Maemo Diablo Getting Started

Training Material

for maemo 4.1

February 9, 2009

Contents

Preface

Legal notice

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

Chapter 1

Introduction

1.1 Introduction to Maemo Getting Started

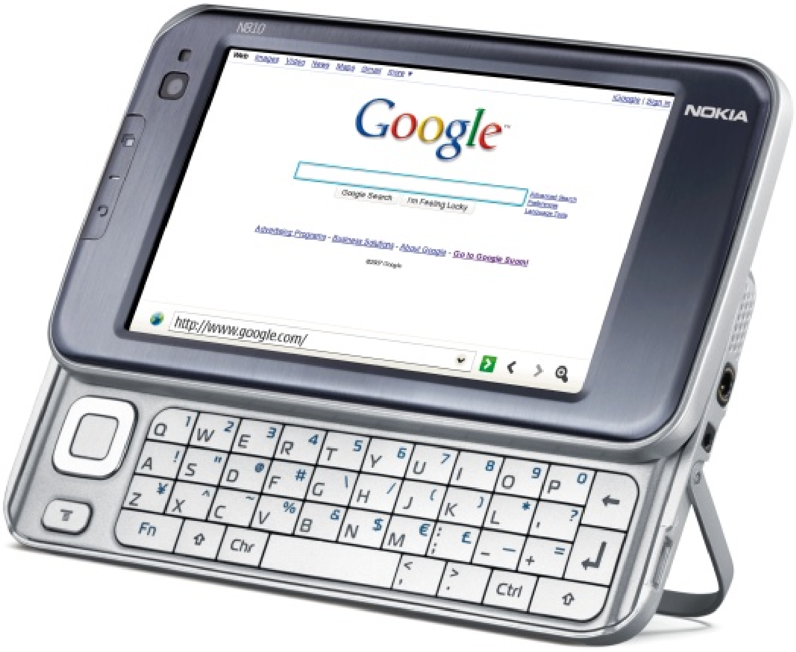


Figure 1.1: Nokia N810

On the 25th of May 2005 Nokia released the first version of maemo$^{\text{TM}}$software development kit to the public. After the first release, multiple updates have been made to the SDK. The SDK is meant for software development with the new Internet Tablet-class devices and this is the first time that a major company is basing their commercial product so much on free software. Since the underlying software in the devices consists of mainly open source software, this makes the devices an attractive platform for which to write software. Basing the SDK and the device on open source software also means that porting existing software is fairly easy.

This material covers the basics of installing the maemo SDK version 4.1, Diablo.

Pre-requisites for using this material effectively are: knowledge of C programming in Linux-environment, basic knowledge about Debian command-line tools and utilities used in software installation and development in Debian based Linux systems.

The SDK contains a broad range of software which is in use in a lot of free software projects. The material will however only cover the minimal amount required to get you up and running with maemo development.

The material starts by covering the basic environment of the target devices and then covers the range of tools to use and their best uses in software development.

Where applicable, the material will also note the recommended development and design practises, since the target devices are not "ordinary PCs". If your background is working with restricted resource device environments, you'll already know most of these rules, but there are also some additional hints sprinkled throughout the material.

More information about the maemo training material is available from the maemo training wiki pages (http://wiki.maemo.org/Maemo$_{T}$*raining*) maintained by maemo community. Notice that the information in maemo wiki is not verified by Nokia and thus Nokia cannot be responsible of that information.

We hope that you will enjoy the environment, your imagination is the only thing stopping you now!

Chapter 2

What is Maemo

2.1 What is this thing called maemo^{TM}?

Maemo is an open source development platform for Internet Tablets. It means the collection of software that is used to develop and test software for the Internet Tablet-class devices, the first of which was the Nokia 770. It was later followed by the Nokia N800 and the Nokia N810. In this material we will be referring to all of these devices as **Internet Tablets**. Maemo is a registered trademark of Nokia Corporation.

This version of the material covers maemo SDK version 4.1.x as well as Nokia N800 and Nokia N810 Internet Tablets running OS2008.

Figure 2.1: Nokia N800

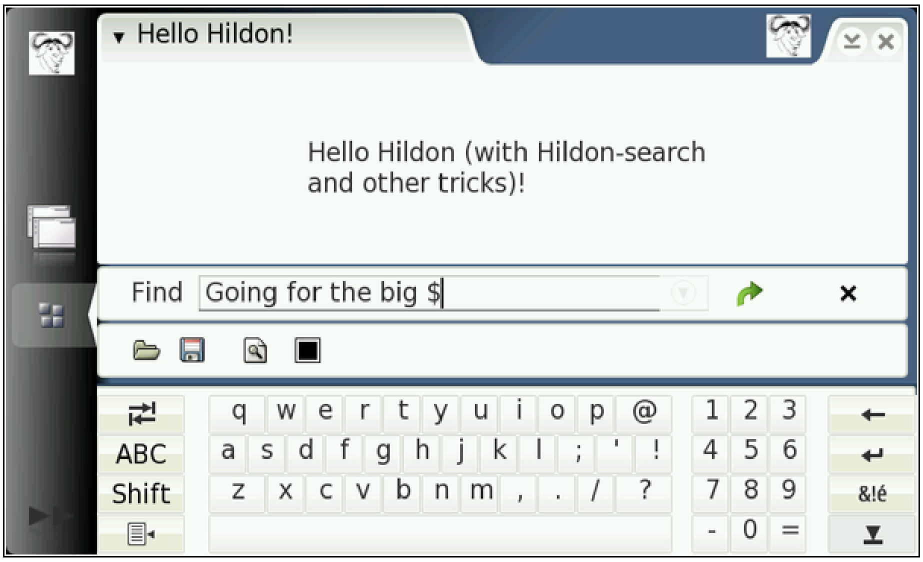
For a programmer, the Internet Tablets are really interesting as so much in them is based on free software and thus it's possible to use the same tools that are used in normal software development on other free and open source environments.

If you're coming from the Windows-world, or even the Symbian-world, this might be a new kind of encounter for you. All the tools, libraries and development processes that are used in maemo are equally used and applied in the desktop application arena, as well as for building server software. This is in part due to the GNU project (gnu.org), which has implemented a lot of the tools infrastructure in a highly portable way. The main graphical interface libraries come from the GNOME project (gnome.org), which is one of the most popular graphical environments used in Linux distributions.

By reusing existing portable and tested tools, we gain in an accelerated application development time. This also means that we can take the tool-set and apply it for writing software for embedded systems.

2.2 Internet Tablet overview

The devices are smaller than a laptop, larger than a PDA, and quite lightweight. Some of them (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 your software will need to be designed this in mind. There is also a possibility of using an on-screen keyboard with the stylus and this includes a hand-writing recognition and a predictive input system to aid the user. In all devices, there is a limited set of hardware buttons available for applications.



▾ Hello Hildon!

Hello Hildon (with Hildon-search and other tricks)!

Find: Going for the big $

Figure 2.2: The Virtual Keyboard (VKB)

As programmers appreciate knowing a bit about the fundamentals of the devices for which they program, table 2.1 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 optimised 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 filesystem		
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: • **TMS320C55x DSP logic** (Backward compatibility with the 54x-series) • **ARM1136 core** ("ARMv6") with an MMU (Backward compatibility with ARM926)		

Table 2.1: 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:

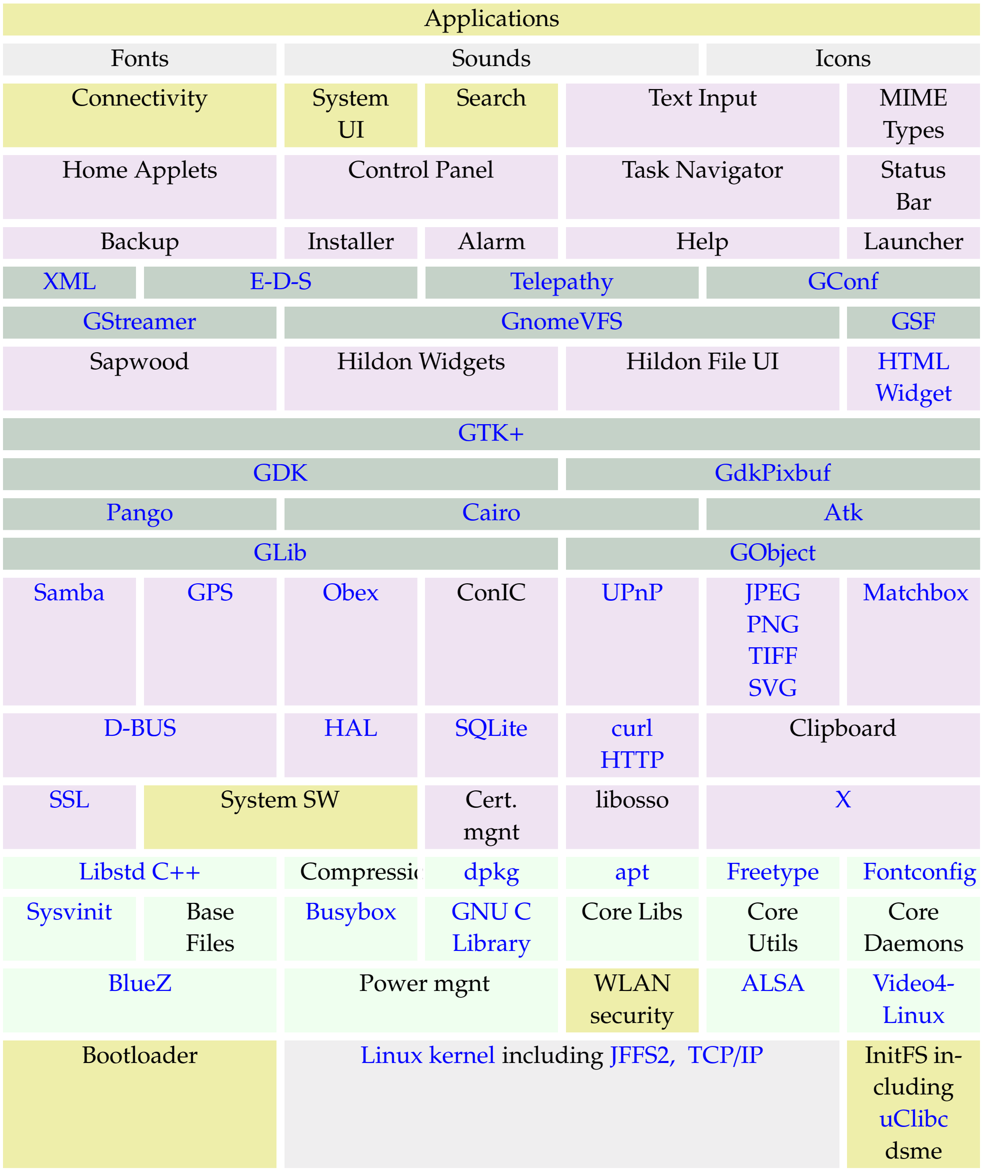
- There is not a lot of RAM (compared to a "PC"), and the memory is shared between all the applications that are executing at any given time.
- The system runs a modified Linux kernel 2.6 (omap-port).
- The system library is GNU libc 2, meaning that most software can be ported without too much effort (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. This means that about 192 MiB is available to be shared between applications.
- The built-in flash uses a filesystem specifically designed for flash memory, and contains transparent compression and decompression. This means that sometimes optimising 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 filesystem. 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 utilised, 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.

2.3 Maemo runtime environment

Below is a table of the software "stack" for the maemo platform:



We'll start from the bottom layer and go upwards by covering the services:

Linux 2.6 kernel Processes hardware events, system-wide memory allocation, process creation and everything that you would expect from a modern multi-tasking UNIX-like kernel. Not covered in this material.

X Server A program that implements access to the graphics hardware and converts HID (human interface device) events from the kernel into events for the X server's clients. Explained shortly.

D-Bus A service that allows related processes to pass events to each other. The service runs as a daemon, which is a process that runs in the background.

The D-Bus daemon also passes important events from the core system to applications (e.g., "battery low"). Interfacing with D-Bus is an important part of integrating your application with the runtime environment. D-Bus was developed to provide a message bus for Linux desktop applications, the D comes from "Desktop". In fact, normally one would have at least two daemons, one that processes and sends system level events and one to allow related processes to communicate with each other inside one user's graphical session. D-Bus is more thoroughly covered in the "maemo Platform Development" material.

X window manager (customised Matchbox) Controls where the graphical applications' windows will be placed.

Task navigator Graphical program that is used to switch between applications. Always running, even if your application will be full-screen (Task navigator will be invisible in this case). Appears on left side of the screen when applications are not running in full-screen mode.

Home/Desktop A graphical program that implements a user-selectable background picture. Also provides space for applets which are small programs that draw on the background some useful (or not) information to the user. Applets are not covered in this material.

Status bar Implements the top-right area of the screen that holds the various plug-ins that indicate status and allow the user to easily change settings. Together with Task navigator and Home/Desktop, implements the screen that the user will see when the device has started.

Sapwood A daemon that caches images used to implement the overall graphical look and feel for applications designed for maemo. Used in the background by the GTK+ library.

Control panel A simple application for most system configuration tasks. It is possible to write your own Control panel plug-ins by making them dynamically loadable objects which the Control panel will load on demand. Not covered in this material.

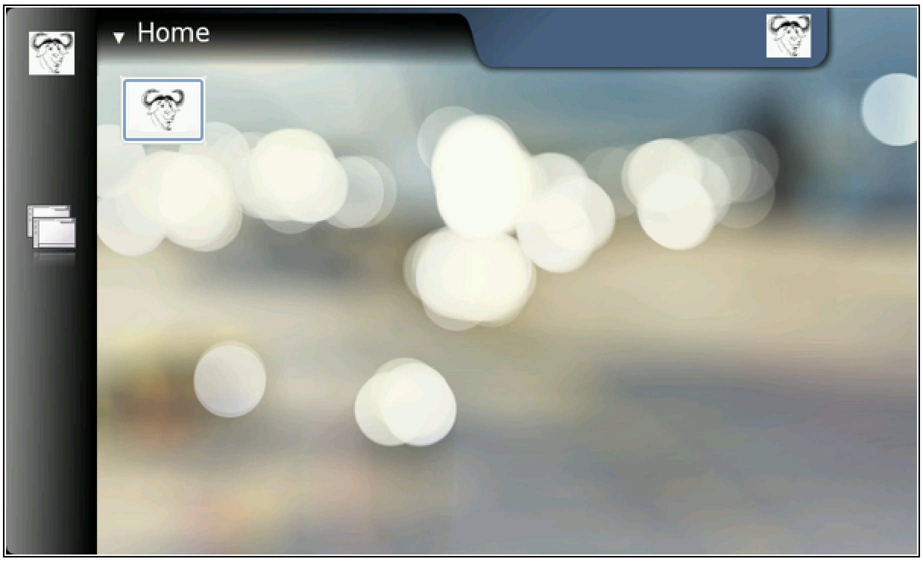


Figure 2.3: Task navigator, Home and Status bar. Also shown are the icons (the GNU heads) of the hello-world-app package.

2.4 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 is 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 will relay keyboard and pointer events back to the client.

When speaking about clients and servers, it's 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 some X server to create a window. A window is a rectangular area into which the client can draw. Note that the client can ask the server to position the window at some screen location, but normally doesn't. 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 minimising, 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 similar ways. The "Desktop" (whatever the word means inside a computer) is normally implemented by yet another client. And yes, the taskbar that you might see is yet another client. Even the screen saver is a separate client. In the 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 (note that 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's 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 practise this is most often a DNS-name or an IP-address of the server, but depending on local NSS (Name Service Switch) 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 you have and it will number its screens starting from 0. You may also omit the screen number 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, you normally can tell it which screen to create and control.

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

To instruct an X client to connect to another X server, it's then necessary to modify the environmental variable: `export DISPLAY=:2.0` for example. Then start your 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, you may also use `export DISPLAY=:2` .

You might be wondering what all of this has to do with maemo, but you'll see in a short while when we install the environment and start testing the applications.

In the case that you skipped the intro, this is a good point to remind you 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 Internet Tablet is based on the `Kdrive` version of the X.Org X server (yes, so many versions). However, your Linux desktop is running a regular X.Org server.

Also note that 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 (x.org) and rahul.net/kenton/xsites.framed.html. Also, ssh can be used to tunnel X11 connections securely over networks. Please see X Over SSH2 Tutorial for examples.

2.5 Typical maemo GUI application

We next take a look at 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 stuff. However, the libraries presented below also provide their own APIs to similar functions, so you should always check whether you can use them directly, and avoid doing POSIX-level and system-level calls when possible. This will make your 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. Not really covered in this material since most of the things that we need are in the higher-level libraries.

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 wouldn't use Xlib API directly but would instead use some easier toolkit which in turn will use Xlib. Not covered in this material other than the introduction and on a "need to know"-basis.

GLib An 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 which wants to implement drawing directly, for example in order to implement custom widgets. In theory, GDK is meant to be graphics system independent and mostly is. Complete abstraction however is not yet complete, but for us 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's not always from top-to-bottom and left-to-right. Uses GLib and GDK. Used by GTK+ for all displayed text. Covered only where necessary in this material.

ATK The Accessibility ToolKit. Provides generic methods by which an application can support people with special needs with respect to using computers. Not covered in this material.

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 behaviour 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. Basic operations are covered in this material.

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 kind of devices and services. By using GnomeVFS, an application doesn't 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. Basic operations are covered in this material.

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 that allows an application to connect to D-Bus in a simple and consistent manner.

Also provides an application state serialisation 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 user switched to another application. Useful to conserve battery life on portable devices. Only basic parts of LibOSSO are covered in this material and more advanced use is covered in "maemo Platform Development" material.

Whew, that was quite a list. As you can imagine, an introductory material like this cannot even try to cover all the possibilities or the API functions available in these libraries. We will try to do our best describing the bare minimum in order for you to start writing applications for maemo.

There are some caveats related to API changes between major GTK+ versions, which will be mentioned in the text, so do not go and copy-paste existing code blindly. Also this is a good place to remind you that most of the source code that you can find easily is covered by either the GPL or LGPL licenses. This material tries to stay away from legalese, but to put it bluntly, **you cannot copy GPL-ed source code into your proprietary projects (i.e., closed source). This is against the license. GPL considers static linking as distribution (copying) too. Even linking dynamically from proprietary code against GPL-ed libraries might be interpreted as prohibited by the license!**

Note that it is normally allowable to read and learn from GPL-ed source code. Indeed, unless you're copying line by line from existing code base, this can be an invaluable tool to learn how things are done in the "real world". This is quite the opposite when reading proprietary source code as there is a risk of learning something that the code owner will consider "intellectual property". If you then use this "knowledge" in a free or open source project, you risk polluting that project with unnecessary legalese.

As a side note, you might be interested to learn that almost all of the GNOME-libraries are released under the LGPL-license. This means that you can create proprietary software which will link into LGPL-libraries dynamically at runtime. If you however modify the LGPL-library, the modifications must be available in source form to the entities you distribute the binaries to under the original license (LGPL).

2.6 Battery Doesn't Last Forever!

Low power consumption is one of main hardware design goals with mobile devices because of the limited electric charge in their power supplies. If the hardware is designed correctly, it may itself contain logic and rules to enter different power saving states. To enter these power saving states the hardware requires that there is no activity in the system, in other words, there is no task ready to be run by the OS kernel scheduling mechanism. Even if power saving functionality is implemented in the hardware, activating it might not always be possible. If the applications running on the hardware are "misbehaving", then the system will be active all of the time and this makes it impossible for the power saving features to be activated at hardware level. Some of these power saving features include: changing the clock frequency dynamically, supporting multiple operating voltages and switching integrated peripherals' sleep modes.

Different parts of the hardware will require different amounts of power to run. The following pie diagram is not based on any real measurements but roughly shows how power consumption is distributed between different subsystems in a device:

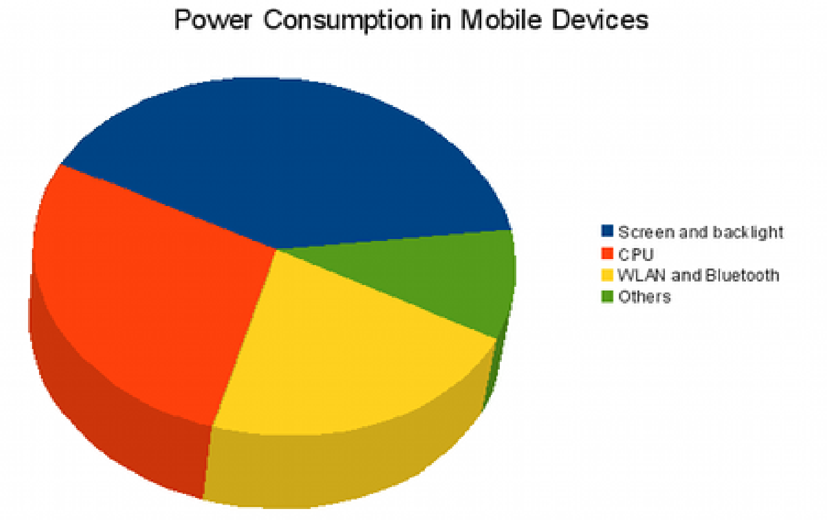


Figure 2.4: Diagram of power consumption in Mobile Devices

Keep the previous diagram in mind when you consider your application's "electricity needs". For example, in a GPS mapping application, the GUI interface is a graphical map with the position and other relevant information. If such display would keep the backlight active all the time, it would cause excessive power usage. In this case, it might be prudent to ask the user whether they really want to keep the backlight on, or provide a user-controllable button for switching the backlight on/off.

When designing for mobile devices, it's important that you (as the application designer) consider different approaches to problems. You should also consider system-wide power usage, i.e., how your application will change the power performance of the device. The above diagram might seem rather obvious, but it is often too tempting to start premature optimisation of memory usage or code speed, and forget about power optimisation. In mobile environments, all three are important.

2.7 maemo development resources

maemo has its own website (at maemo.org), which includes:

- Links to mailing lists (and their archives)
- Link to the defect tracking system (bugzilla) at bugs.maemo.org
- White papers and tutorials

- Licenses used
- Trademark usage guidelines
- Development news
- Download information
- API references library versions used in maemo SDK
- The maemo wiki and other community supported resources (garage)

There is also an IRC channel for developers (#maemo@FreeNode). You can find people related to Scratchbox and maemo hanging there lot of times. Scratchbox also has its own channel.

Note that everything you discuss on the IRC channel and mailing lists (as well as bugs you post using bugzilla) is *public information*. You might want to ask someone whether your working environment has a policy on using public resources before using them.

2.8 Other programming interfaces

Using the previously presented software library stack is not the only possibility. The platform also includes SDL (Simple DirectMedia Layer) which is a library that was originally developed by Loki Entertainment, a company that specialised in creating Linux-versions of popular commercial games. It contains most of the code necessary to write games and implement low-level graphics. SDL is not covered by this material, but you can find the API (and other) documentation at libsdl.org .

If you like to work with APIs that are hard to understand, you can use the Xlib-interface and implement your interactive program directly with it (not recommended for the faint hearted).

Using either SDL or Xlib is not directly supported by the environment and will lead to applications which will not conform to the "look and feel" common to applications designed for maemo. Such problems should be avoided so that end-users do not get confused (they would expect an uniform interface for all the applications they use). You won’t be able to utilise the virtual keyboard or handwriting recognition in your application either. For some full-screen pointer driven games this might not be a big issue, but for "normal" GUI applications that the user will enter data and text, it is an issue.

It is also be possible to use the Python programming language to write GUI programs (there are bindings for the Hildon toolkit as well) and there are community driven projects for using Ruby and Vala as well as others.

Chapter 3

Installing the SDK

3.1 Getting started

This chapter covers the pre-requisites and installation of the development environment. The maemo SDK consists of libraries and tools enabling the development of applications for maemo and Internet Tablets. This SDK must be installed into an development environment called Scratchbox in order for it to be useful.

At this point, you should check the maemo training wiki pages maintained by maemo community. They might contain some information which affects the SDK installation process. Notice that the information in maemo wiki is not verified by Nokia and thus Nokia cannot be responsible of that information.

We'll start by installing Scratchbox first and then proceed by installing the maemo SDK inside Scratchbox. The next chapter covers testing of the installation using simple text and graphical programs.

Installing the SDK can also be done by using automatic installation scripts, using which is covered in the SDK installation instructions (part of the SDK). This material will cover installation in a more step-by-step fashion, so that you may easily create custom Scratchbox targets in the future.

3.2 What is Scratchbox?

Now that you've seen what both Internet Tablet and applications designed for maemo are made of, you might be wondering how to write your own applications. If you've used the various GNU tools before you also might be wondering how all the different versions of tools and libraries are handled during development.

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

The name "Scratchbox" comes from "Linux from scratch" + "chroot jail" (sandbox). This also tells you something about its implementation and intended use. While working inside Scratchbox, you'll be running programs in a changed root environment (chroot). In Linux systems it's 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 else 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 you can 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 (not covered in this material).
- Supports multiple configurations for each developer.
- Supports executing target executables on the hardware target, via a mechanism called `sbrsh` (not covered in this material).
- Supports running non-native binaries on the host system via instruction set emulators (Qemu is used).

Beside these main features, it's possible to develop your 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 we have setup our source files correctly and write a couple of configuration files, we can create binary distribution packages 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 we can install other development packages over the Internet when we need them (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's open for outside contributions. For an in depth coverage on Scratchbox capabilities please see scratchbox.org.

In this material we'll be using only the Scratchbox capabilities that are necessary to use the maemo SDK.

3.3 Scratchbox components

Before installing Scratchbox, we need to cover some terminology that it uses in its documentation. For most of the time Scratchbox will be abbreviated as sbox from now on.

Scratchbox terminology:

core package package that contains the core tools implementing sbox. These also include a host compiler (gcc) that can be used to build additional tools for sbox.

libs package contains the necessary libraries for the core to operate.

devkit a package for sbox that contains additional development tools. We'll be interested in 4 devkits (listed later).

toolchain compilers, linkers and tools for a specific target. We'll be needing two, but we'll use the x86-one for now.

target the active toolchain and configuration we're using currently. A target uses a selected toolchain and contains a filesystem to use and related configuration. You can have multiple targets, even if they all use the same toolchains. This makes it easy to try something different, or start a parallel target to test things from scratch.

- Note that an sbox target doesn't technically mean the same thing as the physical target device you might have (**Internet Tablet**).

rootstrap a target root filesystem image that can be used as a basis for further development. Rootstraps normally contain the necessary files for some specific development target, but sometimes only act as a starting point for the target. There is a rootstrap for developing applications for maemo, and we'll refer to it with "maemo SDK" for the rest of the material. The maemo SDK rootstrap is also slightly special in that one normally will also run `apt-get` to install the "rest of" the SDK after extracting the base rootstrap.

3.4 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:

- 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).

3.5 Automatic install of Scratchbox

Up-to-date installation instructions can be found from maemo.org 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.

3.6 Manual install of Scratchbox

Scratchbox can also be installed manually. The Debian packages (for the real Linux system) are located at scratchbox.org. 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 scratchbox.org.

3.7 Automatic install of the maemo SDK

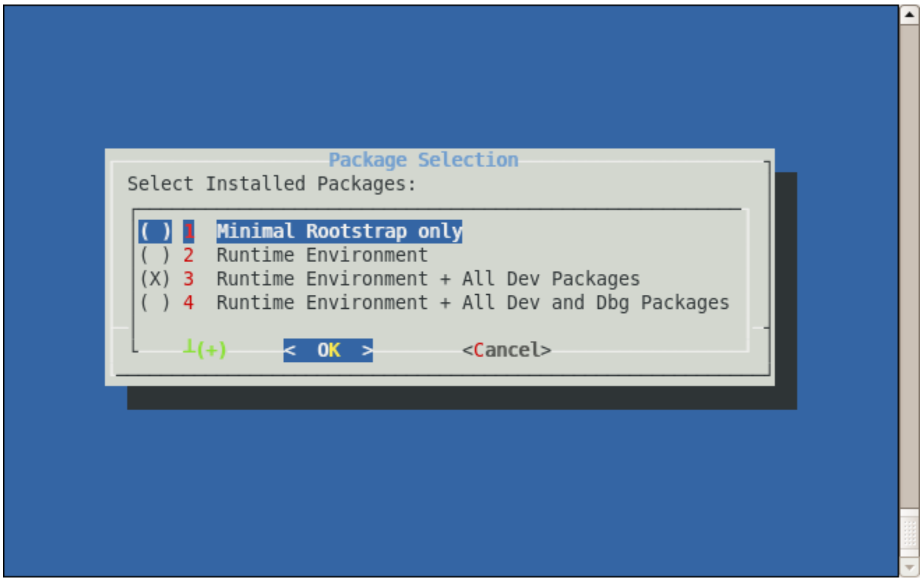
Up-to-date installation instructions can be found from maemo.org, with instructions for each maemo SDK Release.

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.



```
                     Package Selection
 Select Installed Packages:

  ( ) 1  Minimal Rootstrap only
  ( ) 2  Runtime Environment
  (X) 3  Runtime Environment + All Dev Packages
  ( ) 4  Runtime Environment + All Dev and Dbg Packages

       ↓(+)      <  OK  >       <Cancel>
```

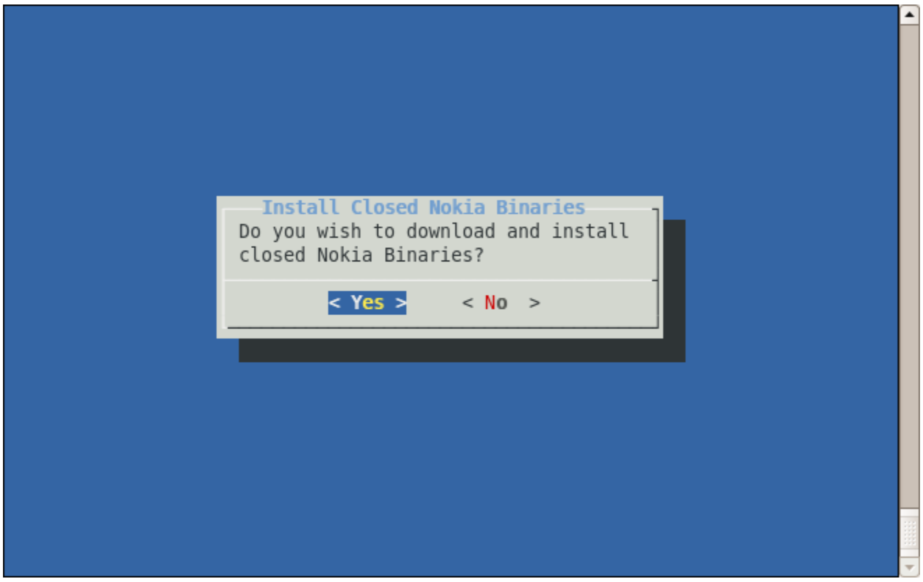
You are presented with four options of installing SDK:

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.

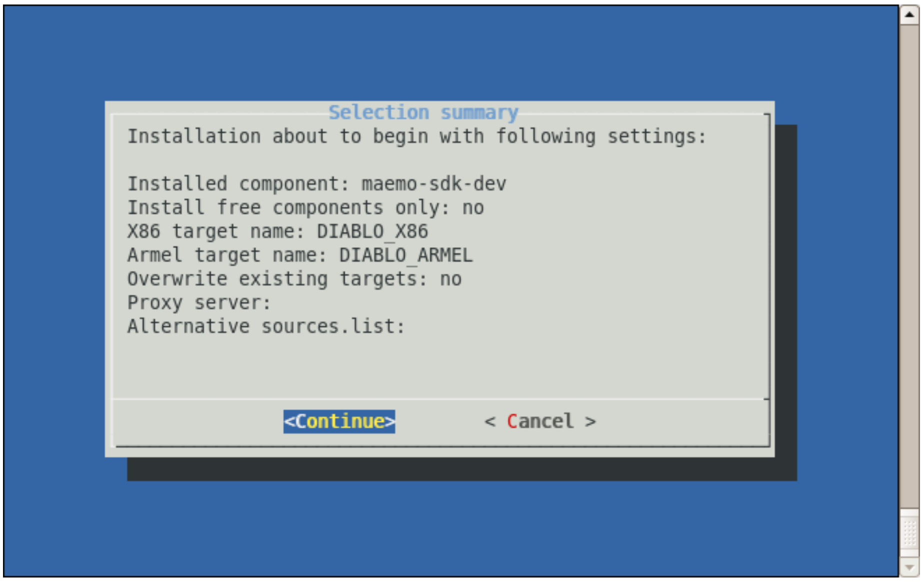
In the next dialog, you can choose to install closed Nokia binaries or not.



```
Do you wish to download and install
closed Nokia Binaries?
```

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.



```
Installation about to begin with following settings:

Installed component: maemo-sdk-dev
Install free components only: no
X86 target name: DIABLO_X86
Armel target name: DIABLO_ARMEL
Overwrite existing targets: no
Proxy server:
Alternative sources.list:
```

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
```

3.8 Manual install of the 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 maemo.org).

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

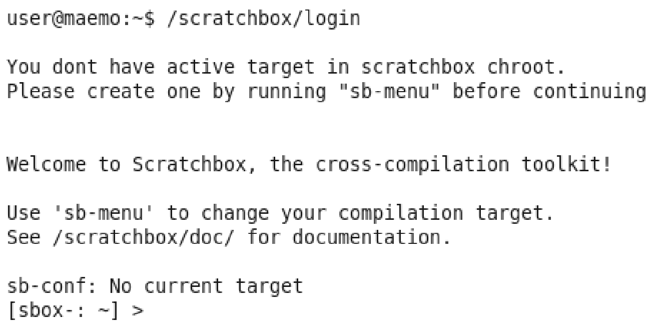
For other versions of the SDK, the exact path names above will need to be adjusted (please consult the SDK installation instructions).

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 sbox 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@maemo:~$ /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-: ~] >
```

Figure 3.1: Log-in to Scratchbox

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
```

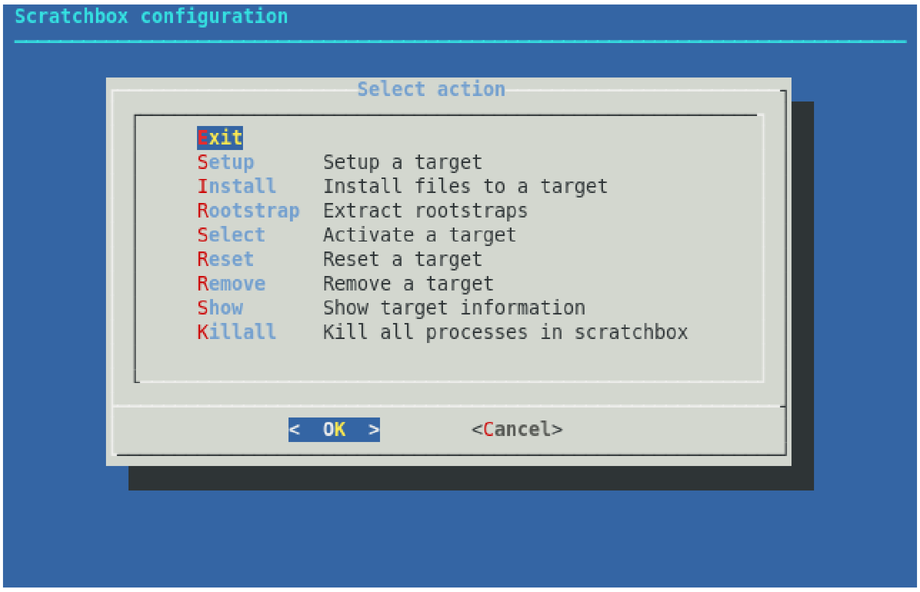
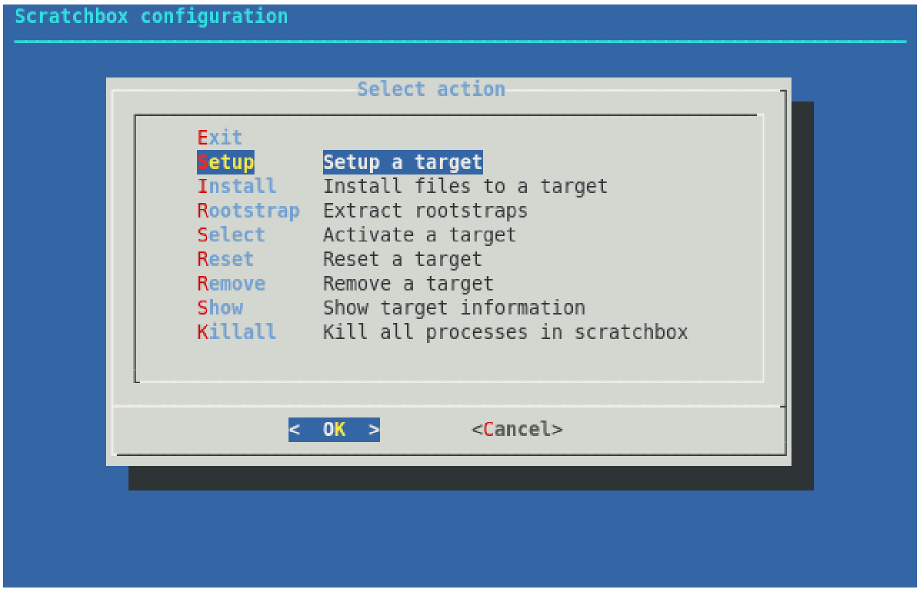


Figure 3.2: Scratchbox menu

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



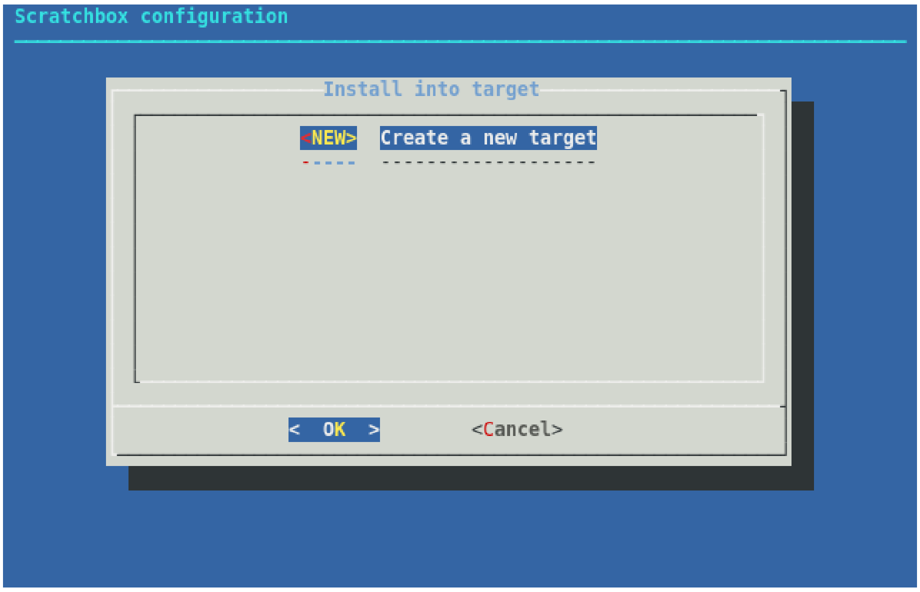
```
Scratchbox configuration

                 Select action
        Exit
        Setup      Setup a target
        Install    Install files to a target
        Rootstrap  Extract rootstraps
        Select     Activate a target
        Reset      Reset a target
        Remove     Remove a target
        Show       Show target information
        Killall    Kill all processes in scratchbox

               <  OK  >        <Cancel>
```

Figure 3.3: Setup 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.



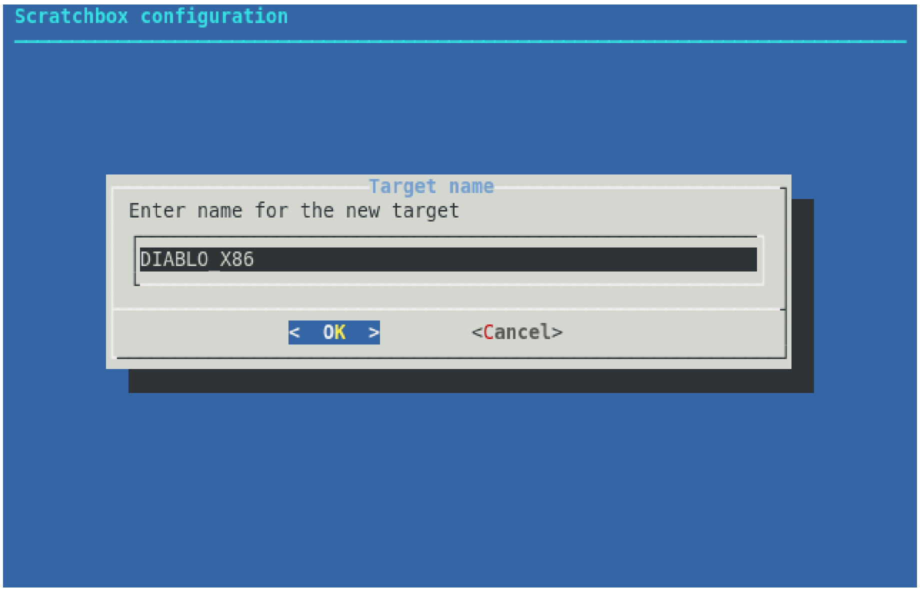
```
Scratchbox configuration

              Install into target
              <NEW>  Create a new target
              -----  -------------------

               <  OK  >        <Cancel>
```

Figure 3.4: Setup 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.



Scratchbox configuration

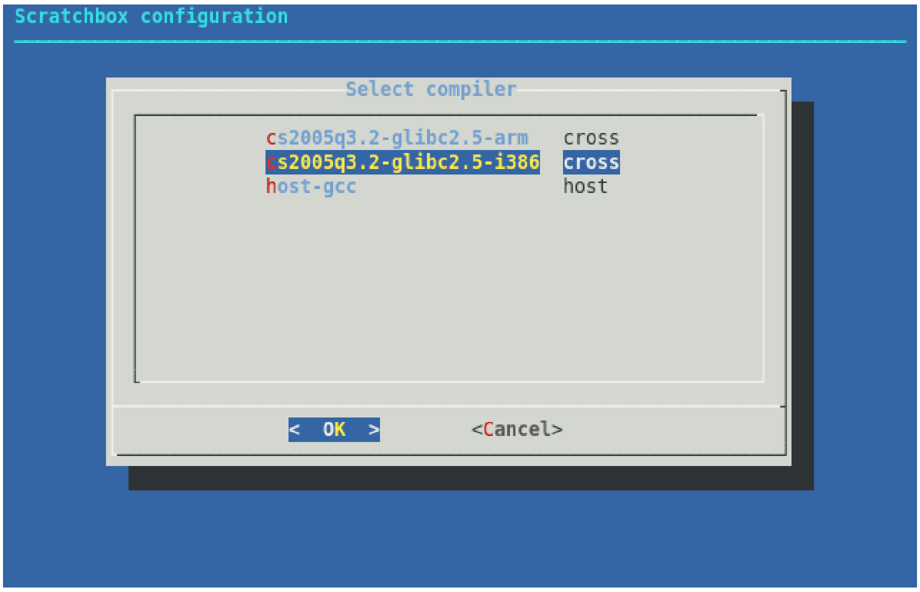
Target name

Enter name for the new target

```
DIABLO_X86
```

Figure 3.5: Name your target

4. Since the first target will be for X86 environment, select the i386 compiler version (`cs2005q3.2-glibc2.5-i386`).



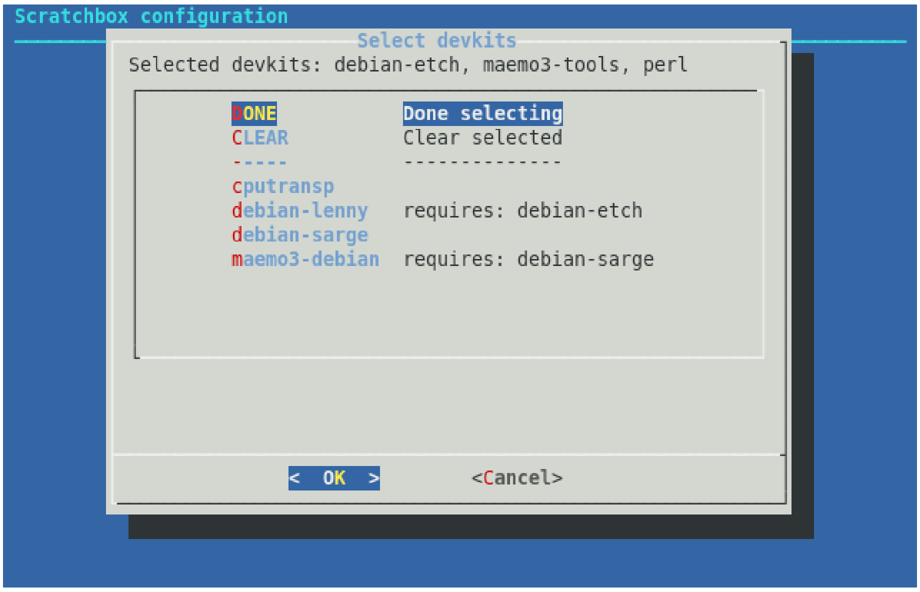
Scratchbox configuration

Select compiler

```
cs2005q3.2-glibc2.5-arm    cross
cs2005q3.2-glibc2.5-i386   cross
host-gcc                   host
```

Figure 3.6: Select i386 compiler

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".



Scratchbox configuration

Select devkits

Selected devkits: debian-etch, maemo3-tools, perl

Figure 3.7: Select devkit packages

6. Since the cputransp devkit was not selected in the previous step, selecting the CPU transparency becomes "none".

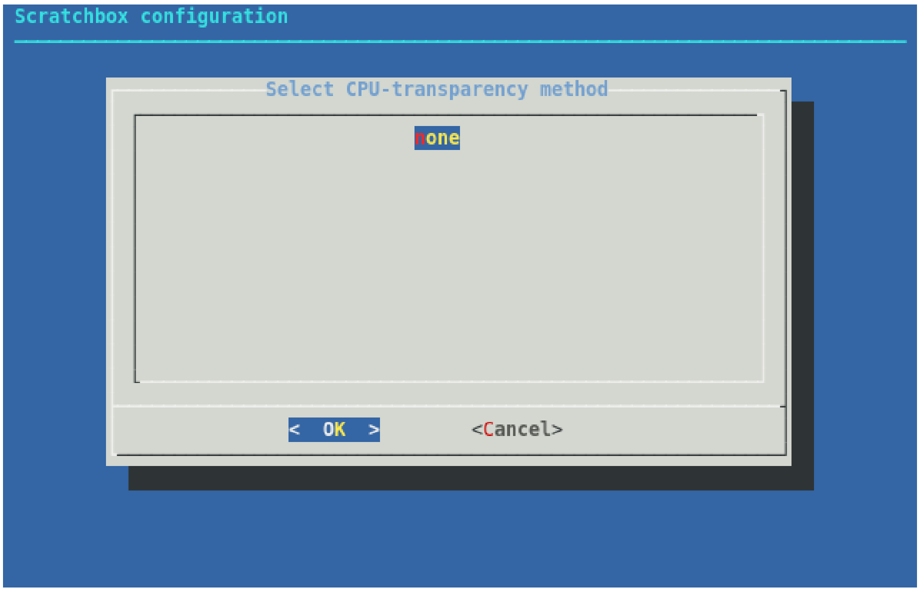
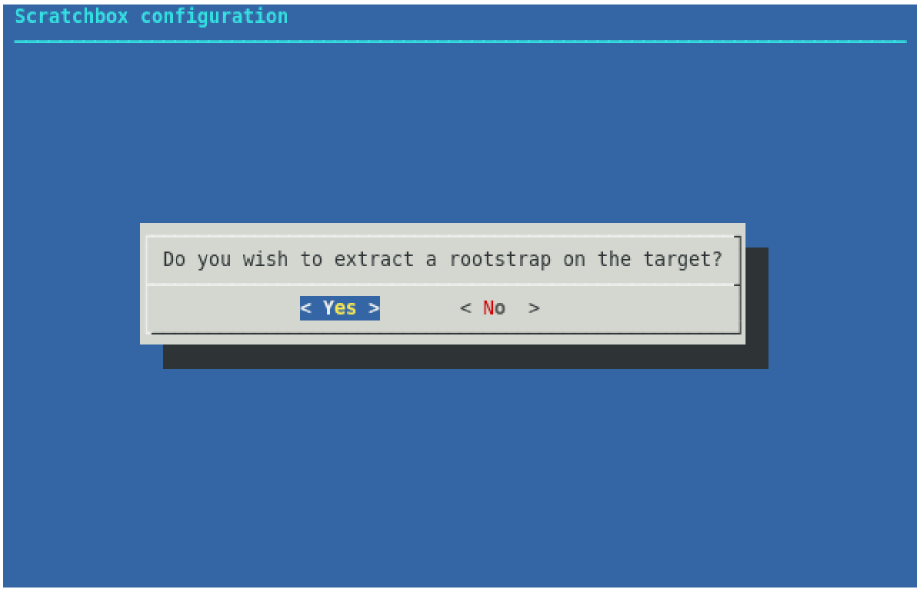


Figure 3.8: No CPU transparency available

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").



Do you wish to extract a rootstrap on the target?

Figure 3.9: Select Yes to install rootstrap package

8. And since the rootstraps were already downloaded and copied to the proper location, select "File".

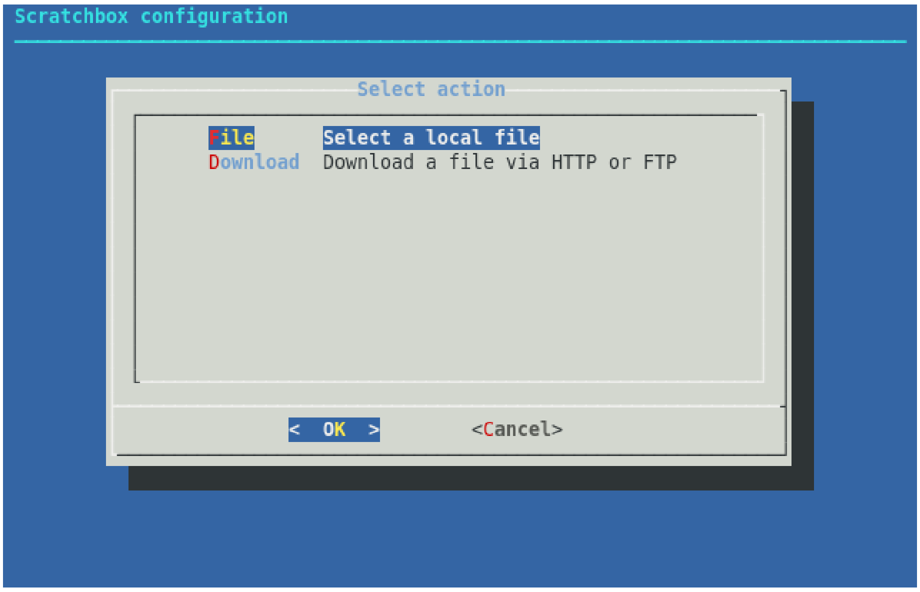
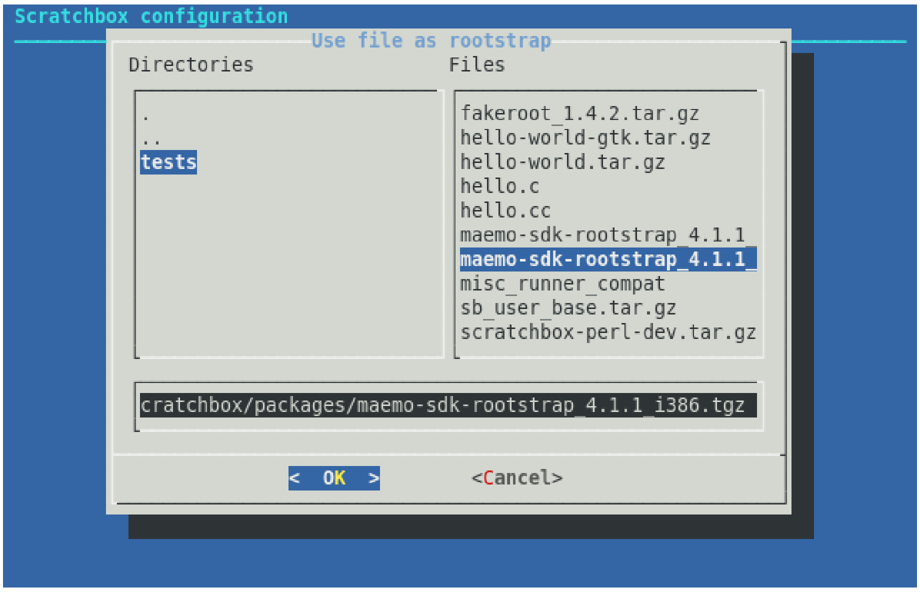


Figure 3.10: Select local file

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.

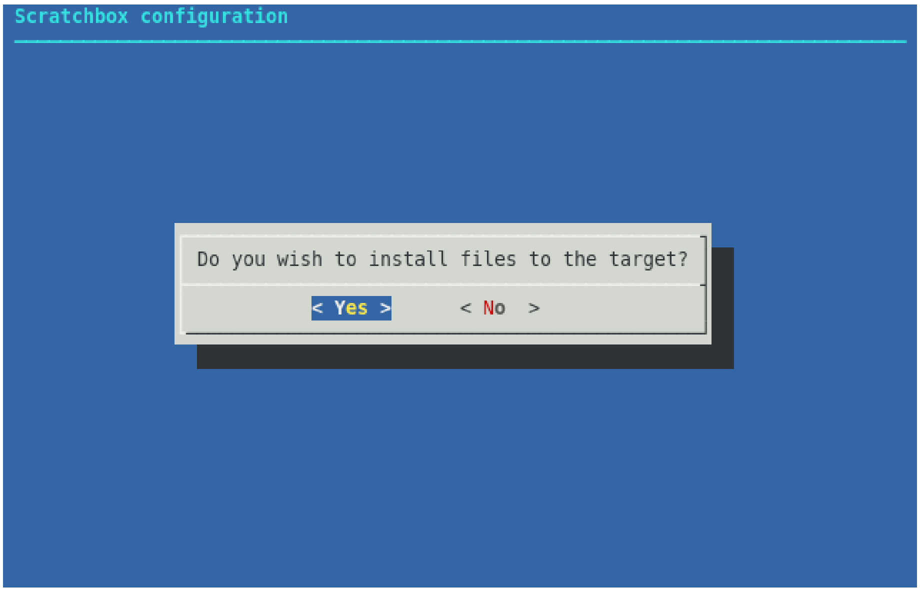


```
fakeroot_1.4.2.tar.gz
hello-world-gtk.tar.gz
hello-world.tar.gz
hello.c
hello.cc
maemo-sdk-rootstrap_4.1.1
maemo-sdk-rootstrap_4.1.1_
misc_runner_compat
sb_user_base.tar.gz
scratchbox-perl-dev.tar.gz
```

```
cratchbox/packages/maemo-sdk-rootstrap_4.1.1_i386.tgz
```

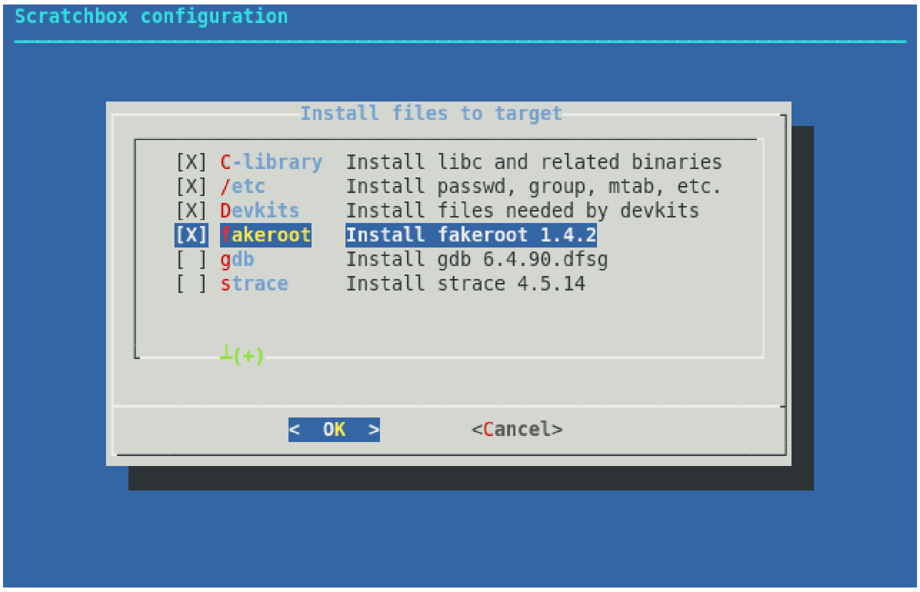
Figure 3.11: Double check for the correct architecture (i386)

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).



Do you wish to install files to the target?

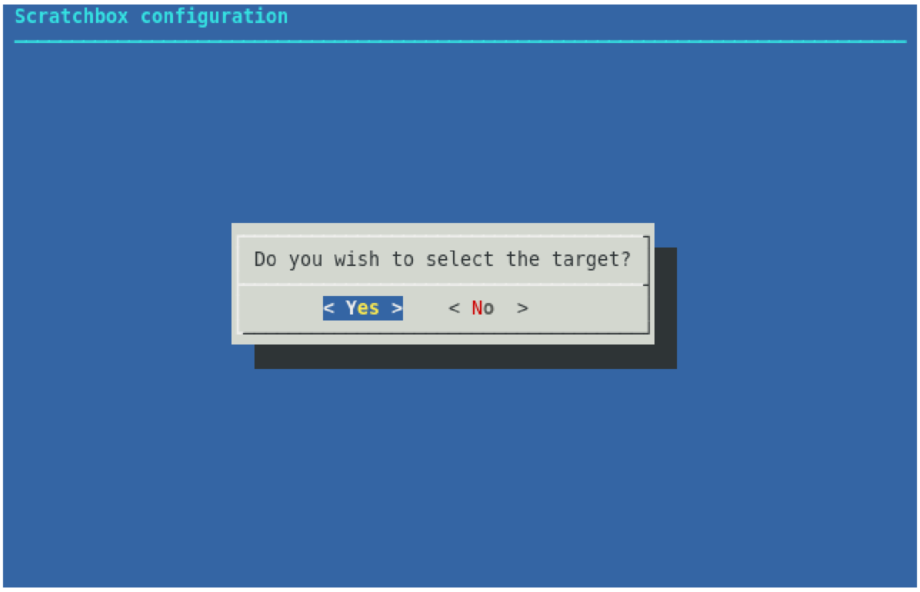
Figure 3.12: Select Yes to install files



```
[X] C-library  Install libc and related binaries
[X] /etc       Install passwd, group, mtab, etc.
[X] Devkits    Install files needed by devkits
[X] fakeroot   Install fakeroot 1.4.2
[ ] gdb        Install gdb 6.4.90.dfsg
[ ] strace     Install strace 4.5.14
```

Figure 3.13: Select the first 4 options

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).



Scratchbox configuration

Do you wish to select the target?

Figure 3.14: Select Yes to activate the just created target

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.

3.9 Manual install of the 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 ("Testing Scratchbox"), at least build the hello world program and verify the architecture of the resulting executable with file command.

Chapter 4

Testing the installation

4.1 Testing Scratchbox

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

```c
/**
 * 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 4.1: Program listing for our Hello World (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 directory should be the 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 customised. 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 favourite 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 sbox use emacsclient filename to open the file for editing in your emacs

4.2 Writing a 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 utilised. 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) {
```

```c
  /* 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 4.2: A simple GTK+ Hello World (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’
```

Compiling leads to a lot of errors

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 - Internationalised text handling
x11            X11 - X Library
```

Listing the installed `pkg-config` packages.

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

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

Listing the version of an installed `pkg-config` package, GTK+ in this case.

```
[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
```

Getting the necessary options and flags for `gcc` that allow using proper header files with GTK+

.

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
```

Getting the necessary options and flags for `gcc` that allow linking against the GTK+ library.

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
```

Trying to build again, with proper C flags.

The command above might seem somewhat strange to someone not having used UNIX command shells. What is happening here is the back-tick expansion. It is an operation where the shell will start another shell to execute just the text inside the back-ticks. 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 back-ticks 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 back-tick since it is also used for character composition (try back-tick and letter ′a′).

Something like $(pkg-config ..) might also be encountered. This is the same operation as back-tick. However, back-tick 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 back-ticks), `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: ~] >
```

Successful compile and linking.

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)
```

Victory at last!

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, whether writing simple a Hello World really is so 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.

4.3 Running the GUI Hello World

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

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

Don't despair, the end is near.

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 sbox, and clients will try to connect to the real X server (and probably fail in authentication).

Xephyr was set up 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.

4.4 Starting virtual X server (Xephyr)

Open another terminal emulator (don't close your sbox session).

Start up the server with:

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

Starting Xephyr for use with the SDK.

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 you should be extremely careful about your networking environment 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.

Figure 4.1: Only the X server running

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).

Note that the terminal emulator (more specifically, the shell that the emulator started) is waiting for Xephyr to end. If you ever need to kill all the graphical applications running in the SDK, you can just close your Xephyr. This will leave the daemons running inside sbox (D-Bus and friends). Normally this is not a good idea. To ask the foreground process to terminate itself, use `Ctrl+c`. Inside sbox this same technique can be used to terminate a graphical client that you start from the command line (as will be done shortly).

4.5 Directing the client to virtual server

Now that we have an X server running, it's time to switch back to sbox.

We start by setting the environmental variable to use a local domain socket to the X server, and tell all X clients to connect to Display number 2, since that's where we just started our Xephyr on:

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

Setting the correct `DISPLAY` content and running a simple GUI program.

We'll need to terminate the program with `Ctrl+c`, since it doesn't implement any graphical methods of closing it. Note that you'll need to set your `DISPLAY` correctly on each sbox login (or target switch).

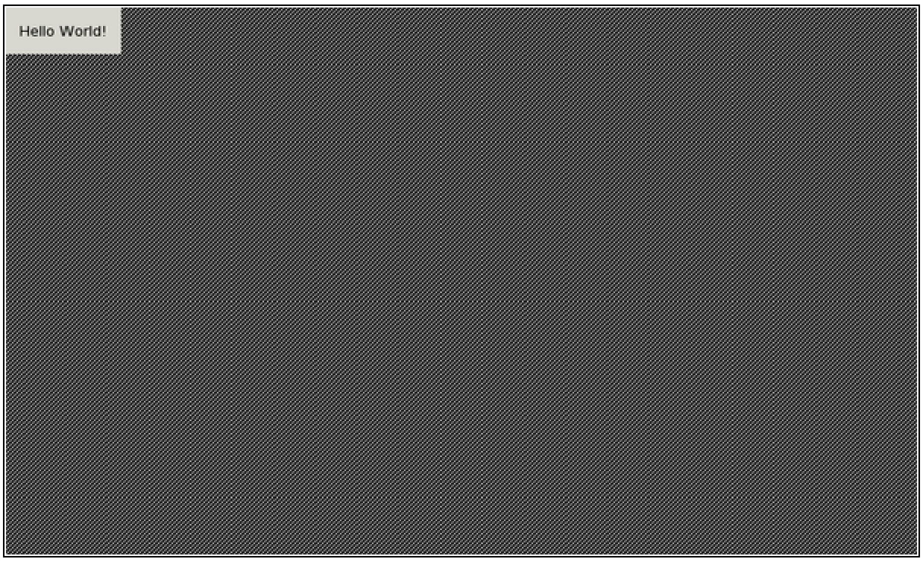


Figure 4.2: A puny little Hello World

Not really impressive, is it? If you remember what a window manager is, you will note that since it's missing (we didn't start any for the X server) you cannot control the Hello World with your mouse. Kill it with `Ctrl+c` for now (we'll start it again in a moment).

To quickly detach the foreground process from your 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".

4.6 Starting the Application Framework

The next step is to have a nice graphical environment that will implement a nicer graphical screen. We will start a series of clients each of which have a specific role. These were introduced before.

To start the Application Framework (referred to as AF from now on), we can 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: ~] >
```

Using the `af-sb-init.sh` script to start the AF.

We use `start` and `stop` parameters to start and stop the graphical environment. If everything works, you should see a screen resembling this one:

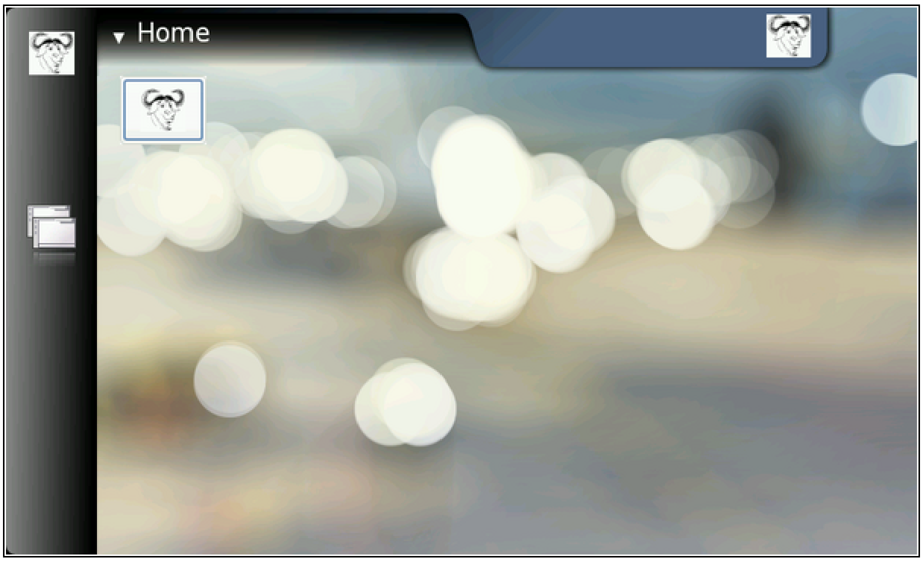
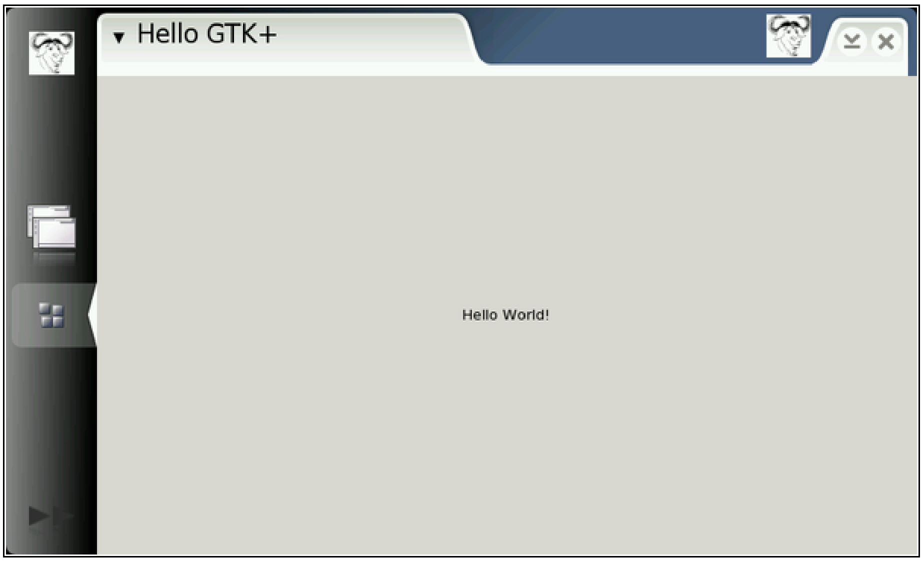


Figure 4.3: Much better

4.7 Running Hello World in the AF

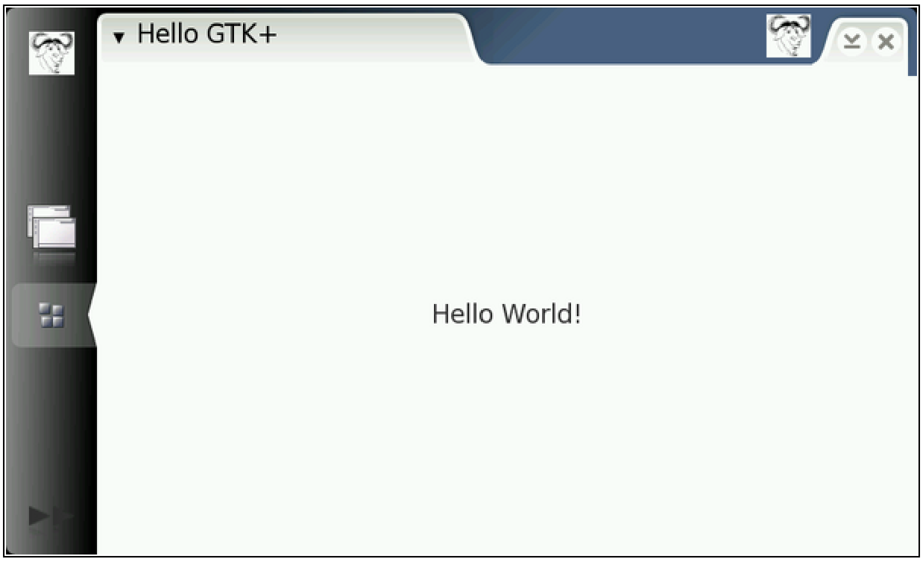
While AF is running, let's start our Hello World again (`./gtk_helloworld-1` in sbox):



Since there now is a window manager running, our client will get much larger window to draw in. GTK+ will scale the widget accordingly (only there is only one widget in our Hello World). Note that the screen still looks a bit off. If you "close the application" by pressing the X in top-right corner, you will notice that our Hello World will disappear from the screen. Since we haven't implemented signals yet and thus don't handle window destruction, our Hello World application will only be hidden. It is still running (as you can see in your sbox terminal emulator since the shell doesn't display its prompt). Stop the Hello World with `Ctrl+c`.

Next we'll use a SDK utility script called `run-standalone.sh`. Its job is to setup 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 (we don't get nice borders around our main GtkLabel widget) but looks already somewhat 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's not gray anymore).

In "maemo Application Development" material you'll learn how to adapt our application for maemo, so that it will "sit better" in the environment. You'll also learn how to react to HID-events, how to use widgets and how to package your software so that it can be easily distributed to users and installed on Internet Tablets.