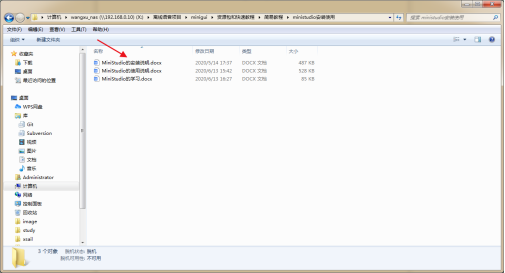
Minigui教程

前言

本文档内容意在带您快速了解minigui相关内容，您无需深究code也能够迅速了解各接口的使用逻辑。其余具体内容则需要您在后续工作中逐步体会。文档内容大致分为 minigui简易架构介绍、常用功能接口、离线语音项目中的GUI框架和逻辑。

# 搭建工作环境

本内容请移步指相应的介绍文档，如ministudio的安装、使用等。目前相应内容放在个人Nas盘中。如图：



# **二、Minigui部分目录结构**

温馨提示：最初您只需要重点关注src的内容即可。

\* doc  
 \* 通过 Doxygen 自动生成的 API 参考手册，包括 HTML（doc/html）、man（doc/man）和 Latex（doc/latex）、RTF（doc/rtf）格式。  
 \* 发表在 IBM developerWorks 中国站点上关于 MiniGUI 的文章（doc/articles）。  
\* etc  
 MiniGUI 的配置文件。目前有用于两个风格的配置文件。

\* examples  
 Doxygen 使用的一些示例代码片段。出现在最终的 API 参考手册中。

\* ext  
 MiniGUIExt 库的源代码  
 \* ext/control：MonthCalendar、SpinBox、CoolBar、ListView、TreeView 等控件。  
 \* ext/image：动画 GIF 格式支持代码。  
 \* ext/include：MiniGUIExt 库的外部头文件。  
 \* ext/res：MiniGUIExt 库使用的一些资源文件。

\* include  
 MiniGUI 库、MyWins 库、VCOnGUI 库的外部头文件。

\* m4  
 Autoconf 脚本使用的自定义 m4 宏。

\* mywindows  
 MyWins 库的源代码

\* src  
 MiniGUI 库的源代码  
 \* src/client：MiniGUI-Lite 版本的客户端特有源代码。  
 \* src/control：MiniGUI 内建控件（static、button、listbox、edit 等等）。  
 \* src/font：设备字体实现（RBF、VBF、TrueType、Type1 字体等）。  
   \* src/font/in-core：嵌入代码的字体。  
 \* src/gal：老的 GAL 引擎。  
   \* src/gal/native：老的 FrameBuffer GAL 引擎。  
 \* src/gdi：老的 GDI 接口实现。  
 \* src/gui：窗口、菜单、加速建等 GUI 元素的实现。  
 \* src/ial：IAL 引擎。  
   \* src/ial/native：Console IAL 引擎。  
 \* src/image：老的 GIF、JPG 图片格式支持，已废弃。  
 \* src/ime：GB2312 输入法支持。  
 \* src/include：库的内部头文件。  
   \* src/include/control：控件相关的内部头文件。  
 \* src/kernel：窗口管理、共享资源管理等的源代码。  
 \* src/main：对 main 函数的封装。  
 \* src/misc：杂项实现：配置文件读写、套接字操作、定点数学函数、通用流函数、绘图同步等等。  
 \* src/mybmp：各种图片格式的支持，包括 GIF、JPG、PNG、BMP 等等。  
 \* src/newgal：新的 GAL 引擎。  
   \* src/newgal/fbcon：基于 FrameBuffer 的新 GAL 引擎。  
   \* src/newgal/dummy：哑 GAL 引擎。  
   \* src/newgal/qvfb：基于 QVFB 的新 GAL 引擎。  
 \* src/newgdi：新的 GDI 接口实现。  
 \* src/server：MiniGUI-Lite 版本服务器端特有源代码。

\* vcongui  
 VCOnGUI 函数库的实现。

您可以参考一下链接：

<https://www.cnblogs.com/sankye/articles/2075228.html>

<http://www.doc88.com/p-786671878309.html>

<https://blog.csdn.net/doon/article/details/7091379>

<https://blog.csdn.net/10km/article/details/83550368>

非常有用，包含很多知识点：

<https://so.csdn.net/so/search?q=MiniGUI&t=blog&u=anyuliuxing&urw=>

<https://blog.csdn.net/anyuliuxing/article/details/90239014>

其次，对于您最初的界面设计工作，您应该重点关注<window.h> ,<static.h>,<common.h>这几个文件的内容。同时：static.h只是static类控件的头文件，您如果使用了其他类型控件，就去找相应类型名字的 .h 即可。

# 三、项目中GUI内容的介绍

日常工作常用的工程路径：

SDK\minigui-3.2-20210521\src\control （控件类）

SDK\minigui-3.2-20210521\src\newgal\xsail\spi

SDK\minigui-3.2-20210521\src\newgal\video.c

（minigui刷新逻辑就在这里改,同时一定要去处理video.c的GAL\_UpdateRect（）函数，如何修改您可以参考之前FreeRTOS的修改内容，理论上可以直接copy）

SDK\minigui-3.2-20210521\src\newgdi

SDK\minigui-3.2-20210521\include

## 1、菜单的基础框架（IRadio为例，其余类似）

各界面才去统一的设计逻辑：每个菜单均对应一个.c/h文件，文件名为他们的各自的界面功能。（如Minigui\_App\_InternetRadio.c等）。

其中.c文件的基本框架为：

1. Create\_Minigui\_IRadioWnd
2. ntCreateIradiomenuEx
3. Iradiomenu\_onCreate
4. Iradiomenu\_onKeyUp
5. IRadioWinProc
6. Set\_MGui\_IRadio\_Content

上述的几个接口是每个界面都会具备的，他们兼顾着界面的 注册，刷新，处理按键事件等功能。当然根据每个界面的不同需求会有其他额外的拓展。比如switch\_channel（）等

接口简介：

1、Create\_Minigui\_IRadioWnd

这里对ntCreateIradiomenuEx接口的一个简单封装，因为你可能在创建洁面时需要依据不同的状态来确定这个界面最开始是什么样的，那么你就可以使用这种接口，并且日后在你新增的界面中，也建议使用这种逻辑。

1. ntCreateIradiomenuEx
2. Iradiomenu\_onCreate
3. Iradiomenu\_onKeyUp
4. IRadioWinProc

这几个接口都是通过ministudio生成的（其实在你熟悉之后，你可以自己手动添加而脱离工具），他们的功能就是注册并初始化界面、创建界面、界面按键消息处理、处理migniui方面的消息的接口。

1. Set\_MGui\_IRadio\_Content

这里是封装好的逻辑，所有的界面通过这个接口去刷新这个界面。当然也存在局部的特殊情况会跳过这个接口来刷新界面某些地方，但这种逻辑并不多。

## 常用的接口

上述框架的几个接口不再多赘述，您在使用时自然就会体会到接口的功能。

1. CreateWindow

这个接口可以创建控件和窗口，所以你需要熟悉这个接口内部的参数含义。

举个栗子：

Control\_test =

CreateWindow(

CTRL\_STATIC,

NULL,

WS\_CHILD|WS\_VISIBLE| SS\_HOFFSET\_VCENTER,

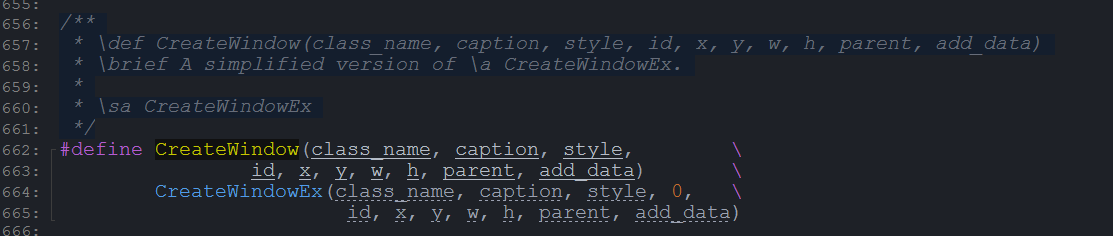
ID\_AIRABLE\_IRADIO\_TEXT1,

0, 1, 320, 33,

self->hwnd,

0);

这里创建了一个 static类 的控件，并将这个控件的句柄赋给 Control\_test，



1. SetWindowBkColor和SetWindowElementAttr

这里是设置窗口、控件的背景色。

举个栗子：

SetWindowBkColor(Control\_test,MGUI\_NICENT\_COLOR\_BLACK);

这里将上述刚刚创建的 Control\_test控件设置一个黑色的背景。

SetWindowElementAttr(Control\_test,WE\_FGC\_WINDOW,MGUI\_NICENT\_COLOR\_WHITE);

这里将上述刚刚创建的 Control\_test 控件设置一个白色的字体色[前景色]

1. Nicent\_SetWindowFont\_Not\_Refresh和SetWindowFont

用来设置窗口、控件的字体风格。当然你的确认你使用的字体是已经被创建的。

请注意Nicent\_SetWindowFont\_Not\_Refresh是属于我们自己拓展的接口，也建议你使用这个接口。使用SetWindowFont可能导致你的界面在你不需要刷新的时候进行了刷新的动作。（刷新机制与消息机制有关，可以大概参考下面介绍的SendMessage和PostMessage的区别）

1. SetWindowCallbackProc

这个接口用来设置某个界面在接受到minigui发送的消息时的 callback。Minigui这里使用的概念为WinProc（窗口处理例程），在使用这个接口设置新的winproc之后，后续所有的消息均会先发送到这里来处理。

请注意，如果你设置的winproc中没有处理某个消息类型（比如你没有处理msg\_timer），那么你应该把这个消息发往minigui的默认处理例程中去。实现这个逻辑的方法就是 ：消息若不处理，则return defaultwinproc（）；

详细的逻辑参考实际代码即可理解。（比如你可以参考IRadioWinProc（）这个接口的逻辑）

工程路径：SDK\apps-0.1.28\airable\src\iRadio\ui

1. SendMessage和PostMessage（非常重要，决定屏幕刷新快慢的关键）

这两个接口均是用来将某个消息发送到某个窗口/控件 的winproc中去的。

但请注意他们的不同之处，这很重要：

SendMessage是将消息发送到对用的winproc中去，并且该消息会立刻被处理，在处理完该消息之后才会执行后续的code。

PostMessage是将消息发送到消息队列中去，该消息只有在minigui从消息队列把这个消息取出来时才会处理这个消息。

举个栗子：

某个界面的winproc函数正在处理消息MSG\_KEYUP，在处理消息KEYUP的过程中使用SendMessage和PostMessage。SendMessage会在MSG\_KEYUP处理例程中直接将消息处理完，之后再继续处理MSG\_KEYUP。而PostMessage，他并不会在MSG\_KEYUP的例程中处理这个消息，他是在处理完MSG\_KEYUP消息后，minigui进入下一轮从消息队列获取消息，如果他拿到了PostMessage之前发过去的消息，那么他就会去处理这个消息。如果没拿到，那说明这个消息还在消息队列的后面，那就等待minigui把这个消息提取出来即可。

1. MGui\_Draw\_Color\_Block\_To\_Screen

目前我们没有处理好EraseBackground这个消息的处理，所以我做了一个新的接口，目前的最终效果是一致的。

这个接口的作用就是将指定的某一个区域刷成一个纯色。比如你想清屏你可以将界面整个刷黑：

MGui\_Draw\_Color\_Block\_To\_Screen(NULL,0,42,320,169,PIXEL\_black);

# 其他内容

## 部分核心的文件

头文件（.c文件和这里的里面是一致的，只是分在不同的目录下）



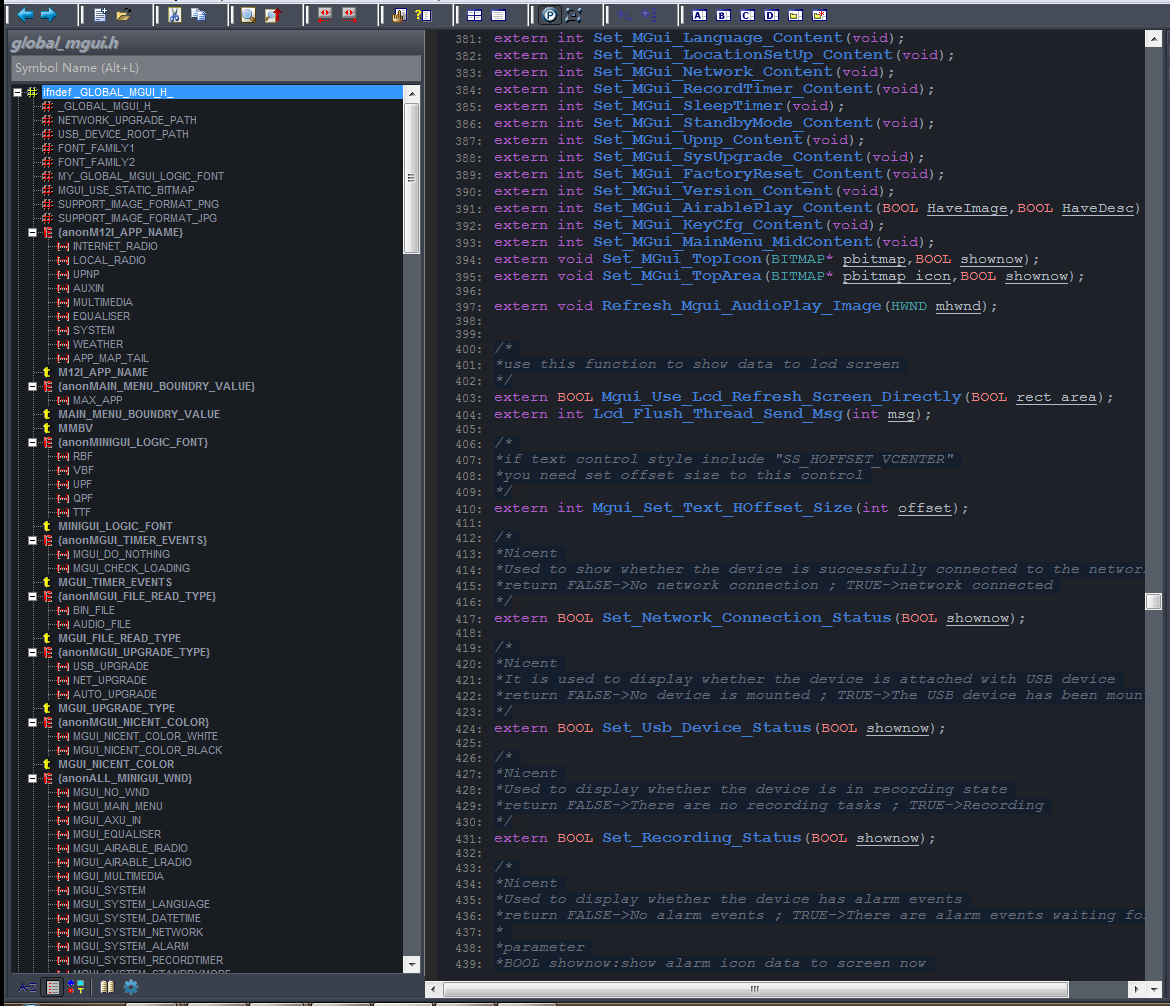
您可以自己去阅读里面的内容，功能就是他们的名字。

control\_id.h，是我们每次创建新的控件时必须要修改的地方，我们所有的控件ID都是在这里定义的，没有使用工具的common.h文件，请注意这一点。我们的工具目前仅用来创建一个空的窗口。不会用来创建任何控件，所有控件均手动添加。

Function.h，这里是一些通用性的接口的定义或者引用，你可以将一些不好归类的接口等放在这里，但建议您最好在后面添加注释，这样方便他人的理解。至于里面的结构，您去阅读一下便知。

Global\_event.h，这里用来处理一些GUI比较通用的事件，当然你可以可以再每个界面去单独实现它，但这样比较麻烦。比如无论哪个界面都有可能刷新时间，所以我们将刷新时间的消息放在这里处理。日后的新的需要统一消息也可以这么做。

Global\_mgui.h这里存放着所有的新建、拓展的与GUI相关的接口，并且每个接口都会有注释，具体接口的功能您看注释即可，该部分的接口内容较多。同时，所有GUI需要用到的一些全局变量等诸如此类都会在这里进行extern，方便各界面中的调用。



对于ncs-windows.h、resource.h，你每次在工具更新了窗口内容后，你需要将工具中的同名文件覆盖到项目中来文件。具体的方式会在其他文档说明。（如果后续工作完全脱离了工具，那么这些东西全可以丢掉）

ResourcePool目录，这里存着了公共资源的信息，比如图片库中各个图片的路径，还有项目中一些静态类的字符串。均在这里通过宏去定义他们，方便你使用。

APP目录，对应着底层模块的封装，这里的理念是，我们不想直接调用底层功能模块的接口，我们希望对他做一个简单的封装，这样在日后如果 底层的接口 有了变化之后，我们只需要维护APP目录的内容，而不需要跑到每个调用的地方去修改。那样过于麻烦且成本很高。

UI，不多介绍，所有的界面都对应着一个头文件

TOOL，这里是用来实现功能需求，比如多媒体播放，文件操作系统，这些功能都会做基于底层接口去实现。后续如果有新的工具类功能模块，可以添加在这里。

## 上层GUI与minigui的关系（我们自己特有）

这点很重要，切记：我们的GUI都是基于自己拓展的接口，宏等实现，所以如果你拥有一份没有被修改过的工程source，对于minigui和GUI的内容，你应该先修改minigui的内容并编译，之后才应该继续修改GUI的内容。由于实际功能的缺失，我目前通过宏 WANGXU\_CLOSE\_GUI\_CODE 关闭了所有功能相关的code，你可以通过搜索该宏去查看我关闭了哪些东西，这样在后续功能逐步完善后，你可以直接打开该部分，但打开宏的时候请注意，函数接口的名字等诸如此类的问题，修改一下就好。

## 部分同类型接口的内部流程

以下的这些接口是每个GUI都会定义的，同样以IRadio为例

### 3.1、Create\_Minigui\_IRadioWnd

这个接口的设计初衷是对工具生成的诸如 ntCreateIradiomenuEx（） 此类的接口进行简单的封装，你就可以不通过minigui的userdata进行数据传递，个人认为也比较耗时，传递一个数据需要通过各种接口，所以并未使用这种方式而是直接自己简单封装一个新的接口。

这个接口的基本用处为:

1、依据你实际的需求，传入一些数据，比如本次窗口的size等等诸如此类。

2、你可以在这里对所有需要的数值进行初始化，也建议在这里去执行该动作。

3、部分的分支逻辑可以再这里构建，如，创建U disk界面，但设备并未插入u disk，自然你创建的窗口应该是个弹窗而不是一个列表界面。这种逻辑你可以在这里处理。目前的思维模式是把弹窗独立出来，统一处理。所有的弹窗类型全部封装在一个接口中，依据实际需要执行相应code，后续会仔细介绍。

### 3.2、ntCreateIradiomenuEx

这个接口有工具生成，很难对该接口做出干涉。但基本结构如下：

NCS\_WND\_EXPORT mMainWnd\* ntCreateIradiomenuEx(HPACKAGE package, HWND hParent, HICON h\_icon, HMENU h\_menu, DWORD user\_data)

{

// please don't change me

return ncsCreateMainWindowIndirectFromID(package,

ID\_IRadioMenu, /\* window id \*/

hParent,

h\_icon,

h\_menu,

mainwnd\_Iradiomenu\_handlers,

NULL,

user\_data);

}

### 3.3、Iradiomenu\_onCreate

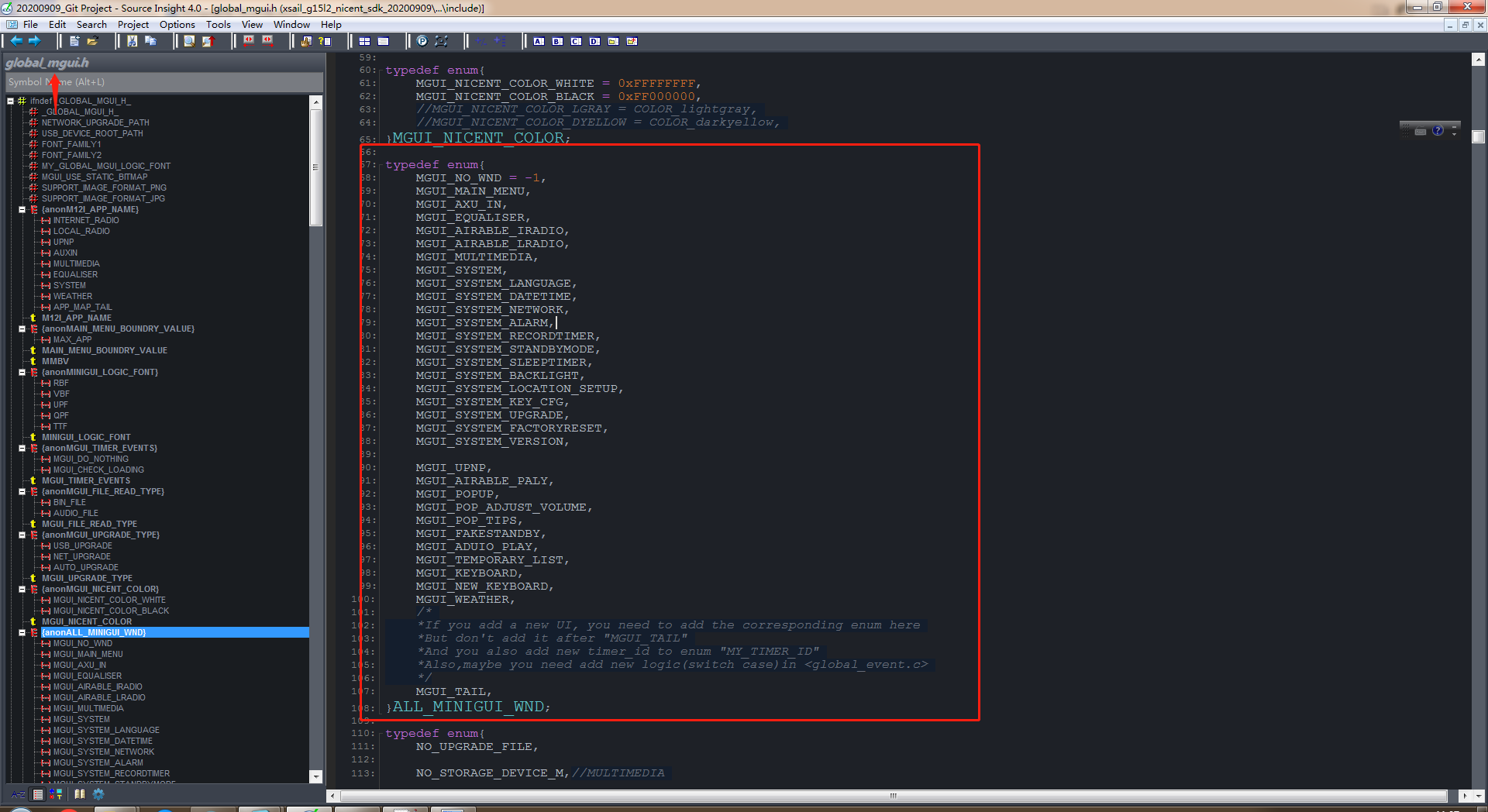
显然该接口用来真正创建一个窗口，具体流程如下;

1. 初始化，你也可以在上面提到的位置做这个动作，均可。不过部分数值是需要在这里初始化的：如

All\_Minigui[MGUI\_AIRABLE\_IRADIO] = self->hwnd;

请注意每次创建窗口时必须将该窗口的句柄赋值给All\_Minigui[],[]内的值你应该通过宏去定义好。新的窗口需要在这里添加宏：global\_mgui.h

宏名字如下：



一些提示，理论上每创建一个新窗口，你也都需要通过 宏 去拓展新的 timer等诸如此类的属性，具体内容，熟悉code之后即可了解。请留意图片中的注释部分。即每当你创建一个全新的窗口你至少要做这些动作：

1. global\_mgui.h中的几个枚举：ALL\_MINIGUI\_WND、MY\_TIMER\_ID
2. 除工具生成的接口外，你至少需要构建三个全新的接口：（IRadio为例）

Create\_Minigui\_IRadioWnd:对原创建窗口的简单封装，并且在 global\_mgui.h中进行申明；

Set\_MGui\_IRadio\_Content:构建一个借口专门用来刷新窗口

IRadioWinProc:每个窗口都需要你构建一个属于自己的winproc处 理例程，方便你处理部分消息事件。

3）在control\_id.h中添加你新的控件ID，毕竟很少有窗口不包含控件。

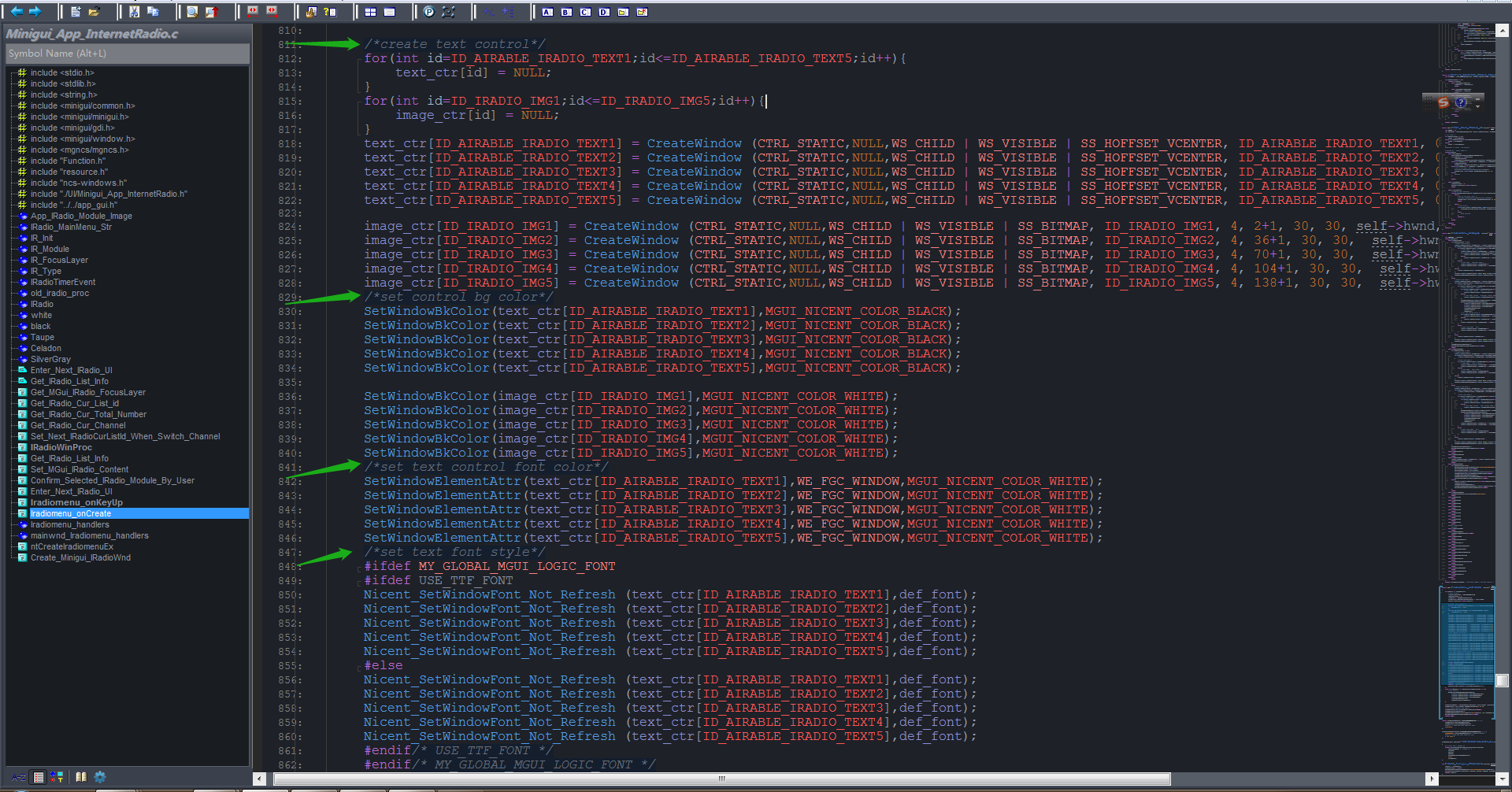
1. 创建控件，注意所有的控件的句柄均应该被存储，目前大致分为两类存储：text\_ctr[]和image\_ctr[]。同样的，你如果有新的控件，自然也需要去拓展新的宏，宏的具体位置为：control\_id.h。所有控件id都在这里定义。其实这里当初在设计时为考虑到text\_ctr[]和image\_ctr[]冲突的问题，他们[]的id是有可能重合的，但后续认为不会影响，暂未处理。日后工作需要留意一下。text\_ctr[]的 控件id 定义在这里 CREATE\_TEXT\_CTR\_ID ，image\_ctr[]的 控件id 定义在这里 CREATE\_IMG\_CTR\_ID ，具体细节看code即可。下面是个创建控件的实例：

text\_ctr[ID\_AIRABLE\_IRADIO\_TEXT1]

=

CreateWindow (CTRL\_STATIC,NULL,WS\_CHILD | WS\_VISIBLE | SS\_HOFFSET\_VCENTER, ID\_AIRABLE\_IRADIO\_TEXT1, 0, 1, 320, 33, self->hwnd, 0);

1. 设置控件属性。在控件创建完成后，需要设置一些属性，当然部分属性你可以在createWindow时就去设置好。但是某些属性必须要后续自己重新设置。在这里必须要设置的属性有：字体前景色，字体背景色（控件背景色），使用哪种逻辑字体。具体如下：



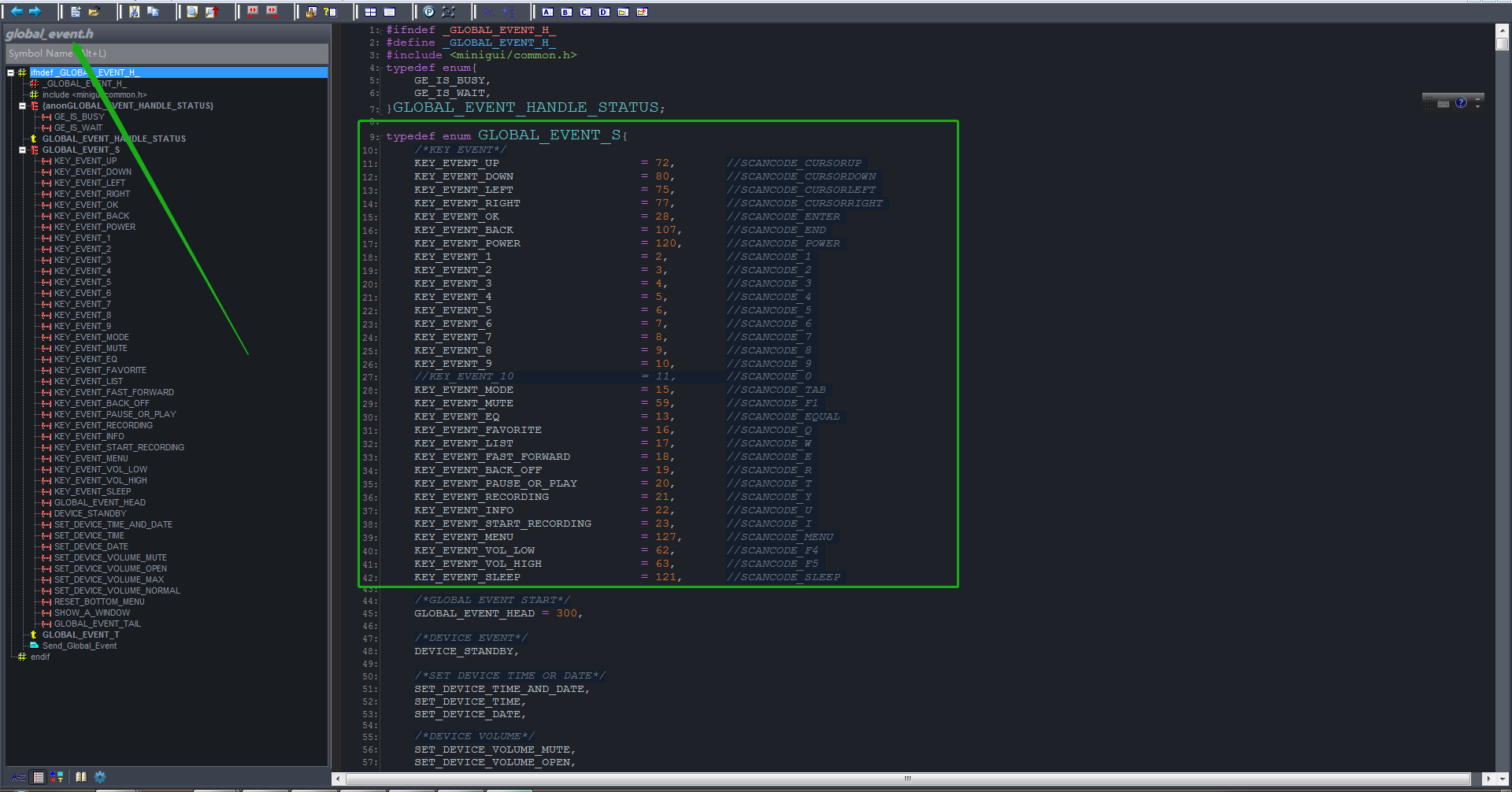
其余你需要设置的属性也建议在这里去做。

1. 给定一个新的窗口callback接口。这里的初衷是：对于部分常用的消息事件，我们希望可以更自由的处理它，所以我们对每个窗口均构建了一个对应的callback接口，如 IRadioWinProc（）。常用的事件会在这里处理。但请注意对于所有的你不处理的消息等，你应该将这些消息丢给其原先默认的callback处理。具体实现方式看IRadio的IRadioWinProc（）实现方式
2. 触发计时器：SendMessage (self->hwnd, MMSG\_CREATE\_TIMER, 0, 0);
3. 填充窗口内容，所有的窗口内容填充逻辑，均封装成了一个接口，如Set\_MGui\_IRadio\_Content（）。建议在对整个窗口填充内容时，均调用此接口。当然，在部分只需要刷新一小部分的内容时，可以直接调用minigui的接口去做，但请注意相关的逻辑控制等问题。
4. 通知lcd刷新屏幕，这里的逻辑目前是没做的，你需要修改minigui下的video.c,xsail\_gal.c等内容，具体的修改参考原FreeRTOS工程目录。如果后续你修改了这一部分内容，请注意我自己之前构建好的逻辑，使用相应的接口。如，你想刷新屏幕，理论上是调用：

Mgui\_Use\_Lcd\_Refresh\_Screen\_Directly(TRUE);这个接口

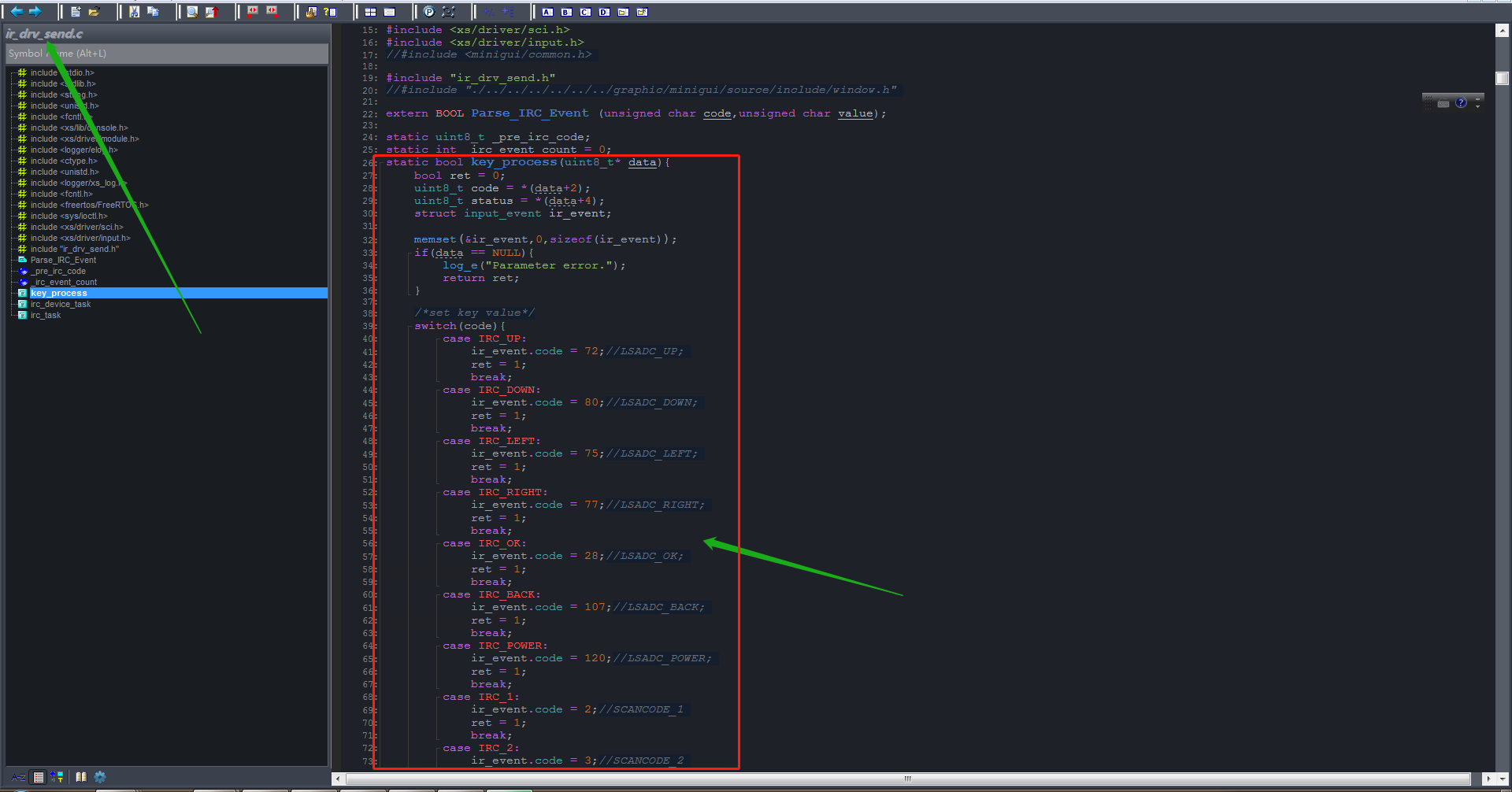
### 3.4、Iradiomenu\_onKeyUp

这个接口专门用于处理 MSG\_CHAR等 按键类的消息。所有的键值我均已用宏定义好，如果后续有新的键值需要定义，那么你应该定义在这里：global\_event.h

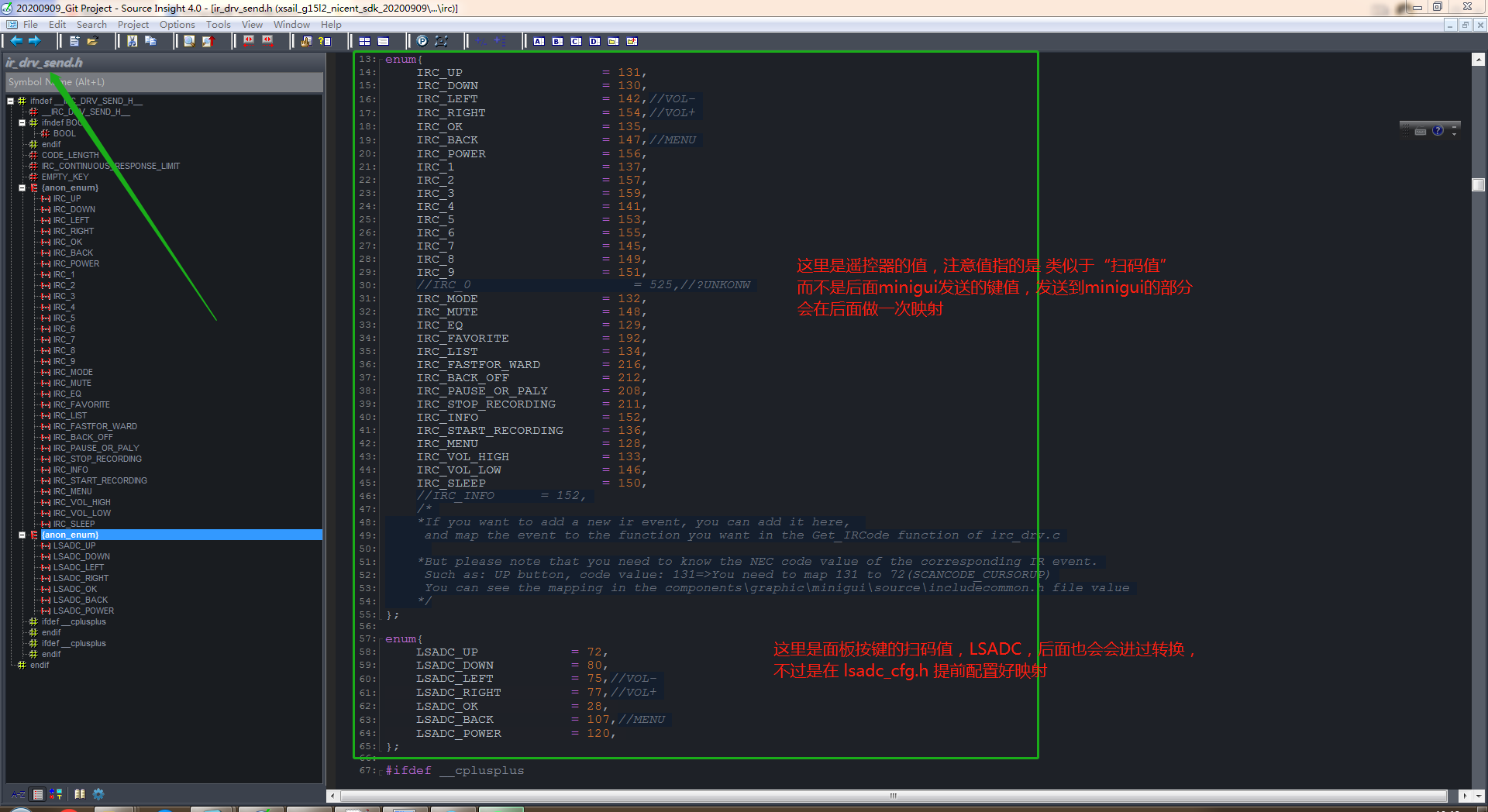


请注意，如果你想知道 键值映射 做的地方 ，那么你可以看这里，不过下面介绍的是FreeRTOS的内容，在linux平台上可能会有不同，需要自行了解一下，并且这些内容都是我们自己做的，xsail不会支持：

这里是 FreeRTOS键值映射 的部分,

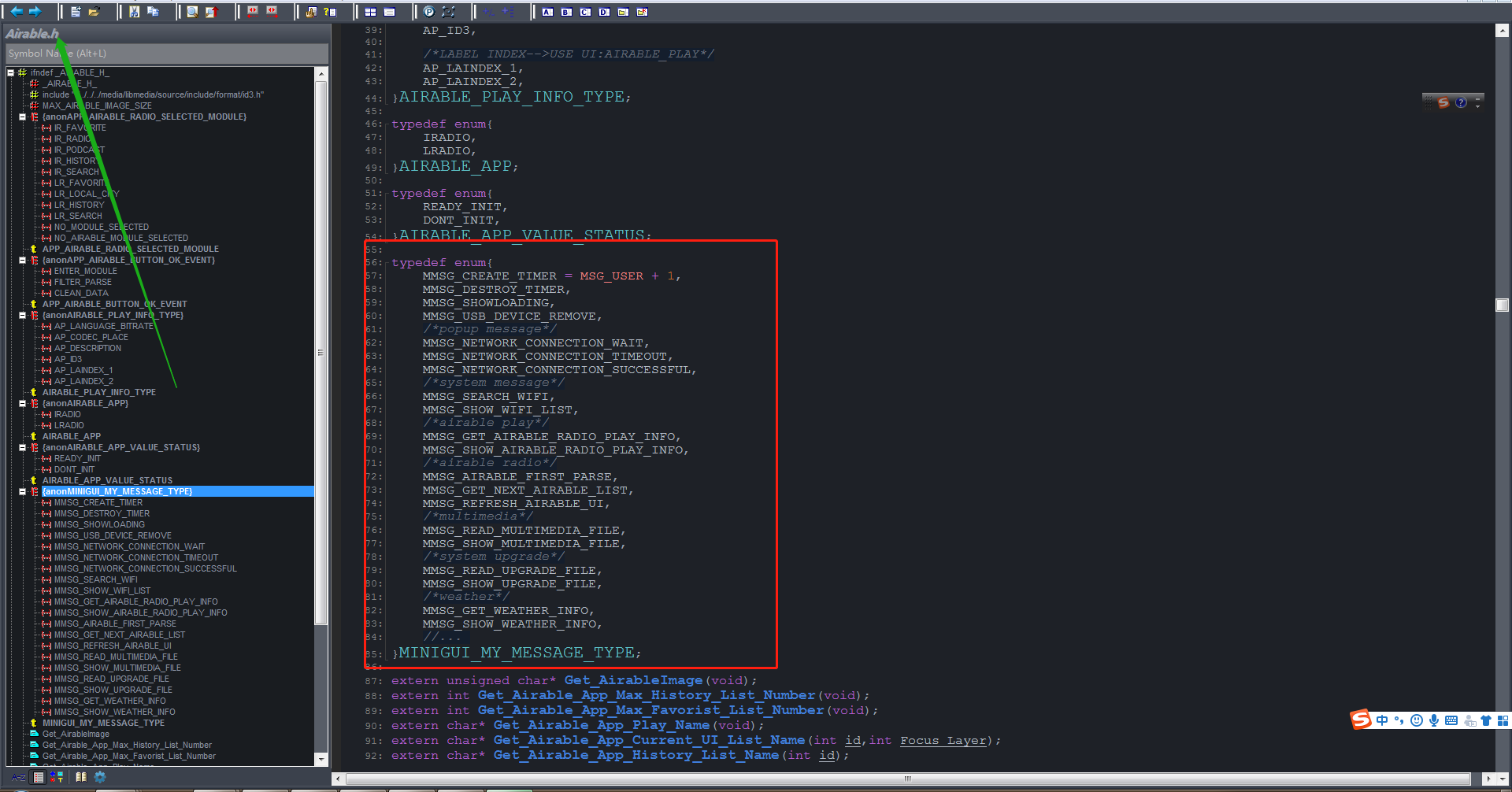


这里是 FreeRTOS记录扫码值 的部分



### 3.5、IRadioWinProc

除了上述的案件消息以外，minigui自然会有很多其他的消息类型需要被处理，如果你有某些消息想要自己做处理，那么你就可以在这个接口里面天剑相应的case。此外，每个winproc函数里面，你必须要自己处理的消息类型有：MSG\_TIMER、MMSG\_CREATE\_TIMER、MMSG\_DESTROY\_TIMER。至于其他的某些消息会经常处理到，不过依据你的需求来定。同时，请注意，按照目前的规则，minigui的源消息类型都是MSG\_\*\*\* , 所有的自定义的消息都是 MMSG\_\*\*\*，某些消息功能和minigui是重复的，但为了管理方便，我们重新定义了它来自己处理。自定义的消息是在这里：



请注意一点，这里由于当初的疏忽，自定义消息被放在了Airable.h这里，所以会看起来很别扭，后续在处理的时候，你应该把这些 枚举类型 转移到在global\_mgui.h中。因为他们本来就属于那里。

#### 3.5.1、自定义消息简介

首先，你需要知道，你自定义消息类型的值不应该小于MSG\_USER + 1，这很重要。

##### MMSG\_CREATE\_TIMER

用来创建计时器，和minigui的MSG\_CREATE\_TIMER一致，不多赘述。

##### MMSG\_DESTROY\_TIMER

用来销毁计时器，和minigui的MSG\_DESTROY\_TIMER一致，不多赘述。

##### MMSG\_SHOWLOADING

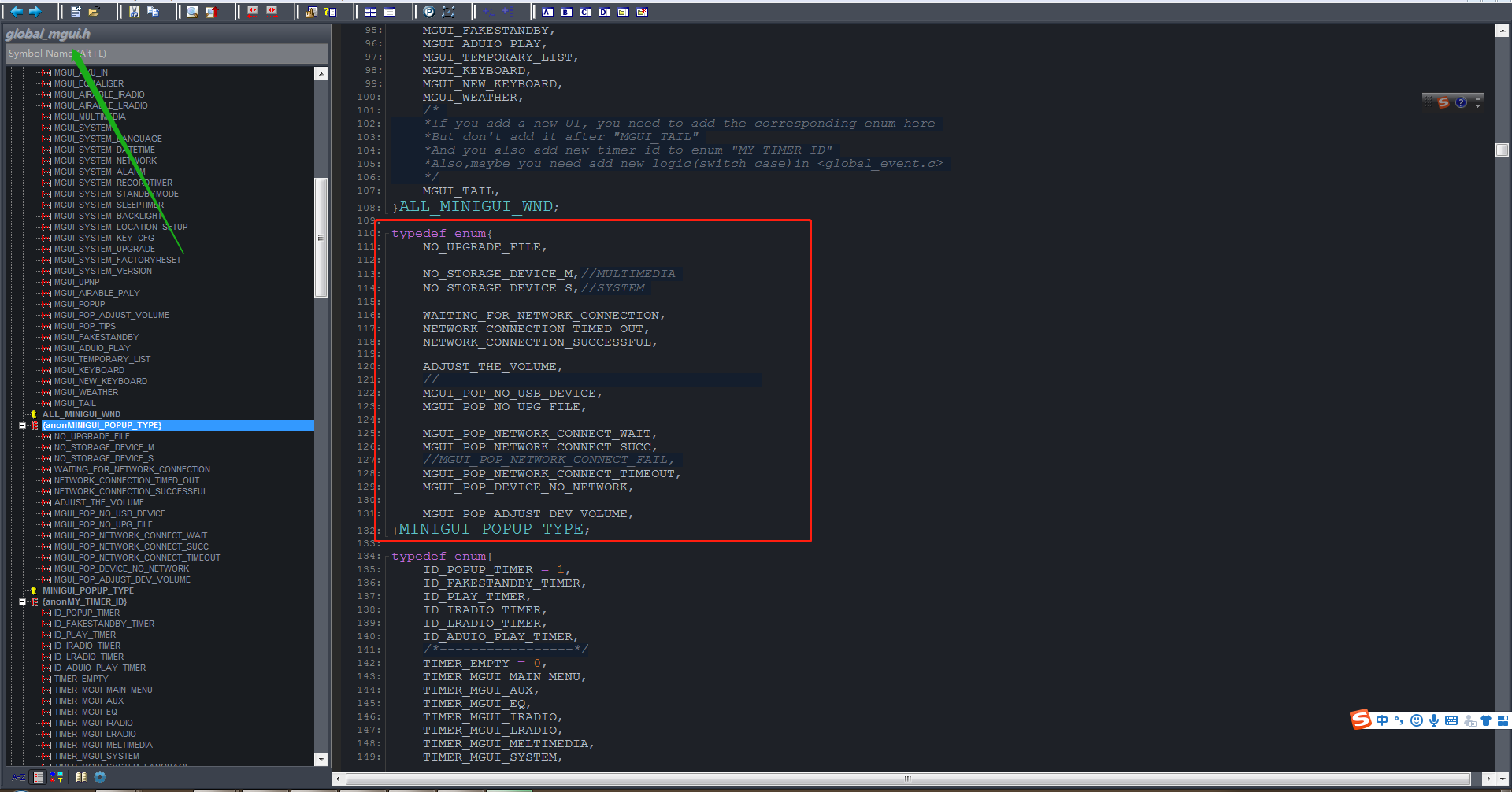
在很多流程控制中，我们的界面都会有暂时显示 “loading...”的需求，你可以将该消息放在各自的winproc中实现。可以参考IRadioWinproc的实现方式。

##### 其余消息

其余消息多数都是针对于每个GUI去拓展的特有的消息，每个消息的功能区阅读其使用位置即可。

## 弹窗的处理

所有的弹窗类型，均放在Minigui\_Popup.c中实现，但是请注意：这里并不是将所有的弹窗都使用同一窗口实现，是每种类型弹窗对应一种窗口，只是code中把这些东西封装在一起，Create\_Minigui\_PopupWnd（）依据传入的参数去构建对应的弹窗。目前具备的弹窗类型有：Adjust Volume Popup ； Popup Tips 。前者调节音量使用，后者适用于 提示“无网络连接”“无usb设备”等诸如如此类的弹窗。当然，提示什么信息你可以通过枚举去决定好，这样就可以统一处理提示信息。



## minigui自身机制

### 5.1、Minigui的消息发送，接收与分发。

发送：

像PostMessage此类的接口（SendMessage不是），最终都会将一个 minigui消息发送到消息队列中去，该消息队列是在minigui初始化的时候创建的，具体内容可自行查阅。

接收：

或者说是提取消息，minigui通过Getmessage（）去提取消息队列中的消息。

分发：

在提取到消息后，消息会通过 TranslateMessage和DispatchMessage 后发送给对应的出汗那个口，之后消息会在窗口的winproc中被处理，当然案件消息经过工具处理，你可以直接跳到对应的OnkeyUP函数。

### 5.2、基础消息框架

如图:

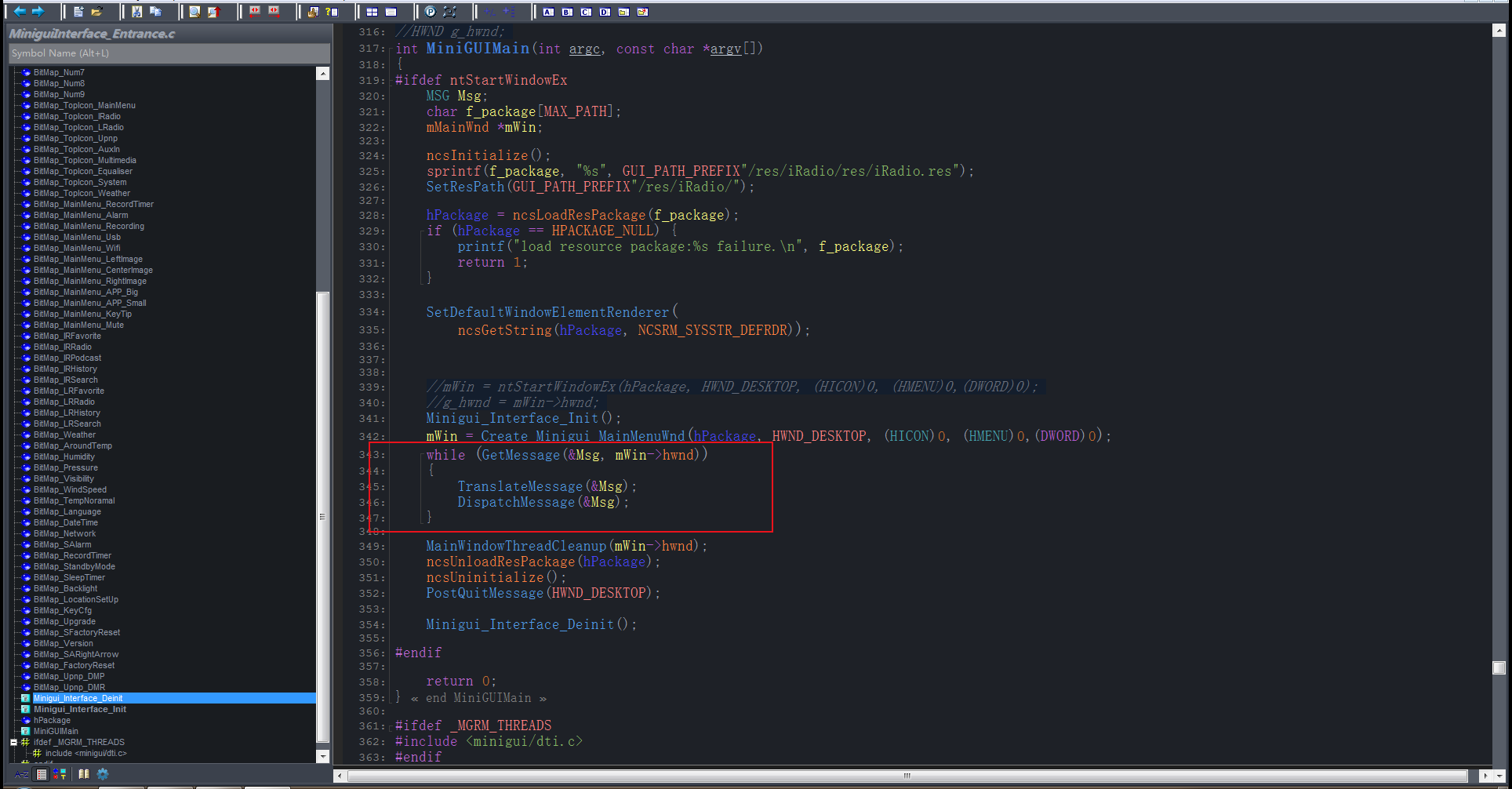
while (GetMessage(&Msg, mWin->hwnd))

{

TranslateMessage(&Msg);

DispatchMessage(&Msg);

}



Minigui绝大多数的消息都会在这里被 取出、分发。

GetMessage负责取出minigui消息队列的消息，之后交由Translate确认消息类型，然后通过DisPatchmessage消息去分发至每个窗口的winproc中去处理。

如果你想要的停止这个消息循环，你只需要发送一个 MSG\_QUIT消息到消息队列中即可，但请注意，你需要释放掉相应的资源，具体逻辑可以参考图片中while{}之后的处理。

你可与根据你实际的逻辑来拓展，比如图中的Minigui\_Interface\_Deinit();就是我们自己拓展的，他用来释放我们自己创建的字体资源等。

### 5.3、minigui的初始化

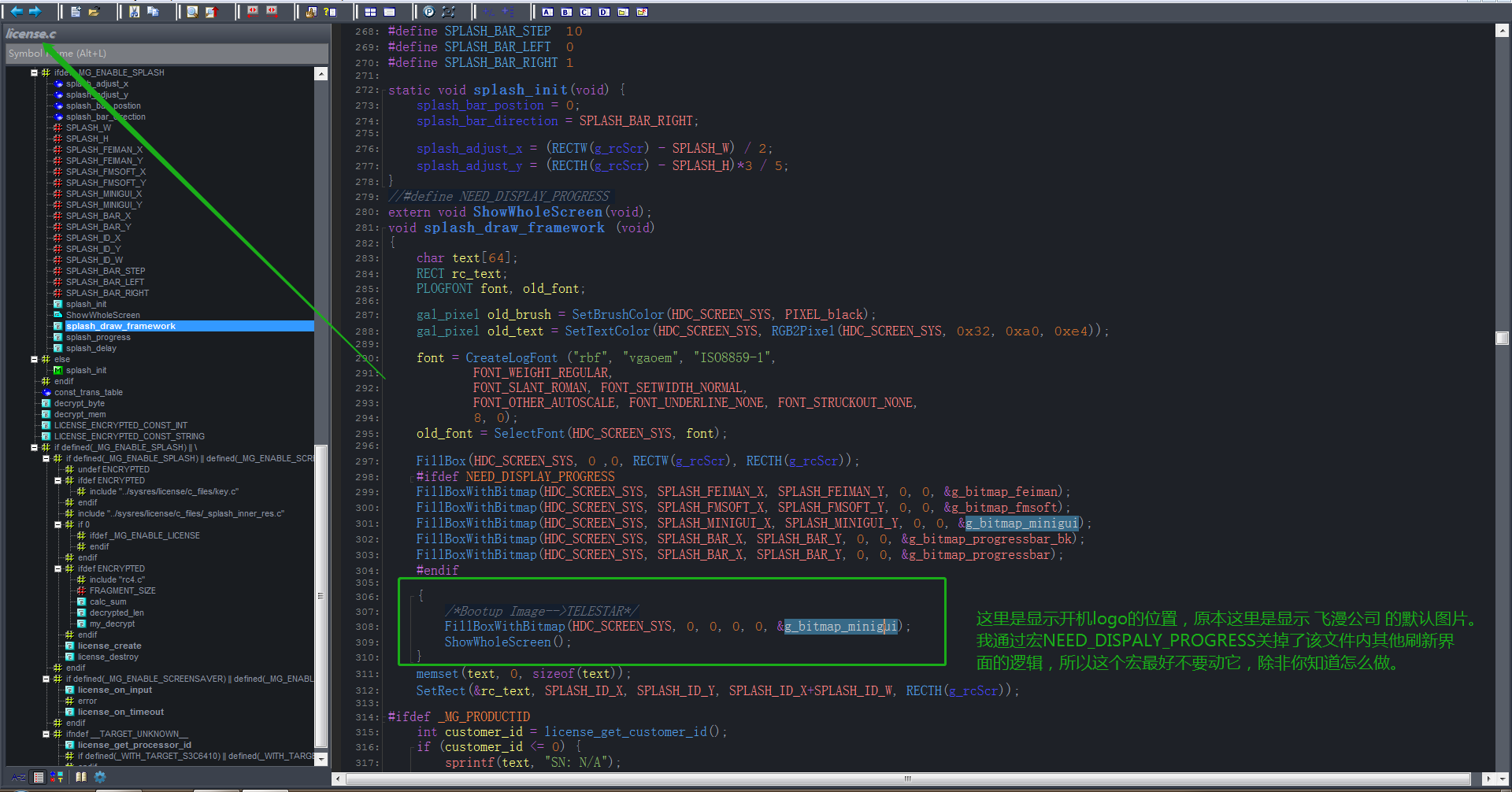
这里参考这个链接，非常详细，请仔细阅读：

<https://blog.csdn.net/anyuliuxing/article/details/90239014>

### 5.4、设备开机Logo

显示开机logo的位置是在这里：

minigui\source\src\misc\license.c

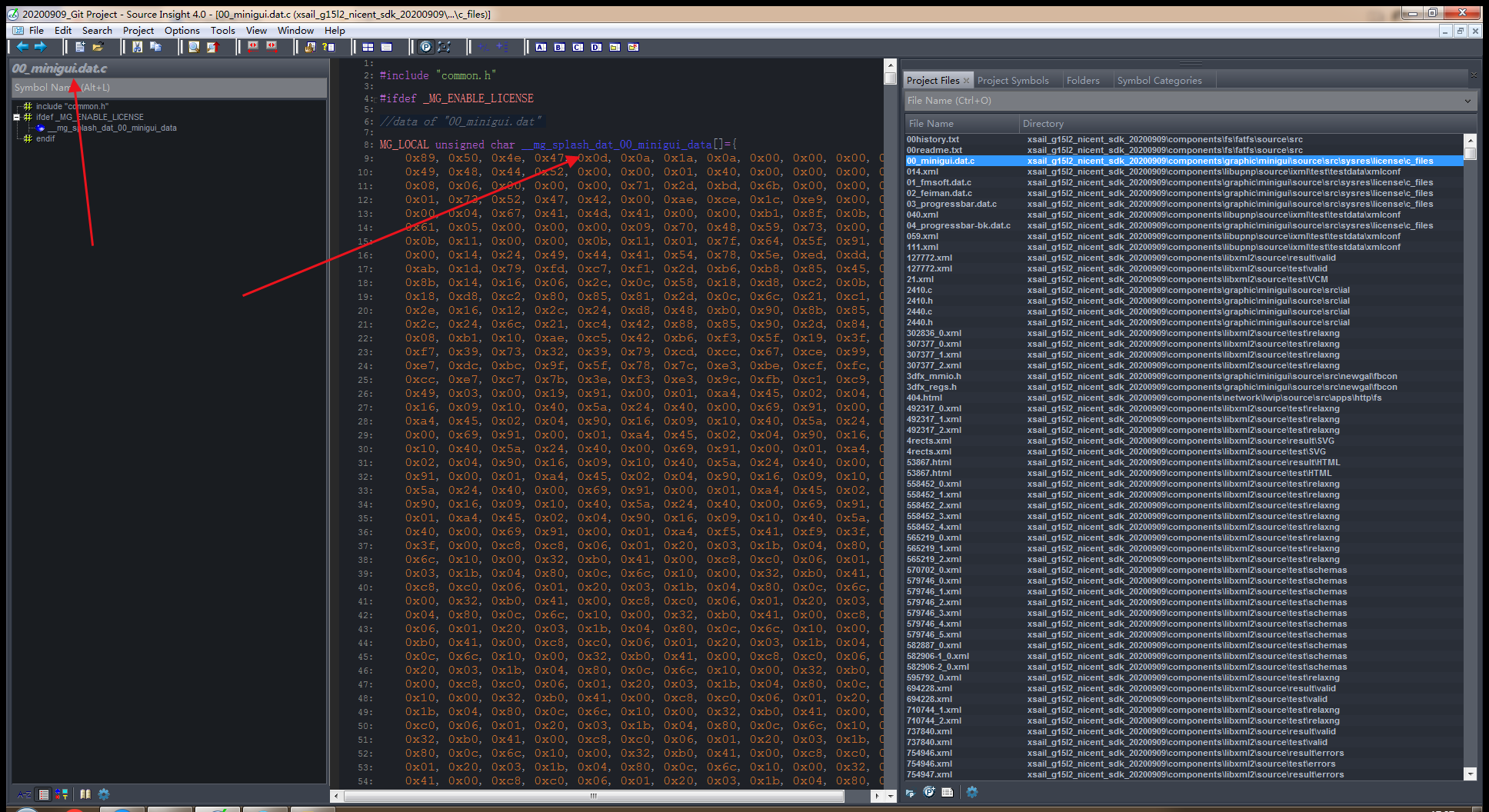


存放开机logo图片的位置是在这里：

minigui\source\src\sysres\license\c\_files\00\_minigui.dat.c

直接到该文件中替换掉图片数据就好。

linux通过xxd命令你可以把图片转成你想要的的数据格式，然后在覆盖掉\_\_mg\_splash\_dat\_00\_minigui\_data的数据之后重新 make minigui-rebuild all就可以。



### 5.5、上层GUI启动流程

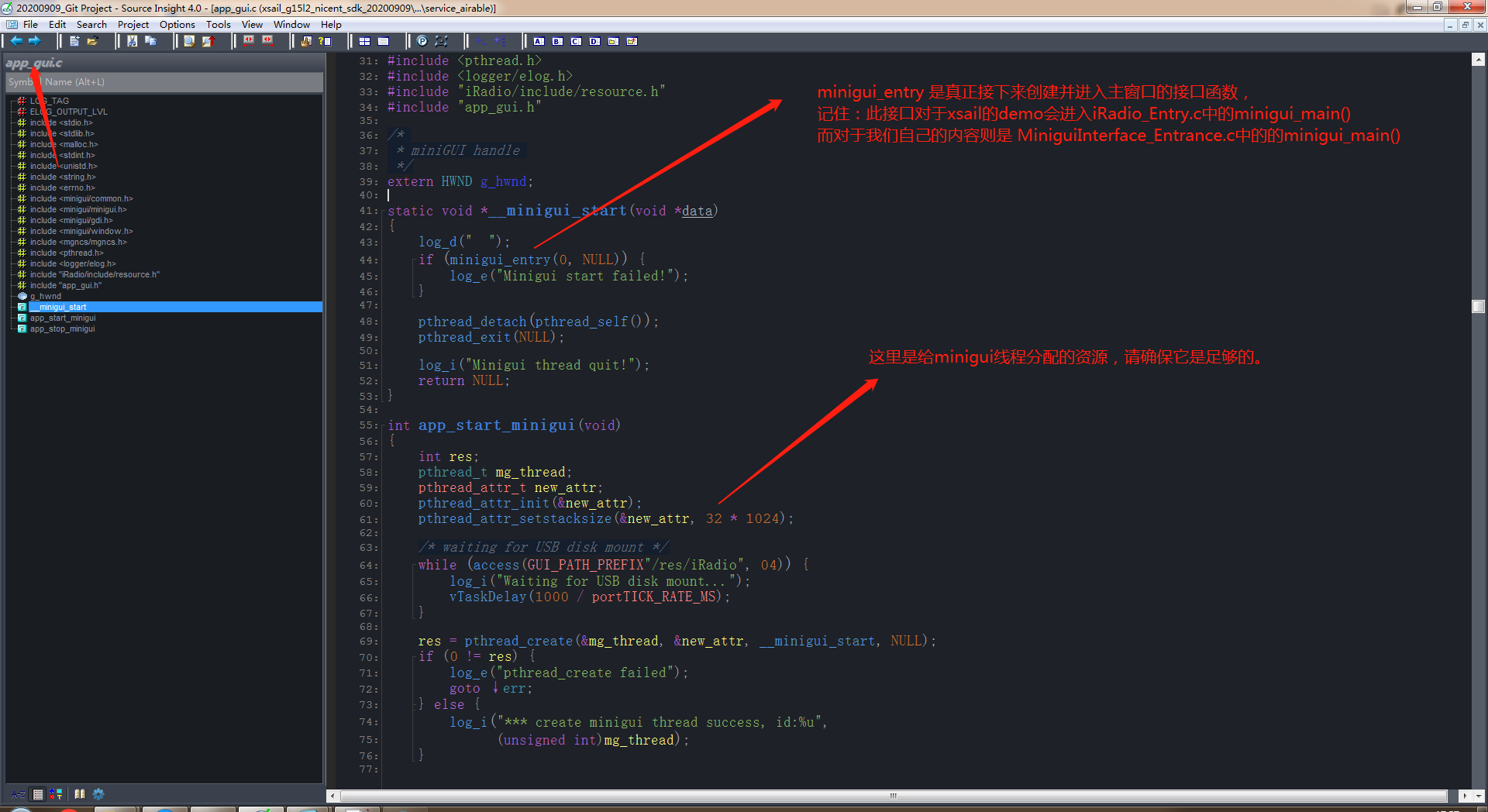
此处的启动流程仅针对minigui的内容，更底层的启动流程需要去阅读对应项目的底层驱动逻辑等内容。

Minigui的自身的初始化流程请参考5.3中提供的链接<https://blog.csdn.net/anyuliuxing/article/details/90239014>

，以下内容直接叙述上层GUI的启动流程，这些内容均在对应的apps-0.1.28模块下（找下工程中有相应的APP字眼的目录就是，请注意不同工程的名称会有些许差异，但是关键字一定都有）。

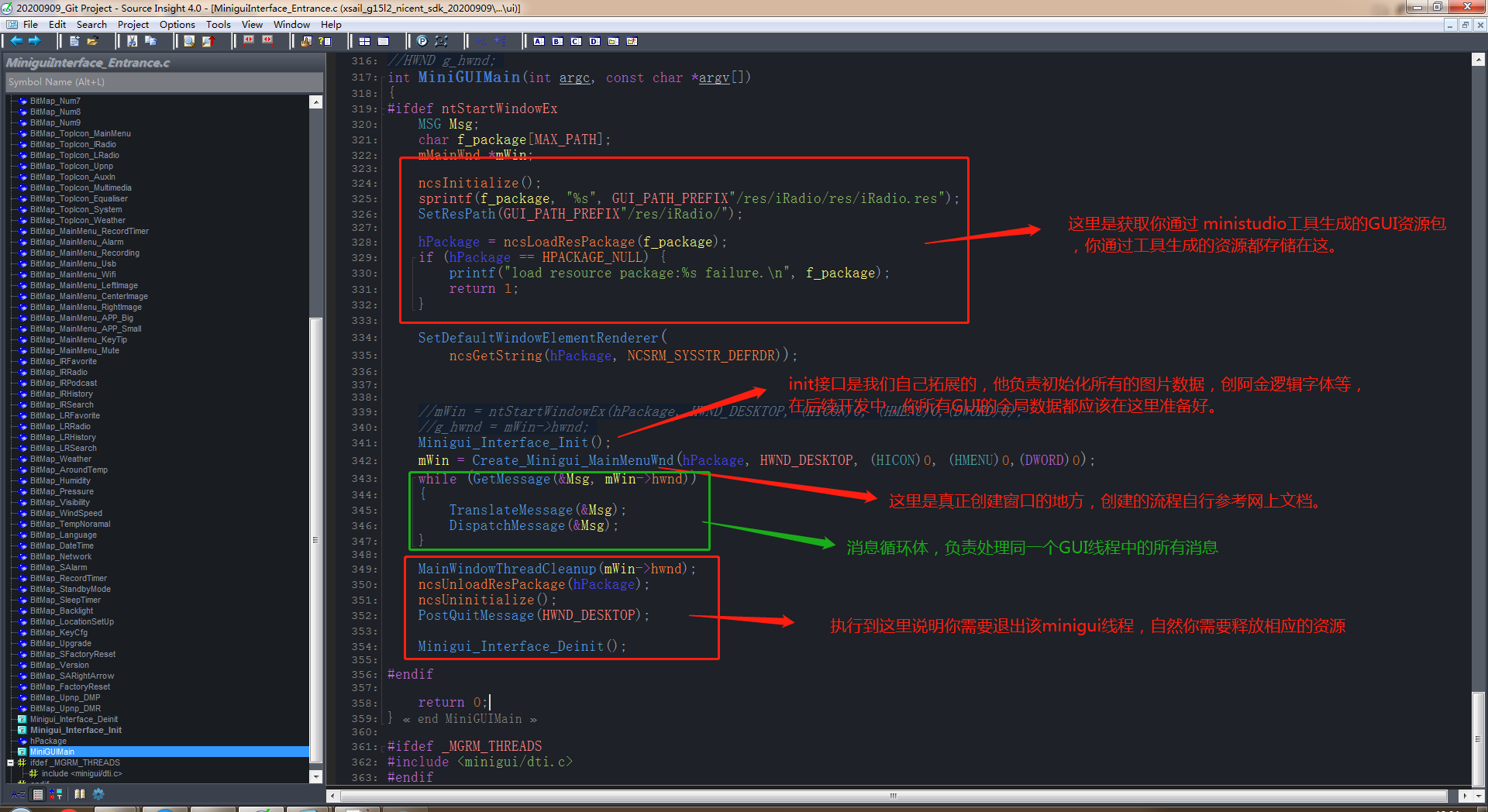
因为xsail提供的最初版demo是可以支持airable 的模块的，所以你需要进入相应的airable（airable 或 service airable）目录下，进入之后可以自行阅读下大致的目录框架，以下针对GUI启动流程进行介绍：

在你未修改的时候，你的GUI主程序应该是在app\_gui.c启动的，图中简单介绍了其流程：

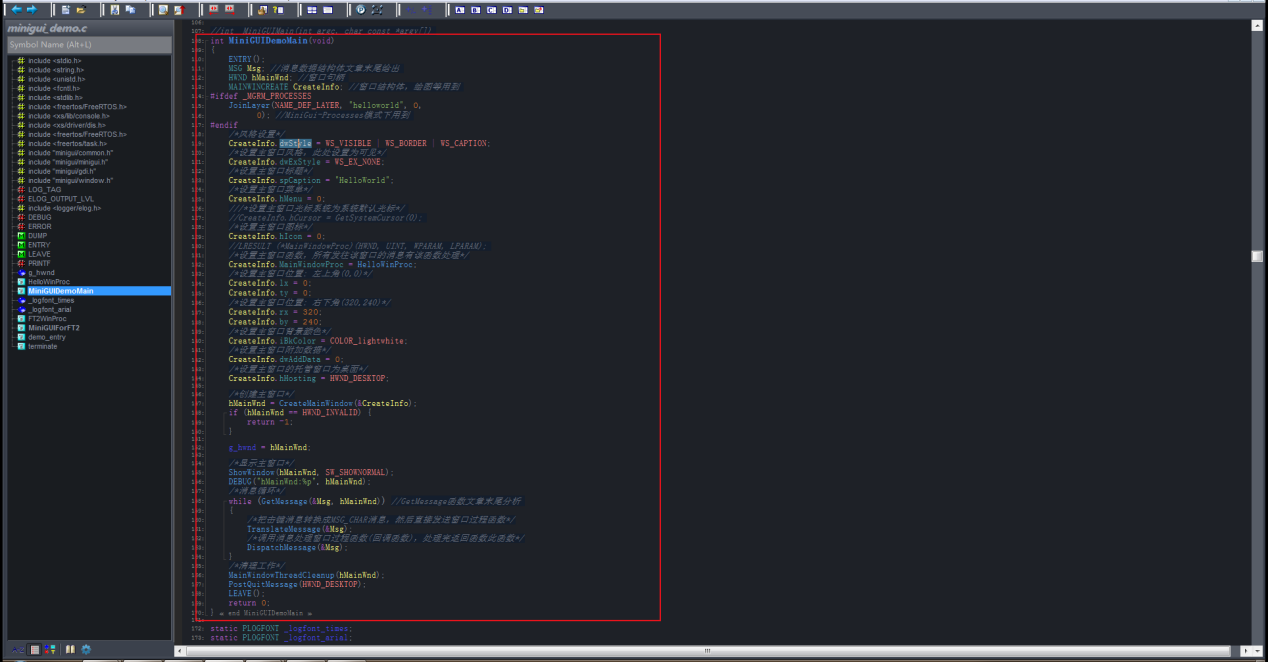


请注意：对于app\_gui.c的 接口 是在这里被调用的：application.c，具体看其中的内容即可理解。

在这之后，就会进入MiniGUIMain（）函数，这里会创建你的第一个窗口并显示它，如下：



请注意，上面的方式是告诉你ministudio的主窗口窗口工作方式，如果不想通过工具生成窗口，那么你可以参考工程中的minigui\_demo.c的内容。同时，目前的GUI架构虽然绝大多数都不在依赖工具，但所有的窗口都是用工具生成的，请知悉。



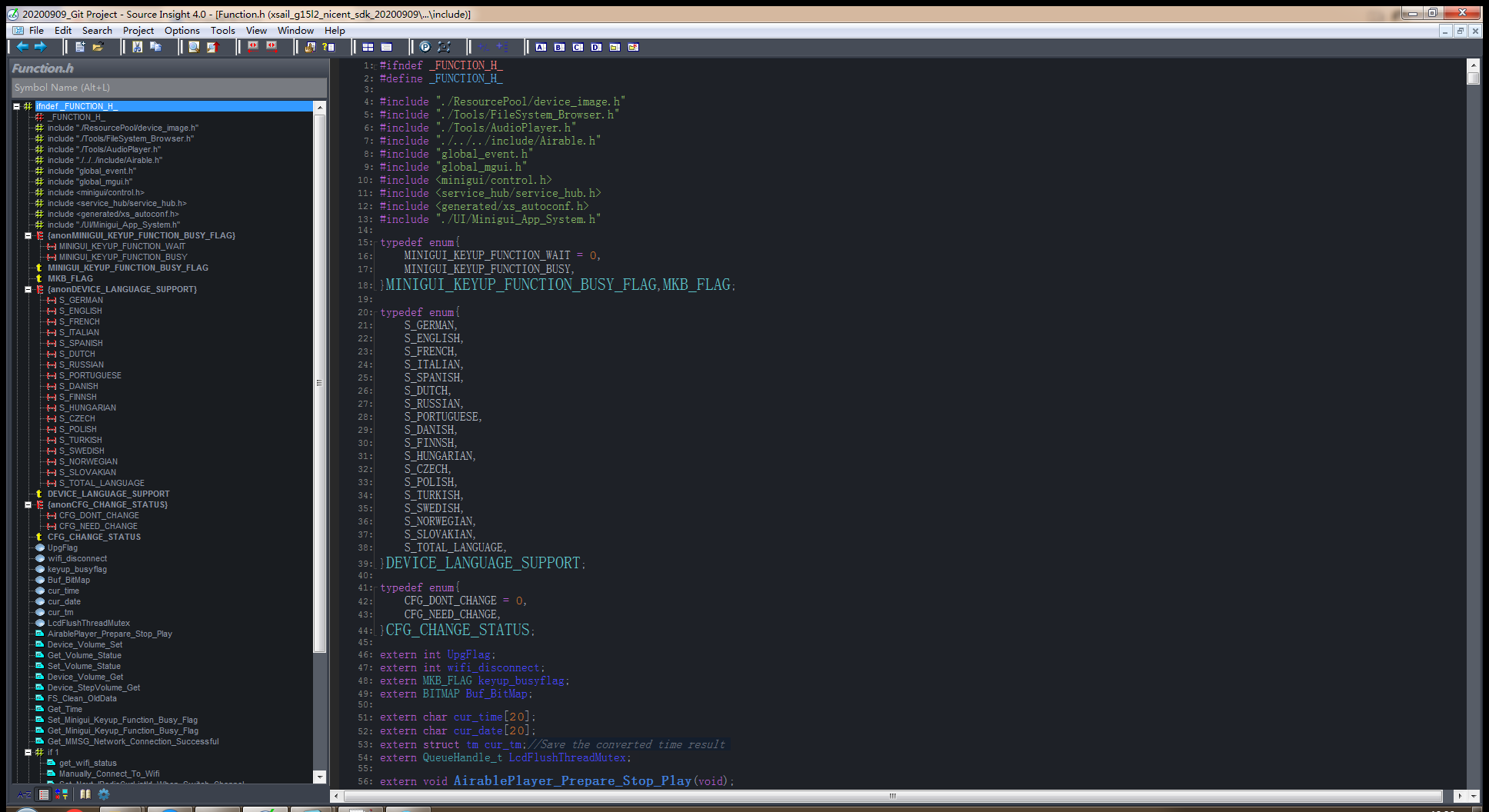
### 5.6、GUI工作中重要逻辑介绍（重要）

对于你几乎所有GUI都会处理的内容，依据功能你可以放在global\_event.c和global\_mgui.c和Function.c中去：

event是用来处理你各GUI中通用的事件，如开关机，刷新时间等等。

Mgui是用来处理minigui相关的内容，比如我们封装了一个统一的创建逻辑字体的接口，识别图片的格式。还有一定注意，我们也封装了一些刷新的接口，请务必熟练使用它们，具体的介绍你可以阅读相应的global\_mgui.h中的注释，里面说的很详细。再说一次，请务必熟练使用。

Function.c存放的比较杂，在你熟悉之后，你可以新建你喜欢的分支目录，目前一些不好分类的接口都放在这里。同时请务必知悉一个逻辑：对于Function文件，它有个特殊用途，那就是Function.h文件，我们几乎把所有的头文件都添加到了这里，所有对于绝大多数的GUI中，你引入Function.h文件就几乎可以满足你觉得部分的需求。很重要！！！同时Function.h中也引入了其他功能的接口，详情请看其中的注释。



### 5.7、Minigui.cfg（重要）

请务必参考以下链接，所有的Minigui.cfg内容里面均有涉及，这对你熟悉Minigui.cfg很重要：

https://blog.csdn.net/yuanbinquan/article/details/43019129?ops\_request\_misc=%25257B%252522request%25255Fid%252522%25253A%252522160836231816780279128818%252522%25252C%252522scm%252522%25253A%25252220140713.130102334.pc%25255Fblog.%252522%25257D&request\_id=160836231816780279128818&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~blog~first\_rank\_v1~rank\_blog\_v1-1-43019129.pc\_v1\_rank\_blog\_v1&utm\_term=times.ttf

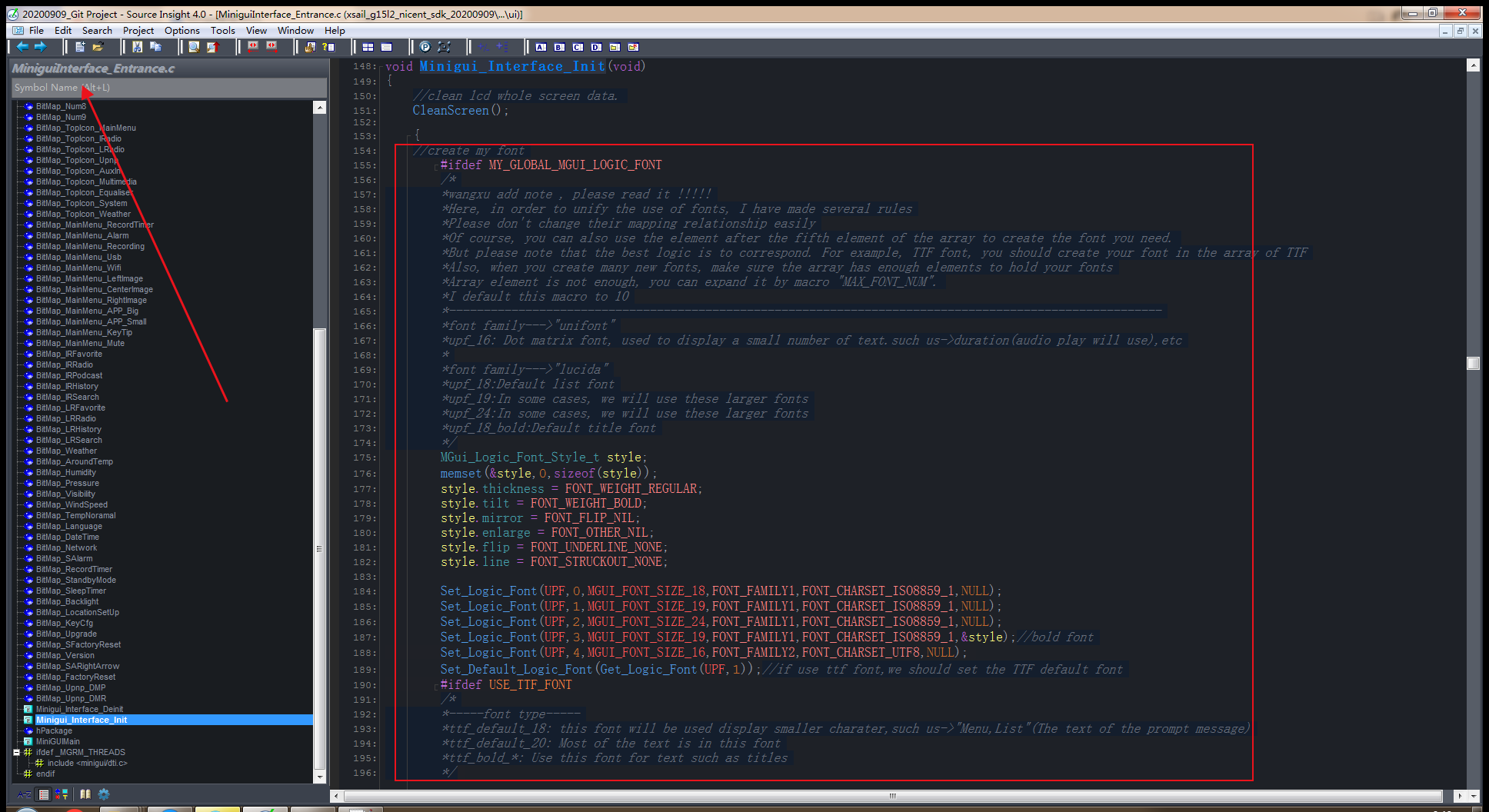
#### 5.7.1、设备字体和逻辑字体

Minigui的字体分为设备字体和逻辑字体两种。

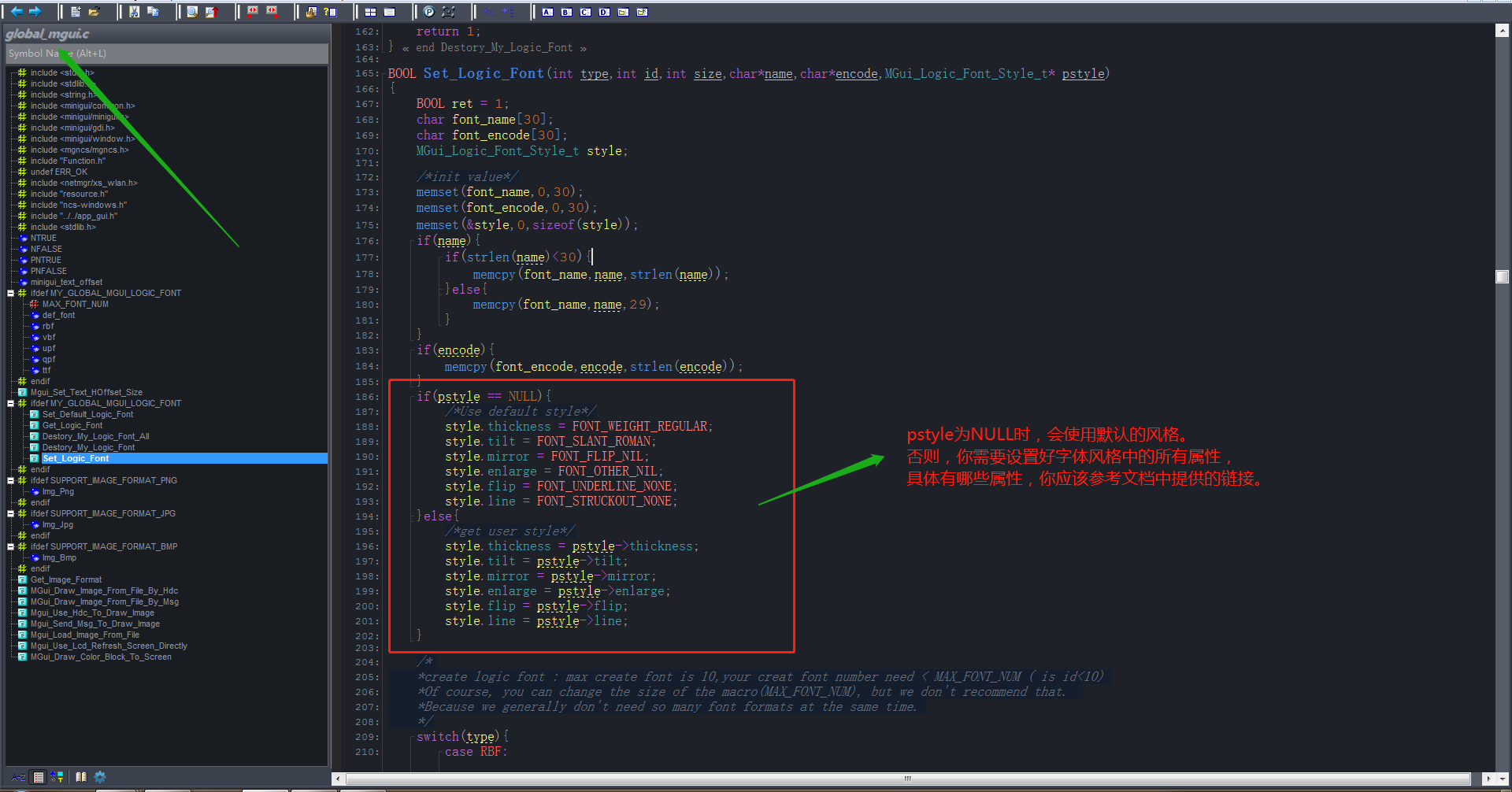
设备字体是在Mnigui.cfg中配置好的内容，存在多少设备字体取决于你cfg中配置了多少。至于具体的配置方式可以参考提供的链接。

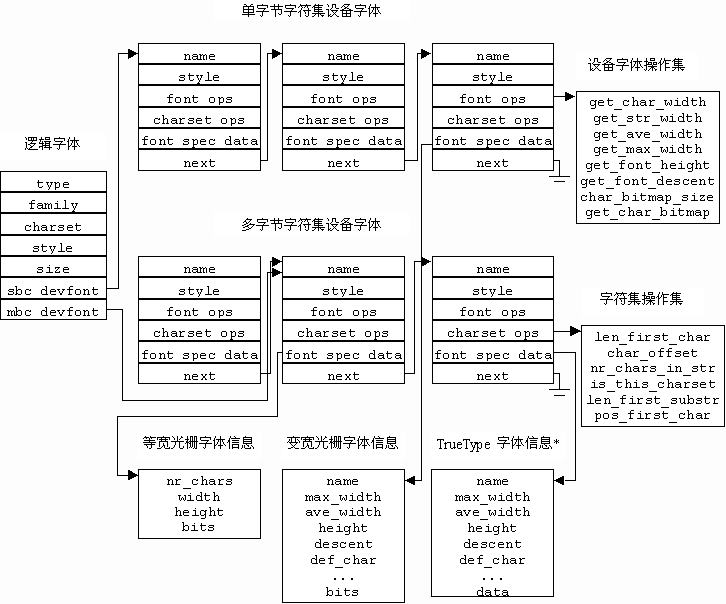
逻辑字体是实际在GUI工作中使用的字体内容，请注意，逻辑字体是基于设备字体才可以创建的。对于逻辑字体的创建方式：我在global\_mgui.c中封装了一个接口BOOL Set\_Logic\_Font(int type,int id,int size,char\*name,char\*encode,MGui\_Logic\_Font\_Style\_t\* pstyle)

，你可以很方便的使用它而无需过多关心字体的风格，接口具体的使用方式参考图片。而且理论上，关于创建字体的逻辑，你都应该在这里处理好：



你可以将Set\_Logic\_Font（）接口中最后一个参数为NULL，这样你就可以使用我设置的默认字体风格。如果你觉得这个默认风格不好，你也可以去修改它。具体的位置是在这里：





### 5.8、文件系统

目前linux工程未了解文件系统接口位置，以下提供了在原FreeRTOS系统中基于文件系统封装的文件浏览接口，文件名《FileSystem\_Broswer.c》具体内容可以阅读其中的内容，以下提供了一盒核心的接口逻辑：

int FileSystem\_Save\_Required\_SubFile(char \*path,MRF\_NUM\_ETIME int MRead,TOF\_TO\_STORE int SF\_Type,BOOL SeeDetail,int\* poffset)

{

int id = 0;

int ret = 0;

int offset = \*poffset;

int len = strlen(path);

DIR \*dirp;

struct stat st;

struct dirent \*entry;

struct tm \*tm\_local = NULL;

char fmt\_time[64];

char item\_path[CUR\_FILE\_SYSTEM\_PATH\_LEN];

memset(MGF\_List,0,sizeof(MGF\_List));

if ((dirp = opendir(path)) == NULL) {

/\*open dir fail\*/

return ret = -1;

}

/\*If you want to store more files at a time, you can expand this macro 'CUR\_FILE\_SYSTEM\_LIST\_SIZE'.

\*The default value of this macro here is 100,

\*The value passed in should not exceed it.

\*/

if(MRead>CUR\_FILE\_SYSTEM\_LIST\_SIZE){

MRead = CUR\_FILE\_SYSTEM\_LIST\_SIZE;

}

while(1){

if(strlen(MGF\_List[MRead-1].FileName) == 0){

/\*The maximum number has not been read, continue to read\*/

entry = readdir(dirp);

if(entry){

/\*read dir next file succ , The last file in the file list has not been read ,will save\*/

switch(SF\_Type){

case 0:/\*get .bin file\*/

case 1:/\*get music file\*/

{

if (S\_ISDIR(entry->d\_type)) {

/\*is dir , the file type that must be stored\*/

if(id >=offset\*MRead){

/\*

\*We don't need to store all the files that meet the requirements in the U disk at one time,

\*but only take part of them at a time.

\*Because our UI will not display all the files at one time,

\*we can record an offset to store the next batch of files from here

\*/

/\*save this folder\*/

MGF\_List[id-offset\*MRead].FileType = 1;

memcpy(MGF\_List[id-offset\*MRead].FileName,entry->d\_name,

(sizeof(entry->d\_name)/sizeof(char))<=CUR\_FILE\_SYSTEM\_PATH\_LEN?sizeof(entry->d\_name):CUR\_FILE\_SYSTEM\_PATH\_LEN);

/\*show this file info\*/

if (SeeDetail) {

/\*show detail\*/

memset(item\_path, 0, CUR\_FILE\_SYSTEM\_PATH\_LEN);

strcpy(item\_path, path);

if (item\_path[len - 1] != '/') {

item\_path[len] = '/';

}

strcpy(item\_path + strlen(item\_path),MGF\_List[id-offset\*MRead].FileName);

ret = stat((const char \*)item\_path, &st);

if (ret != 0){

return ret;

}

tm\_local = gmtime(&st.st\_mtime);

strftime(fmt\_time, 64, "%b %d %Y %H:%M:%S",tm\_local);

printf("\n[st\_size = %10lu][fmt\_time = %s][filename%d = %s]\n",st.st\_size, fmt\_time,id-offset\*MRead,MGF\_List[id-offset\*MRead].FileName);

}else{

/\*only show name\*/

printf("\nfilename%d = %s\n",id-offset\*MRead,MGF\_List[id-offset\*MRead].FileName);

}

id++;

}else{

/\*read next file\*/

continue;

}

}

else {

/\*not dir , Store the corresponding file types according to the actual needs\*/

if(FileSystem\_Check\_File\_Suffix(SF\_Type,entry->d\_name)){

/\*this file need store\*/

if(id >=offset\*MRead){

/\*

\*We don't need to store all the files that meet the requirements in the U disk at one time,

\*but only take part of them at a time.

\*Because our UI will not display all the files at one time,

\*we can record an offset to store the next batch of files from here

\*/

/\*save this file\*/

MGF\_List[id-offset\*MRead].FileType = 2;

memcpy(MGF\_List[id-offset\*MRead].FileName,entry->d\_name,

(sizeof(entry->d\_name)/sizeof(char))<=CUR\_FILE\_SYSTEM\_PATH\_LEN?sizeof(entry->d\_name):CUR\_FILE\_SYSTEM\_PATH\_LEN);

/\*show this file info\*/

if (SeeDetail) {

/\*show detail\*/

memset(item\_path, 0, CUR\_FILE\_SYSTEM\_PATH\_LEN);

strcpy(item\_path, path);

if (item\_path[len - 1] != '/') {

item\_path[len] = '/';

}

strcpy(item\_path + strlen(item\_path),MGF\_List[id-offset\*MRead].FileName);

ret = stat((const char \*)item\_path, &st);

if (ret != 0){

return ret;

}

tm\_local = gmtime(&st.st\_mtime);

strftime(fmt\_time, 64, "%b %d %Y %H:%M:%S",tm\_local);

printf("\n[st\_size = %10lu][fmt\_time = %s][filename%d = %s]\n",st.st\_size, fmt\_time,id-offset\*MRead,MGF\_List[id-offset\*MRead].FileName);

}else{

/\*only show name\*/

printf("\nfilename%d = %s\n",id-offset\*MRead,MGF\_List[id-offset\*MRead].FileName);

}

id++;

}

else{

/\*read next file\*/

continue;

}

}

else{

/\*read next file\*/

continue;

}

}

#if 0

/\*printf every file name\*/

printf("\nfilename = %s\n", entry->d\_name);

#endif

}

break;

default :

break;

}

}

else{

if(strlen(MGF\_List[0].FileName) == 0){

/\*

\*Although we have read to the end of the list,

\*we have not stored any data,

\*so we will read again at the beginning of the list.

\*

\*This is because the logic of offset is not rigorous enough,

\*resulting in unnecessary reduction of efficiency

\*The logic here will be redesigned later.

\*/

/\*reset value,Start reading again in the head\*/

closedir(dirp);

dirp = opendir(path);

id = 0;

offset = 0;

\*poffset = 0;//ui->multimedia/system upgrade/etc.

continue;

}else{

/\*Read to the bottom of the list and store the data to end the loop\*/

break;

}

}

}

else{

/\*Enough information has been read to end the loop\*/

break;

}

}

closedir(dirp);

return ret;

}

### 5.9、多行文本的显示

详情参考这个链接，非常详细：

<https://cloud.tencent.com/developer/article/1433551>

我们会在Airable Play的界面使用多行文本的逻辑，因为description字符串可能会很长。

