交流·分享
Rockbox中文社区

Rockbox 中文开发手册

编写 | Rockbox 中文社区开发小组

入门、目录结构、函数手册，帮助您熟悉 Rockbox 源码、开始 Rockbox 开发工作。

主编 | PurlingNayuki
封面设计 | PurlingNayuki
审阅 | 罗勇

网址 | http://rbtheme.5d6d.com
群 | 34785321

目录

阅读须知

读者/用户权利

在您使用本手册之前，您应该先了解您的权利与义务。

您可以自由：

1. ***复制、发行、展览、表演、放映、广播或通过信息网络传播本手册。***

惟须遵守下列条件：

1. ***署名。****您必须按照作者或者许可人指定的方式对作品进行署名。*
2. ***非商业性使用。****您不得将本作品用于商业目的。*
3. ***相同方式共享。****如果您改变、转换本作品或者以本作品为基础进行创作，您**只能**采用与本协议相同的许可协议发布基于本作品的演绎作品。*
4. *对任何再使用或者发行，您都**必须向他人清楚地展示本作品使用的许可协议条款。***
5. *如果得到著作权人的许可，您可以不受任何这些条件的限制。*

注：本许可中的内容不损害或限制著作权所有者的道德权利。**如果您不同意以上任何一项条款，请立即删除此手册。**

您的合理使用以及其他权利不受上述规定的影响。

这是一份普通人可以理解的法律文本（许可协议全文）的概要。

引用内容

本手册的一些部分由 Rockbox Wiki 中的内容翻译而来。根据 Rockbox Wiki 使用的开源协议，本手册在预先声明的情况下引用其中的内容是合法的、不违反协议的，特此声明。

本手册的其余部分为本手册作者的原创内容，如果您需要引用其中的内容，请您先阅读读者/用户权利一节，并尽可能的预先与作者取得联系。

相关链接

您可以在本手册官方网站免费下载本手册最新版本。如果您对本手册感兴趣，或者想加入 Rockbox 开发者行列，请在本手册官方网站相关公告中申请，或者加入我们的中文交流群。

如果您发现本手册中的任何错误，欢迎您到本手册官方网站反映。一旦采纳修改，您的名字就会进入我们的文档贡献者列表，对此我们表示衷心的感谢。

本手册官方网站：

http://rbtheme.5d6d.com/bbs.php

本手册约定

本手册中的一般文字使用文泉驿微米黑字体。
本手册中需要您注意的地方使用**粗体**强调，无论是在代码中还是在叙述中。
本手册中的命令总是以$符号开头，使用 Consolas 字体列出并有浅蓝色底纹。在您输入命令之后应该按下回车键。
本手册中终端显示的信息使用*文泉驿等宽微米黑字体并且使用斜体*列出，并且有浅灰色底纹。
本手册中的 C、汇编以及其他编程语言的代码段使用 Consolas 字体列出，并有浅橙色底纹。

在本手册陈列优缺点时，为了清楚起见，本手册使用绿色字体来叙述优点，使用红色字体来叙述缺点。

准备工作

操作系统[1]

Rockbox 的开发工作可以在 GNU/Linux、Windows®、及 Mac 等主流的操作系统上完成。相对来说，在 GNU/Linux 和 Mac 上完成开发、测试工作最简单，因为这些操作系统都原生支持用命令行作为主要用户接口。而在 Windows®上工作，则需要安装 Cygwin 等 Unix 命令行模拟软件。当然，您可以使用您自己最喜欢的操作系统，但是**我们强烈推荐您使用 Linux 或者 Mac OS 来完成开发工作**,因为使用 Linux 或 Mac OS 完成 Rockbox 开发工作最便捷,相比较其他系统环境，可能出现的问题较少。

安装所需软件

无论在哪一个操作系统中进行开发工作，都需要安装一些额外的软件。

Unix/Linux

Unix/Linux 系统通常可以使用包管理系统来安装所需软件。常见的包管理系统有 apt-get、yum 及 pacman[2]，可以（**以 root 用户身份**）运行包管理器来安装：

apt-get

```
$apt-get install patch texinfo makeinfo autoconf perl
```

yum

```
$yum install patch texinfo makeinfo autoconf perl
```

pacman

```
$pacman -S patch texinfo makeinfo autoconf perl
```

Mac

Mac 系统可以安装 Mac Port 软件来模拟 Linux 环境，包的安装方法见 Mac Port 帮助文档。

Windows®

Windows 中有两个最流行的 Linux 命令行模拟器（Cygwin 与 MinGW），本手册主要介绍 Cygwin。请参见 Cygwin 一节。另外，您还可以选择安装 co-Linux 或者使用虚拟机的方法来进行开发工作。

Cygwin

您可以从 http://www.cygwin.com/下载 Cygwin 的安装程序。**如果您在中国，可以在选择源界面中添加并使用 163 源（http://mirrors.163.com/cygwin/）以加快下载速度。**

1 在本手册以后的例子中以 GNU/Linux 作为操作系统。除非有特别提示，否则在其他系统的操作方式相近。
2 为了方便起见，本手册以后都使用 apt-get 系统作为示范，请您使用您的包管理系统安装相应软件包。

选择要安装的软件包时，请您务必选择安装以下软件包以保证 Rockbox 开发工作能够正常进行：

- **Archive - zip**
- **Archive - unzip**
- **Devel - binutils**
- **Devel - gcc**
- **Devel - gcc-g++**
- **Devel - gcc-mingw-core**
- Devel - gdb (用于调试模拟器代码)
- **Devel - libiconv**
- **Devel - make**
- **Devel - mingw-runtime**
- **Devel - patchutils**
- **Devel - subversion**
- **Editors - nano (或者其他编辑器如 Vim)**
- **Interpreters - perl**
- **Net - Curl**
- Text - tetex-base (用于生成用户指南)
- Text - tetex-extra (用于生成用户指南)
- X11 - libfreetype-devel (用于转换抗锯齿字体)
- X11 - libfreetype6 (用于转换抗锯齿字体)

获取源代码[1]

Rockbox 的源代码可以免费从 Rockbox 服务器上取得。目前 Rockbox 使用 subversion 系统管理源代码，因此首先应该安装 subversion 软件：

```
$apt-get install subversion
```

安装完毕后，就可以使用 subversion 来获取源代码了：

```
$svn checkout svn://svn.rockbox.org/rockbox/trunk rockbox
```

subversion 软件会开始下载最新的源代码。一切完成之后，您就可以在 rockbox 目录中浏览 Rockbox 的源代码了。

值得注意的是，subversion 并不会自动更新，因此（如有必要的话）您应该经常同步以得到最新的代码：

```
$svn update
```

搭建编译环境

Rockbox 是播放器固件而不是 PC 软件[2]，因此编译 Rockbox 需要使用特定版本的编译工具链。使用哪一个取决于您运行 Rockbox 的设备。您可以自己配置好编译工具链与环境变量，然而使用 Rockbox 源代码附带的工具自动配置也许是更好的选择。有了 Rockbox 的源代码之后，就可以搭建 Rockbox 编译环境了。

首先进入 rockbox 源代码的 tools 文件夹：

```
$cd rockbox/tools/
```

1 在以后的叙述中，本手册假定您将源代码下载到了您的用户主目录中的 rockbox 文件夹中（即：~/rockbox/）。如果您下载到了别的文件夹，请注意替换。

2 但是可以使用 SDL 来编译模拟器，在 PC 上模拟 Rockbox 的运行。这在主题、插件调试中极为方便、实用。

然后给 rockboxdev.sh 脚本赋予执行权限并且（**以 root 用户身份**）执行它：

```
$chmod +x ./rockboxdev.sh
$./rockboxdev.sh
```

脚本会列出支持的编译链选项并且提示您输入需要的工具链：

```
Download directory : /tmp/rbdev-dl (set RBDEV_DOWNLOAD to change)
Install prefix     : /usr/local    (set RBDEV_PREFIX to change)
Build dir          : /tmp/rbdev-build (set RBDEV_BUILD to change)
Make options       :    (set MAKEFLAGS to change)

Select target arch:
s    - sh          (Archos models)
m    - m68k        (iriver h1x0/h3x0, iaudio m3/m5/x5 and mpio hd200)
e    - arm-eabi    (ipods, iriver H10, Sansa, D2, Gigabeat, etc)
a    - arm         (older ARM toolchain, deprecated)
i    - mips        (Jz4740 and ATJ-based players)
separate multiple targets with spaces
(Example: "s m a" will build sh, m68k and arm)
```

此时您可以根据您的需要来选择需要编译的工具链。

Archos 系列请输入 s 并回车；

iRiver（艾利和）h1x0、h3x0 系列、爱欧迪 m、x 系列及 MPIO（美播） HD 系列请输入 m 并回车；

iPod 系列、iRiver H10、Sansa 系列、Cowon D2 及 Gigabeat 系列等请输入 e 并回车[1]；

Onda（昂达）系列请输入 i 并回车。

也可以同时安装多个工具链。比如您需要安装 mips 及 arm-eabi 工具链，请输入

```
i e
```

并且回车。注意两个（多个）字幕之间有空格。

如果您不能确定您需要哪一个工具链，也可以选择全部安装：

```
s m e i
```

并按回车。之后该脚本就会自动下载、编译工具链，配置环境变量。最后，脚本会提示：

```
Done!
```

此时，恭喜您，编译环境已经搭建好了。

1 这一系列还可以使用 arm（即：输入 a 并回车）工具链，但这是旧的、不推荐的工具链。

Rockbox 开发基本技能

Rockbox 源代码目录结构

- **apps/**：包含了几乎所有硬件无关（即高层）代码，比如解码器、图像、菜单定义、设置定义、播放列表处理、目录遍历[1]代码以及媒体库代码。
    - **apps/bitmaps/**：与主题、WPS 无关的图像（包括 Rockbox 的启动图像以及 USB 模式图像）。
    - **apps/codecs/**:音频编码、解码器。
    - **apps/eqs/**：预置 EQ 配置文件。
    - **apps/gui/**：主 LCD 以及线控 LCD 的 GUI、WPS 代码，另外还包括类似列表、菜单等组件。
    - **apps/keymaps/**:使用新的按键动作系统包含了每个 target 的按键定义。
    - **apps/lang/**：不同语言的 UI 翻译文件。
    - **apps/menus/**：新的菜单生成代码。
    - **apps/player/**：一些 Archos Player 专用的代码。
    - **apps/plugins/**:Rockbox 的插件源代码。
    - **apps/radio/**：与收音功能有关的代码。
    - **apps/recorder/**：与录音功能有关的代码，以及一些与使用点阵 LCD 的机型有关的代码，另外还有一些 Archos Recorder 专用的代码。
- **bootloader/**：不同机型的 Bootloader 源代码。
- **docs/**:Rockbox 的文档 (有一些过期的、旧的文档，可以在 rockbox.org wiki 查看新的开发者文档).
- **firmware/**：核心代码以及关于硬件的（底层的）代码，包括了 Rockbox 内核、专用的芯片驱动、FAT 文件系统处理代码、字符处理及显示代码、库文件（Library code，部分兼容 C/POSIX API）以及 RoLo。
    - **firmware/common/**:常规（标准）C 文件以及 Rockbox 库（Rockbox libraries）（如字符串操作，文件系统访问等）。
    - **firmware/export/**:会被 Rockbox 的其他部分（如 apps/中的“userspace”）调用的公共头文件
    - **firmware/target/*[平台]*/*[芯片]*/*[设备型号]***：特定型号的驱动等代码。（目前一些型号的专用代码仍然被写在 Rockbox 的通用代码之中，整理工作还在进行中；新的专用代码仍然应该写在这里。）
- **flash/**：存放了 Rockbox 安装程序用以格式化已支持型号的 NAND[2]以安装 Rockbox 的工具代码。
- **fonts/**：随 Rockbox 分发而分发的 fnt 字体源文件（BDF 格式）。
- **gdb/**：为 Archos 的编写在 gbd 上的串口调试工具。
- **manual/**：用 TeX 格式编写的官方 Rockbox 手册。
- **rbutil/**：用 C++语言与 Qt4 库编写的跨平台 Rockbox 安装工具。
- **tools/**：包括编译文件建立工具（tools/configure）在内的各种开发及安装工具。
- **uisimulator/**:UI 模拟器允许您在 Unix/Linux 或者 Windows/Cygwin 环境内调试插件或者其他代码。
- **utils/**：各种与 Rockbox 有关但不是必要的工具。

1 原文为：directory-traversing。
2 原文为：Flash memory

- **wps/**:Rockbox 自带的 WPS 及主题。

Rockbox 编译入门

编译 Rockbox 固件

要编译 Rockbox，您需要先搭建好编译环境。如果您还没有搭建，请参考搭建编译环境一节完成。

首先进入 rockbox 源代码目录，在其中新建一个文件夹并进入其中。**文件夹的名称可以是任意合法的名字**，比如说 my-rockbox-build。在终端输入：

```
$cd ~/rockbox/
$mkdir my-rockbox-build
$cd mkdir my-rockbox-build
```

然后在文件夹中运行源代码 tools 文件夹中的 configure 脚本来生成 Makefile：

```
$../tools/configure
```

此时脚本会在终端中列出所有支持的播放器型号，并让您输入您希望编译的型号：

```
Using temporary directory /tmp
Enter target platform:
 ==Archos==                ==iriver==              ==Apple iPod==
  0) Player/Studio        10) H120/H140            20) Color/Photo
  1) Recorder             11) H320/H340            21) Nano 1G
  2) FM Recorder          12) iHP-100/110/115      22) Video
  3) Recorder v2          13) iFP-790              23) 3G
  4) Ondio SP             14) H10 20Gb             24) 4G Grayscale
  5) Ondio FM             15) H10 5/6Gb            25) Mini 1G
  6) AV300                                         26) Mini 2G
                          ==Toshiba==              27) 1G, 2G
 ==Cowon/iAudio==          40) Gigabeat F/X        28) Nano 2G
 30) X5/X5V/X5L           41) Gigabeat S           29) Classic/6G
 31) M5/M5L
 32) 7                    ==Olympus=               ==SanDisk==
 33) D2                   70) M:Robe 500           50) Sansa e200
 34) M3/M3L               71) M:Robe 100           51) Sansa e200R
                                                   52) Sansa c200
 ==Creative==             ==Philips==              53) Sansa m200
 90) Zen Vision:M 30GB    100) GoGear SA9200       54) Sansa c100
 91) Zen Vision:M 60GB    101) GoGear HDD1630/     55) Sansa Clip
 92) Zen Vision                 HDD1830            56) Sansa e200v2
                          102) GoGear HDD6330      57) Sansa m200v4
 ==Onda==                                          58) Sansa Fuze
 120) VX747               ==Meizu==                59) Sansa c200v2
 121) VX767               110) M6SL                60) Sansa Clipv2
 122) VX747+              111) M6SP                61) Sansa View
 123) VX777               112) M3                  62) Sansa Clip+
```

```
                                        63) Sansa Fuze v2
 ==Samsung==          ==Tatung==        64) Sansa Fuze+
 140) YH-820          150) Elio TPJ-1022
 141) YH-920                            ==Logik==
 142) YH-925          ==Packard Bell==  80) DAX 1GB MP3/DAB
 143) YP-S3           160) Vibe 500
                                        ==Lyre project==
==Application==       ==MPIO==          130) Lyre proto 1
 200) SDL             170) HD200        131) Mini2440
 201) Android         171) HD300
 202) Nokia N8xx
 203) Nokia N900      ==ROCKCHIP==
 204) Pandora         180) rk27xx generic
```

此时您可以输入您想运行 Rockbox 的设备的代号并回车。如果您想编译 Onda VX747，就输入前面的代号 120 并回车；如果您想编译 Cowon D2，就输入前面的代号 33 并且回车。此处作为例子我们输入 120。此时，脚本会询问您想编译什么类型的 Rockbox：

```
Platform set to ondavx747
Build (N)ormal, (A)dvanced, (S)imulator, (B)ootloader, (C)heckWPS, (D)atabase tool: (Defaults to N)
```

一般来说，我们会编译 Normal 类型。这样会得到 Rockbox 可执行文件（即：官方编译版本中的.rockbox 文件夹）。如果您需要编译 Rockbox 启动引导器，请选择 Bootloader 类型。这里我们假定您需要编译 Normal 类型，输入 N 并且回车。此时脚本会显示以下信息并提示创建 Makefile 成功：

```
Normal build selected
Using source code root directory: /home/rockbox/rockbox
Using mipsel-elf-gcc 4.1.2 (401)
Using mipsel-elf-ld 2.17
Created Makefile
```

提示信息中显示了您的源代码目录，即将使用的编译器以及连接器。提示信息会因为您的 Rockbox 源码路径、编译的型号、系统环境不同而不同，不过这没有关系。只要 configure 脚本没有提示错误就表示已经完成编译环境的配置了。

然后就可以输入 make 让系统进行自动编译工作了：

```
$make
```

如果您的处理器是多核心处理器[1]，还可以加上-j 参数加速编译过程：

```
$make -j
```

编译完成之后还要生成 Rockbox 压缩包：

```
$make zip
```

这样生成的压缩包是不附带字体的（即：官方的做法），如果您需要 Rockbox 自带的字体，可以输入：

```
$make fullzip
```

这样会把附带的字体也打包进压缩包中。之后就可以在目录中找到一个名为 rockbox.zip 的压缩包。其中的内容与官方编译版本基本相同，您也可以像安装 Rockbox 那样将压缩包中的文件夹解压到您的播放器。

此时，恭喜您，您成功的编译了您的第一个 Rockbox 版本。

1 为了方便起见，本手册以后都不加-j 参数。多核心处理器即使不加-j 参数也能够正常工作，只是速度稍慢。您可以按照实际决定是否加-j 参数。

编译 Rockbox 模拟器[1]

模拟器的工作方式

当您使用 Rockbox 模拟器时，模拟的函数只是被重定向至源代码目录的 uisimulator/目录中提供的函数。模拟器会尝试去模拟尽可能多的 Rockbox 函数，将模拟器层变得更加“底层”[2]。**因为 Rockbox 模拟器运行在 PC 机上，因此您不需要编译特定的交叉编译工具链，只需要 PC 上的 C 编译器。**

在 firmware/目录及 apps/目录中的代码就是被模拟的主要对象（最终的目标当然是让 apps/目录中的代码完全忽略它们是在真实设备上运行还是在模拟器中运行，但是目前还没有完全做到）

uisimulator/目录中的代码是用来做模拟工作的。这些代码也许永远不会包含那些在 firmware/include/目录中出现的文件。

模拟器中可用的 API

- 文件输出/输入 （open, close, read, filesize 等）
- 目录处理 （opendir, readdir 等）
- LCD （lcd_puts, lcd_update 等）
- 按键 （button_get 等）
- 插件 （类似于 Rockbox 中的可装入式插件，大部分插件的 API 都可用）

文件及目录的操作都被重定向至模拟器所在文件夹中的 simdisk 目录，这一目录就是模拟器中的根目录。LCD 的操作都反映在您的计算机屏幕上的一个窗口上，且按键应该是可用的。[3]

Linux 版本

由于 Rockbox 模拟器使用 SDL 作为底层接口，您需要先安装 SDL。在多数 Linux 上您可以通过安装 libsdl-dev 包来完成 SDL 安装：

```
$apt-get install libsdl-dev
```

然后，与编译 Rockbox 相似，建立并进入文件夹、运行脚本并选择编译目标：

```
$mkdir my-rockbox-simulator-build
$cd my-rockbox-simulator-build
$../tools/configure
```

作为例子，我们依然输入 120 表示我们选择 Onda VX747。脚本会询问您编译何种类型的 Rockbox：

```
Platform set to ondavx747
Build (N)ormal, (A)dvanced, (S)imulator, (B)ootloader, (C)heckWPS, (D)atabase tool: (Defaults to N)
```

注意，因为您是要编译模拟器，这里应该选择 Simulator。输入 s 并回车，终端中会提示：

```

```

然后输入 make 进行自动编译，并且执行 install 复制文件到正确的位置：

```
$make
$make install
```

类似的，这样并没有复制附带的字体到模拟器中相关文件夹。如果您需要附带的字体，应该输入：

---

1 原文为：Simulator，注意此处非仿真器（即：Emulator）。虽然其 API 与 Rockbox 基本相同，但是 Rockbox 模拟器使用 SDL 相关函数来替换底层驱动，并非模拟硬件的工作方式。举个例子，虽然 Rockbox 模拟器中的插件也是 rock 文件，但是其启动代码是 PC 的代码而不是您的设备的代码。

2 原文为：The simulator tries to use as many Rockbox functions as possible, putting the simulated layer as "low-level" as possible.

3 原文为：File and dir operations are redirected to use the local directory named "simdisk" as a simulated root dir. LCD operations should operate in a window on your screen and buttons should be usable.

```
$make fullinstall
```

一切顺利的话，恭喜您，您已经成功的亲手编译出了 Rockbox 模拟器。要运行模拟器，在终端中输入：

```
$./rockboxui
```

也可以使用—help 选项来查看模拟器支持的选项：

```
$./rockboxui --help
```

Windows®版本[1]

在 Linux 中编译 Windows®版本

可以在 Linux 中编译 Windows 版本的模拟器。首先您需要安装 Mingw32 交叉编译器[2]：

```
$apt-get install mingw
```

然后，下载并配置 SDL：

```
$wget http://www.libsdl.org/release/SDL-1.2.14.tar.gz
$tar xvzf SDL-1.2.14.tar.gz
$cd SDL-1.2.14
$./configure --host=i586-mingw32msvc --prefix=${HOME}/mingw32-sdl
```

您可以把—prefix 之后的路径换成任何您想放置 SDL 的路径，也可以不加—prefix 选项。但是我们推荐您加入此选项，否则新编译的 SDL（为编译 Windows®服务的包）会覆盖已经安装的 SDL（为编译 Linux 平台而服务的包）的配置。

配置完成之后，就可以编译并安装 SDL 了：

```
$make
$make install
```

最后，您需要把—prefix 选项后的路径加入 PATH：

```
export PATH=$PATH:$HOME/mingw32-sdl/bin/ >$HOME/.bashrc
```

这样 SDL 就配置完成了。编译 Windows®版本模拟器的步骤与编译 Linux 版本模拟器的步骤基本相同，脚本询问：

Platform set to ondavx747
Build (N)ormal, (A)dvanced, (S)imulator, (B)ootloader, (C)heckWPS, (D)atabase tool: (Defaults to N)

此时应该选择 Advanced。输入 a 并回车，脚本会再次询问您：

Advanced build selected

Enter your developer options (press only enter when done)
(D)EBUG, (L)ogf, Boot(c)hart, (S)imulator, (P)rofiling, (V)oice, (W)in32 crosscompile, (T)est plugins, S(m)all C lib:

您应该选择 Simulator 与 Win32 crosscompile 两项。输入 s 后回车，脚本提示：

Simulator build enabled

然后再输入 w 并回车，脚本提示：

Enabling Windows 32 cross-compiling

此时再按回车，脚本就会自动生成 Makefile 并提示成功：

然后进行自动编译与安装：

```
$make
$make install
```

这样就编译好 Windows®版本的模拟器了。

注意：**这样编译出来的模拟器体积会比较大，而且许多解码器、插件都不能正常运行。可以对模拟器使用 i586-mingw32msvc-strip 来使他们正常运行：**

1 除了编译之外，也可以直接从 http://rasher.dk/rockbox/simulator/下载非官方但可信的 Windows®预编译版本。

2 这在 Debian 中由 mingw 包提供，如果您使用别的 Linux 系统，可以查阅相关文档安装对应的包。

```
find -name "*rock"               -exec i586-mingw32msvc-strip "{}" ";"
find -name "*codec"              -exec i586-mingw32msvc-strip "{}" ";"
find -name "rockboxui.exe" -exec i586-mingw32msvc-strip "{}" ";"
```

不过这也并非没有副作用。一旦您这样做了，您就不能再用 GDB 来跟踪调试这些文件了。

另外，如果您的终端提示您：

**** Your compiler (/usr/bin/gcc) does not produce Win32 executables!*

您可以尝试重新配置 sdl 路径：

```
$CC=/usr/bin/i586-mingw32msvc-gcc CXX=/usr/bin/i586-mingw32msvc-gcc ./configure
--host=i586-mingw32msvc --prefix=${HOME}/mingw32-sdl
```

请您将—prefix 后的路径替换为您配置 SDL 时的路径。

在 Windows 中编译 Windows®版本

您可以在 Windows®系统中使用 MSYS 与 Mingw 来编译 Windows®版本的模拟器。

您需要安装好 MSYS 与以下包：

- MingW & MSYS
- SDL
- 将 MSYS 的 make 升级至 3.81
- Windows®版本的 zip 与 unzip
- perl
- mingw-crypto

目前还不能在 Windows®上编译 Windows®版本的 Rockbox 模拟器。下列网页中有一些相关的修复：
http://www.rockbox.org/wiki/UiSimulatorWindows

Mac 版本

Rockbox 模拟器可以在 Mac OS X 上编译与运行（使用 libSDL 1.2.9），但是声音输出仍然有问题。

- 安装 OSX 安装光盘上的 XCode 包
- 在 http://fink.sourceforge.net/下载并安装最新的包
- 启动 FinkCommander.app
- 使用 FinkCommander 自动安装 sdl。它会自动配置好所有的依赖关系（dl, sdl-image, sdl-image-, sdl-mixer, sdl-mixer-, sdl-shlibs, sdl-ttf, sdl-ttf-shlibs, smpeg-shlibs）
- 使用 nkCommander 来安装 svn-client
- 启动 Terminal.app，参考 Linux 环境的做法编译模拟器。

Rockbox 设计结构

本章中将会介绍 Rockbox 的底层设计以及一些 Rockbox 开发基础知识，以便您更好、更有效率的完成您的开发工作。

Rockbox 内核设计

固件与应用程序

目前 Rockbox（在逻辑结构上）被分为 Rockbox 固件（Rockbox 系统）与应用程序两个部分。**Rockbox 固件是依赖于具体硬件体系的**，而**应用程序**（至少在理论上）**只依赖于您的设备有没有某一项功能**（比如说您的设备是否能够显示彩色图像）**而不是具体硬件**。Rockbox 的一个设计目标就是尽可能的精简固件而使用应用程序以简化开发过程[1]。

未来 Rockbox 可能会被拆分为逻辑驱动代码[2]以及依赖于具体硬件体系的底层代码[3]。这样做必然要编写全新的底层驱动的 API，但是当再移植 Rockbox 到新的硬件上时，只需要重新实现这些底层代码来提供 API 供逻辑驱动代码控制硬件。

多线程

Rockbox 系统包含有联合多线程调度机制[4]。目前这种机制主要用于固件部分的代码，插件（在运行时）被包含在一个单独的线程（即：主线程）中。

固件中的一些功能被拆分到不同线程中：

- ATA 线程 （硬盘播放器）：负责控制硬盘起停转
- 背光线程：负责控制背光开关
- MMC（多媒体存储卡）线程 （闪存式播放器）：负责处理存储卡热插拔
- 音频回放线程：负责回放、录音底层处理
- 电源管理线程：负责管理电池及软充电
- 滚动线程：负责文字滚动
- USB 线程：负责处理 USB 插入、移除

这些线程中的一部分会通过事件队列相互传递信息[5]。

为什么 Rockbox 中没有 malloc()函数

最开始为难一些新的 Rockbox 开发者的事情就是 Rockbox 没有 malloc()一类的动态内存函数。动态内存分配在“正常”的程序设计中是非常常见的，这就导致一些人认为调整概念，不使用动态内存分配难以设计程序。

那么，为什么 Rockbox 中没有 malloc()函数呢？

1 原文为：One of the design goals of Rockbox is that there should be as few as possible calls from the firmware into the application itself, to ease porting of the application to new architectures.
2 或者可以成为“中层驱动代码”，即这部分代码控制硬件，但是仍然不直接访问硬件而是使用其他接口来完成硬件控制。
3 原文为：Possibly, the Rockbox firmware would further be split into logical driver code and low level bit twiddling which is architecture dependent.
4 原文为：co-operative multi-threading scheduler
5 原文为：Some of these threads pass messages to each other via means of an associated event queue.

总是使用所有可用的 RAM

硬盘起转十分耗电，因此我们想尽可能的扩大缓冲区空间来减少硬盘读写次数。也可以说，我们想尽可能的延长电池续航时间。

动态内存分配需要建立一个堆栈[1]来存储未被使用的内存，以在 malloc()要求时从中分配内存给您的代码。这样做就要求在所有的时刻设备中都有剩余的内存空间。这样音频缓存就会变小，导致硬盘读写次数增多，最后导致电池续航时间变短。

不会用尽所有内存

我们可不希望 Rockbox 总是提示“Out of memory”。我们想知道最坏的内存状况，并且确保 Rockbox 能够解决它。最繁重的音频解码、最大的播放列表、最复杂的 ID3 标签、体积最大的插件——同时出现。

在理论上，一个有动态内存分配机制的系统可以完美的解决这些，但是这十分困难并且需要许多运行时数据统计及高级的工具。而 Rockbox 使用静态内存分配，如果我们的代码占用了太大的内存，编译器可以直接拒绝编译。

调试更加简单

如果使用动态内存分配机制，您可能会在内存分配、回收及指针管理中犯下一系列错误。比如说，内存泄露问题难以追踪修正并且消耗时间。

在 Rockbox 中代码被分配到固定大小的内存，内存泄露是几乎不可能的。当然，我们还会使用指针，并且偶尔还会写指针调试代码，但是调试变得容易多了。

鼓励更加经济的程序设计

一些程序设计经验来自于 Windows®或者 Unix 编程的开发者在编写程序时并不考虑内存的使用量。在将您的代码中每一个缓冲区、每一个变量设置到最大值之前，我们建议您统计并分析到底每一个缓冲区、每一个变量真正需要多少空间。

没有虚拟内存

许多运行 Rockbox 的设备并没有 MMU[2]，因此在这些设备上不能使用虚拟内存。**而在有 MMU 的设备上，目前我们的选择是不启用 MMU** 以简化系统设计，也避免了将 Rockbox 内核分成 non-MMU 版本与 MMU 两个版本。如果没有 MMU，我们就不能正确实现 free()函数，因此所有使用 malloc()函数的代码最终都会导致内存泄露。如果 Rockbox 采用动态内存机制，为了解决内存泄露，我们必须让设备周期性的清理及重分配以清理堆栈空间[3]。这样做不但不切实际而且十分不受欢迎，因为在重分配内存期间会停止音频回放。

编译时刻[4]RAM 的层次结构

作为一个例子，下面的表格显示了一个编译好的 rockbox.elf 可执行文件在编译时刻如何分配内存到不同的段中。分段的安排会随着不同平台而变化，并且还由连接脚本决定[5]。其中最有趣的部分是音频缓存、解码缓存及插件缓存[1]。

1 原文为：heap
2 硬件内存管理单元
3 原文为：To work around this we would have to periodically free and reallocate all memory to defragment the heap space.
4 原文为：compile-time
5 在这个例子中，连接脚本是 firmware/target/arm/as3525/app.lds，编译目标为 Sansa e200v2。

大小	名称	解释
32 B	.vectors	ARM 终端向量表
460 KB	.text	可执行机器码
102 KB	.rodata	所有的只读**静态**数据
8 KB	.data	所有的数据
8 KB	.stack	用于存储已初始化线程的堆栈
384 KB	.bss	所有数据，被初始化为 0
5679 KB	audiobuf	用于储存音频与/或其他一些数据的**音频缓存**
1024 KB	codecbuf	用于存储解码器以及解码器数据的**解码缓存**
512 KB	pluginbuf	用于存储插件代码以及插件数据的**插件缓存**
7 KB	.iram	所有存储在快速的 IRAM 中的可执行机器码及数据（.icode，.irodata，.idata）
97 KB	.ibss	存储在快速的 IRAM 中的所有数据，被初始化为 0

当载入一个插件时，插件代码与数据都进入 pluginbuf 中。解码器也会类似的进入 codecbuf 中。

为代码分配内存的方法

在 Rockbox 中您不可以使用 malloc()来分配内存，您可以阅读为什么 Rockbox 中没有 malloc()函数一章来了解为什么。虽然您可能渐渐会习惯没有动态内存机制的程序设计，但是有的时候确实很需要临时分配一些缓存，或者使用别的内存区域[2]。那么应该怎么做到呢？此时您必须使用以下方法中的一种或多种来获取更大的缓存，并使用 TLSF 来分配大缓存。

所有变量

这是目前最简单的方法：

```
static char my_buffer[BUFFER_SIZE];
```

这种方法可以在任何地方使用。如果在一个插件中使用，将会由插件缓存提供空间直到该插件从缓存中卸载。类似的，解码器则由解码缓存提供空间直到该解码器从缓存中卸载。不同的，**.data、.rodata 及.bss 段的空间会被永久保留**。

特点：

- 简单、快速。
- 不能被手动释放或者调整大小。
- 必须在编译时刻指定大小。

从音频缓存（soundbuf）中分配

```
char *my_buffer = buffer_alloc(BUFFER_SIZE);
```

1 即表格中的 audiobuf, codecbuf 与 pluginbuf。

2 原文为：but sometimes you really do need to grab some buffer possibly temporarily

这种形式可以从音频缓存的前端得到一些空间。您不能在插件或者解码器中使用这种方法，因为这样得到的空间不能被释放。当空间的大小在每次 Rockbox 启动时就可以得知时，这种方式十分实用。

特点：

- 简单、快速。
- 不能被释放或者调整大小。
- 当需要更多空间时，必须停止音频回放以分配内存。

使用解码/插件缓存的剩余空间

```
size_t size_available;

//在解码器中:

void *my_buffer = ci->codec_get_buffer(&size_available);

//在插件中:

void *my_buffer = rb->plugin_get_buffer(&size_available);
```

这种方法会让解码器或插件得到相应缓存的空间。无论如何，这些空间都是为插件与解码器保留的，因此您最好尽可能的使用此方法。

特点：

- 简单、快速。
- 可用空间会随着设备而变化，并且受到限制。

使用音频缓存

```
size_t size_available;

void *my_buffer = rb->plugin_get_audio_buffer(&size_available);
```

如果您的插件设计会停止音频回放，您可以使用这种方法得到更大的空间。

- 可用空间相对比较大。
- 在您取得缓存空间时必须停止音频回放。

使用 buflib

BuflibMemoryManagement 是一项使内存分配变得动态化的项目[1]。该项目在 2011 年 5 月刚刚被发起。

- 自由、灵活。
- 当 buflib 被使用时或其他线程在运行时，分配到的内存位置可能会变化。您必须持续跟踪目前得到的内存位置。
- Other aspects of the project have not yet been determined.

[1] 原文为：BuflibMemoryManagement is a project to allocate memorymore dynamically.

Rockbox 内存布局[1]

软件解码（SWCODEC）

用途	起始指针	大小
字体缓冲区	mbuf	MAX_FONT_SIZE
目录缓存	audiobuf (初始化)	varies
曲目数据库		varies
曲目库搜索		varies
存储（当前）播放列表		configurable
存储目录树		可以手动配置
LastFM 记录缓存	scrobbler_cache	SCROBBLER_MAX_CACHE * SCROBBLER_CACHE_LEN（可选的）
Cue 曲目表缓存		2 * sizeof(struct cuesheet)（可选的）
语音预存储缓存 (.talk 语音剪辑文件)		MAX_THUMBNAIL_BUFSIZE
语音文件	audiobuf （在初始化完成之后可用）	talk_get_bufsize()
解码器 malloc 缓存	&audiobuf[talk_get_bufsize()]	MALLOC_BUFSIZE
已压缩的解码器数据（以及保护缓存[2]）	filebuf = &audiobuf[talk_get_bufsize()+MALLOC_BUFSIZE]	as big as possible
音频解码器的 IRAM	filebuf + filebuflen	CODEC_IRAM_SIZE
语音解码器的 IRAM		CODEC_IRAM_SIZE (if talk_voice_required())
音频解码器的 DRAM		CODEC_SIZE
语音解码器的 DRAM		CODEC_SIZE (if talk_voice_required())
PCM 缓存、关键帧	audiobufend - (pcmbuf_size + pcmbufdesc_size +	pcmbuf_size + pcmbufdesc_size

[1] 本节的信息刚刚开始收集，可能会有错误或者不完整。如果您了解该方面，请联系作者或者官方修正问题。

[2] 原文为：guard buffer

用途	起始指针	大小
	PCMBUF_MIX_CHUNK*2)	
PCM 淡入淡出缓存	audiobufend - PCMBUF_MIX_CHUNK*2	PCMBUF_MIX_CHUNK
PCM 语音混合缓存	audiobufend - PCMBUF_MIX_CHUNK	PCMBUF_MIX_CHUNK
(结束)	audiobufend	

MASCODEC

（等待官方文档完善，如果您想帮助完善文档，请联系作者或者向官方提交文档）

Rockbox 插件中的内存布局

在一个 Rockbox 插件中包含下列段：

.header	插件头部结构体（即：插件头部，参见 apps/plugin.h）
.text	可执行代码
.rodata	已初始化的只读数据（比如字符串、常数以及常数结构体）
.data	已初始化的数据（比如已分配的全局变量[1]）
.bss	未初始化的数据（比如未分配的全局变量）

同时，其中还有一些最终会被载入到 IRAM 中的特殊段：

.iram	由.icode（IRAM 中的可执行代码）、.irodata（IRAM 中已初始化的只读数据）及.idata（IRAM 中已初始化的数据）组成
.ibss	IRAM 中未初始化的数据

IRAM 是一块（有固定地址的）能快速的被 CPU 访问的特殊内存区域。如果需要经常读取或修改一个变量，应该把这个变量放到 IRAM 段中。类似的，如果需要经常调用一个函数，可以将该函数放到 IRAM 段中。可以在变量或者函数的声明后加入 IRAM 属性来做到这一点：

```
static struct dsp_config *dsp IDATA_ATTR = audio_dsp; //变量

static void pcmbuf_callback(unsigned char** start, size_t* size) ICODE_ATTR; //函数声明
```

如果这样做，以下符号将在连接期间[2]被声明（参见 plugins/plugin.lds）：

plugin_start_addr	.header 段的起始地址。因为.header 段是一个插件的第一段，因此该地址也是插件的地址。
iramcopy	定义了插件的.iram 段将被载入内存的地址。This is immediately after the end of the .data section
iramstart	插件.iram 段的起始地址（偏移地址）
iramend	插件.iram 段的终止地址（偏移地址）。iramend-iramstart 将得到插件的只读数据的大小（包括代码以及变量）

[1] 原文为：global variables with assignment
[2] 原文为：linkage

iedata	插件.ibss 段的起始地址（偏移地址）
iend	插件.ibss 段的终止地址（偏移地址）。 iend-iedata 可以得到插件.ibss 段的大小
plugin_bss_start	插件.bss 的起始地址
plugin_end_addr	插件.bss 段的终止地址。plugin_end_addr- plugin_bss_start 可以得到插件.bss 段的大小。因为这是插件中最后一个段，因此当插件被载入到 RAM 中时此地址也是插件终止地址

.iram 段的偏移地址[1]就是 IRAM 在 RAM 中的起始地址。然而，当插件被载入时它的.iram 段会被载入到另一个地址（即 iramcopy）。

上表中所提到的段在内存中的放置方式如下表。这也是当插件被载入到 RAM 中后该内存区域[2]的样子。

.header	← plugin_start_addr
.text	
.rodata	
.data	
.iram	← iramcopy
.ibss	
.bss	← plugin_bss_start ← plugin_bss_end

如果插件要使用 IRAM（存储代码或者数据），最开始的一步应该是在代码中调用 PLUGIN_IRAM_INIT(api)。这样会把插件的.iram 段（.iramcopy 指向此段）复制到 IRAM 中，并清理包含插件的.iram 及.ibss 段的内存区域。在完成调用之后，iramcopy（大小为 iramend-iramstart）及 iedata（大小为 iend-iedata）数组可以被此插件任意使用。如果插件不使用 IRAM，则 iramend 与 iramstart 及 iend 与 iedata 将会相等[3]。

完成调用 PLUGIN_IRAM_INIT(api)之后，RAM 看起来会像是下表的样子：

更低位的地址[4]
IRAM
插件.iram 段中的内容
未初始化区域[5]，大小为插件.ibss 段的大小
…（IRAM 的剩余部分）
“普通”内存
…（Rockbox 可执行代码等）
.header （插件从这里开始）
.text
.rodata
.data
被清空的区域，大小为插件.iram 段的大小（在调用之前包含.iram 段的数据，完成调用之后可以任意使用）

[1] 原文为：relocation address

[2] 该内存区实际上就是 pluginbuf 缓冲区，参见 apps/plugin.c。

[3] 即：将两个地址相减会得到 0。

[4] 原文为：Lower addresses。

[5] 原文为：Not initialised area，此处应指内存中的其他部分。

被清空的区域，大小为插件.ibss 段的大小（在调用之前包含.ibss 段的数据，完成调用之后可以任意使用）
.bss
…（RAM 的剩余部分）

调用 PLUGIN_IRAM_INIT(api)的动作是由插件而不是插件头部来完成的。如果一个插件要使用 IRAM，音频回放就必须停止[1]。因此如果由插件头部来完成调用，则音频回放就一定会被停止；而如果由插件自身来完成调用，则插件可以有机会来询问用户是否允许停止音频回放并启动插件。然而，**这将要求开发者足够小心，不会忘记在他的代码中调用 PLUGIN_IRAM_INIT(api)**。

[1] 音频回放也使用 IRAM，因此必须停止音频回放以重新分配 IRAM。

开始 Rockbox 开发

本章中，您将开始真正的 Rockbox 开发。您将会了解 Rockbox 内核、创建线程、USB 处理、创建列表、创建菜单等等基本开发应用。阅读本章前，建议您仔细阅读前一章以了解 Rockbox 底层结构。

列表控件（List Widget）

列表控件介绍

虽然列表可以在许多场合（比如说文件浏览、媒体库、菜单）中使用，但是其最常用的场合还是实现浏览文本、显示文件属性等等。如果您需要一段有效率、通用的、稳定的代码来帮助您显示一个列表，您可以在 apps/gui/list.c 中找到这段代码。它可以帮助您快速的创建一个列表。

当您听到“列表”这个词时，您可能会想到类似于链表一类的标准数据存储结构。然而，**列表控件并不是被设计来存储数据的。它是用来显示某种格式的列表中的数据的，并且可以用来处理如文字滚动、选择列表中的项目等用户交互**。

您可能会问：“如果列表控件不能存储数据，那么数据都存储在哪儿，又是怎样被存储的呢？”

如何存储、存储的位置完全取决于您的选择，这就是列表控件最吸引人的特性。数据一般都存储在链表中，但也可以在一个数组中或者文件中，这完全由您来决定。让列表正常工作的唯一条件就是您需要通过它的索引来引用正确的数据[1]。

您可能还会问：“我已经知道列表控件自身不存储数据，那么它又怎么知道应该显示什么呢？”

控件本身不会保存数据，因此它需要一个到数据的链接。在 Rockbox 的列表控件实现中，这个链接是一个指向函数的指针。**为了让列表控件显示列表中的项目，您需要设计一个函数。当给这个函数传递一个项目的索引编号作为参数时，函数返回这个项目的内容**[2]。

除了允许您简捷的列出列表之外，列表控件还包括以下实用的特性：

- 可选的滚动条，便于用户得知总的项目数以及目前所在的位置
- 可以在列表到达顶部时返回底部，反之亦然
- 如果一个项目的文字太长以至于不能完全显示，可以使其滚动显示
- 显示图标来提示项目的类型

如何使用列表控件

定义数据结构

现在，我们假设有一些数据存储在链表中。链表中的每一个元素的结构如下：

```
struct names_list {
       char first_name[10];
       char last_name[10];
       struct names_list *next;
}
```

然后我们再定义一个链表头指针指向链表的第一项，然后定义一个 curr 指针表示目前指向的项目：

```
struct names_list *people, *curr;
```

[1] 原文为：All that is needed for the list to work is that you're able to reference the appropriate item in the list by its index.

[2] 原文为：For the widget to show the labels of the items in the list, you implement a function that will provide the label when the index of the item to be shown is handed over to it (as well as a couple other parameters I'll discuss later).

这样我们就可以通过改变 curr 指针来找到下一项：

```
curr = people->next;
```

声明列表

为了使用列表，我们必须在代码中包含 plugin.h 头文件。该文件包含了从数据结构到函数指针的所有代码。然后，我们先声明一个列表控件结构：

```
#include "plugin.h"
struct gui_synclist gui_people;
```

初始化列表控件

因为控件本身不存储数据，所以必须使用函数创建数据链接。使用函数创建数据链接的方法很多。但是无论如何，以下代码提供了一种可能的方法：

```
char * get_person_label(int selected_item, void *data, char *buffer, size_t buffer_len){
    int i;
    struct names_list *temp_node = (struct names_list *)data;

    for (i=0;i<selected_item && temp_node != NULL;i++){
        temp_node = temp_node->next;
    }
    if (temp_node == NULL){
        return NULL;
    }
    rb->snprintf(buffer, buffer_len,"%s, %s", temp_node->last_name, temp_node->first_name);

    return buffer;
}
```

在代码中您可以看到：

1、该函数接受一个指向某些数据的指针。虽然**这是可选的**，但是您可以通过在初始化列表时这样做来为您的数据结构提供那一个指针[1]；

2、列表控件通过传递 selected_item 参数来通知该函数它需要的项目。如何通过该参数处理、返回数据由您来设计。

3、该函数将返回的数据通过 snprintf 写入 buffer 指向的缓存中，然后将 buffer 的指针返回给列表控件。**您可以任意命名缓存，但是请保证您返回的指针是您写入数据的缓存**。

完成这些之后就可以开始真正的列表初始化过程了：

```
rb->gui_synclist_init(&gui_people, &get_person_label, people, false, 1);
```

这些参数都代表了什么？联系一下我们之前定义过的结构体和函数：

1、**&gui_people**：指向列表（注意不是链表）的指针。回想一下，我们把 gui_people 声明为一个变量而不是指针，因此需要做取地址运算；

2、**&get_person_label**：指向能够根据特定参数提供数据的函数的指针，这里我们把它指向之前定义的 get_person_label 函数；

3、**people**：该可选参数允许我们把链表的入口传递给 get_person_label 函数；

4、**false**：如果将该选项设置成 true，则整个列表里所有不能完全显示的项都会滚动。如果设置为 false，则只有选中的项目可以滚动。

[1] 原文为：While this is optional, you can provide that pointer to your data structures when initializing the list.

5、**1**：该参数决定用户一次能在列表中选择多少行。举例来说，如果该参数设置为 3，则用户点击一次，三行内容同时会被选中。当您需要把一个项目拆分到多行以便显示详情时，该功能十分实用。当**然，您也需要自行处理多行选项的显示及用户交互**。比如说，如果您将该参数设置为 3，您可能需要使用 selected_index / 3 来获取实际的项目编号，并且使用 selected_index % 3 来得知您正在显示一个项目的哪一部分[1]。

设置列表属性

现在列表已经被初始化了，但是可能需要设置一些属性使列表成为您希望中的样子。

列表标题

```
rb->gui_synclist_set_title(&gui_people, "Contacts", NOICON);
```

这里我们直接传递字符串作为列表标题。您也可以使用字符数组来作为标题。最后一个参数（从枚举变量 themable_icons 中）指出列表图标。

图标返回值函数

```
rb->gui_synclist_set_icon_callback(&gui_people, &get_person_icon);
```

&get_person_icon 同样是指向函数的指针，并且该函数的工作方式与 get_person_label 函数相似，只是它的返回值是您希望显示在某个项旁的图标编号。**图标编号存储在 themable_icons 枚举变量中**。

列表项数目

```
rb->gui_synclist_set_nb_items(&gui_people, people_count);
```

回忆一下，列表控件本身是否不存储数据？因此，您需要让它知道项目的数目，以便于控件查找、显示。

滚动上限

```
rb->gui_synclist_limit_scroll(&gui_people, true);
```

如果该项设置为 false，则用户将列表滚动到最顶部时会回到最底部，反之亦然。如果设置为 true 则不允许列表回环滚动。

默认选择项

```
rb->gui_synclist_select_item(&gui_people, 0);
```

这样做可以让列表得知默认的选项。当然，您也可以在别的代码中使用它，但是您仍然需要让列表控件在一开始就得知默认选项。

处理用户交互行为

设置完属性后，我们就可以将列表显示在屏幕上，并且开始处理用户交互了。我们让代码处于一个无限循环之中，等待用户操作并且响应。但是有一些人反对这样做，他们更倾向于使用 switch…case 语句并且设置一个标志。无论如何，我们至少拥有可以正常工作的代码：

```
int curr_selection  = 0; /* 并非一定需要此声明*/
    int button;
    while (true) {
        rb->gui_syncstatusbar_draw(rb->statusbars,  true); /* 绘制状态栏，应该时常执行该函数  */
        button = rb->get_action(CONTEXT_LIST,TIMEOUT_BLOCK); /* user input */
```

[1] 原文为：set to show 3 lines per item, divide the selected_index by 3 to know the actual item number and calculate selected_index % 3 to know the part of the item you're showing.

```
        if (rb->gui_synclist_do_button(&gui_people, &button, LIST_WRAP_UNLESS_HELD)) { /* 自动处理用户输入事件。 _UNLESS_HELD 也可以是_ON 或者_OFF */
            continue;
        }
        switch (button) { /* 处理用户输入 */
            case ACTION_STD_OK:
                return rb->gui_synclist_get_sel_pos(&gui_people);
            break;
            case ACTION_STD_CANCEL:
                return NULL;
            break;
        }
    }
```

为了使代码正常工作，我们让代码一直等待用户的操作并且将操作事件传递给控件。然后，我们可以让列表控件去处理用户的操作以完成交互功能。

关于以上代码的一些说明：

1、**强制绘制状态栏**。在代码中，每经过一次循环就绘制一次状态栏，以此保证状态栏始终显示。因为状态栏是不用于列表控件的另一个控件，如果不强制绘制状态栏，那么状态栏可能不显示，取而代之的是一行空白；

2、**绘制列表**。在绘制的时候，会根据目前列表的位置调用您之前定义的函数取得相应的数据及图标，并且绘制在正确的位置。**请您只在退出、重新进入或者改变选项时才重新绘制列表否则所有的文字都不会滚动**。

3、**让列表控件来处理用户动作**。函数仅返回一个代表用户动作的值，如果没有读取到用户动作则返回 0。这样的实现给了您选择是否继续处理动作的自由。当然每一种情况都会被以不同方式处理[1]。

4、**最后您自行编写代码响应不同的用户动作**[2]。在此处，我们使用控件函数来读取目前选择的列表项位置并且返回该值。

LIST_WRAP_UNLESS_选项

选项	选项说明
LIST_WRAP_UNLESS_HELD	Don't let the list wrap if the user is holding down up/down, so should be used everywhere except special cases.
LIST_WRAP_ON	Always allow the list to wrap... not very nice. Shouldn't be used.
LIST_WRAP_OFF	Never allow the list to wrap... So it should only be used by the volume setting.

更多有关列表控件的信息

本手册中介绍了一些关于列表控件的基础用法。事实上，列表控件的功能十分强大，举例说，有相关的函数可以添加、删除项，您可以在 apps/gui/list.h 文件中查看更多关于列表控件的函数。

[1] 原文为：This lets us decide if we want to continue handling the action or not. Of course every situation can be handled differently.

[2] 原文为：Further handle the action ourselves.

关于图标，您可以在 Rockbox 插件以及其核心代码中找到很多例子（甚至连 WPS 中都包含一些相关的函数），此处不再赘述。

Rockbox 内核

概述

Rockbox 内核是一个抢占式多优先级联合内核，由许多能够相互传递消息以及锁定数据结构的线程构成[1]。其工作方式如下图所示。

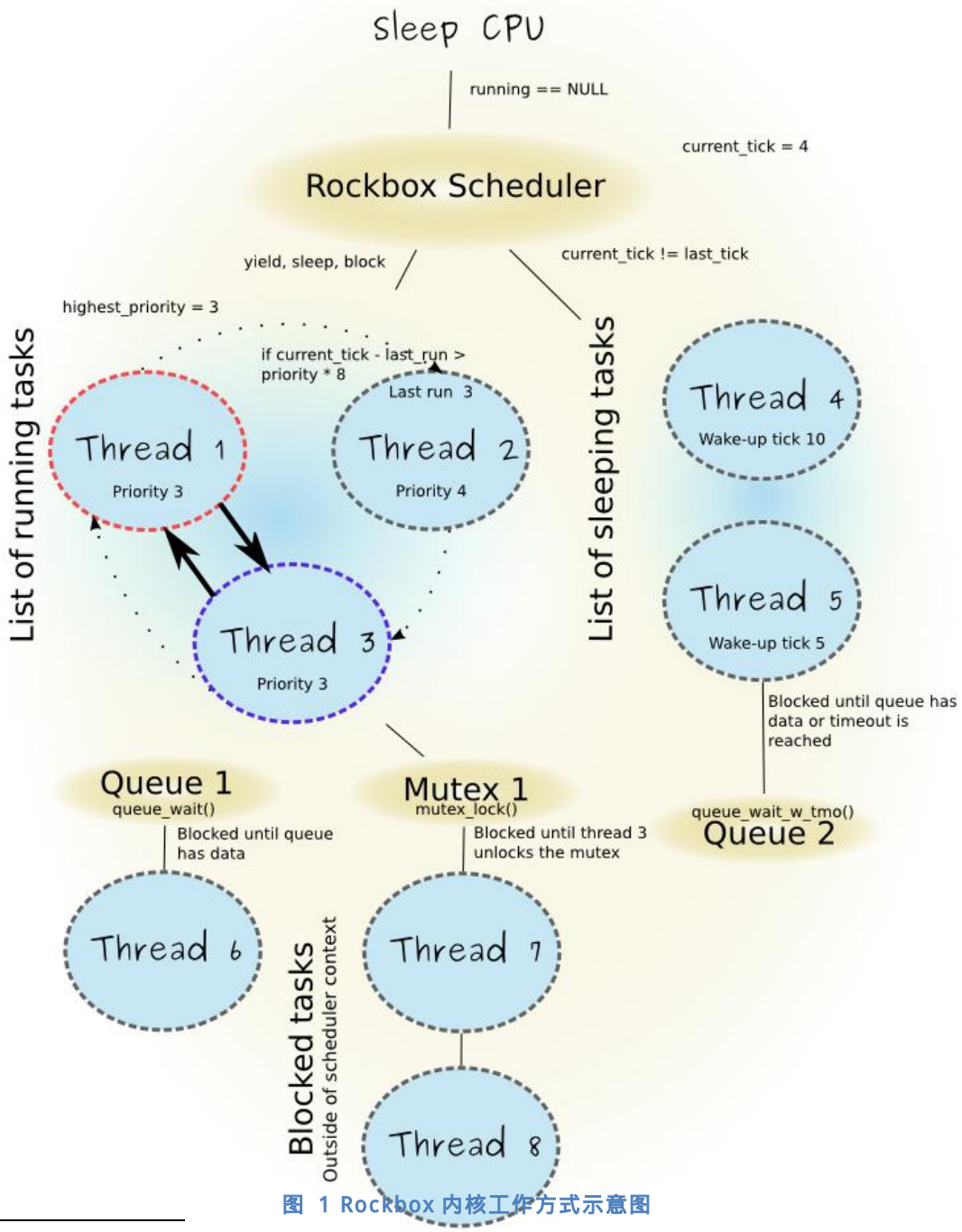


图 1 Rockbox 内核工作方式示意图

[1] 原文为：The Rockbox kernel is an advanced priority aware cooperative kernel. It consists of thin threads with some means of passing messages and locking data structures.

线程

概述

一个线程就是一个从不返回值的函数[1]。它拥有自己的堆栈，并且直到它允许内核执行其他线程之前，它一直是活跃的（被执行的）。各个线程按照被创建的顺序轮流执行。

与内核有关的函数都声明在 thread.h 及 kernel.h 头文件中。

内核转去执行其他线程的动作叫做**环境切换**[2]。您可以使用 yield()函数来强制进行环境切换，因此，环境切换常被称为**休眠**[3]；可能引起休眠的线程称为**中断线程**[4]。

线程环境[5]

线程环境由被标准 C 调用保护的 CPU 寄存器以及堆栈指针组成，但不包括临时寄存器[6]。同时，类似于 MAC 累加器等的附加寄存器也不会被保护[7]。

线程函数

unsigned int create_thread(void (function)(void), void stack, size_t stack_size, unsigned flags, const char *name IF_PRIO(, int priority) IF_COP(, unsigned int core))

该函数用于创建一个线程。该线程会被立即添加到线程队列中并准备被执行。一旦一个返回一个值，它会被立即从线程队列中移除。您也可以在代码的其他部分使用 remove_thread()函数移除某个线程。

priority 代表着该线程的优先级，**在 thread.h 文件中有定义**。可能的优先级列表如下（优先级由高至低）：

1. *PRIORITY_REALTIME*
2. *PRIORITY_PLAYBACK_MAX*
3. *PRIORITY_BUFFERING*
4. *PRIORITY_USER_INTERFACE*
5. *PRIORITY_RECORDING*
6. *PRIORITY_PLAYBACK*
7. *PRIORITY_SYSTEM*
8. *PRIORITY_BACKGROUND*

core 代表着将会运行该线程的核心。您可以设定在拥有多核心处理器的设备（如 iPod，iRiver H10，Sansa e200 等）上应该由哪一个核心运行线程。在多数情况下，这个值应该设定为 CPU，但是在某些特殊情况下可能需要设定为 COP[8]。

flags 也是线程的属性。该标志可能的值是 CREATE_THREAD_FROZEN。这样做会在创建完线程后冻结它，除非您使用 thread_thaw()函数，否则该线程不会被运行。

create_thread()函数会返回新线程的 ID，可以其传递给其他线程函数以便引用新线程[9]。

---

[1] 原文为：A thread is simply a normal C function that never returns.

[2] 原文为：context switch。

[3] 原文为：yielding。

[4] 原文为：A function that may cause a context switch is often called a blocking function, or a function that blocks.

[5] 原文为：The thread context。此处保持与前文相同的译法，译作“环境”。

[6] 原文为：The thread context consists of the CPU registers that are normally preserved in normal C calls, plus the stack pointer. The "scratch registers" are not included.

[7] 原文为：Also, no extra registers, such as MAC accumulators or status registers are preserved.

[8] 即：coprocessor，协同处理器。

[9] 原文为：A handle id is returned, which should be passed to other thread functions for referencing the created thread.

void remove_thread(unsigned int thread_id)

结束指定的线程并将其从线程队列中移除。thread_id 就是调用 create_thread()函数创建线程时返回的值。

void thread_thaw(unsigned int thread_id)

如果您创建线程时设置了 CREATE_THREAD_FROZEN 属性，您可以在需要的时候使用此函数将该线程唤醒。否则，该线程将不会被执行。

void thread_wait(unsigned int thread_id)

中断目前的线程直到线程 ID 被终止，类似与其他线程 API 中的 join()[1]。

void thread_exit(void)

立即退出目前的线程。

void yield(void)

强制执行一次环境切换。即使这样做，您也不能决定接下来应该执行哪一个线程。系统会自动根据优先级来决定接下来执行的线程。

void sleep(int ticks)

让线程睡眠一定的时钟周期。由于时钟周期可能会改变，线程睡眠的时长可能比您设计的更久，因此请不要太过依赖于具体的时钟长度[2]。稍后我们会介绍代替的方法。

使用线程（实例）

您可以三步创建出一个线程：

1. *编写一个 C 函数。注意，该函数不接受任何参数，也不能返回任何值。换句话说，该函数的类型应该是 void (*)(void)。*
2. *预分配堆栈空间。一个普通的线程不应使用过多的堆栈空间，如果您不能确定需要使用多少，请使用 DEFAULT_STACK_SIZE 宏指定空间大小。对于简单的线程，这一大小一般是足够的。大部分 CPU 希望堆栈是长整数格式对齐[3]的，因此您应该使用 long 格式。*
3. *调用 create_thread()函数。*

按照前文中的设计，我们有了可用的代码：

```
#include "thread.h"

/* My thread function */
void my_thread(void)
{
   while(1) {
      /* 您可以在这里做一些有趣的事情，不过这只是个示例，因此我们决定什么都不做 */
      yield(); /* 我们可不希望 Rockbox 在这里挂起，因此我们也要让其他线程运行 */
   }
}

static long my_stack[DEFAULT_STACK_SIZE/sizeof(long)];
```

[1] 原文为：Blocks the current thread until the thread identified by *thread_id* terminated (similarly to join() in other threading APIs).
[2] 原文为：Due to the cooperative nature, it can last longer, so do not rely on the exact ticks too much.
[3] 原文为：long-aligned

```
static unsigned int my_thread_id = -1;

void init_my_thread(void)
{
   my_thread_id = create_thread(my_thread, my_stack, sizeof(my_stack), "my thread", IF_PRIO(,
PRIORITY_BACKGROUND) IF_COP(, CPU));
}
```

系统时钟周期

概述

Rockbox 中有一种通用的计时机制，称为时钟周期。这是一个代表了每秒钟经过的时钟周期数的 32 位计数器[1]。您可以通过读取 current_tick 这个全局变量来了解目前的时钟周期计数。

时钟周期辅助函数

TIME_AFTER(a, b)

如果 a 计数器的值大于（晚于）b 则返回 true，否则返回 false。

示例：

```
unsigned long my_delay = current_tick + HZ;

   if(TIME_AFTER(current_tick, my_delay)) {
      splash(HZ, true, "1 second has passed");
   }
```

TIME_BEFORE(a, b)

如果 a 计数器的值小于（早于）b 则返回 true。

可以用该函数实现一个简单的 sleep()函数功能：

```
unsigned long my_delay = current_tick + HZ;

   while(TIME_BEFORE(current_tick, my_delay)) {
      yield();
   }
   splash(HZ, true, "1 second has passed");
```

内核周期任务

您可以声明一个函数并令其在每一个内核周期都自动执行一次。该函数会在每个中断环境中执行一次，而且**必须要短小、简单**[2]。

int tick_add_task(void (*f)(void))

加入一个内核周期任务。如果添加成功则返回 0，否则返回-1。

示例：

[1] 原文为：It is a 32-bit counter that is increased HZ times per second.

[2] 原文为：It will be executed in interrupt context and has to be very short and simple.

```
void my_tick_task(void)
{
   /* 做一些有趣的事情 */
}

void my_init(void)
{
   tick_add_task(my_tick_task);
}
```

int tick_remove_task(void (*f)(void))

从任务列表中移除一个任务。如果成功则返回 0，如果找不到指定的任务则返回-1。

事件队列

概述

线程与线程之间可以通过事件队列相互沟通。事件队列是一个循环的事件缓存。一个线程在其他线程被执行时可以等待消息进入事件队列。当事件到达时，线程将在下一次计划的时刻收到消息[1]。

一个事件类似于：

```
struct event
{
    long id;
    void *data;
};
```

id 变量就是该事件的编号，一般是任何 32 位的正整数，**而负数是为系统事件预留的**。data 指针可以指向任何您想通过该事件传递的数据。

如果您需要在您的线程中接收消息，应该定义一个全局事件队列，并在使用它之前初始化：

```
struct event_queue my_queue;
queue_init(&my_queue);
```

事件队列函数

void queue_init(struct event_queue *queue)

用于初始化事件队列。在使用一个事件队列之前，必须使用该函数初始化它。

示例：

```
struct event_queue my_queue;
queue_init(&my_queue);
```

void queue_delete(struct event_queue *q)

将 q 指向的事件队列（注意：不是事件队列中的事件而是队列本身）从队列列表中移除。您应该在事件列表被废弃后立即删除它，比如，插件中的事件列表在插件结束后就可以被废弃了。否则，当您使用 queue_broadcast()广播事件或者往该事件列表中插入一个事件时可能会带来不良后果。

[1] 原文为：When an event arrives, the thread will receive the event the next time it is scheduled.

void queue_wait(struct event_queue *q, struct event *ev)

等待一个事件，函数将当前事件复制到 ev 指向的事件结构中并从事件队列中移除它[1]。

示例：

```
struct event ev;

   queue_wait(&my_queue, &ev);
   if(ev.id == 1) {
       splash(HZ, true, "Got event 1");
   }
```

void queue_wait_w_tmo(struct event_queue *q, struct event *ev, int ticks)

作用与 void queue_wait()相似，但是如果等待的周期超过 ticks 的值，ev.id 的值会被设置为 SYS_TIMEOUT。

示例：

```
struct event ev;

   queue_wait_w_tmo(&my_queue, &ev, HZ);
   switch(ev.id) {
      case 1:
          splash(HZ, true, "Got event 1");
          break;
      case SYS_TIMEOUT:
          splash(HZ, true, "Timeout");
          break;
   }
```

bool queue_empty(const struct event_queue* q)

如果 q 指向的事件队列是空的则返回 true。

示例：

```
while(queue_empty(&my_queue)) {
     calculate();
     yield();
   }
   queue_wait(&my_queue, &ev);
   if(ev.id == 1) {
       splash(HZ, true, "Got event 1");
   }
```

void queue_clear(struct event_queue* q)

清空 q 指向的事件队列。

void queue_post(struct event_queue *q, long id, void *data)

向事件队列中添加一个事件。可以在另一个线程中或者中断处理[2]中调用该函数。

[1] 原文为：Waits for an event to arrive, removes the event from the queue and copies it to the event structure pointed to by *ev*.

[2] 原文为：interrupt handler。

示例：

```
extern struct event_queue your_queue;
queue_post(&your_queue, 1, NULL);
```

int queue_broadcast(long id, void *data)

广播事件，将该事件插入到所有线程的事件队列中，包括自己的队列。

保护共享数据

简介

Rockbox 中有一种用于保护共享数据的机制称为 mutex。这是一种简单的、允许独占资源的锁定机制。**mutex 只能在线程中使用，不能在中断处理中使用。**

Mutex 函数

void mutex_init(struct mutex *m)

初始化一个 mutex 结构。

void mutex_lock(struct mutex *m)

尝试锁定一个 mutex。如果该 mutex 已经被锁定，则直到它被解锁之前，锁定请求会被拒绝。

void mutex_unlock(struct mutex *m)

解锁 mutex。

Mutex 使用示例

```
struct mutex my_mutex;

void my_protected_function(void)
{
   mutex_lock(&my_mutex);
   /* Do the stuff that requires exclusive access */
   mutex_unlock(&my_mutex);
}

void init_my_protected_function(void)
{
   mutex_init(&my_mutex);
}
```

给开发者的一些建议

当您编写 Rockbox 代码时，请一定要顾及到其他系统线程。以下是一些建议：

1. ***不要在中断处理或者系统周期任务上花费太多执行时间。***
2. ***让您的线程不时休眠（yield）或睡眠（sleep）。****系统中的许多线程需要被经常性的执行。如果您的线程需要进行类似于 Mandelbrot 计算等冗长的操作，请一定偶尔调用一次 yield()，保证其他进程有机会运行。*

3. ***通常来说睡眠（sleep）比强制切换（yield）好***，*因为如果所有的线程都在睡眠状态，内核可以执行睡眠指令[1]以减少电力消耗。*

PCM 缓冲区

简介

PCM 缓冲区包含了即将（或者正在）被发送到音频输出的原始 PCM 数据。解码器在 PCM 缓冲区中分得空间，并且将解码得到的音频数据放入该空间内。另外，该缓冲区内还有两个辅助缓冲区，即用于合成淡入淡出的缓冲区及合成语音的缓冲区。

普通 PCM 回放

您可以通过调用 pcmbuf_request_buffer()函数来为后续的音频回放数据分配空间。当您完成写入音频数据并调用 pcmbuf_write_complete()函数时，该空间会被加入回放队列以完成后续回放。

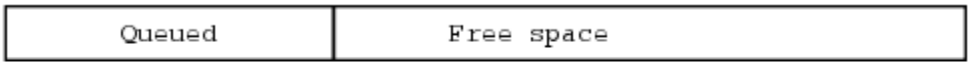
请求缓存空间

当您需要向 PCM 缓冲区写入数据时，您必须先请求缓冲区空间。您可以使用 pcmbuf_request_buffer()函数请求空间：

```
char *ptr;   /* 指向分配到的缓冲区的指针 */
size_t size; /* 函数会返回它可以为您分配的空间大小 */

ptr = pcmbuf_request_buffer(0x2000, &size);
/* 现在，size 变量保存了已分配空间的大小 */
```

在请求 PCM 缓冲区空间前，PCM 缓冲区看起来如下图：



PCM buffer before pcmbuf_request_buffer()

图 2 请求缓冲区空间前缓冲区的结构

分配完成之后：

PCM buffer after pcmbuf_request_buffer()

图 3 分配完成后的缓冲区结构

写入数据并插入队列

在得到 PCM 缓冲区的空间后，您应该将解码得到的原始音频数据写入缓冲区中并调用 pcmbuf_write_complete()函数将该缓冲区加入音频回放队列。作为例子，这里我们在缓冲区中填充 0：

```
memset(ptr, 0, size);
pcmbuf_write_complete(size);
/* 调用该函数后，除非再进行分配，否则就不应该再使用 ptr 指针指向的内存区域作为缓冲 */
```

完成以上操作之后，缓冲区的结构如下图：

[1] 原文为：SLEEP instruction。

PCM buffer after pcmbuf_write_complete()

图 4 写入数据并添加至队列后的缓冲区结构

特殊情况 1 淡入淡出

特殊情况 2 语音

USB 事件处理（从属设备模式）

概述

当运行 Rockbox 的设备连接到 PC 时，PC 将会识别到一个通用的 USB 大容量存储设备。当用户插入 USB 线时，Rockbox 应该停止所有的硬盘访问并进入 USB 模式。当用户拔出 USB 线时，Rockbox 应该继续按照正常方式运行。

USB 线程

USB 线程负责管理 USB 控制器并监听 USB 端口的连接情况。当用户连接 USB 线时，该线程给所有其他线程发送消息以停止硬盘访问并要求当其他线程完成停止访问硬盘后发送报告。当所有的线程都准备完毕时，USB 线程将硬盘访问权交给 USB 控制芯片，并等待 USB 端口断开连接。当用户断开 USB 线时，USB 线程回收控制权并告知其他线程可以继续访问硬盘了。

信号发送机制

USB 线程使用 queue_broadcast()函数通知其他线程停止访问硬盘。这意味着如果某个线程没有事件队列，它将不会被通知到。

USB 线程不会使 Rockbox 进入 USB 模式直到所有线程准备就绪为止。

步骤一：发送 SYS_USB_CONNECTED 标志到所有线程。

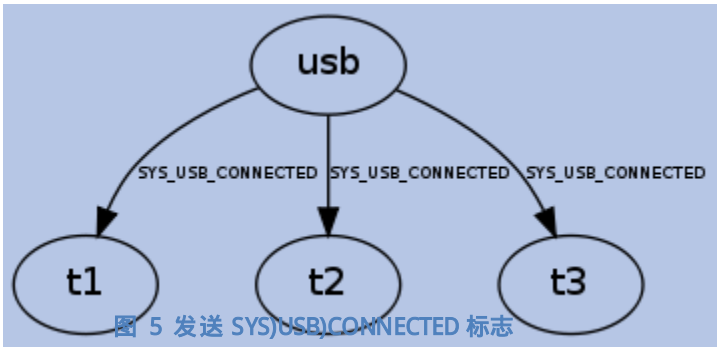


图 5 发送 SYS_USB_CONNECTED 标志

步骤二：所有（准备完毕的）线程返回 SYS_USB_CONNECTED_ACK 消息。

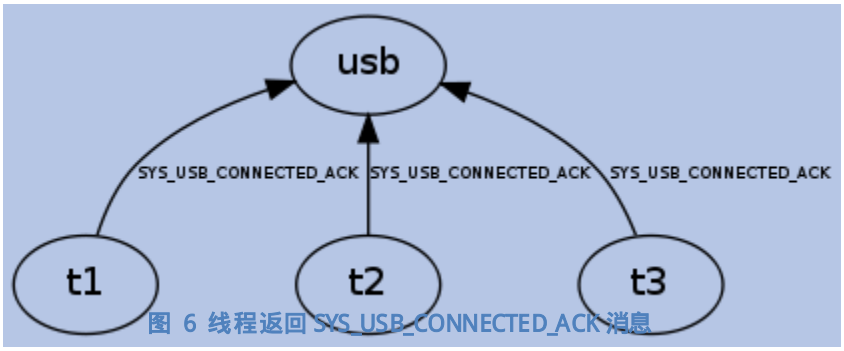


图 6 线程返回 SYS_USB_CONNECTED_ACK 消息

SYS_USB 事件

Event	Description
SYS_USB_CONNECTED	当连接 USB 线时，向所有线程发送该消息。
SYS_USB_CONNECTED_ACK	当线程收到 SYS_USB_CONNECTED 消息并准备就绪时，返回该消息。
SYS_USB_DISCONNECTED	当 USB 线拔出时，向所有线程发送该消息。
SYS_USB_DISCONNECTED_ACK	当线程准备就绪退出 USB 模式时返回该消息。

相关辅助函数

void usb_acknowledge(long id)

可以使用该函数发送 USB 事件的回复[1]。

示例：

```
usb_acknowledge(SYS_USB_CONNECTED_ACK);
```

void usb_wait_for_disconnect(struct event_queue *q)

可以使用该函数等待 USB 线程发送 SYS_USB_DISCONNECTED 事件，同时该函数会给您的线程发送 SYS_USB_DISCONNECTED_ACK 事件。该函数需要您提供一个指向事件队列的指针。

```
usb_wait_for_disconnect(&my_queue);
```

int usb_wait_for_disconnect_w_tmo(struct event_queue *q, int ticks)

与 usb_wait_for_disconnect()函数基本相同，但是您可以指定一个超时时长。一旦超过您指定的超时时长，无论 USB 是否断开函数都会返回值。若超时则返回 1，否则返回 0。

应用示例

```
    static struct event_queue my_queue;

void my_thread(void)
{
```

[1] 原文为：Call this to send an acknowledge to a USB event.

```
    struct event ev;

  while(1) {
     queue_wait_w_tmo(&my_queue, &ev, 0);

     switch(ev.id) {
     case SYS_USB_CONNECTED:
        /* 完成清理工作，如关闭文件等 */
        usb_acknowledge(SYS_USB_CONNECTED_ACK);

        /* 一直等待 USB 断开 */
        usb_wait_for_disconnect(&my_queue);
        break;
     }
  }
```

特殊情况：主线程

主线程负责在 USB 模式中显示“USB 已连接”等，因此对于 USB 事件主线程有与其他线程不同的处理方式。SYS_USB_CONNECTED 消息是由按键事件队列（有序队列）接收的，因此您需要在按键事件处理代码中一起处理 USB 事件。为了您的方便，默认事件处理器可以帮您处理这些事情。

您可以在 case 语句的 default 分支中调用默认事件处理器。它会返回一个事件的值。

举例说，如果您要在 USB 模式时退出函数，您可以这样做：

```
while (!exit) {
       key = button_get();
       switch( key ) {
       case BUTTON_STOP:
           exit = true;
           break;

       default:
           if(default_event_handler(key) == SYS_USB_CONNECTED) {
               return MYFUNC_ATTACHED_USB;
           }
           break;
       }
    }
```

创建菜单和菜单项

概述

本节将介绍如何使用 Rockbox 提供的创建菜单、菜单项的 API。您可以在 Rockbox 系统和插件中使用它们。如果您在插件中使用，请不要忘记在函数前面加入 rb->。

所有有关菜单的宏都定义在 apps/menu.h 文件中。

宏

Rockbox 中定义了一些重要的宏：

- MENUITEM_SETTING(name, var, callback)
- MENUITEM_SETTING_W_TEXT(name, var, str, callback)
- MENUITEM_STRINGLIST(name, str, callback, ...)
- MENUITEM_RETURNVALUE(name, str, val, callback, icon)
- MENUITEM_RETURNVALUE_DYNTEXT(name, val, callback, text_callback, text_cb_data, icon)
- MENUITEM_FUNCTION(name, flags, str, func, param, callback, icon)
- MENUITEM_FUNCTION_DYNTEXT(name, flags, func, param, text_callback, text_cb_data, callback, icon)
- MAKE_MENU(name, str, callback, icon, ...)
- In all the above, **name** is the name of the menu item variable, **callback** is the function to call before the menu item is displayed, and at certain times while the menu is running (explained later). **icon** is the icon to use when the menu item is displayed (the enum list is in apps/gui/icons.h)

在以上宏中，name 参数定义了菜单名称，callback 为现实菜单前调用的函数（稍后我们会讨论该项），icon 定义了菜单的图标。

附录

附录 A 中英文对照表

该表收录了本手册中出现的一些英语词汇，方便您对照。如果有翻译得不准确的，请您与作者联系修正，我们对此表示十分感谢。

中文	英语	中文	英语
编译时刻	Compile-time	连接时刻	Linkage
列表控件	List widget	环境切换	Context switch
线程环境	Thread context	时钟周期	Tick
事件队列	Event queue	中断处理	interrupt handler
线程环境	Thread context		

致谢

因为有了广大网友和您的支持，本手册才得以完成及不断修正更新。对于所有的贡献者，我们都表示衷心的感谢。

编写者

PurlingNayuki
罗勇

贡献者

Rockbox Wiki 所有贡献者
Rockbox 中文社区开发小组成员
Rockbox 中文社区主题小组成员

（排名不分先后，仅按照贡献顺序）