Polarix

Xuyulin91@163.com

摘要

简单说明HMI引擎的作用与运行机理。

SimpleGUI

HMI模型简要说明

1. **说明**

SimpleGUI的演示例程中涉及到了一个简单的用于界面交互的机制，我称之为HMI（Human Machine Interface）引擎，本文将对在这个HMI引擎做简单的原理阐述和使用说明。

首先声明，这个HMI引擎是一个临时的式作品，在SimpleGUI的不断改进过程中，HMI引擎将可能发生结构性的变化，介时此文档将会有对应的更新，敬请关注。

1. **设计背景**

HMI引擎本身和SimpleGUI没有具体的联系，SimpleGUI的本质是实现点、线、平面几何体、位图（包括文字）的绘制，进而将这些基本的绘图操作进行组合，实现诸如提示框（NoticeBox）、列表（List）等的显示反馈。进而用户可以使用这些绘图接口，构建自己需要的界面，而对于列表、文本框等，也提供了相关的操作做和更新接口。按照SimpleGUI的原始设计来讲，每一个SimpleGUI提供的界面或控件，只要原始数据不改变，就可以通过操作做和更新接口在屏幕上重新显示。

通常来讲，用户只需要将对应的交互操作如按键、触屏等的响应函数与对应的显示功能函数对应起来就行了，但是有些情况下，这样的操作使得系统运行时的开销很大，例如在多级列表时，在显示列表1时，选择该列表中的项目2时会跳转至列表2，在列表2中选择项目1时又会跳转至列表3……这种情况下，如果用普通的if-else或者switch-case结构，在当前函数没有运行完成时，又跳转到了一个新的函数中，新的函数没有执行完，又会跳转到另一个新的函数中……而这期间又难免会有一些和外部逻辑器件、传感器等的交互操作。

对计算机系统原理稍有了解的人应该了解，函数调用的过程中、函数参数的传递和函数内局部变量的声明都是要占用栈空间的，每进行一次函数调用、函数的参数、局部变量等就会被压入栈中，函数调用结束后再从栈中弹出。如果出现之前描述的情况，用户编写的函数较为复杂，调用层次又比较多的时候，对栈空间的开销时非常大的。甚至不排除会有内存耗尽的情况。

鉴于此，我设计了这个简单的HMI引擎，用来临时的解决一下在这个问题。临时起意的小设计，还不是很完善，欢迎大家吐槽。

1. **原理阐述**

为了解决上述问题，HMI的设计机理就是“无嵌套”概念，即代码层面的逻辑上，屏幕画面没有主从之分，只会从一个画面迁移到另一个画面或者从当前画面返回到之前的画面。所以，再这个HMI引擎中，用户需要将在自己定义的画面设计、封装好，然后由HMI引擎调用。同一时刻被激活并显示的画面中只有一个，在画面进行跳转之前，当前显示的画面的数据一定会被释放。

1. **机能简介**

HMI机能的代码实现位于以下两个文件中。

|  |  |
| --- | --- |
| HMI\inc\HMI\_Engine.h | 数据结构与控制宏定义。 |
| HMI\src\HMI\_Engine.c | 功能实现。 |

HMI引擎的操作函数如下：

|  |  |
| --- | --- |
| HMI\_PrepareEngine | 初始化HMI引擎对象。 |
| HMI\_AddScreen | 添加用户画面。 |
| HMI\_ActiveEngine | 激活引擎，同时指定初始画面。 |
| HMI\_StartEngine | 启动引擎，并使用参数数据初始化并显示初始画面。 |
| HMI\_ProcessEvent | 执行动作，执行用户、中断或传感器动作。 |
| HMI\_PostProcess | 后处理，HMI\_ProcessEvent执行成功后需要做的处理。 |
| HMI\_Goto | 画面跳转。 |
| HMI\_GoBack | 返回前画面。 |

HMI引擎相关的宏定义参数如下：

|  |  |
| --- | --- |
| HMI\_SCREEN\_HISTORY\_MAX | 返回历史的保存数量。指用户可以GoBack多少次。 |
| HMI\_SCREEN\_LIST\_MAX | HMI引擎可以最大拥有的画面数。由于同一个画面可能在一个操作做流程中进入多次，所以和HMI\_SCREEN\_HISTORY\_MAX参数区别设计。 |
| HMI\_SCREEN\_START | 默认的起始画面索引。 |

用户再在组织画面时候，需要参照以下结构体的函数指针类型定义，针对每一个画面建立五个函数。

|  |  |
| --- | --- |
| Initialize | 画面数据初始化函数 |
| Prepare | 画面数据准备/预设定函数 |
| Repaint | 画面更新/刷新函数 |
| ProcessEvent | 事件处理函数 |
| PostProcess | 事件后处理函数 |

结构体定义参见HMI\_Engine.h文件中HMI\_SCREEN\_ACTION类型的定义。

|  |
| --- |
| typedef struct  {  HMI\_ENGINE\_RESULT (\*Initialize)(void);  HMI\_ENGINE\_RESULT (\*Prepare)(const void\* pstParameters);  HMI\_ENGINE\_RESULT (\*Repaint)(const void\* pstParameters);  HMI\_ENGINE\_RESULT (\*ProcessEvent)(HMI\_EVENT\_TYPE eEventType, const HMI\_EVENT\* pstEvent);  HMI\_ENGINE\_RESULT (\*PostProcess)(SGUI\_INT iActionResult);  }HMI\_SCREEN\_ACTION; |

HMI中各功能函数的返回值类型均为HMI\_ENGINE\_RESULT，此类型为一个枚举类型，具体定义如下。

|  |  |
| --- | --- |
| HMI\_RET\_ERROR | 一般逻辑错误。 |
| HMI\_RET\_INVALID\_DATA | 无效数据，一般般用于事件处理中。 |
| HMI\_RET\_NO\_SPACE | 无可用空间，一般用于HMI引擎有初始化处理添加画面时。 |
| HMI\_RET\_MEMORY\_ERR | 内存错误，如malloc失败时。 |
| HMI\_RET\_UNDEFINED\_ACTION | 未定义的动作。 |
| HMI\_RET\_ERROR\_STATE | 状态错误。 |
| HMI\_RET\_NORMAL | 处理正常结束。 |
| HMI\_RET\_INITIALIZE | 初始化完成。 |
| HMI\_RET\_CONFIRM | 提交处理，通常指用户触发确认操作，需要画面迁移时。 |
| HMI\_RET\_CANCEL | 取消处理，通常指用户触发取消操作，需要画面回滚时。 |
| HMI\_RET\_FOLLOWUP | 需要画面迁移，通常指画面自动发生改变。 |
| HMI\_RET\_FALLBACK | 回滚道初始画面 |
| HMI\_RET\_INPROCESSING | 动作处理中，不能执行新的操作做。 |
| HMI\_RET\_NOACTION | 处理完成，但不需要有任何后续动作。 |

此定义用户可以根据自己的实际需要重新定义，但需要保留HMI\_RET\_NORMAL、HMI\_RET\_NOACTION、HMI\_RET\_ERROR三个关键值，HMI\_RET\_NORMAL定义为0，处理正确时返回值大于0，错误时返回值小于0.

最后是HMI对象数据结构HMI\_ENGINE\_OBJECT的定义：

|  |
| --- |
| typedef struct  {  HMI\_SCREEN\_OBJECT\* Screen[HMI\_SCREEN\_LIST\_MAX];  SGUI\_INT ScreenCount;  HMI\_SCREEN\_OBJECT\* CurrentScreenObject;  HMI\_HISTORY\_STACK History;  HMI\_ENGINE\_STATE State;  }HMI\_ENGINE\_OBJECT; |

描述如下：

|  |  |
| --- | --- |
| Screen | 指针数组，指向已加入HMI引擎的各个画面数据。 |
| ScreenCount | 画面总数。 |
| CurrentScreenObject | 当前活动画面的指针。 |
| History | 画面迁移历史。 |
| State | 当前HMI对象状态，空闲中或动作处理中。 |

明确以上定义后，接下来以演示例程中的HMI引擎实例和演示例程的List画面为例，对照讲解HMI引擎的使用。

1. **使用方法**

由于HMI引擎始终处于动态，所以演示例程和具体的硬件平台有所关联，本讲解以模拟器环境为例。

首先，需要定位的时整体的环境入口，在DemoProc.c文件中。

模拟器启动并初始化完成时，会调用InitializeEngine函数，InitializeEngine函数中的调用InitializeEngine\_Internal函数完成HMI引擎实例的初始化和添加屏幕动作，整体逻辑处理比较简单，请自行参照源代码。此处不作详细描述。

初始化完成后，HMI引擎进入等待状态，接下来的所有操作全部围绕HMI引擎的HMI\_ProcessEvent函数，该函数会调用当前被激活画面对象的ProcessEvent函数指针指向的函数，并作后续处理。

打开DemoProc\src\List.c文件，此文件为演示程序中列表控件的演示处理。在此文件中，可以中找到前文所述的画面的五个动作的处理函数的定义：

|  |
| --- |
| HMI\_ENGINE\_RESULT HMI\_DemoList\_Initialize(void); |
| HMI\_ENGINE\_RESULT HMI\_DemoList\_Prepare(const void\* pstParameters); |
| HMI\_ENGINE\_RESULT HMI\_DemoList\_RefreshScreen(const void\* pstParameters); |
| HMI\_ENGINE\_RESULT HMI\_DemoList\_ProcessEvent(HMI\_EVENT\_TYPE eEvent, const HMI\_EVENT\* pstEvent); |
| HMI\_ENGINE\_RESULT HMI\_DemoList\_PostProcess(SGUI\_INT iActionResult); |

以及画面操作函数结构体数据的的定义：

|  |
| --- |
| HMI\_SCREEN\_ACTION s\_stDemoListActions =  {  HMI\_DemoList\_Initialize,  HMI\_DemoList\_Prepare,  HMI\_DemoList\_RefreshScreen,  HMI\_DemoList\_ProcessEvent,  HMI\_DemoList\_PostProcess  }; |

和List演示画面的定义：

|  |
| --- |
| HMI\_SCREEN\_OBJECT g\_stHMIDemo\_List =  {  HMI\_SCREEN\_ID\_DEMO\_LIST,  &s\_stDemoListActions  }; |

此时，如果用户按下键盘按键，则模拟器会捕捉键盘消息，并组织键盘数据然后调用EventProcess函数，由于当前激活画面是List画面，所以HMI的CurrentScreenObject 指针指向的是g\_stHMIDemo\_List，EventProcess函数会通过调用HMI\_ProcessEvent函数，调用g\_stHMIDemo\_List的ProcessEvent函数，也就是HMI\_DemoList\_ProcessEvent函数。

HMI\_DemoList\_ProcessEvent函数的主要处理如下：

|  |
| --- |
| switch(\*(parrKeyValue+1))  {  case KEY\_VALUE\_ENTER:  {  eProcessResult = HMI\_RET\_CONFIRM;  break;  }  case KEY\_VALUE\_ESC:  {  eProcessResult = HMI\_RET\_CANCEL;  break;  }  case KEY\_VALUE\_UP:  {  SGUI\_List\_SelectUpItem(&s\_stDemoListObject);  break;  }  case KEY\_VALUE\_DOWN:  {  SGUI\_List\_SelectDownItem(&s\_stDemoListObject);  break;  }  case KEY\_VALUE\_RIGHT:  {  if((\*(parrKeyValue+0) & KEY\_OPTION\_SHIFT) != 0)  {  SGUI\_List\_SetListItemValue(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Valid.Value, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Decimal.Value-1);  }  else  {  SGUI\_List\_SetListItemValue(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Valid.Value+1, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Decimal.Value);  }  break;  }  case KEY\_VALUE\_LEFT:  {  if((\*(parrKeyValue+0) & KEY\_OPTION\_SHIFT) != 0)  {  SGUI\_List\_SetListItemValue(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Valid.Value, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Decimal.Value+1);  }  else  {  SGUI\_List\_SetListItemValue(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Valid.Value-1, SGUI\_List\_GetListItemPtr(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex)->Decimal.Value);  }  break;  }  #if (\_SIMPLE\_GUI\_ENABLE\_DYNAMIC\_MEMORY\_ > 0)  case KEY\_VALUE\_F8:  {  SGUI\_List\_RemoveItem(&s\_stDemoListObject, s\_stDemoListObject.ControlVariable.SelectIndex);  SGUI\_List\_Refresh(&s\_stDemoListObject);  break;  }  case KEY\_VALUE\_F9: // Insert to head.  {  SGUI\_List\_InsertItem(&s\_stDemoListObject, &s\_arrstAppendListItems[0], 0);  SGUI\_List\_Refresh(&s\_stDemoListObject);  break;  }  case KEY\_VALUE\_F10: // Insert to intermediate.  {  SGUI\_List\_InsertItem(&s\_stDemoListObject, &s\_arrstAppendListItems[1], 5);  SGUI\_List\_Refresh(&s\_stDemoListObject);  break;  }  case KEY\_VALUE\_F11: // Insert to end.  {  SGUI\_List\_InsertItem(&s\_stDemoListObject, &s\_arrstAppendListItems[2], s\_stDemoListObject.Data.Count);  SGUI\_List\_Refresh(&s\_stDemoListObject);  break;  }  #endif  default:  {  break;  }  } |

可以看到，针对不同的按键处理，画面大的动作和返回值不尽相同，当用户按下回车时，返回值被设定为HMI\_RET\_CONFIRM，按键为ESC时，返回值被设定为HMI\_RET\_CANCEL，而其它时候均保持HMI\_RET\_NORMAL。

所以，当用户按下回车，EventProcess中HMI\_ProcessEvent的最终返回值为HMI\_RET\_CONFIRM，然后将此返回值传入HMI\_PostProcess后处理函数中继续处理，同理，此函数最终会调用List画面定义的后处理函数HMI\_DemoList\_ProcessEvent。在这个函数中，如果发现返回值为HMI\_RET\_CONFIRM，则认为用户在执行了确认操作，调用HMI\_Goto函数跳转至相应画面。如果是HMI\_RET\_CANCEL，则认为用户要返回，调用HMI\_GoBack函数，返回之前的画面。

至此，HMI引擎完成一次完整的动作响应。

1. **写在最后**

HMI引擎的出现，最早就是为了解决我想到的可能出现的栈空间耗尽的情况，将RAM尽可能多的割让给其他的处理，因为我认为GUI其实就是用于向人返回机器的运行结果和接受人的指令，自己本身的存在其实时一种对资源的浪费，如果不不考虑易用性，串口控制台其实是更好的选择。

当然，这个HMI引擎这只是对付当时在自己的一时之需，并没有经过更多的斟酌和打磨，只是想到可能会有其他人用到，所以就整合了进来，如果对这方面没有需求，HMI完全可以从SimpleGUI中剔除掉。

当然，最初创建HMI的目的也有提升代码易读性的初衷，但目前看来好像适得其反，应该还是我的设计存在问题吧。当然，后续HMI引擎一定会根据SimpleGUI的进化而改进的，我也有信心让他称为一个优秀的交互解决方案。

本文仅仅是对于HMI的简要说明，具体内容还请在阅读时参照源代码。HMI引擎本身用到了大量的函数指针，只要明白了各个函数指针的意义，理解起来应该还是比较容易的。

1. **联系开发者**

首先，感谢您对SimpleGUI的赏识与支持。

虽然最早仅仅作为一套GUI接口库使用，但我最终希望SimpleGUI能够为您提供一套完整的单色屏GUI及交互设计解决方案，如果您有新的需求、提议亦或想法，可以联系QQ 326684221或电子邮件xuyulin91@163.com，也可以在以下地址留言：

SimpleGUI@开源中国：<https://www.oschina.net/p/simplegui>

SimpleGUI@码云：<https://gitee.com/Polarix/simplegui>

本人并不是全职的开源开发者，依然有工作及家庭的琐碎事务要处理，所以对于大家的需求和疑问反馈的可能并不及时，多有怠慢，敬请谅解。

最后，再次感谢您的支持。