三方应用

Version 1.1

修订日期： 2024-02-21

内容

1. QT

1.1. QT Cross Compile

1.1.1. 建立git仓库进行跟踪

因为在编译的时候会对QT 进行部分的代码修改，建立仓库可以很好的跟踪修改记录

.gitignore 会把一些不需要的中间文件不进行跟踪

```
git init
git add *
vim .gitignore
git add .gitignore
git commit -m "init"

cat .gitignore
*.a
*.asn1.[ch]
*.bin
*.bz2
*.dwo
*.elf
*.gcno
*.gz
*.i
*.ko
*.lex.c
*.ll
*.lst
*.lz4
*.lzma
*.lzo
*.o
*.o.*
*.s
*.so
*.so.dbg
```

1.1.2. 配置编译工具链

工具链的配置建议使用qt提供的qmake机制进行配置，配置文件在 `mkspecs/qws`目录下，通过两个文件完成配置

```
cd mkspecs/qws/
cp -r linux-arm-gnueabi-g++ linux-riscv-gnueabi-g++
cd linux-riscv-gnu-g++/
```

1. **qplatformdefs.h**
    - QT 代码本身兼容32位和64位，因此对于64位CPU，不需要进行过多的设置
    - qplatformdefs进行一些基础的数据变量的定义，如type，ipc等，和linux平台的关联性更强，和arm riscv 的关联性没有

      ```
      cat qplatformdefs.h
      #include "../../linux-g++-64/qplatformdefs.h"

      cat ../../linux-g++-64/qplatformdefs.h
      #include "../linux-g++/qplatformdefs.h"
      ```

2. **qmake.conf**
    - qmake 配置了交叉编译的工具链信息，是交叉编译的主要
    - qmake 并没有指定工具链的路径，因此需要编译前把路径手工加入到PATH环境变量

- export PATH=/xxx/d211/bin/:$PATH
- 包含工具参数的定义：QMAKE_COPY = cp –f
- qws.conf中定义基础功能：QT += core gui network

```
#
# qmake configuration for building with arm-none-linux-gnueabi-g++
#

include(../../common/linux.conf)
include(../../common/gcc-base-unix.conf)
include(../../common/g++-unix.conf)
include(../../common/qws.conf)

# modifications to g++.conf
QMAKE_CC                = riscv64-unknown-linux-gnu-gcc
QMAKE_CXX               = riscv64-unknown-linux-gnu-g++
QMAKE_LINK              = riscv64-unknown-linux-gnu-g++
QMAKE_LINK_SHLIB        = riscv64-unknown-linux-gnu-g++

# modifications to linux.conf
QMAKE_AR                = riscv64-unknown-linux-gnu-ar cqs
QMAKE_OBJCOPY           = riscv64-unknown-linux-gnu-objcopy
QMAKE_STRIP             = riscv64-unknown-linux-gnu-strip

load(qt_config)
```

3. **环境变量**

   和传统的交叉编译不同，QT 采用的是QMake进行系统编译，编译方案是先进行一些Host 的工具编译（采用CC），再进行交叉编译（采用QMAKE_CC），因此在设置环境变量时， 只可以添加交叉编译工具链到PATH，设置QMAKE_CC， 绝对不能设置CC的传统的交叉编译变量，否则出现莫名的问题。

   如果需要directfb的支持，则需要明确指定一下directfb的路径，导如下两个环境变量

```
export QT_CFLAGS_DIRECTFB="-I/xxx/install/include/directfb/"
export QT_LIBS_DIRECTFB="-L/xxx/install/lib -ldirectfb -lfusion -ldirect -lpthread -lz"
```

1.1.3. 编译

1. configure

   - 用最简单的编译方式编译，大部分功能先disable，如果需要再打开
   - 通过–xplatform qws/linux–arm–gnueabi–g++指定交叉编译工具链
   - configure 完成后会生成Makefile
   - make distclean命令无效，无法清除生成的Makefile

```
./configure -opensource -confirm-license -xplatform qws/linux-riscv-gnueabi-g++ \
-prefix /xxx/install -plugin-gfx-directfb -no-qt3support \
-nomake demos -nomake examples -no-glib -depths 8,16,32 -qt-gfx-linuxfb -no-gfx-transformed \
-no-gfx-qvfb -no-gfx-vnc -no-gfx-multiscreen -no-mouse-pc -no-mouse-linuxtp -no-mouse-linuxinput \
-no-mouse-tslib -no-mouse-qvfb -no-kbd-tty -no-kbd-linuxinput -no-kbd-qvfb -release -shared \
-little-endian -embedded arm -no-gif -no-libmng -no-accessibility  -no-libjpeg -no-libtiff -no-freetype \
-no-openssl -no-opengl -no-sql-sqlite -no-xmlpatterns -no-multimedia -no-audio-backend -no-phonon \
-no-phonon-backend -no-svg -no-webkit -script -no-scripttools -no-stl -no-declarative -no-pch \
-no-xinerama -no-cups -no-nis -no-separate-debug-info -fast -no-rpath
```

2. **swpb 汇编**

   我们的工具链没有提供swpb汇编，需要直接使用C语言的交换方式完成

```
../../include/QtCore/../../src/corelib/arch/qatomic_armv5.h:128: Error: unrecognized opcode `swpb a3,a4,[s0]'

diff --git a/src/corelib/arch/qatomic_armv5.h b/src/corelib/arch/qatomic_armv5.h
index 27e23d9e..b265d149 100644
```

```
--- a/src/corelib/arch/qatomic_armv5.h
+++ b/src/corelib/arch/qatomic_armv5.h
@@ -125,10 +125,12 @@ Q_CORE_EXPORT __asm char q_atomic_swp(volatile char *ptr, char newval);
inline char q_atomic_swp(volatile char *ptr, char newval)
{
    register char ret;
-    asm volatile("swpb %0,%2,[%3]"
-                 : "=&r"(ret), "=m" (*ptr)
-                 : "r"(newval), "r"(ptr)
-                 : "cc", "memory");
+    //asm volatile("swpb %0,%2,[%3]"
+    //             : "=&r"(ret), "=m" (*ptr)
+    //             : "r"(newval), "r"(ptr)
+    //             : "cc", "memory");
+    ret = *ptr;
+    *ptr = newval;
    return ret;
}

@@ -227,10 +229,12 @@ inline int QBasicAtomicInt::fetchAndStoreOrdered(int newValue)
{
    int originalValue;
#ifndef QT_NO_ARM_EABI
-    asm volatile("swp %0,%2,[%3]"
-                 : "=&r"(originalValue), "=m" (_q_value)
-                 : "r"(newValue), "r"(&_q_value)
-                 : "cc", "memory");
+    //asm volatile("swp %0,%2,[%3]"
+    //             : "=&r"(originalValue), "=m" (_q_value)
+    //             : "r"(newValue), "r"(&_q_value)
+    //             : "cc", "memory");
+    originalVlaue = _q_value;
+    _q_value = newValue;
```

3. **gnu兼容**

   工具链版本的不同，会使某些C++ code 的兼容性会有差异，需要指定gnu++98

```
error: 'std::tr1' has not been declared

--- a/mkspecs/common/g++-base.conf
+++ b/mkspecs/common/g++-base.conf
@@ -15,7 +15,7 @@ QMAKE_LINK_C_SHLIB = $$QMAKE_CC

QMAKE_CFLAGS_RELEASE_WITH_DEBUGINFO += -O2 -g

-QMAKE_CXX = g++
+QMAKE_CXX = g++ -std=gnu++98

QMAKE_LINK       = $$QMAKE_CXX
QMAKE_LINK_SHLIB = $$QMAKE_CXX

--- a/mkspecs/common/gcc-base.conf
+++ b/mkspecs/common/gcc-base.conf
@@ -42,7 +42,7 @@ QMAKE_CFLAGS_STATIC_LIB     += -fPIC
QMAKE_CFLAGS_YACC           += -Wno-unused -Wno-parentheses
QMAKE_CFLAGS_HIDESYMS       += -fvisibility=hidden

-QMAKE_CXXFLAGS            += $$QMAKE_CFLAGS
+QMAKE_CXXFLAGS            += $$QMAKE_CFLAGS -std=gnu++98
QMAKE_CXXFLAGS_DEPS       += $$QMAKE_CFLAGS_DEPS
```

1.1.4. Directfb 集成

configure会优先使用pkg_config, directfb-config 两个工具进行directfb的路径获取，会很容易的受到系统directfb的影响， 因此此处最简单的办法的绕开系统的变量，直接设置QT_CFLAGS_DIRECTFB 和QT_LIBS_DIRECTFB的变量值

```
The DirectFB screen driver functionality test failed!

--- a/configure
+++ b/configure
@@ -6641,13 +6641,13 @@ if [ "$PLATFORM_QWS" = "yes" ]; then
fi

if [ "${screen}" = "directfb" ] && [ "${CFG_CONFIGURE_EXIT_ON_ERROR}" = "yes" ]; then
-           if test -n "$PKG_CONFIG" && "$PKG_CONFIG" --exists directfb 2>/dev/null; then
-               QT_CFLAGS_DIRECTFB=`$PKG_CONFIG --cflags directfb 2>/dev/null`
-               QT_LIBS_DIRECTFB=`$PKG_CONFIG --libs directfb 2>/dev/null`
-           elif directfb-config --version >/dev/null 2>&1; then
-               QT_CFLAGS_DIRECTFB=`directfb-config --cflags 2>/dev/null`
-               QT_LIBS_DIRECTFB=`directfb-config --libs 2>/dev/null`
-           fi
+           #if test -n "$PKG_CONFIG" && "$PKG_CONFIG" --exists directfb 2>/dev/null; then
+           #    QT_CFLAGS_DIRECTFB=`$PKG_CONFIG --cflags directfb 2>/dev/null`
+           #    QT_LIBS_DIRECTFB=`$PKG_CONFIG --libs directfb 2>/dev/null`
+           #elif directfb-config --version >/dev/null 2>&1; then
+           #    QT_CFLAGS_DIRECTFB=`directfb-config --cflags 2>/dev/null`
+           #    QT_LIBS_DIRECTFB=`directfb-config --libs 2>/dev/null`
+           #fi
```

1.1.5. config.test

config.tests 是configure的测试目录，其中有不少简单的verify示例，以directfb为例

```
config.tests/qws/directfb
directfb.cpp  directfb.o  directfb.pro  Makefile

cat Makefile
CC            = riscv64-unknown-linux-gnu-gcc
CXX           = riscv64-unknown-linux-gnu-g++
DEFINES       =
CFLAGS        = -pipe -O2 -Wall -W $(DEFINES)
CXXFLAGS      = -pipe -std=gnu++98 -D_REENTRANT -O2 -Wall -W $(DEFINES)
INCPATH       = -I../../../mkspecs/qws/linux-riscv-gnueabi-g++ -I. -I/usr/include/directfb
LINK          = riscv64-unknown-linux-gnu-g++
LFLAGS        = -Wl,-O1
LIBS          = $(SUBLIBS)   -L/usr/lib/x86_64-linux-gnu -ldirectfb -lfusion -ldirect -lpthread
AR            = riscv64-unknown-linux-gnu-ar cqs
```

- 如果某一test没有过，可以通过查看Makefile 确认相关参数是否正确，如上的include参数不正确
- 如果参数正确，可以直接进入该目录执行make命令确认出错原因
- 最初编译directfb是动态link zlib，因此需要明确link zlib，第一次仿照上面的LIBS设置，则在编译的时候报告zlib 的错误

1.2. QT Luban

1.2.1. 编译配置

1. make menuconfig 中打开 QT + Directfb 编译开关：

   ```
   [*] Third-party packages
       [*] Qt --->
           [*]   Use prebuilt binary instead of building from source
   ```

2. 如果勾选 Use prebuilt binary instead of building from source 则使用预编译的二进制

   如果不勾选prebuilt binary instead of building from source 则使用源码编译

   ```
   [*] Third-party packages
       [*] Qt --->
           [*]   Use prebuilt binary instead of building from source
   ```

1.2.2. 目录解释

- `dl`：以压缩包的方式存放源码包
- `package/third-party/qt/`: 存放将注入到源码中的补丁
- `prebuilt/riscv64-linux-gnu/`：存放预编译的
- `source/third-party/qt-4.8.7/`: 编译目录，源码 + 补丁 + 中间文件组成

1.2.3. 编译命令

make qt– + tab 显示所有qt的编译命令

- make qt–show–build–order：显示编译依赖和顺序
- make qt–extract： 把dl中的压缩包解压到source目录
- make qt–patch：把package中的patch打到source中的源码中
- make qt–reconfigure：对该源码包重新执行配置、编译、安装
- make qt–rebuild：对该源码包进行重新编译
- make qt–reinstall：对该源码包进行重新安装
- make qt–prebuilt：为该源码包生成预编译二进制压缩包，然后可以上传
- make qt–clean: 删除该源码包的所有编译输出
- make qt–distclean: 删除该源码包的源码
- make qt: 完成从 extract/patch/../build/install 的所有过程

1.3. QT 性能

D211 有两部分硬件加速可以优化 QT 的性能

- GE： ArtInChip 的 Graphic Engine 的简称，其提供基础的 2D 操作，如缩放，旋转，alpha等， QT 应用借助于 GE 的支持动画效果应该更流畅。
- VE： ArtInChip 的 Video Engine 的简称，其提供 png， jpeg 图片的解码操作， QT 应用借助于 VE 的支持可以更快速的显示图片，并节省 CPU 资源。

1.3.1. GE

在 QT 中有两种方案进行硬件加速

1. directFB：提供 QT 硬件加速的传统框架， ArtInChip 主要在 directFB 中添加了三块逻辑来实现 GE 的支持。
   - DMA Buffer 接口，提供连续内存块的管理: src/core/dmabuf_surface_pool.c
   - GE Driver 接口实现，gfxdrivers/ge/
   - GE Device 接口实现，gfxdrivers/ge/
2. AIC_MPP：mpp （Media Process Platform）是 ArtInChip 封装的一套操作 DMA Buffer， GE, VE 的统一接口，可以在 QT 代码中直接调用这些接口来完成硬件加速

   直接使用 mpp 接口的效率最高，对资源的使用也更安全，但对编码要求也更高

   具体代码参考： `source/artinchip/qtlauncher/views/aicdashboardview.cpp`

1.3.2. PNG

QT 对 PNG 文件的使用有三种方案：

- QT 代码直接使用 libpng， 此处因为无法直接访问使用 DMA Buffer，内存复制浪费比较多，无法对接 VE 的解码接口
- QT 使用 directFB 中对 libpng 的封装，directFB 中的 surface 结构实现了对 DMA Buffer 的封装，对接 VE 的解码接口比较高效, 具体代码参考：directfb-1.7.7/interfaces/IDirectFBImageProvider/idirectfbimageprovider_png.c
- QT 应用代码中直接调用 aic_mpp 封装的 VE, GE 的接口，直接获得处理后的图像内容，然后作为 QT 的控件的一部分使用，此用法结合图片缓存，可以防止 DMA Buffer 的碎片化，具体代码参考： source/artinchip/qtlauncher/views/aicdashboardview.cpp

1.3.3. JPEG

JPEG 的使用路径类似 PNG，但 VE 解码 JPEG 后的格式为 YUV， 该格式一般无法直接在 surface 中使用， 因此需要调用 GE 进行一次格式转换， 如果操作比较频繁，容易造成 DMA Buffer 的内存碎片，因此并没有直接对接 JPEG 的解码接口

1.4. QT Windows IDE

1.4.1. QT 可以做什么

- 漂亮的UI
- 多线程，数据库，图像处理，音频视频处理，网络通讯，文件操作
- 1997年， 开发了KDE，Linux 下开发C++ GUI 的事实标准
- WPS, YY 语音，Skype， VirtualBox，Opera，Google地图

1.4.2. QML

- QML 是一种描述性脚本语言
- QT + QML 是为了适应手机移动应用开发
- QT5 开始支持，如果UI 是QT4.X的，则不能使用

1.4.3. QT Creator

是跨平台的QT IDE 工具，是QT被Nokia收购后推出的一款新的轻量级集成开发环境。

1.4.4. Windows环境搭建

在Windows 上搭建QT开发环境，需要三个软件的安装支持

- mingw-4.8.2
- qt-4.8.7
- qt-creator-4.2.0

QT库的安装需要设置MinGW 路径，而qtcreator 需要配置MinGW 和QT 来进行编译

Windows 上有32位和64的区别，其中QT-4.8.7只支持32位的，因此mingw 和 qt-creator 也必须是32位的

文件命名上，x86_64的为64位，仅x86的一般为32位

1. mingw-4.8.2
   - 文件：i686-4.8.2-release-posix-dwarf-rt_v3-rev3
   - 链接： http://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20Win32/Personal%20Builds/mingw-builds/4.8.2/threads-posix/dwarf/i686-4.8.2-release-posix-dwarf-rt_v3-rev3.7z/download

   mingw的安装比较简单，无特殊选项
2. QT4.8.7

- 文件： qt-opensource-windows-x86-mingw482-4.8.7.exe
- 链接：https://download.qt.io/archive/qt/4.8/4.8.7/

在QT 的安装过程中，会要求设置MinGW 的安装路径

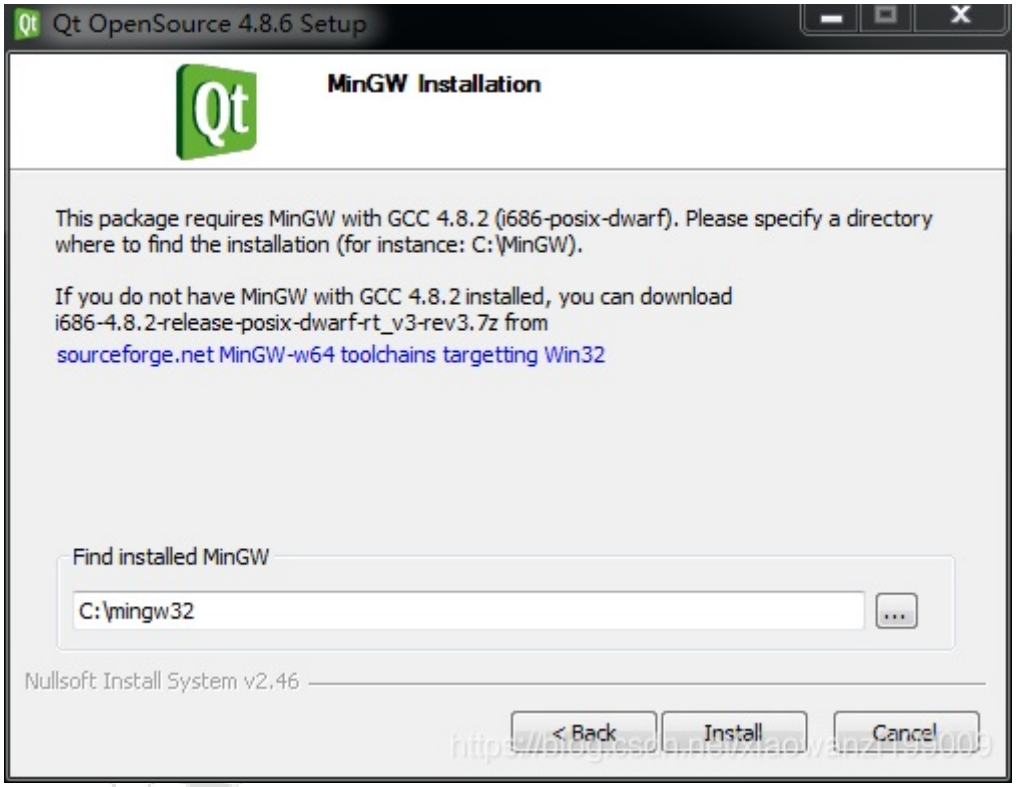


图 1-1 设置MinGW 的路径

3. qt-creator-4.2.0

qt-creator 的安装比较简单，复杂点在安装后的编译参数设置

- 文件：qt-creator-opensource-windows-x86-4.2.0.exe
- 链接：https://download.qt.io/archive/qtcreator/4.2/4.2.0/

1.4.5. QT Creator 主界面

安装完成后的主界面为：

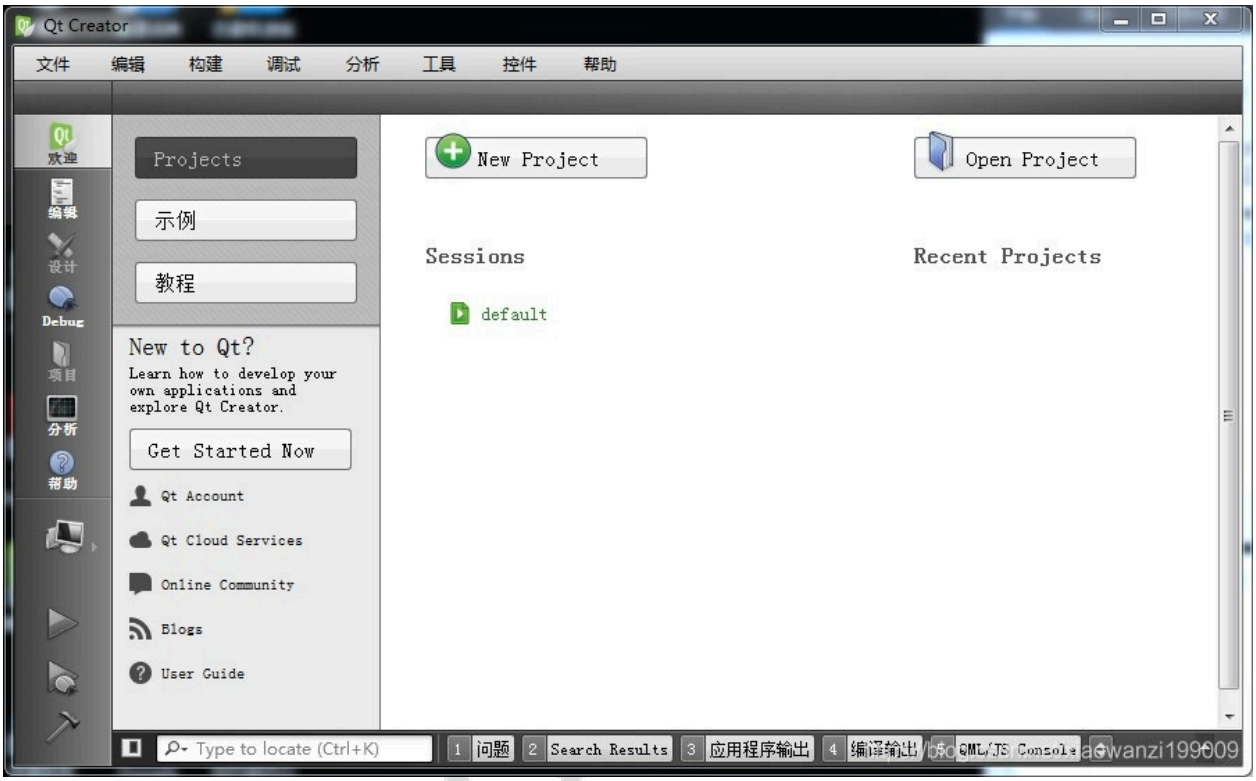


图 1–2 QT–Creator 主界面

1.4.6. QtCreator 构建和运行

初始的QtCreator 还没有配置，尚不能编译程序，通过菜单“工具” –>“选项”，在对话框左边选“构建和运行”进行构件和运行设置

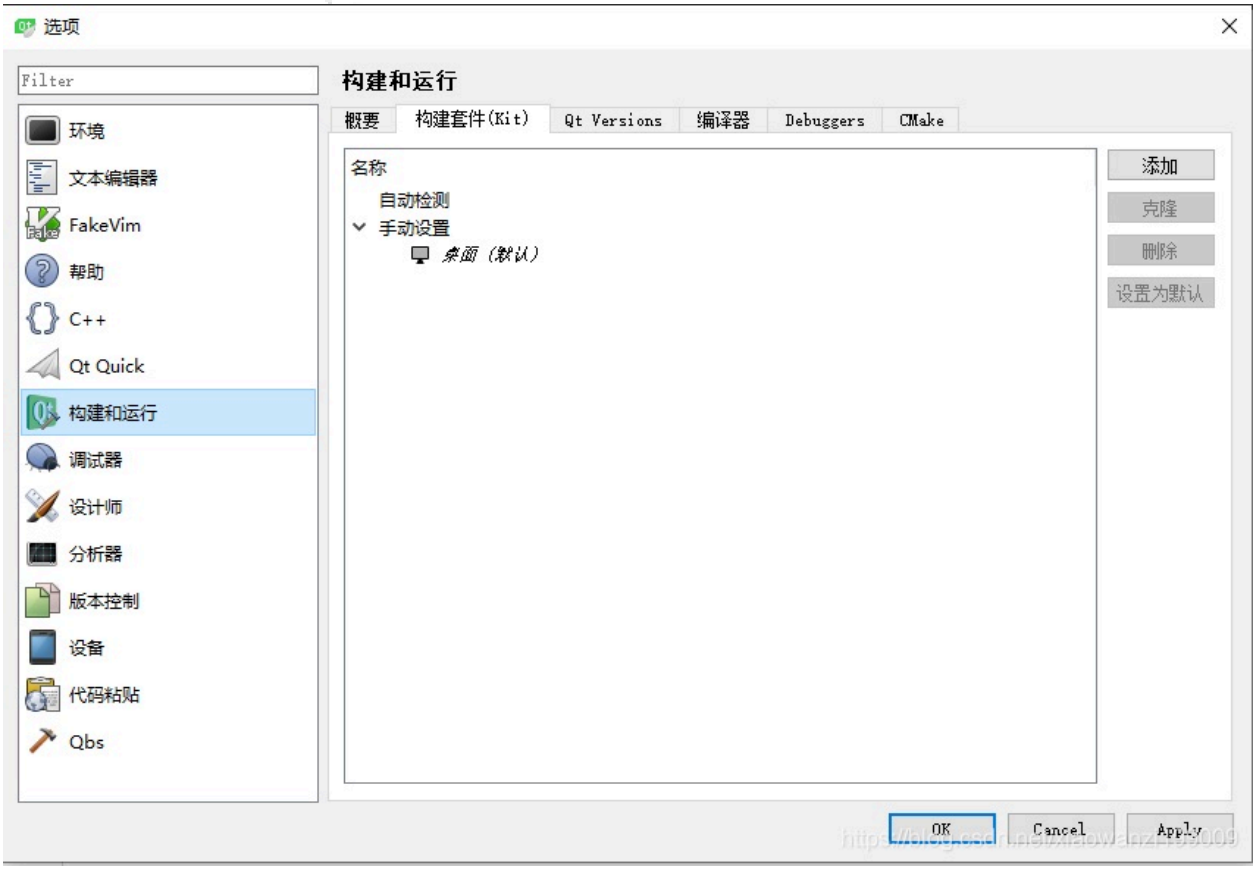


图 1–3 构建和运行设置界面

1.4.7. Debuggers

- 点击Add，进入新的Debugger 设置界
- 设置Name 为GDB
- 通过浏览找到MinGW 的gdb.exe
- “Apply” 设置好调试器

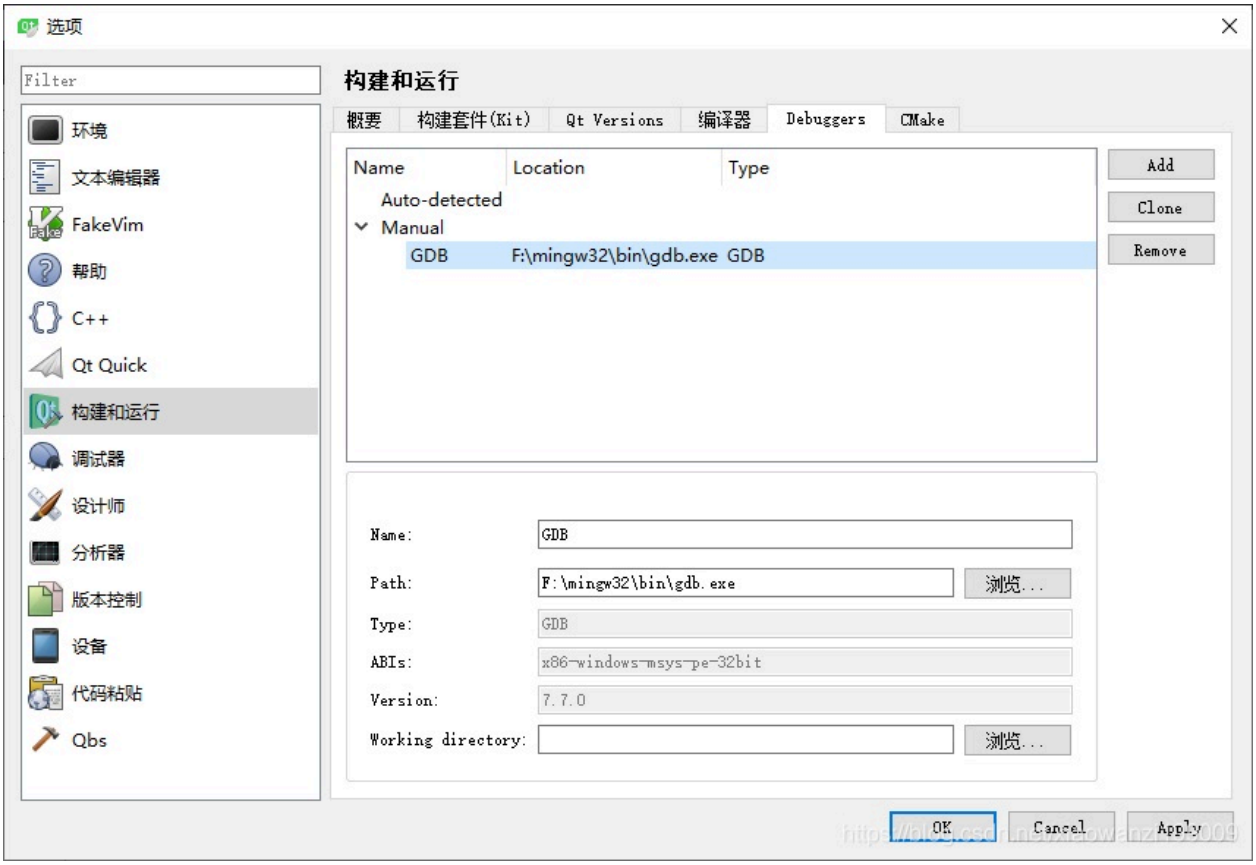


图 1–4 Debuggers 设置界面

1.4.8. 编译器

- 编译器主要设置C,C++工具
- 点击右边“添加”，弹出菜单中选择MinGW，分别添加C、C+
- 名称均设为MinGW
- C++ 设置为MinGW 的g++.exe
- C 设置为MinGW 的gcc.exe

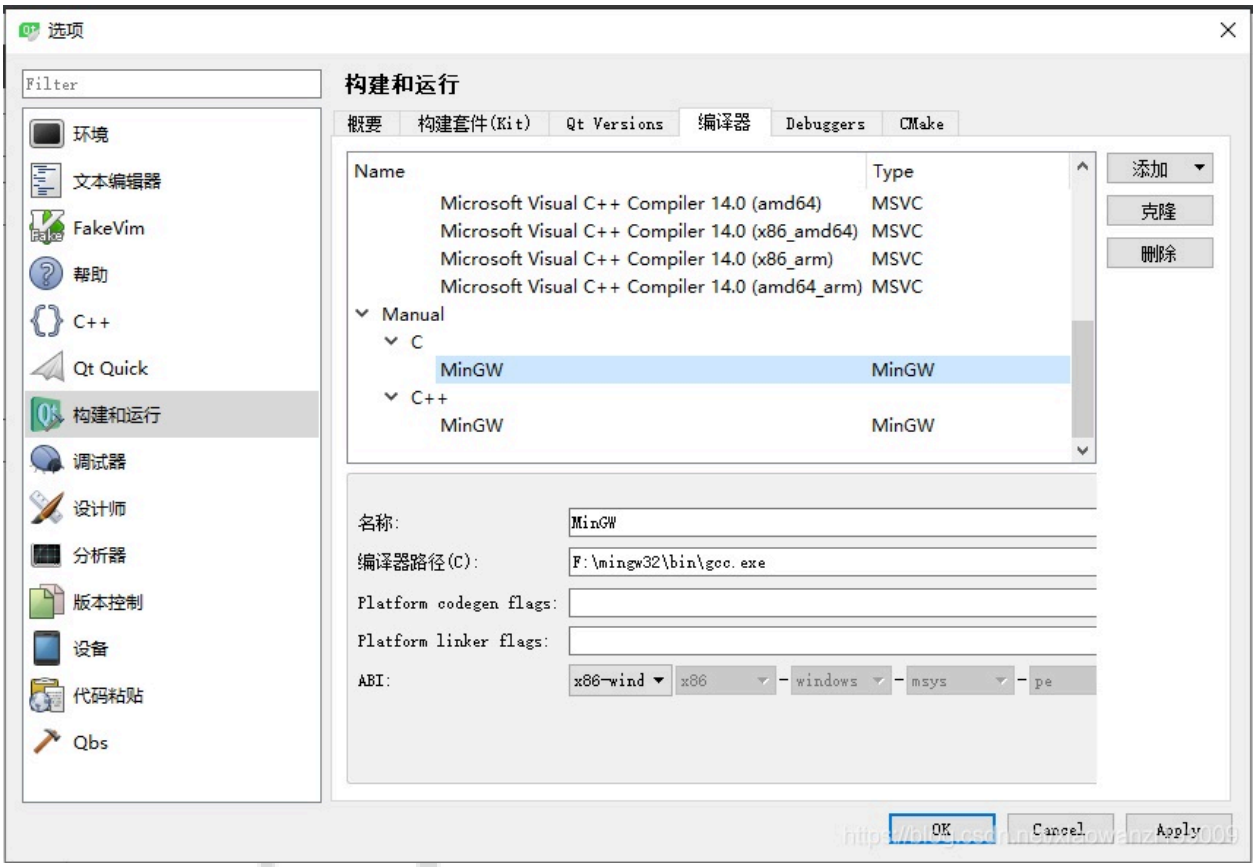


图 1–5 编译器设置界面

1.4.9. Qt Versions

- 配置QT 的qmake 工具
- 点击“添加”，浏览找到QT–4.8.7 下的qmake.exe

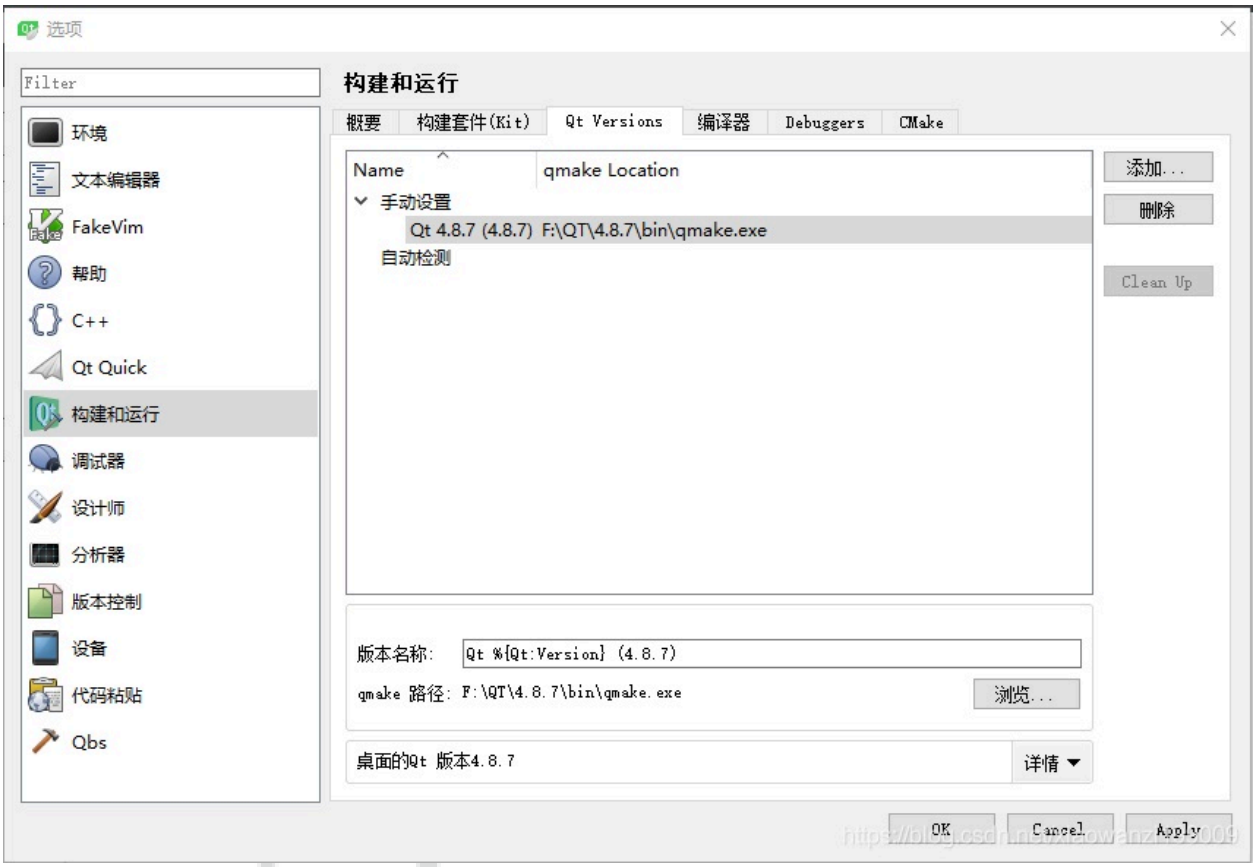


图 1–6 QT 版本设置界面

1.4.10. Kit

- 名称可以选一个有记录意义的名字
- 设备类型选择桌面
- 编译器C/C++分别选择上面设置的MinGW
- 调试器选择上面设置的GDB
- Qt 版本选择Qt 4.8.7
- Qt mkspec可以不用管，因为qmake.exe可以自动设置

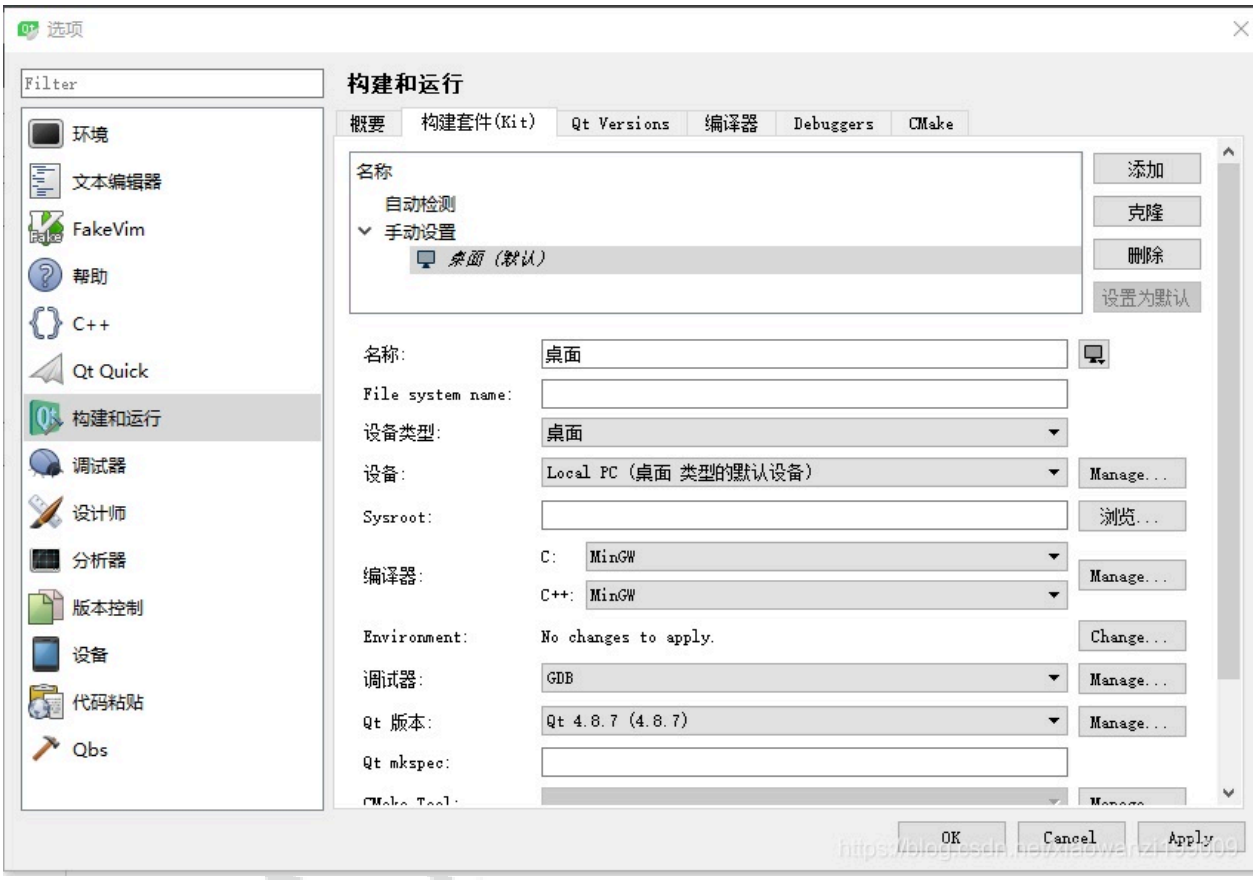


图 1–7 构件套件设置界面

1.4.11. qt–demo 运行

配置成功后，打开一demo工程，则Run 按钮可用，点击demo app 运行成功，则说明各项配置成功

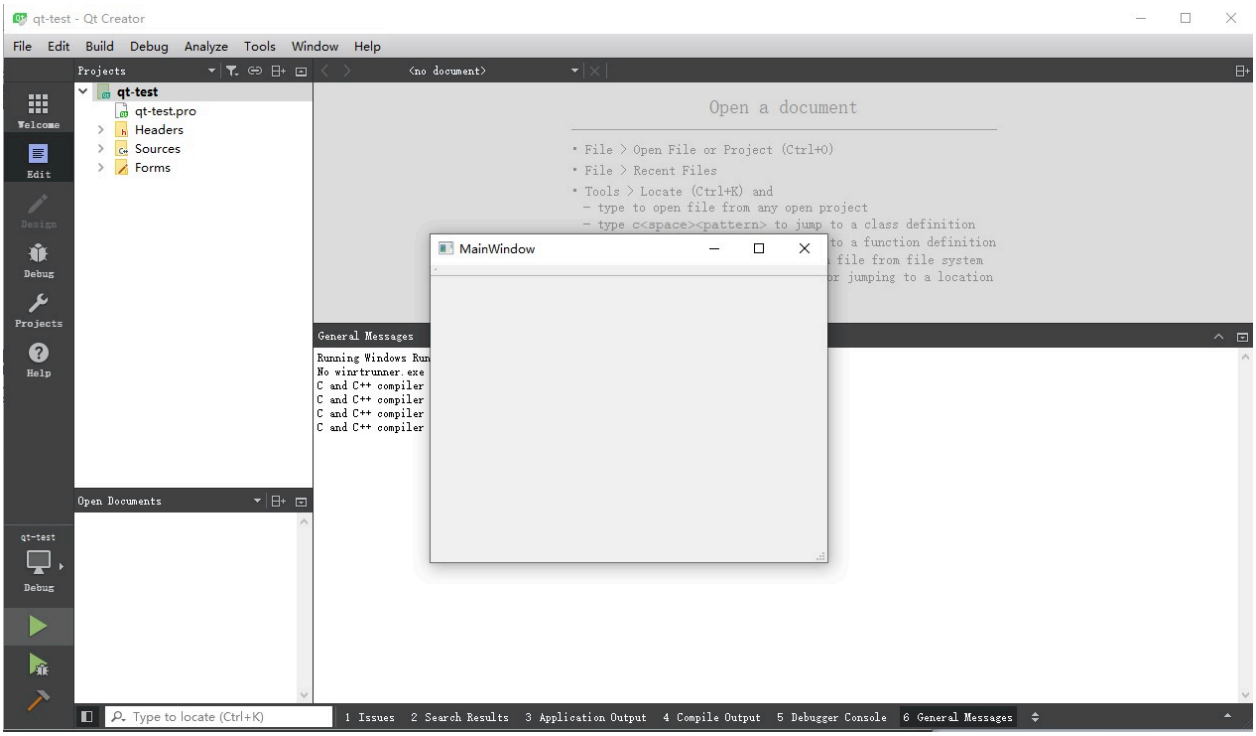


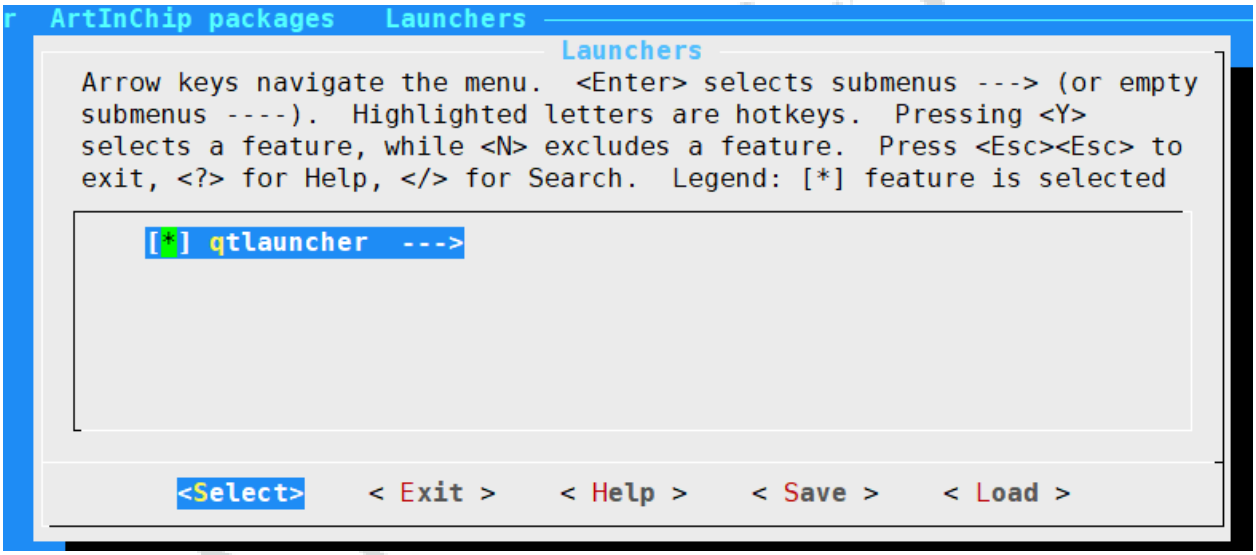
图 1–8 编译运行界面

1.5. QT 应用开发

QtLauncher 是 ArtInChip 基于 QT4.8.7 开发的一款应用程序，也是 QT 应用开发的一个典型示例，可以直接在 Luban 系统上运行。

1. 编译

   - 在SDK 根目录执行 make menuconfig 打开 Luban 配置界面
   - 选择 **ArtInChip packages > Launchers > qtlauncher**



   - 因为要开机运行，所以不能有其他的开机运行界面存在，如 test-lvgl
   - 打开 qtlauncher 后默认会自动选择qt4.8.7等依赖
   - 最好 make clean；make； 全编译一次SDK

2. 调整分区

   因为 QT 库比较大，开发板原始为了演示更多功能，分区设置的比较保守，因此打开 QT 支持后一般需要调整一下分区大小，否则会报告如下错误：

   Error: max_leb_cnt too low (266 needed)

3. 自动运行

   Luban 的开机自动运行使用的是 init.d 机制，qtlauncher 的自动运行是通过 package/artinchip/qtlauncher/S99qtlauncher 实现的， 编译时 S99qtlauncher 会被复制到目标机的 /etc/init.d/ 下

4. G2D

   QT 中集成了 G2D 的演示示例，代码在 source/artinchip/qtlauncher/views/aicdashboardview.cpp 中， G2D 示例中主要用到了 png 解码和 blit，rotate 等功能。

   为了在 Windows 上也可以编译和调试该 qtlauncher 程序，代码中对 G2D 的代码进行了宏屏蔽。

1.5.1. 调整分区

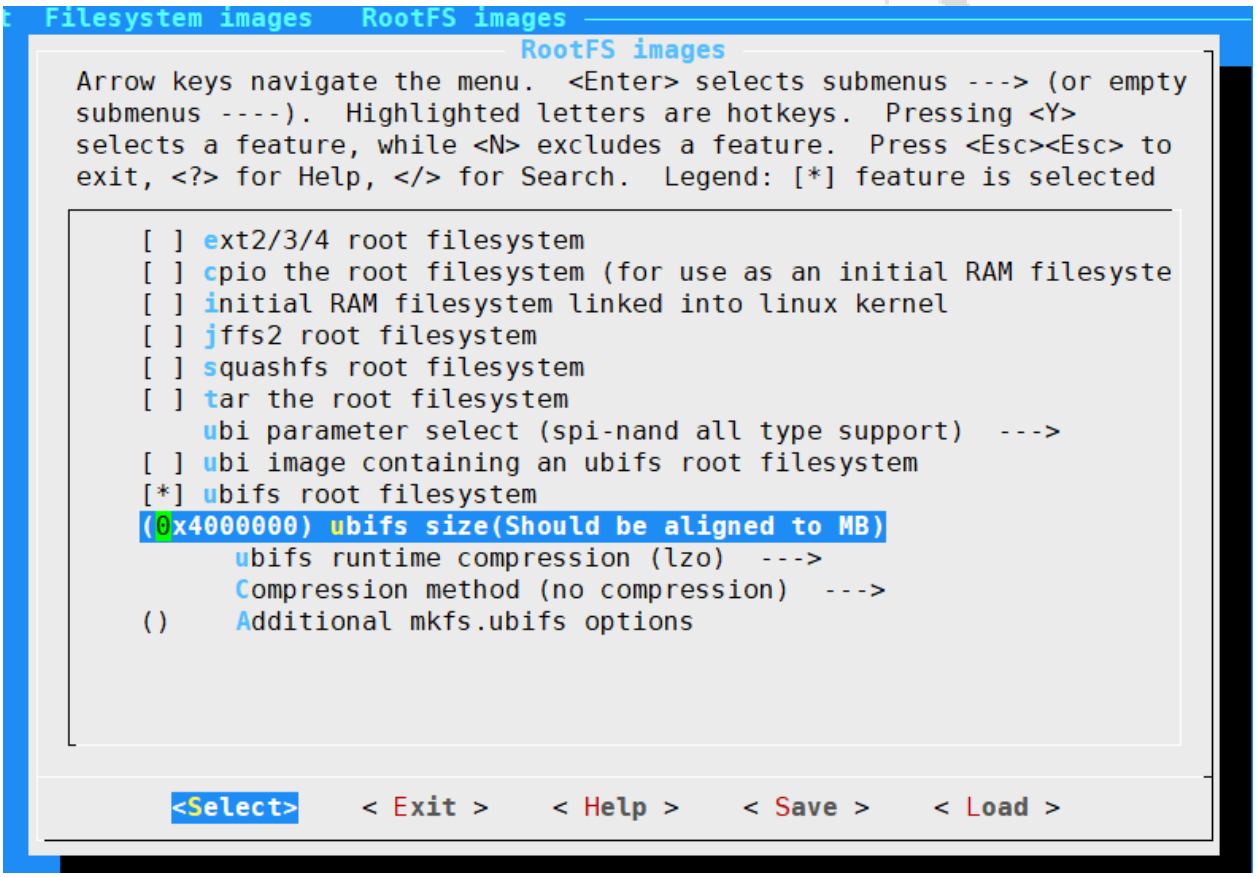
目前SDK中分区调整有两块工作，以 demo100_nand 为例：

1. 通过修改 `target/d211/demo100_nand/image_cfg.json` 调整

```
--- a/d211/demo100_nand/image_cfg.json
+++ b/d211/demo100_nand/image_cfg.json
@@ -13,7 +13,7 @@
                "kernel":   { "size": "12m" },
                "recovery": { "size": "10m" },
                "ubiroot":  {
-                       "size": "32m",
+                       "size": "64m", //由32m增加到64m
                        "ubi": { // Volume in UBI device
```

```
                        "rootfs": { "size": "-" },
                    },
@@ -21,7 +21,6 @@
                "ubisystem": {
                    "size": "-",
                    "ubi": { // Volume in UBI device
-                       "ota":  { "size": "48m" }, //删除ota分区，因为总大小只有128m
                        "user":  { "size": "-" },
                    },
                },
```

2. 通过 make menuconfig 调整，在 Filesystem images->RootFS images 中修改，ubifs size 调整为0x4000000（64m）



3. 修改后target/configs/d211_demo100_nand_defconfig 中有体现

```
--- a/configs/d211_demo100_nand_defconfig
+++ b/configs/d211_demo100_nand_defconfig
-BR2_TARGET_ROOTFS_UBIFS_MAX_SIZE=0x2000000
+BR2_TARGET_ROOTFS_UBIFS_MAX_SIZE=0x4000000
```

4. 上述两项修改后，重新编译系统，刷机后应该有 qtlauncher 运行界面

1.5.2. 应用开发示例

以 QTLauncher 为例，讲解一下 Luban 的 QT 应用开发方式：

1. 搭建 QtCreator，具体步骤可参考 QT Windows IDE。
2. 使用 QtCreator 打开 `source/artinchip/qtlauncher/QtLauncher.pro` 文件即可编译、调试和运行 qtlauncher 应用。

   推荐使用 QtCreator 在 Windows 上开发，可以单步调试和模拟。
3. QtCreator 会创建一个 Project 文件来组织代码编译和最终应用程序的生成，Luban 也要依赖该 Project 文件进行交叉编译。

   QtLauncher 的源码和 Project（QtLauncher.pro）均在 source/artinchip/qtlauncher 目录下

   编译文件是 Luban 用来描述代码如何编译，文件如何安装，宏定义如何使用的机制，在 package/artinchip/qtlauncher 目录下，qtlauncher 中总共有三个文件

   - `Config.in`：配置是否打开该模块，配置其他宏定义等，宏的定义方式为 BR2_PACKAGE_PACKAGENAME 大写

     该 Config.in 文件被上一级 Config.in 文件引用

```
menuconfig BR2_PACKAGE_QTLAUNCHER
    bool "qtlauncher"
    default n
    select BR2_PACKAGE_QT
    select BR2_PACKAGE_DIRECTFB
    help
    ArtInChip's Launcher App Developed with QT.

if BR2_PACKAGE_QTLAUNCHER
config BR2_QTLAUNCHER_GE_SUPPORT
    bool "use GE to render image"
    default y
config BR2_QTLAUNCHER_SMALL_MEMORY
    bool "small memory device"
    default y
endif
```

   - qtlauncher.mk：mk 文件，用于描述如何使用source/artinchip/qtlauncher 下的源码

```
QTLAUNCHER_ENABLE_TARBALL = NO   //源码方式
QTLAUNCHER_ENABLE_PATCH = NO    //是否有patch

QTLAUNCHER_DEPENDENCIES += qt directfb  //需要qt 和 directfb的支持

QTLAUNCHER_CONF_OPTS = $(QTLAUNCHER_SRCDIR)/QtLauncher.pro //project 文件的名字

ifeq ($(BR2_QTLAUNCHER_GE_SUPPORT),y)
export QTLAUNCHER_GE_SUPPORT = YES  //Luban 的宏转为 QT的宏
endif
ifeq ($(BR2_QTLAUNCHER_SMALL_MEMORY),y)
export QTLAUNCHER_SMALL_MEMORY = YES
endif

define QTLAUNCHER_INSTALL_TARGET_CMDS、、
   mkdir -p $(TARGET_DIR)/usr/local/launcher/
   cp -a $(@D)/qtlauncher $(TARGET_DIR)/usr/local/launcher/  //安装编译后的文件

   $(INSTALL) -m 0755 -D package/artinchip/qtlauncher/S99qtlauncher \
      $(TARGET_DIR)/etc/init.d/S99qtlaunche  //安装自动启动脚本
endef

$(eval $(qmake-package)) //使用qmake编译源码
```

- S99qtlauncher：qtlauncher 应用的自启动脚本，S99 为启动顺序，属于比较低的等级，S00 为最高等级。

1.6. Directfb Cross Compile

Directfb的编译要依赖于libz，libpng，freetype 等，pkgconfig虽然不是强依赖，但其提供的强大的包管理能力， 会让后续的编译容易很多，因此强烈建议编译。

1. 设置环境变量

   Directfb及其相关的依赖的编译采用Makefile的方式，因此环境变量的设置采用通用做法

```
export PRFIX=/xxx/QT/install
export CROSS_COMPILE=riscv64-unknown-linux-gnu
export PATH=/xxx/toolchain/d211/bin/:$PATH
export PKG_CONFIG_PATH="${PRFIX}/lib/pkgconfig"


export ARCH=riscv
export AS=riscv64-unknown-linux-gnu-as
export LD=riscv64-unknown-linux-gnu-ld
export CC=riscv64-unknown-linux-gnu-gcc
export GCC=riscv64-unknown-linux-gnu-gcc
export CPP=riscv64-unknown-linux-gnu-cpp
export CXX=riscv64-unknown-linux-gnu-g++
export RANLIB=riscv64-unknown-linux-gnu-ranlib
export NM=riscv64-unknown-linux-gnu-nm
export STRIP=riscv64-unknown-linux-gnu-strip
export OBJCOPY=riscv64-unknown-linux-gnu-objcopy
export OBJDUMP=riscv64-unknown-linux-gnu-objdump


export CPPFLAGS="-I${PRFIX}/include"
export CFLAGS="-I${PRFIX}/include"
export LDFLAGS="-L${PRFIX}/lib"
export LIBPNG_LIBS=-lpng16
```

2. 编译pkg config

```
./configure --host=riscv64-unknown-linux-gnu  --prefix=$PRFIX
make
make install
```

3. 编译zlib

```
cd zlib-1.2.11
./configure --prefix=$PRFIX
make
make install
```

4. 编译libpng

   本次使用的libpng版本是1.6，而directfb自己的代码的比较老，因此需要通过上面的LIBPNG_LIBS=-lpng16 指定一下png的版本

```
cd libpng-1.6.37
./configure --target=riscv64-unknown-linux-gnu --host=riscv64-unknown-linux-gnu  --prefix=$PRFIX
make
make install
```

5. 编译freetype

   freetype是一个字体库引擎，需要libz进行自体库的解码，需要libpng进行自体的渲染，但默认支持的是png1.2版本，因此需要LIBPNG_LIBS=-lpng16 指定使用1.6版本

```
cd freetype-2.10.4
export LIBPNG_LIBS=-lpng16
./configure --target=riscv64-unknown-linux-gnu --host=riscv64-linux-gnu  --prefix=$PRFIX
make
make install
```

6. 编译directcb

```
cd DirectFB-1.7.7
export LIBS=-lz
./configure  --host=riscv64-unknown-linux-gnu  --prefix=$PRFIX --disable-gtk-doc \
--disable-gtk-doc-html --disable-docs --disable-documentation \
--with-xmlto=no --with-fop=no --disable-dependency-tracking --enable-ipv6 \
--disable-nls --enable-static --enable-shared --enable-zlib --enable-freetype \
--enable-fbdev --disable-sdl --disable-vnc --disable-osx --disable-video4linux \
--disable-video4linux2 --without-tools --disable-x11 --disable-multi \
--disable-multi-kernel --enable-debug-support --disable-divine --disable-sawman \
--with-gfxdrivers=none --with-inputdrivers=none --disable-gif --disable-tiff \
--disable-png --disable-jpeg --disable-svg --disable-imlib2 --with-dither-rgb16=none
```

1.7. QT + Directfb + GE

GE (Graphics Engine) 是一个用来进行2D图形加速的硬件模块。主要包括格式转换、旋转、 镜像、缩放、Alpha混合、Color Key、位块搬移、Dither等功能， 以下特性应该会被Directfb使用的到。

- 支持水平和垂直Flip
- 所有格式支持90/180/270度旋转
- RGB格式支持任意角度旋转
- 支持1/16x ~ 16x缩放
- 支持porter-duff规则的Alpha混合
- 支持矩形填充
- 位块搬移(bit block transfer)

1.7.1. 功能移植

因为平台的差异，需要在Directfb代码中注入如下信息来支持：

- dmabuf surface支持
- fbdev接口
- gfxdrivers接口
- gfx逻辑

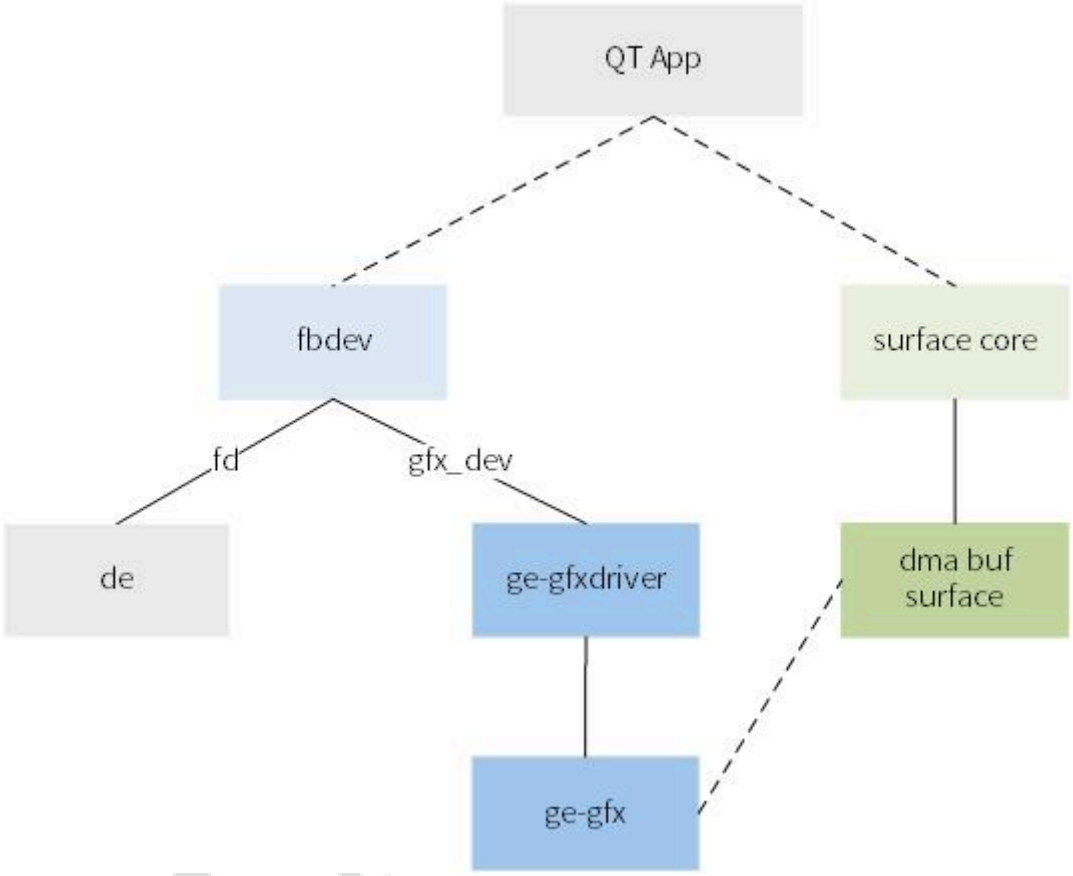


图 1–9 移植架构图

1.7.2. 模块添加

需要在Directfb中添加对ge模块的编译，修改对象为“configure.in” 文件，具体方法是参考某一个模块仿写， 譬如：vmware， 在相应的地方添加ge 的check 为 “yes” 。

```
@@ –1925,6 +1925,7 @@ sis315=no
vdpau=no
+ge=yes
vmware=no

if test "$have_linux" = "yes"; then
@@ –1938,7 +1939,7 @@ AC_ARG_WITH(gfxdrivers,
                          [radeon, savage, sh772x, sis315, tdfx, unichrome,]
–                         [vdpau, vmware. @<:@default=all@:>@]),
+                         [vdpau, ge, vmware. @<:@default=all@:>@]),
          [gfxdrivers="$withval"], [gfxdrivers=all])
@@ –1966,6 +1967,7 @@ if test "$gfxdrivers" = "all"; then
checkfor_vdpau=yes
+  checkfor_ge=yes
checkfor_vmware=no

              checkfor_vdpau=yes
              ;;
+         ge)
+              checkfor_ge=yes
+              ;;
       *)
```

1.7.3. dma buf

AIC GE 需要连续的内存作为操作buf，因此需要在surface pool 中添加 aic的操作接口

1. 代码修改

```
modified:   src/core/local_surface_pool.c
    modified:   src/core/prealloc_surface_pool.c
    modified:   src/core/surface_buffer.h
    modified:   src/core/surface_core.c
add:        src/core/dmabuf_surface_pool.c
```

2. 核心代码

   - 添加width 和 height 变量的使用
   - 实现dmabufSurfacePoolFuncs的函数

```
const SurfacePoolFuncs dmabufSurfacePoolFuncs = {
   .PoolDataSize       = dmabufPoolDataSize,
   .PoolLocalDataSize  = dmabufPoolLocalDataSize,
   .AllocationDataSize = dmabufAllocationDataSize,

   .InitPool           = dmabufInitPool,
   .JoinPool           = dmabufJoinPool,
   .DestroyPool        = dmabufDestroyPool,
   .LeavePool          = dmabufLeavePool,

   .AllocateBuffer     = dmabufAllocateBuffer,
   .DeallocateBuffer   = dmabufDeallocateBuffer,

   .Lock               = dmabufLock,
   .Unlock             = dmabufUnlock,
};
```

1.7.4. fbdev

1. 代码修改

```
modified:   systems/fbdev/fb.h
modified:   systems/fbdev/fbdev.c
modified:   systems/fbdev/fbdev.h
modified:   systems/fbdev/fbdev_surface_pool.c
```

2. 核心代码

   - 添加width 和 height 变量的使用
   - 添加gfxdriver的接口变量

```
--- a/systems/fbdev/fbdev_surface_pool.c
+++ b/systems/fbdev/fbdev_surface_pool.c
@@ -66,6 +66,8 @@ typedef struct {
typedef struct {
   int    magic;

+  int    width;
+  int    height;
   Chunk *chunk;
} FBDevAllocationData;

@@ -278,6 +280,9 @@ fbdevAllocateBuffer( CoreSurfacePool       *pool,
   surface = buffer->surface;
   D_MAGIC_ASSERT( surface, CoreSurface );

+  alloc->width = buffer->config.size.w;
+  alloc->height = buffer->config.size.h;
```

1.7.5. gfxdriver

gfxdirver是directfb 集成GE的接口，主要实现 ge_driver 和 ge_device

- gfxdriver 是模拟 Linux 的 driver 的方式，向 QT 注册自己

```
driver_init_device
driver_init_driver
driver_probe
```

- **功能声明**

```
#define GE_SUPPORTED_DRAWINGFLAGS (DSDRAW_BLEND)
#define GE_SUPPORTED_BLITTINGFLAGS (DSBLIT_BLEND_ALPHACHANNEL |\
                                    DSBLIT_BLEND_COLORALPHA   |\
                                    DSBLIT_SRC_COLORKEY       |\
                                    DSBLIT_ROTATE90           |\
                                    DSBLIT_ROTATE180          |\
                                    DSBLIT_ROTATE270          |\
                                    DSBLIT_FLIP_HORIZONTAL    |\
                                    DSBLIT_FLIP_VERTICAL)
```

```
#define GE_SUPPORTED_BLITTINGFUNCTIONS  (DFXL_BLIT | \
                                         DFXL_STRETCHBLIT)

#define GE_SUPPORTED_DRAWINGFUNCTIONS (DFXL_FILLRECTANGLE)
device_info->caps.accel    = GE_SUPPORTED_DRAWINGFUNCTIONS |
                             GE_SUPPORTED_BLITTINGFUNCTIONS;

device_info->caps.drawing  = GE_SUPPORTED_DRAWINGFLAGS;
device_info->caps.blitting = GE_SUPPORTED_BLITTINGFLAGS;
```

- **功能函数**

  通过src/core/gfxcard.h 中的 GraphicsDeviceFuncs 实现

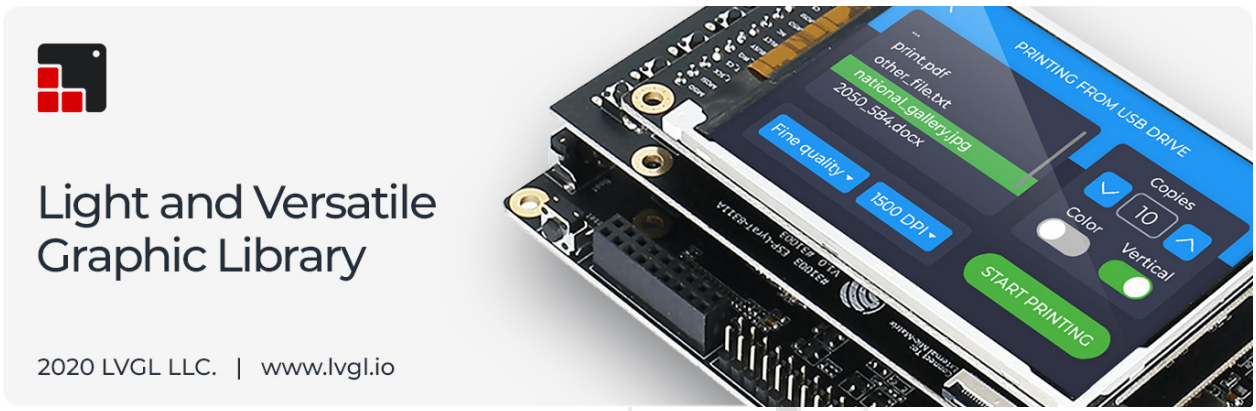
```
funcs->CheckState    = ge_check_state;
funcs->SetState      = ge_set_state;
funcs->EngineSync    = ge_sync;
funcs->EngineReset   = ge_reset;
funcs->FlushTextureCache  = ge_flush_texture_cache;

funcs->FillRectangle = ge_fill_rectangle;
funcs->Blit          = ge_blit;
funcs->StretchBlit   = ge_stretch_blit;
```

1.7.6. GFX

GFX driver 是驱动，是和DirectFB的接口描述，真正的实现是通过调用 aic_mpp 库

```
--- a/src/gfx/Makefile.am
+++ b/src/gfx/Makefile.am
+libdirectfb_gfx_la_LDFLAGS = -lmpp_decoder -lmpp_ge
```

2. LVGL



LVGL(轻量级和通用图形库)是一个免费和开源的图形库，它提供了创建嵌入式 GUI 所需的一切，具有易于使的图形元素，美丽的视觉效果和低内存占用

主要特性

- 丰富且强大的模块化图形组件：按钮、图标、列表、互动条、图片等
- 先进的图形界面：动画、抗锯齿、透明度、平滑滚动等效果
- 支持不同的输入设备包括键盘，鼠标，触摸屏，编码器等
- UTF-8编码支持多语言
- 多显示器支持，可以同时使用多个TFT或单色显示
- 可以通过类 CSS的方式来设计、布局图形界面
- 不限制芯片类型、硬件，可在各种微控制器或显示器上使用LVGL
- 配置可裁剪（最低资源占用：64 KB Flash，16 KB RAM）
- 支持操作系统、外部存储和GPU，但都不是硬性要求
- 即使单缓冲区(frame buffer)也能实现高级图形效果
- 不需要嵌入式硬件环境在PC模拟器就可以调试GUI
- 支持 Micropython 编程
- 有用于快速GUI设计的教程、示例、主题
- 详尽的文档以及 API 参考手册，可线上查阅或可下载为 PDF 格式
- 在 MIT 许可下免费和开源

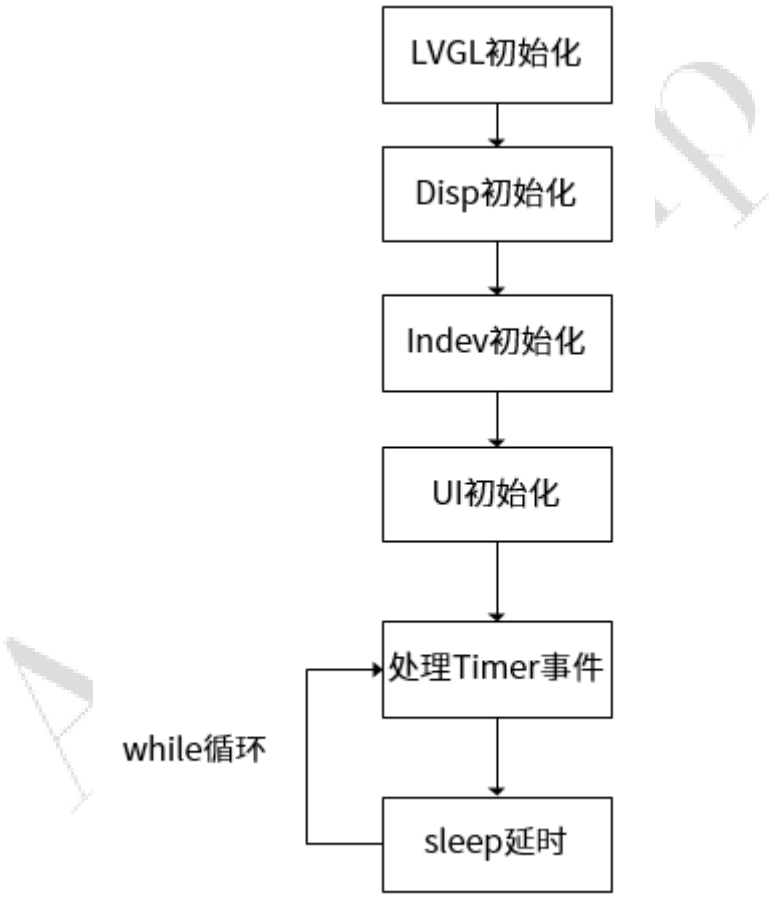
配置要求

- 16、32或64位微控制器或处理器
- 最低 16 MHz 时钟频率
- Flash/ROM: >64 KB(建议 180 KB)
- RAM: 8 KB(建议 24 KB)
- 显示缓冲区: >水平分辨率像素(建议为1/10屏幕大小)
- 支持C99编程
- 具备基本的C或C++知识

2.1. 使用指南

代码目录：

```
source/artinchip/lvgl-ui
 ├─── lvgl          // lvgl库
 ├─── lv_drivers    // lvgl平台适配
 ├─── base_ui       // base_ui测试用例
 ├─── lv_conf.h     // lvgl配置文件
 └─── main.c        // lvgl应用入口
```



LVGL框架的运行都是基于LVGL中定义的Timer定时器，系统需要给LVGL一个“心跳”， LVGL才能正常的运转起来。LVGL 整体流程的两个关键函数：

- lv_tick_get(), 获取以ms为单位的tick时间

  文件lv_hal_tick.c中的lv_tick_get的实现代码如下：

```
uint32_t lv_tick_get(void)
{
#if LV_TICK_CUSTOM == 0

    /*If `lv_tick_inc` is called from an interrupt while `sys_time` is read
    *the result might be corrupted.
    *This loop detects if `lv_tick_inc` was called while reading `sys_time`.
    *If `tick_irq_flag` was cleared in `lv_tick_inc` try to read again
    *until `tick_irq_flag` remains `1`.*/
    uint32_t result;
    do {
        tick_irq_flag = 1;
        result        = sys_time;
    } while(!tick_irq_flag); /*Continue until see a non interrupted cycle*/

    return result;
#else
    return LV_TICK_CUSTOM_SYS_TIME_EXPR;
#endif
}
```

- lv_timer_handler()，在while循环中的基于定时器的任务处理，函数lv_task_handler会调用lv_timer_handler, lv_tick_get决定了lv_timer_handler基于定时器的任务处理的时间的准确性。

头文件lv_conf.h中定义了上述函数中的LV_TICK_CUSTOM_SYS_TIME_EXPR

```
#define LV_TICK_CUSTOM 1
#if LV_TICK_CUSTOM
   #define LV_TICK_CUSTOM_INCLUDE <aic_ui.h>
   #define LV_TICK_CUSTOM_SYS_TIME_EXPR (custom_tick_get()) /*system time in ms*/
#endif   /*LV_TICK_CUSTOM*/
```

LVGL应用主函数代码如下所示：

```
#define IMG_CACHE_NUM 10

#if LV_USE_LOG
static void lv_user_log(const char *buf)
{
   printf("%s\n", buf);
}
#endif /* LV_USE_LOG */

int main(void)
{
#if LV_USE_LOG
   lv_log_register_print_cb(lv_user_log);
#endif /* LV_USE_LOG */

   /*LittlevGL init*/
   lv_init();

#if LV_IMG_CACHE_DEF_SIZE == 1
   lv_img_cache_set_size(IMG_CACHE_NUM);
#endif

   aic_dec_create();

   lv_port_disp_init();
   lv_port_indev_init();

   /*Create a Demo*/
#if LV_USE_DEMO_MUSIC == 1
   void lv_demo_music(void);
   lv_demo_music();
#else
   void base_ui_init();
   base_ui_init();
#endif

   /*Handle LitlevGL tasks (tickless mode)*/
   while (1) {
      lv_timer_handler();
      usleep(1000);
   }

   return 0;
}

/*Set in lv_conf.h as `LV_TICK_CUSTOM_SYS_TIME_EXPR`*/
uint32_t custom_tick_get(void)
{
   static uint64_t start_ms = 0;
   if (start_ms == 0) {
      struct timeval tv_start;
      gettimeofday(&tv_start, NULL);
      start_ms = (tv_start.tv_sec * 1000000 + tv_start.tv_usec) / 1000;
   }

   struct timeval tv_now;
   gettimeofday(&tv_now, NULL);
   uint64_t now_ms;
   now_ms = (tv_now.tv_sec * 1000000 + tv_now.tv_usec) / 1000;

   uint32_t time_ms = now_ms - start_ms;
   return time_ms;
}
```

- 其中在函数lv_port_disp_init()中实现显示接口的对接以及硬件2D加速的对接
- 在函数lv_port_indev_init()中实现触摸屏的对接

- 函数aic_dec_create()注册硬件解码器
- 用户只需替换base_ui_init()的实现来对接自己的应用

2.1.1. LVGL层次结构

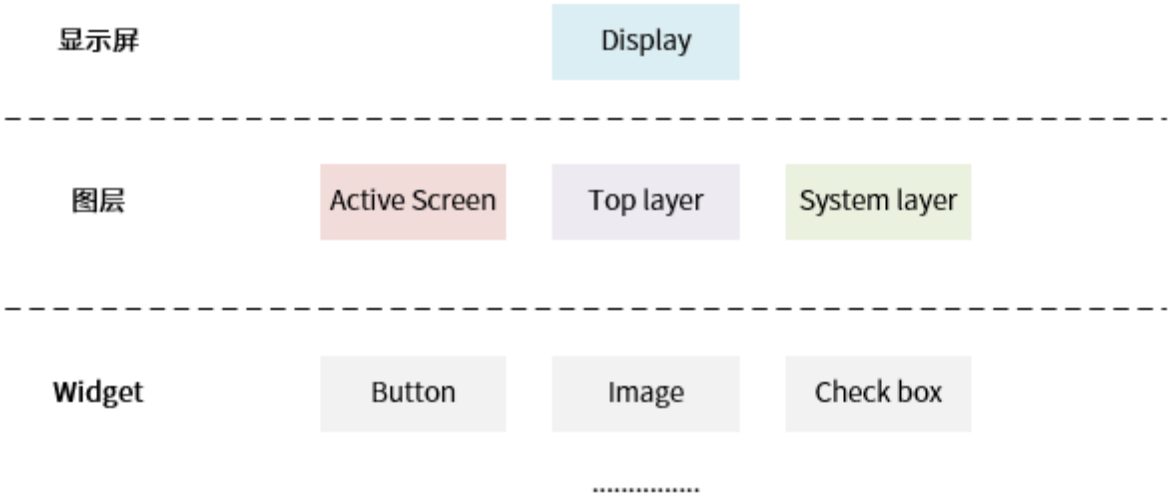


图 2–1 层次结构

- LVGL的display是对显示驱动的封装和抽象
- display包含Active Screen、Top layer、System layer
- Active Screen、Top layer、System layer是不同的screen对象，这里的screen用layer表达更准确一点， 表示的是图层的概念，其中Active Screen在最底层，System layer在最顶层
- 一般在Active Screen实现不同的app界面，用户可以创建多个screen，但只能有一个screen设置为Active Screen
- Top layer在Active Screen之上，可以用来创建弹出窗口，Top layer永远在Active Screen之上
- System layer在最顶层，比如鼠标可以在System layer，永远不会被遮挡

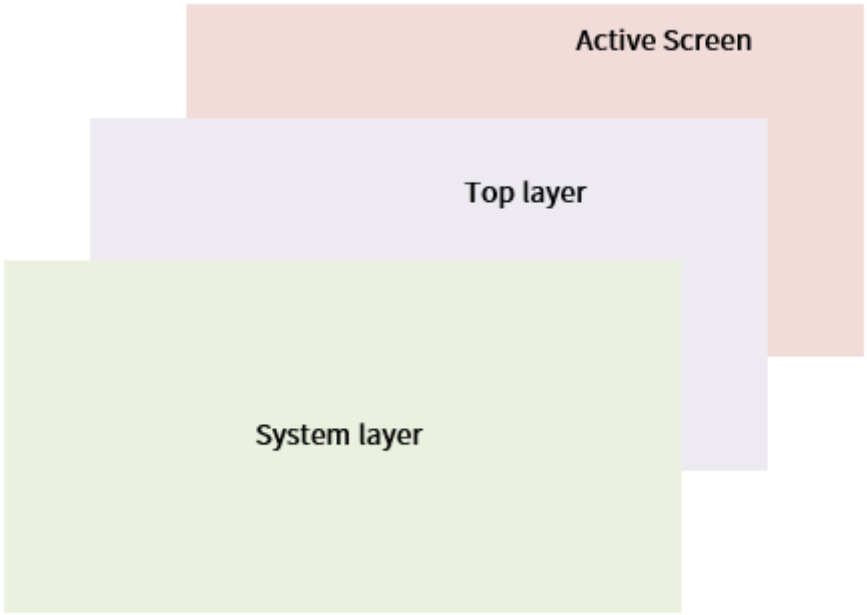


图 2–2 图层叠加

2.1.2. 父子结构

LVGL是面向对象的基于父子结构的设计，每一个对象都包含一个父对象（screen对象除外）， 但是一个父对象可以包含任意数量的子对象。

```
/*
 * 创建对象的时候，需要传入父对象的指针，
 * 如果父对象对NULL, 表示创建的是screen对象
 */
lv_obj_create(NULL);
```

• 父子对象一起移动

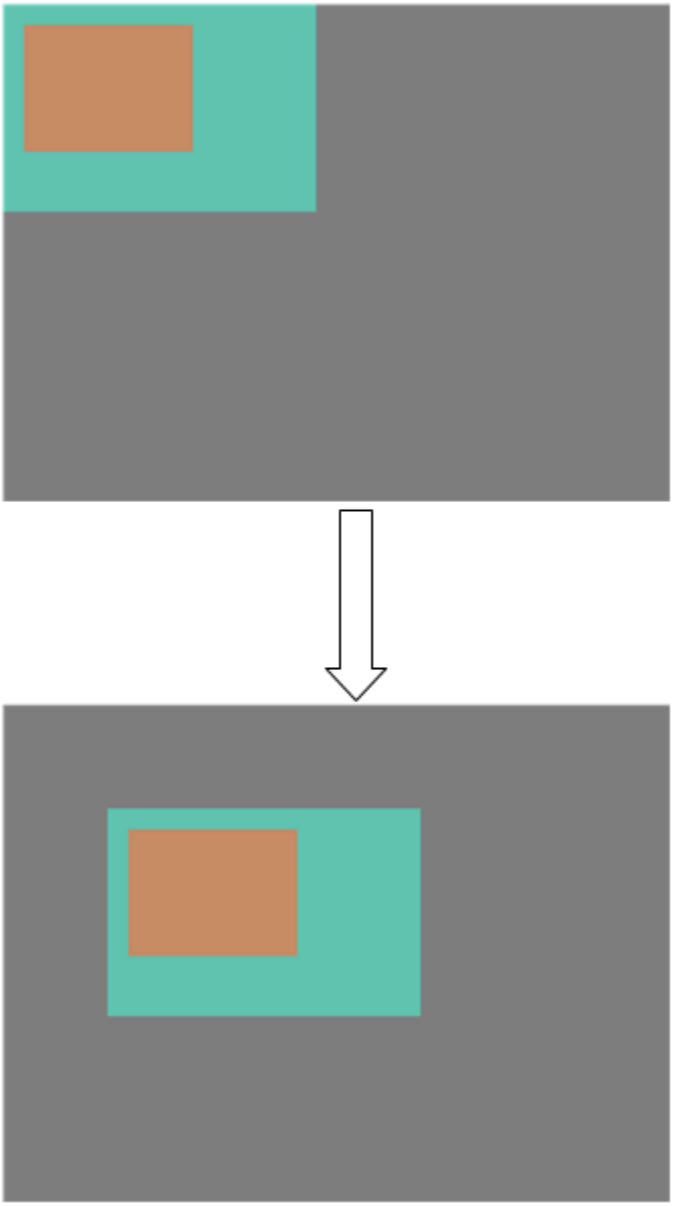


图 2–3 父子对象移动

• 子对象超出父对象部分不可见

图 2–4 子对象可见区域

2.1.3. 显示对接

主要包括三部分：

1. 绘制buffer初始化。函数如下：

```
void lv_disp_draw_buf_init(lv_disp_draw_buf_t * draw_buf, void * buf1, void * buf2, uint32_t size_in_px_cnt)
```

   • buf1：当为单缓冲或多缓冲的时候，都要设置此buffer

   • buf2：当选择双缓冲的时候，需要配置此buffer，单缓冲不需要

   • size_in_px_cnt： 以像素为单位的buf大小

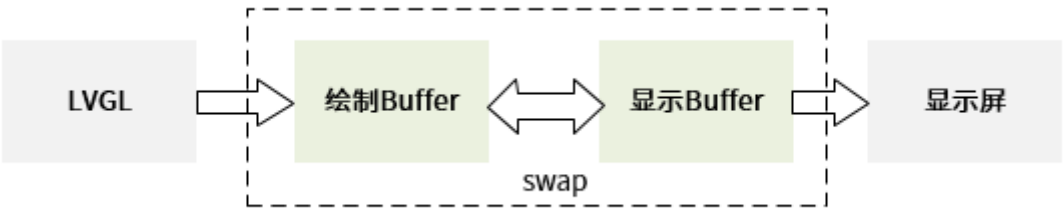


图 2–5 双缓冲

2. flush_cb对接

   下方以双缓冲为例说明 flush_cb回调函数的处理流程。绘制模式有full_refresh和direct_mode两种：

• 全刷新模式，每一帧都刷新整个显示屏

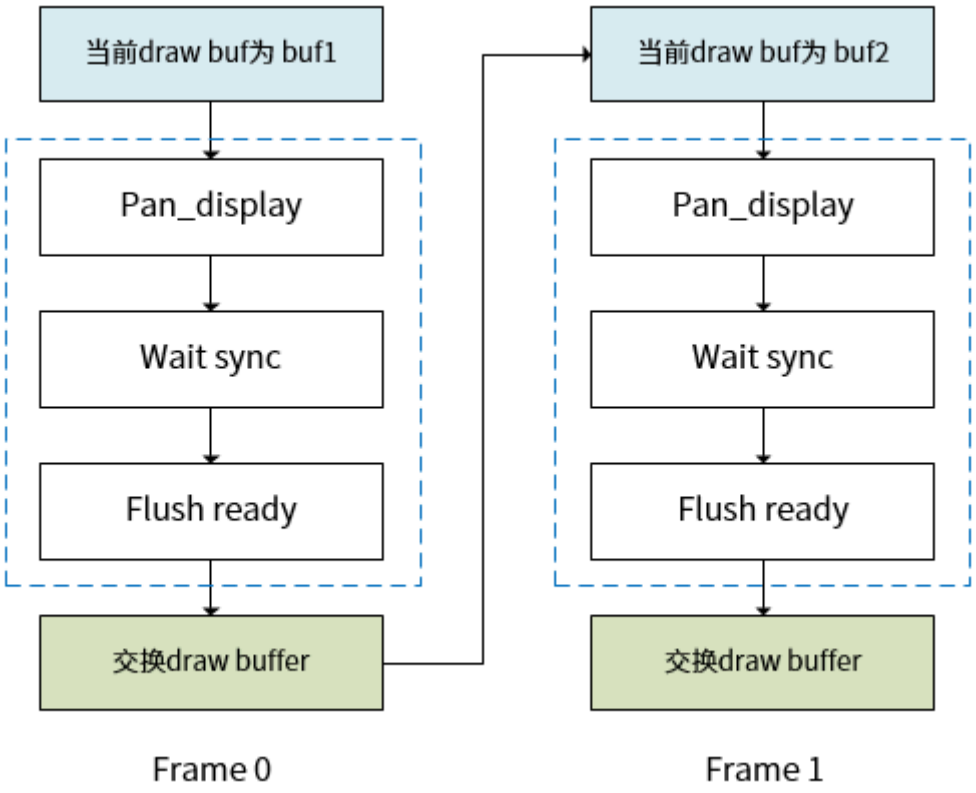


图 2–6 全刷新模式

在虚线框中为flush_cb中处理部分，在全刷新的流程中，直接通过 pan_display接口送当前绘制buffer到显示，然后等待vsync中断， 等到中断后，当前的绘制buffer就真正的在显示屏中显示出来，然后调用lv_disp_flush_ready通知LVGL框架已经flush结束， 最后在LVGL框架中会进行绘制buffer的交换。

• 局部刷新，每一帧只刷新需要更新的无效区域（可以有多个无效区域）

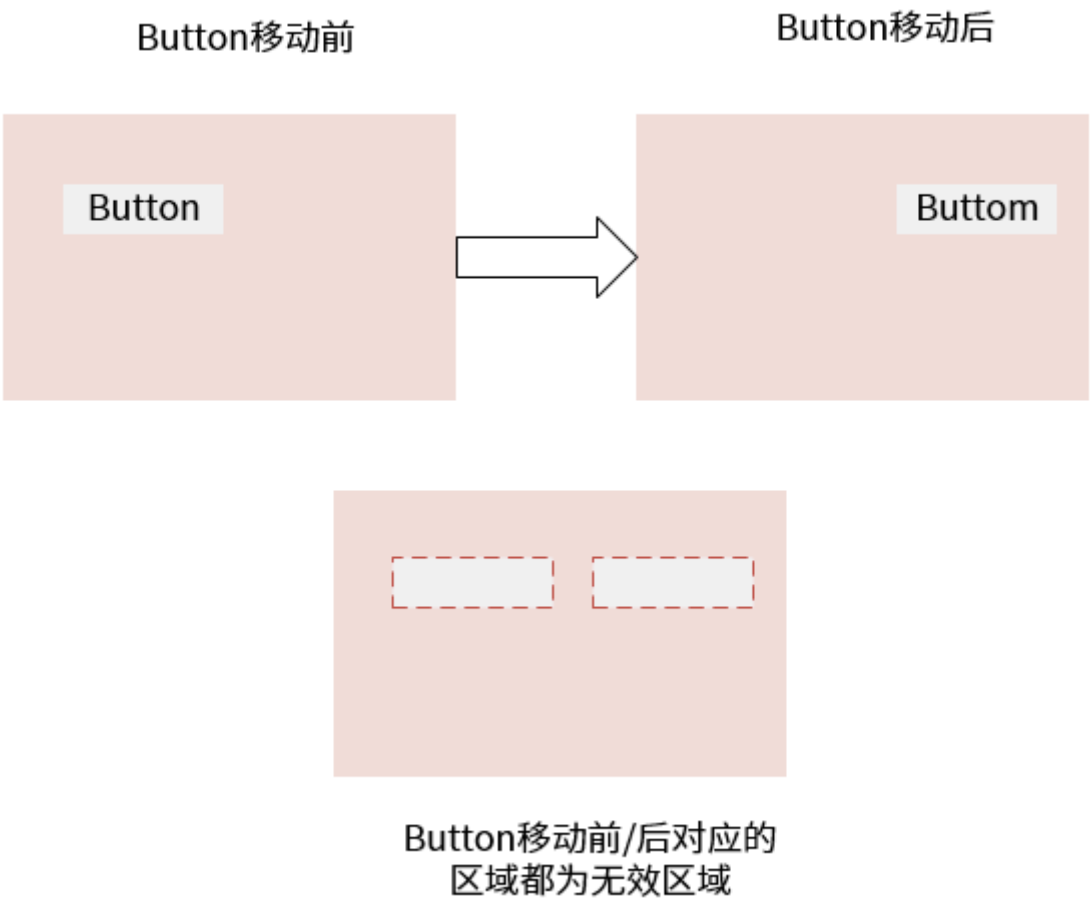


图 2–7 无效区域

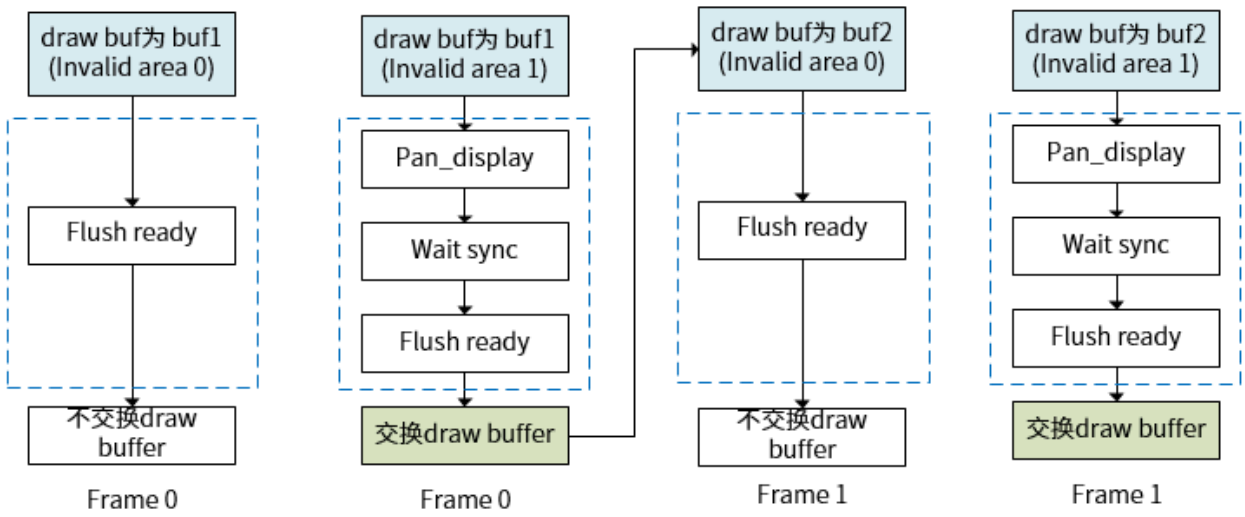


图 2–8 局部刷新模式

上图中的示例，为了方便描述每一帧都有两个无效区域（invalid area0 和invalid area1），LVGL可以支持更多的无效区域，到了最后一个无效区域， 说明当前帧的数据已经处理完，才把绘制buffer送显示，然后进行buffer交换

flush_cb的实现代码fbdev_flush如下：

```
static void fbdev_flush(lv_disp_drv_t * drv, const lv_area_t * area, lv_color_t *color_p)
{
  lv_disp_t * disp = _lv_refr_get_disp_refreshing();
  lv_disp_draw_buf_t * draw_buf = lv_disp_get_draw_buf(disp);
```

```
    if (!disp->driver->direct_mode || draw_buf->flushing_last) {
        struct fb_var_screeninfo var = {0};

        if (ioctl(g_fb, FBIOGET_VSCREENINFO, &var) < 0) {
            LV_LOG_WARN("ioctl FBIOGET_VSCREENINFO");
            return;
        }

        if (color_p == (lv_color_t *)g_frame_buf[0]) {
            var.xoffset = 0;
            var.yoffset = 0;
        } else {
            var.xoffset = 0;
            var.yoffset = disp_drv.ver_res;
        }

        if (ioctl(g_fb, FBIOPAN_DISPLAY, &var) == 0) {
            int zero = 0;
            if (ioctl(g_fb, AICFB_WAIT_FOR_VSYNC, &zero) < 0) {
                LV_LOG_WARN("ioctl AICFB_WAIT_FOR_VSYNC fail");
                return;
            }
        } else {
            LV_LOG_WARN("pan display err");
        }

        if (drv->direct_mode == 1) {
            for (int i = 0; i < disp->inv_p; i++) {
                if (disp->inv_area_joined[i] == 0) {
                    sync_disp_buf(drv, color_p, &disp->inv_areas[i]);
                }
            }
        }

        lv_disp_flush_ready(drv);
    } else {
        lv_disp_flush_ready(drv);
    }
}
```

3. 2D 硬件加速对接，主要对接 lv_draw_ctx_t中的绘制函数

成员	说明	是否硬件加速
void *buf	当前要绘制的buffer	–
const lv_area_t * clip_area	绘制区域裁剪（以屏幕为参考的绝对坐标）	–
void (*draw_rect)()	绘制矩形（包括圆角、阴影、渐变等）	否
void (*draw_arc)()	绘制弧形	否
void (*draw_img_decoded)()	绘制已经解码后的图像	是
lv_res_t (*draw_img)()	绘制图像（包括图片解码）	是
void (*draw_letter)()	绘制文字	否
void (*draw_line)()	绘制直线	否
void (*draw_polygon)()	绘制多边形	否

在lv_draw_aic_ctx_t（重定义了lv_draw_sw_ctx_t）结构体中包含lv_draw_ctx_t和blend函数：

```
typedef struct {
    lv_draw_ctx_t base_draw;

    /** Fill an area of the destination buffer with a color*/
    void (*blend)(lv_draw_ctx_t * draw_ctx, const lv_draw_sw_blend_dsc_t * dsc);
} lv_draw_sw_ctx_t;
```

在draw_rect、draw_line等操作的功能由多个步骤组成，虽然我们没有对这些接口进行硬件加速，但是这些操作的部分实现 会调用到blend，我们对blend接口进行了硬件加速对接：

```
void lv_draw_aic_ctx_init(lv_disp_drv_t * drv, lv_draw_ctx_t * draw_ctx)
{
    lv_draw_sw_init_ctx(drv, draw_ctx);
    lv_draw_aic_ctx_t * aic_draw_ctx = (lv_draw_aic_ctx_t *)draw_ctx;

    aic_draw_ctx->blend = lv_draw_aic_blend;
```

```
    aic_draw_ctx->base_draw.draw_img = lv_draw_aic_draw_img;
    aic_draw_ctx->base_draw.draw_img_decoded = lv_draw_aic_img_decoded;

    return;
}
```

先调用lv_draw_sw_init_ctx函数把所有绘制操作都初始化为软件实现，然后对可以硬件加速的接口重新实现， 覆盖原来的软件实现。

4. 所有的显示相关功能都包含在lv_disp_drv_t结构体中：
   a. 通过lv_disp_drv_init来初始化lv_disp_drv_t结构体
   b. 通过lv_disp_draw_buf_init初始化绘制buffer
   c. 通过回调flush_cb来注册显示接口
   d. 通过lv_draw_aic_ctx_init来注册2D硬件加速相关接口
   e. 通过lv_disp_drv_register来注册lv_disp_drv_t

在源文件lv_port_disp.c中的函数lv_port_disp_init配置刷新模式，局部刷新模式配置如下：

```
disp_drv.full_refresh = 0;
disp_drv.direct_mode = 1;
```

全刷新模式参数配置如下：

```
disp_drv.full_refresh = 1;
disp_drv.direct_mode = 0;
```

2.1.4. 硬件解码对接

2.1.4.1. lv_img_decoder_t注册

我们通过lv_img_decoder_t来注册硬件解码器接口，主要实现了三个接口：

函数	说明
aic_decoder_info	获取图片宽、高、图片格式信息
aic_decoder_open	申请解码输出buffer，硬件解码输出
aic_decoder_close	释放硬件解码资源（包括输出buffer）

注册解码器过程；

```
void aic_dec_create()
{
    lv_img_decoder_t *aic_dec = lv_img_decoder_create();

    /* init frame info lists */
    mpp_list_init(&buf_list);
    lv_img_decoder_set_info_cb(aic_dec, aic_decoder_info);
    lv_img_decoder_set_open_cb(aic_dec, aic_decoder_open);
    lv_img_decoder_set_close_cb(aic_dec, aic_decoder_close);
}
```

绘制函数draw_img_decoded需要的解码后数据，需要通过注册解码器回调去获取， 这是我们默认的图片处理流程：

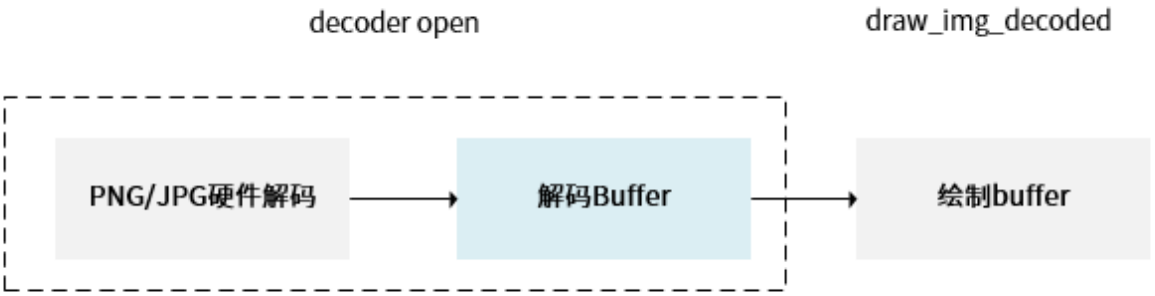


图 2–9 draw_img_decoded

- 采用此流程需要额外申请一块解码buffer，占用内存增加
- 缓存解码后的buffer，下次再显示同样的image，不用重复解码，加快UI加载速度

当绘制函数为draw_img的时候，硬件解码在函数draw_img内部，无需注册解码回调函数，我们默认不采用此方法， 当在内存受限的场景下，可以评估此方法是否可满足场景需求。

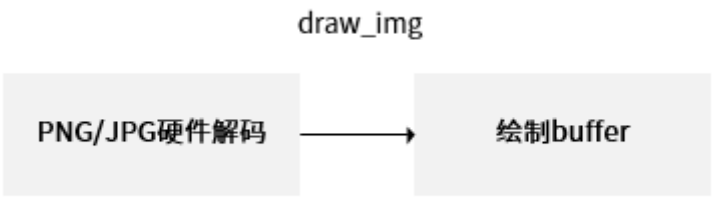


图 2–10 draw_img

- 采用此流程无需额外申请解码buffer，直接解码到绘制buffer
- 当需要进行alpha blending的时候，此方法不可行
- 每次都要重新对image解码，速度不如draw_img_decoded
- 当硬件解码不支持裁剪的时进行局部更新，此方法不可行

2.1.4.2. 图片cache机制

1. 采用lv_img_decoder_t提供的接口注册的解码器可以采用LVGL内部的图片缓冲机制， 在lv_conf.h 中宏定义 LV_IMG_CACHE_DEF_SIZE为1的时候，表示打开图片缓冲机制， 当LV_IMG_CACHE_DEF_SIZE为0的时候，图片缓冲机制关闭。
2. 通过void lv_img_cache_set_size(uint16_t entry_cnt)来设置缓冲的图片张数，图片以张数为单位进行缓存。
3. 当图片缓存到设置的最大张数的时候，如果需要新的缓存，图片缓存机制内部会进行图片缓存价值的判断， 例如：如果某一张图片解码的时间比较久，或者某一张图片使用的更频繁，那么这种图片的缓存价值打分会更高，优先缓存这些缓存价值更高的图片。

如果一些图片的读取时间或者解码时间比较长，采用图片缓存机制可以提升UI流畅性

2.1.5. LVGL库中demos使用

在目录luban/source/artinchip/lvgl-ui/lvgl/demos下lvgl官方提供了多个示例demo

1. 在lv_conf.h宏定义中打开#define LV_USE_DEMO_MUSIC 1， 则main.c中会调用相应的demo

```
  /*Create a Demo*/
#if LV_USE_DEMO_MUSIC == 1
```

```
  void lv_demo_music(void);
  lv_demo_music();
#else
  void base_ui_init();
  base_ui_init();
#endif
```

2. 如果要调用lvgl-ui/lvgl/demos下的benchmark，则需关闭LV_USE_DEMO_MUSIC， 打开宏定义#define LV_USE_DEMO_BENCHMARK 1， 修改main.c中的base_ui_init()为需要的demo入口函数即可, 如下所示：

```
  /*Create a Demo*/
#if LV_USE_DEMO_MUSIC == 1
  void lv_demo_music(void);
  lv_demo_music();
#else
  //void base_ui_init();
  //base_ui_init();
  void lv_demo_benchmark();
  lv_demo_benchmark();
#endif
```

2.1.6. LVGL库中samples使用

官方LVGL库lvgl-ui/lvgl/examples目录下是各种控件的测试用例， 以调用get_started目录下的lv_example_get_started_1.c为例进行流程说明：

1. samples相应的宏定义在lv_conf.h中已经默认打开：#define LV_BUILD_EXAMPLES 1
2. 修改main.c中的入口函数：

```
/*Create a Demo*/
#if LV_USE_DEMO_MUSIC == 1
  void lv_demo_music(void);
  lv_demo_music();
#else
  //void base_ui_init();
  //base_ui_init();
  void lv_example_get_started_1();
  lv_example_get_started_1();
#endif
```

2.1.7. 第三方库支持

- freetype库支持
  1. 选择freetype包，在 Luban根目录下执行 `make menuconfig`，进入 menuconfig

```
ArtInChip Luban SDK Configuration  --->
    Third-party packages  --->
    [*] freetype  --->
```

  2. 目录lvgl-ui/lvgl/examples/libs/freetype/lv_example_freetype_1.c下示例调用

     在lv_conf.h头文件中打开宏定义

```
#define LV_USE_FREETYPE 1
#define LV_BUILD_EXAMPLES 1
```

  3. 修改main.c中的入口函数：

```
/*Create a Demo*/
#if LV_USE_DEMO_MUSIC == 1
    void lv_demo_music(void);
    lv_demo_music();
#else
    //void base_ui_init();
    //base_ui_init();
    void lv_example_freetype_1(void);
    lv_example_freetype_1();
#endif
```

  4. 设置lvgl-ui/lvgl/examples/libs/freetype/Lato-Regular.ttf字体的打包目录， 复制字体到lvgl-ui/base_ui/asserts/font目录下，则会把字体打包到系统目录/usr/local/share/lvgl_data/font目录下

5. 修改代码lv_example_freetype_1.c中字体文件路径

```
void lv_example_freetype_1(void)
{
    /*Create a font*/
    static lv_ft_info_t info;
    /*FreeType uses C standard file system, so no driver letter is required.*/
    //info.name = "./lvgl/examples/libs/freetype/Lato-Regular.ttf";
    info.name = "/usr/local/share/lvgl_data/font/Lato-Regular.ttf";
    info.weight = 24;
    info.style = FT_FONT_STYLE_NORMAL;
    info.mem = NULL;
    if(!lv_ft_font_init(&info)) {
        LV_LOG_ERROR("create failed.");
    }

    /*Create style with the new font*/
    static lv_style_t style;
    lv_style_init(&style);
    lv_style_set_text_font(&style, info.font);
    lv_style_set_text_align(&style, LV_TEXT_ALIGN_CENTER);

    /*Create a label with the new style*/
    lv_obj_t * label = lv_label_create(lv_scr_act());
    lv_obj_add_style(label, &style, 0);
    lv_label_set_text(label, "Hello world\nI'm a font created with FreeType");
    lv_obj_center(label);
}
```

2.2. SquareLine Studio

SquareLine GUI是LVGL官网推荐的GUI 图形化辅助设计工具，可使用LVGL图形库开发UI，且支持多个平台， 如MacOS、Windows和Linux。在该工具中，我们通过拖放就可以在屏幕上添加和移动小控件，图像和字体的处理也变得十分简单， 但不具备编译调试代码的功能，界面设计完成后需要导出工程到其他IDE工具进行模拟和调试。 可以通过访问LVGL官网 http://lvgl.io 获取更多信息。

2.2.1. 导出工程

SquareLine工具自带了LVGL官方提供的一些demo示例，在学习控件用法时候可以查看这些demo示例， 以打开Futuristic_Ebike为例，从SquareLIne工具左上角File菜单选择new或open, 在弹出的选择对话框中选择example选项卡，然后双击Futuristic_Ebike图标

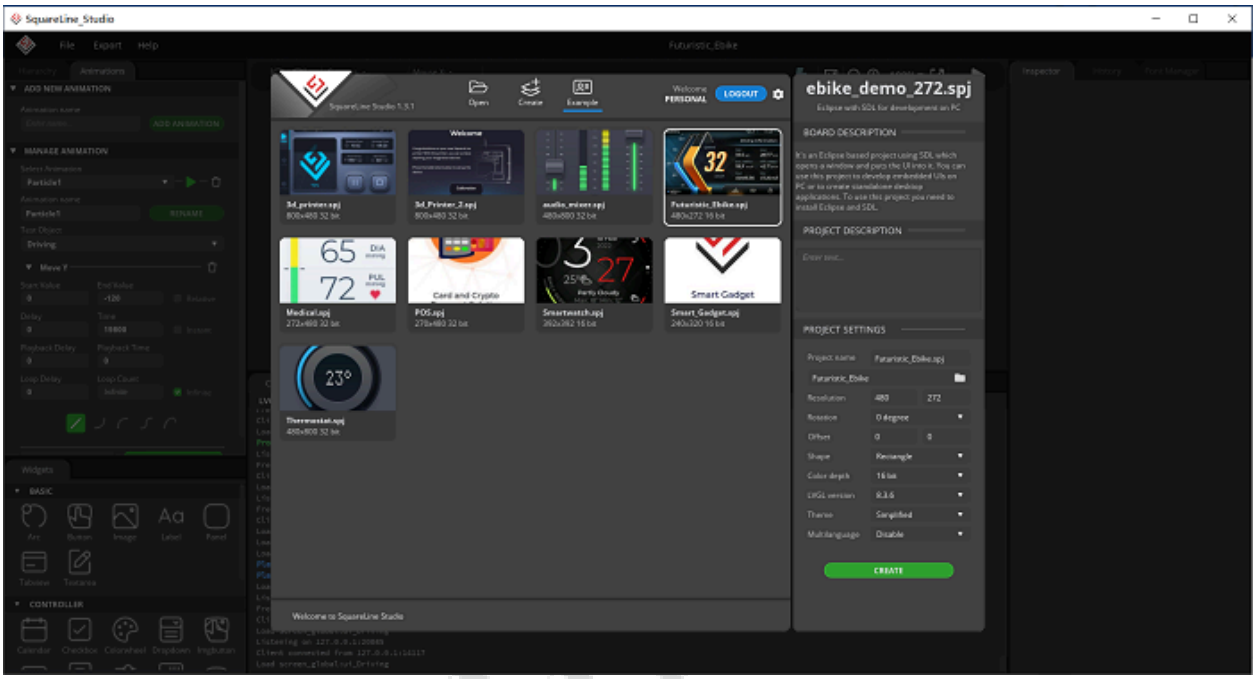


图 2–11 example打开示意图

Futuristic_Ebike工程打开后如下图所示，可以点击右上角的播放按钮，通过鼠标的滑动和点击事件来查看当前的UI效果， SquareLine 提供了widget拖拽功能，通过属性设置控件的一些行为动作，该工具还提供了便捷的动画设计功能。

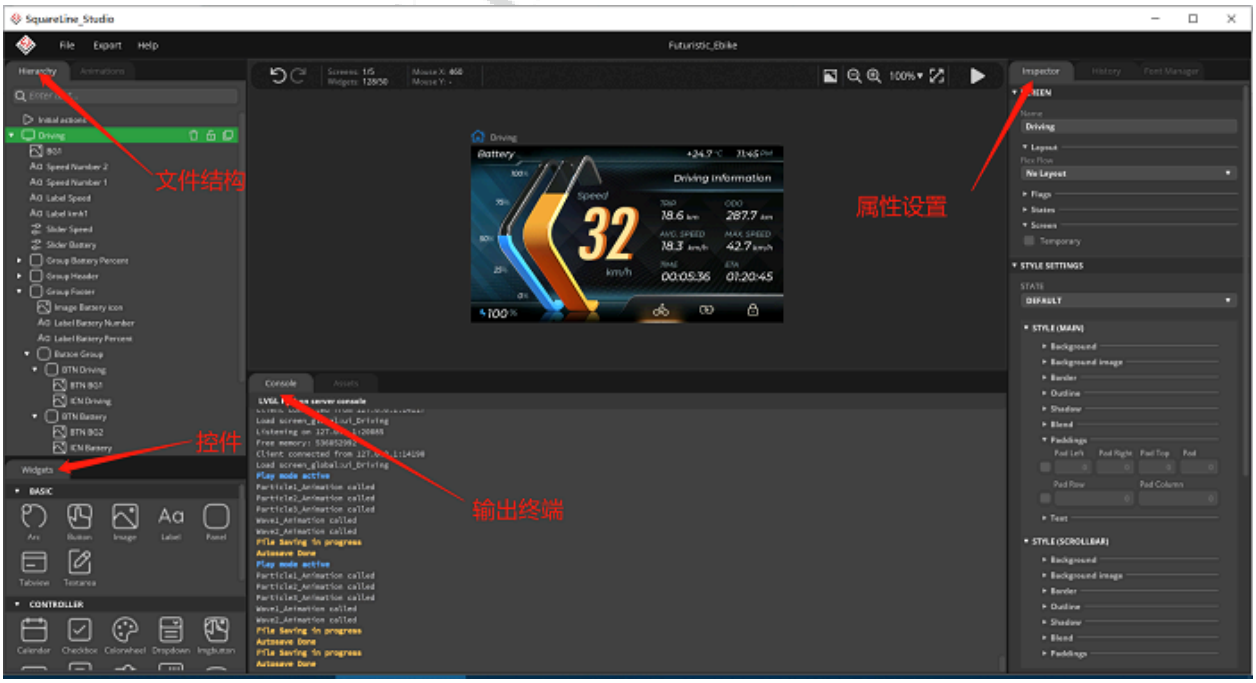


图 2–12 工程窗口示意图

UI效果调整或设计完成后，可以通过export导出功能导出UI文件或项目代码工程，UI文件是png资源图片通过SquareLine转换成C数组的源文件， 导出的项目工程是可以在相应的IDE工具进行模拟调试的工程，例如Eclipse, Vistal Studio, VsCode等。

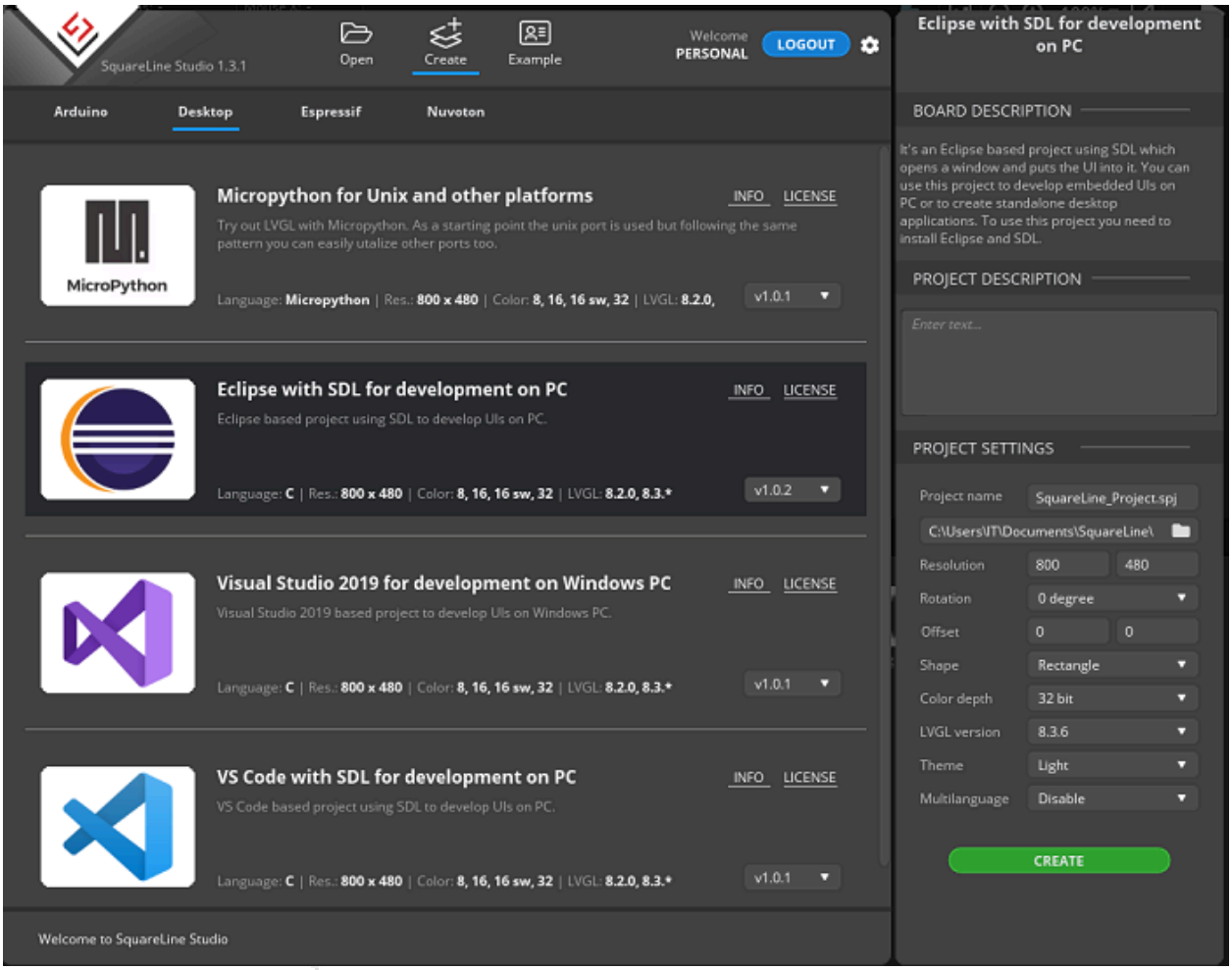


图 2–13 创建新工程示例图

图 2–14 导出工程目录结构

2.2.2. 编译导出的UI代码

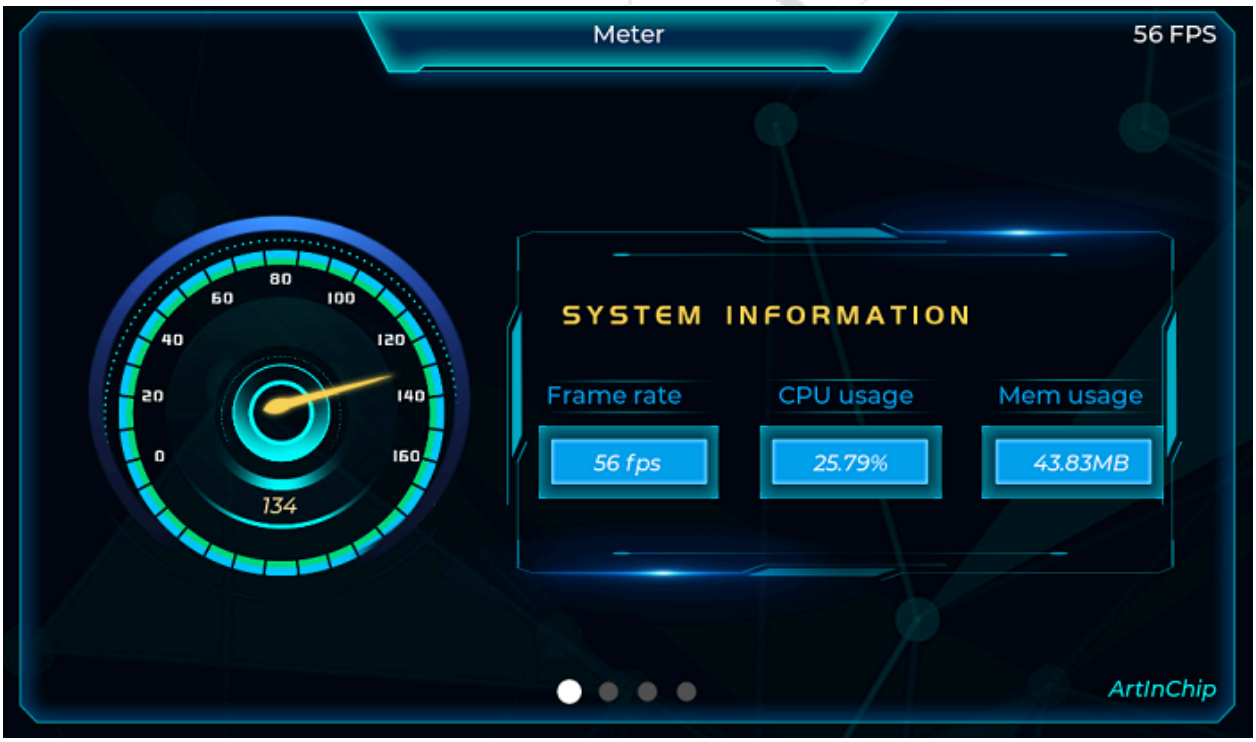
- 复制ui目录到sdk中的目录luban/source/artinchip/lvgl–ui下
- 修改luban/source/artinchip/lvgl–ui/CMakeLists.txt，中的app目录配置为ui

- 修改main.c中的入口函数：
- 重新编译lvgl–ui

> **注：**
>
> Squareline Studo生成的image图片都是build–in的图片，当图片比较大的时候会占用比较多的存储空间，可参考base_ui示例中的方法， 修改为从文件中读取png、jpg图片，从文件读取png、jpg图片，默认会使用硬件解码

2.3. lvgl–ui

lvgl–ui 是 ArtInChip 开发的一款用于演示 LVGL 基本操作的一个 demo，包含 png、jpg 硬件解码和 build–in 图片使用方式：



lvgl–ui 一共有4四个页面，功能包括：

- 仪表演示第二个页面为音乐播放演示、第三个页面为菜单演示、
- 音乐播放演示
- 图片菜单演示
- 播放器演示

播放器演示页面需要打开base_ui.c中的宏定义VIDEO_PLAYER，默认为打开状态

2.3.1. 打开lvgl–ui

在 Luban根目录下执行 `make menuconfig`，进入 menuconfig

```
ArtInChip Luban SDK Configuration  --->
    ArtInChip packages  --->
    [*] lvgl-ui  --->
```

编译选择lvgl–ui将生成lvgl库和相应的应用程序:test_lvgl

2.3.2. 功能选择

- 用户可以通过lv_conf.h中的宏定义去配置LVGL的功能参数
- 在lv_conf.h中至少需要有宏定义LV_COLOR_DEPTH，LV_COLOR_DEPTH的值可以是32或者16，分别表示argb8888格式和rgb565格式， LV_COLOR_DEPTH的设置需要和dts中framebuffer的格式设置保持一致
- 目前我们在lv_conf.h只是加入了最常用的宏定义，如果需要添加更多的宏定义可以参考lvgl库目录下lv_conf_template.h中的定义， 复制相关的宏定义到lv_conf.h即可

2.3.3. 运行

在目录luban/package/artinchip/lvgl-ui/S00lvgl下的启动脚本，编译后会打包到系统路径/etc/init.d/S00lvgl，开机自动运行lvgl_ui

2.3.4. 打印输出重定向

lvgl-ui 默认日志输出到 /dev/null，不进行显示，如果要进行调试可以在 S00lvgl 中进行重定向修改

```
# 屏蔽打印
PID=`$DAEMON > /dev/null 2>&1 & echo $!
# 打印在控制台输出
PID=`$DAEMON > /dev/stderr 2>&1 & echo $!
```

需要重新编译模块，才能生效

```
make clean
make lvgl-ui-rebuild
```

2.3.5. LVGL的打印宏

- 在lv_conf.h中打开宏定义 #define LV_USE_LOG 1
- 当打开LV_USE_LOG后，可以设置打印级别，默认打印级别设置为LV_LOG_LEVEL_WARN

2.3.6. 图片缓存开关

- 通过lv_conf.h中宏定义LV_IMG_CACHE_DEF_SIZE来控制是否缓存图片
- 通过main.c中的宏定义IMG_CACHE_NUM来控制缓存的图片的张数

2.3.7. 代码说明

- 界面滑动

  不同页面通过滑动操作切换，页面滑动使用了控件tabview

```
lv_obj_set_size(main_tabview, 1024, 600);
lv_obj_set_pos(main_tabview, 0, 0);
lv_obj_set_style_bg_opa(main_tabview, LV_OPA_0, 0);

lv_obj_t *main_tab0 = lv_tabview_add_tab(main_tabview, "main page 0");
lv_obj_t *main_tab1 = lv_tabview_add_tab(main_tabview, "main page 1");

lv_obj_set_style_bg_opa(main_tab0, LV_OPA_0, 0);
lv_obj_set_style_bg_opa(main_tab1, LV_OPA_0, 0);
lv_obj_set_size(main_tab0, 1024, 600);
lv_obj_set_size(main_tab1, 1024, 600);

lv_obj_set_pos(main_tab0, 0, 0);
lv_obj_set_pos(main_tab1, 0, 0);
```

- 背景图片

  背景图片通过image控件来创建，是一个名字为global_bg.png的png图片，此图片会采用注册的硬件解码器进行解码

```
static lv_obj_t *img_bg = NULL;
img_bg = lv_img_create(lv_scr_act());
```

```
lv_img_set_src(img_bg, LVGL_PATH(global_bg.png));
lv_obj_set_pos(img_bg, 0, 0);
```

- 菜单图片

  菜单图片也通过image控件来创建，是png图片，此图片也会采用注册的硬件解码器进行解码

```
lv_obj_t *sub_image00 = lv_img_create(sub_tab0);
lv_img_set_src(sub_image00, LVGL_PATH(cook_0.jpg));
lv_obj_set_pos(sub_image00, 36, 100);
```

- fake image

  fake image不是一个真实的图片，通过此方式可以方便的对一个矩形区域进行填充：包括alpha、red、green、blue

```
static lv_obj_t *img_bg = NULL;
FAKE_IMAGE_DECLARE(bg_dark)  //声明（bg_dark名字可修改）

/* 最后一个参数为要设置的颜色值：bit31:24 为alpha */
FAKE_IMAGE_INIT(bg_dark, 1024, 600, 0, 0x00000000);

lv_img_set_src(img_bg, FAKE_IMAGE_NAME(bg_dark)); //设置fake image数据源
```

- build-in image

  build-in image是通过数组变量在程序中表示图像，图片转换成.c文件的工具参考官网：http://lvgl.io/tools/imageconverter

```
uint8_t circle_white_map[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x50, 0xff, 0x7f, 0xff,
        ....................................................};

const lv_img_dsc_t circle_white = {
    .header.cf = LV_IMG_CF_TRUE_COLOR_ALPHA,
    .header.always_zero = 0,
    .header.reserved = 0,
    .header.w = 20,
    .header.h = 20,
    .data_size = 400 * LV_IMG_PX_SIZE_ALPHA_BYTE,
    .data = circle_white_map,
};

static lv_obj_t * circle_0 = lv_img_create(img_bg);
lv_img_set_src(circle_0, &circle_white);
lv_obj_align(circle_0, LV_ALIGN_BOTTOM_MID, -16, -28);
```