshold values (minimum and maximum), and the object will emit signals, whenever a set operation falls outside the thresholds.

A signal will also be emitted, whenever a value is changed (i.e. the binary content of the new value is different from the old one).

In order to make the signals available to introspection data, the interface XML file will be modified accordingly:

```
<node>
  <interface name="org.maemo.Value">

    <!-- ... Listing cut for brevity ... -->

    <!-- Signal (D-Bus) definitions -->

    <!-- NOTE: The current version of dbus-bindings-tool doesn't
```

```

        actually enforce the signal arguments _at_all_. Signals need
        to be declared in order to be passed through the bus itself,
        but otherwise no checks are done! For example, you could
        leave the signal arguments unspecified completely, and the
        code would still work. -->

<!-- Signals to tell interested clients about state change.
      We send a string parameter with them. They never can have
      arguments with direction=in. -->
<signal name="changed_value1">
  <arg type="s" name="change_source_name" direction="out"/>
</signal>

<signal name="changed_value2">
  <arg type="s" name="change_source_name" direction="out"/>
</signal>

<!-- Signals to tell interested clients that values are outside
      the internally configured range (thresholds). -->
<signal name="outofrange_value1">
  <arg type="s" name="outofrange_source_name" direction="out"/>
</signal>
<signal name="outofrange_value2">
  <arg type="s" name="outofrange_source_name" direction="out"/>
</signal>

</interface>
</node>

```

Listing 7.50: glib-dbus-signals/value-dbus-interface.xml

The signal definitions are required, if planning to use the `dbus-bindings-tool`; however, the argument specification for each signal is not required by the tool. In fact, it will just ignore all argument specifications, and as can be seen below, a lot of "manual coding" has to be made in order to implement and use the signals (on both the client and server side). The `dbus-bindings-tool` might get more features in the future, but for now, some labor is required.

Emitting Signals from GObject

So that the signal names can easily be changed later, they will be defined in a header file, and the header file will be used in both the server and client. The following is the section with the signal names:

```

/* Symbolic constants for the signal names to use with GLib.
   These need to map into the D-Bus signal names. */
#define SIGNAL_CHANGED_VALUE1    "changed_value1"
#define SIGNAL_CHANGED_VALUE2    "changed_value2"
#define SIGNAL_OUTOFRANGE_VALUE1 "outofrange_value1"
#define SIGNAL_OUTOFRANGE_VALUE2 "outofrange_value2"

```

Listing 7.51: glib-dbus-signals/common-defs.h

Before a GObject can emit a GLib signal, the signal itself needs to be defined and created. This is best done in the class constructor code (since the signal types need to be created only once):

```

/**
 * Define enumerations for the different signals that we can generate
 * (so that we can refer to them within the signals-array [below]

```

```

* using symbolic names). These are not the same as the signal name
* strings.
*
* NOTE: E_SIGNAL_COUNT is NOT a signal enum. We use it as a
*       convenient constant giving the number of signals defined so
*       far. It needs to be listed last.
*/
typedef enum {
    E_SIGNAL_CHANGED_VALUE1,
    E_SIGNAL_CHANGED_VALUE2,
    E_SIGNAL_OUTOFRANGE_VALUE1,
    E_SIGNAL_OUTOFRANGE_VALUE2,
    E_SIGNAL_COUNT
} ValueSignalNumber;

/*... Listing cut for brevity ...*/

typedef struct {
    /* The parent class state. */
    GObjectClass parent;
    /* The minimum number under which values will cause signals to be
       emitted. */
    gint thresholdMin;
    /* The maximum number over which values will cause signals to be
       emitted. */
    gint thresholdMax;
    /* Signals created for this class. */
    guint signals[E_SIGNAL_COUNT];
} ValueObjectClass;

/*... Listing cut for brevity ...*/

/**
 * Per class initializer
 *
 * Sets up the thresholds (-100 .. 100), creates the signals that we
 * can emit from any object of this class and finally registers the
 * type into the GLib/D-Bus wrapper so that it may add its own magic.
 */
static void value_object_class_init(ValueObjectClass* klass) {

    /* Since all signals have the same prototype (each will get one
       string as a parameter), we create them in a loop below. The only
       difference between them is the index into the klass->signals
       array, and the signal name.

       Since the index goes from 0 to E_SIGNAL_COUNT-1, we just specify
       the signal names into an array and iterate over it.

       Note that the order here must correspond to the order of the
       enumerations before. */
    const gchar* signalNames[E_SIGNAL_COUNT] = {
        SIGNAL_CHANGED_VALUE1,
        SIGNAL_CHANGED_VALUE2,
        SIGNAL_OUTOFRANGE_VALUE1,
        SIGNAL_OUTOFRANGE_VALUE2 };
    /* Loop variable */
    int i;

    dbg("Called");
    g_assert(klass != NULL);

```

```

/* Setup sane minimums and maximums for the thresholds. There is no
   way to change these afterwards (currently), so you can consider
   them as constants. */
klass->thresholdMin = -100;
klass->thresholdMax = 100;

dbg("Creating signals");

/* Create the signals in one loop, since they all are similar
   (except for the names). */
for (i = 0; i < E_SIGNAL_COUNT; i++) {
    guint signalId;

    /* Most of the time you will encounter the following code without
       comments. This is why all the parameters are documented
       directly below. */
    signalId =
        g_signal_new(signalNames[i], /* str name of the signal */
                     /* GType to which signal is bound to */
                     G_OBJECT_CLASS_TYPE(klass),
                     /* Combination of GSignalFlags which tell the
                        signal dispatch machinery how and when to
                        dispatch this signal. The most common is the
                        G_SIGNAL_RUN_LAST specification. */
                     G_SIGNAL_RUN_LAST,
                     /* Offset into the class structure for the type
                        function pointer. Since we're implementing a
                        simple class/type, we'll leave this at zero. */
                     0,
                     /* GSignalAccumulator to use. We don't need one. */
                     NULL,
                     /* User-data to pass to the accumulator. */
                     NULL,
                     /* Function to use to marshal the signal data into
                        the parameters of the signal call. Luckily for
                        us, GLib (GCClosure) already defines just the
                        function that we want for a signal handler that
                        we don't expect any return values (void) and
                        one that will accept one string as parameter
                        (besides the instance pointer and pointer to
                        user-data).

                        If no such function would exist, you would need
                        to create a new one (by using glib-genmarshal
                        tool). */
                     g_cclosure_marshal_VOID__STRING,
                     /* Return GType of the return value. The handler
                        does not return anything, so we use G_TYPE_NONE
                        to mark that. */
                     G_TYPE_NONE,
                     /* Number of parameter GTypes to follow. */
                     1,
                     /* GType(s) of the parameters. We only have one. */
                     G_TYPE_STRING);

    /* Store the signal Id into the class state, so that we can use
       it later. */
    klass->signals[i] = signalId;

    /* Proceed with the next signal creation. */
}
/* All signals created. */

```

```

dbg("Binding to GLib/D-Bus");

/*... Listing cut for brevity ...*/

}

```

Listing 7.52: glib-dbus-signals/server.c

The signal types will be kept in the class structure, so that they can be referenced easily by the signal emitting utility (covered next). The class constructor code will also set up the threshold limits, which in this implementation will be immutable (i.e. they cannot be changed). It is advisable to experiment with adding more methods to adjust the thresholds as necessary.

Emitting the signals is then quite easy, but in order to reduce code amount, a utility function will be created to launch a given signal based on its enumeration:

```

/**
 * Utility helper to emit a signal given with internal enumeration and
 * the passed string as the signal data.
 *
 * Used in the setter functions below.
 */
static void value_object_emitSignal(ValueObject* obj,
                                   ValueSignalNumber num,
                                   const gchar* message) {

    /* In order to access the signal identifiers, we need to get a hold
       of the class structure first. */
    ValueObjectClass* klass = VALUE_OBJECT_GET_CLASS(obj);

    /* Check that the given num is valid (abort if not).
       Given that this file is the module actually using this utility,
       you can consider this check superfluous (but useful for
       development work). */
    g_assert((num < E_SIGNAL_COUNT) && (num >= 0));

    dbg("Emitting signal id %d, with message '%s'", num, message);

    /* This is the simplest way of emitting signals. */
    g_signal_emit(/* Instance of the object that is generating this
                   signal. This will be passed as the first parameter
                   to the signal handler (eventually). But obviously
                   when speaking about D-Bus, a signal caught on the
                   other side of D-Bus will be first processed by
                   the GLib-wrappers (the object proxy) and only then
                   processed by the signal handler. */
                 obj,
                 /* Signal id for the signal to generate. These are
                    stored inside the class state structure. */
                 klass->signals[num],
                 /* Detail of signal. Since we are not using detailed
                    signals, we leave this at zero (default). */
                 0,
                 /* Data to marshal into the signal. In our case it's
                    just one string. */
                 message);

    /* g_signal_emit returns void, so we cannot check for success. */

    /* Done emitting signal. */
}

```

Listing 7.53: glib-dbus-signals/server.c

So that it would not be necessary to check the threshold values in multiple places in the source code, that will also be implemented as a separate function. Emitting the "threshold exceeded" signal is still up to the caller.

```
/**
 * Utility to check the given integer against the thresholds.
 * Will return TRUE if thresholds are not exceeded, FALSE otherwise.
 *
 * Used in the setter functions below.
 */
static gboolean value_object_thresholdsOk(ValueObject* obj,
                                          gint value) {

    /* Thresholds are in class state data, get access to it */
    ValueObjectClass* klass = VALUE_OBJECT_GET_CLASS(obj);

    return ((value >= klass->thresholdMin) &&
            (value <= klass->thresholdMax));
}
```

Listing 7.54: glib-dbus-signals/server.c

Both utility functions are then used from within the respective set functions, one of which is presented below:

```
/**
 * Function that gets called when someone tries to execute "setvalue1"
 * over the D-Bus. (Actually the marshalling code from the stubs gets
 * executed first, but they will eventually execute this function.)
 */
gboolean value_object_setvalue1(ValueObject* obj, gint valueIn,
                                GError** error) {

    dbg("Called (valueIn=%d)", valueIn);

    g_assert(obj != NULL);

    /* Compare the current value against old one. If they're the same,
     * we don't need to do anything (except return success). */
    if (obj->value1 != valueIn) {
        /* Change the value. */
        obj->value1 = valueIn;

        /* Emit the "changed_value1" signal. */
        value_object_emitSignal(obj, E_SIGNAL_CHANGED_VALUE1, "value1");

        /* If new value falls outside the thresholds, emit
         * "outofrange_value1" signal as well. */
        if (!value_object_thresholdsOk(obj, valueIn)) {
            value_object_emitSignal(obj, E_SIGNAL_OUTOFRANGE_VALUE1,
                                    "value1");
        }
    }

    /* Return success to GLib/D-Bus wrappers. In this case we don't need
     * to touch the supplied error pointer-pointer. */
    return TRUE;
}
```

Listing 7.55: glib-dbus-signals/server.c

The role of the "value1" string parameter that is sent along both of the signals above might raise a question. Sending the signal origin name with the signal allows one to reuse the same callback function in the client. It is quite rare that this kind of "source naming" would be useful, but it allows for writing a slightly shorter client program.

The implementation of setvalue2 is almost identical, but deals with the gdouble parameter.

The getvalue functions are identical to the versions before as is the Makefile.

Next, the server is built and started on the background (in preparation for testing with dbus-send):

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > make server
dbus-binding-tool --prefix=value_object --mode=glib-server \
  value-dbus-interface.xml > value-server-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include \
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -g -Wall \
-DG_DISABLE_DEPRECATED -DNO_DAEMON -DPROGNAME=\"server\" \
-c server.c -o server.o
cc server.o -o server -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh ./server &
[1] 15293
server:main Connecting to the Session D-Bus.
server:main Registering the well-known name (org.maemo.Platdev_ex)
server:main RequestName returned 1.
server:main Creating one Value object.
server:value_object_class_init: Called
server:value_object_class_init: Creating signals
server:value_object_class_init: Binding to GLib/D-Bus
server:value_object_class_init: Done
server:value_object_init: Called
server:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
server: Not daemonizing (built with NO_DAEMON-build define)
[sbox-DIABLO_X86: ~/glib-dbus-signals] >
```

The next step is to test the setvalue1 method:

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.setvalue1 int32:10
server:value_object_setvalue1: Called (valueIn=10)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
method return sender=:1.38 -> dest=:1.41
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.setvalue1 int32:-200
server:value_object_setvalue1: Called (valueIn=-200)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_emitSignal: Emitting signal id 2, with message 'value1'
method return sender=:1.38 -> dest=:1.42
```

And then setvalue2 (with doubles). At this point, something strange might be noticed in the threshold triggering:

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.setvalue2 double:100.5
server:value_object_setvalue2: Called (valueIn=100.500)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
method return sender=:1.38 -> dest=:1.44
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.setvalue2 double:101
server:value_object_setvalue2: Called (valueIn=101.000)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
server:value_object_emitSignal: Emitting signal id 3, with message 'value2'
method return sender=:1.38 -> dest=:1.45
```


Since the threshold testing logic will truncate the `gdouble` before testing against the (integer) thresholds, a value of 100.5 will be detected as 100, and will still fit within the thresholds.

Instead of printing out the emitted signal names, their enumeration values are printed. This could be rectified with a small enumeration to string table, but it was emitted from the program for simplicity.

It can also be noticed that other than seeing the server messages about emitting the signals, there is not a trace of them being sent or received. This is because `dbus-send` does not listen for signals. There is a separate tool for tracing signals, and it will be covered at the end of this chapter (`dbus-monitor`).

Catching Signals in GLib/D-Bus Clients

In order to receive D-Bus signals in the client, one needs to do quite a bit of work per signal. This is because `dbus-bindings-tool` does not generate any code for signals (at the moment). The aim is to make the GLib wrappers emit `GSignals`, whenever an interesting D-Bus signal arrives. This also means that it will be necessary to register the interest for a particular D-Bus signal.

When implementing the callbacks for the signals, it is necessary to take care to implement the prototype correctly. Since the signals will be sent with one attached string value, the callbacks will receive at least the string parameter. Besides the signal attached arguments, the callback will receive the proxy object, through which the signal was received, and optional user-specified data (which will not be used in this example, so it will be always `NULL`).

```
/**
 * Signal handler for the "changed" signals. These will be sent by the
 * Value-object whenever its contents change (whether within
 * thresholds or not).
 *
 * Like before, we use strcmp to differentiate between the two
 * values, and act accordingly (in this case by retrieving
 * synchronously the values using getvalue1 or getvalue2.
 *
 * NOTE: Since we use synchronous getvalues, it is possible to get
 * this code stuck if for some reason the server would be stuck
 * in an eternal loop.
 */
static void valueChangedSignalHandler(DBusGProxy* proxy,
                                     const char* valueName,
                                     gpointer userData) {
    /* Since method calls over D-Bus can fail, we'll need to check
     * for failures. The server might be shut down in the middle of
     * things, or might act badly in other ways. */
    GError* error = NULL;

    g_print(PROGNAME ":value-changed (%s)\n", valueName);

    /* Find out which value changed, and act accordingly. */
    if (strcmp(valueName, "value1") == 0) {
        gint v = 0;
        /* Execute the RPC to get value1. */
        org_maemo_Value_getvalue1(proxy, &v, &error);
        if (error == NULL) {
            g_print(PROGNAME ":value-changed Value1 now %d\n", v);
        } else {
            /* You could interrogate the GError further, to find out exactly
```

```

        what the error was, but in our case, we'll just ignore the
        error with the hope that some day (preferably soon), the
        RPC will succeed again (server comes back on the bus). */
        handleError("Failed to retrieve value1", error->message, FALSE);
    }
} else {
    gdouble v = 0.0;
    org_maemo_Value_getvalue2(proxy, &v, &error);
    if (error == NULL) {
        g_print(PROGNAME ":value-changed Value2 now %.3f\n", v);
    } else {
        handleError("Failed to retrieve value2", error->message, FALSE);
    }
}
/* Free up error object if one was allocated. */
g_clear_error(&error);
}

```

Listing 7.56: glib-dbus-signals/client.c

The callback will first determine, what was the source value that caused the signal to be generated. For this, it uses the string argument of the signal. It will then retrieve the current value using the respective RPC methods (getvalue1 or getvalue2), and print out the value.

If any errors occur during the method calls, the errors will be printed out, but the program will continue to run. If an error does occur, the GError object will need to be freed (performed with g_clear_error). The program will not be terminated on RPC errors, since the condition might be temporary (the Value object server might be restarted later).

The code for the outOfRangeSignalHandler callback has been omitted, since it does not contain anything beyond what valueChangedSignalHandler demonstrates.

Registering for the signals is a two-step process. First, it is necessary to register the interest in the D-Bus signals, and then install the callbacks for the respective GLib signals. This is done within main:

```

/**
 * The test program itself.
 *
 * 1) Setup GType/GSignal
 * 2) Create GMainLoop object
 * 3) Connect to the Session D-Bus
 * 4) Create a proxy GObject for the remote Value object
 * 5) Register signals that we're interested from the Value object
 * 6) Register signal handlers for them
 * 7) Start a timer that will launch timerCallback once per second.
 * 8) Run main-loop (forever)
 */
int main(int argc, char** argv) {

    /*... Listing cut for brevity ...*/

    remoteValue =
        dbus_g_proxy_new_for_name(bus,
                                   VALUE_SERVICE_NAME, /* name */
                                   VALUE_SERVICE_OBJECT_PATH, /* obj path */
                                   VALUE_SERVICE_INTERFACE /* interface */);
    if (remoteValue == NULL) {
        handleError("Couldn't create the proxy object",

```

```

        "Unknown(dbus_g_proxy_new_for_name)", TRUE);
    }

    /* Register the signatures for the signal handlers.
       In our case, we'll have one string parameter passed to use along
       the signal itself. The parameter list is terminated with
       G_TYPE_INVALID (i.e., the GType for string objects. */

    g_print(PROGNAME ":main Registering signal handler signatures.\n");

    /* Add the argument signatures for the signals (needs to be done
       before connecting the signals). This might go away in the future,
       when the GLib-bindings will do automatic introspection over the
       D-Bus, but for now we need the registration phase. */
    { /* Create a local scope for variables. */

        int i;
        const gchar* signalNames[] = { SIGNAL_CHANGED_VALUE1,
                                         SIGNAL_CHANGED_VALUE2,
                                         SIGNAL_OUTOFRANGE_VALUE1,
                                         SIGNAL_OUTOFRANGE_VALUE2 };

        /* Iterate over all the entries in the above array.
           The upper limit for i might seem strange at first glance,
           but is quite common idiom to extract the number of elements
           in a statically allocated arrays in C.
           NOTE: The idiom will not work with dynamically allocated
                 arrays. (Or rather it will, but the result is probably
                 not what you expect.) */
        for (i = 0; i < sizeof(signalNames)/sizeof(signalNames[0]); i++) {
            /* Since the function doesn't return anything, we cannot check
               for errors here. */
            dbus_g_proxy_add_signal(/* Proxy to use */
                                   remoteValue,
                                   /* Signal name */
                                   signalNames[i],
                                   /* Will receive one string argument */
                                   G_TYPE_STRING,
                                   /* Termination of the argument list */
                                   G_TYPE_INVALID);
        }
    } /* end of local scope */

    g_print(PROGNAME ":main Registering D-Bus signal handlers.\n");

    /* We connect each of the following signals one at a time,
       since we'll be using two different callbacks. */

    /* Again, no return values, cannot hence check for errors. */
    dbus_g_proxy_connect_signal(/* Proxy object */
                                remoteValue,
                                /* Signal name */
                                SIGNAL_CHANGED_VALUE1,
                                /* Signal handler to use. Note that the
                                   typecast is just to make the compiler
                                   happy about the function, since the
                                   prototype is not compatible with
                                   regular signal handlers. */
                                G_CALLBACK(valueChangedSignalHandler),
                                /* User-data (we don't use any). */
                                NULL,
                                /* GClosureNotify function that is
                                   responsible in freeing the passed

```

```

        user-data (we have no data). */
        NULL);

dbus_g_proxy_connect_signal(remoteValue, SIGNAL_CHANGED_VALUE2,
                            G_CALLBACK(valueChangedSignalHandler),
                            NULL, NULL);

dbus_g_proxy_connect_signal(remoteValue, SIGNAL_OUTOFRANGE_VALUE1,
                            G_CALLBACK(outOfRangeSignalHandler),
                            NULL, NULL);

dbus_g_proxy_connect_signal(remoteValue, SIGNAL_OUTOFRANGE_VALUE2,
                            G_CALLBACK(outOfRangeSignalHandler),
                            NULL, NULL);

/* All signals are now registered and we're ready to handle them. */
g_print(PROGNAME ":main Starting main loop (first timer in 1s).\n");

/* Register a timer callback that will do RPC sets on the values.
   The userdata pointer is used to pass the proxy object to the
   callback so that it can launch modifications to the object. */
g_timeout_add(1000, (GSourceFunc)timerCallback, remoteValue);

/* Run the program. */
g_main_loop_run(mainloop);
/* Since the main loop is not stopped (by this code), we shouldn't
   ever get here. The program might abort() for other reasons. */

/* If it does, return failure as exit code. */
return EXIT_FAILURE;
}

```

Listing 7.57: glib-dbus-signals/client.c

When adding the argument signatures for the signals (with `dbus_g_proxy_add_signal`), one needs to be very careful with the parameter list. The signal argument types must be exactly the same as are sent from the server (irrespective of the argument specification in the interface XML). This is because the current version of `dbus-bindings-tool` does not generate any checks to enforce signal arguments based on the interface. In this simple case, only one string is received with each different signal, so this is not a big issue. The implementation for the callback function will need to match the argument specification given to the `_add_signal`-function, otherwise data layout on the stack will be incorrect, and cause problems.

Building the client happens in the same manner as before (i.e. `make client`). Since the server is still (hopefully) running on the background, the client will now be started in the same session:

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh ./client
client:main Connecting to Session D-Bus.
client:main Creating a GLib proxy object for Value.
client:main Registering signal handler signatures.
client:main Registering D-Bus signal handlers.
client:main Starting main loop (first timer in 1s).
server:value_object_setvalue1: Called (valueIn=-80)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
client:timerCallback Set value1 to -80
server:value_object_setvalue2: Called (valueIn=-120.000)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
server:value_object_emitSignal: Emitting signal id 3, with message 'value2'
client:timerCallback Set value2 to -120.000
client:value-changed (value1)
server:value_object_getvalue1: Called (internal value1 is -80)
client:value-changed Value1 now -80
client:value-changed (value2)
server:value_object_getvalue2: Called (internal value2 is -120.000)
client:value-changed Value2 now -120.000
client:out-of-range (value2)!
client:out-of-range Value 2 is outside threshold
server:value_object_setvalue1: Called (valueIn=-70)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
client:timerCallback Set value1 to -70
server:value_object_setvalue2: Called (valueIn=-110.000)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
server:value_object_emitSignal: Emitting signal id 3, with message 'value2'
client:timerCallback Set value2 to -110.000
client:value-changed (value1)
server:value_object_getvalue1: Called (internal value1 is -70)
client:value-changed Value1 now -70
client:value-changed (value2)
server:value_object_getvalue2: Called (internal value2 is -110.000)
client:value-changed Value2 now -110.000
...
```

The client will start with the timer callback being executed once per second (as before). Each iteration, it will call the `setvalue1` and `setvalue2` RPC methods with increasing values. The number for `value2` is intentionally set below the minimum threshold, so that it will cause an `outofrange_value2` signal to be emitted. For each set, the `changed_value` signals will also be emitted. Whenever the client receives either of the value change signals, it will perform a `getvalue` RPC method call to retrieve the current value and print it out.

This will continue until the client is terminated.

Tracing D-Bus Signals

Sometimes it is useful to see which signals are actually carried on the buses, especially when adding signal handlers for signals that are emitted from undocumented interfaces. The `dbus-monitor` tool will attach to the D-Bus daemon, and ask it to watch for signals and report them back to the tool, so that it can decode the signals automatically, as they appear on the bus.

While the server and client are still running, the next step is to start the `dbus-monitor` (in a separate session this time) to see whether the signals are transmitted correctly. It should be noted that signals will appear on the bus even if there are no clients currently interested in them. In this case, signals are emitted by the server, based on client-issued RPC methods, so if the client is terminated, the signals will cease.

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-monitor type='signal'
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value1
  string "value1"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value2
  string "value2"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=outofrange_value2
  string "value2"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value1
  string "value1"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value2
  string "value2"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=outofrange_value2
  string "value2"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value1
  string "value1"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value2
  string "value2"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value1
  string "value1"
signal sender=:1.38 -> dest=(null destination)
  interface=org.maemo.Value; member=changed_value2
  string "value2"
```

The tool will automatically decode the parameters to the best of its ability (the string parameter for the signals above). It does not know the semantic meaning for the different signals, so sometimes it will be necessary to perform some additional testing to decide what they actually mean. This is especially true, when mapping out undocumented interfaces (for which there might not be source code available).

Some examples of displaying signals on the system bus on a device follow:

- A device turning off the backlight after inactivity

```
Nokia-N810-xx-xx:~# run-standalone.sh dbus-monitor --system
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=display_status_ind
  string "dimmed"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=system_inactivity_ind
  boolean true
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=save_unsaved_data_ind
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=display_status_ind
  string "off"
```

- A device coming back to life after a screen tap

```
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=system_inactivity_ind
  boolean false
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=display_status_ind
  string "on"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
  interface=com.nokia.mce.signal; member=tklock_mode_ind
  string "unlocked"
```

- A device going into offline mode

```

signal sender=:1.0 -> dest=(null destination)
path=/org/freedesktop/Hal/devices/computer_logicaldev_input_0;
interface=org.freedesktop.Hal.Device; member=Condition
    string "ButtonPressed"
    string "power"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
interface=com.nokia.mce.signal; member=sig_device_mode_ind
    string "offline"
signal sender=:1.26 -> dest=(null destination)
path=/com/nokia/wlancond/signal;
interface=com.nokia.wlancond.signal; member=disconnected
    string "wlan0"
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
interface=com.nokia.icd; member=status_changed
    string "SSID"
    string "WLAN_INFRA"
    string "DISCONNECTING"
    string "com.nokia.icd.error.network_error"
signal sender=:1.22 -> dest=(null destination) path=/org/bluez/hci0;
interface=org.bluez.Adapter; member=ModeChanged
    string "off"
signal sender=:1.25 -> dest=(null destination)
path=/com/nokia/btcond/signal;
interface=com.nokia.btcond.signal; member=hci_dev_down
    string "hci0"

```

- A device going back into normal mode

```

signal sender=:1.0 -> dest=(null destination)
path=/org/freedesktop/Hal/devices/computer_logicaldev_input_0;
interface=org.freedesktop.Hal.Device; member=Condition
    string "ButtonPressed"
    string "power"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
interface=com.nokia.mce.signal; member=sig_device_mode_ind
    string "normal"
signal sender=:1.39 -> dest=(null destination)
path=/org/kernel/class/firmware/hci_h4p;
interface=org.kernel.kevent; member=add
signal sender=:1.39 -> dest=(null destination)
path=/org/kernel/class/firmware/hci_h4p;
interface=org.kernel.kevent; member=remove
signal sender=:1.25 -> dest=(null destination)
path=/com/nokia/btcond/signal;
interface=com.nokia.btcond.signal; member=hci_dev_up
    string "hci0"
signal sender=:1.22 -> dest=(null destination) path=/org/bluez/hci0;
interface=org.bluez.Adapter; member=ModeChanged
    string "connectable"
signal sender=:1.22 -> dest=(null destination) path=/org/bluez/hci0;
interface=org.bluez.Adapter; member=NameChanged
    string "Nokia N810"
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
interface=com.nokia.icd; member=status_changed
    string "SSID"
    string "WLAN_INFRA"
    string "CONNECTING"
    string ""
signal sender=:1.26 -> dest=(null destination)
path=/com/nokia/wlancond/signal;
interface=com.nokia.wlancond.signal; member=connected
    string "wlan0"
    array [
        byte 0
        byte 13
        byte 157
        byte 198
        byte 120
        byte 175
    ]
    int32 536870912
signal sender=:1.100 -> dest=(null destination) path=/com/nokia/eap/signal;
interface=com.nokia.eap.signal; member=auth_status
    uint32 4
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
interface=com.nokia.icd; member=proxies
    uint32 0
    string ""
    int32 0
    string ""
    int32 0
    string ""
    int32 0
    string ""
    int32 0
    string ""
    int32 0
    string ""
    array [ ]
    string ""
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
interface=com.nokia.icd; member=status_changed
    string "SSID"
    string "WLAN_INFRA"
    string "CONNECTED"
    string ""

```

It is also possible to send signals from the command line, which is useful when wanting to emulate some feature of a device inside the SDK. This (like RPC method calls) can be performed with the `dbus-send` tool, and an example of this kind of simulation was given in the LibOSSO section.

7.3.5 Asynchronous GLib/D-Bus

Asynchronicity in D-Bus clients

So far all the RPC method calls that have been implemented here have been "fast", i.e. their execution has not depended on an access to slow services or external resources. In real life, however, it is quite likely that some services cannot be provided immediately, but will have to wait for some external service to complete, before completing the method call.

The GLib wrappers provide a version of making method calls, where the call will be launched (almost) immediately, and a callback will be executed when the method call returns (either with a return value, or an error).

Using the asynchronous wrappers is important, when the program needs to update some kind of status, or be reactive to the user (via a GUI or other interface). Otherwise, the program would block waiting for the RPC method to return, and would not be able to refresh the GUI or screen when required. An alternative solution would be to use separate threads that would run the synchronous methods, but synchronization between threads would become an issue, and debugging threaded programs is much harder than single-threaded ones. Also, implementation of threads might be suboptimal in some environments. These are the reasons why the thread scenario will not be covered here.

Slow running RPC methods will be simulated here by adding a delay into the server method implementations, so that it will become clear why asynchronous RPC mechanisms are important. As signals, by their nature, are asynchronous as well, they do not add anything to this example now. In order to simplify the code listings, the signal support from the asynchronous clients will be dropped here, even though the server still contains them and will emit the.

Slow Test Server

The only change on the server side is the addition of delays into each of the RPC methods (setvalue1, setvalue2, getvalue1 and getvalue2). This delay is added to the start of each function as follows:

```
/* How many microseconds to delay between each client operation. */
#define SERVER_DELAY_USEC (5*1000000UL)

/*... Listing cut for brevity ...*/

gboolean value_object_setvalue1(ValueObject* obj, gint valueIn,
                                GError** error) {

    dbg("Called (valueIn=%d)", valueIn);
    g_assert(obj != NULL);

    dbg("Delaying operation");
    g_usleep(SERVER_DELAY_USEC);

    /* Compare the current value against old one. If they're the same,
       we don't need to do anything (except return success). */
    if (obj->value1 != valueIn) {
```

Listing 7.58: glib-dbus-async/server.c

Building the server is done as before, but we'll notice the delay when we call an RPC method:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > run-standalone.sh ./server &
server:main Connecting to the Session D-Bus.
server:main Registering the well-known name (org.maemo.Platdev_ex)
server:main RequestName returned 1.
server:main Creating one Value object.
server:value_object_class_init: Called
server:value_object_class_init: Creating signals
server:value_object_class_init: Binding to GLib/D-Bus
server:value_object_class_init: Done
server:value_object_init: Called
server:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
server: Not daemonizing (built with NO_DAEMON-build define)
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
method return sender=:1.54 -> dest=:1.56
int32 0

real    0m5.066s
user    0m0.004s
sys     0m0.056s
```

In the example above, the time shell built-in command was used. It will run the given command while measuring the wall clock time (a.k.a. real time), and time used while executing the code and system calls. In this case, only the real time is of any interest. The method call will delay for about 5 seconds, as it should. The delay (even if given with microsecond resolution) is always approximate, and longer than the requested amount. Exact delay will depend on many factors, most of which cannot be directly influenced.

The next experiment deals with a likely scenario, where another method call comes along while the first one is still being executed. This is best tested by just repeating the sending command twice, but running the first one on the background (so that the shell does not wait for it to complete first). The server is still running on the background from the previous test:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue1 &
[2] 17010
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
--type=method_call --print-reply --dest=org.maemo.Platdev_ex \
/GlobalValue org.maemo.Value.getvalue1
method return sender=:1.54 -> dest=:1.57
int32 0

real    0m5.176s
user    0m0.008s
sys     0m0.092s
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
method return sender=:1.54 -> dest=:1.58
int32 0

real    0m9.852s
user    0m0.004s
sys     0m0.052s
```

What can be seen from the above output is that the first client is delayed for about 5 seconds, while the second client (which was launched shortly after the

first) is already delayed by a much longer period. This is to be expected, as the server can only process one request at a time, and will delay each request by 5 seconds.

Some server concurrency issues will be covered later, but for now, it is necessary that the clients are able to continue their "normal work" while they wait for the response from the server. Since this is just example code, "normal work" for our clients would be just waiting for the response, while blocking on incoming events (converted into callbacks). However, if the example programs were graphical, the asynchronous approach would make it possible for them to react to user input. D-Bus by itself does not support cancellation of method calls, once processing has started on the server side, so adding cancellation support would require a separate method call to the server. Since the server only handles one operation at a time, the current server cannot support method call cancellations at all.

Asynchronous Method Calls Using Stubs

When the glib-bindings-tool is run, it will already generate the necessary wrapping stubs to support launching asynchronous method calls. What is then left to do is implementing the callback functions correctly, processing the return errors and launching the method call.

```
/* Pull in the client stubs that were generated with
   dbus-binding-tool */
#include "value-client-stub.h"
```

Listing 7.59: glib-dbus-async/client-stubs.c

The client has been simplified, so that it now only operates on value1. The callback that is called from the stub code is presented next:

```
/**
 * This function will be called when the async setValue1 will either
 * complete, timeout or fail (our server however does not signal
 * errors, but the client D-Bus library might). When this example
 * program is left running for a while, you will see all three cases.
 */
* The prototype must match the one generated by the dbus-binding-tool
* (org_maemo_Value_setValue1_reply).
*
* Since there is no return value from the RPC, the only useful
* parameter that we get is the error object, which we'll check.
* If error is NULL, that means no error. Otherwise the RPC call
* failed and we should check what the cause was.
*/
static void setValue1Completed(DBusGProxy* proxy, GError *error,
                               gpointer userData) {

    g_print(PROGNAME ":%s:setValue1Completed\n", timestamp());
    if (error != NULL) {
        g_printerr(PROGNAME "          ERROR: %s\n", error->message);
        /* We need to release the error object since the stub code does
           not do it automatically. */
        g_error_free(error);
    } else {
        g_print(PROGNAME "          SUCCESS\n");
    }
}
```

Listing 7.60: glib-dbus-async/client-stubs.c

Since the method call does not return any data, the parameters for the callback are at minimum (those three will always be received). Handling errors must be performed within the callback, since errors could be delayed from the server, and not visible immediately at launch time. N.B. The callback will not terminate the program on errors. This is done on purpose in order to demonstrate some common asynchronous problems below. The timestamp function is a small utility function to return a pointer to a string, representing the number of seconds since the program started (useful to visualize the order of the different asynchronous events below).

```
/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * It will launch asynchronous RPC method to set value1 with ever
 * increasing argument.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

    /* Local value that we'll start updating to the remote object. */
    static gint localValue1 = -80;

    /* Start the RPC.
     * This is done by calling the stub function that will take the new
     * value and the callback function to call on reply getting back.
     *
     * The stub returns a DBusGProxyCall object, but we don't need it
     * so we'll ignore the return value. The return value could be used
     * to cancel a pending request (from client side) with
     * dbus_g_proxy_cancel_call. We could also pass a pointer to
     * user-data (last parameter), but we don't need one in this example.
     * It would normally be used to "carry around" the application state.
     */
    g_print(PROGNAME ":%s:timerCallback launching setvalue1\n",
            timestamp());
    org_maemo_Value_setvalue1_async(remoteobj, localValue1,
                                    setValue1Completed, NULL);
    g_print(PROGNAME ":%s:timerCallback setvalue1 launched\n",
            timestamp());

    /* Step the local value forward. */
    localValue1 += 10;

    /* Repeat timer later. */
    return TRUE;
}
```

Listing 7.61: glib-dbus-async/client-stubs.c

Using the stub code is rather simple. For each generated synchronous version of a method wrapper, there will also be a `_async` version of the call. The main difference with the parameters is the removal of the `GError` pointer (since errors will be handled in the callback), and the addition of the callback function to use when the method completes, times out or encounters an error.

The main function remains the same from the previous client examples (a once-per-second timer will be created and run from the mainloop, until the

program is terminated).

Problems with Asynchronicity

When the simple test program is built and run, it can be seen that everything starts off quite well. But at some point, problems will start to appear:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > make client-stubs
dbus-binding-tool --prefix=value_object --mode=glib-client \
  value-dbus-interface.xml > value-client-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include \
  -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -g -Wall \
  -DG_DISABLE_DEPRECATED -DNO_DAEMON -DPROGNAME=\"client-stubs\" \
  -c client-stubs.c -o client-stubs.o
cc client-stubs.o -o client-stubs -ldbus-glib-1 -ldbus-1 -lgobject-2.0
-lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-async] > run-standalone.sh ./client-stubs
client-stubs:main Connecting to Session D-Bus.
client-stubs:main Creating a GLib proxy object for Value.
client-stubs: 0.00:main Starting main loop (first timer in 1s).
client-stubs: 1.00:timerCallback launching setvalue1
client-stubs: 1.00:timerCallback setvalue1 launched
server:value_object_setvalue1: Called (valueIn=-80)
server:value_object_setvalue1: Delaying operation
client-stubs: 2.00:timerCallback launching setvalue1
client-stubs: 2.00:timerCallback setvalue1 launched
client-stubs: 3.01:timerCallback launching setvalue1
client-stubs: 3.01:timerCallback setvalue1 launched
client-stubs: 4.01:timerCallback launching setvalue1
client-stubs: 4.01:timerCallback setvalue1 launched
client-stubs: 5.02:timerCallback launching setvalue1
client-stubs: 5.02:timerCallback setvalue1 launched
server:value_object_emitSignal: Emitting signal id 0, with message '
  value1'
server:value_object_setvalue1: Called (valueIn=-70)
server:value_object_setvalue1: Delaying operation
client-stubs: 6.01:setValue1Completed
client-stubs      SUCCESS
client-stubs: 6.02:timerCallback launching setvalue1
client-stubs: 6.02:timerCallback setvalue1 launched
client-stubs: 7.02:timerCallback launching setvalue1
client-stubs: 7.02:timerCallback setvalue1 launched
...
client-stubs:25.04:timerCallback launching setvalue1
client-stubs:25.04:timerCallback setvalue1 launched
server:value_object_emitSignal: Emitting signal id 0, with message '
  value1'
server:value_object_setvalue1: Called (valueIn=-30)
server:value_object_setvalue1: Delaying operation
client-stubs:26.03:setValue1Completed
client-stubs      SUCCESS
client-stubs:26.05:timerCallback launching setvalue1
client-stubs:26.05:timerCallback setvalue1 launched
client-stubs:27.05:timerCallback launching setvalue1
client-stubs:27.05:timerCallback setvalue1 launched
client-stubs:28.05:timerCallback launching setvalue1
client-stubs:28.05:timerCallback setvalue1 launched
client-stubs:29.05:timerCallback launching setvalue1
client-stubs:29.05:timerCallback setvalue1 launched
client-stubs:30.05:timerCallback launching setvalue1
client-stubs:30.05:timerCallback setvalue1 launched
client-stubs:31.02:setValue1Completed
```

```

client-stubs      ERROR: Did not receive a reply. Possible causes
include:
  the remote application did not send a reply, the message bus security
  policy
  blocked the reply, the reply timeout expired, or the network
  connection was
  broken.
server:value_object_emitSignal: Emitting signal id 0, with message '
  value1'
server:value_object_setvalue1: Called (valueIn=-20)
server:value_object_setvalue1: Delaying operation
client-stubs:31.05:timerCallback launching setvalue1
client-stubs:31.05:timerCallback setvalue1 launched
client-stubs:32.03:setValue1Completed
client-stubs      ERROR: Did not receive a reply. Possible causes
include:
  the remote application did not send a reply, the message bus security
  policy
  blocked the reply, the reply timeout expired, or the network
  connection was
  broken.
client-stubs:32.05:timerCallback launching setvalue1
client-stubs:32.05:timerCallback setvalue1 launched
client-stubs:33.03:setValue1Completed
client-stubs      ERROR: Did not receive a reply. Possible causes
include:
  the remote application did not send a reply, the message bus security
  policy
  blocked the reply, the reply timeout expired, or the network
  connection was
  broken.
client-stubs:33.05:timerCallback launching setvalue1
client-stubs:33.05:timerCallback setvalue1 launched
client-stubs:34.03:setValue1Completed
client-stubs      ERROR: Did not receive a reply. Possible causes
include:
  the remote application did not send a reply, the message bus security
  policy
  blocked the reply, the reply timeout expired, or the network
  connection was
  broken.
client-stubs:34.06:timerCallback launching setvalue1
client-stubs:34.06:timerCallback setvalue1 launched
client-stubs:35.03:setValue1Completed
client-stubs      ERROR: Did not receive a reply. Possible causes
include:
  the remote application did not send a reply, the message bus security
  policy
  blocked the reply, the reply timeout expired, or the network
  connection was
  broken.
client-stubs:35.05:timerCallback launching setvalue1
client-stubs:35.05:timerCallback setvalue1 launched
client-stubs:36.04:setValue1Completed
client-stubs      ERROR: Did not receive a reply. Possible causes
include:
  the remote application did not send a reply, the message bus security
  policy
  blocked the reply, the reply timeout expired, or the network
  connection was
  broken.
server:value_object_emitSignal: Emitting signal id 0, with message '

```

```

    value1'
server:value_object_setvalue1: Called (valueIn=-10)
server:value_object_setvalue1: Delaying operation
client-stubs:36.06:timerCallback launching setvalue1
client-stubs:36.06:timerCallback setvalue1 launched
client-stubs:37.04:setValue1Completed
client-stubs      ERROR: Did not receive a reply. Possible causes
    include:
    the remote application did not send a reply, the message bus security
    policy
    blocked the reply, the reply timeout expired, or the network
    connection was
    broken.
[Ctrl+c]
[sbox-DIABLO_X86: ~/glib-dbus-async] >
server:value_object_emitSignal: Emitting signal id 0, with message '
    value1'
server:value_object_setvalue1: Called (valueIn=30)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message '
    value1'
server:value_object_setvalue1: Called (valueIn=40)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message '
    value1'
server:value_object_setvalue1: Called (valueIn=50)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message '
    value1'
...

```

What happens above is rather subtle. The timer callback in the client launches once per second and performs the RPC method launch. The server, however, still has the 5 second delay for each method call in it. It can be seen that the successive launches go on without any responses for a while. The first response comes back at about 6 seconds from the starting of the client. At this point, the server already has four other outstanding method calls that it has not handled. Slowly the method calls are accumulating at the server end, and it does not deal with them quickly enough to satisfy the client.

After about 30 seconds, it can be seen how the `setValue1Completed` callback is invoked, but the method call fails. This has managed to trigger the method call timeout mechanism. After this point, all the method calls that have accumulated into the server (into a message queue) will fail in the client, since they all will now return late, even if the server actually does handle them.

Once the client is terminated, it can be seen that the server is still happily continuing serving the requests, oblivious to the fact that there is no client to process the responses.

The above test demonstrates quite brutally that the services need to be designed properly, so that there is a clearly defined protocol what to do in case a method call is delayed. It is also advisable to design a notification protocol to tell clients that something has completed, instead of forcing them to time out. Using D-Bus signals is one way, but it is necessary to take care not to generate signals, when there is nothing listening to them. This can be done by only sending signals when an long operation finishes (assuming this has been documented as part of the service description).

One partial fix would be for the client to track and make sure that only one method call to one service is outstanding at any given time. So, instead of just blindly launching the RPC methods, it should defer from launching, if it has not yet received a response from the server (and the call has not timed out).

However, this fix is not complete, since the same problem will manifest itself once there are multiple clients running in parallel and requesting the same methods. The proper fix is to make the server capable of serving multiple requests in parallel. Some hints on how to do this are presented later on.

Asynchronous Method Calls Using GLib Wrappers

Sometimes the interface XML will be missing, so the dbus-bindings-tool cannot be run to generate the stub code. The GLib wrappers are generic enough to enable building own method calls, when necessary.

It is often easiest to start with some known generated stub code to see, which parts can possibly be reused with some modifications. This is what is shown in the last step of this example, in order to make a version of the asynchronous client that will work without the stub generator.

The first step is to take a peek at the stub-generated code for the `setvalue1` call (when used asynchronously):

```
typedef void (*org_maemo_Value_setvalue1_reply) (DBusGProxy *proxy,
                                                GError *error,
                                                gpointer userdata);

static void
org_maemo_Value_setvalue1_async_callback (DBusGProxy *proxy,
                                          DBusGProxyCall *call,
                                          void *user_data)
{
    DBusGAsyncData *data = user_data;
    GError *error = NULL;
    dbus_g_proxy_end_call (proxy, call, &error, G_TYPE_INVALID);
    (*(org_maemo_Value_setvalue1_reply)data->cb) (proxy, error,
                                                  data->userdata);
    return;
}

static
#ifdef G_HAVE_INLINE
inline
#endif
DBusGProxyCall*
org_maemo_Value_setvalue1_async (DBusGProxy *proxy,
                                const gint IN_new_value,
                                org_maemo_Value_setvalue1_reply callback,
                                gpointer userdata)
{
    DBusGAsyncData *stuff;
    stuff = g_new (DBusGAsyncData, 1);
    stuff->cb = G_CALLBACK (callback);
    stuff->userdata = userdata;
    return dbus_g_proxy_begin_call (
        proxy, "setvalue1", org_maemo_Value_setvalue1_async_callback,
        stuff, g_free, G_TYPE_INT, IN_new_value, G_TYPE_INVALID);
}
```

Listing 7.62: glib-dbus-async/value-client-stub.h

What is notable in the code snippet above is that the `_async` method will create a temporary small structure that will hold the pointer to the callback function, and a copy of the userdata pointer. This small structure will then be passed to `dbus_g_proxy_begin_call`, along with the address of the generated callback wrapper function (`org_maemo_Value_setvalue1_async_callback`). The GLib async launcher will also take a function pointer to a function to use when the supplied "user-data" (in this case, the small structure) will need to be disposed of after the call. Since it uses `g_new` to allocate the small structure, it passes `g_free` as the freeing function. Next comes the argument specification for the method call, which obeys the same rules as the LibOSSO ones before.

On RPC completion, the generated callback will be invoked, and it will get the real callback function pointer and the userdata as its "user-data" parameter. It will first collect the exit code for the call with `dbus_g_proxy_end_call`, unpack the data and invoke the real callback. After returning, the GLib wrappers (which called the generated callback) will call `g_free` to release the small structure, and the whole RPC launch will end.

The next step is to re-implement pretty much the same logic, but also dispose of the small structure, since the callback will be implemented directly, not as a wrapper-callback (it also omits the need for one memory allocation and one free).

The first step for that is to implement the RPC asynchronous launch code:

```
/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * It will launch asynchronous RPC method to set value1 with ever
 * increasing argument.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

    /* Local value that we'll start updating to the remote object. */
    static gint localValue1 = -80;

    /* Start the first RPC.
     * The call using GLib/D-Bus is only slightly more complex than the
     * stubs. The overall operation is the same. */
    g_print(PROGNAME ":timerCallback launching setvalue1\n");
    dbus_g_proxy_begin_call(remoteobj,
        /* Method name. */
        "setvalue1",
        /* Callback to call on "completion". */
        setValue1Completed,
        /* User-data to pass to callback. */
        NULL,
        /* Function to call to free userData after
         * callback returns. */
        NULL,
        /* First argument GType. */
        G_TYPE_INT,
        /* First argument value (passed by value) */
        localValue1,
        /* Terminate argument list. */
        G_TYPE_INVALID);
```

```

g_print(PROGNAME ":timerCallback setValue1 launched\n");

/* Step the local value forward. */
localValue1 += 10;

/* Repeat timer later. */
return TRUE;
}

```

Listing 7.63: glib-dbus-async/client-glib.c

And the callback that will be invoked on method call completion, timeouts or errors:

```

/**
 * This function will be called when the async setValue1 will either
 * complete, timeout or fail (same as before). The main difference in
 * using GLib/D-Bus wrappers is that we need to "collect" the return
 * value (or error). This is done with the _end_call function.
 *
 * Note that all callbacks that are to be registered for RPC async
 * notifications using dbus_g_proxy_begin_call must follow the
 * following prototype: DBusGProxyCallNotify .
 */
static void setValue1Completed(DBusGProxy* proxy,
                               DBusGProxyCall* call,
                               gpointer userData) {

    /* This will hold the GError object (if any). */
    GError* error = NULL;

    g_print(PROGNAME ":setValue1Completed\n");

    /* We next need to collect the results from the RPC call.
     * The function returns FALSE on errors (which we check), although
     * we could also check whether error-ptr is still NULL. */
    if (!dbus_g_proxy_end_call(proxy,
                               /* The call that we're collecting. */
                               call,
                               /* Where to store the error (if any). */
                               &error,
                               /* Next we list the GType codes for all
                                * the arguments we expect back. In our
                                * case there are none, so set to
                                * invalid. */
                               G_TYPE_INVALID)) {
        /* Some error occurred while collecting the result. */
        g_printerr(PROGNAME " ERROR: %s\n", error->message);
        g_error_free(error);
    } else {
        g_print(PROGNAME " SUCCESS\n");
    }
}

```

Listing 7.64: glib-dbus-async/client-glib.c

The generated stub code is no longer needed, so the dependency rules for the stubless GLib version will also be somewhat different:

```

client-glib: client-glib.o
             $(CC) $^ -o $@ $(LDFLAGS)
# Note that the GLib client doesn't need the stub code.
client-glib.o: client-glib.c common-defs.h

```

```
$(CC) $(CFLAGS) -DPROGNAME=\"$(basename $@)\\" -c $< -o $@
```

Listing 7.65: glib-dbus-async/Makefile

Since the example program logic has not changed from the previous version, testing client-glib is not presented here (it can of course be tested if so desired, since the source code contains the fully working program). This version of the client will also launch the method calls without waiting for previous method calls to complete.

7.3.6 D-Bus Server Design Issues

Definition of Server

When talking about software, a server is commonly understood to mean some kind of software component that provides a service to its clients. In Linux, servers are usually implemented as daemons, which is a technical term for a process that has detached from the terminal session, and performed other preparatory actions, so that it will stay running on the background until it terminates (or is terminated).

Sometimes people might refer to servers as engines, but it is a more generic term, and normally is not related directly to the way a service is implemented (as a separate process, or as part of some library, directly used from within a client process). Broadly defined, an engine is the part of application that implements the functionality, but not the interface, of an application. In Model-View-Controller, it would be the Model.

The servers in these examples have so far been running without daemonization, in order to display debugging messages on the terminal/screen more easily. Often a server can be started with a "--stay-on-foreground" option (or -f or something similar), which means that they will not daemonize. This is a useful feature to have, since it will allow the use of simpler outputting primitives, when testing the software.

By default, when a server daemonizes, its output and input files will be closed, so reading user input (from the terminal session, not GUI) will fail, as will each output write (including printf and g_print).

Daemonization

The objective of turning a process into a daemon is to detach it from its parent process, and create a separate session for it. This is necessary, so that parent termination does not automatically cause the termination of the server as well. There is a library call that will perform most of the daemonization work, called daemon, but it is also instructive to see what is necessary (and common) to do when implementing the functionality oneself:

- Fork the process, so that the original process can be terminated and this will cause the child process to move under the system init process.
- Create a new session for the child process with setsid.
- Possibly switch working directory to root (/), so that the daemon will not keep file systems from being unmounted.

- Set up a restricted umask, so that directories and files that are created by the daemon (or its child processes) will not create publicly accessible objects in the filesystem. In Internet Tablets, this does not really apply, since the devices only have one user.
- Close all standard I/O file descriptors (and preferably also files), so that if the terminal device closes (user logs out), it will not cause SIGPIPE signals to the daemon, when it next accesses the file descriptors (by mistake or intentionally because of g_print/printf). It is also possible to reopen the file descriptors, so that they will be connected to a device, which will just ignore all operations (like /dev/null that is used with daemon).

The daemon function allows to select, whether a change of the directory is wanted, and to close the open file descriptors. That will utilize in the servers of this example in the following way:

```
#ifndef NO_DAEMON

/* This will attempt to daemonize this process. It will switch this
process working directory to / (chdir) and then reopen stdin,
stdout and stderr to /dev/null. Which means that all printouts
that would occur after this, will be lost. Obviously the
daemonization will also detach the process from the controlling
terminal as well. */
if (daemon(0, 0) != 0) {
    g_error(PROGNAME ": Failed to daemonize.\n");
}
#else
g_print(PROGNAME
        ": Not daemonizing (built with NO_DAEMON-build define)\n");
#endif
```

Listing 7.66: glib-dbus-sync/server.c

This define is then available to the user inside the Makefile:

```
# -DNO_DAEMON : do not daemonize the server (on a separate line so can
#               be disabled just by commenting the line)
ADD_CFLAGS += -DNO_DAEMON

# Combine flags
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
```

Listing 7.67: glib-dbus-sync/Makefile

Combining the options so that CFLAGS is appended to the Makefile provided defaults allows the user to override the define as well:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > CFLAGS='-UNO_DAEMON' make server
dbus-binding-tool --prefix=value_object --mode=glib-server \
value-dbus-interface.xml > value-server-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
-g -Wall -DG_DISABLE_DEPRECATED -DNO_DAEMON -UNO_DAEMON
-DPROGNAME=\"server\" -c server.c -o server.o
cc server.o -o server -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
```

Since all -D and -U options will be processed from left to right by gcc, this allows the -UNO_DAEMON to undefine the symbol that is preset in the Makefile. If the user does not know this technique, it is also possible to edit the

Makefile directly. Grouping all additional flags that the user might be interested in to the top of the Makefile will make this simpler (for the user).

Running the server with daemonization support is performed as before, but this time the `&` (do not wait for child exit) token for the shell will be left out:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh ./server
server:main Connecting to the Session D-Bus.
server:main Registering the well-known name (org.maemo.Platdev_ex)
server:main RequestName returned 1.
server:main Creating one Value object.
server:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
[sbox-DIABLO_X86: ~/glib-dbus-sync] >
```

Since server messages will not be visible any more, some other mechanism is needed to find out that the server is still running:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > ps aux | grep "\./server" | grep -v pts
user 8982  0.0  0.1 2780 664 ? Ss 00:14 0:00 ./server
```

The slightly convoluted way of using `grep` was necessary to only list those lines of the `ps` report, which have `./server` in them, and to remove the lines which do not have `pts` in them (so that it is possible to see processes which have no controlling terminals).

The client could have been used to test, whether the server responds, but the above technique is slightly more general. If the `pstree` tool is available, it could be run it with `-pu` options to see how the processes relate to each other, and that the daemonized server is running directly as a child of `init` (which was the objective of the fork).

Event Loops and Power Consumption

Most modern CPUs (even for desktops and servers) allow multiple levels of power savings to be selected. Each of the levels will be progressively more power-conservative, but there is always a price involved. The deeper the power saving level required, the more time it normally takes to achieve it, and the more time it also takes to come out of it. In some CPUs, it also requires special sequences of instructions to run, hence taking extra power itself.

All this means that changing the power state of the CPU should be avoided whenever possible. Obviously, this is in contrast to the requirement to conserve battery life, so in effect, what is needed is to require the attention of the CPU as rarely as possible.

One way at looking the problem field is contrasting event-based and polling-based programming. Code that continuously checks for some status, and only occasionally performs useful work, is clearly keeping the CPU from powering down properly. This model should be avoided at all cost, or at least its use should be restricted to bare minimum, if no other solution is possible.

In contrast, event-based programming is usually based on the execution of callback functions when something happens, without requiring a separate polling loop. This then leaves the question of how to trigger the callbacks, so that they will be issued, when something happens. Using timer callbacks might seem like a simple solution, so that it continuously (once per second or more often) checks for status, and then possibly reacts to the change in status. This model is undesirable as well, since the CPU will not be able to enter into deep sleep modes, but fluctuate between full power and high-power states.

Most operating system kernels provide a mechanism (or multiple mechanisms) by which a process can be woken up when data is available, and kept off the running queue of the scheduler otherwise. The most common mechanism in Linux is based around the select/poll system calls, which are useful when waiting for a change in status for a set of file descriptors. Since most of the interesting things in Linux can be represented as a "file" (an object supporting read and write system calls), using select and poll is quite common. However, when writing software that uses GLib (implicitly like in GTK+ or explicitly like in the non-GUI examples in this document), the GMainLoop structure will be used instead. Internally, it will use the event mechanism available on the platform (select/poll/others), but the program will need to register callbacks, start the main loop execution and then just execute the callbacks as they come.

If there are some file descriptors (network sockets, open files, etc), they can be integrated into the GMainLoop using GIOChannels (please see the GLib API reference on this).

This still leaves the question of using timers and callbacks that are triggered by timers. They should be avoided when:

- You plan to use the timer at high frequencies (> 1 Hz) for long periods of time (> 5 sec).
- There is a mechanism that will trigger a callback when something happens, instead of forcing you to poll for the status "manually" or re-execute a timer callback that does the checking.

As an example, the LibOSSO program (FlashLight) that was covered before, will have to use timers in order to keep the backlight active. However, the timer is very slow (only once every 45 seconds), so this is not a big issue. Also, in flashlight's defense, the backlight is on all the time, so having a slow timer will not hurt battery life very much anyway.

Another example could be a long-lasting download operation, which proceeds slowly, but steadily. It would be advisable to consider, whether updating a progress bar after each small bit of data is received makes sense (normally it does not). Instead, it is better to keep track of when was the last time when the progress bar was updated, and if enough time has passed since the last time, update the GUI. In some cases, this will allow the CPU to be left in a somewhat lower power state than full-power, and will allow it to fall back to sleep more quickly.

Having multiple separate programs running, each having their own timers, presents another interesting problem. Since the timer callback is not precise, at some time the system will be waking at a very high frequency, handling each timer separately (the frequency and the number of timers executing in the system is something that cannot be controlled from a single program, but instead is a system-wide issue).

If planning a GUI program, it is fairly easy to avoid contributing to this problem, since it is possible to get a callback from LibOSSO, which will tell when the program is "on top", and when it is not visible. When not visible, the GUI does not need to be updated, especially with timer-based progress indicators and such.

Since servers do not have a GUI (and their visibility is not controlled by the window manager), such mechanism does not exist. One possible solution

in this case would be avoiding using timers (or any resources for that matter), when the server does not have any active clients. Resources should only be used when there is a client connection, or there is a need to actually do something. As soon as it becomes likely that the server will not be used again, the resources should be released (timer callbacks removed, etc.).

If possible, one should try and utilize the D-Bus signals available on the system bus (or even the hardware state structure via LibOSSO) to throttle down activity based on the conditions in the environment. Even if making a non-GUI server, the system shutdown signals should be listened to, as they will tell the process to shutdown gracefully.

All in all, designing for a dynamic low-powered environment is not always simple. Four simple rules will hold for most cases (all of them being important):

- Avoid doing extra work when possible.
- Do it as fast as possible (while trying to minimize resource usage).
- Do it as rarely as possible.
- Keep only those resources allocated that you need to get the work done.

For GUI programs, one will have to take into account the "graphical side" of things as well. Making a GUI that is very conservative in its power usage will, most of the time, be very simple, provide little excitement to users and might even look quite ugly. The priorities for the programmer might lie in a different direction.

Supporting Parallel Requests

The value object server with delays has one major deficiency: it can only handle one request at a time, while blocking the progress of all the other requests. This will be a problem, if multiple clients use the same server at the same time.

Normally one would add support for parallel requests by using some kind of multiplexing mechanism right on top of the message delivery mechanism (in this case, libdbus).

One can group the possible solutions around three models:

- Launching a separate thread to handle each request. This might seem like an easy way out of the problem, but coordinating access to shared resources (object states in this case) between multiple threads is prone to cause synchronization problems, and makes debugging much harder. Also, performance of such an approach would depend on efficient synchronization primitives in the platform (which might not always be available), as well as lightweight thread creation and tear-down capabilities of the platform.
- Using an event-driven model that supports multiple event sources simultaneously and "wakes up" only when there is an event on any of the event sources. The select and poll (and epoll on Linux) are very often used in these cases. Using them will normally require an application design that is driven by the requirements of the system calls (i.e. it is very difficult to retrofit them into existing "linear" designs). However, the event-based

approach normally outperforms the thread approach, since there is no need for synchronization (when implemented correctly), and there will only be one context to switch from the kernel and back (there will be extra contexts with threads). GLib provides a high-level abstraction on top of the low-level event programming model, in the form of GMainLoop. One would use GIOChannel objects to represent each event source, and register callbacks that will be triggered on the events.

- Using fork to create a copy of the server process, so that the new copy will just handle one request and then terminate (or return to the pool of "servers"). The problem here is the process creation overhead, and the lack of implicit sharing of resources between the processes. One would have to arrange a separate mechanism for synchronization and data sharing between the processes (using shared memory and proper synchronization primitives). In some cases, resource sharing is not actually required, or happens at some lower level (accessing files), so this model should not be automatically ruled out, even if it seems quite heavy at first. Many static content web servers use this model because of its simplicity (and they do not need to share data between themselves).

However, the problem for the slow server lies elsewhere: the GLib/D-Bus wrappers do not support parallel requests directly. Even using the fork model would be problematic, as there would be multiple processes accessing the same D-Bus connection. Also, this problem is not specific to the slow server only. The same issues will be encountered when using other high-level frameworks (such as GTK+) whenever they cannot complete something immediately, because not all data is present in the application. In the latter case, it is normally sufficient to use the GMainLoop/GIOChannel approach in parallel with GTK+ (since it uses GMainLoop internally anyway), but with GLib/D-Bus there is no mechanism which could be used to integrate own multiplexing code (no suitable API exists).

In this case, the solution would be picking one of the above models, and then using libdbus functions directly. In effect, this would require a complete rewrite of the server, forgetting about the GType implementation, and possibly creating a light-weight wrapper for integrating libdbus functions into GLib GMainLoop mechanism (but dropping support for GType).

Dropping support for the GType and stub code will mean that it would be necessary to implement the introspection support manually and be dependent on possible API changes in libdbus in the future.

Another possible solution would be to "fake" the completion of client method calls, so that the RPC method would complete immediately, but the server would continue (using GIOChannel integration) processing the request, until it really completes. The problem in this solution is that it is very difficult to know, which client actually issued the original method call, and how to communicate the final result (or errors) of the method call to the client once it completes. One possible model here would be using signals to broadcast the end result of the method call, so that the client would get the result at some point (assuming the client is still attached to the message bus). Needless to say, this is quite inelegant and difficult to implement correctly, especially since sending signals will cause unnecessary load by waking up all the clients on the bus (even if they are not interested in that particular signal).

In short, there is no simple solution that works properly when GLib/D-Bus wrappers are used.

Debugging

The simplest way to debug servers is intelligent use of print out of events in the code sections that are relevant. Tracing everything that goes on rarely makes sense, but having a reliable and working infrastructure (in code level) will help. One such mechanism is utilizing various built-in tricks that gcc and cpp provide. In the server example, a macro called `dbg` is used, which will expand to `g_print`, when the server is built as non-daemonizing version. If the server becomes a daemon, the macro expands to "nothing", meaning that no code will be generated to format the parameters, or to even access them. It is advisable to extend this idea to support multiple levels of debugging, and possibly use different "subsystem" identifiers, so that a single subsystem can be switched on or off, depending on what it is that is to be debugged.

The `dbg` macro utilizes the `__func__` symbol, which expands to the function name where the macro will be expanded, which is quite useful so that the function name does not need to be explicitly added:

```
/* A small macro that will wrap g_print and expand to empty when
server will daemonize. We use this to add debugging info on
the server side, but if server will be daemonized, it does not
make sense to even compile the code in.

The macro is quite "hairy", but very convenient. */
#ifdef NO_DAEMON
#define dbg(fmtstr, args...) \
    (g_print(PROGNAME ":%s: " fmtstr "\n", __func__, ##args))
#else
#define dbg(dummy...)
#endif
```

Listing 7.68: glib-dbus-sync/server.c

Using the macro is then quite simple, as it will look and act like a regular printf-formatting function (`g_print` included):

```
dbg("Called (internal value2 is %.3f)", obj->value2);
```

Listing 7.69: glib-dbus-sync/server.c

The only small difference here is that it is not necessary to explicitly add the trailing newline (`\n`) into each call, since it will be automatically added.

Assuming `NO_DAEMON` is defined, the macro would expand to the following output when the server was run:

```
server:value_object_getvalue2: Called (internal value2 is 42.000)
```

For larger projects, it is advisable to combine `__file__`, so that tracing multifile programs will become easier.

Coupled with proper test cases (which would be using the client code, and possibly also `dbus-send` in D-Bus related programs), this is a very powerful technique, and often much easier than single stepping through the code with a debugger (gdb), or setting evaluation breakpoints. It can also be of interest to use Valgrind to help detecting memory leaks (and some other errors).

More information on these topics and examples is available in section *Maemo Debugging Guide* 14.2 of chapter *Debugging* on Maemo Reference Manual.

7.4 Application Preferences - GConf

GConf is used by the GNOME desktop environment for storing shared configuration settings for the desktop and applications. The daemon process GConfd follows the changes in the database. When a change occurs in the database, the daemon applies the new settings to the applications using them. For example, the control panel application uses GConf.

If settings are used only by a single application, Glib utility for .ini style files should be used instead. Applications should naturally have working default settings. Settings should be saved only when the user changes them.

GConf Basics

GConf is a system for GNOME applications to store settings into a database system in a centralized manner. The aim of the GConf library is to provide applications a consistent view of the database access functions, as well as to provide tools for system administrators to enable them to distribute software settings in a centralized manner (across multiple computers).

The GConf "database" may consist of multiple databases (configured by the system administrator), but normally there will be at least one database engine that uses XML to store settings. This keeps the database still in a human readable form (as opposed to binary), and allows some consistency checks via schema verifications.

The interface for the client (program that uses GConf to store its settings) is always the same, irrespective of the database back-end (the client does not see this).

What makes GConf interesting, is its capability of notifying running clients that their settings have been changed by some other process than themselves. This allows for the clients to react soon (not quite real-time), and this leads to a situation where a user will change, for example, the GNOME HTTP proxy settings, and clients that are interested in that setting will get a notification (via a callback function) that the setting has changed. The clients will then read the new setting and modify their data structures to take the new setting into account.

7.4.1 Using GConf

The GConf model consists of two parts: the GConf client library (which will be used here) and the GConf server daemon that is the guardian and reader/writer of the back-end databases. In a regular GNOME environment, the client communicates with the server either by using the Bonobo library (lightweight object IPC mechanism) or D-Bus.

As Bonobo is not used in maemo (it is quite heavy, even if lightweight), the client will communicate with the server using D-Bus. This also allows the daemon to be started on demand, when there is at least one client wanting to use that service (this is a feature of D-Bus). The communication mechanism is encapsulated by the GConf client library, and as such, will be transparent.

In order to read or write the preference database, it is necessary to decide on the key to use to access the application values. The database namespace is hierarchical, and uses the '/'-character to implement this hierarchy, starting from a root location similar to UNIX file system namespace.

Each application will use its own "directory" under **/apps/Maemo/app-name/**. N.B. Even when the word "directory" is seen in connection to GConf, one has to be careful to distinguish **real directories** from **preference namespaces** inside the GConf namespace. The **/apps/Maemo/appname/** above is in the GConf namespace, so there will not actually be a physical directory called **/apps/** on a system.

The keys should be named according to the platform guidelines. The current guideline is that each application should store its configuration keys under **/apps/Maemo/appname/**, where appname is the name of the application. There is no central registry on the names in use currently, so names should be selected carefully. Key names should all be lowercase, with underscore used to separate multiple words. Also, ASCII should be used, since GConf does not support localization for key names (it does for key values, but that is not covered in this material).

GConf values are typed, which means that it is necessary to select the type for the data that the key is supposed to hold.

The following types are supported for values in GConf:

- gint (32-bit signed)
- gboolean
- gchar (ASCII/ISO 8859-1/UTF-8 C string)
- gfloat (with the limitation that the resolution is not guaranteed nor specified by GConf because of portability issues)
- a list of values of one type
- a pair of values, each having their own type (useful for storing "mapping" data)

What is missing from the above list is storing binary data (for a good reason). The type system is also fairly limited. This is on purpose, so that complex configurations (like the Apache HTTP daemon uses, or Samba) are not attempted using GConf.

There is a diagnostic and administration tool called gconftool-2 that is also available in the SDK. It can be used to set and unset keys, as well as display their current contents.

Some examples of using gconftool-2 (on the SDK):

- Displaying the contents of all keys stored under **/apps/** (listing cut for brevity)

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 -R /apps
/apps/osso:
/apps/osso/inputmethod:
  launch_finger_kb_on_select = true
  input_method_plugin = himExample_vkb
  available_languages = [en_GB]
  use_finger_kb = true
/apps/osso/inputmethod/hildon-im-languages:
  language-0 = en_GB
  current = 0
  language-1 =
  list = []
/apps/osso/fontconfig:
  font_scaling_factor = Schema (type: 'float' list_type:
    '*invalid*' car_type: '*invalid*' cdr_type: '*invalid*'
    locale: 'C')
/apps/osso/apps:
/apps/osso/apps/controlpanel:
  groups = [copa_ia_general,copa_ia_connectivity,
    copa_ia_personalisation]
  icon_size = false
  group_ids = [general,connectivity,personalisation]
/apps/osso/osso:
/apps/osso/osso/thumbnailers:
/apps/osso/osso/thumbnailers/audio@x-mp3:
  command = /usr/bin/hildon-thumb-libid3
/apps/osso/osso/thumbnailers/audio@x-m4a:
  command = /usr/bin/hildon-thumb-libid3
/apps/osso/osso/thumbnailers/audio@mp3:
  command = /usr/bin/hildon-thumb-libid3
/apps/osso/osso/thumbnailers/audio@x-mp2:
  command = /usr/bin/hildon-thumb-libid3
```

- Creating and setting the value to a new key.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
--set /apps/Maemo/testing/testkey --type=int 5
```

- Listing all keys under the namespace **/apps/Maemo/testing**.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
-R /apps/Maemo/testing
testkey = 5
```

- Removing the last key will also remove the key directory.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
--unset /apps/Maemo/testing/testkey
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
-R /apps/Maemo/testing
```

- Removing whole key hierarchies is also possible.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
--recursive-unset /apps/Maemo/testing
```

For more detailed information, please see GConf API documentation [\[57\]](#).

7.4.2 Using GConf to Read and Write Preferences

Section *Application Settings* 6.9.3 of chapter *Application Development* in Maemo Reference Manual presents a short introductory example of gconf usage. The following example is a bit more complicated.

The example is required to:

- Store the color that the user selects when the color button (in the toolbar) is used.

- Load the color preference on application startup.

Even if GConf concepts seem to be logical, it can be seen that using GConf will require one to learn some new things (e.g. the GError-object). Since the GConf client code is in its own library, the relevant compiler flags and library options need to be added again. The pkg-config package name is gconf-2.0

```
/**
 * hildon_helloworld-9.c
 *
 * 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.
 *
 * We'll store the color that the user selects into a GConf
 * preference. In fact, we'll have three settings, one for each
 * channel of the color (red, green and blue).
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
 */

#include <stdlib.h>
#include <hildon/hildon-program.h>
#include <hildon/hildon-color-button.h>
#include <hildon/hildon-find-toolbar.h>
#include <hildon/hildon-file-chooser-dialog.h>
#include <hildon/hildon-banner.h>
#include <libgnomevfs/gnome-vfs.h>
/* Include the prototypes for GConf client functions (NEW). */
#include <gconf/gconf-client.h>

/* The application name -part of the GConf namespace (NEW). */
#define APP_NAME "hildon_hello"
/* This will be the root "directory" for our preferences (NEW). */
#define GC_ROOT "/apps/Maemo/" APP_NAME "/"

/*... Listing cut for brevity ...*/

/**
 * NEW
 *
 * Utility function to store the given color into our application
 * preferences. We could use a list of integers as well, but we'll
 * settle for three separate properties; one for each of RGB
 * channels.
 *
 * The config keys that will be used are 'red', 'green' and 'blue'.
 *
 * NOTE:
 * We're doing things very non-optimally. If our application would
 * have multiple preference settings, and we would like to know
 * when someone will change them (external program, another
 * instance of our program, etc), we'd have to keep a reference to
 * the GConf client connection. Listening for changes in
 * preferences would also require a callback registration, but this
 * is covered in the "maemo Platform Development" material.
 */
static void confStoreColor(const GdkColor* color) {

    /* We'll store the pointer to the GConf connection here. */
    GConfClient* gcClient = NULL;
```

```

/* Make sure that no NULLs are passed for the color. GdkColor is
   not a proper GObject, so there is no GDK_IS_COLOR macro. */
g_assert(color);

g_print("confStoreColor: invoked\n");

/* Open a connection to gconfd-2 (via D-Bus in maemo). The GConf
   API doesn't say whether this function can ever return NULL or
   how it will behave in error conditions. */
gcClient = gconf_client_get_default();
/* We make sure that it's a valid GConf-client object. */
g_assert(GCONF_IS_CLIENT(gcClient));

/* Store the values. */
if (!gconf_client_set_int(gcClient, GC_ROOT "red", color->red,
                          NULL)) {
    g_warning(" failed to set %s/red to %d\n", GC_ROOT, color->red);
}
if (!gconf_client_set_int(gcClient, GC_ROOT "green", color->green,
                          NULL)) {
    g_warning(" failed to set %s/green to %d\n", GC_ROOT,
              color->green);
}
if (!gconf_client_set_int(gcClient, GC_ROOT "blue", color->blue,
                          NULL)) {
    g_warning(" failed to set %s/blue to %d\n", GC_ROOT,
              color->blue);
}

/* Release the GConf client object (with GObject-unref). */
g_object_unref(gcClient);
gcClient = NULL;
}

/**
 * NEW
 *
 * A utility function to get an integer but also return the status
 * whether the requested key existed or not.
 *
 * NOTE:
 * It's also possible to use gconf_client_get_int(), but it's not
 * possible to then know whether the key existed or not, because
 * the function will return 0 if the key doesn't exist (and if the
 * value is 0, how could you tell these two conditions apart?).
 *
 * Parameters:
 * - GConfClient: the client object to use
 * - const gchar*: the key
 * - gint*: the address to store the integer to if the key exists
 *
 * Returns:
 * - TRUE: if integer has been updated with a value from GConf.
 * - FALSE: there was no such key or it wasn't an integer.
 */
static gboolean confGetInt(GConfClient* gcClient, const gchar* key,
                           gint* number) {

    /* This will hold the type/value pair at some point. */
    GConfValue* val = NULL;
    /* Return flag (tells the caller whether this function wrote behind

```

```

        the 'number' pointer or not). */
gboolean hasChanged = FALSE;

/* Try to get the type/value from the GConf DB.
NOTE:
    We're using a version of the getter that will not return any
    defaults (if a schema would specify one). Instead, it will
    return the value if one has been set (or NULL).

    We're not really interested in errors as this will return a NULL
    in case of missing keys or errors and that is quite enough for
    us. */
val = gconf_client_get_without_default(gcClient, key, NULL);
if (val == NULL) {
    /* Key wasn't found, no need to touch anything. */
    g_warning("confGetInt: key %s not found\n", key);
    return FALSE;
}

/* Check whether the value stored behind the key is an integer. If
it is not, we issue a warning, but return normally. */
if (val->type == GCONF_VALUE_INT) {
    /* It's an integer, get it and store. */
    *number = gconf_value_get_int(val);
    /* Mark that we've changed the integer behind 'number'. */
    hasChanged = TRUE;
} else {
    g_warning("confGetInt: key %s is not an integer\n", key);
}

/* Free the type/value-pair. */
gconf_value_free(val);
val = NULL;

return hasChanged;
}

/**
 * NEW
 *
 * Utility function to change the given color into the one that is
 * specified in application preferences.
 *
 * If some key is missing, that channel is left untouched. The
 * function also checks for proper values for the channels so that
 * invalid values are not accepted (guint16 range of GdkColor).
 *
 * Parameters:
 * - GdkColor*: the color structure to modify if changed from prefs.
 *
 * Returns:
 * - TRUE if the color was been changed by this routine.
 * - FALSE if the color wasn't changed (there was an error or the
 *   color was already exactly the same as in the preferences).
 */
static gboolean confLoadCurrentColor(GdkColor* color) {

    GConfClient* gcClient = NULL;
    /* Temporary holders for the pref values. */
    gint red = -1;
    gint green = -1;
    gint blue = -1;

```

```

/* Temp variable to hold whether the color has changed. */
gboolean hasChanged = FALSE;

g_assert(color);

g_print("confLoadCurrentColor: invoked\n");

/* Open a connection to gconfd-2 (via d-bus). */
gcClient = gconf_client_get_default();
/* Make sure that it's a valid GConf-client object. */
g_assert(GCONF_IS_CLIENT(gcClient));

if (confGetInt(gcClient, GC_ROOT "red", &red)) {
    /* We got the value successfully, now clamp it. */
    g_print(" got red = %d, ", red);
    /* We got a value, so let's limit it between 0 and 65535 (the
       legal range for guint16). We use the CLAMP macro from GLib for
       this. */
    red = CLAMP(red, 0, G_MAXUINT16);
    g_print("after clamping = %d\n", red);
    /* Update & mark that at least this component changed. */
    color->red = (guint16)red;
    hasChanged = TRUE;
}
/* Repeat the same logic for the green component. */
if (confGetInt(gcClient, GC_ROOT "green", &green)) {
    g_print(" got green = %d, ", green);
    green = CLAMP(green, 0, G_MAXUINT16);
    g_print("after clamping = %d\n", green);
    color->green = (guint16)green;
    hasChanged = TRUE;
}
/* Repeat the same logic for the last component (blue). */
if (confGetInt(gcClient, GC_ROOT "blue", &blue)) {
    g_print(" got blue = %d, ", blue);
    blue = CLAMP(blue, 0, G_MAXUINT16);
    g_print("after clamping = %d\n", blue);
    color->blue = (guint16)blue;
    hasChanged = TRUE;
}

/* Release the client object (with GObject-unref). */
g_object_unref(gcClient);
gcClient = NULL;

/* Return status if the color was been changed by this routine. */
return hasChanged;
}

/**
 * MODIFIED
 *
 * Invoked when the user selects a color (or will cancel the dialog).
 *
 * Will also write the color to preferences (GConf) each time the
 * color changes. We'll compare whether it has really changed (to
 * avoid writing to GConf is nothing really changed).
 */
static void cbActionColorChanged(HildonColorButton* colorButton,
                                ApplicationState* app) {

    /* Local variables that we'll need to handle the change (NEW). */

```



```

gboolean hasChanged = FALSE;
GdkColor newColor = {};
GdkColor* curColor = NULL;

g_assert(app != NULL);

g_print("cbActionColorChanged invoked\n");
/* Retrieve the new color from the color button (NEW). */
hildon_color_button_get_color(colorButton, &newColor);
/* Just an alias to save some typing (could also use
   app->currentColor) (NEW). */
curColor = &app->currentColor;

/* Check whether the color really changed (NEW). */
if ((newColor.red != curColor->red) ||
    (newColor.green != curColor->green) ||
    (newColor.blue != curColor->blue)) {
    hasChanged = TRUE;
}
if (!hasChanged) {
    g_print(" color not really changed\n");
    return;
}
/* Color really changed, store to preferences (NEW). */
g_print(" color changed, storing into preferences.. \n");
confStoreColor(&newColor);
g_print(" done.\n");

/* Update the changed color into the application state. */
app->currentColor = newColor;
}

/*... Listing cut for brevity ...*/

/**
 * MODIFIED
 *
 * The color of the color button will be loaded from the application
 * preferences (or keep the default if preferences have no setting).
 */
static GtkWidget* buildToolbar(ApplicationState* app) {

    GtkWidget* toolbar = NULL;
    GtkWidget* tbOpen = NULL;
    GtkWidget* tbSave = NULL;
    GtkWidget* tbSep = NULL;
    GtkWidget* tbFind = NULL;
    GtkWidget* tbColorButton = NULL;
    GtkWidget* colorButton = NULL;

    g_assert(app != NULL);

    tbOpen = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
    tbSave = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);
    tbSep = gtk_separator_tool_item_new();
    tbFind = gtk_tool_button_new_from_stock(GTK_STOCK_FIND);

    tbColorButton = gtk_tool_item_new();
    colorButton = hildon_color_button_new();
    /* Copy the color from the color button into the application state.
       This is done to detect whether the color in preferences matches
       the default color or not (NEW). */

```

```

hildon_color_button_get_color(HILDON_COLOR_BUTTON(colorButton),
                              &app->currentColor);
/* Load preferences and change the color if necessary. */
g_print("buildToolbar: loading color pref.\n");
if (confLoadCurrentColor(&app->currentColor)) {
    g_print(" color not same as default one\n");
    hildon_color_button_set_color(HILDON_COLOR_BUTTON(colorButton),
                                  &app->currentColor);
} else {
    g_print(" loaded color same as default\n");
}
gtk_container_add(GTK_CONTAINER(tbColorButton), colorButton);

/*... Listing cut for brevity ...*/
}

```

Listing 7.70: hildon_helloworld-9.c

Since the graphical appearance of the program does not change (except that the ColorButton will display the correct initial color), a look will be taken at the stdout display of the program.

```

[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh ./hildon_helloworld-9
buildToolbar: loading color pref.
confLoadCurrentColor: invoked
hildon_helloworld-9[19840]: GLIB WARNING ** default -
confGetInt: key /apps/Maemo/hildon_hello/red not found
hildon_helloworld-9[19840]: GLIB WARNING ** default -
confGetInt: key /apps/Maemo/hildon_hello/green not found
hildon_helloworld-9[19840]: GLIB WARNING ** default -
confGetInt: key /apps/Maemo/hildon_hello/blue not found
loaded color same as default
main: calling gtk_main
cbActionMainToolbarToggle invoked
cbActionColorChanged invoked
color changed, storing into preferences..
confStoreColor: invoked
done.
main: returned from gtk_main and exiting with success

```

When running the program for the first time, warnings about the missing keys can be expected (since the values were not present in GConf).

Run the program again and exit:

```

[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh ./hildon_helloworld-9
buildToolbar: loading color pref.
confLoadCurrentColor: invoked
got red = 65535, after clamping = 65535
got green = 65535, after clamping = 65535
got blue = 0, after clamping = 0
color not same as default one
main: calling gtk_main
main: returned from gtk_main and exiting with success

```

The next step is to remove one key (red), and run the program again (this is to test and verify that the logic works):

```
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh gconftool-2 \
--unset /apps/Maemo/hildon_hello/red
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh gconftool-2 \
-R /apps/Maemo/hildon_hello
green = 65535
blue = 0
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh ./hildon_helloworld-9
buildToolBar: loading color pref.
confLoadCurrentColor: invoked
hildon_helloworld-9[19924]: GLIB WARNING ** default -
confGetInt: key /apps/Maemo/hildon_hello/red not found
got green = 65535, after clamping = 65535
got blue = 0, after clamping = 0
color not same as default one
main: calling gtk_main
main: returned from gtk_main and exiting with success
```

7.4.3 Asynchronous GConf

Listening to changes in GConf

Now it is time to see how to extend GConf to be more suitable in asynchronous work, and especially when implementing services.

When the configuration for the service are simple, and reacting to configuration changes in "realtime" is desired, it is advisable to use GConf. Also, people tend to use GConf when they are too lazy to write their own configuration file parsers (although there is a simple one in GLib), or too lazy to write the GUI part to change the settings. This example program will simulate the first case, and react to changes in a subset of GConf configuration name space when the changes happen.

The application will be interested in two string values; one to set the device to use for communication (connection), and the other to set the communication parameters for the device (connectionparams). Since this example will be concentrating on just the change notifications, the program logic is simplified by omitting the proper set-up code in the program. This means that it is necessary to set up some values to the GConf keys prior to running the program. For this, gconftool-2 will be used, and a target has been prepared in the **Makefile** just for this (see section *GNU Make and Makefiles* 4.2 in chapter *GNU Build System* if necessary):

```
# Define a variable for this so that the GConf root may be changed
gconf_root := /apps/Maemo/platdev_ex

# ... Listing cut for brevity ...

# 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)
```

Listing 7.71: gconf-listener/Makefile

The next step is to prepare the keyspace by running the primekeys target, and to verify that it succeeds by running the dumpkeys target:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make primekeys
gconftool-2 --set --type string \
    /apps/Maemo/platdev_ex/connection btcomm0
gconftool-2 --set --type string \
    /apps/Maemo/platdev_ex/connectionparams 9600,8,N,1
[sbox-DIABLO_X86: ~/gconf-listener] > make dumpkeys
Keys under /apps/Maemo/platdev_ex:
connectionparams = 9600,8,N,1
connection = btcomm0
```

Implementing Notifications on Changes in GConf

The first step here is to take care of the necessary header information. The GConf namespace settings have been all implemented using cpp macros, so that one can easily change the prefix of the name space if required later on.

```
#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"
```

Listing 7.72: gconf-listener/gconf-key-watch.c

The main starts innocently enough, by creating a GConf client object (that encapsulates the connection to the GConf daemon), and then displays the two values on output:

```
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.\n");

    /* 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);

```

Listing 7.73: gconf-listener/gconf-key-watch.c

The dispStringKey utility is rather simple as well, building on the GConf material that was covered in the previous section 7.4.2:

```

/**
 * Utility to retrieve a string key and display it.
 * (Just as a small refresher on the API.)
 */
static void 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. */
    }

    g_print(PROGNAME ": Value for key '%s' is set to '%s'\n",
            keyname, valueStr);

    /* 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);
}

```

Listing 7.74: gconf-listener/gconf-key-watch.c

Next, the GConf client is told to attach itself to a specific name space part that this example is going to operate with:

```

/**
 * 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 GCClient: %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");

```

Listing 7.75: gconf-listener/gconf-key-watch.c

Proceeding with the callback function registration, we have:

```

/* 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");

```

Listing 7.76: gconf-listener/gconf-key-watch.c

When dealing with regular desktop software, one could use multiple callback functions; one for each key to track. However, this would require implementing multiple callback functions, and this runs a risk of enlarging the size of the code. For this reason, the example code uses one callback function, which will internally multiplex between the two keys (by using strcmp):

```

/**
 * 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.\n");

    /* 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!\n");
        /* 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!\n");
}

/* 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]\n",
            strValue);
} else if (strcmp(keyname, SERVICE_KEY_CONNECTIONPARAMS) == 0) {
    g_print(PROGNAME ": Connection params setting changed: [%s]\n",
            strValue);
} else {
    g_print(PROGNAME ":Unknown key: %s (value: [%s])\n", keyname,
            strValue);
}

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

g_print(PROGNAME ": keyChangeCallback done.\n");
}

```

Listing 7.77: gconf-listener/gconf-key-watch.c

The complications in the above code rise from the fact that GConf communicates values using a GValue structure, which can carry values of any simple type. Since GConf (or the user for that matter) cannot be completely trusted to return the correct type for the value, it is necessary to be extra careful, and not assume that it will always be a string. GConf also supports "default" values, which are communicated to the application using NULLs, so it is also necessary to guard against that. Especially since the application does not provide a schema for the configuration space that would contain the default values.

The next step is to build and test the program. The program will be started on the background, so that gconftool-2 can be used to see how the program reacts to changing parameters:


```

[sbox-DIABLO_X86: ~/gconf-listener] > make
cc -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -I/usr/include/gconf/2 \
-I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include -Wall -g \
-DPROGRAMNAME=\"gconf-key-watch\" gconf-key-watch.c -o gconf-key-watch \
-lgconf-2 -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/gconf-listener] > run-standalone.sh ./gconf-key-watch &
[2] 21385
gconf-key-watch:main Starting.
gconf-key-watch:main GType and GConfClient initialized.
gconf-key-watch: Value for key '/apps/Maemo/platdev_ex/connection'
is set to 'btcomm0'
gconf-key-watch: Value for key '/apps/Maemo/platdev_ex/connectionparams'
is set to '9600,8,N,1'
gconf-key-watch:main Added /apps/Maemo/platdev_ex.
gconf-key-watch:main CB registered & starting main loop
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type string \
/apps/Maemo/platdev_ex/connection ttyS0
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection type setting changed: [ttyS0]
gconf-key-watch: keyChangeCallback done.
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type string \
/apps/Maemo/platdev_ex/connectionparams ''
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: []
gconf-key-watch: keyChangeCallback done.

```

The latter change is somewhat problematic (which the code needs to deal with as well). It is necessary to decide how to react to values that are of the correct type, but with senseless values. GConf in itself does not provide syntax checking for the values, or any semantic checking support. It is recommended that configuration changes will only be reacted to when they pass some internal (to the application) logic that will check their validity, both at syntax level and also at semantic level.

One option would also be resetting the value back to a valid value, whenever the program detects an invalid value set attempt. This will lead to a lot of problems, if the value is set programmatically from another program that will obey the same rule, so it is not advisable. Quitting the program on invalid values is also an option that should not be used, since the restricted environment does not provide many ways to inform the user that the program has quit.

An additional possible problem is having multiple keys that are all "related" to a single setting or action. It is necessary to decide, how to deal with changes across multiple GConf keys that are related, yet changed separately. The two key example code demonstrates the inherent problem: should the server re-initialize the (theoretic) connection, when the connection key is changed, or when the connectionparams key is changed? If the connection is re-initialized when either of the keys is changed, then the connection will be re-initialized twice when both are changed "simultaneously" (user presses "Apply" on a settings dialog, or gconftool-2 is run and sets both keys). It is easy to see how this might be an even larger problem, if instead of two keys, there were five per connection. GConf and the GConfClient GObject wrapper that has been used here do not support "configuration set transactions", which would allow setting and processing multiple related keys using an atomic model. The example program ignores this issue completely.

The next step is to test how the program (which is still running) reacts to other problematic situations:

```
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type int \
/apps/Maemo/platdev_ex/connectionparams 5
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: [5]
gconf-key-watch: keyChangeCallback done.
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type boolean \
/apps/Maemo/platdev_ex/connectionparams true
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: [true]
gconf-key-watch: keyChangeCallback done.
```

The next example removes the configuration keys, while the program is still running:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make clearkeys
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch[21403]: GLIB ERROR **: default -
file gconf-key-watch.c: line 129 (keyChangeCallback):
assertion failed: (value != NULL)
aborting...
/usr/bin/run-standalone.sh: line 11: 21403 Aborted (core dumped) "$@"
[1]+ Exit 134 run-standalone.sh ./gconf-key-watch
```

Since the code (in the callback function) contains an assert that checks for non-NULL values, it will abort when the key is removed, and that causes the value to go to NULL. So the abortion in the above case is expected.

7.5 Alarm Framework

The maemo alarm framework provides an easy way to manage timed events in the device. It is powerful, and not restricted only to wake up alarms. The framework provides many other features, including:

- Setting multiple alarm events
- Configuring the number of recurrences and the time between each one
- Choosing whether the alarm dialog should be shown
- Setting the title and the message to be shown in the alarm dialog
- Selecting a custom icon file to be shown in the alarm dialog
- Selecting a custom sound file to be played while the alarm dialog is shown
- Choosing to boot the device if it is turned off
- Choosing to run the alarm only if the device is connected
- Choosing to run the alarm in the system start up if it is missed
- Choosing to postpone the alarm if it is missed
- Executing a given file (e.g. a program or a script)
- Sending a message to a D-Bus message bus
- Querying for existing alarm events in a given period of time
- Deleting an existing alarm event
- Configuring the snooze time for the event

- Configuring the default snooze time for all alarm events

This section shows how to use the alarm interface, describes all its functions and gives examples of use cases.

Timed event functionality is provided by the `alarmd` daemon. It makes it possible for applications to get D-Bus messages or `exec` calls at certain times.

Applications register an event happening at a certain time. The `alarmd` daemon will then call the application when the event is due, so the application does not need to be running during the waiting period. This is useful for applications that need to update their status at regular intervals.

A single event may be set to repeat itself over and over again, at regular intervals. The `alarmd` daemon will remember these events over system shutdowns too.

The `alarmd` daemon can also be asked to start the system up for an event, even when powered off. These events may also be set to run in a hidden power-on mode, where the display stays off, but the device is on. The `alarmd` daemon also supports the use of `osso-systemui-alarm` to show an alarm dialog for the event, which can be followed by a power up device confirmation dialog, if the device is in the hidden power-up mode.

Communication with the `alarmd` daemon is performed through D-Bus. It listens to both system and session buses. The easiest way is to use the C client API library, which offers straightforward API to modify the queue.

The alarm queue used by the `alarmd` daemon is stored in `/var/lib/alarmd/alarm_queue.xml` file.

7.5.1 Alarm Events

Alarm events and the functions to manage them are defined in `alarm_event.h` header file located in `/usr/include/alarmd`. It is a part of the `libalarm-dev` debian package.

Each alarm event is identified by a unique key, also known as a "cookie". These identifiers are obtained, when a new event is added to the queue. They are used whenever needed, and to retrieve information about a specific event, or delete it. An alarm event is identified by the `alarm_event_t` structure, described as follows:

```
/**
 * cookie_t:
 *
 * Unique identifier type for the alarm events.
 */
typedef long cookie_t;

/**
 * alarm_event_t:
 *
 * Describes an alarm event.
 */
typedef struct {
    time_t alarm_time;
    uint32_t recurrence;
    int32_t recurrence_count;
    uint32_t snooze;
    char *title;
```

```
char *message;  
char *sound;  
char *icon;  
char *dbus_interface;  
char *dbus_service;  
char *dbus_path;  
char *dbus_name;  
char *exec_name;  
int32_t flags;  
uint32_t snoozed;  
} alarm_event_t;
```

Listing 7.78: alarmd/alarm_event.h

Description of structure `alarm_event_t` fields

Name	Type	Description
<code>alarm_time</code>	<code>time_t</code>	The number of seconds elapsed since 00:00:00 on January 1, 1970 when the event should be launched.
<code>recurrence</code>	<code>uint32_t</code>	Interval in minutes, over which the event should be repeated. 0 if the event should only happen once.
<code>recurrence_count</code>	<code>int32_t</code>	Number of times the event should repeat. -1 for infinite.
<code>snooze</code>	<code>uint32_t</code>	Number of minutes the event is postponed, when Snooze button is pressed in alarm dialog. 0 for default setting.
<code>title</code>	<code>char *</code>	Title shown in the alarm dialog. May contain gettext identifiers, as described in Localized Strings section.
<code>message</code>	<code>char *</code>	Message shown in the alarm dialog. May contain gettext identifiers, as described in Localized Strings section.
<code>sound</code>	<code>char *</code>	Path to the sound file that should be played when the alarm dialog is shown.
<code>icon</code>	<code>char *</code>	Path to the icon file or GtkIconTheme icon name to show in the alarm dialog.
<code>dbus_interface</code>	<code>char *</code>	Interface for D-Bus action taken when event is due.
<code>dbus_service</code>	<code>char *</code>	Service to be called on D-Bus action. If other than NULL, the D-Bus action will be a method call.
<code>dbus_path</code>	<code>char *</code>	Path for D-Bus action. If defined, the action on event will be a D-Bus action.
<code>dbus_name</code>	<code>char *</code>	Name for D-Bus action. Either the D-Bus method name or signal name.
<code>exec_name</code>	<code>char *</code>	Command to run as events action. Only used, if the <code>dbus_path</code> field is not set. The format is as accepted by glib's <code>g_shell_parse_argv</code> .
<code>flags</code>	<code>int32_t</code>	Options for the event; the value consist of members of enumeration <code>alarmeventflags</code> , ORed together bitwise.
<code>snoozed</code>	<code>uint32_t</code>	The number of times the event has been snoozed, only applies to getting an event. When setting one, this has no effect.

The `flags` field in the `alarm_event_t` structure above holds the options of an alarm event. This field is a bitwise of one or more items described in `alarmeventflags` enumeration:

```
typedef enum {
    ALARM_EVENT_NO_DIALOG = 1 << 0,
    ALARM_EVENT_NO_SNOOZE = 1 << 1,
    ALARM_EVENT_SYSTEM = 1 << 2,
```

```

ALARM_EVENT_BOOT = 1 << 3,
ALARM_EVENT_ACTDEAD = 1 << 4,
ALARM_EVENT_SHOW_ICON = 1 << 5,
ALARM_EVENT_RUN_DELAYED = 1 << 6,
ALARM_EVENT_CONNECTED = 1 << 7,
ALARM_EVENT_ACTIVATION = 1 << 8,
ALARM_EVENT_POSTPONE_DELAYED = 1 << 9,
ALARM_EVENT_BACK_RESCHEDULE = 1 << 10,
} alarmeventflags;

```

Listing 7.79: alarmd/alarm_event.h

Description of alarmeventflags values

Value	Description
ALARM_EVENT_NO_DIALOG	When the alarm is launched, it will not show a dialog, but instead perform the action immediately.
ALARM_EVENT_NO_SNOOZE	When the alarm dialog is shown, the Snooze button will be disabled.
ALARM_EVENT_SYSTEM	The D-Bus call should be performed over system bus, instead of session bus.
ALARM_EVENT_BOOT	The event should start the device, if turned off.
ALARM_EVENT_ACTDEAD	If the device is turned off, it will be started with display off, and turned off again after the action is performed.
ALARM_EVENT_SHOW_ICON	While the event is on queue, an icon should be shown in the status bar.
ALARM_EVENT_RUN_DELAYED	Defines whether the event should be run in case it has been missed or jumped over.
ALARM_EVENT_CONNECTED	Defines that the event should only be run, when there is an active connection. If no connection is available, the event will be launched when a connection becomes available.
ALARM_EVENT_ACTIVATION	Defines whether the D-Bus action of the event should use D-Bus activation, i.e. whether the service should be started, if not already running.
ALARM_EVENT_POSTPONE_DELAYED	If the event is missed by more than a day, it will be moved forward to the next day. If less than a day, it will be launched.
ALARM_EVENT_BACK_RESCHEDULE	If time is changed backwards, the event is moved backwards also. Only applies to recurring alarms.

7.5.2 Managing Alarm Events

Adding Alarm Event to Queue

To add a new alarm event to the queue, a new instance of `alarm_error_t` structure is needed, and the necessary fields have to be filled in. The only mandatory one is `alarm_time`. Check `time(2)` and `ctime(3)` manpages for more instructions on how to manipulate a `time_t` type.

With the fields of the structure set, all that is needed is a call to the `alarm_event_add` function, passing a pointer to the structure as parameter.

```
/**
 * alarm_event_add:
 *
 * Adds an event to the alarm queue.
 */
cookie_t alarm_event_add(alarm_event_t *event);
```

Listing 7.80: `alarmd/alarm_event.h`

On success, this function returns the cookie for the alarm event, otherwise it returns 0. For example, the following program sets an alarm event to 2 seconds from now:

```
alarm_event_t event;
time_t current_time;
struct tm *alarm_time;

/* Reset the event-structure to zero */
memset(&event, 0, sizeof(event));

/* Get current time and add two seconds to it,
 * meaning that alarm should be sent at
 * this time */
time(&current_time);
alarm_time = localtime(&current_time);
alarm_time->tm_sec += 2;
/* Set the alarm time */
event.alarm_time = mktime(alarm_time);

/* Set interface, service name, path and method
 * name of the HelloAlarm D-BUS interface */
event.dbus_interface = "org.maemo.alarm_example";
event.dbus_service = "org.maemo.alarm_example";
event.dbus_path = "/org/maemo/alarm_example";
event.dbus_name = "hello_alarm";

/* The application presents its own dialog,
 * so the system provided isn't needed */
event.flags = ALARM_EVENT_NO_DIALOG;

/* Add the alarm to alarmd and
 * verify that it was correctly created */
if(!alarm_event_add(&event))
{
    hildon_banner_show_information(button, "gtk-dialog-error",
        "Couldn't set alarm");
    g_warning("Couldn't set alarm. Error code %d\n", alarmd_get_error());
    return;
}
```

```
hildon_banner_show_information(button, NULL,
    "Alarm set");
```

Listing 7.81: maemo-examples/example_alarm.c

Fetching Details of Alarm Event

Having set an alarm event, it is not possible to edit its details anymore, but it is possible to fetch the alarm details using the `alarm_event_get` function, passing the event cookie as parameter.

```
/**
 * alarm_event_get:
 *
 * Finds an alarm with given identifier
 * and returns alarm_event struct describing it.
 */
alarm_event_t *alarm_event_get(cookie_t event_cookie);
```

Listing 7.82: alarmd/alarm_event.h

This function returns a newly allocated `alarm_event_t` structure. After manipulating the structure fields, it is possible to set another alarm event with the same details. It is necessary to free the allocated structure obtained with the `alarm_event_get` function. For this task, it is necessary a call to the `alarm_event_free` function, receiving the alarm event structure as parameter:

```
/**
 * alarm_event_free:
 *
 * Frees given alarm_event struct.
 */
void alarm_event_free(alarm_event_t *event);
```

Listing 7.83: alarmd/alarm_event.h

Deleting Alarm Event

Finally, to delete an alarm event, the `alarm_event_del` function needs to be called, passing the event cookie as a parameter.

```
/**
 * alarm_event_del:
 *
 * Deletes alarm from the alarm queue.
 */
int alarm_event_del(cookie_t event_cookie);
```

Listing 7.84: alarmd/alarm_event.h

On success, this function returns 1, otherwise it returns 0.

7.5.3 Checking for Errors

Whenever an error occurs during a call to any of the functions, an error code is set, and it can be obtained with the `alarmd_get_error` function. It has the following signature:


```

/**
 * alarmd_get_error:
 *
 * Gets the error code for previous action.
 */
alarm_error_t alarmd_get_error(void);

```

Listing 7.85: alarmd/alarm_event.h

The possible values returned by this function are defined by `alarm_error_t` enumeration, described below:

```

typedef enum {
    ALARM_SUCCESS,
    ALARM_ERROR_DBUS,
    ALARM_ERROR_CONNECTION,
    ALARM_ERROR_INTERNAL,
    ALARM_ERROR_MEMORY,
    ALARM_ERROR_ARGUMENT,
    ALARM_ERROR_NOT_RUNNING
} alarm_error_t;

```

Listing 7.86: alarmd/alarm_event.h

Description of `alarm_error_t` values

Value	Description
<code>ALARM_SUCCESS</code>	No error has occurred.
<code>ALARM_ERROR_DBUS</code>	An error with D-Bus occurred, probably could not get a D-Bus connection.
<code>ALARM_ERROR_CONNECTION</code>	Could not contact alarmd via D-Bus.
<code>ALARM_ERROR_INTERNAL</code>	Some alarmd or libalarm internal error, possibly a version mismatch.
<code>ALARM_ERROR_MEMORY</code>	Memory allocation failed.
<code>ALARM_ERROR_ARGUMENT</code>	An argument given by caller was invalid.
<code>ALARM_ERROR_NOT_RUNNING</code>	Alarmd was not running.

7.5.4 Localized Strings

If the message in the alarm dialog is needed to be available in the language that is being used in the device, specially formatted strings can be used. The translations are fetched with `dgettext`. To get the message translated, it is necessary to specify the package from which to fetch the translation and the message id. The format is as follows:

```
{message domain,message id}
```

If the translation contains "%s", the replacements for these can be given too:

```
{message domain,message id,replacement1,replacement2...}
```

The replacement may be in form ..., but cannot contain anything else; the translated part can be embedded in the main string. Positional parameters or any other format options than %s are not supported.

Escaping Strings

If the message comes from user, the `alarm_escape_string` function can be used to escape all such formatting from the string. `alarm_escape_string` escapes the string to be used as title or message of the alarm dialog, so that it is not considered as a localization string. All occurrences of `"`, `'` characters should be escaped with a backslash, and all backslashes should be duplicated. The function receives a string as parameter, and returns a newly allocated string that should be freed with `free`. This function has the following syntax:

```
/**
 * alarm_escape_string:
 *
 * Escapes a string to be used as alarm dialog message or title.
 */
char *alarm_escape_string(const char *string);
```

Listing 7.87: `alarmd/alarm_event.h`

Unescaping Strings

To unescape a string escaped with `alarm_escape_string`, or any other escaped string that uses backslashes to escape characters, the `alarm_unescape_string` function must be called. It receives a string as parameter, and returns a newly allocated string that should be freed with `free`. The syntax is as follows:

```
/**
 * alarm_unescape_string:
 *
 * Unescapes a string escaped with alarm_escape_string.
 */
char *alarm_unescape_string(const char *string);
```

Listing 7.88: `alarmd/alarm_event.h`

Alternatively, the `alarm_unescape_string_noalloc` can be used to unescape a string. The difference between the functions is that `alarm_unescape_string` allocates memory for the returned string, whereas `alarm_unescape_string_noalloc` modifies the passed string. This is possible, since the unescaped string cannot be longer than the original string.

```
/**
 * alarm_unescape_string_noalloc:
 *
 * Unescapes a string escaped with alarm_escape_string.
 */
char *alarm_unescape_string_noalloc(char *string);
```

Listing 7.89: `alarmd/alarm_event.h`

The unescape functions have checking for strings ending with double backslashes, and the single backslash is put into the resulting string. If `NULL` is passed, `NULL` is returned by all functions.

7.6 Usage of Back-up Application

The back-up application saves and restores user data stored in /MyDocs (by default) and setting directories/files /etc/osso-af-init/gconf-dir (a link to GConf database /var/lib/gconf), /etc/osso-af-init/locale, and /etc/bluetooth/name. It can be configured to back up other locations and files as well, by custom configuration files.

The back-up application must not be disrupted by other applications writing or reading during a back-up or restore operation.

For restore process, back-up therefore will, if the user approves, ask the application killer to close all applications, and then wait until it has been done.

For backing up, the backup_start and backup_finish D-BUS signals will be emitted on the session bus, indicating to applications that they should not write to disk.

D-BUS description and methods of back-up Application

```
Service      com.nokia.backup
Interfaces   com.nokia.backup
Object Paths /com/nokia/backup

Method: cancel

Name          cancel
Parameters    none
Returns       Empty reply
Description    Cancels any ongoing back-up or restore operation

Method: activate

Name          activate
Parameters    none
Returns       Empty reply
Description    Used to activate the application with auto-activation
```

7.6.1 Custom Back-up Locations

For other data not normally backed up, the so-called locations configuration is used. It is important that the locations configuration paths **MUST NOT** overlap with the documents path.

The locations configuration lets applications install a configuration file with a list of files and directories that should be included in the back-up. The files should be installed into /etc/osso-backup/applications, named <application>.conf and consist of simple XML format. For the application, the example_libosso.conf looks like the following:

```
<backup-configuration>
  <locations>
    <location type="file"
              category="settings" auto="true">/etc/example.ini</location>
    <location type="dir"
              category="documents">/home/user/foo</location>
    <exclusion type="file"
              category="settings">/home/user/bigfile</exclusion>
    <exclusion type="file"
              category="settings">/tmp/*.jpg</exclusion>
  </locations>
```

```
</backup-configuration>
```

With the `<location>` tag, different locations can be given to be backed up. N.B. The path must be absolute. `<exclusion>` tag can be used, if some files in certain directories are not wanted to be backed up, for example, in the case of some setting file changing from one software version to another. This way, the new setting file of updated software will not be destroyed, if the back-up is restored. Wild cards '?' and '*' are also supported.

Both tags have TYPE and CATEGORY arguments and `<location>` tag additional AUTO argument:

TYPE - Its value can be "file" or "dir" for a file or directory. This argument must be provided.

CATEGORY - It is used for handling selective back-up and restore. It may be omitted, in which case the location will only be backed up when backing up or restoring everything. The value can be:

- emails
- documents
- media
- contacts
- bookmarks
- settings
- applications

AUTO - Its value can be true or false. It is used to tell the back-up application not to request a user response in case of a conflict, but automatically replace the file. N.B. The auto argument is only used for files, not for directories. It may be omitted, in which case it defaults to false.

7.6.2 After Restore Run Scripts

The back-up application makes it possible to execute scripts after a restore operation. There are two kinds of scripts. First, there can be scripts that are executed after every restore operation. Then there are also scripts that are executed only, if the restore is made from a back-up created in an earlier product.

For the scripts that are used to transform data between the device software versions, the location for applications to install the scripts is `/etc/osso-backup/restore.d/<dir>`, where `<dir>` is a subdirectory for each transition between two different consecutive version of the platform. For transforming between IT-2006 and IT-2007 versions, the directory is `/etc/osso-backup/restore.d/it2007/`. For scripts that are executed after every restore, the location is `/etc/osso-backup/restore.d/always`.

The files installed here should have the executable bit set. Any files ending with "~" or ".bak" are ignored, just like directories or files starting with a dot (".").

Each script will be executed with a command line argument that is the path of a file containing the list of all files that have been restored, per category. The format of this file is:

```
[CATEGORY]
/path/to/file1
/path/to/file2
...
[CATEGORY]
...
```

CATEGORY is one of the OSSO categories (emails, documents, media, contacts, bookmarks, settings or applications). This makes it possible for the scripts to check, which files they need to upgrade, if any at all. The format is chosen to be easy to parse with a simple script or program.

The scripts will be executed after a successful restoration, or after a restoration has been canceled. In both cases, the scripts will only be executed if any files were actually restored. Scripts should clean up after transforming, so that old files are not left behind. The script or program should use the common convention and return zero on success, and non-zero in case of failure. Application developers should try and make their programs execute and finish quickly.

7.7 Using Maemo Address Book API

The maemo platform features a centralized storage for address book data, which is implemented using the Evolution Data Server (EDS). This allows different applications to effortlessly share contacts in a standardized form, so that the user does not have to specify them in every program separately.

This section describes how to use address book APIs provided by the libosso-abook component in the maemo platform. Libosso-abook is a collection of powerful widgets and objects, allowing the creation of programs that integrate fully into the address book and communication frameworks of the maemo platform. More in-depth information can be found at the Maemo API Reference[57], and the Evolution project's web site[16].

This section will look at the steps required in creating a simple application showing all of the user's contacts, allowing some interaction with them, and being able to dump the raw VCard data to the terminal. This section shows how to open and manipulate an address book, how to create and populate the models and views required to display the contacts, and how to use some of them.

7.7.1 Using Library

Include Files

There are only two include files needed to deal with the functions in this tutorial. All the relevant data for the Evolution Data Server functions is included with statement

```
#include <libebook/e-book.h>
```

and all the data for the libosso-abook functions is included with statement

```
#include <libosso-abook/osso-abook.h>
```

Notes on how to compile a program using libosso-abook can be found at the end of this how-to, including a description on how to add a check for libosso-abook to the configuration file.

Application Initialization

The start of the program is similar to other maemo applications. The first step that is needed is the initialization of libosso to notify that the application has started.

```
osso_context_t *osso_context;

/* Initialize osso */
osso_context = osso_initialize ("com.nokia.osso-abook-example",
                               VERSION, TRUE, NULL);

if (osso_context == NULL) {
    g_critical ("Error initializing osso");
    return 1;
}
```

When initializing libosso-abook, this `osso_context` variable needs to be passed to `osso_abook_init`. It is necessary for supplying the help dialogs and for integrating certain features into the whole of the maemo platform. `osso_abook_init` also initializes GTK+, gnome-vfs and Galago, so the application does not have to initialize them itself.

This is the prototype for `osso_abook_init`:

```
gboolean osso_abook_init (int *argc, char ***argv, osso_context_t *
                          osso_context);
```

Listing 7.90: libosso-abook/osso-abook-init.h

`osso_abook_init` also takes the command line arguments passed to the application, and uses them to initialize GTK+. In certain cases, such as plug-ins and libraries, however, it is not possible to pass these arguments. For such cases, libosso-abook also provides `osso_abook_init_with_name`, where the name parameter is used to name the Galago connection. The name should not contain any spaces or special characters:

```
gboolean osso_abook_init_with_name (const char *name, osso_context_t *
                                    osso_context);
```

Listing 7.91: libosso-abook/osso-abook-init.h

When the example application can pass the command line arguments, the first version can be used.

```
/* Init abook */
if (!osso_abook_init (&argc, &argv, osso_context)) {
    g_critical ("Error initializing libosso-abook");

    osso_deinitialize (osso_context);
    return 1;
}
```

The rest of the main function just creates the application and enters the main loop. After the main loop quits, all the resources are tidied up, libosso is deinitialized and the program exits normally.

```

/* Make our UI */
app = app_create ();

/* Run */
gtk_main ();

/* Shutdown */
app_destroy (app);

osso_deinitialize (osso_context);
return 0;

```

7.7.2 Accessing Evolution Data Server (EDS)

Loading Contacts from EDS

In EDS, contacts are stored in a database called a book. There are three different ways to retrieve contacts: as a single contact, as a static list of contacts meeting unchanging criteria, or as a live view on the book that changes as contacts are added or removed. The last method is the most powerful, and it is also the way that most applications will use.

Opening Book

Before any operation can be performed on the contacts, the book containing them needs to be created and then opened. There are a number of different calls that create a book. The call to create a book depends on which kind of a book is required.

```

EBook *e_book_new (ESource *source, GError **error);
EBook *e_book_new_from_uri (const char *uri, GError **error);
EBook *e_book_new_system_addressbook (GError **error);
EBook *e_book_new_default_addressbook (GError **error);

```

Listing 7.92: libebook/e-book.h

The calls create different kinds of books. The first two calls can create any kind of a book, the latter two can only create pre-defined books. In nearly all circumstances, the call to use is `e_book_new_system_addressbook`.

Once an `EBook` object is created, the book must be opened. Books can either be opened synchronously, meaning that the application will pause until the book is opened; or asynchronously, meaning that control will return to the application immediately, but the book will not be ready for use until the `open_response` callback is called.

The following are the two prototypes for opening books:

```

gboolean e_book_open (EBook *book, gboolean only_if_exists, GError **
    error);
guint e_book_async_open (EBook *book, gboolean only_if_exists,
    EBookCallback open_response, gpointer closure);

```

Listing 7.93: libebook/e-book.h

There is a parameter called `only_if_exists`. This parameter controls whether a database is created on disk. If set to `TRUE` and the application attempts

to open a book that does not already exist, an error will be returned. If set to FALSE and a non-existing book is opened, the database structure will be created on the disk with no contacts in it. On a new system, there will not be any address books, so TRUE should never be passed, as then the call would fail.

As mentioned above, when using the asynchronous version, control will return to the application immediately. When the book is opened, the open_response callback will be called, and closure will be passed to it. The EBookCallback is defined as:

```
void (*EBookCallback) (EBook *book, EBookStatus status, gpointer
closure);
```

Listing 7.94: libebook/e-book.h

When the open_response callback and the status do not report any errors, the book is open and ready for use.

The only difficulty in opening a book is that the main loop needs to be running for it to work. This means that if the main loop is not running, and the book needs to be opened automatically, then the e_book_new call needs to be made in an idle handler. The example program does this at the end of the app_create function, passing in the AppData structure.

```
AppData *app;

/* Install an idle handler to open the address book
when in the main loop */
g_idle_add (open_addressbook, app);
```

This means that once the main loop is running, open_addressbook will be called. open_addressbook is the function that creates and opens the book using the asynchronous functions discussed above.

```
static gboolean
open_addressbook (gpointer user_data)
{
    AppData *app;
    GError *error;

    app = user_data;

    error = NULL;
    app->book = e_book_new_system_addressbook (&error);
    if (!app->book) {
        g_warning ("Error opening system address book: %s",
            error->message);
        g_error_free (error);
        return FALSE;
    }

    e_book_async_open (app->book, FALSE, book_open_cb, app);

    /* Return FALSE so this callback is only run once */
    return FALSE;
}
```

This function attempts to open the system address book, and creates it, if it does not exist already. Error checking is performed by passing a GError into a

function, and if the function returns NULL or FALSE (depending on the return type of the function), an error has occurred and will be reported in the GError.

Retrieving Contacts

Once the book has been successfully opened, either directly after a call to `e_book_open`, or in the `open_response` callback of `e_book_async_open`, the contacts can be manipulated.

Even though the book view is the way used by most applications to get the contacts, it is possible to get single contacts and perform synchronous queries. For getting single contacts, the contact id is required. This means that all the contacts must have been retrieved before using one of the other two methods.

This operation can be performed either synchronously or asynchronously, as required, by using `e_book_get_contact` or `e_book_async_get_contact`:

```
gboolean e_book_get_contact (EBook *book, const char *id,
                             EContact **contact, GError **error);
guint e_book_async_get_contact (EBook *book,
                                const char *id,
                                EBookContactCallback cb,
                                gpointer closure);
```

Listing 7.95: libebook/e-book.h

The `EBookContactCallback` is defined as

```
void (*EBookContactCallback) (EBook *book,
                               EBookStatus status,
                               EContact *contact,
                               gpointer closure);
```

Listing 7.96: libebook/e-book.h

Queries

Queries are used to define the required contacts both when retrieving multiple contacts and when the book view is used. These queries are built from various basic queries, combined using the Boolean operations AND, OR and NOT. The basic queries are created with the following four functions:

```
EBookQuery *e_book_query_field_exists (EContactField field);
EBookQuery *e_book_query_vcard_field_exists (const char *field);
EBookQuery *e_book_query_field_test (EContactField field,
                                     EBookQueryTest test, const char *
                                     value);
EBookQuery *e_book_query_any_field_contains (const char *value);
```

Listing 7.97: libebook/e-book-query.h

Of these, `e_book_query_any_field_contains` is the most generic and, if passed an empty string (`""`), acts as the method for getting all the contacts in the book. The other three functions check for specific fields. Functions `e_book_query_field_exists` and `e_book_query_vcard_field_exists` both check for the existence of a specific field. The difference between them is the method of checking: `e_book_query_field_exists` uses the `EContactField` enumeration that contains many common field types, such as `E_CONTACT_FULL_NAME`, `E_CONTACT_HOMEPAGE_URL` and

E_CONTACT_EMAIL_1 (the full list can be found in the include file libebook/e-contact.h).

The final function, `e_book_query_field_test`, is the most powerful, and uses the `EBookQueryTest` to define the type of test to perform on the field. There are four possible tests: `IS`, `CONTAINS`, `BEGINS_WITH` and `ENDS_WITH`. These are defined in the file `libebook/e-book-query.h`. The term the query should search for is given in the third parameter. For the call:

```
query = e_book_query_field_test (E_CONTACT_FULL_NAME ,
                                E_BOOK_QUERY_BEGINS_WITH, "ross");
```

the resulting query will match contacts with the full name field containing "Ross Smith" and "Ross Bloggs" (queries are case insensitive), but will not match "Jonathan Ross" or "Diana Ross".

These basic queries can be combined in any desired way using the following functions that implement the Boolean operations:

```
EBookQuery *e_book_query_and (int nqs, EBookQuery **qs, gboolean unref)
;
EBookQuery *e_book_query_or (int nqs, EBookQuery **qs, gboolean unref);
EBookQuery *e_book_query_not (int nqs, EBookQuery **qs, gboolean unref)
;
```

Listing 7.98: libebook/e-book-query.h

Each of these three functions takes in a number of queries, combines them and returns a new query that represents the correct operation applied to all the original queries.

The following code example will combine the three queries, so that the contacts returned are contacts who have a blog, and whose name begins with either Matthew or Ross.

```
EBookQuery *name_query[2], *blog_query[2];
EBookQuery *query;

name_query[0] = e_book_query_field_test (E_CONTACT_FULL_NAME,
                                         E_BOOK_QUERY_BEGINS_WITH, "
                                         ross");
name_query[1] = e_book_query_field_test (E_CONTACT_FULL_NAME,
                                         E_BOOK_QUERY_BEGINS_WITH, "
                                         matthew");

blog_query[0] = e_book_query_field_exists (E_CONTACT_BLOG_URL);

/* Combine the name queries with an OR operation
 * to create the second bit of the blog query. */
blog_query[1] = e_book_query_or (2, name_query, TRUE);

query = e_book_query_and (2, blog_query, TRUE);

/* carry out query on book */
e_book_query_unref (query);
```

When combining queries, the Boolean operation functions are able to take the ownership of the queries that have been passed in, meaning that the application only needs to unref the result.

Getting Static List of Contacts

When using this query to get a list of contacts, the contacts in the list will never change. As with most functions that operate on the book, there is a synchronous and an asynchronous way to get a static list of contacts.

```
gboolean e_book_get_contacts (EBook *book, EBookQuery *query,
                             GList **contacts, GError **error);
guint e_book_async_get_contacts (EBook *book, EBookQuery *query,
                                EBookListCallback cb, gpointer closure
                                );
```

Listing 7.99: libebook/e-book.h

The EBookListCallback is defined as

```
void (*EBookListCallback) (EBook *book, EBookStatus status, GList *list
, gpointer closure);
```

Listing 7.100: libebook/e-book.h

Once the application has finished and the contacts have been returned, they should be unrefed with `g_object_unref` and call `g_list_free` on the list to free all the memory used.

Getting Dynamic Set of Contacts

Even though under certain circumstances, getting single contacts and static lists of contacts is useful, most applications need to use the book view method to get contacts. A book view is a dynamic representation of a query performed on a book.

The book view is an object that has signals to indicate when contacts have been modified, removed or added. If the application uses the libosso-abook widgets, this will be handled automatically, and any changes in the book view will be correctly updated in the widgets. However, if the application is performing actions with book views that are not covered by the supplied widgets, then it is necessary to listen for these signals, and deal with them accordingly.

With a query created, the application can request a book view from the open book. Again, as with opening a book, there are two methods for getting a book view from a book: synchronous and asynchronous.

```
gboolean e_book_get_book_view (EBook *book, EBookQuery *query,
                              GList *requested_fields,
                              int max_results,
                              EBookView **book_view, GError **error);
guint e_book_async_get_book_view (EBook *book, EBookQuery *query,
                                  GList *requested_fields,
                                  int max_results,
                                  EBookBookViewCallback cb,
                                  gpointer closure);
```

Listing 7.101: libebook/e-book.h

These functions are more powerful than needed for this example. The only requirement is the query `"-1"` passed in for the `max_results`, meaning that all the results will be returned, and `NULL` can be passed in for `requested_fields`. These two parameters are not as much explicit controls as they are suggestions.

The default back-end will ignore them and just return all the results, with all the fields available. In the LDAP back-end, there is support for them.

The book view can be used once it has been returned, either from the `e_book_get_book_view` call, or from the callback by `e_book_async_get_book_view`.

```
static void book_open_cb (EBook *book, EBookStatus status, gpointer
    user_data)
{
    AppData *app;
    EBookQuery *query;

    app = user_data;
    if (status != E_BOOK_ERROR_OK) {
        g_warning ("Error opening book");
        return;
    }

    query = e_book_query_any_field_contains ("");
    e_book_async_get_book_view (app->book, query, NULL, -1,
        get_book_view_cb, app);
    e_book_query_unref (query);
}
```

`EBookBookViewCallback` has the following prototype:

```
void (*EBookBookViewCallback) (EBook *book, EBookStatus status,
    EBookView *book_view, gpointer closure);
```

Listing 7.102: libebook/e-book.h

The status parameter returns information on whether the call to `_book_async_get_book_view` was successful, and the new book view is in the `book_view` parameter.

Finally, the book view needs to be told to start sending signals to the application when contacts are modified. The function for this is `e_book_view_start`. There is also a similar function for stopping the book view: `e_book_view_stop`.

```
void e_book_view_start (EBookView *book_view);
void e_book_view_stop (EBookView *book_view);
```

Listing 7.103: libebook/e-book-view.h

The application should connect signals to the book view before calling `e_book_view_start`, in case the reporting of some contacts is missed.

7.7.3 Creating User Interface

The tutorial application simply displays all the contacts in the system address book as a list. There are two parts in creating this list: the list widget, which is derived from the `GtkTreeView` widget, and the model that drives the aforementioned widget and stores all the data.

Creating Contact Model

Libosso-abook provides a model derived from the `GtkTreeModel` object to store the contact data, and handles all that is required to deal with contacts being added and changed. The model is called `OsoABookContactModel`, and it is created with `osso_abook_contact_model_new()`, which takes no arguments.

```
app->contact_model = osso_abook_contact_model_new ();
```

OssoABookContactModel populates itself from an EBookView. All that is required to populate it, is to set the book view if the application has one. In the `get_book_view_cb` discussed earlier, the line

```
osso_abook_tree_model_set_book_view (OSSO_ABOOK_TREE_MODEL (app->
    contact_model), book_view);
```

will perform everything that is required.

Creating Contact View

To create a contact view, the application just uses the function `_abook_contact_view_new_basic ()`. This function takes the contact model that will be used to obtain the contacts.

```
app->contact_view = osso_abook_contact_view_new_basic (app->
    contact_model);
g_object_unref (app->contact_model);
```

This new widget can be treated just like any other widget, and placed in the UI in the usual way. Once the contact view has been created, the model is unrefed, as the application does not need to hold a reference to it anymore. This means that when the contact view is destroyed, the model will be as well. If the application needed to keep the model around after the view had been destroyed, then unrefing it would not be necessary here.

There is another, more powerful (but also more complicated) way of creating views that can automatically filter contacts based on a filter model: the function `osso_abook_contact_view_new ()`.

```
GtkWidget *osso_abook_contact_view_new (OssoABookContactModel *model,
                                         OssoABookFilterModel *
                                         filter_model);
```

Listing 7.104: `libosso-abook/osso-abook-contact-view.h`

Performing Actions on Contacts

In the Osso-Addressbook application, the main way of interacting with contacts is through the "contact starter dialog". This dialog is able to start chat and voip sessions with contacts, to start the composer to write an e-mail to a contact, and to edit, delete or organize contacts. It is a very powerful dialog, but one that is very simple to use in other applications.

In order to use the contact starter dialog, there has to be a contact (obviously), and a book view. The contact is needed to get the contact information, and the book view is used to listen to any changes in the contact that may happen while the dialog is open. **N.B.** Multiple applications are able to access the address book simultaneously, and while one program has a dialog open, another program may change some details in the contact, or even delete it. The stock dialogs and widgets in `libosso-abook` handle these cases, but it is important to remember that applications doing anything else with the address book need to handle the changes as well.

To create the contact starter dialog, the function `_abook_contact_starter_new()` is used. This function returns a widget derived from `GtkDialog`, and so can be manipulated using all the standard `GtkDialog` functions.

```
GtkWidget *osso_abook_contact_starter_new ();
```

Listing 7.105: libosso-abook/osso-abook-contact-starter.h

The contact and book view can be set on the dialog with the following functions:

```
void osso_abook_contact_starter_set_contact (OssoABookContactStarter *
    starter,
                                           EContact *contact);
void osso_abook_contact_starter_set_book_view (OssoABookContactStarter
    *starter,
                                           EBookView *book_view);
```

Listing 7.106: libosso-abook/osso-abook-contact-starter.h

After this, the `ContactStarter` can be treated just as a normal dialog.

Connecting ContactStarter Dialog to View

It is necessary to have a way to make the contact starter appear for a contact. Even though applications can perform in any chosen way, for this tutorial the dialog will appear whenever the user double-clicks on a contact in the list. The `OssoABookContactView` contains a signal that can help here, `contact_activated`. Its signature is

```
void (* contact_activated) (OssoABookContactView *view, EContact *
    contact);
```

Listing 7.107: libosso-abook/osso-abook-contact-view.h

It returns the contact that received the double-click, making it ideal in this situation.

```
static void
contact_activated_cb (OssoABookContactView *view,
                     EContact *contact,
                     gpointer user_data)
{
    AppData *app;
    GtkWidget *starter;

    app = user_data;
    starter = osso_abook_contact_starter_new ();
    osso_abook_contact_starter_set_book_view (
        OSSO_ABOOK_CONTACT_STARTER (starter),
        app->book_view);

    osso_abook_contact_starter_set_contact (
        OSSO_ABOOK_CONTACT_STARTER (starter),
        contact);

    gtk_dialog_run (GTK_DIALOG (starter));

    gtk_widget_destroy (starter);
}
```

```

/* ... */
g_signal_connect (app->contact_view, "contact-activated",
                  G_CALLBACK (contact_activated_cb), app);

```

With this simple callback function, the application can present the user with the ability to carry out very powerful functionality, and to integrate fully into the maemo platform.

Accessing Raw VCard Data

An EContact is an object derived from the EVCard object. This means that EContacts can be treated as EVCards, and anything that can be performed to an EVCard, can be performed to an EContact. To demonstrate this, the application has a button to dump the raw vcard data of any selected contacts.

This button is just a normal button, created with `gtk_button_new_with_label`, and added to the UI in the usual way.

```

app->dump_button = gtk_button_new_with_label ("Dump VCards");
gtk_box_pack_start (GTK_BOX (box), app->dump_button, FALSE, FALSE, 0);
gtk_widget_show (app->dump_button);

g_signal_connect (app->dump_button, "clicked", G_CALLBACK (dump_vcards)
                  , app);

```

In the `dump_vcards` function, the application gets a list of selected contacts and iterates through them, printing the vcards. There was an example above showing how to get a single contact that was selected from the contact view, but this requires a list. The `OssosBookContactView` contains a function that performs exactly that:

```

GList *osso_abook_contact_view_get_selection (OssosBookContactView *
                                              view);

```

Listing 7.108: `libosso-abook/osso-abook-contact-view.h`

This function returns a list of all the selected EContact objects. Iterating through them is a simple list operation. As mentioned above, the EContact object is also an EVCard object, meaning that the function `e_vcard_to_string` will return the raw VCard data. This function needs to know the format of the raw vcard data, either version 2.1 or version 3.0 of the VCard specification. The enums to do this are `EVC_FORMAT_VCARD_21` and `_FORMAT_VCARD_30` for version 2.1 and 3.0 respectively.

```

static void
dump_vcards (GtkWidget *button,
             gpointer user_data)
{
    AppData *app;
    GList *contacts, *c;
    int count;

    app = user_data;

    count = 1;
    contacts = osso_abook_contact_view_get_selection
               (OSSO_ABOOK_CONTACT_VIEW (app->contact_view));

```

```

    for (c = contacts; c; c = c->next) {
        EContact *contact = E_CONTACT (c->data);
        EVCard *vcard = E_VCARD (contact);
        char *v;

        v = e_vcard_to_string (vcard, EVC_FORMAT_VCARD_30);
        g_print ("Card %d\n", count);
        g_print ("%s", v);
        g_print ("-----\n");

        g_free (v);

        count++;
    }

    g_list_free (contacts);
}

```

It is not necessary to free the contacts in the list, but the list itself does need to be freed, or else it will leak.

Listening to Selection Change on Contact View

It is useful to know when the selection on the contact view has changed: the "Dump VCard" button in the tutorial is not of much use when there is no contact selected, so the proper way is to make it insensitive in these cases. Once again, `OssoABookContactView` provides a signal that is useful in these situations:

```

void (* selection_changed) (OssoABookContactView *view, guint
    n_selected_rows);

```

Listing 7.109: `libosso-abook/osso-abook-contact-view.h`

`selection_changed` tells the callback the number of selected rows, and that can be used to decide whether various controls are to be made sensitive.

```

static void
selection_changed_cb (OssoABookContactView *view,
                     guint n_selected_rows,
                     gpointer user_data)
{
    AppData *app;

    app = user_data;

    gtk_widget_set_sensitive (app->dump_button, (n_selected_rows >
        0));
}

```

Compiling Programs That Use Libosso-Abook

Libosso-abook provides a `pkgconfig` file for getting all the required `cflags` and library link flags to compile programs.

7.7.4 Using Autoconf

Applications using `autoconf` can add the lines


```
libosso
osso-addressbook-1.0
```

to the `PKG_CHECK_MODULES` macro to enable the libosso-abook options.

7.8 Clipboard Usage

In maemo, there is a number of clipboard enhancements to the X clipboard and Gtk+, in order to

- Support retaining the clipboard data when applications that own the clipboard exit.
- Be able to copy and paste rich text data between Gtk+ text views in different applications.
- Provide a generally more pleasant user experience; make it easy for application developers to gray out "Paste" menu items when the clipboard data format is not supported by the application.

7.8.1 GtkClipboard API Changes

```
gboolean gtk_clipboard_set_can_store (GtkClipboard *clipboard
                                     GtkTargetEntry *targets,
                                     gint             n_targets);
```

This function sets what data targets the current clipboard owner can transfer to the clipboard manager. NULL can be passed as targets, together with 0 as n_targets to indicate that all targets can be transferred.

When the clipboard owner changes, these values are reset.

```
void gtk_clipboard_store (GtkClipboard *clipboard);
```

This function tells the clipboard to try and store the contents of the targets specified using `gtk_clipboard_set_can_store`. If no such call has been made, or if there is no clipboard manager around, this function is simply a no-op.

Applications can call this function when exiting, but it is called automatically, when the application is quitting, if quitting with `gtk_main_quit()`. If the application is not the owner of the clipboard, the function will simply be a no-op.

In addition, adding a convenience function for finding out if a target is supported (in order to be able to gray out "Paste" items, if none of the existing clipboard targets are supported)

```
gboolean gtk_clipboard_wait_is_target_available (GtkClipboard *
                                                clipboard,
                                                GdkAtom         target);
```

7.8.2 GtkTextBuffer API Changes

In order to support rich text copy and paste, some new functions were introduced:

```

void
gtk_text_buffer_set_enable_paste_rich_text (GtkTextBuffer *buffer,
                                           gboolean
                                           can_paste_rich_text);

gboolean
gtk_text_buffer_get_enable_paste_rich_text (GtkTextBuffer *buffer);

```

The setter function toggles, whether it should be possible to paste rich text in a text buffer.

To prevent applications from getting confused, when text with unexpected tags is pasted to a buffer, the notion of "rich text format" was added:

```

void
gtk_text_buffer_set_rich_text_format (GtkTextBuffer *buffer,
                                     const gchar *format);
G_CONST_RETURN *
gtk_text_buffer_get_rich_text_format (GtkTextBuffer *buffer);

```

When a buffer has a certain text format, it can only paste rich text from buffers that have the same text format. If the formats differ, only plain text will be pasted. If a buffer has its format set to NULL, it means that it can paste from any format. For example, a format called "html" could include the tags "bold", "italic" etc. Thus, it would only be possible to paste text from buffers having the same format specified.

N.B. The string is just an identifier. It is up to the application developers to make sure that when specifying an application as supporting a certain format, also the tags in the buffer are specified for that format.

For further details, MaemoPad source code is a good example to study.

7.9 Global Search Usage

Global Search framework is provided for making different global searches for the content. It has capabilities to search local, Bluetooth gateway and UPnP server file systems. Some subsystems have their own search plug-ins, e.g. for searching bookmarks or e-mails.

Global Search is launched with a D-BUS message defining the search type:

OGS_DBUS_METHOD_SEARCH_BROAD	Search on all content.
OGS_DBUS_METHOD_SEARCH_FILE	Search only files.
OGS_DBUS_METHOD_SEARCH_EMAIL	Search only e-mails.
OGS_DBUS_METHOD_SEARCH_BOOKMARK	Search only bookmarks.

D-BUS call launches a Global Search dialog, which is used to make the search. Below is a simple example of a Global Search D-BUS call, showing the user a file search dialog.

```

#include <libogs/ogs.h>

osso_context_t *osso = osso_initialize("ogs_test", "0.1", FALSE, NULL);
osso_rpc_t foo;

osso_rpc_run_with_defaults(osso, OGS_DBUS_SERVICE,
                           OGS_DBUS_METHOD_SEARCH_FILE,
                           &foo, DBUS_TYPE_INVALID);

```

```
osso_rpc_free_val(&foo);
```

7.9.1 Global Search Plug-ins

Search plug-ins are libraries that are loaded into the process memory, when needed for searching. They are implemented as GObjects, using GTypeModule. They are searched for during the runtime and linked dynamically. For exact API descriptions, see the header files in libogs-dev package.

The following `ogs_module_*` functions are used for loading, registering and unloading the plug-in.

```
#include <libogs/ogs.h>

G_MODULE_EXPORT void ogsModule_load(OgsModule *module) {
    gstest_email_plugin_get_type (G_TYPE_MODULE (module));
}

G_MODULE_EXPORT void ogsModule_query(OgsModule *module) {
    ogs_search_module_register_plugin (OGS_SEARCH_MODULE (module),
                                       OGS_SEARCH_CATEGORY_EMAIL,
                                       gstest_email_plugin_type);
}

G_MODULE_EXPORT void ogsModule_unload (OgsModule *module) {
    /* free allocated memory here if needed */
}
```

`gstest_email_plugin_get_type()` simply registers the new type in the type system using a `GTypeInfo` structure, and finally registering the type with parent type `OGS_TYPE_PLUGIN`. In plug-in class initialization, query and finalize functions and search options were set.

```
static void gtest_email_plugin_class_init (GTestEmailPluginClass *
class)
{
    OgsPluginClass *plugin_class;
    GObjectClass *object_class;

    plugin_class = OGS_PLUGIN_CLASS (class);
    object_class = G_OBJECT_CLASS (class);

    parent_class = g_type_class_peek_parent(class);
    plugin_class->query = gtest_email_plugin_query;

    plugin_class->options_type = OGS_TYPE_SEARCH_OPTIONS;
    plugin_class->hit_type = OGS_TYPE_EMAIL_HIT;
    object_class->finalize = gtest_email_plugin_finalize;
}
```

In plug-in initialization, it is possible for `wexample` to set properties.

```
static void gtest_email_plugin_init(GTestEmailPlugin *plugin)
{
    g_object_set(plugin,
                 "id", "gtest-email-plugin",
                 "category", (gint) OGS_SEARCH_CATEGORY_EMAIL,
                 NULL);
}
```

```

    /* ... */
}

```

Query function does the searching, and emits a signal when new hits are found.

```

static void gstest_email_plugin_query(OgsPlugin      *plugin,
                                      const gchar     *query,
                                      OgsSearchContext *context,
                                      OgsFileSystem    *fs,
                                      OgsSearchCategory category,
                                      OgsSearchOptions *options)
{
    GSList *l;
    gulong timestamp;

    if (!ogs_plugin_supports_category (plugin, category)) {
        return;
    }

    if (query == NULL || strlen (query) == 0) {
        return;
    }

    timestamp = time (NULL);

    for (l = GSTEST_EMAIL_PLUGIN(plugin)->mails; l; l = l->next) {
        OgsSearchHit *hit;
        TestEmail *mail;
        int i;

        mail = (TestEmail *) l->data;

        if (!strstr(mail->sender, query) &&
            !strstr(mail->subject, query) &&
            !strstr(mail->mailbox, query)) {
            continue;
        }

        hit = g_object_new(OGS_TYPE_EMAIL_HIT,
                           "name", mail->subject,
                           "folder", mail->mailbox,
                           "contact", mail->sender,
                           "mime-type", "email",
                           "timestamp", timestamp,
                           "size", (gulong) 12345,
                           "category", (gint) category,
                           "has_attachment", mail->has_attachment,
                           NULL);

        timestamp += 60*60*24*3;

        g_signal_emit_by_name(plugin, "new-hit", hit, context);

        g_object_unref(hit);

        for (i = 0; i < 4; ++i) {
            if (ogs_search_context_is_cancelled(context)) {
                return;
            }

            g_usleep(500);
        }
    }
}

```

```

    }
}

```

Finalize function is naturally for finalizing the plug-in by freeing memory.

```

static void gstest_email_plugin_finalize(GObject *object)
{
    GSList *l;
    GSTestEmailPlugin *plugin;

    plugin = GSTEST_EMAIL_PLUGIN (object);

    /* ... */

    G_OBJECT_CLASS(parent_class)->finalize(object);
}

```

7.10 Writing "Send Via" Functionality

Send Via functionality is provided by the platform to enable applications to send data via e-mail, or over a Bluetooth connection. Since several applications share this functionality, the platform provides a public interface to facilitate deployment of these services in user applications. The interfaces are defined in two header files: libmodest-dbus-client.h (in package libmodest-dbus-client-dev) and conbtdialogs-dbus.h (in package conbtdialogs-dev). The following sample code is an example of the usage of these interfaces. See MaemoPad source code for a fully functional application using these services.

```

/*send via email*/
#include <libmodest-dbus-client/libmodest-dbus-client.h>
/*send via bt */
#include <conbtdialogs-dbus.h>

/* ... */

void callback_sendvia_email ( GtkAction * action, gpointer data )
{
    gboolean result = TRUE;
    GSList *list = NULL;
    AppUIData *mainview = NULL;
    mainview = ( AppUIData * ) data;

    /* Attach the saved file (and not the one currently on screen). If
       the file
       * has not yet been saved, nothing will be attached */

    if (mainview->file_name) {
        list = g_slist_append(list, mainview->file_name);
        result = libmodest_dbus_client_compose_mail(mainview->data->
            osso, /*osso_context_t*/
            NULL, /*to*/
            NULL, /*cc*/
            NULL, /*bcc*/
            NULL, /*body*/
            NULL, /*subject*/
            list /*attachments*/);
    }
}

```

```

g_slist_free(list);

if (result == FALSE) {
    g_print("Could not send via email\n");
}
}

gboolean rpc_send_via_bluetooth(gchar *path)
{
    DBusGProxy *proxy = NULL;
    DBusGConnection *sys_conn = NULL;
    GError *error = NULL;
    gboolean result = TRUE;
    gchar **files = NULL;

    sys_conn = dbus_g_bus_get(DBUS_BUS_SYSTEM, &error);

    if(sys_conn == NULL)
    {
        return FALSE;
    }

    files = g_new0(gchar*, 2);
    *files = g_strdup(path);
    files[1] = NULL;

    /* Open connection for btdialogs service */
    proxy = dbus_g_proxy_new_for_name(sys_conn,
                                      CONBTDIALOGS_DBUS_SERVICE,
                                      CONBTDIALOGS_DBUS_PATH,
                                      CONBTDIALOGS_DBUS_INTERFACE);

    /* Send send file request to btdialogs service */
    if (!dbus_g_proxy_call(proxy, CONBTDIALOGS_SEND_FILE_REQ,
                          &error, G_TYPE_STRV, files,
                          G_TYPE_INVALID, G_TYPE_INVALID))
    {
        g_print("Error: %s\n", error->message);
        g_clear_error(&error);
        result = FALSE;
    }

    g_strfreev(files);

    g_object_unref(G_OBJECT(proxy));
    return result;
}

```

Listing 7.110: maemopad/src/ui/callbacks.c

7.11 Using HAL

This section gives a quick tour of HAL (Hardware Abstraction Layer). For in-depth background information, technical documentation and a specification, consult <http://www.freedesktop.org/wiki/Software/hal>.

7.11.1 Background

The purpose of HAL is to provide means for storing data about hardware devices, gathered from multiple sources and to provide an interface for applications to access this data. In essence, HAL provides a list of devices, and a way to add configuration values for each device. As a concrete example of what kind of information HAL contains, the command line application 'lshal' can be used to query the HAL daemon for all device objects that it is aware of. Here is some example output from Internet Tablet device:

```
Dumping 54 device(s) from the Global Device List:
-----
udi = '/org/freedesktop/Hal/devices/computer'
  info.addons = {'hald-addon-cpufreq'} (string list)
  info.bus = 'unknown' (string)
  info.callouts.add = {'hal-storage-cleanup-all-mountpoints'} (string
    list)
  info.capabilities = {'cpufreq_control'} (string list)
  info.interfaces = {'org.freedesktop.Hal.Device.SystemPowerManagement
    ',
    'org.freedesktop.Hal.Device.CPUFreq'} (string list)
  info.product = 'Computer' (string)
  info.subsystem = 'unknown' (string)
  info.udi = '/org/freedesktop/Hal/devices/computer' (string)
  org.freedesktop.Hal.Device.SystemPowerManagement.method_argnames =
{'num_seconds_to_sleep', 'num_seconds_to_sleep', '', '', '', '
  enable_power_save'}
(string list)
  org.freedesktop.Hal.Device.SystemPowerManagement.method_execpaths =
{'hal-system-power-suspend', 'hal-system-power-suspend-hybrid', 'hal-
  system-power-hibernate',
  'hal-system-power-shutdown', 'hal-system-power-reboot', 'hal-system-
  power-set-power-save'}
(string list)
  org.freedesktop.Hal.Device.SystemPowerManagement.method_names =
{'Suspend', 'SuspendHybrid', 'Hibernate', 'Shutdown', 'Reboot', '
  SetPowerSave'}
(string list)
  org.freedesktop.Hal.Device.SystemPowerManagement.method_signatures =
{'i', 'i', '', '', '', 'b'} (string list)
  power_management.can_hibernate = false (bool)
  power_management.can_suspend = true (bool)
  power_management.can_suspend_hybrid = false (bool)
  power_management.can_suspend_to_disk = false (bool)
  power_management.can_suspend_to_ram = true (bool)
  power_management.is_powersave_set = false (bool)
  system.firmware.name = 'NOLO' (string)
  system.firmware.product = 'N800' (string)
  system.firmware.version = '1.1.6' (string)
  system.formfactor = 'unknown' (string)
  system.hardware.product = 'RX-34' (string)
  system.hardware.serial = '0000000000000000' (string)
  system.hardware.uuid = '24202524' (string)
  system.hardware.vendor = 'Nokia' (string)
  system.hardware.version = '1301' (string)
  system.kernel.machine = 'armv6l' (string)
  system.kernel.name = 'Linux' (string)
  system.kernel.version = '2.6.21-omap1' (string)

udi = '/org/freedesktop/Hal/devices/computer_mtd_0_8'
  info.bus = 'mtd' (string)
```

```

info.parent = '/org/freedesktop/Hal/devices/computer' (string)
info.product = 'MTD Device' (string)
info.subsystem = 'mtd' (string)
info.udi = '/org/freedesktop/Hal/devices/computer_mtd_0_8' (string)
linux.device_file = '/dev/mtd4ro' (string)
linux.hotplug_type = 2 (0x2) (int)
linux.subsystem = 'mtd' (string)
linux.sysfs_path = '/sys/class/mtd/mtd4ro' (string)
mtd.host = 0 (0x0) (int)

...

udi = '/org/freedesktop/Hal/devices/
platform_mmci_omap1_mmc_host_mmc_card_rca58916_storage'
block.device = '/dev/mmcblk0' (string)
block.is_volume = false (bool)
block.major = 254 (0xfe) (int)
block.minor = 0 (0x0) (int)
block.storage_device =
'/org/freedesktop/Hal/devices/
platform_mmci_omap1_mmc_host_mmc_card_rca58916_storage'
(string)
info.capabilities = {'storage', 'block'} (stringlist)
info.category = 'storage' (string)
info.parent =
'/org/freedesktop/Hal/devices/
platform_mmci_omap1_mmc_host_mmc_card_rca58916'
(string)
info.udi =
'/org/freedesktop/Hal/devices/
platform_mmci_omap1_mmc_host_mmc_card_rca58916_storage'
(string)
linux.hotplug_type = 3 (0x3) (int)
linux.sysfs_path = '/sys/block/mmcblk0' (string)
storage.automount_enabled_hint = true (bool)
storage.bus = 'mmc' (string)
storage.drive_type = 'sd_mmc' (string)
storage.hotpluggable = true (bool)
storage.media_check_enabled = false (bool)
storage.model = '' (string)
storage.no_partitions_hint = false (bool)
storage.originating_device =
'/org/freedesktop/Hal/devices/
platform_mmci_omap1_mmc_host_mmc_card_rca58916' (string)
storage.partitioning_scheme = 'none' (string)
storage.physical_device =
'/org/freedesktop/Hal/devices/
platform_mmci_omap1_mmc_host_mmc_card_rca58916' (string)
storage.removable = false (bool)
storage.removable.media_available = true (bool)
storage.removable.media_size = 125960192 (0x7820000) (uint64)
storage.requires_eject = false (bool)
storage.size = 125960192 (0x7820000) (uint64)
storage.vendor = '' (string)

```

As can be seen in the output, the configuration data for each device, identified by a UDI (Unique Device Identifier), is stored as key-value pairs. Similar information can be queried about each device known to HAL. For instance, the type and size of an MMC memory card can be found in the HAL database, as well as other essential information (see the last device in the above example). HAL also contains information about disk volumes, such as file system types,

mount points, volume sizes etc. The 'category' and 'capabilities' fields of the configuration describe what the device is and what it does. For a complete description of the device properties, see [HAL specification](#) [43].

The following example illustrates data contained in HAL about the Internet Tablet's camera device. The example demonstrates an important aspect of HAL: its ability to monitor the state of each device in real-time. State changes in devices known to HAL are broadcast to D-BUS. Thus, the state of each interesting device from an application's point of view can be asynchronously monitored. Notice how the database keeps track of the exact state of the camera in the following example: whether the camera is active, and whether it has been turned 180 degrees.

```
udi = '/org/freedesktop/Hal/devices/platform_cam_turn'
button.has_state = true (bool)
button.state.value = true (bool)
button.type = 'activity' (string)
info.addons = {'hald-addon-omap-gpio'} (string list)
info.bus = 'platform' (string)
info.capabilities = {'button'} (string list)
info.linux.driver = 'gpio-switch' (string)
info.parent = '/org/freedesktop/Hal/devices/platform_gpio_switch' (string)
info.product = 'Platform Device (cam_turn)' (string)
info.subsystem = 'platform' (string)
info.udi = '/org/freedesktop/Hal/devices/platform_cam_turn' (string)
linux.hotplug_type = 2 (0x2) (int)
linux.subsystem = 'platform' (string)
linux.sysfs_path = '/sys/devices/platform/gpio-switch/cam_turn' (string)
platform.id = 'cam_turn' (string)

udi = '/org/freedesktop/Hal/devices/platform_cam_act'
button.has_state = true (bool)
button.state.value = true (bool)
button.type = 'activity' (string)
info.addons = {'hald-addon-omap-gpio'} (string list)
info.bus = 'platform' (string)
info.capabilities = {'button'} (string list)
info.linux.driver = 'gpio-switch' (string)
info.parent = '/org/freedesktop/Hal/devices/platform_gpio_switch' (string)
info.product = 'Platform Device (cam_act)' (string)
info.subsystem = 'platform' (string)
info.udi = '/org/freedesktop/Hal/devices/platform_cam_act' (string)
linux.hotplug_type = 2 (0x2) (int)
linux.subsystem = 'platform' (string)
linux.sysfs_path = '/sys/devices/platform/gpio-switch/cam_act' (string)
platform.id = 'cam_act' (string)
```

It is important to notice, that the devices in HAL database do not necessarily have a one-to-one correspondence with a physical device. For instance, in the previous example the camera has two separate devices in HAL for two different functionalities. This is intentional: the purpose of HAL is to abstract the entire set of physical devices into a *functional* categorization. The devices in HAL represent the smallest addressable unit. [43]

7.11.2 C API

In order to make developers' life easier, the communication to the HAL daemon has been wrapped into a C library. Thus, applications can easily use the services provided by the library to monitor and query different devices. For a complete description of the API, see `libhal.h` (in package `libhal-dev`). Here, the basic steps to set up a connection to HAL and some basic functions are described. Error checking etc. are omitted from the examples, for the sake of clarity.

Setting up a connection to HAL is performed as follows:

```
include <libhal.h>

void set_up(void)
{
    LibHalContext *ctx;
    DBusConnection *dbus_connection;
    DBusError error;

    ctx = libhal_ctx_new();

    dbus_error_init(&error);
    dbus_connection = dbus_bus_get(DBUS_BUS_SYSTEM, &error);
    libhal_ctx_set_dbus_connection(ctx, dbus);

    /* ... */
}
```

After setting up a connection, listener callbacks can be registered:

```
libhal_ctx_set_device_added (ctx, _device_added);
libhal_ctx_set_device_removed (ctx, _device_removed);
libhal_ctx_set_device_property_modified(ctx, _property_modified);
```

Finally, initialize the connection and start listening to HAL:

```
libhal_ctx_init(ctx, &error);
libhal_device_property_watch_all(ctx, &error);
```

The listener functions and the application in general can obtain information about devices using functions in libhal. For instance, as a device gets connected, the following functions can be used to obtain information about it:

```
/* If UDI is known (the callbacks provide a UDI for added/removed
   devices),
   existence of specific capabilities can be queried with the
   following function: */
if(libhal_device_query_capability(ctx, udi, "volume", NULL)) {
    /* Capability "volume" exists */
    ...
}

/* The library also provides plenty of get/set functions for
   querying specific
   information about a device by its UDI. For instance, querying
   the storage.
   drive_type property of the device, which is created by HAL when
   a memory card
   is inserted into the Internet Tablet device, returns 'sd_mmc',
   indicating an
   active memory card reader for SecureDigital/MultiMediaCard
   memory cards. */
libhal_device_get_property_string (ctx, udi, "storage.drive_type",
    NULL);
```

All the query functions follow the same pattern. The HAL specification describes the type of each property. The types can optionally be queried with the 'lshal' tool, which adds type information into its output. For each type, libhal provides a get/set function, such as libhal_device_get_property_string(), libhal_device_get_property_int(), etc. The library also provides functions for listing all devices and their properties: libhal_get_all_devices(), _device_get_all_properties(), if more user-controlled filtering is needed.

7.12 Certificate Storage Guide

This section presents the first material to be read by developers involved with the Maemo Certificate Manager API. It offers a gradual approach to certificates and their management on the maemo platform.

This material is complemented by the [Maemo Certificate Manager API Reference](#). Even though this material introduces some cryptographic notions and OpenSSL commands/functions, it is not intended in any way to be a reference on OpenSSL or cryptography.

It is recommended that the [API sample programs](#) are downloaded and compiled before reading this section, since some sections refer to these programs.

7.12.1 Digital Certificates

Public key encryption, also known as asymmetric encryption, is essentially a scheme where the encryption key differs from the decryption key. Once the message is encrypted, the encryption key cannot decrypt it.

The encryption key is also known as the public key, since it can be distributed without fear of revealing the decryption (i.e. private) key. Typically, a user creates a pair of keys, then distributes the public key (for example, on a public key server), and when this is done, the user is able to receive securely encrypted messages, and can be assured that nobody else is able to decrypt the received messages, unless they have access to the private key.

Therefore, keeping the private key safe is a major concern, and most schemes encrypt the private key locally, demanding a password or security card to disclose it.

In most public key systems, the keys are mathematically related and can be used in inverse order, so encrypting with the private key and decrypting with public key also works. However, encrypting a message with the private key does not guarantee secrecy, since anybody can decrypt it with the widely known public key.

But it allows a second, equally important feature of public key systems: authentication or signing. Encrypting with the private key proves that the sender possesses the private key in question, so (provided the private key is safely guarded) it must be the same person who has issued the public key. Authentication also means non-repudiation (i.e. the sender cannot deny sending the message).

Public key systems based on prime numbers are very slow. So, most practical encryption schemes do not apply it to whole messages, only to message hashes (for authentication) and for symmetric keys (for encryption).

X.509 certificates are a portable and standardized way to distribute public keys along with the identity of the holder. It contains some information about the holder, including but not limited to:

- The name of the certificate holder, also known as Distinguished Name (DN)
- Contact e-mail
- Internet domain

- Address
- Issue date
- Activation date
- Expiration date
- Public key
- Private key (must never be included in the publicly available certificate).
- Certification authority (see below)
- Certification authority signature (see below)

Since it is easy to generate a certificate claiming to be anybody, a trustable certificate is not generated by the holder. Instead, it is generated and signed by a trusted third-party - a certification authority (CA), e.g. Verisign or CACert. The CA is expected to make positive checks on the certificate requester in order to verify the identity of the requester (without this check, the CA signature would be worthless).

If, for example, the Web client of a bank needs to establish an encrypted connection, it asks for the certificate and checks the signature against the CA's certificate. Obviously, the CA certificate is necessary to do this. In practice, most products, including Web browsers, come bundled with several well-known CA certificates, so the user rarely needs to download and install a new CA certificate.

A CA certificate is generally self-signed, since it is the highest trusted entity in the certificate hierarchy. It is also possible to build a chain of trust: a CA certificate may be signed by another CA. More commonly, an entity generates its own certificates by itself, signing them with the CA-issued one. This is commonly done, for instance, for the establishment of VPNs and wireless connections, where the certificate is only to be used within the entity.

Each CA maintains a monotonic serial number counter that is incremented with every new certificate. In this way, every CA-signed (and therefore widely trustable) certificate has a worldwide unique ID, consisting of the certificate issuer and the serial number.

Self-signed non-CA certificates are also common on the Internet. Since CA certificates cost money, several sites choose to generate the certificate themselves. Most web browsers warn the user about these untrusted certificates, and prompt for permission to continue connecting. An alternative method would be to request a certificate from CACert (www.cacert.org), which is free of charge. The required root certificate is already included in the product, and in most of the free web browsers as well.

Visiting again the Web client/bank example, most HTTPS connections only check the certificate of the server. It is also possible for a client to have its own certificate, and some services may even demand it, since the server needs to be sure that the client is who it claims to be.

X.509 certificates are incompatible with PGP, and different from it in several aspects. X.509 certificates are formally signed by trusted authorities wishing to earn money with this service; PGP/GPG keys rely on a web of trust and are

signed by other PGP key holders (a key signed by a trusted person becomes itself trusted, at least in the niche to which the signed key belongs).

S/MIME is a standard separate from X.509, but both appear together in this maemo API, because an S/MIME message sender needs to know the public key and cryptographic capabilities of the message recipient. That information is provided by the X.509 public certificate of the recipient.

Certificate Revocation Lists (CRL) are lists of certificates that have actively expired before their natural expiration date. A typical reason for premature expiration is the leaking of the private key. The diffusion of CRLs in a timely manner remains a challenge.

7.12.2 Certificates in Maemo Platform

The maemo platform offers an API to deal with certificate manager storage and manipulation. This enables all the software to have access to all certificates so that, for example, installing a new CA certificate takes immediate effect in all relevant programs (such as Web browser, e-mail, VPN and wireless connection). This saves both effort and disk space.

7.12.3 Creating Own Certificates with OpenSSL

It is useful to explain how certificates are created by using them in the code examples. OpenSSL tools are used for that purpose. OpenSSL tools are not installed by default, so they must be installed manually. In order to install the package "openssl", the user must be either root or set Application manager to the Red Pill mode (see section *Red Pill Mode 13.3.3* of the chapter *Packaging, Deploying and Distributing* in Maemo Reference Manual). The important configurations for certification creation at `/etc/ssl/openssl.cnf` are:

```
[ CA_default ]
dir = /root/certificates      # Where all the files are kept
certs = $dir/certs           # Where the issued certs are kept
crl_dir = $dir/crl            # Where the issued crls are kept
database = $dir/index.txt     # database index file.
new_certs_dir = $dir/newcerts # default place for new certs.
certificate = $dir/my-ca.crt  # The CA certificate
serial = $dir/serial          # The current serial number
crl = $dir/crl.pem            # The current CRL
private_key = $dir/my-ca.key  # The private key
default_days = 1095           # how long to certify for
```

Once configured, the Certificate Authority can be created with the command line tool openssl:

```
cd /root/certificates
touch index.txt
echo 01 > serial
openssl req -nodes -new -x509 -keyout my-ca.key -out my-ca.crt
```

All required fields should be filled with sensible data, otherwise the certificate may be rejected by some software. The `my-ca.key` file contains the CA's private key, which must be kept safe, because the CA security depends on its secrecy. `my-ca.crt` is the public distributable CA certificate.

When this has been done, it is possible to create certificates. The first command creates the certificate, the second command signs it:

```
openssl req -nodes -new -keyout certificate.key -out certificate.csr
openssl ca -out certificate.crt -in certificate.csr
```

OpenSSL learns from the `/etc/ssl/openssl.cnf` configuration, which CA to use for signing the new regular certificate. Again, all requested data fields should be filled.

The `certificate.key` file contains the private key, and it must be safely guarded by the certificate holder; `certificate.crt` is the signed public certificate; `certificate.csr` is the unsigned intermediate certificate that can be discarded.

Queries can be made to a certificate file content by issuing

```
openssl x509 -in cert01.crt -noout -text
```

OpenSSL can also be ordered to include all the data in human-readable text format, alongside the binary (base64-encoded) certificate. Finally, to make an actual encryption test with public keys (this test should be performed using small messages):

```
# encrypt to a public certificate recipient
cat certificate.crt | openssl rsautl -encrypt -in message -certin -out message.enc

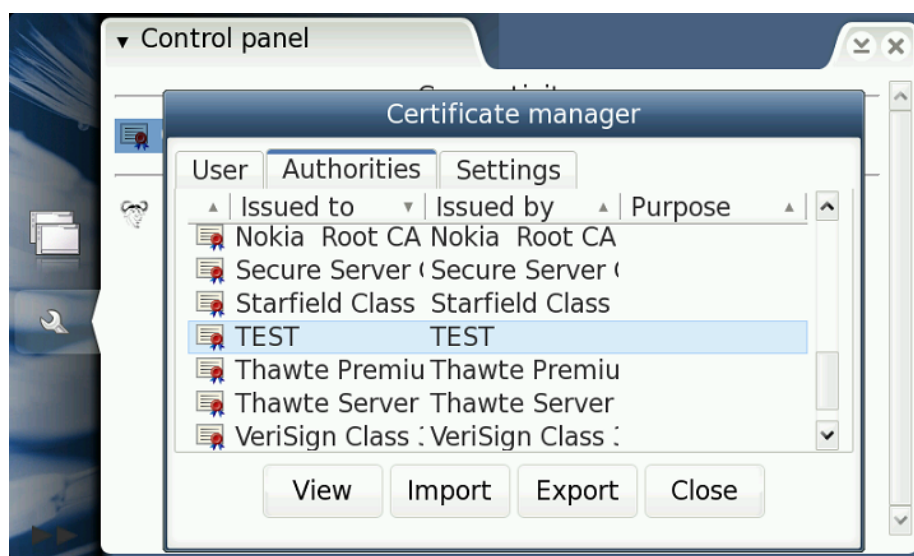
# recipient decrypts using the private key
openssl rsautl -decrypt -in message.enc -inkey certificate.key
```

7.12.4 Maemo Certificate Databases

In maemo, certificates are stored in Berkeley DB version 1 format files. Fortunately, the API user never needs to cope with low-level details of BDB databases. The default maemo certificate file is `/usr/share/certs/certman.cst`, even though alternative databases can be created and used.

The figure below shows the Certificate Manager plug-in for the maemo control panel, with the focus on a certification authority (CA) that has been created using OpenSSL and manually inserted in the default maemo certificate file, using the sample import program with following command:

```
./import /usr/share/certs/certman.cst ca my-ca.crt
```



Inside every database, there is a set of "folders", a simple classification of the certificates useful to limit the scope of searches. The constants for these folders are: CST_FOLDER_CA, CST_FOLDER_OTHER, CST_FOLDER_PERSONAL, CST_FOLDER_SITE and CST_FOLDER_UNKNOWN. These constants can be found in most code samples.

Even though certificate UIDs are said to be unique, most functions in the API refer to certificates by a storage ID. This ID has no relationship with UID, and it is unique only within a specific database.

The following sections comment on snippets of code with basic certificate manager operations. It is recommended that the accompanying example programs are tried while reading this material. The samples must be compiled inside Scratchbox with the maemo SDK installed.

7.12.5 Creating Databases

```
$ ./dbcreate test.db
```

This program only opens the specified database file. If it does not exist, it must be created. This behavior is provided by the open file API call:

```
// CST_open_file(filename, readonly [ignored], password [ignored]);
storage = CST_open_file(argv[1], FALSE, NULL);
if (! storage) {
    printf("Could not create certificate database %s.\n", argv[1]);
    return 1;
}
```

Listing 7.111: certman-examples/dbcreate.c

After opening or creating the database, the program counts the available certificates, walking the GSList tree. This list is found by searching by purpose, specifying "any" as the purpose.

```
// get any certificate we find
certificates = CST_search_by_purpose(storage, CST_PURPOSE_NONE);
for (i = certificates; i; i = i->next) {
    ++count;
}
if (certificates > 0) {
    printf("Database already exists and it has %d certificates
        inside.", count);
}
```

Listing 7.112: certman-examples/dbcreate.c

Finally, all allocated resources must be freed:

```
g_slist_free(certificates);
CST_free(storage);
```

Listing 7.113: certman-examples/dbcreate.c

7.12.6 Importing Certificates and Keys

```
$ ./import test.db ca certs/ca.crt
$ ./import test.db server certs/cert01.crt
$ ./import test.db client certs/cert02.crt certs/cert02.key
```

The first example imports a CA certificate. The second one imports a regular SSL server certificate. The third one imports a SSL client certificate. Since this is supposed to be your own certificate, you also need to import the related private key in order to sign and decrypt messages.

The most important API call here is `CST_import_cert`. An open file must be provided for it (with the certificate inside).

```
// get present list
oldlist = CST_search_by_purpose(storage, CST_PURPOSE_NONE);
fcert = fopen(cert, "r");
if (! fcert) {
    printf("Certificate file could not be open, errno = %d\n",
        errno);
    CST_free(storage);
    return 1;
}
err = CST_import_cert(storage, fcert, NULL);
if (err) {
    printf("Error %d when trying to import certificate %s\n", err,
        cert);
    CST_free(storage);
    return 1;
}
```

Listing 7.114: certman-examples/import.c

It is necessary to know, which certificate has just been imported. There is a quick method to pinpoint it: by comparing two lists, one collected before import, the second collected just after.

```
// get new list
list = CST_search_by_purpose(storage, CST_PURPOSE_NONE);

// discover new certID by comparing the two lists
for (i = list; i; i = i->next) {
    if (! g_slist_find(oldlist, i->data)) {
        certID = GPOINTER_TO_UINT(i->data);
    }
}

g_slist_free(list);
g_slist_free(oldlist);
if (! certID) {
    printf("Newly imported certificate not found!\n");
    CST_free(storage);
    return 1;
}
```

Listing 7.115: certman-examples/import.c

After getting the `certID` of the new certificate, its purpose should be set (based on command line parameter) and its account name gotten, since this name is the identifier for the forthcoming import key.

```
CST_set_purpose(storage, certID, purpose, TRUE);

printf("Getting x.509 certificate...\n");
x509cert = CST_get_cert(storage, certID);

printf("Getting x.509 subject name...\n");
// should NOT be freed since X509_get_*(X509* certificate)
```



```
// return pointers to certificate's own memory
account = X509_get_subject_name(x509cert);
```

Listing 7.116: certman-examples/import.c

If it is necessary to import a private key, CST_import_priv_key does the hard work.

```
if (privkey) {
    fprivkey = fopen(privkey, "r");
    if (! fprivkey) {
        printf("Key file could not be open, errno = %d\n",
            errno);
        CST_free(storage);
        return 1;
    }
    printf("Importing key...\n");
    err = CST_import_priv_key(storage, account, fprivkey,
        password, password);
    if (err) {
        printf("Error %d when trying to import private
            key %s\n",
            err, privkey);
        CST_free(storage);
        return 1;
    }
    /* ... */
}
```

Listing 7.117: certman-examples/import.c

Now both the certificate and the private key are in storage, but they are not bound to each other. The private key can be searched for by its account name. Then CST_assign can be called to bind the certificate to the key.

```
keylist = CST_priv_key_search_by_name(storage, account);

if (! keylist) {
    printf("Error %d when trying to list of appended keys\n",
        CST_last_error());
    CST_free(storage);
    return 1;
}

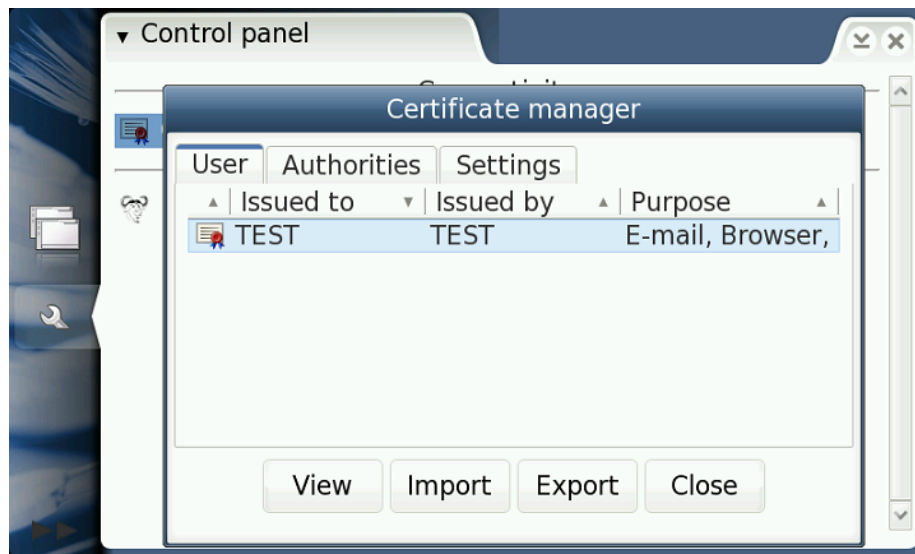
// picks the first of the list
keyID = GPOINTER_TO_UINT(keylist->data);

printf("Newly imported private key is keyID %d\n", keyID);

err = CST_assign(storage, certID, keyID, password);
```

Listing 7.118: certman-examples/import.c

The figure below shows the Certificate Manager with a client certificate created by OpenSSL and manually inserted in maemo certificate file using the import sample program.



7.12.7 Sample Program for Searching and Listing Certificates

```
$ ./listcerts testdb.cst ca
$ ./listcerts testdb.cst any
```

This program collects a list of certificates given a purpose, and lists the certIDs on the screen in a simple manner. The list collecting is here performed by `CST_search_by_purpose`, but there is a whole family of `CST_search_*` functions to choose from, covering all search needs. Most search functions return lists instead of X.509 certificates, since most search arguments are not unique (i.e. they may match with an unbound number of registers). Code snippet:

```
certificates = CST_search_by_purpose(storage, purpose);
for (i = certificates; i; i = i->next) {
    printf("CertID found: %d\n", GPOINTER_TO_UINT(i->data));
    ++count;
}
```

Listing 7.119: certman-examples/listcerts.c

An interesting detail about searching by purpose: searching by `_PURPOSE_NONE` returns all stored certificates, since only certificates matching all specified purpose bits are returned; `CST_PURPOSE_NONE` equals zero, so no purpose is specified, so all certificates are fit.

7.12.8 Deleting Certificates

```
$ ./delcert testdb.cst 101
```

This sample deletes a certificate with a given storage and certID. (The ID of a certificate can be obtained by using the `listcerts` program). The X.509 certificate must be retrieved through `CST_get_cert()`, only to make sure that it exists.

```
certificate = CST_get_cert(storage, certID);
if (! certificate) {
    if (! CST_last_error()) {
```

```

        printf("Certificate %d does not exist\n", certID);
        CST_free(storage);
        return 1;
    } else {
        printf("Error %d while getting certificate\n",
            CST_last_error());
        CST_free(storage);
        return 1;
    }
}

```

Listing 7.120: certman-examples/delcert.c

Before releasing the X.509 certificate, some data should be chosen from it using OpenSSL functions. This is different from most functions that return pointers; the functions `X509_get_*_name` return `X509_NAME` pointers to the certificate memory, so they need not be released.

```

issuer = X509_NAME_oneline(X509_get_issuer_name(certificate), NULL, 0);
subject = X509_NAME_oneline(X509_get_subject_name(certificate), NULL,
    0);
printf("Issuer: %s Subject (holder): %s\n", issuer, subject);

```

Listing 7.121: certman-examples/delcert.c

As `X509_NAME` objects cannot be displayed directly, they should be converted to characters using `X509_NAME_oneline()`. This function returns buffers that must be freed.

```

free(issuer);
free(subject);
X509_free(certificate);

```

Listing 7.122: certman-examples/delcert.c

Finally, the certificate should be deleted from the storage:

```

err = CST_delete_cert(storage, certID);

```

Listing 7.123: certman-examples/delcert.c

7.12.9 Validating Certificate Files

```

$ ./validate test.db certs/cert01.crt

```

This program validates a certificate file. Since `CST_is_valid*` functions still do not check the trust chain, they accept any non-corrupted certificate as valid. This behavior will change in future API versions (this is why this function already needs the storage parameter). Code:

```

fcert = fopen(argv[2], "r");
if (! fcert) {
    printf("Certificate file could not be open, errno = %d\n",
        errno);
    CST_free(storage);
    return 1;
}
if (CST_is_valid_f(storage, fcert, NULL)) {
    printf("Certificate is valid.\n");
} else if (CST_last_error()) {

```

```

    printf("Error while validating certificate\n.");
} else {
    printf("Certificate is invalid (corrupted or not trusted)\n");
}
fclose(fcrt);

```

Listing 7.124: certman-examples/validate.c

7.12.10 Exporting Certificates

```
$ ./export test.db 101
```

The program source is almost equal to the delcert program, except that the certificate is exported to stdout instead of deleting it:

```
err = CST_export_cert_by_id(storage, certID, stdout);
```

Listing 7.125: certman-examples/export.c

7.13 Extending Hildon Input Methods

7.13.1 Overview

Maemo platform is intended to be used on embedded devices. It is a quite straightforward request that one might want to have different input methods from the ones available by default, or just simply want a different layout for the virtual keyboard. For this reason, maemo 4.1 introduces a way to enable writing custom plug-ins for Hildon Input Method.

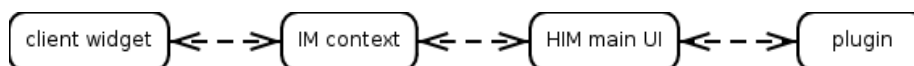
This section describes writing a simple plug-in for Hildon Input Method. This example plug-in is going to implement a very basic virtual keyboard.

The illustration below shows what area of the Hildon Input Method can be redefined with the custom plug-in.



Technically, the plug-in is an almost standard GTK widget (with additional steps to support dynamic loading). The widget of the plug-in will be placed in the Plug-in area.

The illustration below shows the data flow of a user's input:



User inputs directly to the plug-in, then the plug-in sends the inputted data to HIM main user interface. The HIM main UI will interact with the IM context, and then commit the inputted text to the client widget.

In case of a custom plug-in, the plug-in itself - i.e. its writer - is responsible for handling all the inputs (even the buttons that are part of HIM main UI, e.g. tab or enter) and propagate them to the right modules (e.g. IM context).

The function and outlook of the buttons in HIM main UI can be customized, but one cannot remove them completely from the UI - only dim them (see handwriting plug-in), and they cannot be rearranged (for further information, see section Common buttons).

7.13.2 Plug-in Features

Hildon Input Method plug-in must be a GTK widget. In addition to the GTK widget interface, the plug-in must implement certain functions.

Interface

As already mentioned, the plug-in must handle all the inputs - both real user input and management signals from the system - that are coming to the HIM main UI. The first step is to take a look at which functions need to be implemented and provided by the plug-in to the Hildon Input Method Plug-in Interface. Later on, it will be shown how these functions are actually registered for the HIM Plug-in Interface.

The essential functions that must be implemented by a basic plug-in:

- `void (*enable) (HildonIMPlugin *plugin, gboolean init);`

Listing 7.126: hildon-input-method/hildon-im-plugin.h

enable is called whenever the plug-in becomes available to the user. **init** holds TRUE whenever this is the initialization time.

```
/* Called when the plugin is available to the user */
static void
enable (HildonIMPlugin *plugin, gboolean init)
{
    HimExampleVKBPrivate *priv;
    HimExampleVKB *vkb;

    vkb = HIMEXAMPLE_VKB (plugin);
    priv = HIMEXAMPLE_VKB_GET_PRIVATE (vkb);
    if (init == TRUE)
    {
        hildon_im_ui_button_set_toggle (priv->ui,
                                         HILDON_IM_BUTTON_MODE_A, TRUE);
        hildon_im_ui_button_set_toggle (priv->ui,
                                         HILDON_IM_BUTTON_MODE_B, TRUE);
        hildon_im_ui_button_set_label (priv->ui,
                                       HILDON_IM_BUTTON_MODE_A, "ABC");
        hildon_im_ui_button_set_label (priv->ui,
                                       HILDON_IM_BUTTON_MODE_B, "Shift");
    }

    hildon_im_ui_send_communication_message(priv->ui,
                                             HILDON_IM_CONTEXT_DIRECT_MODE);
}
```

Listing 7.127: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*disable) (HildonIMPlugin *plugin);
```

Listing 7.128: hildon-input-method/hildon-im-plugin.h

**disable** is called whenever the plug-in becomes unavailable to the user (e.g. when the main UI is closed).

```
/* Called when the plugin is disabled */
static void
disable(HildonIMPlugin *plugin)
{
 /* not implemented */
}
```

Listing 7.129: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*settings_changed) (HildonIMPlugin *plugin,
                          const gchar *key,
                          const GConfValue *value);
```

Listing 7.130: hildon-input-method/hildon-im-plugin.h

settings_changed is called whenever the HIM main UI receives a notification from GConf about Hildon Input Method settings being changed. The affected settings are all settings residing in **/apps/osso/inputmethod** path. **key** and **value** hold the GConf key and its value respectively.

```
/* Called when the standard input method settings
   has been changed */
static void
settings_changed (HildonIMPlugin *plugin,
                  const gchar *key, const GConfValue *value)
{
    /* not implemented */
}
```

Listing 7.131: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*input_mode_changed) (HildonIMPlugin *plugin);
```

Listing 7.132: hildon-input-method/hildon-im-plugin.h

**input\_mode\_changed** is called whenever the input mode is changed. Input mode is changed to what has been specified by the client widget. The input mode puts constraints to the plug-in to limit whether input shall be accepted or ignored.

```
/* Called when input mode changed */
static void
input_mode_changed (HildonIMPlugin *plugin)
{
 /* not implemented */
}
```

Listing 7.133: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*clear) (HildonIMPlugin *plugin);
```

Listing 7.134: hildon-input-method/hildon-im-plugin.h

clear is called whenever the HIM main UI requests the plug-in to clear or refresh its user interface.

```
/* Called when the plugin is requested to 'clear'/refresh its UI
   */
static void
clear(HildonIMPlugin *plugin)
{
    /* not implemented */
}
```

Listing 7.135: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*client_widget_changed) (HildonIMPlugin *plugin);
```

Listing 7.136: hildon-input-method/hildon-im-plugin.h

**client\_widget\_changed** is called whenever the client widget is changed from one to another. For instance, the case could be that the user taps on another text entry.

```
/* Called when the client widget changed */
static void
client_widget_changed (HildonIMPlugin *plugin)
{
 /* not implemented */
}
```

Listing 7.137: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*save_data) (HildonIMPlugin *plugin);
```

Listing 7.138: hildon-input-method/hildon-im-plugin.h

save_data is called whenever the HIM main UI is requested to save its (and the plug-in's) data. Usually it is called when the main UI is requested to quit.

```
/* Called when the plugin is requested to save its data */
static void
save_data(HildonIMPlugin *plugin)
{
    /* not implemented */
}
```

Listing 7.139: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*mode_a) (HildonIMPlugin *plugin);
```

Listing 7.140: hildon-input-method/hildon-im-plugin.h

**mode\_a** is called whenever the *Mode A* (Caps Lock in virtual keyboard plug-in) is pressed.

```

/* Called when the MODE_A button is pressed */
static void
mode_a(HildonIMPlugin *plugin)
{
 HimExampleVKBPrivate *priv;
 HimExampleVKB *vkb;

 vkb = HIMEXAMPLE_VKB (plugin);

 priv = HIMEXAMPLE_VKB_GET_PRIVATE (vkb);
 if (hildon_im_ui_button_get_active (priv->ui,
 HILDON_IM_BUTTON_MODE_B)) {
 hildon_im_ui_button_set_active (priv->ui,
 HILDON_IM_BUTTON_MODE_B, FALSE);
 if (hildon_im_ui_button_get_active (priv->ui,
 HILDON_IM_BUTTON_MODE_A)) {
 priv->case_mode = CASE_UPPER;
 } else {
 priv->case_mode = CASE_LOWER;
 }
 } else {
 if (hildon_im_ui_button_get_active (priv->ui,
 HILDON_IM_BUTTON_MODE_A)) {
 priv->case_mode = CASE_UPPER;
 } else {
 priv->case_mode = CASE_LOWER;
 }
 }

 update_layout (vkb);
}

```

Listing 7.141: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```

void (*mode_b) (HildonIMPlugin *plugin);

```

Listing 7.142: hildon-input-method/hildon-im-plugin.h

mode_b is called whenever the *Mode B* (Shift in virtual keyboard plug-in) is pressed.

```

/* Called when the MODE_B button is pressed */
static void
mode_b(HildonIMPlugin *plugin)
{
    HimExampleVKBPrivate *priv;
    HimExampleVKB *vkb;

    vkb = HIMEXAMPLE_VKB (plugin);

    priv = HIMEXAMPLE_VKB_GET_PRIVATE (vkb);

    if (hildon_im_ui_button_get_active (priv->ui,
        HILDON_IM_BUTTON_MODE_B)) {
        if (hildon_im_ui_button_get_active (priv->ui,
            HILDON_IM_BUTTON_MODE_A)) {
            priv->case_mode = CASE_LOWER;
        } else {
            priv->case_mode = CASE_UPPER;
        }
    } else {

```



```

        if (hildon_im_ui_button_get_active (priv->ui,
            HILDON_IM_BUTTON_MODE_A)) {
            priv->case_mode = CASE_UPPER;
        } else {
            priv->case_mode = CASE_LOWER;
        }
    }

    update_layout (vkb);
}

```

Listing 7.143: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*backspace) (HildonIMPlugin *plugin);
```

Listing 7.144: hildon-input-method/hildon-im-plugin.h

**backspace** is called whenever the virtual backspace key is pressed.

```

/* Called when the backspace button is pressed */
static void
backspace (HildonIMPlugin *plugin)
{
 HimExampleVKBPrivate *priv;

 priv = HIMEXAMPLE_VKB_GET_PRIVATE (HIMEXAMPLE_VKB (plugin));
 hildon_im_ui_send_communication_message (priv->ui,
 HILDON_IM_CONTEXT_HANDLE_BACKSPACE);
}

```

Listing 7.145: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*enter) (HildonIMPlugin *plugin);
```

Listing 7.146: hildon-input-method/hildon-im-plugin.h

enter is called whenever the virtual enter key is pressed.

```

/* Called when the enter button is pressed */
static void
enter (HildonIMPlugin *plugin)
{
    HimExampleVKBPrivate *priv;

    priv = HIMEXAMPLE_VKB_GET_PRIVATE (HIMEXAMPLE_VKB (plugin));
    hildon_im_ui_send_communication_message (priv->ui,
        HILDON_IM_CONTEXT_HANDLE_ENTER);
}

```

Listing 7.147: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*tab) (HildonIMPlugin *plugin);
```

Listing 7.148: hildon-input-method/hildon-im-plugin.h

**tab** is called whenever the virtual tab key is pressed.

```

/* Called when the tab button is pressed */
static void
tab (HildonIMPlugin *plugin)
{
 HimExampleVKBPrivate *priv;

 priv = HIMEXAMPLE_VKB_GET_PRIVATE (HIMEXAMPLE_VKB (plugin));
 hildon_im_ui_send_communication_message (priv->ui,
 HILDON_IM_CONTEXT_HANDLE_TAB);
}

```

Listing 7.149: hildon-input-method-plugins-example/src/him-vkb-example.c

Couple of functions related to changing language:

- ```
void (*language) (HildonIMPlugin *plugin);
```

Listing 7.150: hildon-input-method/hildon-im-plugin.h

language is called whenever a new language is selected from the HIM main UI menu.

```

/* Called when the language has been changed */
static void
language (HildonIMPlugin *plugin)
{
    /* not implemented */
}

```

Listing 7.151: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
void (*language_settings_changed) (HildonIMPlugin *plugin, gint index);
```

Listing 7.152: hildon-input-method/hildon-im-plugin.h

**language\_settings\_changed** is called whenever the language settings for the specified language index have been changed.

## Plug-in Loading

A nature of a plug-in is that it can be dynamically loaded. In this case also, the HIM plug-in is loaded whenever the user selects it, so in order to support the dynamic loading, the plug-in has to provide the following three specific functions for Hildon Input Method plug-in system:

- ```
void module_init (GTypeModule *module);
```

This function initializes the plug-in as a module, meaning the type of `GTypeInfo` needs to be registered, and the interface and instance information (`GInterfaceInfo`) need to be added to the module.

```

void
module_init(GTypeModule *module)
{
    static const GTypeInfo type_info = {

```

```

    sizeof(HimExampleVKBClass),
    NULL, /* base_init */
    NULL, /* base_finalize */
    (GClassInitFunc) himExample_vkb_class_init,
    NULL, /* class_finalize */
    NULL, /* class_data */
    sizeof(HimExampleVKB),
    0, /* n_preallocs */
    (GInstanceInitFunc) himExample_vkb_init,
};

static const GInterfaceInfo plugin_info = {
    (GInterfaceInitFunc) himExample_vkb_iface_init,
    NULL, /* interface_finalize */
    NULL, /* interface_data */
};

himExample_vkb_type =
    g_type_module_register_type(module,
                                GTK_TYPE_WIDGET, "
                                HimExampleVKB",
                                &type_info,
                                0);

g_type_module_add_interface(module,
                            HIMEXAMPLE_VKB_TYPE,
                            HILDON_IM_TYPE_PLUGIN,
                            &plugin_info);
}

```

Listing 7.153: hildon-input-method-plugins-example/src/him-vkb-example.c

The `himExample_vkb_iface_init` function should register the custom interface functions in `HildonIMPluginIface`:

```

/* Standard GTK stuff */
static void
himExample_vkb_iface_init (HildonIMPluginIface *iface)
{
    iface->enable = enable;
    iface->disable = disable;
    iface->enter = enter;
    iface->tab = tab;
    iface->backspace = backspace;
    iface->clear = clear;
    iface->input_mode_changed = input_mode_changed;
    iface->client_widget_changed = client_widget_changed;
    iface->save_data = save_data;
    iface->language = language;
    iface->mode_a = mode_a;
    iface->mode_b = mode_b;
    iface->language_settings_changed = language_settings_changed;
    iface->settings_changed = settings_changed;

    return;
}

```

Listing 7.154: hildon-input-method-plugins-example/src/him-vkb-example.c

- `void module_exit (void);`

This function defines actions when the module is unloaded from the memory.

```
void
module_exit(void)
{
    /* empty */
}
```

Listing 7.155: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```
HildonIMPlugin* module_create (HildonIMUI *ui);
```

This function creates and returns the plug-in widget to the main UI.

```
HildonIMPlugin*
module_create (HildonIMUI *keyboard)
{
 return HILDON_IM_PLUGIN (himExample_vkb_new (keyboard));
}
```

Listing 7.156: hildon-input-method-plugins-example/src/him-vkb-example.c

## Plug-in Info

Some basic information of the plug-in needs to be provided to the HIM plug-in system, so it can keep up with the kinds of plug-ins that are available in the system. This is performed with two specific functions:

- ```
const HildonIMPluginInfo* hildon_im_plugin_get_info(void);
```

The function creates **HildonIMPluginInfo** struct for providing the required information.

```
/* Input Method plugin information.
 * This structure tells the main UI about this plugin */
const HildonIMPluginInfo *
hildon_im_plugin_get_info(void)
{
    static const HildonIMPluginInfo info =
    {
        "HIM VKB Example",           /* description */
        "himExample_vkb",           /* name */
        "Keyboard (EXAMPLE)",        /* menu title */
        NULL,                        /* gettext domain */
        TRUE,                        /* visible in menu */
        FALSE,                       /* cached */
        HILDON_IM_TYPE_DEFAULT,      /* UI type */
        HILDON_IM_GROUP_LATIN,       /* group */
        HILDON_IM_DEFAULT_PLUGIN_PRIORITY, /* priority */
        NULL,                        /* special character
        plugin */
        "",                          /* help page */
        FALSE,                       /* disable common UI
        buttons */
        HILDON_IM_DEFAULT_HEIGHT,    /* plugin height */
        HILDON_IM_TRIGGER_STYLUS     /* trigger */
    };
}
```

```

    return &info;
}

```

Listing 7.157: hildon-input-method-plugins-example/src/him-vkb-example.c

- ```

gchar** hildon_im_plugin_get_available_languages (gboolean *free);

```

The function returns a NULL terminated array of string, containing language codes supported by the plug-in. Set **free** to **TRUE** if HIM main UI should free the returned value when no longer used.

```

/*
 * This function returns the list of available languages supported
 * by the plugin.
 */
gchar **
hildon_im_plugin_get_available_languages (gboolean *free)
{
 static gchar *list[] = { "en_GB", NULL };
 *free = FALSE;

 return list;
}

```

Listing 7.158: hildon-input-method-plugins-example/src/him-vkb-example.c

### 7.13.3 Interaction with Main User Interface

As mentioned above, the plug-in is placed inside the HIM main UI. This section deals with interacting with it. This is mainly done by calling its functions, as defined in **hildon-im-ui.h**.

#### Handling Input

```

void hildon_im_ui_send_utf8(HildonIMUI *main_ui, const gchar *text);

```

Listing 7.159: hildon-input-method/hildon-im-ui.h

The plug-in can request the main UI to commit a UTF-8 encoded text by calling this function.

```

void hildon_im_ui_send_communication_message(HildonIMUI *main_ui, gint
message);

```

Listing 7.160: hildon-input-method/hildon-im-ui.h

The plug-in can use this function to tell the main UI that **Enter**, **Backspace**, or **Tab** virtual buttons are pressed. Simply call this function and pass one of the constants below as the **message** argument:

- **HILDON\_IM\_CONTEXT\_HANDLE\_ENTER**
- **HILDON\_IM\_CONTEXT\_HANDLE\_BACKSPACE**
- **HILDON\_IM\_CONTEXT\_HANDLE\_TAB**

## UI Visibility

```
void hildon_im_ui_set_visible(HildonIMUI *ui, gboolean visible);
```

Listing 7.161: hildon-input-method/hildon-im-ui.h

The plug-in can request the main UI to set it's visibility by calling this function.

```
gboolean hildon_im_ui_get_visibility(HildonIMUI *main_ui);
```

Listing 7.162: hildon-input-method/hildon-im-ui.h

By calling this function, the plug-in can get the visibility status of the main UI.

## Get Input Method State

The following two state reader functions could be very handy in case the plug-in receives a state change notification (e.g. language change), because this way all the state information does not need to be saved.

```
HildonIMCommand hildon_im_ui_get_autocase_mode(HildonIMUI *main_ui);
```

Listing 7.163: hildon-input-method/hildon-im-ui.h

The function returns the auto-capitalization mode of the current client widget.

```
const gchar * hildon_im_ui_get_active_language(HildonIMUI *main_ui);
```

Listing 7.164: hildon-input-method/hildon-im-ui.h

The function returns the current language code.

## Common Buttons

As mentioned in the *Overview* section, the outlook and even the function of the buttons in the main UI can be modified. **N.B.** It is not recommended to alter buttons, except the *Mode A* and *Mode B* buttons! Other buttons may have hardwired behavior within the main UI.

If the plug-in changes the functionality of a button, one might want to reflect this also in the UI by changing the label of the button. The layout of the buttons can be altered by the following two functions:

```
void hildon_im_ui_button_set_label(HildonIMUI *keyboard,
 HildonIMButton button,
 const gchar *label);
```

Listing 7.165: hildon-input-method/hildon-im-ui.h

With this function, the plug-in can set the label of a button.

Possible values of **HildonIMButton** (see **hildon-im-ui.h**):

- HILDON\_IM\_BUTTON\_TAB
- HILDON\_IM\_BUTTON\_MODE\_A

- HILDON\_IM\_BUTTON\_MODE\_B
- HILDON\_IM\_BUTTON\_INPUT\_MENU
- HILDON\_IM\_BUTTON\_BACKSPACE
- HILDON\_IM\_BUTTON\_ENTER
- HILDON\_IM\_BUTTON\_SPECIAL\_CHAR
- BUTTON\_CLOSE

- ```
void hildon_im_ui_button_set_id(HildonIMUI *self,
                                HildonIMButton button,
                                const gchar *id);
```

Listing 7.166: hildon-input-method/hildon-im-ui.h

This function sets a name to a particular button.

Since every input to the HIM main UI is caught by the plug-in, it is necessary to keep the button state (active or in-active) in sync. The state of a particular button can be changed, queried and toggled with the following functions:

- ```
void hildon_im_ui_button_set_active(HildonIMUI *keyboard,
 HildonIMButton button,
 gboolean active);
```

Listing 7.167: hildon-input-method/hildon-im-ui.h

This function sets the active state of a particular button.

- ```
gboolean hildon_im_ui_button_get_active(HildonIMUI *keyboard,
                                         HildonIMButton button);
```

Listing 7.168: hildon-input-method/hildon-im-ui.h

This function returns the active state of a particular button.

- ```
void hildon_im_ui_button_set_toggle(HildonIMUI *keyboard,
 HildonIMButton button,
 gboolean toggle);
```

Listing 7.169: hildon-input-method/hildon-im-ui.h

The plug-in can set the toggle state of a particular button with this function.

Miscellaneous button manipulation functions:

- ```
void hildon_im_ui_button_set_menu(HildonIMUI *keyboard,
                                   HildonIMButton button,
                                   GtkWidget *menu);
```

Listing 7.170: hildon-input-method/hildon-im-ui.h

With this function, the plug-in can attach a menu - which is a GtkWidget - to a particular button.

```
void hildon_im_ui_button_set_sensitive(HildonIMUI *keyboard,
                                      HildonIMButton button,
                                      gboolean sensitive);
```

Listing 7.171: hildon-input-method/hildon-im-ui.h

All the buttons defined on the HIM main UI may not be needed, or the functionality of a button may be wished to be switched off in some states. In this case, the sensitivity of a particular button can be set by calling this function.

```
void hildon_im_ui_button_set_repeat(HildonIMUI *keyboard,
                                    HildonIMButton button,
                                    gboolean repeat);
```

Listing 7.172: hildon-input-method/hildon-im-ui.h

This function controls whether a particular button will repeat when pressed for a long time.

7.13.4 Component Dependencies

At least the following headers shall be included in the plug-in:

```
#include <hildon-im-plugin.h>
#include <hildon-im-ui.h>
```

hildon-input-method-framework-dev and **libhildon-im-ui-dev** packages.

7.13.5 Language Codes

These are the language codes recognized; they are numbered, the first, `af_ZA`, being 0.

```
af_ZA am_ET ar_AE ar_BH ar_DZ ar_EG ar_IN ar_IQ ar_JO ar_KW
ar_LB ar_LY ar_MA ar_OM ar_QA ar_SA ar_SD ar_SY ar_TN ar_YE
az_AZ be_BY bg_BG bn_IN br_FR bs_BA ca_ES cs_CZ cy_GB da_DK
de_AT de_BE de_CH de_DE de_LU el_GR en_AU en_BW en_CA en_DK
en_GB en_HK en_IE en_IN en_NZ en_PH en_SG en_US en_ZA en_ZW
eo_EO es_AR es_BO es_CL es_CO es_CR es_DO es_EC es_ES es_GT
es_HN es_MX es_NI es_PA es_PE es_PR es_PY es_SV es_US es_UY
es_VE et_EE eu_ES fa_IR fi_FI fo_FO fr_BE fr_CA fr_CH fr_FR
fr_LU ga_IE gd_GB gl_ES gv_GB he_IL hi_IN hr_HR hu_HU hy_AM
id_ID is_IS it_CH it_IT iw_IL ja_JP ka_GE kl_GL ko_KR kw_GB
lt_LT lv_LV mi_NZ mk_MK mr_IN ms_MY mt_MT nl_BE nl_NL nn_NO
no_NO oc_FR pl_PL pt_BR pt_PT ro_RO ru_RU ru-UA se_NO sk_SK
sl_SI sq_AL sr_YU sv_FI sv_SE ta_IN te_IN tg_TJ th_TH ti_ER
ti_ET tl_PH tr_TR tt_RU uk-UA ur_PK uz_UZ vi_VN wa_BE yi_US
zh_CN zh_HK zh_SG zh_TW
```


Chapter 8

Using Multimedia Components

8.1 Introduction

The following code examples are used in this chapter:

- [example_wavlaunch.c](#)
- [example_camera.c](#)
- [crazyparking](#)

Maemo offers lots of possibilities for multimedia applications. Maemo was built with the Internet Tablet devices in mind - hardware in this form factor provides a big high resolution touch screen, audio input and output, video and image capturing through the camera, fast network connections for streaming and a digital signal processor for efficient audio and video manipulation.

For developers, there are open programming interfaces to make use of these features apart from directly programming the DSP. The preferred way of doing video and audio programming for maemo is using the GStreamer framework. For manipulating images there are several libraries. Interactive multimedia for applications like games can be done using SDL framework.

GStreamer - Multimedia Framework

The main multimedia framework in maemo is GStreamer. It is based on the concept of pipelines, which are made of multiple elements. Elements themselves can be practically anything that do something with a data stream.

Maemo includes a variety of these elements to support audio and video effects, encoding and decoding and interfacing with the device's hardware. Using this approach, a developer does not have to bother with details like low level hardware management, audio and video compression schemes etc. If the elements included with the OS are not enough, more can be either compiled for maemo, or new ones can be developed.

Developer documentation for GStreamer can be found at the project's website[33].

Stream Encoding and Decoding

Maemo devices include a wide arsenal of video and audio codecs, which enable the maemo applications to read and write nearly all commonly used video and audio formats. These are supported also by the GStreamer framework. For many purposes the developer does not even have to specify the used codecs implicitly, since GStreamer automatically detects the format and the codec to use. The `playbin`[\[35\]](#) GStreamer base plug-in provides a convenient abstraction layer for all audio and video content.

Due to non-technical reasons, most of the codecs are not distributed with the SDK. This is good to keep in mind, when developing applications relying on such features.

Digital Signal Processor

Inside Internet Tablets, there is a dedicated digital signal processor (DSP). Its design is optimized for tasks like stream coding and audio effects. Maemo has a high level programming support for the DSP in form of GStreamer elements, which can decode most of the supported file formats. By using the DSP, the computing load is also taken off from the main processor, greatly enhancing system performance and responsivity.

Audio

For audio programming, maemo has two main APIs: GStreamer and ESound. Usually system sounds, such as sounds for notifying user of an event, e.g. battery low, is played through ESound. More sophisticated operations, e.g. playing music files or recording audio, should be generally performed using GStreamer, which provides better performance and a much more flexible API. Most of maemo's computing intensive GStreamer elements are implemented using the device's DSP, which greatly enhances their performance.

Linux kernel has also two lower level audio interfaces: ALSA and OSS. Of these, ALSA is supported through a plug-in package that is part of the SDK. The legacy API OSS is not supported by the kernel, but ALSA has an OSS emulation system that works for most purposes.

For the audio APIs' documentation, see the GStreamer web site[\[33\]](#), ESound white paper[\[15\]](#) and ALSA project's web site[\[2\]](#).

Video

Although the framework hides much of the implementation and hardware details, it is good for a developer to know what happens beneath the interfaces. Video capturing is performed via Linux kernel's Video4Linux API, and graphics are displayed using the X Window System. Practically all GNU/Linux applications rely on these components for video tasks, so porting existing applications should be quite effortless, and support easy to find.

Hands-on instructions for using capture and output features are given in section [8.3](#).

Graphics and Images

Maemo includes several libraries for working with images and graphics:

- Cairo is a library for making 2D vector graphics. It features, for example, high quality vector-based graphics, support for importing and exporting

a variety of formats, such as SVG, PNG, PDF and Postscript, and bindings for many programming languages

- GDK and GdkPixbuf are the bitmap graphics libraries that GTK+ is built on. Together, they provide opening and saving for bitmap images in JPEG, PNG, TIFF, ICO and BMP formats, tight integration into GTK+ and tools for drawing with graphic primitives. Also loading of SVG images is supported
- For more detailed control over the different image formats, there are many more specialized libraries, e.g. libpng, libjpeg and libwmf.

Visit Cairo website[6] for guides, tutorials and references. GDK's and Gd-kPixbuf's documentation can be found at GTK+ project's website[38].

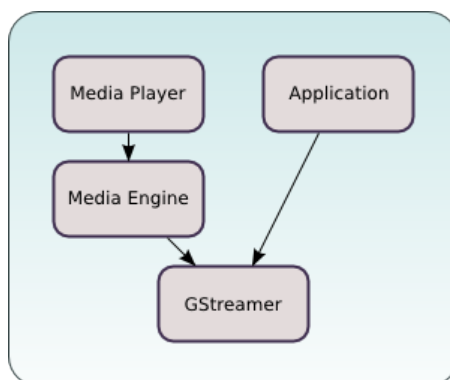
Games

For game developers, maemo offers a common start-up and settings screen, and a framework called *osso-games-startup*. Using this library, game developers and porters can reduce the common start-up screen coding, and concentrate on the real game programming. Osso-games-startup's operating system communication features also offer an interface to make games behave correctly. It eases, for example, the management of full screen mode and exceptional situations, such as battery low notification, in a unified and user friendly manner. See section 8.4 for more in-depth description and API usage.

8.2 Getting Started with Multimedia

This section explains how to get started developing multimedia applications and plug-ins in the maemo SDK. It is recommended to read also [GStreamer's documentation](#).

Here is a diagram of the multimedia architecture from the device's Media Player point of view. It is important, if planning to develop GStreamer plug-ins that can be used by the Media Player.



8.2.1 Simple GStreamer Example

This example demonstrates how to use GStreamer's API to play PCM audio.

First, get the [sample applications](#) (including build scripts), for instance with the following command:

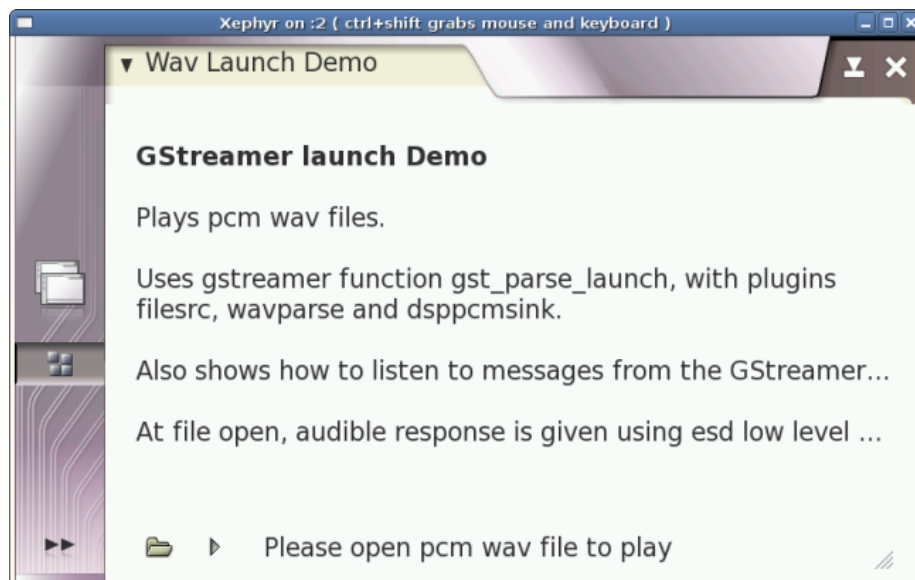
```
svn checkout https://garage.maemo.org/svn/maemoexamples/tags/maemo_4.1/maemo-examples/
```

To compile and run, start Xephyr and execute the following commands:

```
cd maemo-examples
make
af-sb-init.sh start
run-standalone.sh ./example_wavlaunch
```

To use, simply:

- Click Open to open the File Chooser.
- Browse to a wav file and select Open.
- Select Play to start playing.



Many aspects of GStreamer's application development are described in the [GStreamer Application Development Manual](#) [34].

For more information on how to use GStreamer's API, see the [Core API Reference](#) [37].

8.2.2 Plug-in Development

In order to have support for a new media format in the maemo SDK, it is necessary to compile a codec for the format, and compile the GStreamer plug-in supporting that codec.

Codecs are libraries that enable the use of compression for digital audio and video. GStreamer plug-ins are loadable libraries, which provide GStreamer elements that process video and audio streams.

Some plug-ins have the codec embedded, and therefore do not need an additional library. To play some unusual formats, a demuxer GStreamer plug-in

is needed as well. More information about the internals of GStreamer plug-ins can be found in the [Plugin Writers Guide](#) [36].

The list of plug-ins for GStreamer is available [here](#).

To add support for a new media format:

- Get the codec from the codec's manufacturer.
- Get the GStreamer plug-ins from the [GStreamer website](#).
- Extract packages to the Scratchbox environment and follow the compiling instructions for the codec and the GStreamer plug-ins package

8.2.3 Installing OGG Vorbis

To enable playback for ogg-vorbis audio files on the platform, take the following steps:

1. Get the integer-only implementation from [here](#).

```
svn co http://svn.xiph.org/trunk/Tremor/
```

2. Build and install in Scratchbox.

```
./autogen.sh --prefix=/usr  
make install
```

3. Get [gst-plugins-bad-0.10.5](#).

4. Build and install in Scratchbox:

```
./configure --prefix=/usr  
make -C ext/ivorbis install
```

5. Check that it is there (if you have gstreamer-tools):

```
gst-inspect-0.10 tremor
```

6. If you want to try it:

```
gst-launch-0.10 filesrc location=test.ogg ! application/ogg ! tremor ! alsasink
```

8.2.4 Deployment

Copy files from scratchbox to the device (into the same directory):

- `/usr/lib/libvorbisidec.so* -> /usr/lib/`
- `/usr/lib/gstreamer-0.10/libgstivorbis.so -> /usr/lib/gstreamer-0.10`
- If you want to try it:

```
gst-launch-0.10 filesrc location=test.ogg ! application/ogg ! tremor ! dsppcmsink
```

The next step is to tell the file manager and the media player about the new format.

1. Edit `/usr/share/mime/packages/ogg-vorbis.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<mime-info xmlns="http://www.freedesktop.org/standards/shared-mime
-info">
  <mime-type type="audio/x-vorbis">
    <glob pattern="*.ogg"/>
    <magic priority="50">
      <match type="string" value="OggS" offset="0"/>
    </magic>
    <comment>OGG Vorbis audio</comment>
  </mime-type>
</mime-info>
```

N.B. The mime type needs to start with "audio/" to be properly recognized. The same applies to video codecs ("video/").

Update the MIME database:

```
update-mime-database /usr/share/mime
```

2. Add "audio/x-vorbis" to /usr/share/applications/hildon/mp_ui.desktop

Update the Desktop database:

```
update-desktop-database
```

3. Add ogg to the libmetallayer configuration
/usr/share/libmetallayer/metadata_lib.conf

```
ogg libmtext_gst
```

Reload metallayer crawler

```
/etc/init.d/metallayer-crawler0 restart
```



N.B.

This information may contain references to third-party information, software and/or services. Such third-party references in this material do not imply endorsement by Nokia of the third party in any way. Your use of third party software shall be governed by and subject to you agreeing to the terms of separate software licenses and Nokia shall not be responsible or liable for any part of such dealings.

8.3 Camera API Usage

This section explains how to use the Camera API to access the camera hardware that is present in some models of Nokia Internet Tablets.

8.3.1 Camera Hardware and Linux

The Linux operating system supports live video and audio hardware, such as webcams, TV tuners, video capture cards, FM radio tuners, video *output* devices etc. The primary API for the applications to access those devices is [Video4Linux](#).

Video4Linux is a *kernel* API, so there must be kernel drivers for each supported device. At the user level, device access is standardized via device files. In the case of video capture devices like cameras, which are the focus of this material, the files would be `/dev/video0`, `/dev/video1`, etc., as many as there are connected devices.

Data exchanged between the device file and user-level application has a standardized format for each device class. This allows the application to be instantly compatible with every video capture device that has a driver for Linux.

The built-in camera present in some Nokia Internet Tablet devices is compatible with [Video-4-Linux version 2 API](#) [101]. In principle, any application compatible with this API is easily portable to the maemo platform.

Since the maemo platform delegates all multimedia handling to the GStreamer framework, applications that need access to the built-in camera should employ GStreamer for this, instead of directly accessing Video4Linux devices, via the `v4l2src` GStreamer module.

Thanks to the flexibility of GStreamer, a developer can fully test any given application in a regular desktop PC with a connected webcam, and then perform the final test in the Internet Tablet itself, without a single change in the source code, since GStreamer refers to modules as text names.

One important note about the camera in the Tablet: only one application can use it at any given time. So, while using the camera in an application, other tasks that could possibly make use of it (e.g. a video call) will be blocked.

To demonstrate how the camera manipulation is performed, an example application is provided and discussed.

8.3.2 Camera Manipulation in C Language

This C application allows to use the Internet Tablet as a "mirror" (i.e. showing the camera input in the screen), as well as allows to take pictures to be saved as JPEG files (which illustrates the video frame buffer manipulation).

In this example, the function `initialize_pipeline()` is most interesting, since it is responsible for creating the GStreamer pipeline, sourcing data from Video4Linux and sinking it to a `xvimagesink` (which is an optimized X framebuffer). The pipeline scheme is as follows:

```
/* Initialize the the Gstreamer pipeline. Below is a diagram
 * of the pipeline that will be created:
 *
 *                               |Screen| |Screen|
 *                               ->|queue|->|sink|-> Display
 * |Camera| |CSP| |Tee|/
 * |src|->|Filter|->| \ |Image| |Image| |Image|
 *                               ->|queue|-> |filter|->|sink|-> JPEG file
 */
```

Listing 8.1: `example_camera.c`

Between the source and sinks, there are two `ffmpegcolorspace` filters, one to configure the camera frame rate and the picture size expected by the JPEG encoder, and the second to satisfy the video sink. Capabilities ("caps") are employed to tell which format the data needs to have when exiting the filter.

The second filter is necessary, since the video sink may have different requirements (bit depth, color space) from the JPEG encoder. Those requirements can vary also according to the hardware.

Because there are two sinks, the queues are important, since they guarantee that each pipeline segment operates on its own thread downstream the queue. This ensures that the different sinks can synchronize without waiting for each other.

This sample application is not different from other GStreamer applications, be it Linux-generic or maemo-specific apps:

```
static gboolean initialize_pipeline(AppData *appdata,
    int *argc, char ***argv)
{
    GstElement *pipeline, *camera_src, *screen_sink, *image_sink;
    GstElement *screen_queue, *image_queue;
    GstElement *csp_filter, *image_filter, *tee;
    GstCaps *caps;
    GstBus *bus;

    /* Initialize Gstreamer */
    gst_init(argc, argv);

    /* Create pipeline and attach a callback to it's
     * message bus */
    pipeline = gst_pipeline_new("test-camera");

    bus = gst_pipeline_get_bus(GST_PIPELINE(pipeline));
    gst_bus_add_watch(bus, (GstBusFunc)bus_callback, appdata);
    gst_object_unref(GST_OBJECT(bus));

    /* Save pipeline to the AppData structure */
    appdata->pipeline = pipeline;

    /* Create elements */
    /* Camera video stream comes from a Video4Linux driver */
    camera_src = gst_element_factory_make(VIDEO_SRC, "camera_src");
    /* Colorspace filter is needed to make sure that sinks understands
     * the stream coming from the camera */
    csp_filter = gst_element_factory_make("ffmpegcolorspace", "csp_filter");
    /* Tee that copies the stream to multiple outputs */
    tee = gst_element_factory_make("tee", "tee");
    /* Queue creates new thread for the stream */
    screen_queue = gst_element_factory_make("queue", "screen_queue");
    /* Sink that shows the image on screen. Xephyr doesn't support XVideo
     * extension, so it needs to use ximagesink, but the device uses
     * xvimagesink */
    screen_sink = gst_element_factory_make(VIDEO_SINK, "screen_sink");
    /* Creates separate thread for the stream from which the image
     * is captured */
    image_queue = gst_element_factory_make("queue", "image_queue");
    /* Filter to convert stream to use format that the gdkpixbuf library
     * can use */
    image_filter = gst_element_factory_make("ffmpegcolorspace", "image_filter");
    /* A dummy sink for the image stream. Goes to bitheaven */
    image_sink = gst_element_factory_make("fakesink", "image_sink");

    /* Check that elements are correctly initialized */
    if(!(pipeline && camera_src && screen_sink && csp_filter &&
```



```

        screen_queue
        && image_queue && image_filter && image_sink))
{
    g_critical("Couldn't create pipeline elements");
    return FALSE;
}

/* Set image sink to emit handoff-signal before throwing away
 * it's buffer */
gst_object_set(G_OBJECT(image_sink),
    "signal-handoffs", TRUE, NULL);

/* Add elements to the pipeline. This has to be done prior to
 * linking them */
gst_bin_add_many(GST_BIN(pipeline), camera_src, csp_filter,
    tee, screen_queue, screen_sink, image_queue,
    image_filter, image_sink, NULL);

/* Specify what kind of video is wanted from the camera */
caps = gst_caps_new_simple("video/x-raw-rgb",
    "width", G_TYPE_INT, 640,
    "height", G_TYPE_INT, 480,
    NULL);

/* Link the camera source and colorspace filter using capabilities
 * specified */
if(!gst_element_link_filtered(camera_src, csp_filter, caps))
{
    return FALSE;
}
gst_caps_unref(caps);

/* Connect Colorspace Filter -> Tee -> Screen Queue -> Screen Sink
 * This finalizes the initialization of the screen-part of the
 * pipeline */
if(!gst_element_link_many(csp_filter, tee, screen_queue, screen_sink,
    NULL))
{
    return FALSE;
}

/* gdkpixbuf requires 8 bits per sample which is 24 bits per
 * pixel */
caps = gst_caps_new_simple("video/x-raw-rgb",
    "width", G_TYPE_INT, 640,
    "height", G_TYPE_INT, 480,
    "bpp", G_TYPE_INT, 24,
    "depth", G_TYPE_INT, 24,
    "framerate", GST_TYPE_FRACTION, 15, 1,
    NULL);

/* Link the image-branch of the pipeline. The pipeline is
 * ready after this */
if(!gst_element_link_many(tee, image_queue, image_filter, NULL))
    return FALSE;
if(!gst_element_link_filtered(image_filter, image_sink, caps)) return
    FALSE;

gst_caps_unref(caps);

/* As soon as screen is exposed, window ID will be advised to the

```

```

        sink */
g_signal_connect(appdata->screen, "expose-event", G_CALLBACK(
    expose_cb),
    screen_sink);

gst_element_set_state(pipeline, GST_STATE_PLAYING);

return TRUE;
}

```

Listing 8.2: example_camera.c

The following function is called back when the user has pressed the "Take photo" button, and the image sink has data. It will forward the image buffer to `create_jpeg()`:

```

/* This callback will be registered to the image sink
 * after user requests a photo */
static gboolean buffer_probe_callback(
    GstElement *image_sink,
    GstBuffer *buffer, GstPad *pad, AppData *appdata)
{
    GstMessage *message;
    gchar *message_name;
    /* This is the raw RGB-data that image sink is about
     * to discard */
    unsigned char *data_photo =
        (unsigned char *) GST_BUFFER_DATA(buffer);

    /* Create a JPEG of the data and check the status */
    if(!create_jpeg(data_photo))
        message_name = "photo-failed";
    else
        message_name = "photo-taken";

    /* Disconnect the handler so no more photos
     * are taken */
    g_signal_handler_disconnect(G_OBJECT(image_sink),
        appdata->buffer_cb_id);

    /* Create and send an application message which will be
     * caught in the bus watcher function. This has to be
     * sent as a message because this callback is called in
     * a gstreamer thread and calling GUI-functions here would
     * lead to X-server synchronization problems */
    message = gst_message_new_application(GST_OBJECT(appdata->pipeline),
        gst_structure_new(message_name, NULL));
    gst_element_post_message(appdata->pipeline, message);

    /* Returning TRUE means that the buffer can is OK to be
     * sent forward. When using fakesink this doesn't really
     * matter because the data is discarded anyway */
    return TRUE;
}

```

Listing 8.3: example_camera.c

The `xvimagesink` GStreamer module will normally create a new window just for itself. Since the video is supposed to be shown inside the main application window, the X-Window window ID needs to be passed to the module, as soon as the ID exists:

```

/* Callback to be called when the screen-widget is exposed */
static gboolean expose_cb(GtkWidget * widget, GdkEventExpose * event,
    gpointer data)
{
    /* Tell the xvimagesink/ximagesink the x-window-id of the screen
     * widget in which the video is shown. After this the video
     * is shown in the correct widget */
    gst_x_overlay_set_xwindow_id(GST_X_OVERLAY(data),
        GDK_WINDOW_XWINDOW(widget->window));
    return FALSE;
}

```

Listing 8.4: example_camera.c

For the sake of completeness, it follows the JPEG encoding function. It is worthwhile to mention that the buffer that came from GStreamer is a simple linear framebuffer:

```

/* Creates a jpeg file from the buffer's raw image data */
static gboolean create_jpeg(unsigned char *data)
{
    GdkPixbuf *pixbuf = NULL;
    GError *error = NULL;
    guint height, width, bpp;
    const gchar *directory;
    GString *filename;
    guint base_len, i;
    struct stat statbuf;

    width = 640; height = 480; bpp = 24;

    /* Define the save folder */
    directory = SAVE_FOLDER_DEFAULT;
    if(directory == NULL)
    {
        directory = g_get_tmp_dir();
    }

    /* Create an unique file name */
    filename = g_string_new(g_build_filename(directory,
        PHOTO_NAME_DEFAULT, NULL));
    base_len = filename->len;
    g_string_append(filename, PHOTO_NAME_SUFFIX_DEFAULT);
    for(i = 1; !stat(filename->str, &statbuf); ++i)
    {
        g_string_truncate(filename, base_len);
        g_string_append_printf(filename, "%d%s", i,
            PHOTO_NAME_SUFFIX_DEFAULT);
    }

    /* Create a pixbuf object from the data */
    pixbuf = gdk_pixbuf_new_from_data(data,
        GDK_COLORSPACE_RGB, /* RGB-colorspace */
        FALSE, /* No alpha-channel */
        bpp/3, /* Bits per RGB-component */
        width, height, /* Dimensions */
        3*width, /* Number of bytes between lines (ie stride) */
        NULL, NULL); /* Callbacks */
}

```

```

/* Save the pixbuf content's in to a jpeg file and check for
 * errors */
if(!gdk_pixbuf_save(pixbuf, filename->str, "jpeg", &error, NULL))
{
    g_warning("%s\n", error->message);
    g_error_free(error);
    gdk_pixbuf_unref(pixbuf);
    g_string_free(filename, TRUE);
    return FALSE;
}

/* Free allocated resources and return TRUE which means
 * that the operation was succesful */
g_string_free(filename, TRUE);
gdk_pixbuf_unref(pixbuf);
return TRUE;
}

```

Listing 8.5: example_camera.c

8.4 Using Games Start-up Screen

The osso-games-startup application is a generic game start-up interface, providing a common hildonized user interface view for game start-up control and configuration.

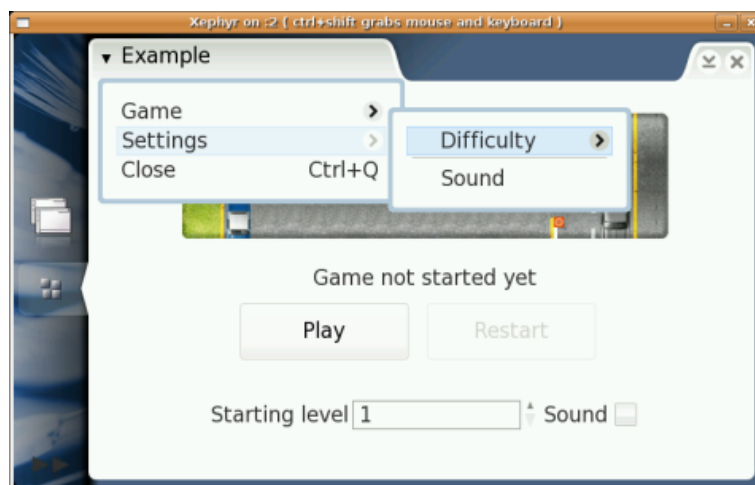


Figure 8.1: Basic osso-games-startup application screen

N.B. The older maemo-games-startup has been replaced by osso-games-startup that is covered in this material.

8.4.1 Application Functionality

To execute the game, the osso-games-startup application should be called. The game configuration file should be passed as an argument. Once loaded, osso-games-startup will create a common interface for all games (see figure 8.1)

and, if needed, will load a specific plug-in for each game. Games are activated through an auto-activating D-BUS message, which tells the game either to start, restart or to continue. In cases where no clean-up routine within the plug-in exists, it can also start the game to clean its state data. In these cases, the game usually does not open its own window, but kills the state data in the background instead.

When the Play button is pressed, the D-BUS service defined in the configuration file will be executed. Games that do not use the glib mainloop must integrate some functionality to their mainloop (for more information about games without a glib mainloop see section 8.4.2).

The service is called every time any communication between osso-games-startup and the game is needed.

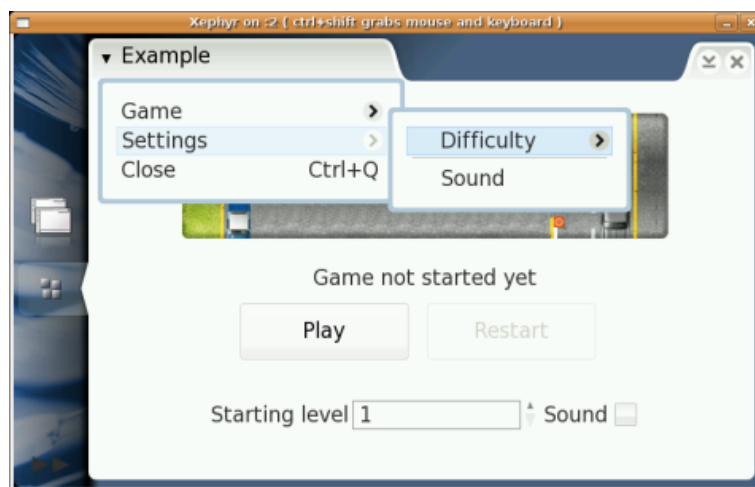


Figure 8.2: The Games Start-up screen with a simple plug-in containing difficulty level and sound configuration

Diagram 8.3 illustrates the operational execution flow of the osso-games-startup application.

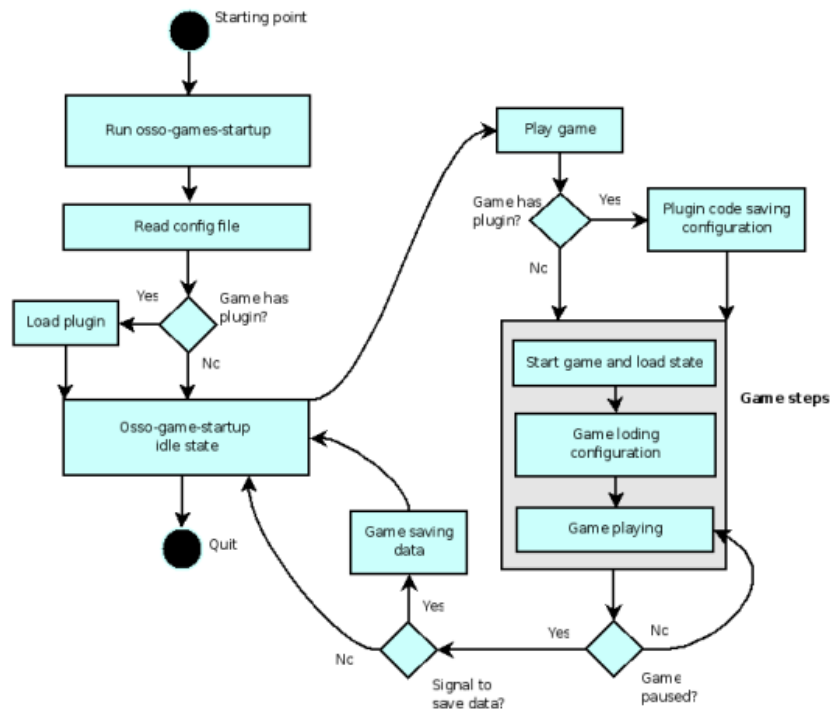


Figure 8.3: Activity Diagram

8.4.2 Integration

The integration tasks depend on the toolkit the game is based on. The following sections describe the integration tasks for:

- Games based on GTK+
- Games based on other toolkits, such as SDL

Integrating GTK+ Games

To integrate games based on GTK+, a configuration file should be created, defining the information necessary for the osso-games-startup application. The following example illustrates the contents of the configuration file:

```
[Startup Entry]
# the name of the application
Name=example
# the current version of the application
Version=0.2.0
# the title that will be used, if not specified by the GettextPackage
Title=Example
# the GettextPackage to be used to locate the title string
GettextPackage=example
# if the TitleId is defined, it will search for it inside the gettext package
TitleId=example_title
# if the game has a plug-in
PluginPath=/usr/share/example/example_plugin.so
# the games-startup screen image
Image=/usr/share/example/pixmaps/example.png
# the D-BUS service, path and interface names
ServiceName=com.domain.example
PathName=/com/domain/example
InterfaceName=com.domain.example
```

The game should be integrated with libosso to enable it to receive and send D-BUS messages, such as "pause", "restart" and "continue". In order to receive messages from the osso-games-startup application, the game must register the service as defined in the configuration file.

When sending messages to osso-games-startup, the game must use the service, path and interface name defined in the configuration file, but with a "_startup" suffix.

Then a D-BUS service (a .service file) should be created, using the name of the service that executes the game binary, as defined in the configuration file. The following example illustrates the contents of the file:

```
[D-BUS Service]
Name=com.domain.example
Exec=/usr/games/wrapper/games/wrapper.1
```

Finally, the game must create a shell script that calls the osso-games-startup application executable, and passes the game configuration file as an argument. The following example illustrates the contents of the script:

```
/usr/bin/osso_games_startup /usr/share/example/example.conf
```

Integrating Non-GTK+ Games

In order to integrate non-GTK+ applications to osso-games-startup, the only thing different is the actual game implementation. The same steps should be followed to create osso desktop files and registering dbus services as for GTK+ based games; the only difference is that Nokia is providing a hildon-games-wrapper library to deal with dbus messaging without the use of libosso.

To initialize game application to receive events from osso-games-startup, the developer can use following approach:

```
#include <hgw/hgw.h>
/* snip */
HgwContext *hgw;
hgw = hgw_context_init();
```

If and when hgw pointer is not null, the game is ready to receive dbus messages that are intended for the game application. The next step is to check, what state the game was started in. This happens by calling `hgw_context_get_start_command()`. The return values indicate the state the game should be going into. This is

necessary, for example, when the game has been paused previously, the state of the game has been saved and the player wishes to continue the current game or restart the on-going game. Again, check the header file for HgwStartCommand for possible start values.

In the game's event loop, the user has to check if events are available by calling `hgw_msg_check_incoming()`. It will return `HGW_ERR_COMMUNICATION` if there are messages available, and other values if there are not or if there was error. (see `hgw/hgw.h` for more details about `HgwError` enum) Example:

```
/* in game event polling */
HgwMessage *msg;
HgwMessageFlags flags;
if ( hgw_msg_check_incoming(hgw,msg,flags) == HGW_ERR_COMMUNICATION ) {
    /* Message Incoming, process msg */
}
```

Pointer `*msg` will then hold information about the received dbus message and the game application should then be able to process and act accordingly. Actual information about the received events is stored in `msg->e_val` which is an enumeration mapped against `HgwCallback`.

When the game ends or the game is requested to pause or quit, a call to `hgw_context_destroy()` should be made with initialized `hgw` as a parameter.

Also, Hildon-games-wrapper library provides an interface to `gconf`. `Gconf` should be used to store persistent data like the level of difficulty, sounds on/off etc. This way is preferred, because the data stored in `gconf` is automatically backed up and restored, when the user so chooses (from desktop, back-up manager). However, this interface is read-only, as writing should be happening in the game-specific plug-in in the `osso-games-startup` screen.

8.4.3 Creating Game-Specific Plug-in

Each game can create a plug-in for specific settings, such as sound control or difficulty level (see figure 8.2). Basically, each plug-in must implement some pre-defined functions that are executed by the `osso-games-startup` application.

The functions that can be implemented by each plug-in are:

- static `GtkWidget *load_plugin (void)` This function creates the game-specific plug-in.
- static void `unload_plugin (void)` This function destroys global variables, if necessary.
- static void `write_config (void)` This function saves the game configuration chosen by the player using the plug-in options.

If the game needs a submenu in the `osso-games-startup` screen main menu (see figure 8.1), the following functions must be used:

- static `GtkWidget **load_menu (guint *)` This function creates the game-specific submenu that is added to the `osso-games-startup` main menu.
- static void `update_menu (void)` This function updates the game-specific menu.

The struct below must be filled and sent to the osso-games-startup application. It specifies which plug-in functions the start-up must call.

```
static StartupPluginInfo plugin_info = {
    GtkWidget * (* load)      (void);
    void (* unload)          (void);
    void (* write_config)     (void);
    GtkWidget ** (* load_menu) (guint *);
    void (* update_menu)     (void);
    void (* plugin_cb)       (GtkWidget *menu_item, gpointer cb_data)
};
```

The last item on the struct is necessary only if the game plug-in requires items such as "save", "save as" or "open" to be a submenu in the default Game submenu.

Each plug-in must contain a reference to the osso-games-startup info. The reference is given when STARTUP_INIT_PLUGIN is called. The following example illustrates a reference to osso-games-startup:

```
static GameStartupInfo gs;
```

The following example illustrates the STARTUP_INIT_PLUGIN that initializes the plug-in. The parameters, in the order they are shown, are:

- Pointer for plug-in information (see the struct shown above)
- GameStartupInfo
- Definition on whether the osso-games-startup should send a D-BUS message on closing
- Definition on whether the osso-games-startup menu has open/save game options

```
STARTUP_INIT_PLUGIN(StartupPluginInfo, GameStartupInfo, gboolean,
    gboolean)
```

In order to inform osso-games-startup that the game has a plug-in, the .conf configuration file of the game must include the PluginPath entry, such as PluginPath=datadir/game01/game01_plugin.so.

Building Example Plug-in for CrazyParking

This section illustrates the plug-in for CrazyParking implementation. The plug-in allows the player to set the initial level of the game, and to define whether the game uses sounds.

Games Start-up screen with CrazyParking plug-in's level and sound configuration:



The following code examples illustrate how the Games Start-up screen works, but the best way to learn to use it is to tinker with the game itself. The source code can be used as wished: as a basic skeleton for the game, or simply to gain a better understanding of the game start-up.

Since the plug-in and osso-games-startup are written with GTK+-2.0, they must include `startup_plugin.h` from `osso-games-startup`. In addition, `Gconf` is used to save the user settings.

```
#include <stdio.h>
#include <gtk/gtk.h>
#include <startup_plugin.h>
#include <gconf/gconf.h>
#include <gconf/gconf-client.h>

#define MENU_SOUND 15
```

Listing 8.6: `crazyparking/src/plugin/plugin.c`

The following example illustrates the labels for retrieving information at `GConf`:

```
#define SETTINGS_LEVEL "/apps/osso/crazyparking/level"
#define SETTINGS_SOUND "/apps/osso/crazyparking/sound"
```

Listing 8.7: `crazyparking/src/plugin/plugin.c`

The following example illustrates the functions that are implemented:

```
static GtkWidget *load_plugin      (void);
static void      unload_plugin    (void);
static void      write_config     (void);
static GtkWidget **load_menu      (guint *);
static void      update_menu      (void);
static void      plugin_callback  (GtkWidget *menu_item, gpointer
data);
static void      settings_callback (GtkWidget *widget, gpointer
data);
static void      sound_callback   (GtkWidget *widget, gpointer
data);
```

The following example illustrates some global variables:

```

GConfClient *gcc = NULL;
GtkWidget *board_box;
GtkWidget *level_1_item;
GtkWidget *level_2_item;
GtkWidget *level_3_item;
GtkWidget *level_4_item;
GtkWidget *level_5_item;
GtkWidget *level_6_item;
GtkWidget *level_7_item;
GtkWidget *level_8_item;
GtkWidget *level_9_item;
GtkWidget *level_10_item;
GtkWidget *level_11_item;
GtkWidget *level_12_item;
GtkWidget *settings_item;
GtkWidget *settings_menu;
GtkWidget *difficulty_item;
GtkWidget *difficulty_menu;
static GameStartupInfo gs;
GtkWidget *menu_items[2];
static int changed = FALSE;
GSList *group = NULL;
GtkWidget *sound_check = NULL;
GtkWidget *sound_item;

```

Listing 8.8: crazyparking/src/plugin/plugin.c

The implemented functions of the plug-in must be sent to osso-games-startup: in this case, a GTK_SPIN_BUTTON and a GTK_CHECK_ITEM. If the plug-in has no specific menu, load_menu and update_menu must be NULL.

```

static StartupPluginInfo plugin_info = {
    load_plugin,
    unload_plugin,
    write_config,
    load_menu,
    update_menu,
    NULL
};

```

Listing 8.9: crazyparking/src/plugin/plugin.c

The following example illustrates the initializing plug-in that informs the application that there is a plug-in:

```

STARTUP_INIT_PLUGIN(plugin_info, gs, FALSE, FALSE);

```

Listing 8.10: crazyparking/src/plugin/plugin.c

The following example illustrates the function that initializes the widgets that localize the osso-games-startup standard buttons:

```

static GtkWidget *load_plugin (void)
{
    int board, enable_sound;
    GtkWidget *game_hbox;
    GtkWidget *board_hbox, *board_label;
    GtkWidget *sound_hbox, *sound_label;

    g_type_init();
    gcc = gconf_client_get_default();

```

```

board = gconf_client_get_int(gcc, SETTINGS_LEVEL, NULL);
enable_sound = gconf_client_get_int(gcc, SETTINGS_SOUND, NULL);

game_hbox = gtk_hbox_new (TRUE, 0);
g_assert (game_hbox);

board_hbox = gtk_hbox_new (FALSE, 4);

board_box = gtk_spin_button_new_with_range (1, 12, 1);
g_assert (board_box);

gtk_spin_button_set_value (GTK_SPIN_BUTTON (board_box), board);
g_signal_connect(G_OBJECT(board_box), "value-changed", G_CALLBACK(
    settings_callback), NULL);
gtk_box_pack_end (GTK_BOX (board_hbox), board_box, FALSE, FALSE, 0);

board_label = gtk_label_new ("Starting level");
g_assert(board_label);

gtk_box_pack_end (GTK_BOX (board_hbox), board_label, FALSE, FALSE, 0)
;

gtk_box_pack_start (GTK_BOX (game_hbox), board_hbox, FALSE, FALSE, 2)
;

sound_hbox = gtk_hbox_new (FALSE, 4);

sound_check = gtk_check_button_new();
g_assert (sound_check);

gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON(sound_check),
    enable_sound);
g_signal_connect (G_OBJECT(sound_check), "clicked",
    G_CALLBACK(sound_callback), NULL);

gtk_box_pack_end (GTK_BOX (sound_hbox), sound_check, FALSE, FALSE, 0)
;

sound_label = gtk_label_new ("Sound");
g_assert (sound_label);

gtk_box_pack_end (GTK_BOX (sound_hbox), sound_label, TRUE, TRUE, 0);

gtk_box_pack_start (GTK_BOX (game_hbox), sound_hbox, FALSE, FALSE, 2)
;

printf ("%s : %s : %d\n", __FILE__, __FUNCTION__, __LINE__);

return game_hbox;
}

```

Listing 8.11: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is responsible for using Gconf to store the user preferences (as is recommended, since the back-up application stores the GConf database):

```

static void write_config (void)
{
    int value;

    value = gtk_spin_button_get_value_as_int (GTK_SPIN_BUTTON(board_box))
;
}

```

```

if (value < 1) value = 1;
else if (value > 12) value = 12;
gconf_client_set_int(gcc, SETTINGS_LEVEL, value, NULL);

gconf_client_set_int(gcc, SETTINGS_SOUND,
    gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(sound_check)),
    NULL);
}

```

Listing 8.12: crazyparking/src/plugin/plugin.c

The following example illustrates the function that initializes the game-specific plug-in menu, which is between the Game and Close submenus of osso-games-startup:

```

static GtkWidget **load_menu (guint *nitems)
{
    int enable_sound;
    //number of entries in maemo-games-startup main menu for this game
    *nitems = 1;
    settings_item = gtk_menu_item_new_with_label ("Settings");
    settings_menu = gtk_menu_new ();
    menu_items[0] = settings_item;
    gtk_menu_item_set_submenu (GTK_MENU_ITEM (settings_item),
        settings_menu);
    //difficulty settings
    difficulty_menu = gtk_menu_new ();
    difficulty_item = gtk_menu_item_new_with_label ("Difficulty");
    gtk_menu_item_set_submenu (GTK_MENU_ITEM (difficulty_item),
        difficulty_menu);
    gtk_menu_append (GTK_MENU (settings_menu), difficulty_item);
    level_1_item = gtk_radio_menu_item_new_with_label (group, "Level 1");
    gtk_menu_append (GTK_MENU (difficulty_menu), level_1_item);
    group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_1_item));
    level_2_item = gtk_radio_menu_item_new_with_label (group, "Level 2");
    gtk_menu_append (GTK_MENU (difficulty_menu), level_2_item);
    group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_2_item));
    level_3_item = gtk_radio_menu_item_new_with_label (group, "Level 3");
    gtk_menu_append (GTK_MENU (difficulty_menu), level_3_item);
    group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_3_item));
    /* ... Listing cut for brevity ... */
    level_12_item = gtk_radio_menu_item_new_with_label (group, "Level 12"
    );
    gtk_menu_append (GTK_MENU (difficulty_menu), level_12_item);
    group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_12_item))
    ;
    g_signal_connect (G_OBJECT (level_1_item), "toggled",
        G_CALLBACK (plugin_callback), (gpointer) LEVEL_1);

    g_signal_connect (G_OBJECT (level_2_item), "toggled",
        G_CALLBACK (plugin_callback), (gpointer) LEVEL_2);

    g_signal_connect (G_OBJECT (level_3_item), "toggled",
        G_CALLBACK (plugin_callback), (gpointer) LEVEL_3);
    /* ... Listing cut for brevity ... */

    g_signal_connect (G_OBJECT (level_12_item), "toggled",
        G_CALLBACK (plugin_callback), (gpointer) LEVEL_12);

    gtk_menu_append (GTK_MENU (settings_menu), gtk_menu_item_new());

    //sound settings
}

```

```

sound_item = gtk_check_menu_item_new_with_label("Sound");
gtk_menu_append (GTK_MENU (settings_menu), sound_item);
g_signal_connect (G_OBJECT (sound_item), "toggled",
                  G_CALLBACK (plugin_callback), (gpointer) MENU_SOUND
                  );
gtk_check_menu_item_set_state (GTK_CHECK_MENU_ITEM(sound_item),
gtk_toggle_button_get_active (GTK_TOGGLE_BUTTON (sound_check)));

return menu_items;
}

```

Listing 8.13: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is called when any configuration is performed in the menu. This function updates the GTK_SPIN_BUTTON (level change) or the GTK_CHECK_MENU_ITEM (sound change).

```

static void plugin_callback (GtkWidget *menu_item, gpointer data)
{
    if (MENU_SOUND == (int) data && !changed){
        changed = TRUE;
        gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (sound_check),
                                     gtk_check_menu_item_get_active (
                                     GTK_CHECK_MENU_ITEM(sound_item)))
        ;
    } else if (!changed) {
        changed = TRUE;
        gtk_spin_button_set_value (GTK_SPIN_BUTTON (board_box), (int)
                                   data);
    }
    changed = FALSE;
}

```

Listing 8.14: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is called to update the menu level option, when the user chooses the level using the GTK_SPIN_BUTTON:

```

static void settings_callback (GtkWidget *widget, gpointer data)
{
    if (!changed) {
        changed = TRUE;
        gint active = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(
            widget));
        if (active == LEVEL_1) {
            gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_1_item),
            TRUE);
        }
        else if (active == LEVEL_2) {
            gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_2_item),
            TRUE);
        }
        else if (active == LEVEL_3) {
            gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_3_item),
            TRUE);
        }
        /*... Listing cut for brevity ...*/
        else if (active == LEVEL_12) {
            gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_12_item),
            TRUE);
        }
    }
}

```

```
    changed = FALSE;
}
```

Listing 8.15: crazyparking/src/plugin/plugin.c

The following example illustrates the function that handles changes to the sound setup:

```
static void sound_callback (GtkWidget *widget, gpointer data)
{
    if (!changed) {
        changed = TRUE;
        gtk_check_menu_item_set_state (GTK_CHECK_MENU_ITEM(sound_item),
                                       gtk_toggle_button_get_active(
                                           GTK_TOGGLE_BUTTON (widget)));
    }
    changed = FALSE;
}
```

Listing 8.16: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is called by osso-games-startup, if necessary.

```
static void update_menu (void)
{
    settings_callback(board_box, NULL);
    sound_callback(sound_check, NULL);
}
```

Listing 8.17: crazyparking/src/plugin/plugin.c

Chapter 9

Using Connectivity Components

9.1 Introduction

The following code examples are used in this chapter:

- [example_bluetooth.c](#)
- [example_gps.c](#)

For communications with the outside world, the maemo platform provides frameworks ranging from Bluetooth file transfer to making video calls. This chapter goes through the components important for the maemo application developer.

Acquiring Internet Access

LibConIC is the library that must be used by all applications wanting to use Internet connectivity. It takes care of e.g. scanning of available WLAN networks, and setting up the IP network after the user has selected a connection. *LibConIC* API works together with the maemo connectivity daemon (*ICd*) that handles both WLAN and Bluetooth connections. *LibConIC* and *ICd* are described more deeply in section [9.2](#).

VoIP, Instant Messaging and Presence

The *Telepathy*[\[94\]](#) communications framework provides a unified API for presence, messaging and voice/video calls. The list of supported protocols is long: IRC, ICQ, XMPP (Jabber), SIP, MSN etc. The connection manager of *Telepathy* is expandable, and uses D-BUS for communication. *Telepathy* usage in maemo is described in more detail in section [9.3](#).

Bluetooth

A high level API for Bluetooth is offered as part of the maemo connectivity subsystem. Using its D-BUS API, a program can find remote Bluetooth devices, such as phones, send files over OBEX object push, and create pairings with remote devices. For these tasks, it is recommended for an application to use

this framework, as it not only has a lot simpler API, but makes the applications look and behave consistently.

For Bluetooth operations that are not supported by the maemo connectivity framework, maemo includes a lower level BlueZ D-BUS API, which is also the main Bluetooth interface for all Linux systems. The BlueZ API has features for practically all aspects of Bluetooth systems, and as a consequence is a lot more complex than the higher level Maemo Connectivity subsystem's offerings.

Section 9.2 describes the high level D-BUS API and its use. More information about the BlueZ API can be found at BlueZ web site[3]. The maemo-example package also includes example code about both libraries.

OBEX

Bluetooth devices use OBEX protocol to exchange data objects. Maemo includes libraries to help working with this protocol. For OBEX FTP, the easiest way is to use GnomeVFS's OBEX backend. For more granular control, there is *libgwobex*, which GnomeVFS uses in its implementation, and an even more low level interface can be found in the OpenOBEX library.

More about GnomeVFS is written in the GnomeVFS section of this document. Section 9.2 contains separate subsections for *libgwobex* and *OpenOBEX*. For *libgwobex* API, see Maemo API Reference[57]. Lots of information about OpenOBEX can be found at the project's web site[81].

9.2 Maemo Connectivity

The maemo connectivity subsystem is implemented by using known Linux conventions. It resides in the user mode area of Linux, and relies on the Linux kernel through standard C libraries. Wireless LAN or WiMAX is the main channel to the Internet, but dial-up connections through cellular networks are also supported. The only medium to the phone is Bluetooth. The Bluetooth software of maemo is based on BlueZ, which is known as the de-facto implementation of Bluetooth for Linux. D-Bus is used for internal application level message exchange.

Even though the connectivity device drivers are closely related to this subsystem, they are considered to be outside of the scope.

Components of the maemo connectivity architecture:

Maemo connectivity UI - User Interface parts of the connectivity. This includes Connection manager, Control Panel applets and several different dialogs.

Maemo connectivity daemon (ICd) - LibConIC API works together with ICd, handling all Internet Access Points (IAPs). IC daemon handles both WLAN and Bluetooth connections.

OBEX wrapper - Interface to OBEX services. The primary target user of this library is the OBEX gnome-vfs module.

OpenOBEX - Open source implementation of the Object Exchange (OBEX) protocol. More information on OpenOBEX can be found from <http://triq.net/obex/>

BlueZ Bluetooth stack - The de-facto implementation of Bluetooth for Linux. More information on BlueZ can be found from <http://www.bluez.org>

BlueZ D-Bus API - BlueZ accepts commands via D-Bus.

WLAN connectivity daemon - The daemon controlling WLAN connections.

WLAN device driver - Device driver for Wireless LAN (IEEE 802.11g). Kernel driver is composed of two parts: a binary part (closed source) and an open source wrapper, binding the binary to the current Linux kernel.

WiMAX onnectivity daemon - The daemon controlling WiMAX connections.

WiMAX device driver - Device driver for mobile WiMAX (IEEE 802.16e).

Internet Access Points (IAP)

The central concept regarding Internet connections from maemo is the Internet Access Point (IAP). It represents a logical Internet (IP) connection, which will be defined by the user according to their needs. An IAP has a unique name usually in form of UUID. It defines the radio bearer (e.g. WLAN, CSD, GPRS) to be applied, and usually the data transfer speed, username, password, proxy server, and the corresponding access point in the Internet or the telephone number of the service provider's modem, among other characteristics.

9.2.1 Connectivity Subsystem

This section describes the system decomposition of the Connectivity subsystem. Maemo applications can open Internet connections by using the LibConIC API. The Internet Access subsystem will take care of the connection to the phone, if necessary, by using the services of the Phone Access subsystem. If an application needs to gain access to the phone's files, then the File selector will consult the Phone Access subsystem.

During the offline mode, none of the radios must be active. The Device System Management Entity (DSME) of maemo provides information about the transitions to and from the offline mode.

Name	Connection Manager
Purpose	Provides the UI for managing phone and Internet connections. Available as a Control Panel applet, and from the Status Bar.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Displaying active connections with statistics • Updating status indicators • Providing dialogs for changing and disconnecting connections
Concurrent usage	(N/A)

Name	Phone Access
Purpose	Provides connections to phones with different Bluetooth profiles
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Searching for phones and inquiring their services • Keeping phone register • Providing status of the phone connections for Connection Manager • Binding RFCOMM devices to DUN and FTP services on the phone • Providing easy access to OpenOBEX
Concurrent usage	Number of clients not limited by maemo. However, some phones may not support more than one Bluetooth profile at a time.

Name	Internet Access
Purpose	Provides Internet connections over different bearers.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Providing means for configuration and management of IAP settings • Providing API for Internet connections over different bearers (e.g. WLAN, WiMAX, Bluetooth dialup) • Providing status of Internet connections for Connection Manager
Concurrent usage	Number of clients not limited, but only one connection to the Internet can exist at any given time

Phone and Internet connections are quite different by nature and behavior. These will be introduced in more detail in the following sections.

Phone Access

Phone Access is the subsystem handling connections to a cellular phone. It has a search utility for finding potential phones and inquiring the services they can offer. This is based on the standard Bluetooth service discovery mechanism. Phone Access also keeps a record of phones having been connected to the device in GConf, and provides a list of them for the user to choose from. Phone Access relies on the Linux Bluetooth implementation called BlueZ. BlueZ offers the Berkeley socket interface to the HCI and to the L2CAP protocol for the user space applications.

In principle, any cellular phone supporting Bluetooth Service Discovery Protocol (SDP), Dial-up Networking profile (DUN) and File Transfer Profile

(FTP) can be connected to maemo. However, there is variation especially in the level of file transfer services and OBEX in different mobile phones. Some products limit the access to the Inbox (Object Push), whereas more sophisticated ones make the Gallery and the memory card available. The recent products support the OBEX Capability request, which can be used to get more specific information about the file system on the phone.

Maemo connects to a phone on an on-demand basis, i.e. when an application requires a connection. For example, when the Internet browser is about to open a URL, it will request the Phone Access to establish a connection to the phone. This makes Phone Access to bind an RFCOMM device to the requested service (in this case DUN) on the phone. In a similar fashion, the File Selector can set up a file transfer connection to the phone using another RFCOMM device. After binding to a service, the application in question can open the local RFCOMM device. Normal file selector access is performed with GnomeVFS layer to get transparent access to phone in the same way as internal flash and MMC are accessed.

The Bluetooth SIM Access (SAP) profile is also needed in maemo to perform WLAN authentication using the EAP-SIM authentication method. In this case, the EAP component will ask the BT sap component to get session keys from a GSM/UMTS phone.

Name	Phone selection UI
Purpose	User interface for managing phone operations. Supports Connection Manager.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Keeping a list of phones • Storing the Bluetooth device addresses of phones to GConf • Invoking device search and capability query
Concurrent usage	N/A

Name	General Bluetooth UI
Purpose	User interface for managing all paired BT devices
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Keeping a list of all paired devices (not only phones)
Concurrent usage	N/A

Name	BT search
Purpose	Searches for available Bluetooth devices
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Bluetooth inquiry • Checking on offline mode state
Concurrent usage	N/A

Name	BT service discovery
Purpose	Checks if a found Bluetooth device is sufficient
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Bluetooth service discovery (SDP)
Concurrent usage	Not limited (but used by Phone selection UI and Phone connection daemon only)

Name	GW OBEX library
Purpose	Provides access to OpenOBEX library for the File selector (Gnome VFS) on a higher abstraction level than OpenOBEX itself supports
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Handling OBEX requests
Concurrent usage	Not limited

Name	BT SAP
Purpose	Obtains session keys for EAP-SIM authentication from the phone
Responsibilities and additional requirements	<ul style="list-style-type: none"> • SIM/PIN entry • Connecting to the phone with SIM Access Profile • Delivering the session keys to EAP
Concurrent usage	N/A

Maemo Bluetooth also supports HID (keyboard) and OPP (object push file transfer) profiles.

Internet Access

The Internet Access subsystem manages connections to the Internet over different bearers. It is also responsible for the configuration and management of Internet Access Points. The Internet Access provides applications with TCP/IP connections. They can be established with:

- WLAN connection to a wireless access point.
- WiMAX connection to a WiMAX base station.
- Bluetooth connection through phone using Point-to-Point Protocol (PPP) and a cellular modem (in the phone).

In the latter case, AT commands are applied to establish a PPP link to the cellular modem and the connection to the Internet.

Name	IC daemon
Purpose	IC daemon establishes Internet connections over different bearers.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Controlling that only one Internet connection (one active IAP) can exist at any given time • Using Phone Access for getting a character device for a dial-up connection, and using WLAN or WiMAX connection daemon for getting a network device and getting the connection authenticated • Starting IP level services like PPP and DHCP • Providing statistics about the usage of IAPs to any application
Concurrent usage	Has multiple clients and limits the connections to one at a time

Name	Internet Connectivity GUI
Purpose	This package has the GUI applications for configuring Internet Access Points and WLAN settings. N.B. The Connection Manager is a separate application.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Providing the UI for configuring an IAP • Providing the UI for WLAN settings • Scanning for available WLAN networks (on request) • Saving IAP and WLAN settings (excluding EAP settings) to GConf • Opening up dialogs and displaying notifications
Concurrent usage	N/A

Name	WLAN connection daemon
Purpose	Manages WLAN network connections
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Starting WLAN driver with the settings provided; stopping WLAN • Requesting authentication when necessary • Relaying WLAN IAP settings from IC daemon to WLAN drivers and authentication requests to EAP • Relaying WLAN events from WLAN drivers • Providing WLAN status information
Concurrent usage	Not limited

Name	EAP UI
Purpose	User interface for EAP authentication
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Invoking a dialog for an authentication password • Showing authentication status
Concurrent usage	N/A

Name	EAP
Purpose	Provides WLAN and WiMAX security excluding basic WEP settings, which are in Wireless Extensions
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Taking care of the authentication process for the active IAP • Delivering progress events to IC daemon • As a special case, controlling the EAP-SIM authentication using the SIM Access Profile • Providing authentication status
Concurrent usage	N/A

Name	WiMAX connection daemon
Purpose	Manages WiMAX network connections
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Starting WiMAX driver with the settings provided; stopping WiMAX • Relaying WiMAX IAP settings from IC daemon to WiMAX drivers • Relaying WiMAX events from WiMAX drivers • Providing WiMAX status information • Scanning WiMAX NAPs and NSPs
Concurrent usage	Not limited

9.2.2 Internet Connectivity Daemon

This section describes how the Internet Connectivity daemon works internally. The following subsections explain the behavior and the decomposition of this component in detail, also covering the interfaces that this component realizes.

Decomposition

When the ICd receives a request to activate or deactivate an IAP, the ICd will activate the IAP or show a ui requesting the user to choose one, if no IAP has been selected as the default. Depending on the type of the IAP, the ICd will use appropriate network type plug-in to activate or deactivate specific network interface.

The ICd tracks the applications requesting IAPs by recording their D-Bus base service names. This allows the ICd to detect situations where processes using an IAP have aborted or crashed. The ICd also implements an idle timeout mechanism to shut down the active IAP, if no packets have been sent in a configured amount of time.

Maemo version 3.0 introduced the automatic connection creation feature in the Internet Connectivity Daemon. In other words, the device will try to connect automatically to the saved IAPs, and keep connected as long as possible, unless the idle timeout is set. With this feature, applications like e-mail and RSS reader will always be up to date. The device will also be always ready for online usage, for example, incoming VoIP calls or IM chat. In former versions, the Internet connection was automatically closed when there were no more applications using it, or when the connection was idle for a given period of time defined by the configuration parameter idle timeout.

When not connected, the device scans for saved IAPs and tries to connect automatically, taking into account the value defined by the configuration parameter for search interval, which can be 5, 10, 30 or 60 minutes. All other values will be automatically mapped to "Never". This setup switches off the automatic connection feature. In this case, the device will behave just like the

former versions: Connections will be created only when required by applications.

The ICd is responsible for connection creation only, as it is the responsibility of each application to keep its data updated, and then providing the always-online feature.

While writing an application making use of the ICd system, the following points should be kept in mind:

- The application must always use the existing available connection.
- As it was done in former versions, if device is not connected but a connection is required by user interaction, the application must require connection creation using LibConIC API.
- The user should be kept aware of updates, making visible when the data was last updated.
- The application must register via LibConIC and listen to signals emitted by the ICd (Connection Created, Lost and Changed), and react as follows:
 - **Connection Created:** Use the connection and update all data.
 - **Connection Lost:** Go to an idle state silently and wait until a new connection is created.
 - **Connection Changed:** Use the new connection.
- Automatic data updates must run in background and silently:
 - Alarming the user with unnecessary banners or dialogs should be avoided.
 - Usernames and passwords should be saved, so that automatic updates can be performed without prompts.
 - In this case, no failures are allowed to show error notification dialogs.
- The connectivity infrastructure takes care of error situations in a centralized way.

Automatic connection creation feature can also be switched off using offline mode (i.e. offline mode). While in this mode, the configuration parameter for allowing WLAN in offline mode is checked. Depending on the state of this configuration parameter, WLAN IAPs are either enabled or disabled in the offline mode. Also other radios like Bluetooth are normally disabled in the offline mode.

Bluetooth Dial-up Networking The ICd uses PPP to establish IP connectivity over Bluetooth DUN interfaces. If there already is a different IAP active using Bluetooth DUN, the old IAP is first deactivated. The IAP is activated according the following action sequence:

- The character device used by the Bluetooth DUN device is acquired from btcond. If the device is not available due to gateway not being present, exhaustion of simultaneous Bluetooth connections, or similar, the ICd shows an error message to the user and aborts with a D-BUS error message.

- The ICd starts PPP using the exec family of system calls. It directs PPP to use the acquired Bluetooth DUN device with the dial-up configuration parameters specified for the configured DUN IAP type. If PPP cannot get the connection established, the ICd will show an error message to the user, and abort with a D-BUS error message. When the PPP connection is established, PPP-specific scripts will be run. These scripts will set dynamic IP connection related configuration entries, and send a state change D-BUS message to all the interested applications to indicate that the IAP has been established.

If the previously active IAP was not using Bluetooth DUN, it will be closed down after establishing the PPP connection.

A Bluetooth DUN is closed down by sending the PPP daemon a SIGINT or SIGTERM signal. This will terminate the PPP daemon, and remove all routing entries associated with the PPP dial-up interface. The PPP shutdown scripts will remove the dynamic IP connection related configuration entries, and send a state change D-BUS message announcing deactivation of the IAP. This is described below.

WLAN When connecting to a WLAN, the ICd needs to associate with the network, and enable EAP authentication and the DHCP client as needed. Independently of whether there is an IAP active using WLAN, the requested WLAN network will first be scanned to ensure that it is available. The current IAP will be deactivated, if the requested network is found and the current IAP is using WLAN. WLAN is activated according to the following procedure:

- If the network requires EAP authentication, the EAP authentication procedure is started. While performing the EAP authentication, the EAP software may show GUI dialogs relating to the EAP authentication procedure. When the EAP authentication has been completed, the EAP software will set security keys for the WLAN network, resulting in state change messages from wlancond. The ICd will receive these messages but ignore them, and wait for the reply from EAP authentication instead. If the EAP authentication fails, the ICd aborts with a D-Bus error message.
- After the EAP process has been started, the ICd instructs wlancond to associate with the WLAN network. Any static security settings relating to pre-shared security keys are also supplied at this point. If a connection to the WLAN network cannot be established, the ICd aborts with an error.
- As the DHCP client is a stand-alone program, it is started by using exec when the WLAN IAP requires dynamic IP address acquisition. When the DHCP client has obtained an IP address, it configures IP-related parameters, and sends a D-Bus signal to the ICd. If the IP address lease cannot be obtained, the ICd will timeout, stop the DHCP client and abort with a D-Bus error message.

9.2.3 LibConIC Library

Internet Connectivity API (in shorter form: Libconic) is an API for applications to manage internet connections on Maemo devices. It was introduced in the first

IT OS 2007 release, deprecating the old OSSO IC API (osso-ic-lib). OSSO IC API was conclusively removed in IT OS 2008 release. The interface documentation to libconic can be found at [Internet Connectivity API](#)

Unlike osso-ic-lib, Libconic is high level and stable object-oriented API. It can be used to

- Request Internet connection
- Listen for Internet connection status events
- Receive statistics of Internet connection
- Get proxy settings for the current connection
- Get list of user-saved connections (IAPs)

Application Requirements

Applications have a few requirements prior using the Libconic API. They are:

1. use non-blocking sockets
2. system D-BUS running
3. *g_type_init()* has to be called
4. no threading support in the Libconic API

If the application is a standard Hildon application, almost all of these requirements are already fulfilled. LibOSSO context initialization connects the application to both session and system D-BUS buses, *g_type_init()* is called as a part of *gtk_init()*, and there probably will be no extra threads used.

Non-Blocking Sockets

Blocking sockets cannot be used, because that would also block receiving the Connectivity events. Non-blocking sockets should be used in order to receive the events properly. For example, GLib IO Channels with the *G_IO_FLAG_NONBLOCK* flag provide non-blocking way to use sockets.

With threads, blocking sockets can be used, although Libconic API itself is not thread safe.

System D-BUS

Libconic API uses internally system D-BUS for delivering messages to the Connectivity components. Application needs to have normal D-BUS dispatch, watch and timeout monitoring running before the Libconic API can be used. If the GLib mainloop is used, this can be accomplished with *dbus_connection_setup_with_g_main()*.

N.B. Setting up LibOSSO context connects the application to required D-BUS.

GType

Libconic API is GObject-based. This means that to get API working, GLib's GType needs to be initialized properly. Use *g_type_init()* to do that.

No Multiple Threads

Libconic API is not thread-safe. If the applications have threads, use Libconic API only from the same context where GMainloop is running.

Libconic Usage

Requesting for Connection Libconic is an asynchronous connection API, which heavily relies on GObject signals. Basically this means that GMainloop must be iterated in order to successfully execute connection requests. After the application has been correctly set up (please see the previous section for some requirements), a connection can be requested with *ConIcConnection* object:

```
gboolean success = FALSE;

/* Create connection object */
ConIcConnection *connection = con_ic_connection_new();

/* Connect signal to receive connection events */
g_signal_connect(G_OBJECT(connection), "connection-event",
                 G_CALLBACK(my_connection_handler), NULL);

/* Request connection and check for the result */
success = con_ic_connection_connect(connection,
                                     CON_IC_CONNECT_FLAG_NONE);
if (!success) g_warning("Request for connection failed");
```

At this point, the application does not yet have an Internet connection. A successful return from *con_ic_connection_connect()* means only that the request was successfully dispatched to the Internet Connectivity daemon. When the daemon has a connection ready, the application will receive an event (as an GObject signal) indicating that the device is connected. If the connection attempt fails, the application will receive a disconnected event with an error describing the reason for the failure.

The connection handler (*my_connection_handler()* function registered in the previous snippet) could look like this:

```
static void my_connection_handler(ConIcConnection *connection,
                                 ConIcConnectionEvent *event,
                                 gpointer user_data)
{
    ConIcConnectionStatus status = con_ic_connection_event_get_status(
        event);
    ConIcConnectionError error;

    const gchar *iap_id = con_ic_event_get_iap_id(CON_IC_EVENT(event));
    const gchar *bearer = con_ic_event_get_bearer_type(CON_IC_EVENT(
        event));

    switch(status) {
        case CON_IC_STATUS_CONNECTED:
            g_debug("Hey, we are connected to IAP %s with bearer %s!",
                    iap_id, bearer);
            break;

        case CON_IC_STATUS_DISCONNECTING:
            g_debug("We are disconnecting...");
            break;
```

```

    case CON_IC_STATUS_DISCONNECTED:
        g_debug("And we are disconnected. Let's see what went wrong
            ...");
        error = con_ic_connection_event_get_error(event);

        switch(error) {

            case CON_IC_CONNECTION_ERROR_NONE:
                g_debug("Libconic thinks there was nothing wrong.");
                ;
                break;

            case CON_IC_CONNECTION_ERROR_INVALID_IAP:
                g_debug("Invalid (non-existing?) IAP was requested.
                    ");
                break;

            case CON_IC_CONNECTION_ERROR_CONNECTION_FAILED:
                g_debug("Connection just failed.");
                break;

            case CON_IC_CONNECTION_ERROR_USER_CANCELED:
                g_debug("User canceled the connection attempt");
                break;

        }
        break;

    default:
        g_debug("Unknown connection status received");
    }
}

```

Listening for Connection Events Sometimes the application does not want to actively start connections, but it is still interested in knowing, when the device is online. It is possible to achieve this with Libconic "automatic events" feature, which is enabled with "automatic-events" GObject property:

```

/* Create connection object */
ConIcConnection *connection = con_ic_connection_new();

/* Connect signal to receive connection events */
g_signal_connect(G_OBJECT(connection), "connection-event",
    G_CALLBACK(my_connection_handler), NULL);

/* Set automatic events */
g_object_set(G_OBJECT(connection), "automatic-events", TRUE, NULL);

```

When automatic events are turned on, the application will receive connected and disconnected events for all Internet connection changes. In addition to this, the application will receive an event for the initial connection status. If the device is disconnected, *ConIcConnectionEvent* with status *CON_IC_STATUS_DISCONNECTED* will be emitted. This event will have NULL IAP ID and bearer, as there is no IAP getting disconnected, but the event just indicates that the device is offline.

N.B. The main loop has to be reiterated in order to receive the event. If the connection status information is needed synchronously, one can always iterate the main loop oneself:

```

static void connection_info(ConIcConnection *connection,
                           ConIcConnectionEvent *event,
                           gpointer user_data)
{
    ConIcConnectionStatus status = con_ic_connection_event_get_status(
        event);
    ConIcConnectionStatus *status_ptr = (ConIcConnectionStatus*)
        user_data;
    *status_ptr = status;
}

/* ... */

/* Create connection object and set on automatic events (see
previous snippet) ... */
static ConIcConnectionStatus status = 0xFFFF;
ConIcConnection *connection = con_ic_connection_new();
g_signal_connect(G_OBJECT(connection), "connection-event",
                G_CALLBACK(connection_info), &status);
g_object_set(G_OBJECT(connection), "automatic-events", TRUE, NULL);

/* Iterate main loop for the first connection event */
while (status == 0xFFFF) g_main_context_iteration(NULL, TRUE);

if (status == CON_IC_STATUS_CONNECTED)
    g_debug("We are connected!");
else
    g_debug("We are not connected!");

```

Receiving Statistics of Connection Receiving statistics of the current Internet connection is achieved with the `con_ic_connection_statistics()` function and the corresponding event handler. If wanted, it is possible to get statistics for a specified IAP, or just for the current default connection. N.B. Currently the Internet Connectivity daemon provides only one Internet connection at the time, so leaving IAP ID NULL is the best option.

```

static void connection_statistics(ConIcConnection *connection,
                                 ConIcStatisticsEvent *event,
                                 gpointer user_data)
{
    g_debug("Here are all kind of nice statistics about the connection:");
    g_debug("Time active: %u, signal strength: %u, received packets: %llu, "
            "sent packets: %llu, received bytes: %llu, sent bytes: %llu",
            con_ic_statistics_event_get_time_active(event),
            con_ic_statistics_event_get_signal_strength(event),
            con_ic_statistics_event_get_rx_packets(event),
            con_ic_statistics_event_get_tx_packets(event),
            con_ic_statistics_event_get_rx_bytes(event),
            con_ic_statistics_event_get_tx_bytes(event));
}

/* ... */

/* ConIcConnection object named "connection" has already been
created */
g_signal_connect(G_OBJECT(connection), "statistics",

```

```
G_CALLBACK(connection_statistics), NULL);
if (!con_ic_connection_statistics(connection, NULL))
    g_warning("Requesting connection statistics failed!");
```

Getting Proxy Settings With Libconic, it is possible to get Internet connection proxy settings for various protocols. The first step is to query, what kind of proxy mode is in use. This is achieved with the `con_ic_connection_get_proxy_mode()` function. After getting the proxy mode, use the following functions to get the actual proxy settings:

- If proxy mode is `CON_IC_PROXY_MODE_NONE`, do not use any proxies.
- If proxy mode is `CON_IC_PROXY_MODE_MANUAL`, use the following functions to query proxy settings:
 - `con_ic_connection_get_proxy_host()` to get the proxy host
 - `con_ic_connection_get_proxy_port()` to get the proxy port
 - `con_ic_connection_get_proxy_ignore_hosts()` to get a list of hosts, for which the proxy should not be used.
- If proxy mode is `CON_IC_PROXY_MODE_AUTO`, use `con_ic_connection_get_proxy_autoconfig_url()` to get a proxy auto configuration URL.
 - Use of auto configuration URL is explained in [Wikipedia](#)

In this example, the "connection-event" handler is modified to print HTTP proxy settings when establishing connection:

```
static void my_connection_handler(ConicConnection *connection,
                                ConicConnectionEvent *event,
                                gpointer user_data)
{
    ConicConnectionStatus status = con_ic_connection_event_get_status(
        event);
    GSList *ignore_hosts;

    if (status == CON_IC_STATUS_CONNECTED) {
        g_debug("We are connected! \
Let's see what kind of settings we have for HTTP proxy...");

        /* Do things based on specified proxy mode */
        switch (con_ic_connection_get_proxy_mode(connection)) {

            case CON_IC_PROXY_MODE_NONE:
                g_debug("No proxies defined, it is direct connection");
                break;

            case CON_IC_PROXY_MODE_MANUAL:
                g_debug("HTTP proxy %s:%d in use",
                    con_ic_connection_get_proxy_host(connection,
                        CON_IC_PROXY_PROTOCOL_HTTP),
                    con_ic_connection_get_proxy_port(connection,
                        CON_IC_PROXY_PROTOCOL_HTTP));
```

```

        g_debug("List of hosts, for which proxy should not be
            used:");
        ignore_hosts = con_ic_connection_proxy_ignore_hosts(
            connection);
        for (GSLIST *iter = ignore_hosts; iter != NULL; iter =
            g_slist_next(iter))
        {
            g_debug("%s", (gchar *)iter->data);
            g_free(iter->data);
        }
        g_slist_free(ignore_hosts);
        break;

    case CON_IC_PROXY_MODE_AUTO:
        g_debug("Proxy auto-config URL %s should be used",
            con_ic_connection_get_proxy_autoconfig_url(
                connection));
        break;
    }
}
}

```

There are also proxy functions for each individual protocol (like `con_ic_connection_get_proxy_ftp_host()`), but these functions are deprecated and should be avoided in newly-written code.

Getting List of User-Saved Connections All user-saved connections (IAPs) can be queried with the `con_ic_connection_get_all_iaps()` function. The function returns simply a singly linked list of `ConIcIap` objects:

```

/* ConIcConnection object named "connection" has already been
   created */
GSLIST *saved_iaps = con_ic_connection_get_all_iaps(connection);

g_debug("The following connections have been saved by the user:");
for (GSLIST *iter = saved_iaps; iter != NULL; iter = g_slist_next(
    iter)) {

    /* Get IAP object and print some information about it */
    ConIcIap *iap = (ConIcIap *)iter->data;
    g_debug("Connection %s called '%s' using bearer %s",
        con_ic_iap_get_id(iap), con_ic_iap_get_name(iap),
        con_ic_iap_get_bearer_type(iap));

    /* We unref the IAP object as we are not going to use it
       anymore */
    g_object_unref(iap);
}

g_slist_free(saved_iaps);

```

Porting Application from OSSO IC API to Libconic

- If the application has not configured GType, GLib or D-BUS, then set them up:

```

DBusConnection *system_dbus;
GMainloop *main_loop;

```



```

g_type_init();
main_loop = g_main_loop_new(NULL, FALSE);

system_dbus = dbus_bus_get(DBUS_BUS_SYSTEM, NULL);
dbus_connection_setup_with_g_main(system_dbus, NULL);

```

- Include the correct header file:

```

#include <osso-ic.h>

==>

#include <conic.h>

```

- Set up *ConIcConnection* object and "connection-event" handler instead of *osso_iap_cb_t* callback:

```

static void my_connection_cb(struct iap_event_t *event, void
                             *arg)
{
    /* ... */
}

/* ... */

osso_iap_cb(my_connection_cb);

==>

static void my_connection_cb(ConIcConnection *connection,
                             ConIcConnectionEvent *event,
                             gpointer user_data)
{
    /* ... */
}

/* ... */

ConIcConnection *connection = con_ic_connection_new();
g_signal_connect(G_OBJECT(connection), "connection-event",
                 G_CALLBACK(my_connection_cb), app_data)
;

```

- Manage connections through *ConIcConnection* API instead of *osso_iap_connect()* and *osso_iap_disconnect()*:

```

osso_iap_connect(OSSO_IAP_ANY, OSSO_IAP_REQUESTED_CONNECT,
                 app_data);
osso_iap_disconnect(iap_name, app_data);

==>

con_ic_connection_connect(connection,
                          CON_IC_CONNECT_FLAG_NONE);
con_ic_connection_disconnect(connection);

```

- Request statistics with *con_ic_connection_statistics()* instead of *osso_iap_get_statistics()*.

- List all available IAPs with `con_ic_connection_get_all_iaps()` instead of `osso_iap_get_configured_iaps()`.
- Configure autoconf to use Libconic instead of OSSO IC API:

```
PKG_CHECK_MODULES(OSSOIC, osso-ic)
AC_SUBST(OSSOIC_CFLAGS)
AC_SUBST(OSSOIC_LIBS)

====>

PKG_CHECK_MODULES(CONIC, conic)
AC_SUBST(CONIC_CFLAGS)
AC_SUBST(CONIC_LIBS)
```

- In debian/control file "Build-Depends" section, depend on libconic0-dev instead of osso-ic-dev.

9.2.4 Bluetooth Libraries

This section explains how maemo Bluetooth libraries work internally. The following subsections explain the behavior and the decomposition of the Bluetooth library components in detail.

Libgwobex

Libgwobex provides access to libopenobex functionality by providing a helper/wrapper interface for it. Libopenobex is explained in detail in the following section.

The interface to libgwobex can be found at [GW OBEX Library Documentation](#).

Creating Connection

The connection with libgwobex is established using the `gw_obex_setup_dev` function, setting up the connection.

```
#define OBEX_FTP_UUID \
    "\xF9\xEC\x7B\xC4\x95\x3C\x11\xD2\x98\x4E\x52\x54\x00\xDC\x9E\x09"
#define OBEX_FTP_UUID_LEN 16

/* ... */

GwObex* gw_obex_setup_dev (const gchar * device, const gchar * uuid,
                           gint uuid_len,
                           GMainContext * context, gint * error )
```

Listing 9.1: gw-obex.h

The following code snippet illustrates how to open a handle using `gw_obex_setup_dev`.

```
if (ctx->rftcomm_dev) {
    if (ctx->use_ftp)
        ctx->obex = gw_obex_setup_dev(ctx->rftcomm_dev,
                                       OBEX_FTP_UUID, OBEX_FTP_UUID_LEN,
                                       NULL, &err);
}
```

```

else
    ctx->obex = gw_obex_setup_dev(ctx->rftcomm_dev, NULL,
                                  0, NULL, &err);

    if (ctx->obex == NULL)
        printf("OBEX setup failed: %s\n", response_to_string(
            err));
}

```

In this example, `ctx->rftcomm_dev` points to a string containing the device node name (e.g. `/dev/rftcomm0`). `ctx->use_ftp` dictates whether standard folder browsing services should be set up. If `use_ftp` is untrue, then INBOX is connected.

Closing Connection

For closing a gwobex connection, it is possible to use the function

```
void gw_obex_close ( GwObex * ctx );
```

Listing 9.2: gw-obex.h

The following code demonstrates this usage.

```

if (ctx->obex) {
    gw_obex_close(ctx->obex);
    ctx->obex = NULL;
}

```

If `ctx->obex` is not NULL, it will simply be passed as an argument to `gw_obex_close()`.

Using Connection

The libgwobex library provides general file handling functionality, including reading directory structure, browsing in different folders and getting files.

For reading entries from an opened directory, it is possible to use the function

```

gboolean gw_obex_read_dir (GwObex * ctx,
                           const gchar * dir,
                           gchar ** buf,
                           gint * buf_size,
                           gint * error );

```

Listing 9.3: gw-obex.h

`gw_obex_read_dir` reads an entry from the selected folder and returns the result in the `buf` argument given to the function.

```

gboolean ret;

/* ... */

ret = gw_obex_read_dir(ctx->obex, dir, buf, buf_size, err);

```

This reads an entry from the directory `dir` (`char *`) and returns it in `buf` (`char **`).

For changing the current directory, it is possible to use the function:

```
gboolean gw_obex_chdir (GwObex * ctx, const gchar * dir, gint * error )
;
```

Listing 9.4: gw-obex.h

which changes the directory of the FTP connection. Below is a code example using this function.

```
/* Ignore parent dir pointers */
if (g_str_equal(name, ".."))
    return TRUE;

if (!gw_obex_chdir(ctx->obex, name, err)) {
    printf("Could not chdir to %s\n", name);
    return FALSE;
}
```

To retrieve files over the OBEX connection, the `gw_obex_get_file` function can be used.

```
gboolean gw_obex_get_file (GwObex * ctx,
                           const gchar * local,
                           const gchar * remote,
                           const gchar * type,
                           gint * error);
```

Listing 9.5: gw-obex.h

`gw_obex_get_file` uses the `ctx` context for retrieving the remote file to local file.

```
gboolean ret;

ret = gw_obex_get_file(ctx->obex, name, name, err);
```

There are a lot more functions that can be performed by using libgwobex wrapper directly, for full list of functions and their usage, see the [API document](#).

Libopenobex

The LibOpenOBEX library implements a generic OBEX Session Protocol. It does not implement the OBEX Application Framework. OBEX is a protocol designed to allow data interchanging between different kinds of connections (e.g. Bluetooth, IrDA). Specific information about the OBEX protocol can be found at <http://www.irda.org>, by selecting the Developer->Specifications category. OBEX is similar to HTTP protocol, expect for a few differences:

- Transports: While HTTP is normally layered above a TCP/IP connection, OBEX is usually transported over IrLAP/IrLMP/Tiny TP (on IrDA) or over Baseband/Link Manager/L2CAP/RFCOMM (on Bluetooth).
- Binary transmissions: OBEX communicates using binary transmissions, as HTTP is transmitted in a human-readable XML-based format.
- Session support: HTTP is stateless, while OBEX maintains the connection.

A fairly good overlook of OBEX can be found at <http://en.wikipedia.org/wiki/OBEX>.

Code examples for libopenobex can be obtained from <http://openobex.triq.net/downloads>, from the example apps package.

Using BlueZ D-Bus API

The BlueZ system exports a D-Bus API that can be employed instead of OSSO Bluetooth tools. See the following documents:

- [BlueZ D-Bus API documentation](#)
- [BlueZ Wiki](#)

9.2.5 Connectivity UI

UI Components

Connectivity UI contains various dialogs and other components used to control the connectivity. The different UI parts are:

- Connection manager
- Connectivity dialogs
- Status bar applets
- Control panel applet
- Bluetooth UIs

The connectivity dialogs are invoked by D-Bus method calls, so for example the ICd is using these D-Bus method calls for showing dialogs when they are needed. The next section specifies the D-Bus API of maemo connectivity UI.

D-Bus Connectivity UI Interface

If some information is needed from the user about the IAP that is about to be connected to, the following can be used.

```
Service:      com.nokia.icd_ui
Interfaces:   com.nokia.icd_ui
Object paths: /com/nokia/icd_ui
```

The Internet Connectivity UIs implement the following D-Bus API used by the ICd and EAP.

```
Method:      show_conn_dlg
Parameters:  none
Return parameters: none
Errors:      com.nokia.icd_ui.error.flight_mode:
             Flight mode enabled, dialog not shown
Description: Shows the Connect Dialog where the user can choose an IAP.
```

```
Method:      show_disconnect_dlg
Parameters:  none

Return Parameters: none
Errors:      com.nokia.icd_ui.error.flight_mode:
             Flight mode enabled, dialog not shown
Description: Shows the disconnect dialog.
```

Method:	show_retry_dlg
Parameters:	1. string Bluetooth address of the device used with SAP 2. string Name of the connection attempt error which selects the retry dialog type.
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows the retry dialog.

Method:	show_change_dlg
Parameters:	1. string Name of the currently active IAP 2. string Name of the IAP to be activated
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows the Change IAP Dialog

Method:	show_passwd_dlg
Parameters:	1. string Username supplied by ICD 2. string Password supplied by ICD 3. string Name of the IAP
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows the username/password dialog.

Method:	show_gtc_dlg
Parameters:	1. string GTC challenge string
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows EAP GTC challenge dialog.

Method:	show_mschap_change_dlg
Parameters:	1. string Supplied username 2. string Old password that is to be changed 3. string Name of the IAP
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows EAP MSCHAPv2 change password dialog.

Method:	show_private_key_passwd_dlg
Parameters:	1. uint32 The private key ID
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows EAP private key password dialog

Method: show_server_cert_dlg
Parameters: 1. string Certificate name
2. string Certificate serial
3. boolean TRUE if certificate is expired, FALSE otherwise
4. boolean TRUE if root CA is unknown or self-signed certificate, FALSE otherwise

Return Parameters: none
Errors: com.nokia.icd_ui.error.flight_mode:
Flight mode enabled, dialog not shown

Description: Shows server certificate error and expiration dialogs. If both boolean arguments are false, the error dialog is shown. If either or both boolean arguments are TRUE, the expiration dialog is shown instead.

Method: strong_bt_req
Parameters: 1. string Bluetooth address of the device to pair with
2. boolean TRUE if strong authentication enabled, FALSE if strong authentication is disabled

Return Parameters: none
Errors: com.nokia.icd_ui.error.flight_mode:
Flight mode enabled, dialog not shown

Description: Requests strong (16 digit) BT PIN dialog for a BT device

Method: show_sim_pin_dlg
Parameters: 1. string Bluetooth address of the device used with SAP
2. boolean TRUE if PIN was incorrect and retry dialog should be displayed before asking PIN. FALSE if this is the first PIN request.

Return Parameters: none
Errors: com.nokia.icd_ui.error.flight_mode:
Flight mode enabled, dialog not shown

Description: Shows SIM PIN dialog

The code example for the application to show the connect dialog using show_conn_dlg is following. Please note the use of macro for doing this.

```
#include <osso-ic-ui-dbus.h>

/* ... */

/* in our code somewhere, where we need the Connect Dialog*/
DBusMessage *uimsg;

/* construct the message for Connect Dialog request*/
uimsg =
    dbus_message_new_method_call(ICD_UI_DBUS_SERVICE,
                                ICD_UI_DBUS_PATH,
                                ICD_UI_DBUS_INTERFACE,
                                /*macro for show_conn_dlg */
                                ICD_UI_SHOW_CONNDLG_REQ);

/* send the message */
reply =
    dbus_connection_send_with_reply_and_block(connection,
                                                uimsg,
                                                reply_timeout,
                                                &error);

if (reply == NULL) {
    DLOG_ERR("Failed to show connect dialog: %s", uierror.message);
}
```

```

        dbus_error_free(&uierror);
    }

    dbus_message_unref(uimsg);
    dbus_message_unref(reply);

    /* ... */

```

The signals emitted from com.nokia.icd_ui interface are listed below.

Signal:	disconnect
Parameters:	1. boolean TRUE if "disconnect" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when disconnect dialog has been closed.

Signal:	retry
Parameters:	1. string The IAP that is to be retried 2. boolean TRUE if "retry" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the retry dialog has been closed.

Signal:	change
Parameters:	1. string Old IAP to change from 2. string New IAP to change to 3. boolean Change to the new IAP If TRUE, keep old if FALSE
Description:	Signal emitted from UI when change connection dialog has been closed.

Signal:	passwd
Parameters:	1. string Username supplied or modified by the user 2. string Password supplied or modified by the user 3. string IAP name 4. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the username/password dialog has been closed

Signal:	gtc_response
Parameters:	1. string Response to the given challenge or empty string if cancelled 2. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the EAP GTC challenge dialog has been closed.

Signal:	mschap_change
Parameters:	1. string Supplied username 2. string The new password or empty string if cancelled 3. string IAP name 4. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the MSCHAPv2 password has been changed

Signal:	private_key_passwd
Parameters:	1. uint32 The id of the private key 2. string Password for the private key or empty string if none 3. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the private key password dialog has been closed

Signal:	server_cert
Parameters:	1. boolean TRUE if strong PIN entered, FALSE if strong PIN dialog was canceled
Description:	Signal emitted from UI when the server certificate error dialog has been closed

Signal:	strong_bt
Parameters:	1. boolean TRUE if strong PIN entered, FALSE if strong PIN dialog was cancelled
Description:	Signal emitted from UI when the strong (16 digit) BT PIN has been entered

Signal:	sim_pin
Parameters:	1. string SIM PIN code or empty string if cancelled 2. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the SIM PIN has been entered.

Bluetooth DBUS UI dialogs

```

/**
 * Example of use (command line):

dbus-send --system --print-reply \
  --dest='com.nokia.icd_ui' /com/nokia/bt_ui \
  com.nokia.bt_ui.show_send_file_dlg \
  array:string:file:///home/user/MyDocs/.documents/testing.txt

dbus-send --system --print-reply \
  --dest=com.nokia.bt_ui /com/nokia/bt_ui
  com.nokia.bt_ui.show_search_dlg \
  string: string: array:string: boolean:true
*/

#ifndef CONBTDIALOGS_DBUS_H
#define CONBTDIALOGS_DBUS_H

#ifdef __cplusplus
extern "C" {
#endif

/** Conbtdialogs service, resides in system dbus */
#define CONBTDIALOGS_DBUS_SERVICE "com.nokia.bt_ui"
/** Conbtdialogs interface */
#define CONBTDIALOGS_DBUS_INTERFACE "com.nokia.bt_ui"
/** Conbtdialogs path */
#define CONBTDIALOGS_DBUS_PATH "/com/nokia/bt_ui"

/**
 * Show send file dialog
 *
 * Arguments:
 *
 * uris: DBUS_TYPE_ARRAY          Array of strings representing
 *       the URIs of the          the files to send.
 *
 * Returns:

```

```

    DBUS_TYPE_BOOLEAN          TRUE, if dialog was shown successfully.
*/
#define CONBTDIALOGS_SEND_FILE_REQ          "show_send_file_dlg"

/**
    File sending result signal

    Arguments:

    success: DBUS_TYPE_BOOLEAN          TRUE, if all files were sent
            successfully or
            FALSE, if error occurred or sending was
            canceled.
*/
#define CONBTDIALOGS_SEND_FILE_SIG          "send_file"

/**
    Show BT device search dialog

    Arguments:

    major_class: DBUS_TYPE_STRING To set filtering based on major_class
            or
            "". Possible major class values are:
            "miscellaneous", "computer", "phone",
            "access point", "audio/video",
            "peripheral", "imaging", "wearable",
            "toy" and "uncategorized".

    minor_class: DBUS_TYPE_STRING To set filtering based on minor_class
            or "".
            Possible minor class values are:
            - Minor classes for "computer":
              "uncategorized", "desktop", "server",
              "laptop", "handheld", "palm", "
              wearable"
            - Minor classes for "phone": "
              uncategorized",
              "cellular", "cordless", "smart phone
              ",
              "modem", "isdn"

    service_classes: DBUS_TYPE_ARRAY To set filtering based on service
            classes.
            Supported classes include "
            positioning",
            "networking", "rendering", "
            capturing",
            "object transfer", "audio", "
            telephony",
            "information". Can be empty list,
            when no
            service class filtering is performed
            .

    bonding: DBUS_TYPE_STRING          Bonding mode for found and selected
            device:

```

```

        "require" for requiring a bonding from
        a
        selected device (i.e. bond device if it
        has not
        been bonded before).

        "force" to always bond (i.e. device
        will be
        bonded even if bonded before).

        Any other string will allow to search
        and
        select device without bonding it.

Returns:

    DBUS_TYPE_BOOLEAN        TRUE, if dialog was shown successfully.
*/
#define CONBTDIALOGS_SEARCH_REQ        "show_search_dlg"

/**
Bluetooth search result signal

Arguments:

    address: DBUS_TYPE_STRING        Bluetooth address of the selected
        device, or ""                if search dialog was cancelled.

    name: DBUS_TYPE_STRING        Name of the device.

    icon: DBUS_TYPE_STRING        Logical name for the icon describing
        the                        device.

    major_class: DBUS_TYPE_STRING Major class of the device.

    minor_class: DBUS_TYPE_STRING Minor class of the device.

    trusted: DBUS_TYPE_BOOLEAN    Defines whether the device is marked as
        a trusted device.

    services: DBUS_TYPE_ARRAY    List of strings describing the service
        classes                    and SDP-based services provided by the
                                    device.

*/
#define CONBTDIALOGS_SEARCH_SIG        "search_result"

```

Listing 9.6: conbtdialogs-dbus.h

```

/**
Bluetooth UI Library for maemo

Copyright (C) 2006 Nokia. All rights reserved.

This sample demonstrates the use of conbtdialogs API and especially
send_file function. Compile the program with conbtdialogs and dbus:

    gcc -Wall 'pkg-config --libs --cflags dbus-glib-1 conbtdialogs' \
    -o send_file conbtdialogs_send_file.c

```

```

Run with list of URLs:

./send_file file:///home/user/MyDocs/.sounds/Everyday.mp3
*/

#define DBUS_API_SUBJECT_TO_CHANGE

#include <glib.h>
#include <conbtdialogs-dbus.h>
#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>

DBusGConnection *connection = NULL;
GMainLoop *mainloop = NULL;

static gboolean initialize(void)
{
    GError *error = NULL;

    g_type_init ();

    /* Create main loop */
    mainloop = g_main_loop_new(NULL, TRUE);
    if ( mainloop == NULL ) return FALSE;

    /* Create DBUS connection */
    connection = dbus_g_bus_get(DBUS_BUS_SYSTEM, &error);

    if (connection == NULL )
    {
        g_print ("Error: %s\n", error->message);
        g_clear_error (&error);
        return FALSE;
    }

    return TRUE;
}

static gboolean uninitialize(void)
{
    /* Quit main loop and unref it */
    if (mainloop != NULL)
    {
        g_main_loop_quit(mainloop);
        g_main_loop_unref(mainloop);
    }

    return TRUE;
}

static DBusHandlerResult file_sent_signal ( DBusConnection *connection,
                                           DBusMessage *message,
                                           void *data )
{
    gboolean success = FALSE;

    /* check signal */
    if (!dbus_message_is_signal(message,

```

```

CONBTDIALOGS_DBUS_INTERFACE,
CONBTDIALOGS_SEND_FILE_SIG))

return DBUS_HANDLER_RESULT_NOT_YET_HANDLED;

/* get args */
if ( !dbus_message_get_args ( message, NULL,
                             DBUS_TYPE_BOOLEAN, &success,
                             DBUS_TYPE_INVALID ) )
    return DBUS_HANDLER_RESULT_NOT_YET_HANDLED;

/* print if file sending was success or failure */
g_print ( "File sending was a " );

if (success) g_print("success\n"); else g_print("failure\n");
dbus_connection_close(connection);
uninitialize();

return DBUS_HANDLER_RESULT_HANDLED;
}

gint main(gint argc, gchar **argv)
{
    GError *error = NULL;
    gchar **files = NULL;
    gint idx = 0;
    DBusGProxy *proxy;
    DBusConnection *sys_conn;
    gchar *filter_string = NULL;

    if (argc < 2) return 1;

    if (initialize() == FALSE) {
        uninitialize();
        return 1;
    }

    /* Copy urls to GLib compatible char array */
    files = g_new0(gchar*, argc);

    for (idx = 1; idx < argc; idx++)
        files[idx-1] = g_strdup(argv[idx]);

    files[argc-1] = NULL;

    /* Open connection for btdialogs service */
    proxy = dbus_g_proxy_new_for_name(connection,
                                      CONBTDIALOGS_DBUS_SERVICE,
                                      CONBTDIALOGS_DBUS_PATH,
                                      CONBTDIALOGS_DBUS_INTERFACE);

    /* Send send file request to btdialogs service */
    if (!dbus_g_proxy_call(proxy, CONBTDIALOGS_SEND_FILE_REQ,
                          &error,
                          G_TYPE_STRV, files, G_TYPE_INVALID,
                          G_TYPE_INVALID))
    {
        g_print("Error: %s\n", error->message);
        g_clear_error(&error);
        g_strfreev (files);
        g_object_unref(G_OBJECT(proxy));
        uninitialize();
    }
}

```

```

        return 1;
    }
    g_strfreev (files);
    files = NULL;

    g_object_unref(G_OBJECT(proxy));

    /* Now wait for file sent signal, use low level bindings as glib
       bindings require signal marshaller registered */
    sys_conn = dbus_bus_get(DBUS_BUS_SYSTEM, NULL);
    g_assert(dbus_connection_add_filter(sys_conn,
                                       file_sent_signal,
                                       NULL,
                                       NULL ));

    filter_string =
        g_strdup_printf ("type='signal',interface='%s'",
                        CONBTDIALOGS_DBUS_INTERFACE);

    dbus_bus_add_match(sys_conn, filter_string, NULL);
    dbus_connection_unref(sys_conn);

    /* Run mainloop */
    g_main_loop_run(mainloop);

    return 0;
}

```

9.2.6 Samba Network Shares

The device has support for connecting Samba network shares. To access Samba network shares with GnomeVFS, a Samba module for the GnomeVFS (libossgnomevfs2-samba) needs to be installed.

Here is a simple example on how to read a file via Samba:

```

#include <libgnomevfs/gnome-vfs.h>
#include <stdio.h>

int main(void)
{
    GnomeVFSHandle *handle = NULL;
    GnomeVFSResult res;
    GnomeVFSFileSize bytes_read;
    gchar buf[1024];

    // URI to open for reading
    gchar *uri = "smb://host/file.txt";

    if (!gnome_vfs_init()) {
        fprintf(stderr, "GnomeVFS initialization failed.\n");
        return 1;
    }

    if (GNOME_VFS_OK != (res = gnome_vfs_open(&handle, uri,
        GNOME_VFS_OPEN_READ))) {
        fprintf(stderr, "GnomeVFS open failed.\n");
        return 2;
    }

    while (GNOME_VFS_OK == res) {
        res = gnome_vfs_read(handle, buf, sizeof(buf)-1, &bytes_read);
    }
}

```

```

        buf[bytes_read] = 0;

        // Write buffer to stdout
        write(1, buf, bytes_read);

        if(bytes_read == 0)
            break;
    }

    if (GNOME_VFS_OK != gnome_vfs_close(handle))
        fprintf(stderr, "GnomeVFS close failed\n");

    gnome_vfs_shutdown();
    return 0;
}

```

Example can be compiled including GnomeVFS:

```
gcc 'pkg-config gnome-vfs-2.0 --libs --cflags' vfscat.c -o vfscat
```

9.2.7 Location

Location framework has one library; liblocation. The liblocation is made up of functions for parsing GPSD's output, for controlling GPSD and for other helper functions.

Using Liblocation

The headers for liblocation are stored in the location subdirectory, and so they should be included as follows.

```
#include <location/location-gps-device.h>
```

Listening to GPSD

GPSD is used in maemo to talk to GPS devices and report position data. Liblocation contains an object that listens to GPSD and converts the output into GObject signals. Creating the object goes as follows

```
LocationGPSDevice *device;

device = g_object_new (LOCATION_TYPE_GPS_DEVICE, NULL);
```

It is then possible to connect to the changed signal to hear about gps fix changes.

```
g_signal_connect (device, "changed", G_CALLBACK (location_changed),
    NULL);
```

And the location changed callback looks like this

```
static void
location_changed (LocationGPSDevice *device, gpointer userdata)
{
    g_print ("Latitude: %.2f\nLongitude: %.2f\nAltitude: %.2f\n",
        device->fix->latitude, device->fix->longitude, device
        ->fix->altitude);
}
```

That is all that is required for a simple GPS client that listens for GPSD. N.B. The LocationGPSTDevice object has some public fields. They are as follows:

- device->online: Whether GPSD is connected to a GPS
- device->status: The GPS status
- device->fix: The GPS fix information
- device->satellites_in_view: The number of satellites that the GPS can see
- device->satellites_in_use: The number of satellites used in fix calculation
- device->satellites: An array of satellite details

The most important of these is device->fix, as this contains the fix location data from GPSD. The fields of the LocationGPSTDeviceFix structure are fairly self-explanatory except for the fields field. This field is a bitmask of what other fields in the structure have valid content. To check whether the latitude and longitude fields are valid, you would do

```
if (fix->fields & LOCATION_GPS_DEVICE_LATLONG_SET)
```

and so on. The bitmask flags are defined in location-gps-device.h

Controlling GPSD

Sometimes a program may need to start or stop GPSD, or find out when GPSD has been started or stopped. This is accomplished using the LocationGPSTDeviceControl object. This object can only be created once in a program, so it is obtained with the location_gpsd_control_get_default() function (see location-gpsd-control.h).

```
LocationGPSTDeviceControl *control;  
  
control = location_gpsd_control_get_default ();
```

There are three signals on this object: error, gpsd_running and gpsd_stopped. Error is emitted when there is an error starting GPSD, gpsd_running is emitted whenever GPSD starts up, and gpsd_stopped is emitted when GPSD stops. Only one application is able to control GPSD at a time, and this will be the application that initially starts it. The control->can_control field will be TRUE if the application can control it, or FALSE if it cannot.

Other Liblocation Functions

Liblocation also comes with a function for getting the distance between two points. This is called the great-circle distance (see [Wikipedia](#) for more details on it). The function location_distance_between() (see location-distance-utils.h) takes the latitude and longitude of two locations and returns the distance between them in kilometers.

```
g_print ("distance between LAX and BNA is %fkm\n",  
        location_distance_between (36.12, -86.67, 33.94, -118.40));
```


Compiling Programs With LibLocation Support

Liblocation comes with a pkgconfig file, so adding support to a program is just a case of adding liblocation to the configure scripts PKG_CHECK_MODULES macro, as follows:

```
PKG_CHECK_MODULES (LOCATION)
AC_SUBST (LOCATION_CFLAGS)
AC_SUBST (LOCATION_LIBS)
```

Then in the Makefile.am, these variables need to be added to the program's CFLAGS and LDADD flags.

9.3 Implementing Custom Connection Managers

This section introduces briefly how to implement a custom connection manager plug-in for the Internet Tablet OS. The messaging framework in Internet Tablet OS is based on [Telepathy framework architecture](#).

Telepathy provides [D-Bus](#)-based framework that unifies all forms of real-time communication, such as instant messaging, IRC, voice and video over Internet. The framework provides an interface for plug-ins to extend the protocol support by implementing new connection managers. These new supported protocols can then be used by all client applications that communicate via [Telepathy framework architecture](#).

For more detailed information and source code examples, see the Telepathy framework [specification](#) by [Freedesktop.org](#).

Since the connection manager uses D-Bus for communication, it can be implemented using any language supporting D-Bus communication, even an interpreted language such as Python. However, in order to enable running on Internet tablets natively, C or C++ is currently preferred.

9.3.1 Connection Manager Implementation Examples

Several open source Telepathy connection manager implementations already exist. See, for instance, [Gabble](#) and [Idle](#) projects. In maemo 4.x, there is a source package of telepathy-gabble that has been modified for maemo platform and packaged as a DEB package.

The package can be downloaded from the 4.x repository. The package can be built the usual way with dpkg-buildpackage.

9.3.2 Connection Manager and Connections

Telepathy connection managers establish the actual connections to IM or VoIP servers. A connection manager provides support for one or more connection protocols, e.g. SIP. Connection managers are started using D-Bus service activation, and they present a connection manager object to the bus. A connection can be established by sending a D-Bus message request to the connection manager object. A new D-Bus object is then created to handle the new connection. The D-Bus interface of a connection manager is:

```

org.freedesktop.Telepathy.ConnectionManager
Methods:
GetParameters ( s: proto ) -> a(susv)
ListProtocols ( ) -> as
The following well-known values should be used when applicable:

    * aim - AOL Instant Messenger
    * gadugadu - Gadu-Gadu
    * groupwise - Novell Groupwise
    * icq - ICQ
    * irc - Internet Relay Chat
    * jabber - Jabber (XMPP)
    * msn - MSN Messenger
    * napster - Napster
    * silc - SILC
    * sip - Session Initiation Protocol (SIP)
    * trepia - Trepia
    * yahoo - Yahoo! Messenger
    * zephyr - Zephyr

RequestConnection ( s: proto, a{sv}: parameters ) -> so
Signals:
NewConnection ( s: bus_name, o: object_path, s: proto )
Sets of flags:
Conn_Mgr_Param_Flags

```

The connection object provides the interfaces for basic connection signaling as well as for requesting communication channels. The D-Bus interface of a connection is:

```

org.freedesktop.Telepathy.Connection
Methods:
Connect ( ) -> None
Disconnect ( ) -> None
GetInterfaces ( ) -> as
GetProtocol ( ) -> s
GetSelfHandle ( ) -> u
GetStatus ( ) -> u
HoldHandles ( u: handle_type, au: handles ) -> None
InspectHandles ( u: handle_type, au: handles ) -> as
ListChannels ( ) -> a(osuu)
ReleaseHandles ( u: handle_type, au: handles ) -> None
RequestChannel ( s: type, u: handle_type, u: handle, b: suppress_handler ) -> o
RequestHandles ( u: handle_type, as: names ) -> au
Signals:
NewChannel ( o:object_path, s:channel_type, u:handle_type, u:handle, b:suppress_handler)
StatusChanged ( u: status, u: reason )

```

9.3.3 Channels and Channel Types

Communication with the server and other contacts is carried out with instances of various channel types. The interface for creating, closing and handling channels is the following:

```

org.freedesktop.Telepathy.Channel
Methods:
Close ( ) -> None
GetChannelType ( ) -> s
GetHandle ( ) -> uu
GetInterfaces ( ) -> as
Signals:
Closed ( )

```

Various channel types are supported in the Telepathy framework to match the needs of the real-time communication protocol. For example, a text channel provides methods to send messages, and is able to send signals to indicate that

messages have been sent and received.

```
org.freedesktop.Telepathy.Channel.Type.ContactList

org.freedesktop.Telepathy.Channel.Type.ContactSearch
Methods:
GetSearchKeys ( ) -> sa{s(bg)}
GetSearchState ( ) -> u
Search ( a{sv}: terms ) -> None
Signals:
SearchResultReceived ( u: contact, a{sv}: values )
SearchStateChanged ( u: state )

org.freedesktop.Telepathy.Channel.Type.StreamedMedia
Methods:
ListStreams ( ) -> a(uuuuuu)
RemoveStreams ( au: streams ) -> None
RequestStreamDirection ( u: stream_id, u: stream_direction ) -> None
RequestStreams ( u: contact_handle, au: types ) -> a(uuuuuu)
Signals:
StreamAdded ( u: stream_id, u: contact_handle, u: stream_type )
StreamDirectionChanged ( u: stream_id, u: stream_direction, u: pending_flags )
StreamError ( u: stream_id, u: errno, s: message )
StreamRemoved ( u: stream_id )
StreamStateChanged ( u: stream_id, u: stream_state )

org.freedesktop.Telepathy.Channel.Type.RoomList
Methods:
GetListingRooms ( ) -> b
ListRooms ( ) -> None
Signals:
GotRooms ( a(usa{sv}): rooms )
ListingRooms ( b: listing )

org.freedesktop.Telepathy.Channel.Type.Text
Methods:
AcknowledgePendingMessages ( au: ids ) -> None
GetMessageTypes ( ) -> au
ListPendingMessages ( b: clear ) -> a(uuuuus)
Send ( u: type, s: text ) -> None
Signals:
LostMessage ( )
Received ( u: id, u: timestamp, u: sender, u: type, u: flags, s: text )
SendError ( u: error, u: timestamp, u: type, s: text )
Sent ( u: timestamp, u: type, s: text )
```

9.3.4 Additional Connection Interfaces

The connection interfaces can handle special needs of the connection protocol, such as aliasing, which means that contacts can have an alias that they can change via the server. The D-Bus interfaces are briefly presented below. More details can be found in the [specification](#).

```

org.freedesktop.Telepathy.Connection.Interface.Aliasing
Methods:
GetAliasFlags ( ) -> u
RequestAliases ( au: contacts ) -> as
SetAliases ( a{us}: aliases ) -> None
Signals:
AliasesChanged ( a(us): aliases )

org.freedesktop.Telepathy.Connection.Interface.Avatars
Methods:
GetAvatarRequirements ( ) -> asqqqqq
GetAvatarTokens ( au: contacts ) -> as
RequestAvatar ( u: contact ) -> ays
SetAvatar ( ay: avatar, s: mime_type ) -> s
Signals:
AvatarUpdated ( u: contact, s: new_avatar_token )

org.freedesktop.Telepathy.Connection.Interface.Capabilities
Methods:
AdvertiseCapabilities ( a(su): add, as: remove ) -> a(su)
GetCapabilities ( au: handles ) -> a(usuu)
Signals:
CapabilitiesChanged ( a(usuuuu): caps )

org.freedesktop.Telepathy.Connection.Interface.ContactInfo
Methods:
RequestContactInfo ( u: contact ) -> None
Signals:
GotContactInfo ( u: contact, s: vcard )

org.freedesktop.Telepathy.Connection.Interface.Forwarding
Methods:
GetForwardingHandle ( ) -> u
SetForwardingHandle ( u: forward_to ) -> None
Signals:
ForwardingChanged ( u: forward_to )

org.freedesktop.Telepathy.Connection.Interface.Presence
Methods:
AddStatus ( s: status, a{sv}: parms ) -> None
ClearStatus ( ) -> None
GetStatuses ( ) -> a{s(ubba{ss})}
RemoveStatus ( s: status ) -> None
RequestPresence ( au: contacts ) -> None
SetLastActivityTime ( u: time ) -> None
SetStatus ( a{sa{sv}}: statuses ) -> None
Signals:
PresenceUpdate ( a{u(ua{sa{sv}})}: presence )

org.freedesktop.Telepathy.Connection.Interface.Privacy
Methods:
GetPrivacyMode ( ) -> s
GetPrivacyModes ( ) -> as
SetPrivacyMode ( s: mode ) -> None
Signals:
PrivacyModeChanged ( s: mode )

org.freedesktop.Telepathy.Connection.Interface.Renaming
Methods:
RequestRename ( s: name ) -> None
Signals:
Renamed ( u: original, u: new )

```

9.4 Using STUN in Applications

This section shows how to use the *libjingle* API to create peer-to-peer connections between parties that are behind NAT routers.

9.4.1 Network Address Translation (NAT)

NAT routers were born because of the imminent IP address exhaustion. The currently most used Internet address family, the Internet Protocol version 4, is 32 bits wide, meaning roughly four billion available addresses.

Even though four billion addresses may seem like an abundant resource, not all addresses can be assigned to hosts, just as not all telephone number permutations can be assigned to users. IP addresses also carry routing information, analogous to the prefix of a telephone number that identifies country, region, city, etc.

The definitive solution for the problem is to increase the address length. For this, IPv6 was designed, introducing 128-bit addresses. Adopting new address families cannot be done over night, so an intermediate solution was found: NAT - Network Address Translation.

A NAT router acts like a switchboard between the public Internet and a private network. The NAT router needs to have at least one valid Internet address in order to be "seen" by the rest of the Internet. The computers in the private network have private address in the ranges 10.0.0.0/8, 192.168.0.0/16 or 172.16.0.0/12, which are not routable in the Internet.

When a private computer, e.g. address 10.0.0.1, tries to connect to a public server 200.215.89.79, the network packet must pass through the NAT router. The NAT router knows that the source address 10.0.0.1 is not routable and replaces 10.0.0.1 with its own valid address (e.g. 64.1.2.3), and sends the packet forward.

The remote server 200.215.89.79 will receive a connection from the host 64.1.2.3, and will reply to that host.

When the response packet comes to the NAT router, it must have a way of telling whether the packet is meant for the router itself or a host in the private network. Once the NAT router resolves the packet to the active connection, it changes the destination address from 64.1.2.3 to 10.0.0.1, and delivers the packet to the correct recipient in the private network.

In more technical depth, NAT is possible because absolutely every network connection on the Internet has a unique tuple consisting of the following values:

- Client address (the host that initiates the connection)
- Server address (the passive side that receives the initiation packet)
- Client port number
- Server port number
- The transport protocol e.g. TCP, UDP, SCTP.

If the NAT router replaces the client address with its own, but also replaces the client port number when necessary to avoid a clash with another active connection, the uniqueness of each connection is retained. NAT allows for a virtually unlimited number of computers in a private network to access the Internet via only one NAT router with one public IP address.

9.4.2 NAT Problems

There are some disadvantages in the NAT system. Firstly, the NAT router needs to keep connection states in memory, which partially breaks a cornerstone in Internet philosophy ("dumb routers, smart hosts"). If a NAT router is reset, all connections will be terminated, while a regular router could be reset without breaking any connection.

Secondly, the hosts in a private network cannot easily provide a service to the public Internet, i.e. these hosts cannot be the "passive" side in connections, since the initiation packets will come to the NAT router and it will have no related connection in its memory.

A partial solution for that problem is to open a "hole" in the NAT for specific ports, for example any connection to the port 8000 of the NAT router should be redirected to the machine 10.0.0.2 port 80. It works, but it does not scale up. The number of ports is limited, some protocols work only on a very specific port numbers, and each port requires manual configuration of the NAT router. Manual configuration is a solution only when there are few protocols and users.

Since the bulk of the Internet traffic is HTTP and initiated from the private network, NATs are adequate for most needs.

9.4.3 NAT Peer-to-Peer Circumvention Techniques

The most problematic services to deploy in presence of NATs are peer-to-peer (P2P) applications, where two clients of the service make direct connections to each other without sending data through an intermediate server. This is a problem for SIP-based VoIP and most P2P file sharing networks.

If one of the P2P parties has a routable IP address, the problem is easily solved: the party behind the NAT router must initiate the connection. Unfortunately, the most common case is where both the P2P parties are behind NAT routers.

Several techniques have been proposed to solve this issue. The most elegant solutions require both software updates in the NAT router itself, and explicit support in the client software. One of these solutions is the UPnP IGD (Internet Gateway Device) protocol. This protocol gives a host means to request to open a hole in the NAT router to make a service accessible from the Internet.

IGD is easy to use and has been enjoying good support from NAT router vendors, but it is still far from ubiquitous, and does not work behind two or more NAT routers, which is a common situation.

It is important to remember that none of the existing NAT circumvention techniques, not even the most elegant, can completely solve the problem. There still has to be a signaling protocol for the parties to exchange connection parameters. In other words, it does not suffice to be able to open a hole in the NAT router; there must be a way to *communicate* the address and the port of the hole to the remote party. A pure P2P service can only be achieved when all parties have public addresses. This is expected to happen when IPv6 is widely deployed.

9.4.4 STUN, TURN and ICE Protocols

There are NAT circumvention techniques that do not require router software updates. STUN (Simple Transversal of UDP through NATs) is the most common one. STUN exploits the connectionless nature of UDP, as well as a security weakness of some NAT implementations. The technique is bound to UDP features; hence STUN cannot be used for example for TCP connections.

The STUN protocol requires a STUN server with a well-known public IP address on the Internet. STUN sends the private address and port in the payload of a UDP datagram to the STUN server. If the packet goes through a NAT router, the address and port are changed in the IP headers, while the ones in the payload remain the same.

The STUN server returns the actual address and port to the sender and the client can resolve the type of NAT, if any, from the response. Some NATs have poor implementations which, in conjunction with STUN, allow for incoming UDP flow. These are called full cone NATs. They always translate the same source address and port tuple to the same NAT router port. This relieves the router from storing full connection state. Any packet coming from the Internet to a router port will be delivered to the private host, despite its source.

This allows for the private host to easily punch a hole in the NAT router. The first packet going to the STUN server opens the hole. In turn, the private host learns from the STUN server the IP address and port number of the hole. These parameters are then sent to the remote peer via a signaling protocol. The remote peer can then send UDP data directly.

Unfortunately, most NAT routers are not that naive; they are "symmetric", i.e. they store full connection state, and do not allow incoming packets from anyone but the party that was first contacted (in this case, the STUN server). In this scenario, STUN is not enough to allow P2P communication in the presence of NAT.

TURN (Traversal Using Relay NAT) comes to rescue. TURN employs the same protocol as STUN, but the TURN server acts as a relay to which both parties behind NATs can connect. Since the relay server adds latency and needs to be maintained (obviously for a cost), TURN should be used only as the last resort.

ICE (Interactive Connectivity Establishment), a draft specification that is employed by Google Talk, is not a protocol in itself. It is a collection of techniques like STUN and TURN. It finds all the possible ways to establish a P2P connection, and picks the best one.

ICE works by finding all possible P2P connection candidates, and sending this data to the remote peer via a signaling protocol. The signaling protocol is not specified by ICE; hence the mechanism can be used with any signaling protocol. The parties agree on the best way to initiate the connection, analogous to a modem handshake. The signaling server works as a proxy until the clients can start to exchange data directly.

The biggest advantage of STUN, TURN and ICE is that they do not depend on support by the NAT routers. ICE will work even if there are several routers in the network path. The downside is the need of public STUN and TURN (relay) servers. This is not a great problem, due to the fact that every P2P service requires a signaling server anyway. The same entity that provides the signaling will certainly provide the auxiliary STUN/TURN services.

Another minor disadvantage of ICE is that the signaling protocol will have to accommodate the "connection candidate" message, either by an explicit provision in the protocol (e.g. XMPP) or by a hack.

9.4.5 NAT Transversal API in Maemo

On maemo platform, the developer does not need to worry about these protocols. Maemo includes the libjingle library (used by Google Talk) that offers an API for P2P connections.

The best way to learn to use the API is by example. The example P2P client here is very simple: it requests a service at random times, and processes requests from other P2P clients. The service in this example is nothing more than adding two byte values.

As already stated, every P2P server must have a signaling protocol, over which the parties can exchange initial P2P connection parameters. What is needed for this is a very simple server and the signaling protocol. Since this is just an example, the server architecture does not attribute IDs for the clients, and therefore can handle only two simultaneous clients.

The signaling protocol is very simple and has only one message: the connection candidate that one peer sends to the other. The message is simply forwarded to the other peer. Apart from encoding and decoding, these messages are completely handled by *libjingle*, so it is not necessary to understand them in depth.

Once the connection parameters have been exchanged via the signaling server, the P2P connection is opened, and the parties will communicate directly without any further signaling. Albeit very simple, this protocol simulates all the basic steps of every real-world P2P service.

If interested in using XMPP/Google Talk as the signaling service, Libjingle source also contains examples of P2P communication that employ XMPP/-Google Talk accounts and servers.

9.4.6 Example: P2P Client

The following C example contains some C++, because Libjingle is written in C++. The example is a P2P client based on Libjingle APIs. It is the smallest possible demonstration of NAT piercing capability, so it does not handle network errors and overflows very well. A production implementation must improve in these directions.

First, there is some boilerplate code: includes and prototypes.

```
#define POSIX
#define SIGSLOT_USE_POSIX_THREADS

#include <libjingle/talk/base/thread.h>
#include <libjingle/talk/base/network.h>
#include <libjingle/talk/base/socketaddress.h>
#include <libjingle/talk/base/physicalsocketserver.h>
#include <libjingle/talk/p2p/base/sessionmanager.h>
#include <libjingle/talk/p2p/base/helpers.h>
#include <libjingle/talk/p2p/client/basicportallocator.h>
#include <libjingle/talk/p2p/client/socketclient.h>

#include <string>
```



```

#include <vector>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>
#include <signal.h>

void signaling_init(const char* ip, unsigned int port);
int signaling_wait(unsigned int timeout);
void signaling_sendall(const char* buffer, unsigned int len);

SocketClient* socketclient_init(const char *stun_ip, unsigned int
    stun_port,
                                const char *turn_ip, unsigned int
                                turn_port);

char* socketclient_add_remote_candidates(SocketClient *sc, char* buffer
);
void socketclient_add_remote_candidate(SocketClient *sc, const char *
candidate);

bool socketclient_is_writable (SocketClient *sc);

void socketclient_send(SocketClient *sc, const char *data, unsigned int
len);

void randomize();

```

For the sake of simplicity, some data is kept in global variables. A production implementation would probably move that data into objects.

The *p2p_state* shows, whether the P2P connection is up. *signaling_socket* is a TCP socket, allowing data exchange via the signaling server before the P2P connection is up. *main_thread* contains a libjingle *Thread* object; libjingle is itself multithreaded, and employs one thread per P2P connection.

```

bool p2p_state = false;
int signaling_socket = -1;

cricket::Thread *main_thread = 0;

```

This is the main program loop. It sets up the signaling connection and forwards the signaling data to Libjingle until the P2P connection is up. The P2P connection is simply used to send bytes at random intervals. If the P2P connection breaks, the loop returns to signaling phase. The program only stops when killed or when an unexpected error occurs.

N.B. *main_thread->Loop(10)* is called from time to time. In a "real" application, this method would be called on idle time (e.g. via GLib's *g_idle_add()*).

There are three IP addresses hardcoded: signaling server, STUN server (if any) and TURN server (if any). These addresses should be dated for the particular environment in question.

```

int main(int argc, char* argv[])
{
    signal(SIGPIPE, SIG_IGN);

```

```

randomize();

// P2P signaling server
const char* signaling_ip = "200.184.118.140";
int signaling_port = 14141;

// STUN server, NULL if none
const char* stun_ip = "200.184.118.140";
// const char* stun_ip = 0;
unsigned int stun_port = 7000;

// TURN server, NULL if none
const char* turn_ip = 0;
// const char* turn_ip = 0;
unsigned int turn_port = 5000;

signaling_init(signaling_ip, signaling_port);

SocketClient* sc = socketclient_init(stun_ip, stun_port,
    turn_ip, turn_port);

sc->getSocketManager()->StartProcessingCandidates();

while (1) {
    char buffer[10000];
    char *buffer_p = buffer;
    char *buffer_interpreted = buffer;

    while (! p2p_state) {
        main_thread->Loop(10);

        if (! signaling_wait(1)) {
            printf("-- tick --\n");
            continue;
        }

        int n = recv(signaling_socket, buffer_p, sizeof
            (buffer)
            - (buffer_p - buffer), 0);
        if (n < 0) {
            printf("Signaling socket closed with
                error\n");
            exit(1);
        } else if (n == 0) {
            printf("Signaling socket closed\n");
            exit(1);
        }
        buffer_p += n;
        buffer_interpreted =
            socketclient_add_remote_candidates(sc,
                buffer_interpreted);
    }

    // P2P connection is up by now.

    while (p2p_state) {
        // sends a byte via P2P connection
        unsigned char data = random() % 256;
        socketclient_send(sc, (char*) &data, 1);
        sleep(random() % 15 + 1);
        main_thread->Loop(10);
    }
}

```

```

        // P2P connection is broken, restart handling
        // connection candidates
    }
}

```

The next function seeds random, otherwise the two peers may end up with exactly the same bytes and time intervals during P2P data exchange.

```

void randomize()
{
    struct timeval tv;
    struct timezone tz;
    gettimeofday(&tv, &tz);
    srand(tv.tv_usec);
}

```

This function creates the signaling socket - just a boring and ordinary TCP connection to the P2P signaling server.

```

void signaling_init(const char* ip, unsigned int port)
{
    struct sockaddr_in sa;

    signaling_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(port);
    inet_aton(ip, &(sa.sin_addr));

    if (connect(signaling_socket, (struct sockaddr*) &sa, sizeof(sa)) < 0) {
        printf("Error in signaling connect()\n");
        exit(1);
    }
}

```

This function waits for something to happen with the signaling socket, until the specified timeout. It is important that the timeout is not too long, since the P2P connection may have been brought up meanwhile.

```

int signaling_wait(unsigned int timeout) {
    fd_set rfd;
    struct timeval tv;
    int retval;

    FD_ZERO(&rfd);
    FD_SET(signaling_socket, &rfd);

    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    retval = select(signaling_socket+1, &rfd, NULL, NULL, &tv);
    if (retval == -1)
        printf("error in select()");

    return (retval > 0);
}

```

Libjingle is C++ and employs a signal architecture to notify the client application about changes in P2P connection status. These two classes accommodate the methods that will be called back when something happens.

Since XMPP is not used as the signaling protocol, the signaling protocol handling needs to be provided, so all signaling handling is carried out separately from those signals.

```
class SignalListener1 : public sigslot::has_slots<>
{
private:
    SocketClient *sc;

public:
    SignalListener1(SocketClient *psc);
    void OnCandidatesReady(const std::vector<Candidate>& candidates);
    void OnNetworkError();
    void OnSocketState(bool state);
};

class SignalListener2 : public sigslot::has_slots<>
{
private:
    SocketClient *sc;

public:
    SignalListener2(SocketClient *psc);
    void OnSocketRead(P2PSocket *socket, const char *data, size_t len);
};

SignalListener1::SignalListener1(SocketClient* psc)
{
    sc = psc;
}

SignalListener2::SignalListener2(SocketClient* psc)
{
    sc = psc;
}

void SignalListener1::OnNetworkError()
{
    printf ("Network error encountered at SocketManager");
    exit(1);
}
```

The first signal callback method. It is called when the P2P socket changes state. The *p2p_state* global variable will be updated with the reported state, and this will drive the main loop behavior.

```
void SignalListener1::OnSocketState(bool state)
{
    printf("Socket state changed to %d\n", state);
    p2p_state = state;
    if (state) {
        printf("Writable from %s:%d to %s:%d\n",
            sc->getSocket()->best_connection()->local_candidate().address().
            IPAsString().c_str(),
            sc->getSocket()->best_connection()->local_candidate().address().
            port(),
```

```

        sc->getSocket()->best_connection()->remote_candidate().address().
            IPAsString().c_str(),
        sc->getSocket()->best_connection()->remote_candidate().address().
            port());
    }
}

```

This function packages all P2P socket creation bureaucracy. It creates the socket object, the socket listeners (whose classes have been defined above), and connects the signal callbacks.

```

SocketClient* socketclient_init(const char *stun_ip, unsigned int
    stun_port,
                                const char *turn_ip, unsigned int
                                turn_port)
{
    cricket::SocketAddress *stun_addr = NULL;
    if (stun_ip) {
        stun_addr = new cricket::SocketAddress(std::string(
            stun_ip), stun_port);
    }

    cricket::SocketAddress *turn_addr = NULL;
    if (turn_ip) {
        turn_addr = new cricket::SocketAddress(std::string(
            turn_ip), turn_port);
    }

    cricket::PhysicalSocketServer *ss = new PhysicalSocketServer();
    main_thread = new Thread(ss);
    cricket::ThreadManager::SetCurrent(main_thread);

    SocketClient *sc = new SocketClient (stun_addr, turn_addr);

    // Note that signal connections pass the SignalListener1 object
    as well as the
    // method. Since a new SocketListener1 is created for every new
    SocketClient,
    // we have the guarantee that each SocketListener1 will be
    called back only
    // in behalf of its related SocketClient.

    sc->sigl1 = new SignalListener1(sc);
    sc->sigl2 = new SignalListener2(sc);
    sc->getSocketManager()->SignalNetworkError.connect(sc->sigl1,
        &SignalListener1::OnNetworkError);
    sc->getSocketManager()->SignalState_s.connect(sc->sigl1,
        &SignalListener1::OnSocketState);
    sc->getSocketManager()->SignalCandidatesReady.connect(sc->sigl1
        ,
        &SignalListener1::OnCandidatesReady);
    sc->CreateSocket(std::string("foobar"));
    sc->getSocket()->SignalReadPacket.connect(sc->sigl2,
        &SignalListener2::
            OnSocketRead);

    return sc;
}

```

The method below is called back when LibJingle has some local candidates for connection that should be sent to the remote site via the signaling protocol.

The beauty of ICE protocol is that both parties will be able to agree on a P2P connection, without having to exchange request and response messages. They just send connection candidates to each other. Each side selects the best way to send data, based on the received candidates. With both sides able to send data directly to the remote party, a bi-directional P2P channel is enabled.

```
void Signallistener1::OnCandidatesReady(const std::vector<Candidate>&
candidates)
{
    printf("OnCandidatesReady called with %d candidates in list\n",
        candidates.size());

    for(std::vector<Candidate>::const_iterator it = candidates.
        begin();
        it != candidates.end(); ++it) {
        char *marshaled_candidate;

        asprintf(&marshaled_candidate,
            "%s %d %s %f %s %s %s\n",
            (*it).address().IPAsString().c_str(),
            (*it).address().port(),
            (*it).protocol().c_str(),
            (*it).preference(),
            (*it).type().c_str(),
            (*it).username().c_str(),
            (*it).password().c_str() );

        printf("Candidate being sent: %s", marshaled_candidate)
            ;

        signaling_sendall(marshaled_candidate, strlen(
            marshaled_candidate));

        free(marshaled_candidate);
    }
}
```

An auxiliary function to send a data buffer through the signaling TCP channel. It does not return until all data has been sent.

```
void signaling_sendall(const char* buffer, unsigned int len)
{
    unsigned int sent = 0;
    while (sent < len) {
        int just_sent;
        just_sent = send(signaling_socket, buffer+sent, len-
            sent, 0);
        if (just_sent < 0) {
            printf("Signaling socket closed with error.\n")
                ;
            exit(1);
        } else if (just_sent == 0) {
            printf("Signaling socket closed.\n");
            exit(1);
        }
        sent += just_sent;
    }
}
```

This function is called by the main loop, when some signaling data arrives.

It will find out if there is a complete P2P connection candidate in the buffer. If there is one, it is decoded.

```
// extracts remote candidates from a buffer, returns a pointer to the  
rest of the buffer  
  
char* socketclient_add_remote_candidates(SocketClient *sc, char* buffer  
    )  
{  
    char *n;  
    char candidate[1024];  
  
    while (1) {  
        n = strchr(buffer, '\n');  
        if (! n) {  
            return buffer;  
        }  
        strncpy(candidate, buffer, n-buffer+1);  
        socketclient_add_remote_candidate(sc, candidate);  
        buffer = n+1;  
    }  
}
```

Here, the P2P connection candidate is decoded and made known to LibJingle. Since LibJingle has its own thread and sockets, all further processing of P2P candidates is fortunately outside the scope of our code.

```
// Inform candidates received from the signaling network to LibJingle  
  
void socketclient_add_remote_candidate(SocketClient *sc, const char*  
    remote_candidate)  
{  
    std::vector<Candidate> candidates;  
  
    char ip[100];  
    unsigned int port;  
    char protocol[100];  
    float preference;  
    char type[100];  
    char username[100];  
    char password[100];  
  
    // WARNING: using fixed-size buffers and sscanf is utterly  
unsafe.  
    // Real implementations must be more robust about data coming  
from the network!  
  
    sscanf(remote_candidate,  
        "%s %d %s %f %s %s %s\n",  
        ip,  
        &port,  
        protocol,  
        &preference,  
        type,  
        username,  
        password);  
  
    printf("Received new candidate: %s:%d pref %f\n", ip, port,  
        preference);  
  
    Candidate candidate;  
    candidate.set_name("rtp");
```

```

        candidate.set_address(SocketAddress(std::string(ip), port));
        candidate.set_username(std::string(username));
        candidate.set_password(std::string(password));
        candidate.set_preference(preference);
        candidate.set_protocol(protocol);
        candidate.set_type(type);
        candidate.set_generation(0);

        candidates.push_back(candidate);

        sc->getSocketManager()->AddRemoteCandidates(candidates);
    }

```

Simple helper function, showing whether a P2P socket is writable (which means that the P2P connection is up).

```

bool socketclient_is_writable(SocketClient *sc)
{
    return sc->getSocketManager()->writable();
}

```

Method that is called back when data arrives from the P2P connection. In this example, it is just a byte of data.

```

void SignalListener2::OnSocketRead(P2PSocket *socket, const char *data,
    size_t len)
{
    printf("Received byte %d from remote P2P\n", data[0]);
}

```

Auxiliary function that sends data via P2P connection. Not really difficult.

```

void socketclient_send(SocketClient* sc, const char *data, unsigned int
    len)
{
    sc->getSocket()->Send(data, len);
    printf("Sent byte %d to remote P2P\n", data[0]);
}

```

In order to compile this, you need jinglebase-0.3 and jinglep2p-0.3 packages:

```

gcc client.cpp -o client `pkg-config --cflags --libs jinglebase-0.3 jinglep2p-0.3`

```

9.4.7 Signaling Server

As already mentioned, this signaling server is incredibly simple, and works more like a network pipe, forwarding data from one side to another. The P2P parties agree on a P2P connection through this channel.


```

from select import select
import socket
import time

read_socks = []
port = 14141

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.bind("", port)
server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_sock.listen(5)

read_socks.append(server_sock)

# We accept two parties only
buffer = ""

while True:
    rd, wr_dummy, ex_dummy = select(read_socks, [], [], 10)

    if not rd:
        print "-- tick --"
        continue

    rd = rd[0]

    if rd is server_sock:
        # incoming new connection
        newsock, address = rd.accept()
        print "New connection from %s" % str(address)
        if len(read_socks) > 3:
            # we only accept two parties at the most
            newsock.close()
            continue
        read_socks.append(newsock)
        if buffer:
            # we already have data to be sent to the new party
            newsock.sendall(buffer)
            print "    sent buffered data"
            buffer = ""
        continue

    data = rd.recv(999999)

    if not data:
        # socket closed, remove from list
        print "Connection closed"
        del read_socks[read_socks.index(rd)]
        buffer = ""
        continue

    if len(read_socks) < 3:
        print "Buffering data"
        # the other party has not connected; bufferize
        buffer += data
        continue

    print "Forwarding data"
    for wr in read_socks:
        if wr is not rd and wr is not server_sock:
            wr.sendall(data)

```

9.4.8 STUN and Relay Servers

The maemo libjingle-utils package includes both a STUN server and a relay server for testing purposes. To run a test server, do the following:

```

$ stunserv &
$ relayserver &

```

The relay server will print console messages, when a P2P connection is

flowing through it. If more detailed feedback is needed (e.g. when debugging a P2P application), a network sniffing tool like tcpdump or Ethereal should be used to monitor UDP packets.

Chapter 10

Customizing User Interface

10.1 Introduction

The following code examples are used in this chapter:

- [dummy-package](#)
- [puro-background](#)

This chapter intends to show how to customize themes, sounds, icons and wallpapers on maemo, and how basic customization can be done.

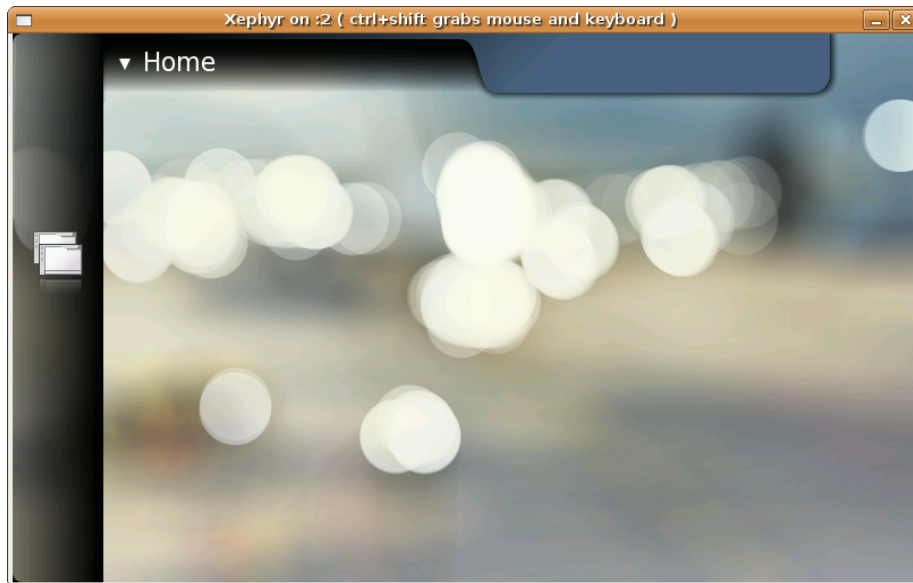
This chapter is for designers interested in customizing maemo, and developers interested in knowing how theme creation works.

Pre-requisites:

- Familiarity with maemo platform
- Familiarity with package installation
- Ability to connect to the device via ssh
- For manual customization, X-terminal and/or openssh installed on the device and device on R&D mode
- Drawing program capable of editing svg-files, e.g. Inkscape or a drawing program capable of editing png-files

Maemo supports various kinds of customization. Theme, icons, sounds, font and other things can be changed. Some customizations are simpler than others, one of the simplest customization currently being drawing an own theme, which is supported with theming tools. Theme means basically the set of graphics that represent the look of the device - in other words, pictures that are placed on top of the widgets, panels, titlebars etc. to decorate them to a desired look (without theme, gtk user interface would look very plain with no fancy eye-candy at all). In maemo, the theming is a bit different from Desktop Gnome theming, because the maemo themes heavily rely on heavy use of bitmap images, whereas in standard Gnome desktop, there are less themeable components.

Plankton development theme as represented on maemo 4.x on Scratchbox:



This chapter shows some basic possibilities to customize maemo themes and layouts using pre-existing helper tools and manual editing of layout definition files. Maemo uses Matchbox as the window manager on top of the X server and thus, Matchbox tutorials will be referred to here, as well.

As basic customization, it is possible to change, for example, just graphics and sounds by replacing the original file. This is not the best way to do it, but it will enable basic customization for end users. It is also possible to create the installation script on the package, installing the customization to back the old files up and, in case of uninstallation, copy the original files back. Creating a theme instead of just replacing files is a lot better way to do it. However, at the moment sound themes are not supported unfortunately, and changing sounds requires replacing the original files. The same thing applies with icons. Icon themes are not selectable on the device; there is always only one icon set selected and it can not be changed in any other way than by replacing its contents.

This chapter explains how to create a basic theme step by step by using theme creation template and tools, and how to test the new theme on the device. Also file reference information to the theme files is included for easy tweaking and also some brief instructions on how to customize sounds and icons are included.

10.2 Theme

Theme is a set of graphics and instructions for the system how to show them up. Theming tools are provided to make this task easier, so that the tool does the "programmer's task" for the graphics designer, and all you have to do is to draw the own images on top of the template.

A theme package consists of graphics and layout files. The graphics can be easily modified by simply replacing the bitmap images of the theme. Layout can also be tweaked if needed, or alternatively the default layout can be used.

Each maemo version comes with a specific default layout, where the number in the end of the package name represents the maemo-version. The graphics designer does not need to care of these, other than answering the question, which theme layout version to use in the `hildon-theme-bootstrap` command as described below.

Icons are not part of the theme, so they have to be edited separately.

Basic View of Hildon Desktop

The basic view consists Task Navigator bar, Status bar and Home, each having their own pixmap images or couple of images they are build of. The icon shown on the Task Navigator is one of the icons on the device, and it is not changed along with the theme, but has to be changed separately.

Simplest Customization

Users can do very simple customization by themselves by e.g. changing the background image. Any image (from the supported file types) can be applied as the background image with several different kind of scaling and stretching options. These include Centered, Scaled, Stretched, Tiled and Cropped. In devices bundled with the software on the level of maemo 4.x SDK, the themes can have translucent areas, and a change of simply a background image can have a dramatic effect of the look and feel of the desktop. The selection of background image hence is recommended to be done very carefully.

Downloading Themes from Internet and Installing Them on Device

To alter the default look of the maemo device, the simplest way to change it to suit the tastes is to check from Internet if there are ready-made themes available. One possible way is to go to the maemo.org downloads page and type keyword theme to the search box. Please make sure before installing a theme that was created by somebody else that it is compatible with the software release you are using.

Some themes also customize icons and sounds as explained in this chapter. Be prepared that after installing such a theme, there may not be going back to the original icons or sounds, if you did not back them up manually from command line, or the theme installer did not back them up. To return the original icons and sounds, once you have overwritten the original ones may require reflashing the device, or copying the icons and sounds manually from another maemo-device running exactly the same release.

Installing Theme Package on Maemo Device

Theme package that is made installable for the Application manager can be simply installed to the device by transferring the `.deb`-package into the device via memory card or by downloading it from the Internet. Alternatively, if you have set up the device in R&D-mode and have installed `ssh`, you can `scp` it to the device from Linux console. If you want to install it still with the application manager, it is advisable to put it into a directory which is visible to the application manager. A recommended place usually could be `/media/mmc1` or `/media/mmc2` (as the memory cards are kept intact even if you need to reflash the device).

Currently, theming is made of fixed size pixmap images. It is thus suitable only for exactly the same resolution as it was designed for. The form factor in the Internet tablet devices is 800x480, and the graphics drawn on top of the [template](#) will be good for exactly that resolution. However, maemo can run

also on other resolutions, e.g. it can be even compiled on a laptop. In such case, if you need some other size graphics, the template provided with the Test theme will no longer apply.

KNOWN LIMITATION: When you install & apply a new theme on the device, the background image is not automatically changed to a matching one. To change the background image, the user has to go to Home menu -> Set background image. There is a hackish workaround to this, which is explained later: you can put to your installation script a line that replaces the home-background.conf with your own custom one. However, in general, using that is not recommended.

Getting Started with Theming

First of all, a theme package is needed. If the theme is created by team of different persons (e.g. designers and programmers), the first thing to do is the programmer's task to create a theme package with theming tools (that is explained later in this chapter).

Second thing is the designer's task. Take the template.svg from the created theme package (the theme tools automatically attach this file on the theme). The task is to draw own theme on top of that.

When the theme is finished, the designer exports the drawn image as bitmap on top of the template.png on the subfolder template on the theme package (it has to have exactly the same dimensions than the original template.png).

When the template.png is saved, the theme can be built with dpkg-buildpackage -rfakeroot, as explained later in this chapter.

As mentioned already, the icons are not part of the theme package. That is partly because different icon themes are not supported on current maemo version. If the designer wants to alter the icons, they have to be changed manually, and an installation Debian package is needed to be created by the programmer. The installation script on that package is responsible of putting the pictures on correct folders on the device. This process completely overwrites the original icon files from the device.

10.2.1 Quick Start with Hildon Theme Tools

A quick way to create a new theme is to use hildon-theme-tools package, which is available for maemo SDK and Ubuntu desktop. First, you must apt-get install both hildon-theme-tools and hildon-theme-layout-4 packages in order to be able to create, edit and package your theme.

After installing the packages you have the command hildon-theme-bootstrap available. Run it in command line and answer some basic questions about your theme. It will create a new theme directory and fetch the current basic theme from subversion (**N.B.** If you run on a desktop machine instead of Scratchbox, you need to install also subversion in order to make things work. In Scratchbox this tool is already available). Here is an example of running hildon-theme-bootstrap:

```

$ hildon-theme-bootstrap // starting the program
Theme bootstrap tool by Michael Dominic K.
Copyright 2007 by Nokia Corporation.

This tool will bootstrap a new theme directory structure.

Which layout do you want to use?
1) hildon-theme-layout-3
2) hildon-theme-layout-4
#? // 2
What's the theme name? [ie. My Theme]
#? // Theme 1

What's the theme directory? [ie. mytheme]
#? // theme1

What's the author name? [ie. John Doe]
#? // Mr Maemo

What's the author's e-mail? [ie. john_doe@gmail.com]
#? // xxxx@maemo.org

Summary:

Theme name      : Theme1
Layout          : hildon-theme-layout-4
Package name    : hildon-theme-theme1
Theme directory : theme1 [/usr/share/themes/theme1]
Author          : Mr Maemo

Is this correct? [y/n]
#? // y
Fetching source from subversion repository...

```

After that you will have a folder called `hildon-theme-theme1`. It contains some files, out of which the most important is `template/template.png`, containing the theme graphics. You are free to edit it with your favorite graphics editor.

When done with editing, you must make a theme package. In order to make the package, change to the directory containing your theme and run the packaging command as follows:

```

$ cd hildon-theme-theme1
$ dpkg-buildpackage -rfakeroot -b

```

If everything went fine, you should have `hildon-theme-theme1_VERSION.deb` file in the parent directory. The file can be transferred to the device, and installed with Application Manager.

Now the customization of the theme is complete; if you wish, you can proceed with manual editing process to gain a full access on theme layouts.

For a more thorough tutorial on `hildon-theme-tools`, visit to the [Hildon Theming Overview](#) [48].

The theme package automatically installs its contents to `/usr/share/themes` on the device.

The images of the theme reside on the device in `/usr/share/themes/[Theme name]/images` folder, for example:

`/usr/share/themes/plankton/images`.

10.3 How to Create Debian Package for Custom Data Files

To install the customized files on the device, a programmer needs to create a Debian package. There is [13.1](#) section about how to do that in detail, but here is a [dummy example package](#) that you can copy-paste for your projects, if you like. Please look at the contents of the package to learn how it works, it is not explained in more detail here. You have to make changes to `configure.ac`, `debian/control`, `debian/rules`, `yourdatadir`, `yourdatadir/Makefile.am`, `yourdatadir/installable_file` to modify it to suit your purposes (you can modify and use it to install sound, graphics, icon, wallpaper etc. files on the device or Scratchbox).

This knowledge will be needed later in this chapter in order to be able to package the data files (such as icons, sounds or wallpapers) to the device.





10.4 Customizing Icons

In order to customize Task Navigator, Status Bar, Control panel and other icons, you must manually change the icons located in the device. The icons that come with the maemo SDK are not the same ones as can be seen on the device. If you want to change just a couple of icons and not all of them, please make sure that you install only those, and do not install the whole icon set from the maemo SDK, because it overwrites the product icons with a less polished set, which is not a recommended thing to do.

Task Navigator Icons

The main icons used in maemo, such as Internet browser, Application menu, Mail and the status bar icons can be modified just by changing the images in the following directory: `/usr/share/icons/hicolor/scalable/hildon/`.













The icons can be customized by changing the following images. Again, the original ones should be backed up first. Also, the same file format and image size should always be used. All icons are 8-bit, 64x54 pixels .png images.

Preview	Description	File name
	Contacts	qgn_grid_tasknavigator_contact.png
	Active Contact	qgn_grid_tasknavigator_contactactive.png
	Mail/Inbox	qgn_grid_tasknavigator_inbox.png
	New message	qgn_grid_tasknavigator_newmessage.png
	Sending / Receiving	qgn_grid_tasknavigator_sendrecieve.png
	Web Browser	qgn_grid_tasknavigator_web.png
	Others / Menu	qgn_grid_tasknavigator_others.png

Status Bar Icons

Also the status bar icon can be customized. Battery status, volume status, connection and phone are multiple state icons, so every single state image should be customized.

Files are located in the directory /usr/share/icons/hicolor/40x40/hildon/.
Battery, Brightness, USB and Alarm icons listing:

Icon	File name	Icon	File Name
	qgn_stat_battery_full100.png		qgn_stat_displaybright4.png
	qgn_stat_battery_full75.png		qgn_stat_displaybright3.png
	qgn_stat_battery_full50.png		qgn_stat_displaybright2.png
	qgn_stat_battery_full25.png		qgn_stat_displaybright1.png
	qgn_stat_battery_low.png		qgn_stat_usbact.png
	qgn_stat_battery_verylow.png		qgn_stat_alarmact.png

Connection and volume icon listing:












Icon	File name	Icon	File Name
	qgn_stat_internetconn_generic.png		qgn_stat_volumelevel4.png
	qgn_stat_internetconn_adhocwlan.png		qgn_stat_volumelevel3.png
	qgn_stat_internetconn_datacall.png		qgn_stat_volumelevel2.png
	qgn_stat_internetconn_packetdata.png		qgn_stat_volumelevel1.png
	qgn_stat_internetconnection_no.png		qgn_stat_volumelevel0.png
	qgn_stat_volumemute.png		

Image Reference with Pixel Sizes for Manual Customization of Icons

The images of the theme reside on the device under the following folder:
/usr/share/icons/hicolor/ under the following subfolders:

```
scalable/hildon
40x40/hildon
250x250/hildon
26x26/hildon
12x12/hildon
34x34/hildon
50x50/hildon
16x16/hildon
```

Designer's task: draw new icons with exactly same sizes as the original ones, and save them on the place of the original ones with exactly same file names.

Programmer's task: Please see chapter 13 to understand how to create the package that installs the files on correct locations and that is installable with the Application Manager. It is recommended that the theme package only replaces the customized icons, and not the whole set, unless all the icons are new ones and the customization is not limited to changing just couple of icons.

10.5 Editing Theme Layouts Manually

This section explains how to edit theme layouts manually. This allows broader customization of the device extending beyond mere modification of graphical elements.

Editing layout files enables the modification of the following:

- Font sizes and styles
- Widgets parameters (spacings, margins, distances)
- Hildon desktop parameters (Task Navigator size, place etc.)

N.B. The modification of colors and graphical elements is easier with hildon-theme-tools than by manipulating layouts. This should be sufficient for most customization needs.

Layouts can be edited either by modifying already existing layouts or creating custom layouts. For instance, in order to apply changes to the package

created by hildon-theme-tools you can edit files under /usr/share/hildon-theme-layout-4, if hildon-theme-layout-4 has been installed. The layout files for the selected layout are used for the theme created by hildon-theme-tools. If you want easily installable themes, it is the preferred way to edit these files.

Theme Files

After installation, all maemo themes are located in the following directory: /usr/share/themes/[Theme_name].

All themes share the same structure, having a pretty straightforward meaning:

```
gtk-2.0/  
  gtkrc  
  gtkrc.maemo_af_desktop  
  images/  
    *.png  
  matchbox/  
    theme.xml  
  index.theme
```

Theme.xml

Some simple window decoration customization can be achieved by modifying the theme.xml. In the maemo platform, the theme.xml defines the main title bar decoration and some dialog decoration as well. More information on theme.xml files can be found in the Matchbox themes tutorial.

Basic Resources

There are three basic resources in a theme.xml file: colors, fonts and pixmaps. All of them have to be declared before using. The syntax is pretty straightforward:

For colors:

```
<color id="lowlightcol" def="#d4e6fd80" />
```

basically it is an "id" followed by a definition, the element real value.

For fonts:

```
<font id="osso-TitleFont" def="Nokia Sans-16.75:30px" />
```

or

```
<font id="osso-TitleFont" def="Nokia Sans,fixed bold 16.75" />
```

again, an id and a definition. In font elements, alternative fonts can be defined by separating them with "|" (pipe). The first match will be used.

A valid font must be used. To avoid problems, the pipe character "|" should be used to give the default Nokia sans font as second alternative:

```
def="your Font,fixed bold 16.75 | Nokia Sans,fixed bold 16.75"
```

For Pixmaps (images):

```
<pixmap id="dialoguptile" filename="../images/qgn_plat_note_frame_tile_top.png" />
```

Id followed by a filename, pointing to a png image.

An example of a layout file can be seen in the hildon-theme-layout-4 package. You can apt-get source it from maemo SDK or Ubuntu repository.

Gtkrc Customization

After modifying theme.xml and the basic images and fonts, the next step on maemo customization and theme creation is to modify the GTK resource file and the resources used by it. This will enable changing the visuals for almost everything else in the system, including the Hildon widgets.

Even with the gtkrc file giving control over some important things, it is recommended to modify only the images in the folders that are used by the gtkrc file. The gtkrc file is too big to describe in full detail in this material.

N.B. It should be noted again, that the most important GUI elements can be modified using the hildon-theme-tools template.

Modifying System-Wide Colors

Some sections of the gtkrc file have to be changed to match the colors of the theme. Most color definitions are located in the beginning of the file. **N.B.** Be careful not to enter invalid values. If possible, always test on Scratchbox first.

Modifying System-Wide Fonts

In addition to the colors, the gtkrc file can be modified to make some custom fonts to work on UI. Again, it is very important to use the correct font name and size, or the rootfs may be damaged.

```
style style-Identifier{ font_name="Font Name size"}
```

Sample of font definition:

```
style "osso-SystemFont" { font_name = "Nokia Sans 16.75"}
```

and font usage:

```
class "*GtkComboBox" style "osso-SystemFont"
```

N.B. This is just a very simple introduction to gtk resource files. The GTK documentation should always be checked, if there are any doubts about the resource files.

Distributing Themes

Themes can be distributed as an Application Manager ready package. This will make it easier for the end users to install and uninstall themes. Themes created using the hildon-theme-tools skeleton are instantly ready for building the package using dpkg-buildpackage -rfakeroot as described above. The resulting .deb file can be then installed on the device using Application Manager by the user.

10.6 Adding New Fonts

The first step is to create a fonts folder for the user:

```
cd /home/user/  
mkdir .fonts
```

Adding new fonts is pretty simple: the font files (.ttf or type1) are to be placed in the fonts folder located on:

```
/home/user/.fonts/
```

To update the font cache, the command `fc-cache -fv` should be run in the *.fonts* directory.

To change the fonts used by the whole system, `theme.xml` file should be modified, as was explained in this chapter.

10.7 How to Customize Device Sounds

Maemo 4.1 does not support sound-theming, however, you can alter the sounds on the device by changing the original files located into `/usr/share/sounds` on the device. These files are not included on the SDK rootstrap, so you have to look at these files on the device directly at that location.

The sounds are in wav-format and each sound has its own logical id as its name (which helps you to determine which sound is in question). For example, you can `ls /usr/share/sounds` on the device with X-Terminal or via ssh.

How to Make Sound Package

A sound designer creates new sounds, and saves them to files with the same names as can be found from the `/usr/share/sounds`. E.g. if the sound designer wants to create their own window opening sound, they have to save the sound with the name `ui-window_open.wav` (please see above how to determine the file names). Each sound file is customized in the same way. Please be sure that the file name is exactly as found on the device, Linux system is case sensitive unlike Windows, and e.g. file named `ui-new_email.wav` has to be exactly `ui-new_email.wav`, and not, for example, `ui-new_email.Wav` or `ui-new_email.WAV`.

Programmer's task is now to create a Debian package that installs these sounds on top of the original ones to the directory `/usr/share/sounds` on the device. Please see chapter *Packaging, Deploying and Distributing* 13 of the Maemo Reference Manual to understand how to create the package that installs the files on correct locations and that is installable with the Application Manager.

10.8 Creating Custom Background Image (Wallpaper)

Example picture, sized 800x480 for using in the maemo devices:



The size of the background image is 800x480 pixels. It is a good idea to create the background image to be exactly that size and aspect ratio. However, the software on the device allows you to scale, stretch, crop and tile images that do not exactly fit nicely on the Home area. The best way, however, is to have the correct size and shape on the image.

In some themes, the Task Navigator and Status bar are translucent, and the background image shows up from below the panels. Therefore, it is recommended to keep this in mind, when designing the background image to ensure that it matches with the theme nicely.

The background images shall be installed into /usr/share/backgrounds. Each background image shall accompany a .desktop file in order to have it visible on the background image selection dialog. To make a background image as default, you have to replace the contents of file default.desktop in the given location to point out to the custom background image.

Format of .desktop file for a background image:

```
[Desktop Entry]
Type = Background Image
Name = titleoftheimage
File = /usr/share/backgrounds/imagename.png
X-Order=01
```

Example puro.desktop:

```
[Desktop Entry]
Type = Background Image
Name = Stream Scene from Lappland
File = /usr/share/backgrounds/puro.png
X-Order=01
```

To make that default, you have to also put the same information inside the default.desktop, default.desktop contents should then look like this:

```
[Desktop Entry]
Type = Background Image
Name = Stream Scene from Lappland
File = /usr/share/backgrounds/puro.png
X-Order=01
```

To make the new image to be default on next boot of the device, you have to configure also hildon desktop. The configuration entry is in yourhomedir/.osso/hildon-desktop/home-background.conf. On the device, the location is /home/user/.osso/hildon-desktop/home-background.conf and in Scratchbox /home/yourusername/.osso/hildon-desktop/home-background.conf.

The contents of the home-background.conf:

```
[Hildon Home]
BackgroundImage=/usr/share/backgrounds/puro.png
CachedAs=
Red=0
Green=0
Blue=0
Mode=Centered
```

N.B. This is not recommended for packages you create, as users may not like if their favorite background image gets replaced without questions. However, this is the way to implement it.

To create a simple package for your own custom background image, you can export this [puro-background](#) package from svn repository, edit the configuration to use your custom image instead. Build it, and if you did everything correctly and there were no errors, you are done and your background image package is ready for distribution.

Try building and installing the wallpaper package in scratchbox:

```
> svn export https://garage.maemo.org/svn/maemoexamples/tags/maemo_4.1/puro-background/
...
> cd puro-background/
> dpkg-buildpackage -rfakeroot
...
> fakeroot dpkg -i ../puro-background_0.0.1_all.deb
```

Start desktop environment:

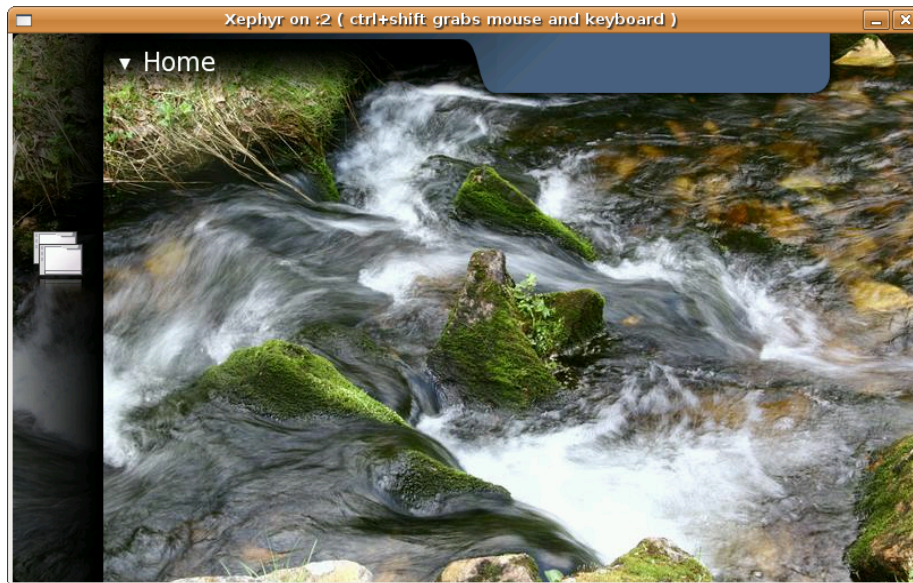
```
> export DISPLAY=:2
```

Outside scratchbox start Xephyr.

Inside scratchbox:

```
> af-sb-init.sh start
```

Observe the Xephyr window:



10.9 Third Party Theme Packages and Theme Tools

Some developers have made their own themes and tools. For example Urho Konttori has made a Java-based theme maker, which runs on any OS and makes the installable Debian package automatically. However, please contact the maker of the given tool to inquire about the updateness of the tool etc. - theme packages created for maemo 3.x and theme tools created for maemo 3.x do not work on maemo 4.x, because there have been a number of changes in theming in maemo 4.x.

10.10 References

- [Hildon Theme Howto](#) [46]
- [Hildon Theming Overview](#) [48]
- [Hildon Theme Tools Overview](#) [47]
- [How to Become Root](#) [49]
- [Matchbox project](#) [76]
- [Open SSH](#) [58]
- [Designing Matchbox Themes HOWTO](#) [77]

More useful information:

- [Icon Theme Specification](#) [51]

Chapter 11

Kernel Guide

This chapter describes how to configure, compile and flash the Linux kernel for the Internet Tablet device. The chapter is targeted at developers wishing to compile their own custom kernels for the device.

11.1 Prerequisites

Before starting, the maemo environment should be set up. The required GCC toolchain that is used to compile the kernel is included in the Scratchbox by default.

It is not mandatory to set up a separate target for kernel compilation, but this example does it in case the default armel target has been modified in some special way.

- Start Scratchbox

```
$ scratchbox
```

- Create a new target called MaemoKernel with qemu-arm CPU transparency. The second command installs the armel rootstraps to the target. The last command installs the C-library, /etc, devkits and fakeroot.

```
[sbox-armel: ~] > sb-conf setup MaemoKernel -c cs2005q3.2-glibc2.5-arm -t  
/scratchbox/devkits/cputransp/bin/qemu-arm-0.8.2-sb2  
[sbox-armel: ~] > sb-conf select MaemoKernel  
[sbox-armel: ~] > sb-conf rs MaemoKernel  
/home/username/maemo-sdk-rootstrap_4.1_armel.tgz  
[sbox-armel: ~] > sb-conf in MaemoKernel -e -d -F
```



N.B.

The username above refers to your login name in the environment. If you have used the maemo installer, the rootstraps are under your home directory. If you have performed a manual installation, the rootstrap is under /scratchbox/packages directory. This should be paid attention to, when running the sb-conf command above.

- Verify that the sources.list file inside the scratchbox environment is correct. If the below lines are not in the /etc/apt/sources.list file, add them there.

```
deb http://repository.maemo.org/ diablo/sdk free non-free
deb-src http://repository.maemo.org/ diablo/sdk free

deb http://repository.maemo.org/ diablo/tools free non-free
deb-src http://repository.maemo.org/ diablo/tools free
```

- Create a working directory (inside Scratchbox) for the kernel sources.

```
[sbox-MaemoKernel: ~] > mkdir ~/maemo_kernel
```

The Scratchbox environment is now ready for compiling the kernel

11.2 Getting Kernel Sources

Kernel sources are not included in the rootstrap, and therefore need to be downloaded from the repository.

- Select the kernel compilation target, if not selected already.

```
[sbox-armel: ~] > sb-conf select MaemoKernel
```

- Update the package database. This requires the earlier modifications in the sources.list file.

```
[sbox-MaemoKernel: ~] > apt-get update
```

- Go to the working directory and fetch the sources.

```
[sbox-MaemoKernel: ~] > cd ~/maemo_kernel
[sbox-MaemoKernel: ~/maemo_kernel ] > apt-get source kernel-source-diablo
```

Kernel sources should now be fetched and ready to be compiled.

11.3 Configuring Source Tree and Compiling Kernel

- There is a source subdirectory. Enter the directory and create the default configuration.

```
[sbox-MaemoKernel: ~/maemo_kernel ] > cd kernel-source-diablo-2.6.21/kernel-source/
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > make \
nokia_2420_defconfig
# lots of output from make program...
```

- Compile the kernel image, and check the image file timestamp to ensure that it is properly created.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > make \
bzImage
# compilation output...
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > ls -l \
arch/arm/boot/zImage
-rwxrwxr-x 1 maemo maemo 1530180 Nov  2 13:40 arch/arm/boot/zImage
```

Now the device can be flashed with the new kernel image using the Flasher tool. This should be performed outside Scratchbox. See section 11.6 for short flashing instructions.

11.4 Changing Default Kernel Configuration

The following steps describe how to change the default kernel configuration.

- Restore the original default configuration, just in case it has been changed.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > make \
nokia_2420_defconfig
```

- Edit the configuration file with your editor of choice. It is all right to edit the file, even though there is a warning against changing it included.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > vi \
.config
```

- Include your changes for compilation.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > make \
oldconfig
```

Now a kernel image with modified configuration can be recompiled, as described in the previous section.

11.5 Configuring and Compiling Kernel Modules

This section explains how to configure and compile additional kernel modules for the Internet Tablet in the maemo environment. NFS is used as an example.

- Go to your working directory with kernel sources.

```
[sbox-MaemoKernel:]> cd ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source
```

- Edit the configuration file and define that the NFS will be a kernel module.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > vi \
.config
```

Find the location of the NFS configuration.

```
#
# Network File Systems
#
# CONFIG_NFS_FS is not set
# CONFIG_NFSD is not set
```

Enable NFS support as modules:

```
#
# Network File Systems
#
CONFIG_NFS_FS=m
CONFIG_NFSD=m
```

- As described in previous sections, refresh the configuration and build new modules.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > make \
oldconfig
# lots of output here. Answer Y to NFSv3, default to others.
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > make \
modules
```

- Check with the find command to see what kernel object files (extension .ko) there now are.

```
[sbox-MaemoKernel: ~/maemo_kernel/kernel-source-diablo-2.6.21/kernel-source] > find \
. -name "*.ko" -ls
1219535 20 -rw-rw-r-- 1 maemo maemo 19452 Nov 2 13:48
./arch/arm/mach-omap2/observability.ko
1219539 12 -rw-rw-r-- 1 maemo maemo 8304 Nov 2 13:48
./arch/arm/mach-omap2/peekpoke.ko
1219543 8 -rw-rw-r-- 1 maemo maemo 5928 Nov 2 13:48
./arch/arm/mach-omap2/snapshot.ko
2967668 168 -rw-rw-r-- 1 maemo maemo 167603 Nov 2 13:48
./net/sunrpc/sunrpc.ko
2710063 32 -rw-rw-r-- 1 maemo maemo 31503 Nov 2 13:48
./drivers/usb/gadget/g_ether.ko
2710071 24 -rw-rw-r-- 1 maemo maemo 22481 Nov 2 13:48
./drivers/usb/gadget/gadgetfs.ko
2710067 36 -rw-rw-r-- 1 maemo maemo 36142 Nov 2 13:48
./drivers/usb/gadget/g_file_storage.ko
2890706 128 -rw-rw-r-- 1 maemo maemo 124939 Nov 2 13:48 ./fs/ext3/ext3.ko
1219547 12 -rw-rw-r-- 1 maemo maemo 9159 Nov 2 13:48 ./fs/mbcache.ko
2890698 8 -rw-rw-r-- 1 maemo maemo 5922 Nov 2 13:48
./fs/exportfs/exportfs.ko
2890702 64 -rw-rw-r-- 1 maemo maemo 60734 Nov 2 13:48 ./fs/ext2/ext2.ko
2890710 68 -rw-rw-r-- 1 maemo maemo 61687 Nov 2 13:48 ./fs/jbd/jbd.ko
2890718 144 -rw-rw-r-- 1 maemo maemo 142363 Nov 2 13:48 ./fs/nfs/nfs.ko
2890722 112 -rw-rw-r-- 1 maemo maemo 109711 Nov 2 13:48 ./fs/nfsd/nfsd.ko
2890714 80 -rw-rw-r-- 1 maemo maemo 75526 Nov 2 13:48
./fs/lockd/lockd.ko
```

The list of .ko files might differ from the list above, depending on whether some additional changes were made to the .config file. The important files are sunrpc.ko, nfs.ko and lockd.ko. Check that you have those.

The kernel modules are kept in /mnt/initfs/lib/modules/2.6.21-omap1/. However, it is write protected. This is not a problem, because the insmod command can be used to load the modules into the running kernel from any directory where they have been saved in.

11.6 Flashing Kernel

The custom kernel can be flashed from outside the Scratchbox with the flasher tool.

```
# flasher -f -k /scratchbox/users/<yourname>/home/<yourname>/maemo_kernel/\
kernel-source-diablo-2.6.21/kernel-source/arch/arm/boot/zImage
```

Chapter 12

Porting Software

12.1 Introduction

The following code examples are used in this chapter:

- [maemo-monkey-bubble](#)
- [MaemoPad](#)

Much effort has been made in the design of maemo platform to allow easy porting of regular GNU/Linux desktop software to the mobile maemo environment. An earlier chapter in this guide explained the basic tools that ease cross-compilation and help coping with the GNU autotools. This chapter gives pointers to the relevant guides focusing on the differences in the application programming and user interfaces.

Command Line Programs

In most cases, porting software with no user interface is trivial and straightforward. First, the source code of a program is unpacked to the home directory of a Scratchbox user. Second, inside ARMEL target *configure* and *make* are run. Then the compiled program can be tested on the device. Finally the software needs to be packaged.

Programs with Graphical User Interface

The porting of an application that uses GTK+ for its graphical user interface begins the same way as above. In addition to this, the user interface part needs to be refactored to use Hildon instead of directly using GTK+. Dependencies to GNOME components, if any, need to be removed or replaced with corresponding maemo SDK components. If the application uses any components not available in the maemo SDK, these must be also ported by the developer.

Applications that do not use GTK+ and instead use e.g. SDL need more work. The first thing that can be noticed is that the virtual keyboard is missing, because the Hildon Input Method is not available. Thus, application user cannot interact with the application.

Section [12.2](#) goes into the necessary details of porting an existing GTK+ application to maemo environment.

Localization

The localization of maemo applications is performed using the common *gettext* package. Translating applications to different languages is described in detail in section [12.3](#).

12.2 Porting Existing Applications to Maemo 4.x

This section describes the process of porting an application to the maemo platform. When starting to port an application to maemo platform, the first step is to set up the development environment. The actual porting after that is described in this section.

The ported game will work on the Scratchbox environment but the hardware key bindings are still missing and therefore it isn't playable on the device yet. For more information about hardware keys can be found in section *Hardware Keys* of chapter *Application Development* in Maemo Diablo Reference Manual.

12.2.1 Introduction

Application that is used as an example for porting is [Monkey Bubble](#), a GNOME/GTK+ based game. It has simple controls, and supports network play. Monkey Bubble version 0.4.0 is used in this example.

The Monkey Bubble interface consists of the main window, menus and a couple of dialogs.

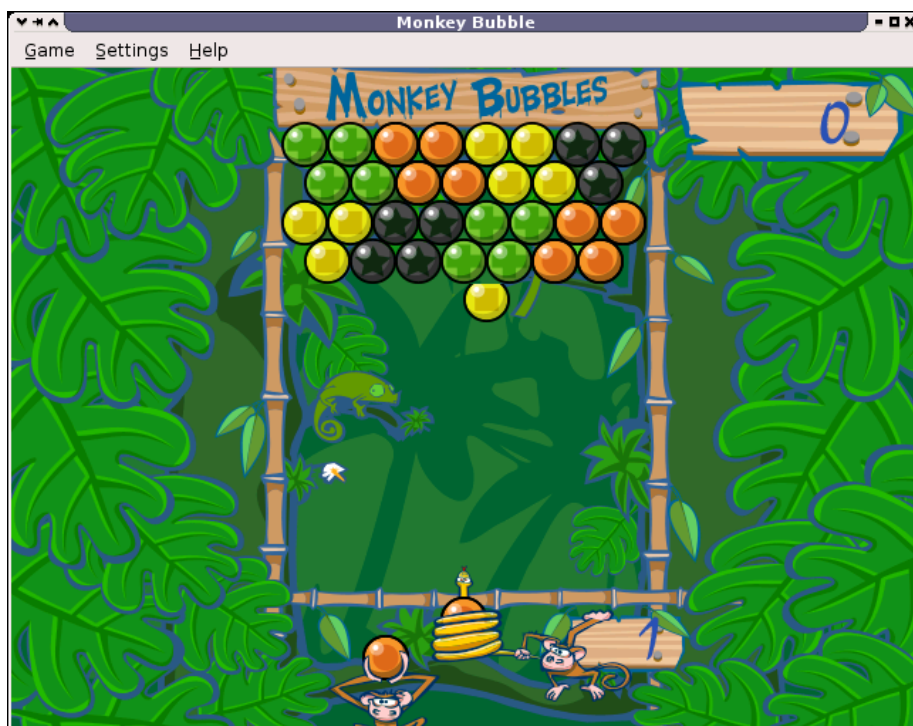


Figure 12.1: Monkey Bubble main window

The figure 12.1 is a screenshot of the main window with an ongoing game. The game is played with configurable keys; the defaults are the arrow keys: left, right and up. The aim is to make all the bubbles disappear. When three or more bubbles with the same color are grouped, they will disappear.

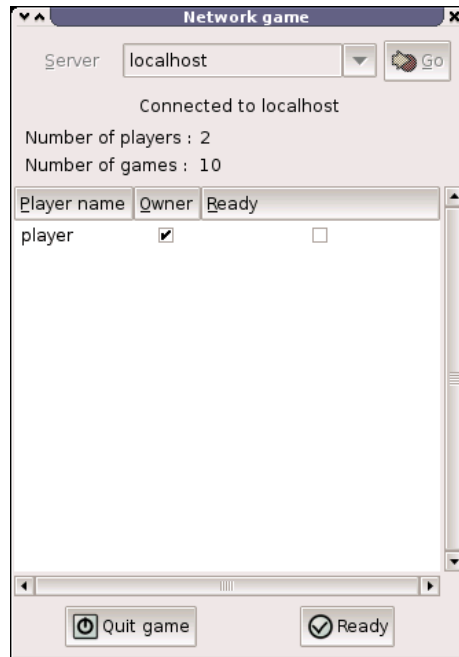


Figure 12.2: Network game window

The figure 12.2 is a screenshot of the network game window, where user can join a network game. It includes a field for selecting server, status about players and game and control buttons.

12.2.2 Application File Structure

Monkey Bubble already has a nicely structured source tree, so there was no need to modify it. XML-formatted [Glade](#) [27] files can be found in data directory. Help files can be found in help directory. Graphics can be found in pixmaps directory. Localization files are located in po directory. Audio files can be found in sounds directory. The source code itself is located under src directory in subdirectories, such as audio, input, monkey, net, ui, util and view.

```

./po
./data
./pixmaps
./pixmaps/bubbles
./pixmaps/frozen-bubble
./pixmaps/snake
./pixmaps/number
./sounds
./src
./src/util
./src/input
./src/monkey
./src/view
./src/audio
./src/net
./src/ui
./help
./help/C
./help/fr

```

See also section *Creating Application File Structure* in chapter *Application Development* of the Maemo Diablo Reference Manual.

12.2.3 Requirements and Configure Changes

Monkey Bubble already uses GNU autotools, so the needed Makefile.am and configure.in files are there. Only autogen.sh was missing, so it was created. Sections below discuss the changes needed in Makefile.am and configure.ac files.

```

#!/bin/sh
set -x
glib-gettextize --copy --force
libtoolize --automake
intltoolize --copy --force --automake
aclocal-1.7
autoconf
autoheader
automake-1.7 --add-missing --foreign

```

Listing 12.1: Created maemo-monkey-bubble/autogen.sh

Remember to set permissions to this executable with "chmod a+x autogen.sh".

Maemo SDK meets most of the Monkey Bubble requirements. Biggest exceptions are libgnomeui, bonobo and librsvg. After checking the sources, the libgnomeui and bonobo dependencies can be removed. So, librsvg and its dependencies are additionally needed to be installed.

Start from librsvg installation. One version is available in maemo.org extras repository. Install from there the following packages:

```

librsvg2-2
librsvg2-common
librsvg2-dev

```

Then the configure.in file needs to be modified. The dependency to libgnomeui-2.0 must be removed. And because the UI is going to be hildonized, hildon-1 and libosso need to be added to the list between PKG_CHECK_MODULES(UI,[and following]).

From Makefile.am, remove the help from SUBDIRS list, because the provided help is incompatible with the Hildon help framework. Remove line

help/Makefile from configure.in AC_OUTPUT section as well. Additionally, help can be implemented but it is out of the scope of this section.

After the changes, autogen.sh must be executed to apply the changes.

12.2.4 Basic Porting

Localization and src/ui/main.c

The very first thing is to get Monkey Bubble to work in the maemo environment. For localization, Monkey Bubble uses bonobo. It must be replaced with a compatible localization, described in more detail in section 12.3.

First, localization must be initialized in src/ui/main.c. The following lines must be removed from it:

```
#include <bonobo/bonobo-i18n.h>
#include <libgnomeui/gnome-ui-init.h>
```

Listing 12.2: maemo-monkey-bubble/src/ui/main.c

After that, these lines must be added:

```
#include <locale.h>
#include <libintl.h>
```

Listing 12.3: maemo-monkey-bubble/src/ui/main.c

And new lines in the main() function before bindtextdomain function call:

```
setlocale(LC_ALL, "");
bind_textdomain_codeset(PACKAGE, "UTF-8");
```

Listing 12.4: maemo-monkey-bubble/src/ui/main.c

Remove also the gnome_program_init function call.

For the localization to work, the line #include <bonobo/bonobo-i18n.h> needs to be removed from every file using localization. In order to make the strings localizable, they need to be wrapped in gettext("String") calls. In practice, writing gettext() for every string is tedious. The common practice is to set the following #defines. The N_ is used for gettext_noop(), but it is not available, so the macro does nothing. So the following lines must be added to the beginning of every file using localization:

```
#include <libintl.h>
#define _(String) gettext(String)
#define N_(String) (String)
```

Listing 12.5: maemo-monkey-bubble/src/ui/ui-main.c

Files using localization include the following:

```
src/ui/keyboard-properties.c
src/ui/ui-network-client.c
src/ui/ui-network-server.c
src/ui/ui-main.c
```

Also po/fr.po file needs couple of small fixes. You need to remove "#, fuzzy" line from the header and set Language-Team to be something else than the default.

Removing GNOME Features

The next step is to remove other GNOME functionality. File `src/ui/ui-main.c` is next to edit. Remove the following include lines:

```
#include <libgnomeui/gnome-about.h>
#include <libgnome/gnome-sound.h>
#include <libgnome/gnome-help.h>
```

Listing 12.6: `maemo-monkey-bubble/src/ui/ui-main.c`

Then comment out contents of `show_help_content` and `about` functions, and remove completely function `show_error_dialog` and its prototype from the beginning.

Now the game can be configured and compiled with the following commands:

```
./autogen.sh
./configure --prefix=/usr
make
make install
monkey-bubble
```

Everything should go fine, and the game should be compiled properly. When trying to install and run the game, the following should be seen:



This is fine, but obviously the borders and menus are not hildonized. Also, when trying menu functionality, it is easy to see that the menus will not fit nicely there. So the next step is customization.

12.2.5 User Interface Changes

Hildonizing Main View

Monkey bubble uses Glade [27] for its UI creation. Unfortunately, Glade does not support Hildon, so some changes have to be made.

First of all, in data/monkey-bubble.glade, the main_window type GtkWidget must be changed to GtkVBox. Then all window properties, except "visible" must be removed. After that, the menu must be removed. That is done by removing completely the child containing GtkMenuBar and its subchildren.

Next the src/ui/ui-main.c function ui_main_new should be changed. Add the following lines to the beginning of the function, and remove the Keyboard-Properties * kp; line:

```
HildonProgram * program;
GtkWidget * container;
GtkWidget * main_menu;
```

Listing 12.7: maemo-monkey-bubble/src/ui/ui-main.c

Then make the following modifications; the old code is commented out and new lines added after it:

```
#ifdef GNOME
    PRIVATE(ui_main)->window = glade_xml_get_widget( PRIVATE(
        ui_main)->glade_xml, "main_window");
#endif
#ifdef MAEMO
    container = glade_xml_get_widget( PRIVATE(ui_main)->glade_xml, "
        main_window");
    program = HILDON_PROGRAM(hildon_program_get_instance());
    PRIVATE(ui_main)->window = hildon_window_new();
    g_signal_connect_swapped(PRIVATE(ui_main)->window, "destroy",
        GTK_SIGNAL_FUNC(quit_program), ui_main);
    hildon_program_add_window(program, HILDON_WINDOW(PRIVATE(ui_main)->
        window));
    gtk_container_add(GTK_CONTAINER(PRIVATE(ui_main)->window),
        GTK_WIDGET(container));
    g_set_application_name(_("Monkey Bubble"));
```

Listing 12.8: maemo-monkey-bubble/src/ui/ui-main.c

Because the menu was removed from the Glade file, it must be constructed manually in a Hildon-compatible way. Also, not all the functionality provided by the old menu is needed, so things can be left out. The necessary parts are new game, join network game, pause and quit. Such a tiny menu can be constructed after gtk_box_pack_end function call:

```
main_menu = gtk_menu_new();

item = gtk_menu_item_new_with_label(_("New game"));
g_signal_connect_swapped( item, "activate", GTK_SIGNAL_FUNC(
    new_1_player_game), ui_main);
gtk_menu_append(main_menu, item);

item = gtk_menu_item_new_with_label(_("Join network game"));
g_signal_connect_swapped( item, "activate", GTK_SIGNAL_FUNC(
    new_network_game), ui_main);
gtk_menu_append(main_menu, item);

item = gtk_menu_item_new_with_label(_("Pause"));
g_signal_connect_swapped( item, "activate", GTK_SIGNAL_FUNC(pause_game),
    ui_main);
gtk_menu_append(main_menu, item);

item = gtk_menu_item_new_with_label(_("Quit"));
```

```
g_signal_connect_swapped( item, "activate", GTK_SIGNAL_FUNC(quit_program)
, ui_main);
gtk_menu_append(main_menu, item);

hildon_window_set_menu(HILDON_WINDOW(PRIVATE(ui_main)->window),
GTK_MENU(main_menu));

gtk_widget_show_all(GTK_WIDGET(main_menu));
```

Listing 12.9: maemo-monkey-bubble/src/ui/ui-main.c

After that the code related to the old menu can be removed, starting from the next line and ending to `g_signal_connect_swapped` function call, the latter being the last removed line. Also the unneeded functions and their prototypes need to be removed:

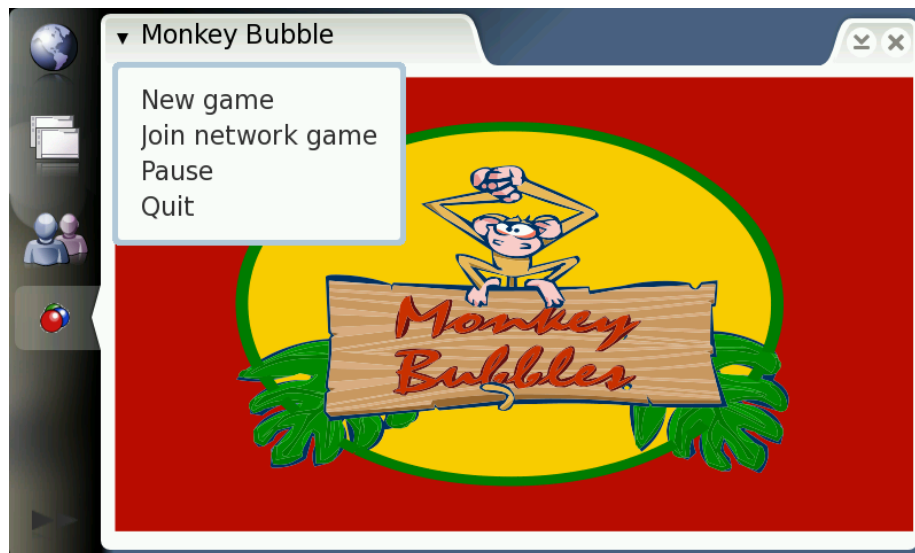
```
ui_main_new_2_player_game
new_2_player_game
new_network_server
stop_game
about
show_help_content
show_preferences_dialog
```

Remember to add the proper include to the beginning of the file:

```
#include <hildon/hildon-program.h>
```

Listing 12.10: maemo-monkey-bubble/src/ui/ui-main.c

Now the main view and menu should be hildonized properly to look like:



Hildonizing Network Window

The Network game window shown in the figure 12.3 is still `GtkWindow`, and is not hildonized properly. The solution is to make it a `GtkVBox` and place it under a new `GtkDialog`.

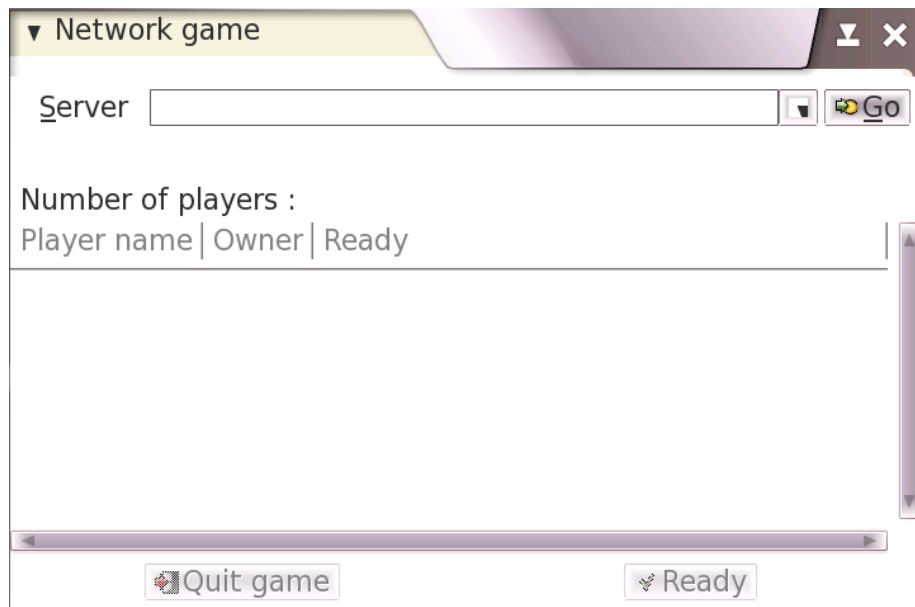


Figure 12.3: Network game window

The starting point here is the glade file located at `data/netgame.glade`. Change the `GtkWindow` to `GtkVBox` and remove all properties except "visible". Change `GtkScrolledWindow` widget property "height_request" to the value 200 to make everything fit on the screen. A close button is also missing now, because Monkey Bubble relies on the close button of `GtkWindow`. There is a suitable place between "quit game" and "ready" buttons. You can just copy-paste the whole child containing "quit_button" widget, rename it as "close_button", change the number values of the other widgets name under it for example to 9 to prevent collisions, and change the `GtkLabel` "Quit game" to "Close". That is enough for the glade file.

Next, the `src/ui/ui-network-client.c` and `ui_network_client_new` functions. Add this line to the beginning of the function:

```
GtkWidget * container;
```

Listing 12.11: `maemo-monkey-bubble/src/ui/ui-network-client.c`

Make the following changes to create a new `GtkDialog` and show `network_window` contents under it. The old code is commented out, and the new lines added after it:

```
#ifdef GNOME
    PRIVATE(ngl)->window = glade_xml_get_widget( PRIVATE(ngl)->
        glade_xml, "network_window");
#endif
#ifdef MAEMO
    PRIVATE(ngl)->window = gtk_dialog_new();
    gtk_window_set_title(GTK_WINDOW(PRIVATE(ngl)->window), _("Network
        game"));
    container = glade_xml_get_widget( PRIVATE(ngl)->glade_xml, "
        network_window");
```

```

gtk_container_add(GTK_CONTAINER(GTK_DIALOG(PRIVATE(ngl)->window)->
    vbox), container);
item = glade_xml_get_widget( PRIVATE(ngl)->glade_xml, "close_button")
;
g_signal_connect_swapped( item, "clicked", GTK_SIGNAL_FUNC(close_signal
    ), ngl);
#endif

```

Listing 12.12: maemo-monkey-bubble/src/ui/ui-network-client.c

And to the end of the `ui_network_client_new` function, before return, add the following line:

```

gtk_widget_show_all(GTK_WIDGET(PRIVATE(ngl)->window));

```

Listing 12.13: maemo-monkey-bubble/src/ui/ui-network-client.c

A new function is needed, called `close_signal`, which is called when the close button is pressed. Add this function before the `ui_network_client_new` function:

```

static gboolean close_signal(gpointer    callback_data,
                             guint       callback_action,
                             GtkWidget   *widget) {

    UiNetworkClient * self;
    self = UI_NETWORK_CLIENT(callback_data);

    quit_signal(callback_data, callback_action, widget);
    gtk_widget_hide_all(PRIVATE(self)->window);
    return FALSE;
}

```

Listing 12.14: maemo-monkey-bubble/src/ui/ui-network-client.c

There is still one problem: The new close button is located in "connected_game_hbox" widget, which is set insensitive when not connected to the server. The ability to close the window is always needed. The sensitivity of the other widgets has to be changed properly. Add this new function after the previously added `close_signal` function:

```

void connected_set_sensitive(UiNetworkClient * ngl, gboolean sensitive)
{
    set_sensitive( glade_xml_get_widget( PRIVATE(ngl)->glade_xml
        , "scrolledwindow2"), sensitive);
    set_sensitive( glade_xml_get_widget( PRIVATE(ngl)->glade_xml
        , "quit_button"), sensitive);
    set_sensitive( glade_xml_get_widget( PRIVATE(ngl)->glade_xml
        , "ready_button"), sensitive);
}

```

Listing 12.15: maemo-monkey-bubble/src/ui/ui-network-client.c

There is one `gtk_widget_set_sensitive` function call in `ui_network_client_new` and multiple `set_sensitive` calls for `connected_game_hbox`; replace these with:

```

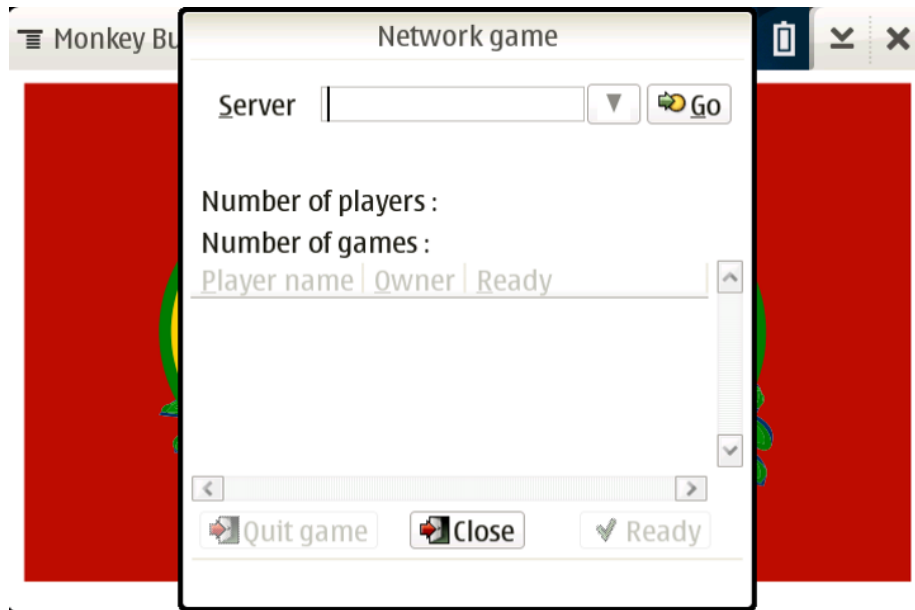
connected_set_sensitive(ngl, FALSE);

```

Listing 12.16: maemo-monkey-bubble/src/ui/ui-network-client.c

There is also one call in `recv_network_xml_message` function, where the boolean value must be TRUE.

That is all; the network game dialog should now be hildonized and contain a new close button. The "quit game", "ready" and player list should change their sensitivity properly upon connected and disconnected states.



Hildonized network game dialog

12.2.6 State Saving

Maemo supports state saving and background killing application. The application can later be loaded up again with the same state as before. This section describes the steps needed to make Monkey Bubble support state savings.

State Saving Changes in `src/ui`

Some new files need to be created, first file for global data `src/ui/global.h`:

```
#ifndef GLOBAL_H
#define GLOBAL_H

#include <libosso.h>

#include "game-1-player.h"

#define MONKEY_TEMP "/tmp/monkey_level_state"

struct GlobalData {
    osso_context_t *osso;
    Game1Player *game;
};

struct StateData {
    int game;
```

```

    int level;
    int score;
    int loadmap;
};

extern struct StateData state;
extern struct GlobalData global;

#endif

```

Listing 12.17: maemo-monkey-bubble/src/ui/global.h

Second one is src/ui/state.h:

```

#ifndef STATE_H
#define STATE_H

#include <glib.h>

gboolean state_load(void);
gboolean state_save(void);
void state_clear(void);

#endif

```

Listing 12.18: maemo-monkey-bubble/src/ui/state.h

Then src/ui/state.c, implementing loading, saving and cleaning the state:

```

#include <libosso.h>

#include "state.h"
#include "global.h"

struct StateData state;

gboolean state_load(void)
{
    osso_state_t osso_state;
    osso_return_t ret;

    osso_state.state_size = sizeof(struct StateData);
    osso_state.state_data = &state;

    ret = osso_state_read(global.osso, &osso_state);
    if (ret != OSSO_OK)
        return FALSE;
    return TRUE;
}

gboolean state_save(void)
{
    osso_state_t osso_state;
    osso_return_t ret;

    osso_state.state_size = sizeof(struct StateData);
    osso_state.state_data = &state;

    ret = osso_state_write(global.osso, &osso_state);

    if (ret != OSSO_OK)
        return FALSE;
    return TRUE;
}

```



```

}

void state_clear(void)
{
    state.game = 0;
    state.level = 0;
    state.score = 0;
    state.loadmap = 0;

    state_save();
}

```

Listing 12.19: maemo-monkey-bubble/src/ui/state.c

These must be added to src/ui/Makefile.am to monkey_bubble_SOURCES list before \$(NULL):

```
state.c state.h global.h \
```

Listing 12.20: maemo-monkey-bubble/src/ui/Makefile.am

Some changes to src/ui/main.c, new includes, definitions and functions:

```

#include <libosso.h>

#include "state.h"
#include "global.h"

#include "game-1-player.h"

#define APPNAME "com.nokia.monkey_bubble"
#define APPVERSION "0.0.1"

struct GlobalData global;

static void _top_cb(const char *args, gpointer data)
{
}

static void _hw_cb(osso_hw_state_t * state, gpointer data)
{
    if(state->shutdown_ind)
    {
        state_save();
        gtk_main_quit();
    }
}

osso_context_t *osso_init(void)
{
    osso_context_t *osso =
        osso_initialize(APPNAME, APPVERSION, TRUE, NULL);

    if (OSSO_OK != osso_application_set_top_cb(osso, _top_cb, NULL))
        return NULL;

    if (OSSO_OK != osso_hw_set_event_cb(osso, NULL, _hw_cb, NULL))
        return NULL;

    return osso;
}

```

Listing 12.21: maemo-monkey-bubble/src/ui/main.c

And change the main() function as follows:

```
global.osso = osso_init();
if (!global.osso) {
    perror("osso_init");
    exit(1);
}
global.game = NULL;

/* ... */

if (!state_load()) state_clear();

if (state.game == 1) {
    continue_game();
}

/* ... */
```

Listing 12.22: maemo-monkey-bubble/src/ui/main.c

Set the topmost callback in src/ui/ui-main.c. It saves the current level and state when Monkey Bubble loses its topmost status, and sets the hibernate flag properly. New continue_game function to continue after loading state:

```
#include "global.h"
#include "state.h"

static void ui_main_new_1_player_game(UiMain * ui_main);

static void ui_main_topmost_cb(GObject *self, GParamSpec *
    property_param, gpointer null)
{
    HildonProgram *program = HILDON_PROGRAM(self);

    if (program == NULL) return;

    if (hildon_program_get_is_topmost(program)) {
        hildon_program_set_can_hibernate(program, FALSE);
    } else {
        if (state.game == 1 && global.game!=NULL) {
            game_1_player_save(global.game);
            state.loadmap=1;
        }
        state_save();
        hildon_program_set_can_hibernate(program, TRUE);
    }
}

void continue_game(void) {
    UiMain * ui_main;
    ui_main = ui_main_get_instance();

    ui_main_new_1_player_game(ui_main);
}
```

Listing 12.23: maemo-monkey-bubble/src/ui/ui-main.c

Add state_clear() call to program quitting callback:

```
static void quit_program(gpointer    callback_data,
    guint        callback_action,
    GtkWidget    *widget) {
```

```

    /* ... */

    state_clear();

    /* ... */
}

```

Listing 12.24: maemo-monkey-bubble/src/ui/ui-main.c

In addition to the `ui_main_new()` function, after `g_set_application_name` function call:

```

g_signal_connect(G_OBJECT(program), "notify::is-topmost",
                 G_CALLBACK(ui_main_topmost_cb), NULL);

```

Listing 12.25: maemo-monkey-bubble/src/ui/ui-main.c

To the `new_1_player_game` function, add before `ui_main_new_1_player_game` function call:

```

state_clear();

```

Listing 12.26: maemo-monkey-bubble/src/ui/ui-main.c

Function prototype must be added to `src/ui/ui-main.h`:

```

void continue_game(void);

```

Listing 12.27: maemo-monkey-bubble/src/ui/ui-main.h

Make the following changes in `src/ui/game-1-player-manager.c` to keep track of level and scores and continue from last level and score:

```

#include "global.h"

/* ... */

static gboolean startnew_function(gpointer data) {

    /* ... */

    PRIVATE(manager)->current_level++;
#ifdef MAEMO
    state.level = PRIVATE(manager)->current_level;
#endif

    /* ... */

}

static void game_1_player_manager_state_changed(Game * game,
                                                Game1PlayerManager *
                                                manager) {

    /* ... */

    PRIVATE(manager)->current_score = game_1_player_get_score(
        GAME_1_PLAYER(game));
#ifdef MAEMO
    state.score = PRIVATE(manager)->current_score;
#endif
}

```

```

    /* ... */
}

static void game_1_player_manager_start_level(Game1PlayerManager * g) {
    /* ... */

#ifdef MAEMO
    global.game = game;
#endif

    /* ... */
}

void game_1_player_manager_start(GameManager * g) {
    /* ... */

#ifdef GNOME
    PRIVATE(manager)->current_level = 0;
    PRIVATE(manager)->current_score = 0;
#endif
#ifdef MAEMO
    if (state.game == 1 && state.level > 0 ) {
        PRIVATE(manager)->current_level = state.level;
        PRIVATE(manager)->current_score = state.score;
    } else {
        state.level = 0;
        PRIVATE(manager)->current_level = 0;
        PRIVATE(manager)->current_score = 0;
    }
    state.game = 1;
    /* ... */
}

```

Listing 12.28: maemo-monkey-bubble/src/ui/game-1-player-manager.c

The changes to src/ui/game-1-player.c to save and load the level state:

```

#include "global.h"

/* ... */

void game_1_player_save(Game1Player *game)
{
    monkey_save(PRIVATE(game)->monkey, MONKEY_TEMP);
}

/* ... */

Game1Player * game_1_player_new(GtkWidget * window, MonkeyCanvas *
    canvas, int level, gint score) {
    /* ... */

#ifdef GNOME
    PRIVATE(game)->monkey =
        monkey_new_level_from_file(DATADIR"/monkey-bubble/
            levels",

```

```

level);
#endif

#ifdef MAEMO
if (state.loadmap==1) {
    PRIVATE(game)->monkey =
        monkey_new_level_from_file(MONKEY_TEMP, 0);
    state.loadmap = 0;
} else {
    PRIVATE(game)->monkey =
        monkey_new_level_from_file(DATADIR"/monkey-bubble/
        levels",
        level);
}
#endif

/* ... */
}

```

Listing 12.29: maemo-monkey-bubble/src/ui/game-1-player.c

Add function prototype to src/ui/game-1-player.h:

```
void game_1_player_save(Game1Player *game);
```

Listing 12.30: maemo-monkey-bubble/src/ui/game-1-player.h

State Saving Changes in src/monkey

Add save feature to src/monkey/board.c:

```

void
board_save_to_file (Board * board, const char *filename)
{
    #define MUL 4
    GError *error = NULL;
    GIOChannel *channel;
    gint i, j, s=0;
    gchar buffer[COLUMN_COUNT*3+2];
    gsize written = 0;

    if (PRIVATE(board)->bubble_array==NULL) return;

    channel = g_io_channel_new_file (filename, "w+", &error);

    if (channel == NULL) return;

    for (i = 0; i < ROW_COUNT; i++)
    {
        for (j = 0; j < COLUMN_COUNT*MUL; j++)
            buffer[j] = ' ';

        if (i%2==1) {
            s = 2;
        } else {
            s = 0;
        }

        for (j = 0; j < COLUMN_COUNT; j++)
        {
            Bubble *b;

```

```

        Color c;

        if (s>0 && (j+1==COLUMN_COUNT)) break;

        b = PRIVATE(board)->bubble_array[i*COLUMN_COUNT
            +j];
        if (b!=NULL) {
            c = bubble_get_color(b);
            buffer[j*MUL+s] = '0'+(int)c;
        } else {
            buffer[j*MUL+s] = '-';
        }
    }
    buffer[COLUMN_COUNT*MUL]='\n';
    buffer[COLUMN_COUNT*MUL+1]=0;

    g_io_channel_write_chars(channel, (const gchar *)&
        buffer,
                                -1, &written, &error);
}

g_io_channel_shutdown (channel, TRUE, &error);
g_io_channel_unref (channel);
}

```

Listing 12.31: maemo-monkey-bubble/src/monkey/board.c

And function prototype to src/monkey/board.h:

```
void board_save_to_file (Board * board, const char *filename);
```

Listing 12.32: maemo-monkey-bubble/src/monkey/board.h

Provide a call to this function in src/monkey/playground.c:

```

void
playground_save(Playground * self, const gchar * level_filename)
{
    board_save_to_file(PRIVATE(self)->board, level_filename);
}

```

Listing 12.33: maemo-monkey-bubble/src/monkey/playground.c

And function prototype to src/monkey/playground.h:

```
void playground_save(Playground * self, const gchar * level_filename);
```

Listing 12.34: maemo-monkey-bubble/src/monkey/playground.h

Finally, a call to the save feature in src/monkey/monkey.c:

```

void monkey_save(Monkey *monkey, const gchar * filename)
{
    playground_save(PRIVATE(monkey)->playground, filename);
}

```

Listing 12.35: maemo-monkey-bubble/src/monkey/monkey.c

And function prototype to src/monkey/monkey.h:

```
void monkey_save(Monkey *monkey, const gchar * filename);
```

Listing 12.36: maemo-monkey-bubble/src/monkey/monkey.h

Now the game should be saving its state, and be background killable. It can be tested by setting Monkey Bubble to background (for example by starting another application) and issuing following command:

```
dbus-send --system /com/nokia/ke_recv/bgkill_on com.nokia.ke_recv.bgkill_on.bgkill_on
```

After changing back to Monkey Bubble from Task Bar, it should show banner "Monkey Bubble - resuming" and load pre-background map.

12.2.7 Network Changes

For the networking code, no specific changes to the existing code are needed, but in order for the application to provide a network connection selection dialog or to automatically select a proper connection, the LibConIC library must be used.

The best place for network connection changes is before opening the join network dialog. So, some modifications should be made to `src/ui/ui-main.c` file. First add a proper header:

```
#include <conic.h>
```

Listing 12.37: `maemo-monkey-bubble/src/ui/ui-main.c`

After that, modifications are needed in the `new_network_game` function, which is called when a new network game is launched. Remove the old code and replace it with the following:

```
UiMain * uimain = UI_MAIN(callback_data);
PRIVATE(uimain)->ic = con_ic_connection_new();
g_signal_connect(PRIVATE(uimain)->ic, "connection-event", (GCallback)
    network_connected, NULL);
con_ic_connection_connect(PRIVATE(uimain)->ic,
    CON_IC_CONNECT_FLAG_NONE);
```

Listing 12.38: `maemo-monkey-bubble/src/ui/ui-main.c`

The code creates a new `ConIcConnection` and connects the `connection-event` signal. This signal is called, when a connection event happens. After that, it calls the `connect` signal in order to request a new connection. Now, one more thing is needed: to specify what happens, when a connection event comes. That is handled in `new_network_connected` function, which is implemented as follows:

```
static void network_connected(ConIcConnection *cnx,
                             ConIcConnectionEvent *event,
                             gpointer user_data)
{
    UiNetworkClient * ngl;

    switch (con_ic_connection_event_get_status(event)) {
        case CON_IC_STATUS_CONNECTED:
            ngl = ui_network_client_new();
            break;
        case CON_IC_STATUS_DISCONNECTED:
        case CON_IC_STATUS_DISCONNECTING:
        default:
            break;
    }
}
```

```
}
```

Listing 12.39: maemo-monkey-bubble/src/ui/ui-main.c

This function simply checks which status the event gives, and if it is `CON_IC_STATUS_CONNECTED`, it creates a new client window, where the user is able to connect to a server.

There is still one more thing to do. Conic must be added to `configure.in` into the list following `PKG_CHECK_MODULES` for UI.

After these changes, starting a new network game will make sure that there is a network connection available.

12.2.8 Integration to Menu

Application integration to menu needs `.desktop` and `.service` files. This section describes the needed additions and changes. If you are not familiar with `.desktop` and `.service` files, see maemo.org documentation [62] first.

1. Monkey Bubble already has a `monkey-bubble.desktop`, which is generated from `monkey-bubble.desktop.in`, so the changes need to be made to it. First of all, the file should be moved under `./data/` directory, in order to make the common maemo application file structure; the path in `Makefile.am` should be changed accordingly:

```
applications_in_files = data/monkey-bubble.desktop.in
```

And same thing in `po/POTFILES.in`:

```
data/monkey-bubble.desktop.in
```

2. Then the `.desktop.in` file needs to be modified: The `Exec` value must have full path, and two new values must be added:

```
Exec=/usr/bin/monkey-bubble
X-Osso-Type=application/x-executable
X-Osso-Service=monkey_bubble
```

3. Create a service file `com.nokia.monkey_bubble.service` in directory `./data/`:

```
[D-BUS Service]
Name=com.nokia.monkey_bubble
Exec=/usr/bin/monkey-bubble
```

4. Add section of service file in `Makefile.am`:

```
dbusdir=$(prefix)/share/dbus-1/services
dbus_DATA=data/com.nokia.monkey_bubble.service
```

5. Change the proper path for desktop file in `Makefile.am`:

```
applicationsdir = $(datadir)/applications/hildon
```

6. These files must be in `Makefile.am`'s `EXTRA_DIST` variable. The desktop file is there already, so only a service file needs to be added before `$(NULL)`:

```
data/com.nokia.monkey_bubble.service \
```


12.2.9 Application Packaging

This section describes how Monkey Bubble sources were modified to make .deb package building possible.

- Rename Monkey Bubble source directory to maemo-monkey-bubble-0.0.1:

```
mv monkey-bubble-0.4.0 maemo-monkey-bubble-0.0.1
```

- Package source dir maemo-monkey-bubble-0.0.1/ in maemo-monkey-bubble-0.0.1.tar.gz

```
tar czvf maemo-monkey-bubble-0.0.1.tar.gz maemo-monkey-bubble-0.0.1
```

- Go to the source directory

```
cd maemo-monkey-bubble-0.0.1
```

- Set full name environment variable:

```
export DEBFULLNAME="Mr Maemo"
```

Now the actual work can be started. [Debian New Maintainers' Guide](#) [10] might be useful in this phase.

- Make initial debianization:

```
dh_make -e xxxxxxxx.xxxxxx@maemo.org -f ../maemo-monkey-bubble-0.0.1.tar.gz
```

- Next dh_make will ask a question, and print a summary of the package:

```
Type of package: single binary, multiple binary, library, or kernel module?
[s/m/l/k] s

Maintainer name : Mr Maemo
Email-Address   : xxxxxxxx.xxxxxx@maemo.org
Date            : Thu, 15 May 2008 13:11:58 +0300
Package Name    : maemo-monkey-bubble
Version         : 0.0.1
License         : blank
Type of Package : Single
Hit <enter> to confirm:
```

- Modify debian dir. Only files changelog, compat, control, copyright, docs and rules are needed. In the control file, packages libhildon1, librsvg2-2 and libglade2-0 must be added to Depends. You should also set the Section field into user/games to make the package compatible with Application Manager. After that, the package structure is ok. Then the configuration files need to be changed to be able to build the package.

N.B. libglade2 must be installed from [Maemo repositories](#) [55]. ARMEL Debian packages of librsvg2-2 and librsvg2-common are available from maemo.org extras [repository](#).

- Because the key definition dialog was disabled, definitions need to be set in debian/postinst:

```
#!/bin/sh

set -e
```

```

case "$1" in
    configure)

        gconftool-2 -s /apps/monkey-bubble/player_1_shoot --type=
            string Up
        gconftool-2 -s /apps/monkey-bubble/player_1_left --type=string
            Left
        gconftool-2 -s /apps/monkey-bubble/player_1_right --type=
            string Right
        ;;

        abort-upgrade|abort-remove|abort-deconfigure)

        ;;

        *)
            echo "postinst called with unknown argument '$1'" >&2
            exit 1
        ;;
    esac
exit 0

```

Listing 12.40: maemo-monkey-bubble/debian/postinst

It should be set executable with "chmod a+x debian/postinst".

- A link to the desktop file must be added to /etc/others-menu/extra_applications to show it in menu. The link is created by making a debian/maemo-monkey-bubble.links file with following contents:

```

usr/share/applications/hildon/monkey-bubble.desktop
etc/others-menu/extra_applications/monkey-bubble.desktop

```

- Add the file maemo-monkey-bubble.files to list all the files that will be installed by the package.
- Add the following lines in configure.in:

```

PKG_CHECK_MODULES(HILDON, hildon-1 >= 0.14.8)
AC_SUBST(HILDON_LIBS)
AC_SUBST(HILDON_CFLAGS)

```

- Make the following two changes in Makefile.am (remember to use tab to indent):

Add items in EXTRA_DIST before \$(NULL):

```

autogen.sh \
debian/changelog \
debian/compat \
debian/copyright \
debian/control \
debian/rules \
debian/docs \

```

Add deb rule:

```

deb: dist
    -mkdir $(top_builddir)/debian-build
    cd $(top_builddir)/debian-build && tar zxf \
    ../$(top_builddir)/$(PACKAGE)-$(VERSION).tar.gz
    cd $(top_builddir)/debian-build/$(PACKAGE)-$(VERSION) && dpkg-buildpackage \
    -rfakeroot
    -rm -rf $(top_builddir)/debian-build/$(PACKAGE)-$(VERSION)

```

Now the source dir should be ready for packaging, described in section [13.1](#). If you want to create a package that is usable with the Application Manager, see section [13.2](#).

12.3 Maemo Localization

12.3.1 Overview

This section describes how to localize maemo applications. Localization is needed to provide native translations of the software. The section contains information on how to localize an application, how to ease extraction of message strings, how to make the translations and how to test the localization.

Changes are presented with code examples to help the localization process. MaemoPad is used as an example here.

12.3.2 Localization

Localization means translating the application to different languages. Maemo localization is based on the standard gettext package, and all the necessary tools are included in Scratchbox. For applications, localization requires a couple of files in the po/ directory. The following files will be used for localization:

```

Makefile.am
en_GB.po
POTFILES.in

```

POTFILES.in contains the list of source code files that will be localized. File *en_GB.po* includes translated text for target en_GB.

12.3.3 Localizing Application

To localize an application, `l10n` needs to be set up before `gtk_init()` by including the following headers.

```

#include <libintl.h>
#include <locale.h>

```

Listing 12.41: maemopad/src/main.c

To initialize the locale functions, the following lines need to be added:

```

setlocale(LC_ALL, "");
bindtextdomain(GETTEXT_PACKAGE, LOCALEDIR);
bind_textdomain_codeset(GETTEXT_PACKAGE, "UTF-8");
textdomain(GETTEXT_PACKAGE);

```

Listing 12.42: maemopad/src/main.c

The most important of these are GETTEXT_PACKAGE and LOCALEDIR, which come from configure.ac:

```
localedir=/usr/share/locale
AC_PROG_INTLTOOL([0.23])
GETTEXT_PACKAGE=maemopad
AC_SUBST( GETTEXT_PACKAGE )
ALL_LINGUAS="en_GB"
AM_GLIB_GNU_GETTEXT
```

Listing 12.43: maemopad/configure.ac

Of these, only GETTEXT_PACKAGE needs special attention, as it is the l10n domain that is to be used, and ALL_LINGUAS lists the available translations.

12.3.4 Easing Extraction of Strings

In the source code, there are multiple strings that eventually get shown in the user interface. For example:

```
g_set_application_name ( _("MaemoPad") );
```

Listing 12.44: maemopad/src/main.c

In order to make the strings localizable, they need to be wrapped in gettext("String") calls. In practice, writing gettext() for every string is tedious. The common practice is to set the following #define

```
#define _(String) gettext (String)
```

Listing 12.45: maemopad/src/main.c

Thus, the l18n version of the example would be:

```
hildon_app_set_title ( app, _("MaemoPad") );
```

Creating Translation Files

Creating *.po files is quite straightforward. Maemopad only has one localization file, *en_GB.po*; others, such as *fi_FI.po*, can be easily made based on this. Localization .po files are filled with simple structures, defining the localization id and the actual text string, e.g.

```
#: src/document.c:727 src/document.c:733
msgid "note_ib_autosave_recover_failed"
msgstr "Recovering autosaved file failed"
```

"msgid" defines the original string (key) used in code, and "msgstr" defines the translated string for localization.

First, a template file is created with all strings from sources for translation. GNU `xgettext` command is used to extract the strings from sources:

```
xgettext -f POTFILES.in -C -a -o template.po
```

Option `"-f POTFILES.in"` uses `POTFILES.in` to get the files to be localized, `"-C"` is for C-code type of strings, `"-a"` is for ensuring that all the strings are received from sources, and `"-o template.po"` defines the output filename.

This may output some warnings. Usually they are not serious, but it is better to check them anyway.

If the translation in question is into Finnish, the edited `po/template.po` needs to be copied to `po/fi_FI.po`, and `fi_FI.po` needs to be added to `ALL_LINGUAS` in `configure.ac`.

Now `po/fi_FI.po` will include lines such as the following:

```
#: ../src/ui/interface.c:123
msgid "maemopad_save_changes_made"
msgstr "Save changes?"
```

All msgstrings should be translated into the desired language:

```
#: ../src/ui/interface.c:123
msgid "maemopad_save_changes_made"
msgstr "Tallenna muutokset?"
```

Autotools should now automatically convert the `.po` file to `.mo` file during the build process, and install it to the correct location.

To do it manually:

```
msgfmt po/fi_FI.po -o debian/maemopad/usr/share/locale/fi_FI/LC_MESSAGES/domain.mo
```

Where `"debian/maemopad/usr/share/locale/fi_FI/LC_MESSAGES/"` is the directory where the software should be installed to.

Testing

Inside scratchbox, the translated application should be tested by setting `LC_ALL` environment variable to contain the newly created locale (in this example case for Finnish translation, `"fi_FI"`):

```
LC_ALL="fi_FI" run-standalone.sh ./maemopad
```

On the target device, a wrapper script is needed to accomplish this, if the device does not have a locale for the locale identifier used. The script can simply consist of:

```
#!/bin/sh
LC_ALL="fi_FI" /usr/bin/maemopad
```

The example above assumes that the installation package installs the executable into `/usr/bin`. This script needs then to be configured to be run in the application's `.desktop` file, instead of the actual executable.

Chapter 13

Packaging, Deploying and Distributing

For installing and managing application packages, maemo uses the popular Debian package management system. From the user's perspective this is quite invisible, as the package management is performed using the Application Manager. It presents a flexible package framework that enables developers to easily create installable and manageable packages, without having to concentrate on the actual implementation details of the management.

Deb Installation Packages

The Debian package management system uses deb-packages which, in addition to files that will be installed, consist of metadata describing the package, dependencies to other packages and optional installation and removal scripts. The process of creating the packages is fortunately made quite simple by various package development tools.

The process of creating packages is described in section [13.2](#).

Package Repositories

The package management system makes use of package repositories, which are essentially web or FTP sites that contain packages and some information about them. It is not mandatory for a package to exist in any repository, but it has significant advantages: the user can easily update packages, other developers can use the packages as automatically installable dependencies and the repository's packages can be found using the Application Manager.

Information for working with repositories can be found in the Debian Repository HOWTO[\[12\]](#).

One-Click-Install

Maemo also has a option to create installation instruction files that enable the users to install applications simply by clicking a link on a web site. The .install files contain the required repository and package names. The format of these files is really simple, and it is described in section [13.2.12](#).

13.1 Creating Debian Packages

When creating a Debian package, the first step is to put all the files (the source code, and the png and other graphics, desktop and service files) in an empty directory called my-application-1.0.0. The directory name should follow the <package-name>-<app-version> convention. This means that the package that is being created for this application is called my-application.

Go to the source directory:

```
cd my-application-1.0.0
```

Set full name environment variable:

```
export DEBFULLNAME="Your Name"
```

Make initial debianization:

```
dh_make -e your.name@example.org
```

Next dh_make will ask a question and print a summary of the package:

```
Type of package: single binary, multiple binary, library, kernel module or cdbbs?
[s/m/l/k/b] s

Maintainer name : Your Name
Email-Address   : your.name@example.org
Date            : Fri, 30 Nov 2007 13:20:48 +0200
Package Name    : my-application
Version         : 1.0.0
License         : blank
Type of Package : Single
Hit to confirm:
```

The dh_make command creates a debian subdirectory containing multiple configuration text files, most of which are templates that can be removed, since the application does not use them. The table below lists the needed files (others can be removed):

File in ./debian	Description
changelog	Application's change log
compat	Debian helper compatibility version. Leave it as it is.
control	Describes the packages to be made. For more information, see the paragraphs below the table.
copyright	Copyright text. Fill in the blanks.
rules	A makefile containing the rules to build all kinds of packages (such as source and binary)

The key files in ./debian are control and rules. They contain a generic template showing what they must look like. In control, the blanks simply have to be filled in, and in rules, the unwanted and unnecessary code will have to be removed.

The following example illustrates what the control file for the example application must contain:

```

Source: my-application
Section: user/other
Priority: optional
Maintainer: Your Name <your.name@example.org>
Build-Depends: debhelper (>= 4.0.0)
Standards-Version: 3.6.0

Package: my-application
Architecture: any
Depends: ${shlibs:Depends}
Description: A simple test application
 A very simple application with a short description.
 Which spans multiple lines actually.
XB-Maemo-Icon-26:
iVBORw0KGgoAAAANSUUEGAAABoAAAAaCAYAAACpSkzOAAAAABmJLR0QA/wD/AP+g
vaeTAAACXBIWMAAAATAAAEwEAMPwYAAAAB3RJTUUH1gURDQoYya0JlAAAAU9J
REFUSMftLL1KA0EUhb/NZL/ggnHQxsJUxt5CUucVJCCKdfgyKdIGG5/A0s5HEBtJ
EdDAQBgmw0YJmMzgXXYZa5CtNkDW9zZw5z7c+ZCgwb/Ai3i9sVl/Bq8RIs4LRK1
gJdsKvJyNXmJMuYTsMoY1zpg0zaABdYArQNPZQ1kfyGU7SpqVwxzAMwABWhgpIwp
4vWBB+AUWAI3ypjnFEXtPU4bLKx9vErTeCeIRSYF+fTn1j5dp2myE9EiU+DSi3wX
ymeQRQAmZ3EcA5E/fG06BULT8zh0crwXoJdrXRa2Lggs2y2odAUcBUIXQdz78YyC
SIdAp8b7+bXrIv9lqjZBiEtqCc2DjbAt4b2WxJkyZLjVuJlwp0U0cPxuLcAIuC+4
dKxFlsDJarvdAGP/b6hFnDImYs+uG3hb02AB3Jbsur63tQM+fFx3bzZocEB8AdV2
gJBZgKTWAAAAAE1FTkSuQmCC

```

The XB-Maemo-Icon-26 field contains the application icon file (in this case, hello_icon_26x26.png) encoded in base64. This is the icon that is shown in the Application Manager, next to the package name. To perform this encoding in Linux, either uuencode or openssl can be used; however, there may also be more suitable applications. The following example encodes an icon file using uuencode inside Scratchbox:

```
[sbox-DIABLO_X86: ~] > uuencode -m icon.png icon.png > icon.png.en
```

In the above example, the result of the encoding is printed into icon.png.en file. Do not forget to put a white space at the beginning of each line containing the icon-encoded text. The white space indicates a multi-line field. The same rule stands for the long package description (A very simple application[...]).

The Application Manager only shows packages in the user section. Thus, the Section: field in the control file must have the Section: user/<SUBSECTION> syntax, where <SUBSECTION> is arbitrary. See section 13.2.6 for information on the suggested values.

The rules file usually does not need to be changed. However, if the created .deb package is empty, the "install:" section should be checked to make sure that the DESTDIR environment variable is pointing to the right place, i.e. where the rest of the packaging scripts are looking for the application to be packaged.

Once the application is properly 'Debianized', building the application is performed easily with the command:

```
[sbox-DIABLO_X86: ~/my-application-1.0.0] > dpkg-buildpackage -rfakeroot
```

The system displays some output, including a couple of warnings near the end of it (about XB-Maemo-Icon-26), but that is normal. The parent directory now has a my-application_1.0.0-0_i386.deb file - the Debian package. This is the file that is distributed to maemo devices and installed using the Application Manager.

To test the package in the SDK, type:

```
[sbox-DIABLO_X86: ~/my-application-1.0.0] > cd ..
[sbox-DIABLO_X86: ~] > fakeroot dpkg -i my-application_1.0.0-0_i386.deb
```


Replace 'my-application...' with the actual name of the package. Packages can be installed in both X86 and ARMEL targets. The package architecture must match the Scratchbox target.

For more information about making Debian packages, see [Debian New Maintainers' Guide](#) [10]. For further information about creating application packages for maemo, see the next section [13.2](#).



N.B.

If the application is deploying icons to `/usr/share/icons/hicolor`, then the `gtk-update-icon-cache /usr/share/icons/hicolor` command should be run in the `postinst` script, otherwise the icons might not show up until the device is restarted!

13.2 Making Application Packages

This section explains how to make software packages that the end user can install to the Internet Tablet using the Application Manager tool in the device.

13.2.1 Prerequisites

This section assumes familiarity with the process of creating a `.deb` package. The basics of Debian packaging can be found in the previous section [13.1](#).

There is also the example "hello-world-app" package that can be used to get started. It can be installed with `apt-get` using the following command:

```
apt-get source hello-world-app
```

13.2.2 Application Manager

The Application Manager (also known as AM) is an end user friendly graphical front-end to the standard Debian package management infrastructure. When using the Application Manager, the end user does not have to use the `apt-get` tools.

The Application Manager uses the same backend tools as Synaptic, Aptitude, or `apt-get` (namely `libapt-pkg`), and it does it in the standard way, without imposing any constraints: packages are installed as root, and can touch the whole system, for example.

The normal way to distribute a package is, therefore, to put it into a repository and make it accessible to `apt`.

Either Application Manager or the `apt-get` tool can be used freely. Changes made to the system via `apt-get` or `dpkg` are picked up by the Application Manager without confusing it, and vice versa.

13.2.3 Packaging

Packages made for the Application Manager should follow a few extra rules, if they want to integrate nicely to the device. These rules are related to:

- general stuff
- dependencies
- sections
- icons
- the installation/removal policy of the Application Manager
- limited feedback to the user, no controlling terminal
- warning about removing running applications
- utilities to use in the maintainer scripts

These are explained in detail below.

13.2.4 General

All strings coming from the control information of a package are interpreted in UTF-8 when they are shown in the UI. If a string is not valid UTF-8, all bytes above 127 are replaced with '?' before displaying it.

13.2.5 Dependencies

This field should now contain all the dependencies for package, such as `${shlibs:Depends}`.

13.2.6 Sections

By default, the Application Manager only shows to the user packages in certain sections. This has been done to hide the existence of the hundreds of system packages making up the IT OS itself. The AM is, at this point, not intended to let the user manage the whole system, only a smaller set of third-party applications.

The AM only shows packages in the "user" section. Thus, the "Section:" field in the control file should be of the form

```
Section: user/<SUBSECTION>
```

where SUBSECTION is one of:

- user/accessories Accessories
- user/communication Communication
- user/games Games
- user/multimedia Multimedia
- user/office Office
- user/other Other
- user/programming Programming
- user/support Support

- user/themes Themes
- user/tools Tools

Thus, if the package is wanted in the Office subsection, the field Section: user/office should be included in the control information.

13.2.7 Icons

A package can have an icon displayed next to its name by the AM. Icons are included in the control information of a package as a base64 encoded field named "Maemo-Icon-26".

The image format of the icon can be anything understood by GdkPixbufLoader, but most commonly the PNG format is used.

The image should be 26x26 pixels with a transparent background.

The hello-world package shows an example of this.

The way to get these fields into the .deb files is to include them with a "XB-" prefix in the debian/control file. For more information, see the [Debian Policy Manual, section 5.7](#) [11].

13.2.8 Installation and Removal Policy

The Application Manager has its own rules for automatically installing and removing packages, in addition to the ones specified by the user. These rules are tuned to eliminate most surprises for simple package management operations, but this makes them less useable for complicated things like "apt-get dist-upgrade". When designing the conflicts of packages, these rules should be accounted for.

Specifically, the AM will never automatically remove a user package.

If a conflict caused by installing a package could be resolved by removing a package, the AM will not do the removal, but will refuse the installation request instead.

When removing a package, all packages that are a direct or indirect dependency of the removed package will be considered for removal. They will, in fact, be removed when they are a non-user package, have been automatically installed by the AM to satisfy a dependency, and are no longer needed.

Unfortunately, the AM is not very good in reporting conflicts: when the package conflicts with a non-user package, the problem reported by the AM will blame the conflict on that non-user package, instead of on the user packages that depend on it.

13.2.9 Feedback from Maintainer Scripts

When the Application Manager runs the maintainer scripts, they have no controlling terminal; their standard input is connected to /dev/null. The DISPLAY variable is set correctly.

The Application Manager collects a transcript of the installation/uninstallation process, including the output of maintainer scripts. However, this output is hidden away in the "Log", and users should not be expected to look there and understand its contents.

Thus, it is the responsibility of the package creator to make sure that the maintainer scripts will not fail. This naturally does not mean that errors should be ignored, but that only things that have a very high chance of succeeding should be done. The simpler, the better.

13.2.10 Removing or Upgrading Running Applications

The Application Manager can run a script provided by the package, before removing or upgrading it. That script can tell the Application Manager to cancel the operation.

The canonical use for this feature is to warn the user when they try to remove or upgrade an application that is currently running. The utility 'maemo-application-running' can be used to perform this test. (See below for details.)

When uninstalling or upgrading a package named PACKAGE, the Application Manager will run the program named /var/lib/hildon-application-manager/info/PACKAGE.checkrm, if it exists. If this program exists with code 111, the operation will be canceled. In all other cases, including when the program terminates with a signal, the operation is carried out.

The arguments given to the *.checkrm program are either:

```
foobarexampleonly.checkrm remove
```

when the package is going to be removed, or

```
foobarexampleonly.checkrm upgrade VERSION
```

when it is going to be upgraded to version VERSION

13.2.11 Utilities to Use in Maintainer Scripts

There are some utilities available that can be used in the maintainer scripts to interact with the user:

```
maemo-select-menu-location <app>.desktop [default-folder]
```

When the package contains a .desktop file, and consequently has an entry in the Desktop menu for this file, it can call maemo-select-menu-location in its postinst script to let the user choose a location for the entry.

The "app.desktop" parameter is the name of the .desktop file, without any directories. The default-folder parameter is optional, and when given, determines the default folder of the menu entry. If omitted, the menu entry will appear in "Extras".

The way to specify a folder that is provided by the system is by giving its logical name as listed in the /etc/xdg/menus/applications.menu file, NOT by giving its English name. Examples of logical names are

```
tana-fi_games
tana-fi_tools
tana-fi_utilities
```

When using a folder name that does not yet exist, it will be created. In that case, a logical name should NOT be used, since there will likely be no translations available for that logical name. When creating a new folder, a plain-text name should be used, in an appropriate language. However, it is advisable to use existing folders as much as possible.

Thus, if the package installs the file `/usr/share/applications/hildon/foobarexampleonly.desktop`, and it is wanted to go to the "Utilities" menu, this invocation should be put into the postinst script:

```
maemo-select-menu-location foobarexampleonly.desktop tana-fi_utilities
```

If it is wanted to go into a non-existing folder, the code used should be something like

```
maemo-select-menu-location foobarexampleonly.desktop "Cute hacks"
```

In order to use `maemo-select-menu-location` in postinst, `Depends` should be included in the "maemo-select-menu-location" package.

It is advisable to skip calling `maemo-select-menu-location`, when merely upgrading, as opposed to installing from scratch.

```
maemo-application-running -x executable-file
maemo-application-running -d app.desktop
```

This utility checks whether the application specified on the command line is currently running. If it is running, it exits with code 0. If it is not running, it exits with code 1. If an error occurs, it exits with code 2.

When using the `-x` option, the utility checks whether any process is currently executing that file, by looking into `/proc/PID/exe`.

When using the `-d` option, the utility uses the given `.desktop` file to find the service name of the application and queries D-BUS whether this service is currently registered. If there is no service name in the `.desktop` file, the utility uses the executable file as with the `-x` option.

In order to use `maemo-application-running` in postinst, `Depends` should be included in the "maemo-installer-utils" package.

```
maemo-confirm-text [title] file
```

Displays the contents of `FILE` in a dialog with "Ok" and "Cancel" buttons. The default title of the dialog is "License agreement".

When the user clicks "Ok", this utility exits with code 0; when they click "Cancel", it exits with code 1; and when an error occurs, it exits with code 2.

If a license agreement is not shown in the postinst script, it is probably not a good idea to make the postinst script fail when the user does not agree to the license terms. Instead, the application could be configured in such a way that it will ask the user to agree to the license agreement again when the application is started, and refuse to run when they disagree.

In order to use `maemo-confirm-text` in postinst, `Depends` should be included in the "maemo-installer-utils" package.

13.2.12 Controlling Installation

In Application Installer (old name for Application Manager), there was a beta stage Single Click Install feature. Application Manager is backwards compatible with it, so the old `.install` files still work, but there are some new features available. This section will give an overview of the `.install` file usage and functionality.

The Application Manager offers three kinds of functionality with `.install` files: adding catalogues, installing packages from remote catalogues and installing packages from a memory card. The `.install` file follows the `GKeyFile`

format, as described by GLib. A GKeyFile consists of a number of groups, and each group contains key/value pairs. Each of the previously mentioned three functionalities is initiated by a specifically named group in the .install file, as described in the next chapters.

Adding Catalogues

Adding catalogues is performed with the "catalogues" group. This group has one similarly named mandatory key, "catalogues". The "catalogues" key is a list of strings referring to the catalogue groups that describe the catalogues to be added. Each catalogue is considered in turn, and the user is asked whether to add it or not. If it should be added, and a catalogue is already configured in the Application Manager that is equal to the one considered, the configured catalogue is removed first. When the user declines the adding, the next catalogue is considered. After considering every catalogue, the user is asked whether to "Refresh the list of applications". Here is an example:

```
[catalogues]
catalogues = extras; sdk

[extras]
name = maemo Extras catalogue
uri = http://repository.maemo.org/extras
components = free non-free

[sdk]
name = maemo SDK catalogue
uri = http://repository.maemo.org/
components = free non-free
```

A Group describing a catalogue can contain the following keys:

- **name:** This key gives the display name of the catalogue as shown to the user in the "Tool > Application catalogues" dialog. This key can be localised by following the GKeyFile conventions. If this key is omitted, the catalogue will have an empty name.
- **uri:** The URI part of the deb line that will be added to sources.list for this catalogue. This key is required for all catalogues except those used with the "card_catalogues" key.
- **file_uri:** When using file_uri instead of uri, the URI part of the deb line will use the "file://" method and the file_uri gives the actual pathname, relative to the location of the .install file. This key is required for all catalogues that are used with the "card_catalogues" key.
- **dist:** The distribution of the deb line that will be added to sources.list for this catalogue. If it is omitted, it will default to the distribution corresponding to the IT OS release on the device. The fact that the distribution should be selected automatically is remembered by the Application Manager. For example, if a backup is made, containing a catalogue with automatic distribution selection, and restored on a different IT OS release, the distribution for the new version will be used automatically.
- **components:** The components part of the deb line that will be added to sources.list for this catalogue. If it is omitted, the components part will be empty.

- `filter_dist`: This catalogue will be ignored when the distribution corresponding to the IT OS release on the device does not match.

When catalogues are compared, they are considered equal when their `uri`, `dist` and `components` fields are equal.

Installing Packages

In order to install packages with an `.install` file, the file must have an "install" group. The group has one mandatory key, "package", and one optional key, "catalogues". The "catalogues" key is handled as follows: each catalogue is considered in turn. The user is asked for confirmation if a catalogue that is not already configured is encountered. Alternatively, when a catalogue is already present but disabled, the user is informed that it needs to be enabled and is asked for a confirmation. If the user declines, the processing of the `.install` file stops, and the changes to the configured catalogues that have been made for it are reverted. After the list of catalogues has been processed, the list of applications is refreshed automatically, and the package is offered to the user for installing. Here is an example:

```
[install]
catalogues = foobar
package = maemofoo

[foobar]
name = Foobar Catalogue
name[en_GB] = Foobar Catalogue
name[de_DE] = Foobar Katalog
uri = http://foobar.com/repository
components = main
```

Installing Packages from Memory Card

Installing packages from a memory card is governed by the "card_install" group. It has two mandatory keys, "packages" and "card_catalogues", and an optional one, "permanent_catalogues". The "packages" key lists the names of packages that can be installed from the memory card, using the "card_catalogues". Installation of the packages happens in a temporary environment: in this environment, the normally configured catalogues are not available, only the catalogues listed by the "card_catalogues" key are configured. All of these catalogues must use "file_uri" instead of "uri".

The user gets to select the packages from a list, after which the installation proceeds one package after another. In case the installation of a package fails, an error note is displayed, and the processing stops. After the installation has completed, the optional group "permanent_catalogues" is processed. It functions similarly as the previously described catalogue adding.

Whenever a memory card is inserted that contains a file called `.auto.install`, that file is processed by the Application Manager. Usually, the `.auto.install` file contains a "card_install" group, of course. For example:

```
$ mkdir .repo
$ cp somewhere/*.deb .repo/
$ (cd .repo && apt-ftparchive packages . >Packages)
```

A matching `.install` file could look like this:

```
[card_install]
card_catalogues = repo
packages = app-1; app-2

[repo]
file_uri = .repo
dist = ./
```

The .repo and the .install files typically go to the root of the memory card. To make the memory card auto-installing, make a copy of the .install file and name it .auto.install.

13.2.13 Signing Package

In order to sign packages, use debsign tool inside Scratchbox. Please refer to [debsign manual page \[13\]](#) for instructions on how to use the tool.

13.3 Deploying Packages

13.3.1 Installing Application Packages to SDK

When installing Application Manager package to maemo SDK, debian package installer, like dpkg, should be used:

```
[sbox-DIABLO_X86: ~] > fakeroot dpkg -i application_i386.deb
```

Replace 'application' with the actual name of the package. Packages can be installed in both PC and ARMEL targets. The package architecture must match the Scratchbox target.

13.3.2 Installing Application Packages to Device

Debian package installer can also be used when installing application packages in maemo devices.

```
BusyBox v1.6.1 (2008-03-06 11:36:58 EET) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

/home/user # dpkg -i application_armel.deb
```

For maemo device the package architecture must be always armel.

13.3.3 Red Pill Mode

The Hildon Application Manager has a special hidden mode that makes it more power user friendly, and gives access to features that are not yet considered to be ready for everybody.

Activation

Go to "Tools > Application catalogue", click "New", enter "matrix" into the "Web Address" field, click "Cancel". Choosing the red pill will activate the red pill mode, obviously, and choosing the blue one will deactivate it.

Settings

After activating the red pill mode, the following additional settings are available in "Tools > Settings".

- *Clean Apt Cache*
If activated, the equivalent of `apt-get clean` is performed after every install or update. (This is the behavior for blue pill mode.)
- *Assume Net Connection*
This will not ask for an active IAP before downloading. This is useful when there is a network connection, but the device connectivity APIs are not available or do not know about it.
- *Break Locks*
This will break needed locks instead of failing. This is done by default in blue pill mode, so that users would not lock themselves out, when a crash leaves a stale lock behind.
- *Show Dependencies*
This adds another tab to the details dialog with some dependencies from the package.
- *Show All Packages*
This will not filter out packages that are not in the "user" section. It will also allow installing packages from any section.
- *Show Magic System Package*
This will include the "magic:sys" package in the list of packages. Updating that package will do something similar to `apt-get upgrade`. It is not yet fully defined what it will do exactly. This feature might become available in blue-pill mode at one point.

Chapter 14

Debugging

14.1 Introduction

The following code examples are used in this chapter:

- [gdb_example](#)
- [MaemoPad](#)
- [valgrind_example](#)

14.2 Maemo Debugging Guide

This section is targeted at beginner-level maemo developers needing to know how to perform debugging in the maemo environment.

This section covers the two basic debugging tools available in the maemo environment, and shows how to use them. The tools are:

- **gdb** - The Gnu Project Debugger. General tool for various debugging needs.
- **valgrind** - Debugger and profiler. Valgrind works only in the X86 environment under Scratchbox, so this tool cannot be used in the device itself.

This section assumes that the developer already knows how to:

- develop software in the Linux environment using the C language
- install software to the tablet device
- gain root access to the device
- install ssh to the device
- configure repositories in the `/etc/apt/sources.list` file
- set up USB networking between a Linux PC and the Tablet (the device can also be used over a local WLAN connection instead)
- work with the Scratchbox environment and Scratchbox targets

14.2.1 Pre-Requisites

To follow the debugging examples the following is needed:

- maemo SDK installed in a Linux PC
- Nokia Internet Tablet device running OS2008.
- USB cable to connect the device with the Linux PC
- Internet access both for the tablet and for the Linux PC
- USB networking (or WLAN) set up between the Linux PC and the device
- root login access to the device over ssh
- package maemo-sdk-debug installed
- osso-xterm installed in the device
- ssh software installed in the device

14.2.2 General Notes on Debugging

Debugging Issues on ARM Architecture

There are some issues one needs to be aware of when debugging on the ARM architecture.

- To make backtraces work properly in ARM side, the dbg packages need to be installed for the libraries the application is using. Profiling and debugging (gdb) tools require code to have either framepointers or debugging symbols to unwind stack. This is needed for showing backtraces or call graphs.
- C language functions with the `__attribute__((__noreturn__))` statement need to be compiled with the gcc option: `-fno-omit-frame-pointer`. Without framepointers, no backtrace can be gotten through "noreturn" functions. In practice, what would happen is that when the bt command is used, this kind of function would repeat infinitely.
- In addition, for the gdb to be able to display the correct function names during debugging, it also needs to have access to the debug symbols. Without them, it shows preceding exported function name for a given address.

Debugging Issues in Scratchbox

It is recommended to use the native gdb in the target, not the Scratchbox host-gdb.

When debugging threads in an application, gdb needs to be linked against the same thread library that the application is using. For this reason, the Scratchbox-provided gdb is not suitable for threads debugging, but the native gdb needs to be used instead. See instructions in the next section on how to start using the native gdb.



N.B.

The above-mentioned problem can produce "warning: Cannot initialize thread debugging library: unknown thread_db error '22'" messages in gdb output, and info threads command in gdb will show nothing.

14.2.3 Using Gdb Debugger

The Gnu Project Debugger, or gdb for short, is a general purpose debugger that can be used for various debugging purposes.

This section does not explain how to use the gdb debugger itself, i.e. it does not explain the specific commands in gdb to perform some specific actions. There are other tutorials and documentation readily available in the [Internet](#) for that purpose. This section focuses on explaining how to set up and perform the basic debugging steps with the gdb in the maemo environment.

For additional gdb documentation, take a look at the maemo tools documentation for [gdb](#).

Setting up Environment

Both the Internet Tablet device, described in section *Setting up USB Networking 3.5*, and the Scratchbox environment, described in section *Installing SDK 3.3*, both in chapter *Development Environment* of the Maemo Reference Manual, need to be set up.

Preparing Scratchbox Environment for Debugging If the maemo SDK is not yet installed, it must be installed before continuing.

After installing maemo SDK, the default target names are armel and x86. These will be used in this example.



N.B.

The Scratchbox provides a gdb debugger. If just [sbox-x86: ~] > gdb ... is run, then that gdb debugger is used. In this material, the native-gdb (i.e. non-Scratchbox version of gdb) is used.

Next, install gdb to the Scratchbox from the maemo repositories.

```
[sbox-x86: ~] > fakeroot apt-get install gdb
[sbox-x86: ~] > fakeroot apt-get install maemo-debug-scripts
[sbox-x86: ~] > fakeroot apt-get install maemo-sdk-debug
```

Now two gdb programs are installed. The one used can be checked with:

```
[sbox-x86: ~] > which gdb
/targets/links/arch_tools/bin/gdb
```

If only using the command gdb, then the gdb used is the one provided by Scratchbox.

Start briefly both gdb debuggers to see that they start properly in the Scratchbox environment. First just run gdb:

```
[sbox-x86: ~] > gdb
GNU gdb 6.4.90
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
(gdb) q
[sbox-x86: ~] >
```

Then run the maemo version of gdb:

```
[sbox-x86: ~] > native-gdb
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(gdb) q
[sbox-x86: ~] >
```

To quit from gdb, type 'q' and hit ENTER.

The native-gdb is used here, since it allows also threads to be debugged.

Both gdb versions are available in the Scratchbox X86 environment. The native-gdb should always be run instead of the Scratchbox version.

Preparing Internet Tablet for Debugging



N.B.

Because additional software is going to be installed, and debugging will be performed in the real device that is running the official Sales Image, it is recommended that a full back-up of the device and any important files is made before continuing.

The gdb needs to be installed in the Internet Tablet device. It is recommended that first osso-xterm and then ssh are installed in the device before continuing.

1. Start osso-xterm.
2. Gain root access to the device while running osso-xterm.
3. Edit the /etc/apt/sources.list file in the device so that it contains line:

```
deb http://repository.maemo.org/ diablo free non-free
```

The /etc/apt/sources.list file can be edited with a text editor (e.g. vi), but the following command given from the osso-xterm is also valid:

```
echo "deb http://repository.maemo.org/ diablo free non-free" >> /etc/apt/sources.list
```

Important: it is not recommended to perform device software updates from the maemo sdk repositories (for example, 'apt-get upgrade' is not recommended for the device). The reason for this is that there might be some software packages in the SDK repositories that are so-called *sdk*

variants. They might create a problem, if directly installed in the actual device. In this example, only gdb software is used from the repository.

4. Perform an `apt-get update` in the device. The update command will refresh the package list database in the device.
5. Perform an `apt-get install gdb`

You should now have the `gdb` and `gdbserver` (included in the `gdb` package) installed in the device.



N.B.

After using the `maemo sdk` repositories in the device `/etc/apt/sources.list` file, remove or comment the line out. This way, you will not accidentally get programs from the wrong repository to the device. See notes above.

Debugging Use Cases with Gdb

Debugging Command Line Application in Scratchbox X86 Environment with Gdb One of the most common debugging case is performing debugging in the Scratchbox X86 environment.

By now, the `gdb` is installed in the `x86` target. Next, download the [gdb_example](#) sources.

The example apps are:

- the `gdb_example.c` is a very simple C application that has some functions that call each other in a row. This is used here to demonstrate how to get backtraces.
- the `gdb_example2.c` is a simple variant of the `gdb_example.c` that has some additional `sleep()` calls. This will be used to demonstrate simple core dump debugging.

Next, the small `gdb_example.c` file should be compiled as shown below, and the `gdb` debugger should be started. This simple example shows how to set breakpoints, and how to get a backtrace from the program. Backtrace tells what functions have been called and what parameters have been used.

Compile the `gdb_example.c` application with the `-g` option like this:

```
[sbox-x86: ~/src/testing/gdb_example] > gcc gdb_example.c -o gdb_example -g
```

Next, start the `gdb` with the `gdb_example` application as a parameter.

```
[sbox-x86: ~/src/testing/gdb_example] > native-gdb gdb_example
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
```

Set the breakpoint (`br`) to function `example_3`

```
(gdb) br example_3
Breakpoint 1 at 0x80483e9: file gdb_example.c, line 48.
```

Run the application under gdb giving a command line parameter foobar to the application.

```
(gdb) run foobar
Starting program: /home/user/src/testing/gdb_example/gdb_example foobar
Now in main().
Now in example_1(2) function.
Now in example_2(parameter here) function.

Breakpoint 1, example_3 () at gdb_example.c:48
48     printf("Now in example_3()\n");
```

The above shows that the running of the application stopped in function `example_3`. This happened, because the breakpoint (`br example_3`) was set.

Now the backtrace (`bt`) can be listed from application to see what functions have been called. The list goes from recent to older. The oldest function was naturally the `main()` function at the end of the list. It shows also the parameters to the called functions.

```
(gdb) bt
#0  example_3 () at gdb_example.c:48
#1  0x080483dc in example_2 (a=0x80484f8 "parameter here") at gdb_example.c:39
#2  0x080483b9 in example_1 (x=2) at gdb_example.c:30
#3  0x08048387 in main (argc=2, argv=0xbfc437e4) at gdb_example.c:18
(gdb)
```

It can be convenient to see what the source code is for a line mentioned in the output:

```
(gdb) list 39
34     *
35     */
36     int example_2(char* a)
37     {
38         printf("Now in example_2(%s) function.\n",a);
39         example_3();
40         return 0;
41     }
42     /*
43     */
(gdb)
```

Also the values of the variables can be inspected:

```
(gdb) br example_2
Breakpoint 1 at 0x80483c4: file gdb_example.c, line 38.
(gdb) run
Starting program: /home/user/src/testing/gdb_example
Now in main().
Now in example_1(1) function.

Breakpoint 2, example_2 (a=0x80484f8 "parameter here") at gdb_example.c:38
38     printf("Now in example_2(%s) function.\n",a);
```

To see the value of variable 'a' just type:

```
(gdb) print a
$1 = 0x80484f8 "parameter here"
(gdb)
```

Essentially, debugging with gdb in the Scratchbox X86 target is similar to debugging with gdb in any Linux host. In this example, only a small subset of gdb's functionality has been used.

Debugging Command Line Application in Internet Tablet Device It is possible to debug an application in the Internet tablet device itself using gdb. Before starting, log in on the device and install (if you have not done so yet) the gdb debugger to the device:

```
/home/user # apt-get install gdb
... etc ...
/home/user #
```

Here it is assumed that ssh and osso-xterm are already installed in the tablet and that it is possible to log in on the device using ssh from a Linux PC. In addition, the maemo repository entries should be set in the /etc/apt/sources.list file as explained previously.

Debugging with the gdb debugger in the device is similar to using gdb in a normal Linux PC environment. The limitations are mostly related to the available free RAM memory, meaning that in the worst case, memory might run out while trying to debug an application in the device.

This example follows the basic logic of the first example, but this time is performed in the device.

1. Compile the gdb_example.c application in the Scratchbox for armel architecture.

```
[sbox-x86: ~] > sb-conf select armel
Hangup
Shell restarting...
[sbox-armel: ~] > pwd
/home/user
[sbox-armel: ~] > cd src/testing/gdb_example
[sbox-armel: ~/src/testing/gdb_example] > gcc gdb_example.c -o gdb_example -g
[sbox-armel: ~/src/testing/gdb_example] > file gdb_example
gdb_example: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for GNU/Linux 2.6.8,
dynamically linked (uses shared libs), not stripped
[sbox-armel: ~/src/testing/gdb_example] >
```

2. Copy the armel version of the gdb_example to the tablet. You need to have the sshd daemon up and running in the device before you can copy files with scp.

```
[sbox-armel: ~/src/testing/gdb_example] > scp gdb_example user@192.168.2.15:
user@192.168.2.15's password: .....
gdb_example                                100% 9135      8.9KB/s   00:00
```

3. Log in on the device with ssh with username user. The IP address is an example, and your device IP address can be different.

```
[sbox-armel: ~/src/testing/gdb_example] > ssh root@192.168.2.15
root@192.168.2.15's password: .....

BusyBox v1.6.1 (2007-09-27 18:08:59 EEST) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

~# cd ~user
```

4. Start the gdb debugger with the gdb_example application.


```

/home/user# gdb ./gdb_example
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "arm-linux-gnueabi"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb)

```

You should now be able to debug this small example application in a similar way as in the previous example.

Debugging Core Files in Device This section explains how to debug core files in the Internet Tablet.

The kernel does not dump cores of a failing process, unless told where to dump the core files.

The default values for core dumps can be seen by reading the file `linuxrc` with the `more` command in the device, and by studying the `enable_coredumps()` function:

```

/home/user # more /mnt/initfs/linuxrc
... snip ...
enable_coredumps()
{
    coredir=/media/mmc1/core-dumps
    echo -n "Enabling core dumps to $coredir/..."
    echo "$coredir/%e-%s-%p.core" > /proc/sys/kernel/core_pattern
    ulimit -c unlimited
    echo "done."
}
... snip ...

```

As can be seen, the default location for core dumps is in `/media/mmc1/core-dumps` directory. The second `echo` command defines the name of the core dump file. The default name contains the name of the executable (`%e`), the signal (`%s`) and the PID number (`%p`).

If these default settings are satisfactory, it is enough to create the core-dumps directory under the `/media/mmc1` directory.

Next, the small example application `gdb_example2` will be used to demonstrate how to debug the core file.

1. Compile the `gdb_example2` in the Scratchbox armel target, and just copy the file to the device using `scp`. Now, unplug the USB cable to get access to memory cards. Then start the `gdb_example2` in the device like this:

```

/home/user # ./gdb_example2 &
/home/user # gdb_example2.
Now in main().
Now in example_1(1) function.

```

2. The `gdb_example2` is now running in the background, and it starts to dump its output to the screen. There are some `sleep()` calls in the `gdb_example2`, so that you have time to kill it with the `SIGSEGV` signal. Now just make it to generate a core dump. Assuming that the process is referred to as `%1`, just use the `kill` command as below and press the ENTER key a couple of times:

```
/home/user # kill -SIGSEGV %1
/home/user #
[1] + Segmentation fault (core dumped) ./gdb_example2
```

3. You should now have a compressed core dump data file under the `/media/mmc1/core-dumps` directory. Its name should include the name of the file and end with the PID number of the `gdb_example2` program. Check that you got it (the number shown below, 1390, will naturally be different in your environment):

```
/home/user # ls -l /media/mmc1/core-dumps/
-rw-r--r-- 1 user root 139264 Nov 9 13:09 gdb_example2-11-1390.rcore.lzo
```

4. You have to extract the compressed data before you can use it with gdb. You need to install `sp-rich-core-postproc` package and run the `extract` command which creates a new directory (given as a second parameter) where to extract data:

```
/home/user # apt-get install sp-rich-core-postproc
... snip ...
/home/user # rich-core-extract /media/mmc1/core-dumps/gdb_example2-11-1390.rcore.lzo \
coredir
```

The newly-created `coredir` includes lots of information in different files about the system. The file to pass for the gdb is named `coredump`.

5. The `gdb_example2` is linked against the `libc` library. If you want to be able to resolve symbols also for the library during debugging, you need to install `libc6-dbg` package in the device. *The same rule applies to other libraries that your application might be linked against.* See the further notes about the DBG packages in this material.

Now install the `libc6-dbg` package to get symbols for the library.

```
/home/user # apt-get install libc6-dbg
Reading Package Lists... Done
Building Dependency Tree... Done
The following NEW packages will be installed:
libc6-dbg
.... snip ...
/home/user #
```

6. Now you can debug the core file together with the `gdb_example2` binary that you compiled with the `-g` flag. Try and see where the execution of the `gdb_example2` was when you used the `-SIGSEGV` signal. Start the gdb and give the `gdb_example2` as the first parameter, and the core file as the second parameter:

```

/home/user # gdb ./gdb_example2 coredir/coredump
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "arm-linux-gnueabi"...
Using host libthread_db library "/lib/libthread_db.so.1".
Reading symbols from /lib/libc.so.6...BFD: /usr/lib/debug/lib/libc-2.5.so:
warning: sh_link not set for section '.ARM.exidx'
Reading symbols from /usr/lib/debug/lib/libc-2.5.so...done.
done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.3...BFD: /usr/lib/debug/lib/ld-2.5.so:
warning: sh_link not set for section '.ARM.exidx'
Reading symbols from /usr/lib/debug/lib/ld-2.5.so...done.
done.
Loaded symbols for /lib/ld-linux.so.3
Core was generated by './gdb_example2'.
Program terminated with signal 11, Segmentation fault.
#0  0x410adcec in nanosleep () from /lib/libc.so.6

```

The example above shows that now gdb is using debug symbols from /usr/lib/debug/lib/libc-2.5.so. Had the libc6-dbg package not been installed, the gdb would not have information available about the libraries' debug symbols.

Now, the gdb is waiting for a command, so just give the bt (backtrace) command. You should see something similar to this:

```

(gdb) bt
#0  0x410adcec in nanosleep () from /lib/libc.so.6
#1  0x410ada54 in sleep () from /lib/libc.so.6
#2  0x0000846c in example_2 (a=0x8570 "parameter here") at gdb_example2.c:41
#3  0x0000842c in example_1 (x=1) at gdb_example2.c:32
#4  0x000083e0 in main (argc=1, argv=0xbee57e24) at gdb_example2.c:19

```



N.B.

For this simple example, the available libc6-dbg package was installed before starting to debug the gdb_example2 application. If the dbg packages for various libraries were not installed, it would mean that the backtrace information coming from the non-debug version of the library could not be trusted.

Depending on at what point the kill -SIGSEGV command was given, the output may be different. In this example, the process was hit when it was calling the sleep function inside the example_1 function in file gdb_example2.c. It can also be seen that the sleep() function has further called the nanosleep() function, and that is when it got the -SIGSEGV signal (see the note above).

Debugging Core File from Device Inside Scratchbox Because the binaries and libraries in the device are prelinked, successful debugging inside the Scratchbox environment using the programs core file (from the device) would require that:

- you copy the relevant libraries (i.e. the ones the application is using) from the device to the Scratchbox. Otherwise addresses in the prelinked and

unprelinked libraries would not match, and gdb backtraces would not load the library.

- If the core file debugging is performed in the Scratchbox ARMEL environment, the native gdb program needs to be used instead of the one provided by Scratchbox. See the previous section on how to set the native gdb as the default one.

After copying the `/lib/libc6` library from the device and setting the native gdb to be used, the application can be debugged normally.



N.B.

Keep in mind that generally the Scratchbox tools override the target tools.

Debugging UI Applications in Scratchbox X86 Many maemo applications use the graphical UI, and debugging these applications differs slightly from debugging simple command line applications.

Here the maemopad will be used as an example application to debug in the X86 target with gdb.

If the Scratchbox environment is set up correctly as explained above, these steps should be easy to follow to perform UI debugging with maemopad application.

1. Activate the X86 target, go to the testing directory and download the source package of maemopad application.

```
[sbox-armel] sb-conf select x86
[sbox-x86] cd ~/src/
[sbox-x86 ~/src] mkdir maemopad
[sbox-x86 ~/src/maemopad] cd maemopad
[sbox-x86 ~/src/maemopad] apt-get source maemopad
Reading package lists... Done
Building dependency tree... Done
Need to get 384kB of source archives.
Get:1 http://juri.research.nokia.com diablo/free maemopad 2.3 (dsc) [476B]
Get:2 http://juri.research.nokia.com diablo/free maemopad 2.3 (tar) [384kB]
Fetched 384kB in 0s (4637kB/s)
dpkg-source: warning: extracting unsigned source package (./maemopad_2.3.dsc)
dpkg-source: extracting maemopad in maemopad-2.3
dpkg-source: unpacking maemopad_2.3.tar.gz
```

2. Check that you have the correct files.

```
[sbox-x86: ~/src/maemopad] > ls -l
total 388
drwxrwxr-x  6 maemo maemo  4096 Sep 26 15:00 maemopad-2.3
-rw-rw-r--  1 maemo maemo   476 Nov  8 17:26 maemopad_2.3.dsc
-rw-rw-r--  1 maemo maemo 383834 Nov  8 17:26 maemopad_2.3.tar.gz
```

3. Go to the `maemopad-2.3` source directory, and set the `DEB_BUILD_OPTIONS` environment variable, so that the generated binaries are not stripped. Then build the maemopad package with `dpkg-buildpackage` command as shown here:

```
[sbox-x86: ~/src/maemopad] > cd maemopad-2.3
[sbox-x86: ~/src/maemopad/maemopad-2.3] > export DEB_BUILD_OPTIONS=debug,\
nostrip
[sbox-x86: ~/src/maemopad/maemopad-2.3] > dpkg-buildpackage -rfakeroot -d
dpkg-buildpackage: source package is maemopad
dpkg-buildpackage: source version is 2.3
dpkg-buildpackage: source changed by Maemo Integration <integration@maemo.org>
dpkg-buildpackage: host architecture i386
dpkg-buildpackage: source version without epoch 2.3
: Using Scratchbox tools to satisfy builddeps
.... etc ....
```



N.B.

In this example, the standard export DEB_BUILD_OPTIONS=debug,nostrip environment variable is used, but there might be source packages that do not support these debug,nostrip options. In that case, one must make sure that the source is compiled with -g flag (usually this option can be added to the CFLAGS variable in the debian/rules file), and that the produced binaries will not be stripped. In the long run, it is better to modify the source package to generate a separate debug symbol (-dbg) package. This requires modifying both the debian/rules and debian/control files.

4. You should now have maemopad binaries generated so that they have the debug symbols in them. Check that you get a not stripped flag from the maemopad binary:

```
[sbox-x86: ~/src/maemopad/maemopad-2.3] > file debian/maemopad/usr/bin/maemopad
debian/tmp/usr/bin/maemopad: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), for GNU/Linux 2.6.0, dynamically linked (uses shared libs), not stripped
```

5. You should now have a maemopad_2.3_i386.deb file in the ~/src/maemopad directory. Check that you have it:

```
[sbox-x86: ~/src/maemopad] > ls -l
total 440
drwxrwxr-x 6 maemo maemo 4096 Nov 9 13:37 maemopad-2.3
-rw-rw-r-- 1 maemo maemo 476 Nov 9 13:36 maemopad_2.3.dsc
-rw-rw-r-- 1 maemo maemo 388593 Nov 9 13:36 maemopad_2.3.tar.gz
-rw-rw-r-- 1 maemo maemo 675 Nov 9 13:37 maemopad_2.3_i386.changes
-rw-r--r-- 1 maemo maemo 42632 Nov 9 13:37 maemopad_2.3_i386.deb
```

6. Install the newly compiled maemopad_2.3_i386.deb file inside the Scratchbox environment:

```
[sbox-x86: ~/src/maemopad] > dpkg -i maemopad_2.3_i386.deb
... output from dpkg ...
```

7. Start the Xephyr server outside the Scratchbox:

```
Linux-PC$ Xephyr :2 -host-cursor -screen 800x480x16 -dpi 96 -ac -extension Composite &
```

8. Start the Application Framework from inside the Scratchbox X86 target:

```
[sbox-x86: ~/src/maemopad/maemopad-2.1] > export DISPLAY=:2
[sbox-x86: ~/src/maemopad/maemopad-2.1] > af-sb-init.sh start
... lots of output from various programs ...
```

You should now have the application framework up and running and the Xephyr window should contain the normal SDK UI.

9. Move to the source directory:

```
[sbox-x86: ~/src/maemopad/maemopad-2.1] > cd src/ui
```

10. In the ui directory, you should have the files:

```
[sbox-x86: ~/src/maemopad/maemopad-2.1/src/ui] ls -l
total 40
-rw-r--r-- 1 maemo maemo 12404 Sep 26 14:13 callbacks.c
-rw-r--r-- 1 maemo maemo 2250 Sep 26 14:11 callbacks.h
-rw-r--r-- 1 maemo maemo 15562 Jun 20 10:42 interface.c
-rw-r--r-- 1 maemo maemo 3282 Jun 20 10:42 interface.h
```

11. In this example, a debugging breakpoint will be set at the callback function `callback_help()` located in the `callbacks.c` file. This function is called when the user clicks the HELP button from the maemopad menu. Set the debugging breakpoint like this:

```
[sbox-x86: ~/src/maemopad/maemopad-2.1/src/ui] > run-standalone.sh gdb \
maemopad
(gdb) br callback_help
Breakpoint 1 at 0x804bf56: file ui/callbacks.c, line 296.
```

12. Start the maemopad application from the gdb.

```
(gdb) run
Starting program: /targets/x86/usr/bin/maemopad
[Thread debugging using libthread_db enabled]
[New Thread -1222154560 (LWP 22944)]
/home/maemo/.osso/current-gtk-key-theme:1: Unable to find include file:
"keybindings.rc"
maemopad[22944]: GLIB WARNING **: GLib-GObject - invalid cast from 'HildonProgram'
to 'GtkWidget'
maemopad[22944]: GLIB CRITICAL **: Gtk - gtk_widget_show: assertion
'GTK_IS_WIDGET (widget)' failed
Audio File Library: could not open file '/usr/share/sounds/ui-window_open.wav'
[error 3]
Audio File Library: could not open file '/usr/share/sounds/ui-window_close.wav'
[error 3]
hello-world background
```



N.B.

Thread-debugging: Now the native gdb program is used. If you need to debug threads in the application, you need to use the native gdb linked against the same thread library that your application is using. Remember that the Scratchbox gdb is not suitable for this purpose. To use the native gdb, native-gdb needs to be used, as mentioned earlier in this material.

13. Now you should be able to see the maemopad application inside the Xephyr window, and you should be able to use it normally. Next, click the upper menu and select the item HELP. In your gdb terminal window you should now see :

```
Breakpoint 1, callback_help (action=0x80bda40, data=0x806eca8) at ui/callbacks.c:296
296      {
```

14. The breakpoint that you set above is now reached, and execution of maemopad application is stopped. The gdb debugger waits for your command now. Try something simple, like using the list command to see where the execution of the application is going. You should get:

```
(gdb) list
291     }
292     }
293
294     /* help */
295     void callback_help( GtkAction * action, gpointer data )
296     {
297         osso_return_t retval;
298
299         /* connect pointer to our AppUIData struct */
300         AppUIData *mainview = NULL;
```

15. You can now debug the maemopad normally. Try and see, if you can execute the maemopad step by step so that you can get the help window on screen. Enter the s command like this:

```
(gdb) s
302         g_assert(mainview != NULL && mainview->data != NULL );
(gdb) s
304         retval = hildon_help_show(
(gdb) s
hildon_help_show (osso=0x80bda40, help_id=0x804cb08 "Example_MaemoPad_Content",
                  flags=1) at osso-helplib.c:636
636     osso-helplib.c: No such file or directory.
      in osso-helplib.c
(gdb)
```

16. You can also try the bt (backtrace) command to see what functions were called. You should see a list of functions that have been called, similar to the following:

```
(gdb) bt
#0 hildon_help_show (osso=0x80bda40, help_id=0x804cb08 "Example_MaemoPad_Content",
  flags=1) at osso-helplib.c:636
#1 0x0804bfb0 in callback_help (action=0x80bda40, data=0x1) at ui/callbacks.c:304
#2 0xb78bc688 in IA__g_cclosure_marshal_VOID__VOID (closure=0x8106ae0,
  return_value=0x0, n_param_values=1, param_values=0xffffffff,
  invocation_hint=0xbfac6798, marshal_data=0x0) at gmarshal.c:77
#3 0xb78a563b in IA__g_closure_invoke (closure=0x8106ae0, return_value=0x1,
  n_param_values=1, param_values=0x1, invocation_hint=0x1) at gclosure.c:490
#4 0xb78bb058 in signal_emit_unlocked_R (node=0x80f2bb8, detail=0,
  instance=0x80bda40, emission_return=0x0, instance_and_params=0xbfac68c0)
  at gsignal.c:2440
#5 0xb78bbfa8 in IA__g_signal_emit_valist (instance=0x80bda40, signal_id=1,
  detail=0, var_args=0xbfac6a4c "\001\020") at gsignal.c:2199
#6 0xb78bc296 in IA__g_signal_emit (instance=0x1, signal_id=1, detail=1)
  at gsignal.c:2243
#7 0xb7def78b in IA__gtk_widget_activate (widget=0x80bda40) at gtkwidget.c:4273
#8 0xb7cd9c62 in IA__gtk_menu_shell_activate_item (menu_shell=0x8103030,
  menu_item=0x80bda40, force_deactivate=1) at gtkmenushell.c:1230
#9 0xb7cda009 in gtk_menu_shell_button_release (widget=0x8103030, event=0x810c060)
  at gtkmenushell.c:748
#10 0xb7cd0f78 in gtk_menu_button_release (widget=0x8103030, event=0x810c060)
  at gtkmenu.c:2978
#11 0xb7cc67d0 in _gtk_marshal_BOOLEAN__BOXED (closure=0x8071028,
  return_value=0xbfac6c90, n_param_values=2, param_values=0xbfac6de0,
  invocation_hint=0xbfac6cb8, marshal_data=0xb7cd0e40) at gtkmarshalers.c:84
#12 0xb78a5979 in g_type_class_meta_marshal (closure=0x8071028, return_value=0x1,
  n_param_values=1, param_values=0xbfac6de0, invocation_hint=0x1, marshal_data=0x1)
  at gclosure.c:567
#13 0xb78a563b in IA__g_closure_invoke (closure=0x8071028, return_value=0x1,
  n_param_values=1, param_values=0x1, invocation_hint=0x1) at gclosure.c:490
#14 0xb78babcb in signal_emit_unlocked_R (node=0x80708c0, detail=0,
  instance=0x8103030, emission_return=0xbfac6d70, instance_and_params=0xbfac6de0)
  at gsignal.c:2478
#15 0xb78bbcf9 in IA__g_signal_emit_valist (instance=0x8103030, signal_id=0, detail=0,
  var_args=0xbfac6f70 "xo %G %00\020\b") at gsignal.c:2209
#16 0xb78bc296 in IA__g_signal_emit (instance=0x1, signal_id=1, detail=1)
  at gsignal.c:2243
#17 0xb7def914 in gtk_widget_event_internal (widget=0x8103030, event=0x810c060)
  at gtkwidget.c:4242
#18 0xb7cc4bd7 in IA__gtk_propagate_event (widget=0x8103030, event=0x810c060)
  at gtkmain.c:2348
#19 0xb7cc4f2f in IA__gtk_main_do_event (event=0x810c060) at gtkmain.c:1582
#20 0xb7b40551 in gdk_event_dispatch (source=0x1, callback=0, user_data=0x0)
  at gdkevents-x11.c:2318
#21 0xb78341fc in IA__g_main_context_dispatch (context=0x8066cc8) at gmain.c:2045
#22 0xb7835bb5 in g_main_context_iterate (context=0x8066cc8, block=1, dispatch=1,
  self=0x80690e0) at gmain.c:2677
#23 0xb7835eda in IA__g_main_loop_run (loop=0x8062200) at gmain.c:2881
#24 0xb7cc41b3 in IA__gtk_main () at gtkmain.c:1155
#25 0x0804a669 in main (argc=1, argv=0xbfac7224) at main.c:108
```

At the top of the backtrace list are functions that were called last, and at the bottom of the list are the applications `main()` function and then the `gtk_main()`, etc.

17. If you now continued debugging with `s` (step) command, you would end up looping the `gtk` main event loop. However, it is better to just give the `c` (continue) command:

```
(gdb) c
Continuing.
```

18. Now the `maemopad` application is running. If you select the `HELP` menu item again, the breakpoint is still set. So if you click `HELP` you would get again:


```
Breakpoint 1, callback_help (action=0x80bda40, data=0x806ec78) at ui/callbacks.c:296
296     {
(gdb)
```

19. You can clear a specific breakpoint using the clear command. Remove the breakpoint in `callback_help()` function:

```
(gdb) clear callback_help
Deleted breakpoint 1
(gdb) c
Continuing.
```

20. Because the breakpoint is now cleared, you can use the application normally under the Xephyr.

14.2.4 Debugging Hildon Desktop Plug-ins

This section explains how to debug Hildon Desktop plug-ins in maemo environment.

Desktop and Control Panel applications are both launched by the maemo-launcher daemon. Their plug-ins are all shared libraries (i.e. `.so` files). Writing these plug-ins is explained in maemo.org documentation [62].

Downloading and Compiling the Example Apps

For this example, you need to first download, compile and install the hello-world-app package.

1. Download the source package of hello-world-app:

```
[sbox-x86: ~/src] > apt-get source hello-world-app
Reading package lists... Done
Building dependency tree... Done
Need to get 235kB of source archives.
Get:1 http://repository.maemo.org diablo/free hello-world-app 2.1 (dsc) [363B]
Get:2 http://repository.maemo.org diablo/free hello-world-app 2.1 (tar) [235kB]
Fetched 235kB in 0s (264kB/s)
dpkg-source: warning: extracting unsigned source package (./hello-world-app_2.1.dsc)
dpkg-source: extracting hello-world-app in hello-world-app-2.1
dpkg-source: unpacking hello-world-app_2.1.tar.gz
```

2. Go to the sources directory ...

```
[sbox-x86: ~/src] > cd hello-world-app-2.1
[sbox-x86: ~/src/hello-world-app-2.1] >
```

3. ... and compile it like this:

```
[sbox-x86: ~/src/hello-world-app-2.1] > export DEB_BUILD_OPTIONS=debug,\
nostrip
[sbox-x86: ~/src/hello-world-app-2.1] > ./autogen.sh
... etc ...
[sbox-x86: ~/src/hello-world-app-2.1] > dpkg-buildpackage -rfakeroot
dpkg-buildpackage: source package is hello-world-app
dpkg-buildpackage: source version is 2.1
dpkg-buildpackage: source changed by Maemo Integration <integration@maemo.org>
dpkg-buildpackage: host architecture i386
dpkg-buildpackage: source version without epoch 2.1
dpkg-checkbuilddeps: Using Scratchbox tools to satisfy builddeps
fakeroot debian/rules clean
dh_testdir
dh_testroot
rm -f build-stamp configure-stamp
# Add here commands to clean up after the build process.
/sratchbox/tools/bin/make clean

... snip, output from compilation ...

dpkg-genchanges: including full source code in upload
dpkg-buildpackage: full upload; Debian-native package (full source is included)
[sbox-x86: ~/src/hello-world-app-2.1] >
```

4. You should now have:

```
[sbox-x86: ~/src/hello-world-app-2.1] > cd ..
[sbox-x86: ~/src] > ls -lt
total 312
-rw-rw-r-- 1 maemo maemo 738 Nov 14 08:27 hello-world-app_2.1_i386.changes
-rw-r--r-- 1 maemo maemo 58402 Nov 14 08:27 hello-world-app_2.1_i386.deb
drwxr-xr-x 5 maemo maemo 4096 Nov 14 08:27 hello-world-app-2.1
-rw-rw-r-- 1 maemo maemo 363 Nov 14 08:26 hello-world-app_2.1.dsc
-rw-rw-r-- 1 maemo maemo 234825 Nov 14 08:26 hello-world-app_2.1.tar.gz
```

5. Install the newly compiled debug version of the package hello-world-app_2.1_i386.deb:

```
[sbox-x86: ~/src] > fakeroot dpkg -i hello-world-app_2.1_i386.deb
(Reading database ... 15186 files and directories currently installed.)
Unpacking hello-world-app (from hello-world-app_2.1_i386.deb) ...
... snip ...
```

Now the Hildon desktop plug-ins example applications should be installed in the Scratchbox X86 target.

How to Debug Applications Started by Maemo-Launcher

Basically, these applications can be debugged by:

- attaching to an already running process
- starting the application using maemo-summoner

Attaching to Maemo-Launched Application with Gdb With maemo-launched applications, it is necessary to give maemo-launcher binary to gdb and attach to the already running process.

```
[sbox-x86: ~/ ] > export DISPLAY=:2
[sbox-x86: ~/ ] > af-sb-init.sh start
... snip ...
[sbox-x86: ~/ ] > pidof hildon-desktop | cut -d' ' -f1
22961
#
# this would take the first (largest) PID value from the returned list. The number 22961 is
just an example.
# smallest PID value is maemo-invoker which had requested maemo-launcher to start
hildon-desktop.
#
[sbox-x86: ~/ ] > native-gdb maemo-launcher
... snip ...
(gdb) attach 22961
Attaching to program: /targets/x86/usr/bin/maemo-launcher, process 22961
... snip...
(gdb)
```

Now it should be possible to debug the application normally with the gdb.

Starting Maemo-Launched Application with Maemo-Summoner The following will start the Control Panel under (native) gdb, and the newly installed Control Panel applet called `hello-world-app` will be debugged. The breakpoint will be set to function `hello_world_dialog_show()`. Pay attention to the question about the "pending shared library load".

```
[sbox-x86: ~/ ] > export DISPLAY=:2
[sbox-x86: ~/ ] > af-sb-init.sh start
... snip ...
[sbox-x86: ~/ ] > run-standalone.sh native-gdb maemo-summoner
... snip ...
(gdb) br hello_world_dialog_show
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (hello_world_dialog_show) pending.
(gdb) run /usr/bin/controlpanel.launch
Starting program: /targets/x86/usr/bin/maemo-summoner /usr/bin/controlpanel.launch
... snip ...
```

This should start the Control Panel in the Xephyr screen. When clicking the hello world plug-in in the Control Panel, the execution should stop at the given breakpoint. Try, for example, to get a backtrace:

```

Breakpoint 2, hello_world_dialog_show () at libhelloworld.c:68
68      GtkWidget *dialog = hello_world_dialog_new ();
(gdb) bt
#0  hello_world_dialog_show () at libhelloworld.c:68
#1  0xb70b05f7 in execute (osso=0x8075948, data=0x8084088, user_activated=1)
    at hello-world-applet.c:11
#2  0xb7ea4a71 in hcp_app_idle_launch (d=0x810ad08) at hcp-app.c:200
#3  0xb776c4d1 in g_idle_dispatch (source=0x810ac98, callback=0x806ac70,
    user_data=0x8075948) at gmain.c:3928
#4  0xb77691fc in IA__g_main_context_dispatch (context=0x806dbc0) at gmain.c:2045
#5  0xb776abb5 in g_main_context_iterate (context=0x806dbc0, block=1, dispatch=1,
    self=0x806f228) at gmain.c:2677
#6  0xb776aeda in IA__g_main_loop_run (loop=0x80deb40) at gmain.c:2881
#7  0xb7bec1b3 in IA__gtk_main () at gtkmain.c:1155
#8  0xb7ea1e37 in main (argc=1, argv=0xbfa7d184) at hcp-main.c:64
#9  0x08048a8b in main (argc=2, argv=0xbfc15bb4) at summoner.c:45
#10 0xb7e667e5 in __libc_start_main () from /lib/libc.so.6
#11 0x08048791 in _start ()
(gdb) list
63     }
64
65     void
66     hello_world_dialog_show ()
67     {
68         GtkWidget *dialog = hello_world_dialog_new ();
69         gtk_dialog_run (dialog);
70         gtk_widget_destroy (dialog);
71     }
72
(gdb) c
Continuing.

```

The backtrace tells what functions were called before the breakpoint was reached at `hello_world_dialog_show()`. Now the plug-in can be debugged normally with `gdb`.

Because the hello world plug-in was compiled with the `-g` option, the source code listing can be viewed with the `list` command.

To do this same for desktop, the `hildon-desktop` process needs to be killed first. The desktop can be killed with the command:

```

[sbox-x86: ~] > kill $(pidof hildon-desktop)
#
# after this you could start the hildon-desktop under gdb like this:
#
[sbox-x86: ~] > run-standalone.sh gdb maemo-summoner
... snip ...
(gdb) run /usr/bin/hildon-desktop.launch
... snip ...

```



N.B.

Doing this on the device would require first disabling the software lifeguard with the `flasher` tool. If this is not done, the device will reboot. The flag is:

```
--set-rd-flags=no-lifeguard-reset
```

14.2.5 Running Out of Memory During Debugging in Device

If running out of RAM memory during debugging in the device, there are a couple of options:

1. Add (more) swap to the device using the Memory applet from the Control Panel.

If there is enough memory, but gdb is still abruptly terminated, you can try setting it OOM-protected as root:

```
/home/user # echo -17 > /proc/[PID of your gdb]/oom_adj
```

By default, processes have OOM (i.e. Out-of-Memory) adjustment value of zero. The value -17 disables the kernel OOM killing for the given process. **N.B.** As a result of this, some other processes might be killed by the kernel.

2. Use gdbserver to debug.

Notes on Using Gdbserver

`gdbserver` is a debugging tool that can be started and run in the Internet Tablet device. Then a connection can be made to this running instance of `gdbserver` program from a Linux PC with a `gdb` program. `Gdbserver` uses a lot less memory than a full scale `gdb` program, thus making it possible to perform debugging in devices that have a limited RAM memory, such as PDAs.

The `gdbserver` does not care about symbols in the binary, but instead the Linux PC side `gdb` expects to have a local copy of the binary being debugged so that the binaries in the device can be stripped.



N.B.

Debugging core files from the device in Scratchbox with `gdbserver` has the same issues. In practice, it is easier to perform the debugging in the device itself. See the section above about prelinked binaries and libraries.

For further information about using `gdbserver`, `gdb` and `DDD`, see:

- scratchbox.org: Running the cross-compiled programs in a debugger
- scratchbox.org: Debugging in Scratchbox

14.2.6 Valgrind Debugger

Valgrind is a CPU simulator with different debugging and analyzing plug-ins. The Valgrind plug-ins are:

- `memcheck`

This plug-in tool is used to debug memory leaks and deallocation errors in applications. The following example will mainly focus on this.

- `massif`

This plug-in produces PostScript graph of process memory usage as a function of time. This also produces an ASCII or HTML report of allocation backtraces.

- **callgrind**

This can be used for profiling performance and getting call traces from programs. These can be further visualized with the Kcachegrind tool.

- **helgrind**

This helps in finding race conditions in threaded programs. This does not work in maemo Valgrind. It works only with Valgrind 2.2.



N.B.

In the maemo environment, you can use the Valgrind debugger only in the Scratchbox X86 target.

Valgrind has many options, but only the basic ones are covered here with examples. The link to the full Valgrind manual is given at the end of this section.

Installing Valgrind Tool

Installing Valgrind is simple. Log in on Scratchbox and run the following commands:

1. Get Valgrind from the repository:

```
[sbox-armel: ~] > sb-conf select x86
[sbox-x86: ~] > apt-get install valgrind
...
Setting up valgrind (3.3.0-1.ossol) ...
[sbox-x86: ~] >
```



N.B.

The maemo Valgrind version depends on the libc6-dbg. On the desktop Linux, some of the debug symbols are included in the libc6 library itself. If the debug symbols are missing from the libraries, Valgrind cannot match the error suppressions to the internal library functions. In the maemo libc6 case, it would show lots of errors for the dynamic linker.

If using a non-maemo version of Valgrind, the following environment variable needs to be set before valgrinding programs using Glib:

```
[sbox-x86: ~] > export \
G_SLICE="always-malloc"
```

Without this, Valgrind will report bogus leaks from Glib.

2. Get the two small example applications written in C language to demonstrate the basic usage of valgrind tool. You can download these from maemo.org: [valgrind_example.tar.gz](http://maemo.org/valgrind_example.tar.gz).

```
[sbox-x86: ~] > mkdir src
[sbox-x86: ~] > cd src
[sbox-x86: ~/src] >
```

After downloading, just copy the `valgrind_example.tar.gz` file to the `~/src/` directory and perform the following:

```
[sbox-x86: ~/src] > tar xvfz valgrind_example.tar.gz
valgrind_example/
valgrind_example/valgrind_example.c
valgrind_example/valgrind_example2.c

[sbox-x86: ~/src] > cd valgrind_example
[sbox-x86: ~/src/valgrind_example] >
```

3. Compile the two small example applications in the following way:

```
[sbox-x86: ~/src/valgrind_example] > gcc valgrind_example.c -o \
valgrind_example -g
[sbox-x86: ~/src/valgrind_example] > gcc valgrind_example2.c -o \
valgrind_example2 -g
```

Using Valgrind Memory Debugger Tool

After compiling the small example application, it can be run under valgrind with the command:

```
[sbox-x86: ~/src/valgrind_example] > valgrind --tool=memcheck --leak-check=yes \
--show-reachable=yes --num-callers=20 --track-fds=yes ./valgrind_example
```

Explanation of the parameters and their meanings:

- `--tool=memcheck`
Defines which tool Valgrind should use. In this example, the memory checker tool is used.
- `--leak-check=yes`
With this option, Valgrind checks the code for potential memory leaks.
- `--show-reachable=yes`
This option creates a stack trace of the creation of the reachable but unfreed memory when the program exits.
- `--num-callers=20`
With this option, the number of function call levels that Valgrind displays can be changed. The default value is 12. Giving higher number takes a bit more memory. It is advisable to tune this option, because for example the gtk callstack can consist of more than 100 items.
- `--track-fds=yes`
Track-fds means Track FileDescriptors. If the application is opening and closing files with `fopen()` or `open()`, this will report which files are not closed.

The output of the example application should look like:

```

==4493== Memcheck, a memory error detector.
==4493== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==4493== Using LibVEX rev 1732, a library for dynamic binary translation.
==4493== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==4493== Using valgrind-3.2.3-Debian, a dynamic binary instrumentation framework.
==4493== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==4493== For more details, rerun with: -v
==4493==
==4493== Conditional jump or move depends on uninitialised value(s)
==4493==   at 0x400A970: _dl_relocate_object (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4003B3: dl_main (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4012C29: _dl_sysdep_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x400193F: _dl_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4000846: (within /targets/x86/lib/ld-2.5.so)
==4493==
==4493== Conditional jump or move depends on uninitialised value(s)
==4493==   at 0x400A99C: _dl_relocate_object (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4003B3: dl_main (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4012C29: _dl_sysdep_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x400193F: _dl_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4000846: (within /targets/x86/lib/ld-2.5.so)
==4493==
==4493== Conditional jump or move depends on uninitialised value(s)
==4493==   at 0x400AFF3: _dl_relocate_object (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4003B3: dl_main (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4012C29: _dl_sysdep_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x400193F: _dl_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4000846: (within /targets/x86/lib/ld-2.5.so)
==4493==
==4493== Conditional jump or move depends on uninitialised value(s)
==4493==   at 0x400A825: _dl_relocate_object (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4003448: dl_main (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4012C29: _dl_sysdep_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x400193F: _dl_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4000846: (within /targets/x86/lib/ld-2.5.so)
==4493==
==4493== Conditional jump or move depends on uninitialised value(s)
==4493==   at 0x400A86C: _dl_relocate_object (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4003448: dl_main (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4012C29: _dl_sysdep_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x400193F: _dl_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4000846: (within /targets/x86/lib/ld-2.5.so)
==4493==
==4493== Conditional jump or move depends on uninitialised value(s)
==4493==   at 0x400A99C: _dl_relocate_object (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4003448: dl_main (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4012C29: _dl_sysdep_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x400193F: _dl_start (in /targets/x86/lib/ld-2.5.so)
==4493==   by 0x4000846: (within /targets/x86/lib/ld-2.5.so)
==4493==
==4493== FILE DESCRIPTORS: 3 open at exit.
==4493== Open file descriptor 2: /dev/pts/2
==4493==   <inherited from parent>
==4493==
==4493== Open file descriptor 1: /dev/pts/2
==4493==   <inherited from parent>
==4493==
==4493== Open file descriptor 0: /dev/pts/2
==4493==   <inherited from parent>
==4493==
==4493==
==4493== ERROR SUMMARY: 11 errors from 6 contexts (suppressed: 0 from 0)
==4493== malloc/free: in use at exit: 40 bytes in 2 blocks.
==4493== malloc/free: 3 allocs, 1 frees, 60 bytes allocated.
==4493== For counts of detected errors, rerun with: -v
==4493== searching for pointers to 2 not-freed blocks.
==4493== checked 56,760 bytes.
==4493==

```



```

==4493==
==4493== 10 bytes in 1 blocks are definitely lost in loss record 1 of 2
==4493==    at 0x4020626: malloc (vg_replace_malloc.c:149)
==4493==    by 0x804841B: main (valgrind_example.c:26)
==4493==
==4493==
==4493== 30 bytes in 1 blocks are definitely lost in loss record 2 of 2
==4493==    at 0x4020626: malloc (vg_replace_malloc.c:149)
==4493==    by 0x8048482: main (valgrind_example.c:48)
==4493==
==4493== LEAK SUMMARY:
==4493==    definitely lost: 40 bytes in 2 blocks.
==4493==    possibly lost: 0 bytes in 0 blocks.
==4493==    still reachable: 0 bytes in 0 blocks.
==4493==    suppressed: 0 bytes in 0 blocks.
[sbox-x86: ~/src/valgrind_example] >

```

The output of the Valgrind tells that there were 40 unallocated bytes for the application ("definitely lost"). This means that the example application is leaking memory, and cannot free it anymore. This means that the code should be studied closely.

Valgrind also tells the lines in the code where these allocations that are not freed are performed. In this example, the lines in question are 41 and 19.

Official Valgrind Manual

Valgrind.org has an official Valgrind manual available in the Internet, explaining the many options and debugging practices that Valgrind can support. For full coverage of Valgrind see [Valgrind User Manual](#) [100].

14.2.7 Other Debugging Tools

The SDK provides a number of other tools that are useful in debugging. See the linked documentation pages to learn more about them.

- *strace* - a system call tracer. Prints out all the system calls made by the traced process.
 - <http://maemo.org/development/tools/doc/strace>
- *ltrace* - a library call tracer. Prints out all library calls made by the traced process.
 - <http://maemo.org/development/tools/doc/ltrace>
- *sp-rich-core* - produce *rich cores* that will provide a snapshot of the system's state at the time of the crash in addition to the core dump file.
 - <http://maemo.org/development/tools/doc/sp-rich-core>
- *syslog* - a logging system that can collect logs in a centralized way.
 - <http://maemo.org/development/tools/doc/syslog>

Please also have a look at the tools main page [Maemo SDK tools](#) [73] for even more tools that could be useful for you.

14.3 Making a Debian Debug Package

Application developers should provide debug packages corresponding to their application packages. This section shows how to do that.

Debian debug packages contain *debug symbol* files that debuggers (like `gdb`) will automatically load when the application is debugged.

On ARM environment, the debugger cannot do backtracing or show correct function names without debugging symbols making it thus impossible to debug optimized binaries or libraries. All libraries in the device are optimized.

In X86 target, debugging can be done without debug symbols.

If the package maintainer provides no debug package, then the other developers in the community need to spend extra time and effort to recompile the application (or the library) with the debugging symbols. This extra (time-consuming) step can be simply eliminated by the package maintainer by providing a ready compiled debug package.

This section covers creating a Debian debug package from the maemo/Scratchbox point of view.

It is the package maintainers role and decision to create and provide a debug package for their application or library. This means that you as the package owner are responsible to modify the debian package configurations, so that the `dpkg-buildpackage` tool will produce the additional `dbg-package`.

If you are a maintainer of a library or binary package, it is strongly recommended to create a corresponding debug package. This is needed by anybody wanting to either debug or profile your software or something using it.

14.3.1 Creating DBG Packages

Steps to create a debian DBG package are:

1. **Clean up any previous dbg configurations from your package**

If your package already provides the debugging symbols the "hard, old way" then you should clean these configurations first.

If you use `dh_strip --dbg-package` option in `debian/rules` file, then it is not necessary to build or copy anything to the `-dbg` package build directory anymore. Check your packages `debian/rules` file and remove all lines that have statements to create `dbg` packages.

If you have any files named `debian/*-dbg.*` then just remove these files. These were required with the old way to use `dh_strip`.

The following steps are the only ones needed to create the debug packages with newer versions of `dh_strip`.

After you have cleaned up any previous `dbg` configurations from your debian files move to the next step.

2. **Define DBG package(s)**

For every package listed in your `debian/control` file that contains libraries or binaries, you need to add corresponding `<package>-dbg` entry. If you have `debian/control.in` then modify that instead.

Make the debug package(s) to depend from the corresponding binary/library package. Here is an example:

```
.
. other package definitions above
.

Package: libgtk2.0-0-dbg
Section: libdevel
Architecture: any
Depends: libgtk2.0-0 (= ${binary-Version})
Description: Debug symbols for the Gtk library
```



N.B.

If the package contains binaries instead of libraries, the Section should be devel.

N.B. The new -dbg package may have a different name from the old style debug package (for example libgtk2.0-0-dbg, not libgtk2.0-dbg). If there are earlier (old style) debug packages in the repositories the new debug package should replace/conflict with the old one.

3. Add option -g to CFLAGS

Make sure you have set option -g in CFLAGS in the debian/rules file, and that this option is effective and always enabled. Otherwise the debug package will not contain the required debug symbols.

For example, use a line like this for CFLAGS

```
CFLAGS = -Wall -g
```

4. Use dh_strip

You can either provide all debugging information for your package, or just a minimal debugging information required to debug something else using your library.

You want to select the latter option only if you want to reveal as little of your code as possible (for example) for contract reasons.

(a) Providing all debug information in your dbg package.

Debian/compat levels smaller than 3 should not be used (1 is default!). If your package sets debian/compat level **below** 5, give the following arguments to dh_strip in debian/rules file:

```
dh_strip --dbg-package=<package1> [--dbg-package=<package2> ...]
```

For example:

```
dh_strip --dbg-package=libgtk2.0-0
```

If compat level is 5 or higher, use syntax:

```
dh_strip --dbg-package=<package1>-dbg [--dbg-package=<package2>-dbg ...]
```

For example:

```
dh_strip --dbg-package=libgtk2.0-0-dbg
```

If you're using `cdb`s instead of `debhelper` (i.e. `dh_*` commands) in your `debian/rules` file, use this instead:

```
DEB_DH_STRIP_ARGS := <dh_strip arguments>
```

For example (for compat levels below 5):

```
DEB_DH_STRIP_ARGS := --dbg-package=<package1> ...
```

(b) Providing just minimal debug information

In case you do not want to reveal all information about your binary (e.g. for contract reasons), you can provide a debug symbol file just with the `.debug_frame` section (which is the minimal information needed by `gdb` to show working backtraces in ARM environment). In addition to information provided by the binary file, it will reveal only static function names and the number of function arguments.

To do this, replace the above `dh_strip` line in `debian/rules` file `binary-arch` target with:

```
chmod +x $(shell pwd)/debian/wrapper
export PATH=$(shell pwd)/debian/wrapper:$PATH; \
dh_strip --dbg-package=<package1> [--dbg-package=<package2> ...]
```

Notice that the wrapper script is set executable because the `dpkg-source` does not preserve the exec permissions.

Store the following wrapper script as `debian/wrapper/objcopy`:

```
#!/bin/sh
#
case " $* " in
  *" --only-keep-debug "*)
    exec /usr/bin/objcopy -R .debug_info -R .debug_aranges \
      -R .debug_pubnames -R .debug_abbrev -R .debug_line \
      -R .debug_str -R .debug_ranges -R .comment -R .note "$@"
    ;;
  esac
exec /usr/bin/objcopy "$@"
```

With this `dh_strip` will use `objcopy` through this wrapper (i.e. remove the other debug sections).

5. Verify the package(s)

Update the Debian changelog with `dch -i` and build the package with `dpkg-buildpackage -rfakeroot`. This will create new Debian source package (`dsc` + `diff`) and binary package(s) which you can install on the target for additional testing.

See also `dh_strip` (1) and `debhelper` (7) manual pages for details about the helper scripts.



N.B.

If the package has any function(s) that have the `noreturn` GCC attribute, you need to make sure that the object(s) containing those are compiled with `-fno-omit-frame-pointer` (or remove the `noreturn` attribute). This is needed for backtraces containing them to be debuggable when the binaries are optimized, the debug symbols are not enough for them. By default GCC omits frame pointer when code is optimized.

14.3.2 Using and Installing DBG Packages

The debug packages are easy to use, just `apt-get install` them and on target `gdb` will load the new style debug symbol files (installed to subdirectories under `/usr/lib/debug/`) automatically.

Inside Scratchbox, debuggers and profilers will search for the debug symbol files under scratchbox target directory under `/usr/lib/debug`, so a target directory needs to be linked under it.

```
mkdir /usr/lib/debug
cd /usr/lib/debug
mkdir targets
ln -s /usr/lib/debug \
  targets/$(sh -c '. /targets/links/scratchbox.config;echo $SBOX_TARGET_NAME')
```

The reason for this is that `realpath` on libraries and binaries inside Scratchbox returns `/scratchbox/targets/.../usr/lib/...` instead of just normal `/usr/lib/...`

Installing the `maemo-debug-scripts` package does this link automatically.

And for thread debugging to work, you need to install `libc6-dbg` package and `gdb` package, the Scratchbox provided host `gdb` doesn't work with threads (or ARM core dumps). To use the native `gdb` in Scratchbox, you need to start `gdb` the following way:

```
SBOX_REDIRECT_IGNORE=/usr/bin/gdb /usr/bin/gdb /usr/bin/my-binary
```

`maemo-debug-scripts` package provides native-`gdb` script for this.

For `gdb` to find old style debug symbol files (installed directly into `/usr/lib/debug/`) you need to use `LD_LIBRARY_PATH` or load them manually in `gdb`.

Files in these old style debug symbol files contain both the binary and debug symbol sections, so they are also larger than the new style debug symbols that `dh_strip` instructions above will create.

14.3.3 For Further Reading

- [Debian New Maintainers' Guide](#)
- [Debian Developer's Reference, Chapter 6 - Best Packaging Practices](#)

Chapter 15

Quality Considerations

The maemo developer needs to take care of many details, when developing a good quality application for an embedded device. Compared to the desktop Linux, a developer needs to be more careful with performance, resource usage, power consumption, security and usability issues. This document provides a generic test plan for developers to help them to create a good quality software.

Performance and Responsiveness

The user wants to be in control: the device must be responsive at all times, and give appropriate feedback. The UI responsiveness can be achieved e.g. in the three following ways: First, using process events by showing a banner and blocking the UI while operating. Second option would be to slice the operation into parts while updating e.g. a progress bar. Thirdly, threading can be achieved with the help of GLib, but this is increasingly hard to implement and debug. For time-consuming tasks, the second option is recommended. If the operation takes only a second or two, the first option can also be used.

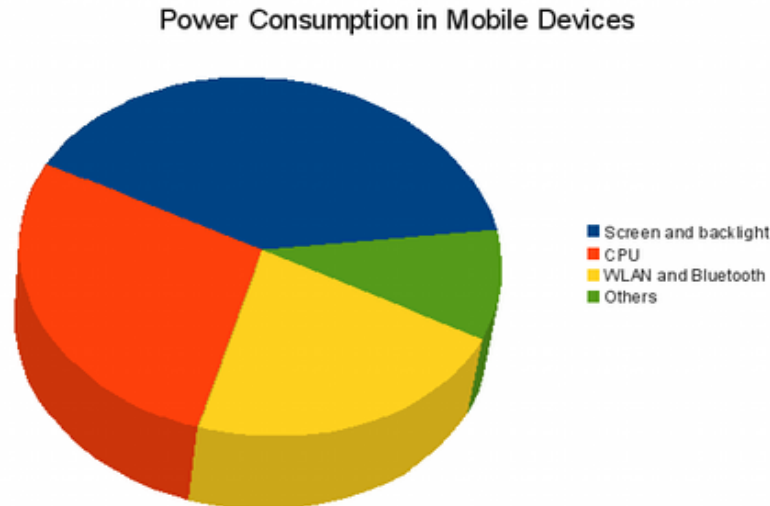
As other general guidelines, CPU over usage must naturally be avoided. This goes also for completely redundant operations, such as unnecessary screen updates. Also, when an application is not visible, it should be inactive to prevent it from slowing down the application user is currently interacting with.

Maemo development environment provides many tools to monitor responsiveness and performance of applications [73].

Resource Use

Resource use has to be taken into account with maemo applications very carefully, because if an application takes up too much resources, it cannot be used. The resources that are scarce are RAM, flash, network bandwidth, file descriptors and IO bandwidth. Also besides base memory usage, also memory leaks must be naturally avoided. There are tools such as Valgrind[99] to help detecting possible memory leaks.

Power Consumption



The device is not usable, if the user needs to recharge it all the time, or if it runs out of battery while idle. Therefore, no polling or busy looping should be used anywhere. Instead, an event-based approach, such as GTK's main loop is a better alternative. Also, too frequent timers or timeouts are to be avoided.

Secure Software Design

Quality and robustness together produce secure software. To achieve this it is important to follow secure software design guidelines and take into account certain issues specific to mobile Linux development.

15.1 Quality Awareness

This section is a generic test plan that defines a set of test cases for applications targeting the maemo platform. Good quality applications should pass all these test cases. If you see missing test cases in this section, please join "Quality Awareness" project in garage.maemo.org, and submit your cases to this material.

Test Cases:

1. *Packaging*

(a) Installation from application catalog

Precondition:

- Device without application and with default set of repositories.

Actions:

- Install application from install file (application catalog) by clicking install file in the browser.

Result:

- Application installs and runs correctly.

- All the needed dependences are downloaded automatically.
- There are no errors in the Application manager log.
- All icons created during installation are visible.

(b) Application upgrade

Precondition:

- Device with an older version of the application.

Actions:

- From the Application manager.
 - Update the repositories.
 - Upgrade application.

Result:

- The new application version works correctly in the device.
- There are no errors in the Application manager log.

(c) Application removal

Precondition:

- Device with the application installed.

Actions:

- Remove application by using Application manager.

Result:

- All the files installed by the package (except for modified configuration files) are removed.
- There are no errors in the Application manager log.
- Kill application process if it is running
- Remove tmp files

2. *Networking*

These cases assume that the application uses network.

(a) Offline mode

Precondition:

- Application is installed and works well in online mode.

Actions:

- Switch device to offline mode.
- Start application and try to use its networking functionality.

Result:

- Application should give indication to user that network is not available.

(b) Offline mode change

Precondition:

- Application is executing and works well in online mode.
- Working network available.

Actions:

- Start application.
- Switch device to offline mode.
- Try to use its networking functionality.

Result:

- Application should give indication to user that network is not available.
- If application UI blocks until network timeouts (in 30 seconds), it first indicates to user that it is working.

(c) Wlan disconnected

Precondition:

- Application is installed and works well in online mode.
- Working network available.

Actions:

- Disconnect network
- Start application and try to use networking functionality

Result:

- Device should connect automatically to network and application works correctly.

(d) WLAN and bluetooth connectivity works

Precondition:

- Application is installed and works well in online mode.
- Working network available and phone with data and bluetooth connectivity.

Actions:

- Run network test cases with WLAN connection
- Run network test cases with phone over bluetooth

Result:

- Application works with both connections

3. *Memory card*

Run these testcases on all memory cards.

(a) Memory card full

Precondition:

- Application installed and works.
- Fill the memory card.

Actions:

- Run the application

Result:

- Application should give correct error message

(b) No memory card

Precondition:

- Application installed and works.
- Remove all memory cards.

Actions:

- Run the application

Result:

- Application should give correct error message

(c) Memory card removed during application usage

Precondition:

- Application installed and works
- Memory card(s) inserted into slot

Actions:

- Start application
- Execute mmc functionality (e.g. access, save, delete, rename, backup)
- Remove mmc from slot

Result:

- Application should give correct error message

Observation:

- MMC functionality needs to take enough time (e.g. saving large sized file) so that memory card removing can be tested during its execution.

4. *Camera*

These cases are relevant if the application uses camera.

(a) Camera retracted

Precondition:

- Application installed and works
- Camera retracted

Actions:

- Start application
- Try to use camera functionality

Result:

- Application works and indicates that camera is not available

(b) Camera popped up during application usage

Precondition:

- Application installed and works
- Camera retracted

Actions:

- Start application
- Pop up camera
- Try to use camera functionality

Result:

- Camera functionality works correctly

(c) Simultaneous camera usage

Precondition:

- Application installed and works
- Other camera application running (e.g. Internet Call)
- Camera retracted

Actions:

- Start application
- Try to use camera functionality

Result:

- If application does not allow simultaneous camera usage, give the correct error message
- If application does allow simultaneous camera usage by either having its own switching mechanism or switching applications on the task navigator, camera functionality works correctly

5. *Platform Integration*

(a) Hildon UI style

Precondition:

- Application installed and works

Actions:

- Test the application so that all UI elements are visible

Result:

- Menus and toolbars are Hildonized
- Buttons sizes are usable with fingers
- Application icons are visible in menus & task bars
- HW Keys are mapped to relevant features
- Full screen mode works correctly

(b) Input method integration

Precondition:

- Application is installed to the device

Actions:

- Test all the text inputs and HW keys in the application UI

Result:

- Input method is invoked correctly when each of the text input widgets is tapped (by default normal input method for stylus tap and thumb keyboard for thumb tap) or if device has HW keyboard it is enabled and ready to use
- Enter key in the keyboard has logical behaviour
- HW keys have logical behaviour

(c) Backup

Precondition:

- Application is installed to the device

Actions:

- Change all application settings and quit the application
- Do a backup
- Flash the latest IT OS release
- Re-install the application
- Restore backup

Results:

- All changed application settings were restored for the selected categories

(d) Help

Result:

- Help item in application menu exists
- Help for the application in the device help exists

6. *Security*

(a) No unsecured TCP/IP ports

Precondition:

- Application is not installed
- Device has either ssh or terminal installed from which netstat can be run

Actions:

- Run "netstat -ta"
- Install application
- Run "netstat -ta"
- Run application
- Run "netstat -ta"
- Perform application use-cases and run "netstat -ta" after each use-case

Result:

- All identified ports should have documented security solution.

(b) Network source verification

Actions:

- Download code from or upload sensitive data to the network

Result:

- HTTPS or public key encryption used for the downloaded content

(c) Bluetooth security

Precondition:

- Application running in the device
- Ensure that your PC and tablet are not paired.

Actions:

- For each RFCOMM channel that your application opens, perform: "rfcomm connect <btaddr> <channel>". To find out which channel is used, publish it in SDP database and use "sdp-tool browse <btaddr>".

Result:

- Make sure that PIN is asked and user is asked to authorize connection in tablet before connection is established. Your security needs may vary, so authorization might not be necessary.

7. Performance

(a) Startup and close time

Actions:

- Boot the device
- Start the application (for the first time), and note when user can interact with the application
- Close application and note when application window disappears
- Start application again and note when the user can interact with the application

It is preferable to repeat these a few times and take the best times.

Result:

- Mandatory:

```
First startup < 6 secs
Second startup < 5 secs
Closing < 2 secs
```

- Desired:

```
First startup < 2 secs
Second startup < 1 sec
Closing < 0.5 secs
```

(b) UI responsiveness and feedback

Actions:

- Go through all application use cases

Result:

- All active UI elements react within 0.5 seconds to user actions (change state etc.)
- If an activity takes more than 3 seconds, there is progress indication

(c) Stress conditions

Precondition:

- "sp-stress" package installed

Actions:

- In (ssh) console run "cpuload 100"

Result:

- Application works without problems

8. *Battery usage*

Precondition:

- ssh package installed to the device and logged through ssh to the device

(a) No unnecessary CPU usage (no polling)

Precondition:

- Tested application is running, but it is not being used

Actions:

- Start "top" in (SSH) console
- Perform all application test cases

Result:

- After each application test case ends, all processes stop their CPU usage according to "top"

(b) No activity when device is idle or background

Precondition:

- "ltrace" package is installed to the device
- "No unnecessary CPU usage" test case is performed and application is left running

Actions:

- Wait until screen blanks
- ltrace the application process and if it's threaded, all of it's threads: "ltrace -S -p <PID1> -p <PID2> ..."
- Restart the application and repeat above steps

Result:

- Application has no activity when screen is blanked.

Note: threaded applications wake up at 8 sec intervals if they use LinuxThreads thread library instead of NPTL.

9. *Reliability*

(a) Memory leaks

Precondition:

- Application compiles and works in x86 Scratchbox
- Read Maemo Debugging Guide
- Install Valgrind to Scratchbox

Actions:

- Go through application use cases under Valgrind/Memcheck

Result:

- Valgrind does not report (definitive) memory leaks for the application

10. *Error handling*

(a) Full Flash memory

Actions:

- Fill internal device flash as "user" user, e.g. by copying a file in File manager
- Start the tested application
- Go through application use cases
- Exit application

Result:

- Application starts
- Application use cases that save data to Flash, give correct error message when the data save fails (this can mean that application shows an error at startup and exits after user OKs it)
- Application exits gracefully

Note: Part of the Flash is reserved for root, so applications cannot rely on just checking whether Flash has free space.

(b) Resource usage

Precondition:

- application running

Action:

- Start browser and browse to slashdot.org

Result:

- The application should not affect to browsing experience

11. *Scalability*

(a) Different amounts of input data

This concerns applications which can load external data, either from the filesystem or network.

Precondition:

- Start "top" in (SSH) console

Actions:

- Load data (files) with sizes of 1KB, 10KB, 100KB, 1MB, 10MB

With compressed formats, the data size can refer to uncompressed data size in memory.

Result:

- Loading of all files succeeds, at least when no other applications are running or swap is used
- Mandatory: The time application spends parsing/loading the data does not increase exponentially nor does the application memory usage increase exponentially

- Desired: With large data files, time and memory usage is less than linear in relation to the data size (application doesn't read all of the data, scales images on the run etc)
- (b) Different screen sizes
- Actions:
- Start application
 - Switch application window between full screen and normal mode
- Result:
- Both modes work

12. *Documentation*

Results:

- Release note
- Change log
- Copyrights in place
- Licensing in place
- Possibility to generate API docs from source if application provides APIs

13. *Policy conformance*

Results:

- Copyright & License defined
- Source code available if license require it
- Maemo trademark followed

15.2 Secure Software Design

15.2.1 Overview

This document tries to offer tools for secure software development, both with regard to processes and tools that can be used, and with special focus on mobile devices and the maemo platform. The target is to provide food for thought for anyone who is making a transition from non-mobile world to mobile world or closed-source to open-source, and would like to learn more about potential failure and vulnerability modes of mobile devices in general.

This document does not intend to be a definitive list of everything that could go wrong when developing software, nor does it try to describe how various attacks are specifically conducted. This sort of information is readily available both on-line and on flattened dead trees; please refer to the literature section for details.

The reader is expected to have knowledge about software development in general, including processes, implementation and testing.

Any tools or literature referred to in this document are listed only because it is believed that they represent potentially helpful resources. This document does not constitute an offer to sell or a solicitation to buy these goods or services.

15.2.2 Elements of Secure Software Development

Makings of secure software

Security in software can be defined in various ways, some addressing the functionality of the system and some building on the system's intended use. One of the best-suited definitions for general software development is a definition that is quality-based: software that fulfills its purpose can be considered as being of high quality. In a similar vein, software that fulfills its purpose under attack or in a hostile environment can be considered as being secure.

This definition already hints towards the main building blocks of secure software:

- The system designers need to understand the hostile environment in order to be able to defend against it; and
- The system needs to be tested properly in an environment that is a realistic hostile environment.

A quality-based definition also highlights synergies between quality and security as cross-cutting concerns. In the same way as a dedicated quality engineer cannot magically convert a product into a good quality one, a security engineer is also relying on everyone else's effort to actually make the product secure. And, as with quality, the metrics of security are not always straightforward and can often be measured only after the system has been taken into use.

Suggested Secure Software Development Lifecycle

Various different models for secure software development exist. One of the best documented is Microsoft's SDL (Secure Development Lifecycle), which has been documented in their book (see the literature section).

It is somewhat of a matter of taste, how many different steps should be included in a secure development lifecycle. However, when comparing the secure software development in many companies, three major steps can be seen as being common over the whole industry:

- Threat analysis
- Security-aware implementation
- Security testing

Of these, threat analysis and security testing are discussed below from the point of view of a mobile device. The second one, security-aware implementation, is obviously important as well, but learning about it is best left for existing literature.

On GNU/Linux (and other UNIX) platforms, perhaps the best source of ideas for security-aware implementation is Secure Programming on Linux and Unix HOWTO (again, see the literature section). There exists also a myriad of other books on this subject, and a proposed reading list can be found in the literature section of this document.

15.2.3 Threat Analysis

Threat Analysis as Basis for Secure Software

Attacking and defending an information system is an exercise in asymmetric warfare. The attacker can usually almost freely choose where to attack, whereas the defender must defend on all fronts.

In addition, malicious thinking does not always come naturally to everyone. Law-abiding people, who have no security background do not instinctively think like a wrongdoer.

As a consequence of both of these traits, a system that has been designed without specific security awareness may be open to attack from various angles. In order to counter this, one needs to conduct threat analysis, which is the basis for any security engineering work.

Threat analysis has two main objects. First, it tries to address the asymmetrical nature of security by identifying as many defensible assets and system aspects as possible. Second, the way in which threat analysis is conducted is designed to help the participants to think like an attacker.

How to Conduct Threat Analysis

There are many ways to do threat analysis. One of the more formal ones is to chart all data flows through an application by using an architectural diagram that displays all interfaces and components. This is best done when the system architecture is clear, and it is based on the observation that generally, most current attacks are due to malicious (malformed or somehow illegal) inputs, and software that fails when subjected to such input. By observing the movement of data through the various components of the application using an architectural diagram, sensitive interfaces can be identified and the main points of the system that need to be hardened can be picked out.

This type of data flow analysis does not, however, typically address cases where the system is attacked by other means than malicious inputs, for example by subjecting it to a different environment than where it was designed to operate. Hence, it is worthwhile also to use freeform methods, such as brainstorming in a group, to cover all possible ways in which an attacker might want to attack the system.

Brainstorming is also a good method to induce malicious thinking. Several people asking "but what if...", supporting each other and inventing new attack vectors can typically find much more creative ways of attacking a system than just someone who has no previous security experience. A hardened security practitioner may be able to pull off a successful security analysis alone, but for laymen (in terms of security), support from a group has been shown to enhance the quality of threat analysis.

Open Source Threat Landscape

General belief holds that open source practitioners are typically rather well security-aware. However, for someone coming from proprietary or closed-source software development, it might be helpful to discuss the specific traits that open source development brings on the security area. It should be noted

that these traits are not security threats by themselves, but they in many cases do have an effect on how security issues can be treated.

First, some open source licences require that any changes that have been made to source code need to be released under a similar licence. When considering security fixes, this means that fixes will get public, and usually it is very straightforward to compare a fixed version against an old version to determine what the security issue was. When utilising open source code, security issues are then automatically exposed in source code releases.

Second, many (but definitely not all!) open source projects are hobbyist projects, and in those cases the main author or maintainer of the project might not get paid for their work. This, in turn, may effect the turnaround time for security fixes, because some people may need to prioritize their other life (work and other personal interests) over the open source project. If your product is depending on a specific service level for security fixes, and the open source project is not one of the large established ones, you should probably be ready to resource this type of work yourself. This is also implied by typical open source licences, which usually release the code 'as is', without a warranty of any kind.

Third, open source may make it easier to develop a working exploit for a found security vulnerability. An exploit for a closed source code may be more of a hit-and-miss, but in open source the context of the vulnerability is clearly seen. This does not make open source any more insecure, but it may have an effect on the time that it takes to productize a security exploit.

On the other hand, open source has clear benefits for security: those open source projects that are widely used also tend to be kept looked at by many security researchers, so problems may get found quicker than in closed source. In addition, open source enables you to actually fix the problems yourself, if needed, so you can get on with the development of your system without having to wait for fixes to be included in closed source components.

Security in Mobile Devices

Mobile devices differ by nature from desktops or servers. Obviously the distinction between mobile and non-mobile systems are blurring all the time, and some of the considerations mentioned here will probably sound archaic in 2015. However, there are certain factors that will still differentiate between mobile and non-mobile devices for years to come, and therefore should be taken into account when thinking about security threats.

Three main categories of mobility-related threats are discussed shortly below:

1. Threats related to mobile data bearers
2. Threats related to the constrained system
3. Threats related to the personal nature of the device

First, the communication channel of a mobile device is typically much less wide than for a server or a desktop system. Even though 3G networks' HSPA, WiMAX and Wi-Fi networks are promising transmission speeds in several megabits per second, there may still be no guarantee that the channels will always be this fast. In rural areas with a 2G, non-EDGE-enabled GSM network,

the data transfer speed may be closer to a modem of early 1990s. From the perspective of an attacker, narrowband channels are easier to congest using a denial of service attack.

Another aspect which is related to the communication channels is the billing model. Most mobile devices insulate the running applications from the specifics of the data bearer. Data transfer might happen over Wi-Fi or cellular, and the bearer might even change during the operation of the application (for example, subsequent HTTP requests might be transferred over different bearers). Therefore, the applications cannot really make a guess whether the bearer they are using is flatrate or whether it is metered per-bit or per-second. Some cellular operators even meter packet data on a per-second basis, and some typically flatrate bearers such as Wi-Fi might have a per-bit charging model, which is still the case in certain hotspots such as hotels. If the application could somehow be coerced into producing a lot of traffic (either bitwise or timewise - think about background polling), this may result in a dramatic effect on the user's bills. Further, as an interesting twist, many operators operate premium-rate numbers for text messages and phone calls, and messages or calls to these numbers might cause unwanted costs - much in the same way as the 'modem hijacking' attacks were done when modem connections were the norm for home computers.

As the bearer might change dynamically, this also poses challenges to any network services that are being used. Cellular connections and text messaging are often thought as being secure, as they utilize closed networks that are typically well secured by operators. However, the bearer might suddenly change into an open Wi-Fi network, where all packets are open to casual inspection and any open ports that are listening to incoming data can be freely accessed. The connection security would really need to be designed with the least secure network in mind.

The second major difference between mobile and non-mobile devices is the power source. Battery technology has been historically very slow to develop, when compared to other features of portable devices. Power-saving and performance tuning of mobile devices is a very complex area, and a single ill-behaving program might cause excess power consumption. Even though this might not immediately sound like a security issue, it might turn out to be one if a flat battery causes a denial of service for the whole device.

This category of mobility-related challenges also applies to other resources than power. Mobile device CPUs usually run on smaller clock speeds, and they have less memory, less mass storage and a smaller screen than desktop or server machines. An attack against any of these more constrained resources might cause a denial of service.

A third category is concerned with the fact that mobile devices are often very personal. They share the same geographical location with their owner or user, and therefore might introduce a way for malicious software to track the user's whereabouts. Also, users typically store a lot of private data on the devices, such as address books, calendar markings, personal notes, photos, and of course, e-mail and text messages. Leaking this data may constitute a privacy problem, and deletion of the data may cause significant annoyance, as can be witnessed by anyone that has lost their mobile phone and has not made a back-up of the address book.

The personal nature of mobile devices is also highlighted in the way that unique but static identifiers are used. A typical mobile device has one or more

MAC addresses (for wireless LAN, Bluetooth, WiMAX), an IMEI (GSM/3G device serial number) and an IMSI (GSM/3G subscriber identity). As these identifiers are globally unique and static, combining them with the user may tag the user by a traceable identity, giving rise to various privacy threat scenarios. Software should therefore pay attention to privacy related issues and restrict the use of such identifiers.

15.2.4 Security Testing

Robustness Testing

So, a project did their security threat analysis and paid attention to secure implementation. The code is now ready - what happens next?

Typically, the implementation will be tested. Part of the testing is 'positive' testing: the implementation is subjected to use cases that are supposed to succeed. Some of the testing may be 'negative', in which the implementation is expected to fail cleanly.

Perhaps the most important security testing that should be performed is a specific type of negative testing. It is often called 'robustness testing', as it tries to make the implementation to break down, i.e. behave in a way that is not very robust. Looking back at the definition of security, this is a necessary part of a quality system being also secure: in a malicious or hostile environment (which is approximated by robustness testing), the system must still remain usable (meaning that it has to be robust).

Robustness testing usually concentrates on inputs to the system. Any system that interacts with its environment has some input interfaces. Typically they are protocol parsers, file parsers, user interface input elements, application programming interfaces, remote procedure call interfaces, and so on. The interface does not need to be intended to be used by a user - even a Domain Name System lookup, which returns something from a name server, is an interface, even though it is usually machine-to-machine. Therefore, any incoming data should be suspect and optimally should be robustness tested.

Obviously, testing everything may not always be an option due to lack of resources. Priority should be then given to those inputs that are unauthenticated (meaning that there is no way of telling from whom the input came) and those that get inputs from outside the system. As an example, robustness properties of a system API are probably tested with a smaller priority than a file parser that the user activates to read an input file, and again that file parser is probably lesser priority than a network daemon that listens on a network port and processes requests coming from an unknown remote computer. The security threat analysis should have already listed these interfaces and specified to which extent these would need to be robustness tested.

Robustness testing is typically built on inputs that are almost, but not quite, well-formed. Different robustness testing methods include fuzzing, in which a valid input is taken and (semi)randomly broken ("fuzzed"). Fuzzing can either happen on bit-level, especially when the input is binary data, or on a higher level that takes the input structure into account. As an example, an XML fuzzer might add non-specified XML elements or truncate existing XML elements.

Another, perhaps a more systematic way of creating robustness test cases is to synthesise broken test cases from the specification of the input. As an

example, if a specification says that a text field can be 16 bytes in length, test cases might try lengths of zero and 17 bytes. This sort of synthesis is typically more efficient than semirandom fuzzing, but comes initially with a much higher amount of work, and requires good knowledge of how parsers typically fail.

Bugs that are found with robustness testing often cause crashes or jams, or at least downgrade the usability. The target should be that any illegally formed content would cause minimum disturbance to normal use; in many cases, a silent discard of malformed data is the best approach to take, and sometimes the user might be notified with an understandable error message.

It may be useful to note at this stage that conformance test suites that exist for various protocols are not necessarily good security test suites. Conformance is by definition usually positive testing, and even though they may test failure modes, it is not guaranteed that these tools actually are malicious enough.

Pointers to robustness testing tools can be found in the literature section.

Static Analysis

Static analysis is a method where the source code itself is analysed to find potential security issues. Analysis tools differ in the ways they work, but typically they detect the use of dangerous functions and system calls, or track the flow of data through the components.

Static analysis tools may produce large amounts of false positives, and usually need to be taught to ignore some of the issues that they spot. Some more elementary tools may not be able to tell the difference between safe and unsafe ways of calling a potentially hazardous function, for example, whereas some sophisticated tools may actually track all the possible inputs to the function and evaluate security based on the input.

The best uses for static analysis are as a help for code review. Problems that are pinpointed by static analysis tools can be treated as a matter of course and code analysis effort can be then used for other types of problems. Large numbers of static analysis findings can also be used as an indicator that a specific part of code should be manually reviewed.

If the number of false positives has been driven successfully down, a static analysis system can also be utilised as a gatekeeper, when checking in new or changed code to version control system, or as one of the phases in unit testing. The first step in static analysis should probably be to turn on all compiler warnings and address all of those.

Some pointers to static analysis tools can be found in the tools and resources section.

15.2.5 Conflicts of Interest?

Usability and Security

Sometimes, usability is seen to be at odds with security, as security features may be perceived to be something that hinders the free use of the system. Even if this is true to some extent - for example, the concept of certificates is not a very easy one for the user to grasp - one could argue that this is actually more due to poor usability of security-related features than security itself. As an example, certificates could in most cases be totally invisible to the user, and new ideas

(a good example is Petname, see [84]) could be fielded that would change the whole security perception.

It is useful to understand that mobile devices are used by people who do not use a "normal" computer in everyday life. This trend will probably be even more significant in the future. Already now there exists a huge population, whose primary means of electronic communication is a mobile device, not a desktop, and they do not necessarily have the same model of (for example) web browser security in their mind as those who are accustomed to use a desktop browser.

When mobile devices are concerned, usable security engineering should be given a lot of thought. Most of the aspects are related to the restricted user interface. As an example, a typical secure password contains both uppercase and lowercase letters, and perhaps special characters. Typing such passwords on a typical mobile device keypad (such as the ITU-T numeric keypad) is time-consuming and error-prone. Therefore, a system that would do away with the need to enter passwords would be probably welcomed by users.

Another point is that all kinds of security indicators such as coloured URL bars, lock icons, and such, take up valuable screen estate.

Interoperability and Security

Interoperability and security may also be sometimes seen as being in conflict. To achieve maximum interoperability, a system "should be liberal in what they accept, and conservative in what they send". However, with liberalism comes a lot of responsibility: even though a system is liberal in accepting inputs, it should really know what it is getting before actually acting on the data.

15.2.6 Tools and Resources

Literature

For implementors on Linux and other Unix platforms, David A. Wheeler's Secure Programming on Linux and Unix HOWTO [103] is a good read. This document lists a lot of pitfalls on a Unix system. However, it does not really get in-depth with how the security is attained in a software project, but as an awareness-raiser for implementors, this is very useful (and free).

If you are managing a software development team, and are looking for insight as to how security could be brought into your software development methodology, SDL: The Security Development Lifecycle by Michael Howard and Steve Lipner (Microsoft Press, 2006) is a revealing account of how the processes were changed at Microsoft. The processes sound pretty waterfallish, and agile methods are given a somewhat cursory treatment, but this is still an interesting insight to how secure development can be pulled off in a large company. Microsoft's SDL people also blog about these issues at <http://blogs.msdn.com/sdl/>, where you can find pointers to SDL documentation updates.

Another book on how to implement security in software development is Secure Coding: Principles and Practices by Mark G. Graff and Kenneth R. van Wyk (O'Reilly, 2003). This is a very concise book, but somewhat misleadingly named - it does not address the actual implementation issues that much, but

instead lists tons of good principles in a checklist-style format for both architectural design and general software development.

Security and usability is not covered very well in books. However, there are two online resources that should be mentioned: the first is Ka-Ping Yee's User Interaction Design for Secure Systems [54], and the second is Peter Gutmann's manuscript Security Usability Fundamentals [85].

There are a few books on security testing, but none of them are above the rest. If you have a specific application in mind, browsing the contents of the security testing related books is recommended: some of them, for example, only talk about web application testing and therefore do not really lend themselves for learning about security testing generally.

Tools

Please note that these pointers are provided for informational purposes only. This document does not constitute an offer to sell or a solicitation to buy these goods or services.

There are a lot of fuzzers and it is not very difficult to implement one yourself, if your project calls for knowledge of the underlying data formats. A lot of tools can be found by searching the web or from Wikipedia [21]. Some of the freely available fuzzing tools are:

Untidy an XML fuzzer with a Python API [98].

WebScarab a fuzzer (and a lot more) for web applications [102].

zzuf a general-purpose binary fuzzer [106].

Bunny the Fuzzer a general-purpose, adaptive fuzzer that uses code instrumentation [4].

An example of a more systematic way for robustness testing are commercial tools developed by Codenomicon [7]. In their toolkits, problems are not randomly introduced but instead the test cases are built on the protocol or file format specifications.

Static analysis tools can also be found in both commercial and noncommercial domains. Commercial ones include offerings from Coverity [8], Fortify [19] and Klocwork [53], to name a few. Free ones include:

Flawfinder C and C++ source code scanner [18].

RATS C, C++, Perl, PHP and Python scanner [89].

For more options, please see Wikipedia [104].

15.2.7 Maemo Security Policies

Reporting Vulnerabilities

If you discover or become aware of a security issue in maemo software (consisting of the maemo platform software and the upstream projects whose software is used within the maemo platform software), please report it by email to security@maemo.org. Please encrypt your message with GnuPG using key ID

0x83AAAB3B (available from PGP key servers). The report will be analysed, and appropriate actions initiated.

If you discover a security issue in an upstream project whose code is used in maemo, or a 3rd party open source software running on the maemo platform, as your first priority, report the problem to the upstream project or their security team and only after that send a copy to maemo security as per above. If you do not know where to report the issue, we suggest you report it to the Open Source CERT at <http://www.ocert.org/>. Please see below for helpful tips on what information would be useful.

Please note that security@maemo.org only handles issues relating to the maemo platform. This email address does not handle security issues related to web sites (including the maemo.org website), 3rd party software running on the maemo platform, or issues specific to Nokia products. In any security issues related to these, please contact the appropriate party.

When reporting a security issue, it would be helpful to know if the security issue has been publicized somewhere, so if you can, please provide a pointer to that (web address, CVE identifier, etc.)

It is also important to understand, where and how the issue manifests itself. Information that is useful is the name of the affected component and package (and the full version number), configuration and environment where the issue was discovered (proof-of-concept code if available).

If you are able to provide more information and details that would be helpful in validating the issue, we would appreciate your contact information.

Any security-related bugs in maemo bugzilla should be tagged with keyword "security".

15.3 Maemo Coding Style and Programming Guidelines

The purpose of this section is to set the guidelines and recommendations for maemo developers, and anyone who writes new code for the maemo platform. This section assumes that programming is done in C language, but most of the guidelines are not language-specific. It is also required that developers possess a certain level of knowledge of Linux and GTK+ programming.

Why are programming guidelines needed? Some reasons for this are:

- Readability of code

Many people write program code for maemo and have different coding styles, because C allows it; therefore C programs can look very different, and be difficult to read for someone who is used to a particular style.

- Correctness and robustness of code

Some people are more disciplined programmers and others are more relaxed; this can lead to "sloppy programming" - that is, incomplete error handling, no sanity checks, hacking etc.

- Maintainability of code

Many programmers tend to write "ingenious" code which is almost impossible for others to read, and thus almost impossible to maintain and debug. So, things should be kept simple.

"Debugging is twice as hard as writing code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it." (Brian W. Kernighan)

- Platform requirements

Some guidelines must be followed by maemo developers, because they have direct impact on the platform (for example, power usage).

In addition to this section, one should read also the GNOME Programming Guidelines [88] and pay attention to Glib/Gnome-specific issues. There is also a coding style document by Linus Torvalds for the Linux kernel [56], containing hints on how to configure one's editor with the specific indentation style used in the maemo code.

Even though proper coding style alone is not a guarantee of good software, it at least guarantees a good level of legibility and maintainability.

15.3.1 Maemo Coding Practices

Use the following maemo coding practices:

- The section below contains generic guidelines about coding style, variable usage and naming, error checking and many generic guidelines that good developers may already be familiar with. The most important thing about code style is consistency; one should pick a specific style and always follow it.
- Maemo developers must be careful with memory usage and memory leaks: one should always consider employing and initializing local variables. Code can be protected from memory leaks by freeing memory as soon as possible, or using automatic variables stored in a stack (not in a heap), guaranteeing automatic allocation and de-allocation.
- Memory pool should not be fragmented with `malloc(3)` and small or multiple instances of `free(3)`. Reusing memory or freeing it in large blocks is recommended. A more effective method than manually searching for memory leaks is to use an automatic tool. For more information, see section below.
- If planning to use threads, one should remember that thread based applications are more difficult to trace and debug. Threads should not be used just for their easiness or fashionability. They should be used only if they bring some clear benefits like better UI responsiveness to the application. Notice also that asynchronous calls to the functions can in some cases be used as replacements to the threads.
- System message notification and application-to-application communication make use of D-BUS, a lightweight message passing protocol. D-BUS must not be used to pass long or frequent messages between applications. D-BUS delivery is not guaranteed, but the application must check for it. For more information on D-BUS guidelines, see section below.

- When developing a new application in the maemo platform, it is the programmer's responsibility to catch hardware events, such as button press and touchpad actions, in the application. Touchpad events are similar to mouse events in a desktop environment. Hardware buttons form a special case, since they have special semantics, such as zooming, canceling operations and various other actions expected by maemo users. For more information on hardware events, see section below.
- Embedded devices offer many challenges, primarily for desktop developers. While a desktop user accepts a certain level of resource wasting, users of maemo devices do not accept such waste. One of the main objectives of maemo developers is to keep the resource usage at minimum. For a full review of this topic, see section below.

15.3.2 Basics of Good Programming

Most of the following conventions are useful for any programmer, and experienced programmers should already know them. However, there are also some that are related to programming for embedded devices, such as avoiding the use of dynamic memory, prioritizing the reliability of code and, above all, saving resources.

Coding Style

It is more preferable to use simple than "clever" solutions, and one should always be wary of lazy coding and especially so-called hacks. Clever code is harder to read, thus harder to debug and more difficult to prove correct. Careful coding means taking care of all possible error situations and taking the time to think while writing the code.

Source code should be commented. The comments should be written in English. In general, comments describe *what* is being done, not *how*. Comments should be made based primarily on what functions do, and they should always be kept synchronized with the code. The formats (such as strings, files and input) should be described to make it easier to understand the code.

It is not advisable to use more than three or four indentation levels. When using tab characters for indentation, one should make sure that the code does not overflow from a terminal screen (width 80 columns) even if the tab size is 8. The functions should be kept short, no longer than 50-100 lines of code, unless absolutely necessary. The line length should be kept at between 1-75 characters.

Empty lines should be used to group code lines that logically belong together. One or two empty lines can be used to separate functions from each other. The command "indent -kr <inputfile> -o <outputfile>" can carry out some of these functions for the code.

Variable Use

Local variables should always be favored over global variables, i.e. the visibility of a variable should be kept at minimum. Global variables should not be used to pass values between functions: this can be performed with arguments. Global

variables require extra care when programming with threads. Thus, these should be avoided.

All variables should be initialized to a known illegal value (such as 0xDEADBEEF), if they cannot be given a meaningful value from the beginning. There are at least two reasons for this:

- Known initial values contribute to predictable behavior in the program, i.e. they reduce random behavior in error situations caused by uninitialized memory. This helps in troubleshooting.
- Illegal initial values increase the probability of early detection of an erroneous state of the program, as an illegal value (such as a null pointer) is more easily detected than a random one.

Error Checking and Handling

The return codes of all system calls and functions that can potentially fail should be examined. Usually system calls only fail in rare situations, but these must also be dealt with. At least, it is good to have a log entry stating the reason for a program exit when the program cannot continue, instead of calling a segmentation fault when memory allocation has failed.

One should be prepared for network disconnections. If a program uses network connections, it must be written in a way that allows it to recover from lost connections and broken sockets. The program must operate correctly even when a disconnection occurs.

Variable and Function Naming

Descriptive, lowercase names should be used for variables and functions. Underscores (`_`) can be used to indicate word boundaries. Non-static and unnecessary global names should be avoided. If it is absolutely necessary to employ a global variable, type or function, one should use a prefix specific to the library or module where the name is, e.g. `gtk_widget_`.

Each variable should be employed for one purpose only. The optimization should be left to the compiler. Short names for local variables are fine when their use is simple, e.g. `i`, `j` and `k` for index variables. In functions with pointer arguments, `const` should be used when possible, in order to indicate what causes changes (or does not cause any changes) in the pointed memory.

Constants and Header Files

So-called magic values should not be used, meaning literal constants (such as numbers) should not be used as coefficients for buffer sizes. In a header file, `#define` can be used to name them. That name can then be used in the code instead of the value. That way, it is easier to adjust constants later and make sure that all occurrences are changed when the constant is modified. However, names like `#define ONE 1` should be avoided. If there are a lot of constants, putting them into a separate header file should be considered.

Header files should be written to be protected against multi-inclusion. Also, it should be made sure that they work when included from C++ code; this

is achieved by using `#ifdef __cplusplus extern "C" { ... }`, or by using `G_BEGIN_DECLS ... G_END_DECLS` in Glib/GTK+ code.

Memory Use

Static memory (variables) should be preferred to dynamic memory, when the needed memory size is not large and/or is used regularly in the program. Static memory does not leak, a pointer to static memory is less likely to cause a segmentation fault, and using static memory keeps the memory usage more constant.

Unneeded dynamic memory should always be freed in order to save resources. N.B. Shared memory is not released automatically on program exit.

Memory heap fragmentation caused by mixing calls of `free()` and `malloc()` should be avoided. In a case where several lumps of allocated memory have to be freed and reallocated, the use of `free()` must be a single sequence, and the successive allocation another sequence.

Thread Use

Threads can cause very nasty synchronization and memory reference bugs so you need to consider are advantages of using them (like improved UI responsiveness) bigger than disadvantages like more complicated and error prone design of your application.

Threads can be used especially for operations which may block application UI for several seconds and where canceling the operation (using the cancel dialog) is not possible. For example, network operations will block when the connection times out due to network becoming unreachable e.g. when user walks out of the Bluetooth or WLAN range.

Doing given functionality concurrently to the rest of the application requires that it is first well isolated from the rest of the code functionality-wise and that isolated code that interacts concurrently with rest of the application and platform uses only APIs which are thread safe. Only after making implementation thread safe code can be run safely in a separate thread.

Using threads may make communication with the other code easier, but as a downside it doesn't offer address space protection like process separation does, and it may be harder to debug and verify to work correctly in all situations.

15.3.3 Maemo Conventions

This section contains a collection of guidelines especially meant for maemo coding that complement those more generic rules given in previous sections.

D-BUS

The D-BUS message bus is a central part of the platform architecture. D-BUS is primarily meant for lightweight asynchronous messaging, not for heavy data transmission. This section contains conventions concerning the use of D-BUS.

One of the design principles of D-BUS was that it must be lightweight and fast. However, it is not as lightweight and fast as a Unix socket. By default, *D-BUS communication is not reliable*. The system has a single D-BUS system bus

that is used by many processes simultaneously. A heavy load on the system bus may slow down the whole system. To summarize:

- The D-BUS system bus should not be used for heavy messaging or regular transfers of large amounts of data, because it slows down the communication of other processes and requires extra processing power (and with it, electrical power). For that purpose, using a Unix socket, pipe, shared memory or file should be considered.
- Broadcasting on a D-BUS should be avoided, especially when there are a lot of listeners, such as on the system bus.
- When sending a message with, for example, `dbus_connection_send()`, it should not be assumed that the message is surely received by the server. Things can happen to make the message disappear on the way: the server can fail before receiving the message when the message is buffered in the client or in the daemon, or message buffer in the daemon or in the server can be full, resulting in discarding the message. Therefore, *replies must be sent and received in order to make sure that the messages have been reliably conveyed.*
- In D-BUS, messages specified by the application framework (including Task Navigator, Control Panel, Status Bar, and some LibOSSO messages), UTF-8 encoding should be used as the character encoding for all string values.
- The libOSSO library of the application framework contains high-level functions (`osso_rpc_*`) that wrap the D-BUS API for message sending and receiving. Notice that provided API for D-BUS is not thread safe and cannot thus be used from more than one thread simultaneously.
- All maemo applications need to be initialized with `osso_initialize()` function, connecting to the D-BUS session bus and the system bus. One symptom of missing initialization is that the application starts from Task Navigator, but closes automatically after a few seconds.
- The underscore (`foo_bar`) should be used instead of "fancy caps" (`FooBar`) in D-BUS names (such as methods and messages). The name of the service or the interface should not be repeated in the method and message names. The D-BUS method names should use a verb describing the function of the method when possible.

Applications should listen to D-BUS messages indicating the state of the device, such as "battery low" and "shutdown". When receiving these messages, the application may, for instance, ask the user to save any files that are open, or perform other similar actions to save the state of the application. There is also a specific system message, emitted when applications are required to close themselves.

Signal Handling

The application must `exit()` cleanly on the TERM signal, since this is the phase when GTK stores the clipboard contents to the clipboard manager, without losing them.

The TERM signal is sent, for example, when the desktop kills the application in the background (if the application has announced itself as killable).

Event Handling

The maemo user interface (Hildon) is based on GTK+ toolkit (the same as used in Gnome), so many aspects of maemo's event handling are similar to GTK+. There are a few points specific to maemo, since it is designed to be used with a touch screen. Some points to remember here are:

- It is not possible to move the cursor position in touch screen without pressing the button all the time. This difference may affect drawing applications or games, where the listening events need to be designed so that the user can click the left and right side of screen without ever moving with mouse from left to right.
- The touch screen can take only one kind of click, so only the left button of the mouse is used, leaving no functionality for the right button.
- When necessary to open menus using the right button, it is still possible to show them by keeping the button down for a little longer than a fast click. These kind of context-sensitive menus are supported by the platform. See the GTK+ widget tap and hold setup function in the Hildon API Documentation [57].
- GtkWidget in maemo has been changed to pass a special `insensitive_press` signal when the user presses on an insensitive widget. This was added because of actions such as showing a banner when pressing disabled widgets, e.g. a menu.

Touch Screen Events The main interaction with users in maemo is through touch screen events. The main actions are:

- Single tap
- Highlight and activate
- Stylus down and hold
- Drag and drop
- Panning
- Stylus down
- Stylus up
- Stylus down and cancel

Of course, all those touch screen actions are treated by GTK as regular mouse events, like button presses, motion and scrolling.

Minimizing Problems in X Applications The application must not grab the X server, pointer or keyboard, because the user would not be able to use the rest of the device. Exceptions: System UI application and Task Navigator, Status Bar, Application menu, context-sensitive menu and combo box pop-up widgets.

Applications must not carry out operations that may block, for example, checking for remote files to see whether to dim a menu item, while menus or pop-ups are open.

If the operation is allowed to block (for example, when the device gets out of WLAN or Bluetooth range), the whole device user interface freezes if blocking happens with the menu open. This kind of check must be performed before opening the menu or pop-up, or in a separate thread.

All application dialogs must be transient to one of the application windows, window group or other dialogs. Dialogs that are not transient to anything are interpreted as system modal dialogs and *block the rest of the UI*.

Hardware Button Events

Hardware key	Event
Home key	HILDON_HARDKEY_HOME
Menu key	HILDON_HARDKEY_MENU
Navigation keys	HILDON_HARDKEY_UP, HILDON_HARDKEY_DOWN, HILDON_HARDKEY_LEFT, HILDON_HARDKEY_RIGHT
Select key	HILDON_HARDKEY_SELECT
Cancel key	HILDON_HARDKEY_ESC
Full screen key	HILDON_HARDKEY_FULLSCREEN
Increase and decrease keys	HILDON_HARDKEY_INCREASE, HILDON_HARDKEY_DECREASE

It is important not to use the GDK_* key codes directly; the HILDON_HARDKEY_* macros should be used instead (they are merely macros for the GDK key codes). This is because hardware keys may relate to different GDK key codes in different maemo hardware.

Also, the application must not act on keystrokes unless it truly is necessary. For example, Home and Fullscreen keys have system-wide meanings, and the application can be notified by other means that it is going to be moved to background or put in full screen mode.

State Saving and Auto Saving

State saving is a method to save the GUI state of an application, so that the same GUI can be rebuilt after the application has restarted. In practice, the state is saved into a file stored in volatile memory, therefore the saved state will not last over a reboot of the device.

The application must save its state on any of the following events:

- When the application moves from the foreground to the background of the GUI.
- Before the application exits due to a memory management measure.

The application must always load the saved state (if it exists) on start-up. However, the application can decide not to use the loaded state, by its own calculation. The saved state must always be usable by the application, even if the application is likely to make the decision not to use it.

Applications must implement *auto saving* if they handle user data. User data is information that the user has entered into the application, usually for permanent storage, e.g. text in a note. Auto saving ensures that the user does not lose too much unsaved data, if the application or the system crashes, or the battery is removed. Auto saving can also mean that a user does not have to explicitly save their data.

Auto saving can be triggered by an event, e.g. the battery low event or system shutdown. The application must auto save:

- When it closes (for example, because of a system shutdown or restart).
- When it is moved to the background.
- At regular, configurable intervals when the application is on top, unless it has recently been auto-saved due to another event.
- On receiving a message requesting auto saving.

Changes made to dialogs are not auto saved. Every application must provide an event handler to implement the fourth item. Naturally, the application does not have to auto save when there is no new, unsaved data.

More information about state saving can be found in LibOSSO documentation [57].

Configurability

Maemo applications must use GConf for saving their configurations if configuration changes need to be monitored. GConf is the configuration management system used in Gnome. It is advisable to read the documentation available on the GConf project page [22].

Normal configurations (that do not need to be monitored) must be placed in a file under the `~/.osso/` folder. The file can be easily written or interpreted by the GKeyFile parser from Glib. Applications must have sensible defaults, ensuring that the application will function even if the configuration file is missing.

Using GConf

Naming GConf keys: first a new GConf directory (for GConf keys) should be made under the GConf directory `/apps/maemo` for the program. For example, if the program is named `tetris`, the private GConf keys go under the GConf directory `/apps/maemo/tetris`.

Notice that GConf operations do not have transactions so you need to be careful when updating or monitoring more than one configuration keys that have some dependency to each other.

File System

File management operations for user files must be performed through an API provided by the application framework. In other words: the Unix file I/O API should not be used for files and folders that are visible to the user. The Unix (POSIX) file I/O API can still be used in other parts of the file system.

The main reason for this is that maemo needs to make the file management operations behave case-insensitively, even on the case-sensitive Linux file system. Additionally, the auto naming of files and other such actions can be implemented in the application framework.

All applications are normally installed under the `/usr` directory, and must use the directory hierarchy described for `/usr` in Filesystem Hierarchy Standard (FHS) [17]. In addition to the directories specified in the document, the following are employed under `/usr`:

- `share/icons/hicolor/<size>/apps` for icon files
- `share/sounds` for sound files
- `share/themes` for GUI themes

Configuration files are placed under `/etc/Maemo/<program name>`.

Variable data files that are not user-specific (but system-specific) must be created under `/var`, as suggested in FHS.

User-specific files should preferably be located under the directory `/home/user/apps/<program name>`. The user's home directory `/home/user` can be used quite freely when there is no risk of a conflict. However, the directory `/home/user/MyDocs` is reserved for the user's own files (such as document, image, sound and video files), i.e. for files that are visible through the GUI.

The directory for the program temporary files is `/tmp/<program name>/`. To create a temporary file, `mkstemp(3)` immediately followed by `unlink(2)` is used. The reason for calling `unlink(2)` early is to ensure that the file is deleted even if the application crashes or is killed with the kill signal. Note that `unlink(2)` does not delete the file until the program exits or closes the file.

The available space on the main flash is very limited, and is very easily exhausted by applications or by the user (for example, by copying images, music or videos to it). If the flash is full, all the system calls trying to write to the disk fail, and the device is slower because JFFS2 file system needs some free space.

Therefore, no file can grow without limitation, and file sizes in general must be kept small. Applications must *not* write to flash when it is starting up, otherwise the write may fail and the application will not start.

Memory Profiling

Memory profiling is a difficult task and requires good tools. Great care must be taken considering memory leaks. Even though desktop users reboot the system often and memory leaks are less visible for them, maemo users expect to reboot the system less (or almost never), so even the smallest memory leak is noticeable.

There are many memory tools [73] available for profiling and finding memory leaks. Using Valgrind is recommended.

When compiling the program with debug and profile information, one should use standard profiling tools like `gprof` to find bottlenecks, as well as `valgrind` to find memory leaks or overruns. The `XResTop` tool (a top-like application to monitor X resource usage by clients) can also be used to monitor the leaking of X resources.

Secure Programming

This section lists conventions to be used to prevent security vulnerabilities in programs. At the same time, these can make programs more robust. For more information on secure programming, see *Secure Programming for Linux and Unix HOWTO* [103] and *Secure Programming* [79].

Hildon applications (such as GTK+) cannot run with elevated privileges, so that should be avoided in `suid/sgid` programs. If the applications really require higher privileges, the code must be split in two: a back-end should be created with higher privileges to communicate with the user interface through IPC.

These are the general rules for secure programming:

- Instead of `system(3)` and `popen(3)`, the `exec(3)` family of functions should be used when executing programs. Also, shell access should not be given to the user.
- Buffer overflows must be prevented.
- Backed-up data should be as secure as the original data, i.e. security and secrecy requirements for the back-up data must be the same as for the original data that was backed up.
- The default permissions for a new file (`umask` value) must have permissions for the user only.
- Temporary files must be created using `mkstemp(3)`.
- Storing secrets, such as passwords, directly into the code should be avoided.
- One should not use `crypt(3)`.
- One should not use higher privileges than necessary for the job at hand. Extra privileges should be dropped when they are no longer needed, if possible.
- The success of all system functions should always be checked, such as `malloc(3)`.
- Random numbers for cryptography must be read from the `/dev/random` or `/dev/urandom` device.

Buffer Overflows To avoid buffer overflows, safe versions of some system functions can be used. These can be seen in the table below. For Glib/GTK code, the Glib versions are best, e.g. `g_strlcpy()` guarantees that the result string is NULL-terminated even if the buffer is too small, while `strncpy()` does not.

System functions:

Unsafe	Safe	Best (GLib)
strcpy()	strncpy()	g_strlcpy()
strcat()	strncat()	g_strlcat()
sprintf()	snprintf()	g_snprintf()
gets()	fgets()	

When using `scanf(3)`, `sscanf(3)` or `fscanf(3)`, the amount of data to be read should be limited by specifying the maximum length of the data in the format string. When reading an environment variable with `getenv(3)`, no assumptions should be made of the size of the variable; it is easy to overflow a static buffer by copying the variable into it.

Packaging

All in maemo platform is installed as Debian packages. It is recommended to read more about packaging from Debian documentation [9], [10] and about maemo specific issues in packaging and package distributing from other parts of maemo documentation.

15.3.4 Power Saving

Modern CPUs are concerned about efficient power usage, and many techniques have been developed in order to reduce this power consumption. One of those techniques is CPU frequency reduction, which scales down the CPU's clock after a period of low usage.

Idle mode represents the CPU's state of minimum resource usage, and it is the best way available to save power. In this state, many core functions are disabled, and only when an external event happens, such as tapping on the screen, the CPU is awakened and starts consuming more energy.

The maemo developers must be aware that maemo platform employs these techniques to reduce power, and programs must be written so that they consume *as little electrical energy as possible* by means of giving maximum opportunity for the CPU to become idle.

Some useful tips:

- If the program is not processing or doing something useful, it should be left idle. Many toolkits (such as GTK+) have an `idle()` function that can be called when the program has nothing to do.
- The CPU should not be unnecessarily encumbered, as power consumption is proportional to the CPU load. Many processes running and competing for the CPU leads to more power waste.
- Periodical alarms or timers should not be used, unless absolutely necessary. One should consider what would be a reasonable period for updating a progress bar on the screen. Continuous (permanent) timers that tick faster than once in every five seconds must not be used.
- Busy-waiting mode is not acceptable, even if it is busy waiting on empty loops or NOPs: significant power saving is achieved only when the CPU

is idle. The following example shows how almost the same code can either prevent or support sleep.

Power Management Unfriendly code:

```
SDL_EventState (SDL_SYSWMEVENT, SDL_ENABLE);

while (!done)
{
    SDL_WaitEvent(0);
    while (SDL_PollEvent (&event))
    {
        if (event.type == SDL_SYSWMEVENT)
        {
            ....
            here, for example, check
            event.syswm.msg->event.xevent.type
            etc ...
        }
    }
}
```

Power Management Friendly code:

```
while(wait_outcome = SDL_WaitEvent(0))
{
    SDL_PollEvent (&event);
    if (event.type == SDL_SYSWMEVENT)
    {
        ....
        here for example check
        event.syswm.msg->event.xevent.type
        etc ...
        break; // <----NOTE THIS
    }
}
if(wait_outcome == 0)
    handle_sdl_error();
```

This optimization also provides power management friendly code with improved responsiveness, because it prevents a waiting application from monopolizing the CPU, leaving machine time to concurrent processes.

- All UI timers should be stopped when moving to the background, and when the screen is turned off. Application-specific exceptions for this rule can be granted, when necessary.
- Updating the GUI should be avoided when the application is running on the background, or when the screen has been turned off. Unnecessary graphical elements or constantly updated screen components should also be removed.
- Doing polling for a resource wastes CPU and power; instead, asynchronous notifications sent by a separate daemon should be used.
- Small writes to file system should be grouped into bigger chunks.
- The use of remote peripherals, such as Bluetooth or Wireless, should be minimized. It is advisable to keep values in cache and avoid repeated and/or redundant queries to such devices. The use of hardware in general should be minimized. This refers to e.g. screen updates or external storage devices.

For more information, see Software Matters for Power Consumption [80].

Chapter 16

maemo.org

The landing site and home for maemo open source community developers is maemo.org. It provides documentation, support, bug reporting tool, project hosting framework, roadmaps and news for the developers. This chapter goes through these services.

16.1 Documentation

The maemo.org site provides official documentation[62] for each maemo major release. Also this manual is part of the official documentation.

The community documentation is viewable for everyone and editable by registered users in the maemo wiki[75]. Contributions to the community wiki and feedback about the official document are welcome!

16.2 Support

The place to get support from the community is the mailing list for maemo developers[68]. Prior to asking a possibly frequently asked question on the list, it is good to take a look at the mailing list archives[67]. A separate list for maemo users is also very active; it is targeted more at the end users than maemo application developers.

The most active maemo instant messaging channel at the time of writing is the maemo *IRC channel*[66] on *freenode*.

Commercial support for the maemo platform will be provided by *Forum Nokia*[20].

16.3 Bug Reporting

The proper way of giving feedback about the maemo platform is by using the public maemo Bugzilla[60]. It can be browsed anonymously, but registration is needed to be able to enter new bug reports or feature requests. The same Bugzilla tool is used to give feedback about the software running in the device, the SDK and the website including its documentation.

16.4 Project Hosting

The *maemo garage*[\[64\]](#) is a complete project hosting site for maemo community. This is the place where the actual development of maemo applications is preferably done, hence the name. Garage provides the developers for each project a version control management system, wiki, issue tracker and forums.

Application catalog[\[63\]](#) or the *Downloads* section of the maemo site is meant for end users to be able to easily download and install open source software products on their Internet Tablets. The maemo Application Catalog manual[\[59\]](#) has instructions for the maintainer on how to add a new tested application to the application catalog list, hopefully incubated first in the garage.

16.5 Roadmap and News

The maemo community members can keep a track on what is happening now by reading the news[\[69\]](#). For future plans, the roadmap[\[70\]](#) is kept open for avoiding overlapping work and giving visibility where the development effort is going for maemo itself.

Bibliography

- [1] Application binary interface (ABI) for the ARM architecture.
<http://www.arm.com/products/DevTools/ABI.html>.
- [2] ALSA projects home page. <http://www.alsa-project.org/>.
- [3] BlueZ projects home page. <http://www.bluez.org/>.
- [4] Bunny the Fuzzer. <http://code.google.com/p/bunny-the-fuzzer/>.
- [5] Busybox. <http://www.busybox.net/>.
- [6] Cairo projects home page. <http://cairographics.org/>.
- [7] Codenomicon. <http://www.codenomicon.com>.
- [8] Coverity. <http://www.coverity.com/>.
- [9] Debian - The Universal Operating System. <http://www.debian.org/>.
- [10] Debian New Maintainers' Guide.
<http://www.debian.org/doc/manuals/maint-guide/>.
- [11] Debian Policy Manual. <http://www.debian.org/doc/debian-policy/>.
- [12] Debian Repository HOWTO. [http://www.debian.org/doc/manuals/
repository-howto/repository-howto](http://www.debian.org/doc/manuals/repository-howto/repository-howto).
- [13] debsign manual page.
<http://www.penguin-soft.com/penguin/man/1/debsign.html>.
- [14] Desktop Entry Specification. [http://standards.freedesktop.org/
desktop-entry-spec/desktop-entry-spec-latest.html](http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html).
- [15] Esound white paper.
<http://developer.gnome.org/doc/whitepapers/esd/>.
- [16] Evolution project's home page.
<http://www.gnome.org/projects/evolution/>.
- [17] Filesystem Hierarchy Standard. <http://www.pathname.com/fhs/>.
- [18] Flawfinder. <http://www.dwheeler.com/flawfinder/>.
- [19] Fortify. <http://www.fortifysoftware.com/>.

- [20] Forum Nokia. <http://www.forum.nokia.com/>.
- [21] Fuzz. http://en.wikipedia.org/wiki/Fuzz_testing.
- [22] GConf configuration system.
<http://www.gnome.org/projects/gconf/>.
- [23] GNU core utilities. <http://www.gnu.org/software/coreutils/>.
- [24] Gdb: The GNU project debugger. <http://sourceware.org/gdb/>.
- [25] GDK Key Values. <http://library.gnome.org/devel/gdk/unstable/gdk-Key-Keyboard-Handling.html>.
- [26] GNU gettext. <http://www.gnu.org/software/gettext/>.
- [27] Glade User Interface Builder. <http://glade.gnome.org/>.
- [28] Gnu C library. <http://www.gnu.org/software/libc/>.
- [29] GNOME: The free software desktop project. <http://www.gnome.org/>.
- [30] GNOME mobile. <http://www.gnome.org/mobile/>.
- [31] Gnomevfs API reference.
<http://library.gnome.org/devel/gnome-vfs-2.0/stable/>.
- [32] GNU 'make'.
<http://www.gnu.org/software/make/manual/make.html>.
- [33] Gstreamer projects home page. <http://gstreamer.freedesktop.org/>.
- [34] GStreamer Application Development Manual.
<http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html>.
- [35] Gstreamer playbin base plugin. <http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-playbin.html>.
- [36] GStreamer Plugin Writers Guide. <http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html>.
- [37] GStreamer Core Reference Manual. <http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/>.
- [38] Gtk+ projects home page. <http://www.gtk.org/>.
- [39] GTK+ Accelerator Groups. <http://library.gnome.org/devel/gtk/unstable/gtk-Key-Keyboard-Accelerators.html>.
- [40] GtkMenu.
<http://library.gnome.org/devel/gtk/unstable/GtkMenu.html>.
- [41] GTK+ Reference Manual.
<http://library.gnome.org/devel/gtk/unstable/index.html>.

- [42] GtkUIManager. <http://library.gnome.org/devel/gtk/unstable/GtkUIManager.html>.
- [43] HAL specification. <http://people.freedesktop.org/~david/hal-spec/hal-spec.html>.
- [44] Hildon. <http://live.gnome.org/Hildon/>.
- [45] Hildonmm project page. <https://garage.maemo.org/projects/maemomm/>.
- [46] Hildon Theme Howto. <http://live.gnome.org/Hildon/ThemeHowTo>.
- [47] Hildon Theme Tools Overview. <http://live.gnome.org/Hildon/ThemeToolsOverview>.
- [48] Hildon Theming Overview. <http://live.gnome.org/Hildon/ThemingOverview>.
- [49] How to Become Root. <http://maemo.org/community/wiki/HowDoIBecomeRoot>.
- [50] htop - an interactive process viewer for linux. <http://htop.sourceforge.net/>.
- [51] Icon Theme Specification. <http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html>.
- [52] Jalimo. <http://www.jalimo.org/>.
- [53] Klocwork. <http://www.klocwork.com>.
- [54] Ka-Ping Yee's User Interaction Design for Secure Systems. <http://www.ischool.berkeley.edu/~ping/sid/uidss.pdf>.
- [55] Maemo repository for libglade2. <http://repository.maemo.org/pool/diablo/free/libg/libglade2/>.
- [56] Linux kernel coding style. <http://pantransit.reptiles.org/prog/CodingStyle.html>.
- [57] Maemo API reference. http://maemo.org/api_refs/.
- [58] Maemo.org Application Catalog. <http://maemo.org/maemowiki/ApplicationCatalog>.
- [59] Maemo application catalog manual. <http://maemo.org/community/application-catalog/user-manual.html>.
- [60] Maemo bugzilla. <http://bugs.maemo.org>.
- [61] Maemo Diablo repository. <http://repository.maemo.org/stable/diablo/>.
- [62] Maemo developer documentation. <http://maemo.org/development/documentation/>.

- [63] Maemo downloads. <http://maemo.org/downloads/>.
- [64] Maemo garage. <http://garage.maemo.org>.
- [65] Maemo home page. <http://maemo.org/>.
- [66] Maemo IRC channel. <http://maemo.org/community/irc.html>.
- [67] Maemo-developers mailing list archive.
<http://lists.maemo.org/pipermail//maemo-developers/>.
- [68] Maemo mailing lists.
<http://maemo.org/community/mailling-lists.html>.
- [69] Maemo news. <http://maemo.org/news/>.
- [70] Maemo roadmap. <http://maemo.org/intro/roadmap.html>.
- [71] Ruby maemo bindings project page.
<https://garage.maemo.org/projects/ruby185/>.
- [72] Maemo SDK releases. <http://maemo.org/development/sdks/>.
- [73] Maemo SDK tools. <http://maemo.org/development/tools/>.
- [74] Xubuntu image with maemo SDK installed.
https://garage.maemo.org/frs/?group_id=277.
- [75] Maemo wiki. <http://maemo.org/community/wiki>.
- [76] Matchbox home page. <http://matchbox-project.org/>.
- [77] Designing Matchbox Themes HOWTO.
<http://matchbox-project.org/documentation/themes/>.
- [78] Mono. <http://www.mono-project.com/Maemo>.
- [79] Oliver Friedrichs (of@securityfocus.com). Secure Programming.
<http://marcin.owsiany.pl/sec/secure-programming.html>.
- [80] Nathan Tennes on Embedded.com. Software Matters for Power Consumption. <http://www.embedded.com/story/OEG20030121S0057>.
- [81] OpenOBEX home page. <http://dev.zuckschwerdt.org/openobex/>.
- [82] Oprofile - a system profiler for linux.
<http://oprofile.sourceforge.net/>.
- [83] Pango Reference Manual.
<http://library.gnome.org/devel/pango/unstable/index.html>.
- [84] Petname. <http://petname.mozdev.org/>.
- [85] Peter Gutmann's manuscript Security Usability Fundamentals.
<http://www.cs.auckland.ac.nz/~pgut001/pubs/usability.pdf>.
- [86] Pymaemo home page. <http://pymaemo.garage.maemo.org/>.

- [87] QEMU, open source processor emulator.
<http://fabrice.bellard.free.fr/qemu/>.
- [88] Federico Mena Quintero, Miguel de Icaza, and Morten Welinder.
GNOME Programming Guidelines. <http://developer.gnome.org/doc/guides/programming-guidelines/book1.html>.
- [89] RATS.
<http://www.fortifysoftware.com/security-resources/rats.jsp>.
- [90] Scratchbox home page. <http://scratchbox.org/>.
- [91] Scratchbox Apophis.
<http://www.scratchbox.org/download/scratchbox-apophis/>.
- [92] Installing Scratchbox. <http://www.scratchbox.org/documentation/user/scratchbox-1.0/html/installdoc.html>.
- [93] strace. <http://sourceforge.net/projects/strace/>.
- [94] Telepathy. <http://telepathy.freedesktop.org/>.
- [95] Ubuntu home page. <http://www.ubuntu.com/>.
- [96] Ubuntu for debian developers.
<https://wiki.ubuntu.com/UbuntuForDebianDevelopers>.
- [97] uclibc. <http://www.uclibc.org/>.
- [98] Untidy. <http://untidy.sourceforge.net/>.
- [99] Valgrind. <http://valgrind.org/>.
- [100] Valgrind User Manual.
<http://valgrind.org/docs/manual/manual.html>.
- [101] Video for Linux Two. <http://www.thedirks.org/v4l2/>.
- [102] WebScarab. http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.
- [103] David A. Wheeler. Secure Programming for Linux and Unix HOWTO.
<http://www.dwheeler.com/secure-programs/>.
- [104] Wikipedia List of Tools for Static Code Analysis. http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.
- [105] Xephyr. <http://www.freedesktop.org/wiki/Software/Xephyr>.
- [106] zzuf. <http://sam.zoy.org/zzuf/>.