Maemo Diablo Reference Manual for maemo 4.1

Architecture

December 22, 2008

Contents

Chapter 1

Architecture

This chapter describes the high-level architecture of the maemo platform from an application developer's point of view (see section 1.3 for a more detailed view). There is also a comparison of maemo and popular Ubuntu and Debian desktop Linux distributions in the chapter.

1.1 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.

1.1.1 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.

1.1.2 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 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.

1.2 Software Layers

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

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 Applications

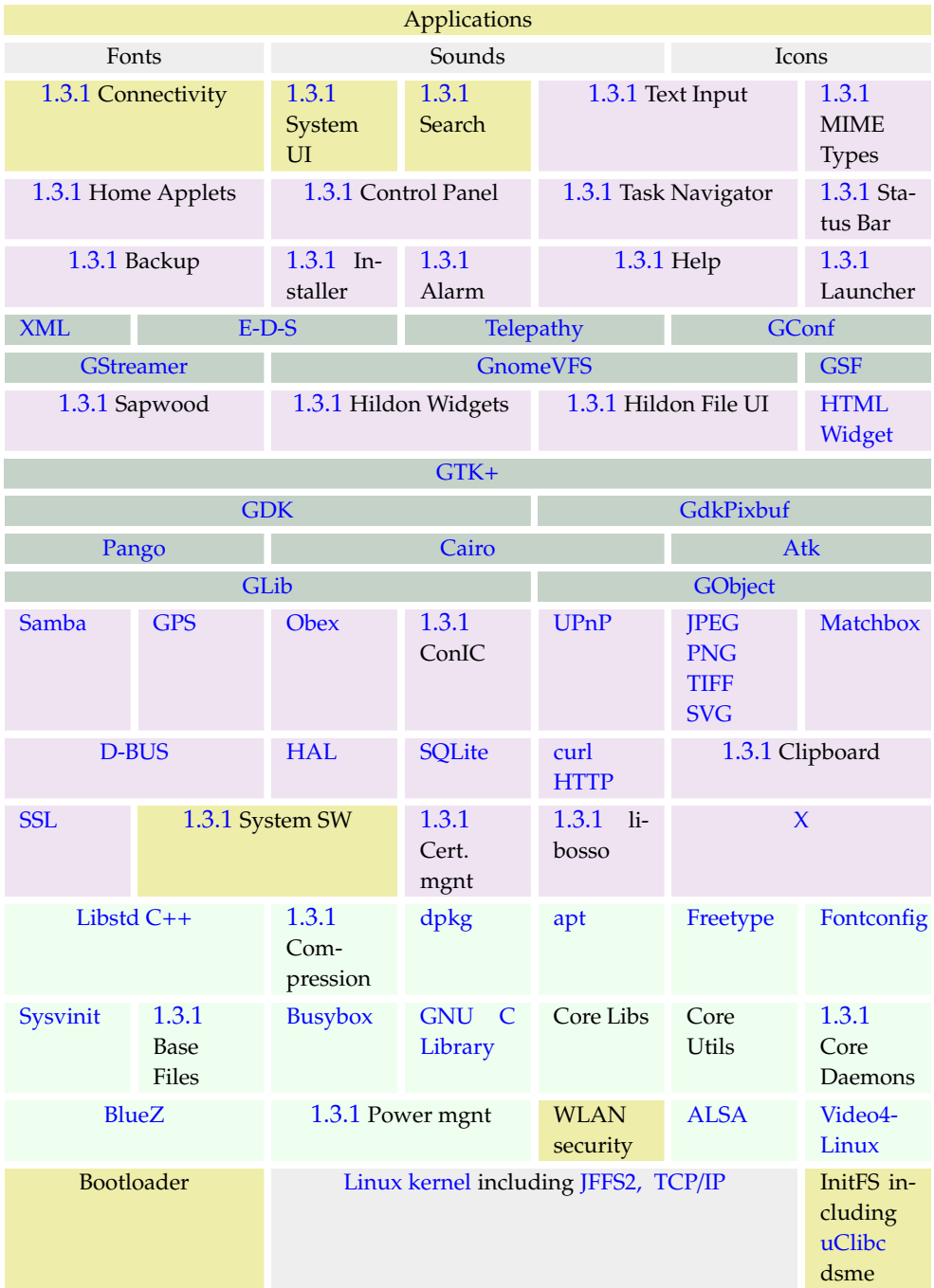
The applications are built on top of the Hildon framework. Simple applications link just with Hildon libraries, GTK+, Glib, and libosso in order to use the

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.

1.3 Software Decomposition View

1.3.1 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).



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 /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 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 misceallaneous daemons.

Packages include

- sysklogd (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

1.3.2 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*[7] 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.

1.3.3 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.

1.3.4 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.

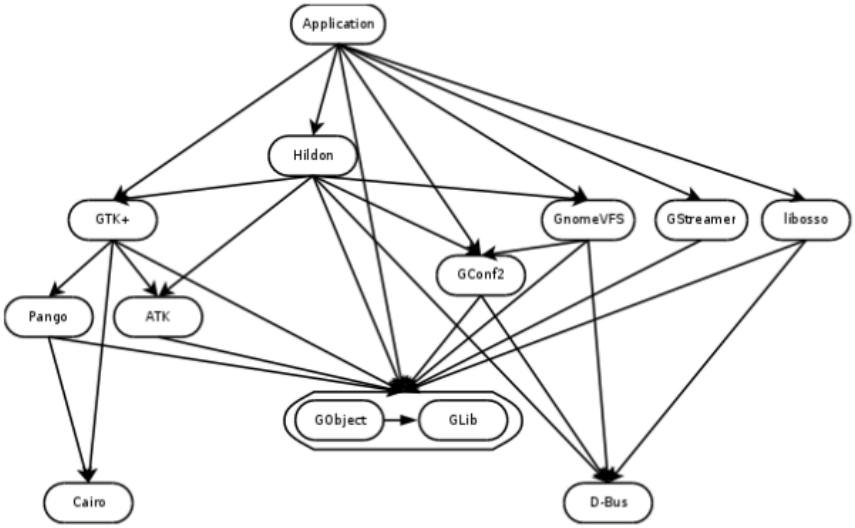


Figure 1.1: Major Application Framework Components

The figure 1.1 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.

1.3.5 Base Distribution

Maemo is based to a large extent on the same open-source components as found in the Debian[3] 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*[2].

Also notably the maemo platform adheres closely to the GNOME Mobile Platform[6]. 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.

1.4 Run Time View

1.4.1 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.

1.4.2 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 1.4.7 for more information.

1.4.3 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 1.2 shows how this happens in practice.

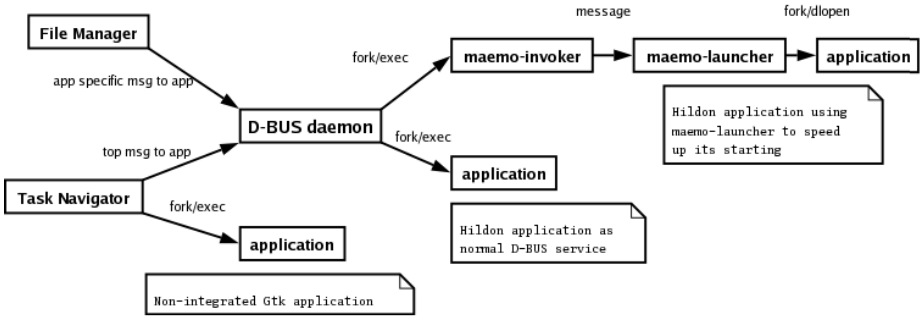


Figure 1.2: Application activation methods

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.

1.4.4 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.

1.4.5 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 existance 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).

1.4.6 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.

1.4.7 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.

1.4.8 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 top 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.

1.5 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.

1.6 Maemo Compared to Desktop Linux Distributions

Ubuntu[8] is a popular Debian-based Linux distribution for the ordinary desktop. The Ubuntu wiki[9] 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[5] 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	Configration 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/password*, 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 [2] replacement for the essential GNU utilities[4] 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[10] 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.

Bibliography

[1] Application binary interface (ABI) for the ARM architecture. http://www.arm.com/products/DevTools/ABI.html.

[2] Busybox. http://www.busybox.net/.

[3] Debian - The Universal Operating System. http://www.debian.org/.

[4] GNU core utilities. http://www.gnu.org/software/coreutils/.

[5] Gnu C library. http://www.gnu.org/software/libc/.

[6] GNOME mobile. http://www.gnome.org/mobile/.

[7] Maemo Diablo Reference Manual for maemo 4.1, chapter Kernel Guide. http://maemo.org/development/documentation/.

[8] Ubuntu home page. http://www.ubuntu.com/.

[9] Ubuntu for debian developers. https://wiki.ubuntu.com/UbuntuForDebianDevelopers.

[10] uclibc. http://www.uclibc.org/.