# 如何不断收取和处理数据包

例如：遥控器向主板发送数据，大约100ms发送一次。数据格式为：

开始字符（0XAA）——设备类型——设备地址——功能码——数据长度——数据（N个字节）——crc校验——结束字符（0X0D）。

主板需要不停接收和处理。

主板流程如果采用轮寻，流程可以如图 1：

\*\*\*\*注意：为什么还要加个读取数据函数？难道不能直接用中断函数中的BUF[]吗？

答：这是因为中断函数中断的数据并不稳定，随时有可能变化，例如writeindex。如果直接将此数据用于“数据处理函数”，会出现BUG。需要使用“读取数据函数”作为缓冲，以提供稳定的数据。



图 1 数据接收处理流程

# 如何使用printf函数打印输出串口数据

标准库函数的默认输出设备是显示器，要实现在串口或LCD输出，必须重定义标准库函数里调用的与输出设备相关的函数.   
      因printf()之类的函数，使用了半主机模式。使用标准库会导致程序无法运行,以下是解决方法:方法1.使用微库,因为使用微库的话,不会使用半主机模式. 如果使用的是MDK，请在工程属性的“Target“-》”Code Generation“中勾选”Use MicroLIB“这样以后就可以使用printf，sprintf函数了 方法2.仍然使用标准库,在主程序添加下面代码:    
FILE \_\_stdout;     
 在独立应用程序中，您不太可能支持半主机操作。 因此，必须确保您的应用程序中没有链接 C库半主机函数。  
  
问题：STM32如何使用printf函数进行串口输出。    
解答：寄存器版的/库函数版的  
第一步，#inlcude "stdio.h"              
第二步，在主函数文件中重写fputc()和fgetc()函数，如下：   
int fputc (intch,FILE\*f)   
  {      
          while(!(USART1->SR & USART\_FLAG\_TXE));   
         USART1->DR =(ch & 0x1FF);   
        return (ch);}   
int fgetc (FILE\* f)    
{      
       while (!(USART1->SR& USART\_FLAG\_RXNE));     
        return((int)(USART1->DR &0x1FF));   
}          
  第三步，工程属性中的Target项中的Code Generation 选择 UseMicroLIB。  即可使用printf进行串口输出。  
库函数版本的：​  
如果使用的是MDK，请在工程属性的“Target“-》”CodeGeneration“中勾选”Use MicroLIB“ 进行如上设置后编译一下，串口输出正确的数据。     
  
#ifndef \_\_UART\_INTERFACE\_H   
#define \_\_UART\_INTEFFACE\_H    
#include "stm32f10x\_lib.h"   
#include "stdio.h"       
void UART\_Init(unsigned long UART\_baud);//波特率，如115200   
int fputc(int ch, FILE \*f);    //fputc重定向  
int fgetc(FILE \*f); //fgetc重定向    
#endif 下面是.C的部分内容，省去了初始化的部分，自己看着写好了   
int fputc(int ch, FILE \*f）//重新定向到串口发送出去的数据  
{  
      USART\_SendData(USART1, (u8) ch);  
      while(!(USART\_GetFlagStatus(USART1,USART\_FLAG\_TXE) == SET))  
    {   }  
     returnch;   
}    
int fgetc(FILE \*f)  //重新定向到串口接收到的数据  
{  
      while(!(USART\_GetFlagStatus(USART1,USART\_FLAG\_RXNE) == SET))   
   {   }  
          return (USART\_ReceiveData(USART1));}   
注释： fgetc 和 fgetc是c语言的标准函数（形参是标准的） FILE \*f是文件指针，具备文件系统的操作系统有用。对于简单的无文件系统的嵌入式系统无用。

# EMWIN

## 移植

本次移植以自己编写驱动为前提。

1. 移植文件在STemWinLibrary522文件夹内。



1. Config文件夹内的“GUIConf”“GUIDRV\_Template”“LCDConf\_FlexColor\_Template”（或者“LCDConf\_Lin\_Template”）加入工程中。

Inc文件夹内所有加入工程中。

Lib文件夹中选择相应的一个库加入工程中。

OS文件夹中GUI\_X加入工程。

1. GUIConf.c：

主要修改#define GUI\_NUMBYTES (1024\*20) 。设定一个GUI使用的内存量，注意根据系统内存量修改。

GUIConf.h：

自己根据功能改。注意，其实所有功能都可以选择支持，即使不用。

1. LCDConf\_FlexColor\_Template.c

主要修改LCD\_X\_Config(void) 和LCD\_X\_DisplayDriver。

void LCD\_X\_Config(void) {

GUI\_DEVICE \* pDevice;

CONFIG\_FLEXCOLOR Config = {0};

GUI\_PORT\_API PortAPI = {0};

//

// Set display driver and color conversion

// 建造一个GUI\_DEVICE。

pDevice = GUI\_DEVICE\_CreateAndLink(GUIDRV\_TEMPLATE, GUICC\_M565, 0, 0);

// Display driver configuration, required for Lin-driver

//设定设备和虚拟设备的X和Y。

LCD\_SetSizeEx (0, XSIZE\_PHYS , YSIZE\_PHYS);

LCD\_SetVSizeEx(0, VXSIZE\_PHYS, VYSIZE\_PHYS);

//

// Orientation

//注意，必须要注释掉此处的源代码，否则会硬件错误。

//Config.Orientation = GUI\_SWAP\_XY | GUI\_MIRROR\_Y;

//GUIDRV\_FlexColor\_Config(pDevice, &Config);

//

// Set controller and operation mode

//因为需要我们自己写驱动，所以不需要后面的代码。

#if 0

PortAPI.pfWrite16\_A0 = LcdWriteReg;

PortAPI.pfWrite16\_A1 = LcdWriteData;

PortAPI.pfWriteM16\_A1 = LcdWriteDataMultiple;

PortAPI.pfReadM16\_A1 = LcdReadDataMultiple;

GUIDRV\_FlexColor\_SetFunc(pDevice, &PortAPI, GUIDRV\_FLEXCOLOR\_F66708, GUIDRV\_FLEXCOLOR\_M16C0B16);

#endif

}

int LCD\_X\_DisplayDriver(unsigned LayerIndex, unsigned Cmd, void \* pData) {

int r;

(void) LayerIndex;

(void) pData;

switch (Cmd) {

case LCD\_X\_INITCONTROLLER: {

//因为LCD初始代码写在外部，所以实际上此函数什么也不做。

// Called during the initialization process in order to set up the

// display controller and put it into operation. If the display

// controller is not initialized by any external routine this needs

// to be adapted by the customer...

//

// ...

return 0;

}

default:

r = -1;

}

return r;

}

1. GUIDRV\_Template.c

自己实现的LCD驱动就在此文件中。

必须至少要实现\_SetPixelIndex。此函数是描绘一个point。其它函数可以暂且不重写。

后期为了加强性能，必须实现其它诸如画线之类的函数。

1. GUI\_X.c

此文件中的GUI\_X\_GetTime和GUI\_X\_DELAY都需要自己编写。

不用重写，emwin也能用。暂不清楚其作用。若带操作系统，最好还是将其实现。

注意，在官方sample->GUI\_X文件夹中，有GUI\_X\_uCOS文件可以直接使用。

1. 启动注意事项：

STM32使用STEMWIN前，必须使能CRC时钟。（为了防止别的芯片盗用）

例如：

RCC\_AHBPeriphClockCmd(RCC\_AHBPeriph\_CRC,ENABLE);//开启 CRC 时钟

## 关于操作系统的说明

根据官方EMWIN说明文档，EMWIN的使用可以分为：

1. 无操作系统：

关于 emWin 的使用没有实际的限定。照例，需要在调用 GUI\_Init() 之后才能使用该软件。之后，任何 API 函数都可使用。如果使用了窗口管理器的回调机制，则必须定期调用 emWin 更新函数。这通常通过从超循环内调用GUI\_Exec()来完成。 各种模块化函数，如GUI\_Delay()和GUI\_ExecDialog()不应在循环中使用，因为它们会阻断其它软件模块。可使用不支持多任务 (#define GUI\_OS 0) 的默认配置；不需要内核接口例程。

1. 带操作系统，只有一个任务调用EMWIN

与（1）一致。

可使用不支持多任务 (#define GUI\_OS 0) 的默认配置；不需要内核接口例程。可使用任何实时内核，商用的或专用的。

1. 带操作系统，多个任务调用EMWIN

需要实现若干API接口，Lib文件需要使用带OS版本，宏定义也需要更改。

很少用，具体看EMWIN文档。

用的最多的是（2）。

## 如何加入触摸屏

参考《EMWIN5中文手册》第22章，内有详细说明。

普遍采用的最简单的方法：使用接口函数 void GUI\_TOUCH\_StoreState(int x, int y);

此函数即将触摸x和y（如果没有触摸，则x=y=-1）存入EMWIN库中。用户需要循环调用此函数，以向EMWIN通知其更新。其它事务由EMWIN自行处理。

## 关于窗口管理器（VM）的使用

1. 背景窗口：

初始化窗口管理器期间，会创建一个包含整个 LCD 区域的窗口作为背景窗口。此窗口的句柄为WM\_HBKWIN。WM 不会自动重绘背景窗口的区域，因为没有默认的背景颜色。也就是说如果创建了另一个窗口然后将其删除，则删除的窗口仍然可见。

1. 控件（小工具）：

控件根据其属性绘制自身，此操作在调用 WM\_Exec()、 GUI\_Exec() 或 GUI\_Delay() 时执行。  
在多任务环境中，通常由后台任务来调用 WM\_Exec() 并更新小工具（以及所有其他具有回调函数的窗口）。小工具的属性更改后，该小工具的窗口 （或部分窗口）会被标记为无效，但不会立即重绘。因此，代码段的执行速度会非常快。重绘由 WM 在稍后执行，或者通过为该小工具调用 WM\_Paint() 强制执行 （或直到重绘所有窗口时调用 WM\_Exec()）。

小工具如何通信：  
小工具通常作为子窗口创建，父窗口可为任何窗口类型，甚至是另一种小工具。为了确保同步，无论  
何时父窗口的任何子项有任何事件发生，通常都应通知父窗口。有事件发生时，子窗口小工具通过发  
送 **WM\_NOTIFY\_PARENT** 消息与其父窗口通信。

举例：如何获取button被按下

1. 先用GUIBuilder建立一个窗口和一个按钮button。
2. 在窗口的回调函数中，switch该窗口的信息ID，由于button是子窗口，所以其信息类型是“WM\_NOTIFY\_PARENT”。
3. 继续寻找其源头。

代码如下：

switch (pMsg->MsgId) {

case WM\_NOTIFY\_PARENT:

Id = WM\_GetId(pMsg->hWinSrc);

NCode = pMsg->Data.v;

switch(Id) {

case ID\_BUTTON\_0:

switch(NCode) {

case WM\_NOTIFICATION\_CLICKED:

//此处加入处理

break;

case WM\_NOTIFICATION\_RELEASED:

// 此处加入处理

break;

}

break;

}

break;

default:

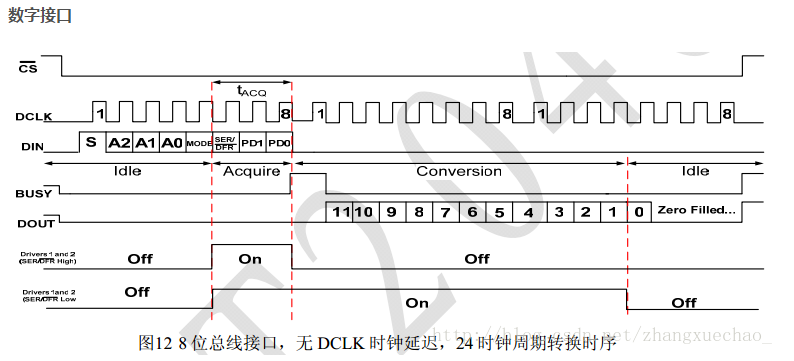
WM\_DefaultProc(pMsg);

break;

}

# 电阻触摸屏 XPT2046

1. 注意：XPT2046输出结果与SPI频率还存在关系。猜测原因：



在conversion时期完成模数转换，内部采用转换时间与其时间有关系。

可以使用36/8 MHZ