

本资料仅供内部使用！

STM32F207 基础版本使用说明

2013 年 7 月 20 日

修改记录

制定日期	生效日期	制定 / 修订 内容摘要	页数	版本	拟稿	审查	批准
2013.07.20		初稿	2	0.01	朱正晶		
2013.08.02		添加更详细内容	14	0.02	朱正晶		
2013.09.16		根据最新工程进行更新	16	0.03	朱正晶		
2013.10.11		添加驱动层 API 参考一章	30	0.04	朱正晶		
2013.12.06		添加 HAL API 参考一节	32	0.05	朱正晶		

目 录

1	文档组成部分	1
2	工程模块结构	2
2.1	APP 目录.....	3
2.2	USER 目录.....	3
2.3	BSP 目录.....	4
2.4	ST 目录	5
2.5	μC/OS-II 目录	6
2.6	Doc 目录.....	6
3	系统服务	7
3.1	TRACE 功能.....	7
3.2	SHELL 功能	8
3.2.1	添加专有 shell 命令.....	9
3.2.2	命令执行.....	10
3.3	函数返回值及参数检查	11
3.4	文件系统	11
3.5	硬件抽象层.....	11
3.6	添加更多系统服务	11
4	APP 应用程序编写	12
4.1	添加新任务	12
4.2	LED 应用程序	13
4.3	WATCHDOG 应用程序.....	13
5	系统参数配置	14
5.1	MALLOC 堆配置.....	14
5.2	μC/OS-II 系统配置	14
5.3	应用程序栈大小设置	15
5.4	时钟配置	15
6	系统版本号定义	16
7	MDK 工程管理	17
7.1	宏定义.....	17
7.2	工程管理	18
8	错误管理系统	19
8.1	错误分类.....	19
8.1.1	系统逻辑错误.....	19
8.1.2	硬件错误.....	19
8.1.3	系统错误（不可恢复）	19
8.2	错误处理	19

9	源代码编写风格	20
9.1	使用 SOURCE INSIGHT 编辑器编辑源代码	20
9.1.1	使用等宽字体	20
9.1.2	TAB 键设置	20
9.1.3	设置自动缩进	21
9.2	源代码编码例子	21
9.3	定义	21
10	硬件抽象层 (HAL) API 参考	22
10.1	USART 串口	22
10.1.1	查询方式	22
10.1.2	DMA 方式	24
10.2	SPI	26
10.2.1	初始化 SPI	26
10.2.2	去初始化	26
10.2.3	发送一个字节	26
10.2.4	接收一个字节	26
10.3	SRAM	27
10.3.1	初始化	27
10.3.2	写操作 (16 位)	27
10.3.3	读操作 (16 位)	27
10.3.4	写操作 (8 位)	27
10.3.5	读操作 (8 位)	27
10.4	COMX100 CANOPEN 主站模块	28
10.4.1	初始化 comX100 task API	28
10.4.2	发送 PDO API	28
10.5	CAN 总线	29
10.5.1	初始化	29
10.5.2	发送数据	29
10.5.3	接收数据	29
10.6	RTC	30
10.6.1	初始化	30
10.6.2	清空 4KB SRAM	30
10.6.3	写 4KB BKS RAM (8 位)	30
10.6.4	读 4KB BKS RAM (8 位)	30
10.6.5	写 4KB BKS RAM (32 位)	30
10.6.6	读 4KB BKS RAM (32 位)	30
10.6.7	写备份寄存器	31
10.6.8	读备份寄存器	31
10.6.9	设置 RTC 时间	31
10.6.10	读 RTC 时间	31
10.6.11	设置 RTC 日期	31
10.6.12	读 RTC 日期	32

1 文档组成部分

本文档对 STM32F207 基础版本作一个整体上的说明，包括：

- 1) 模块的划分
- 2) 系统服务
- 3) μ C/OS-II 应用程序编写
- 4) 系统相关参数配置说明
- 5) 系统版本号定义
- 6) MDK 工程管理方法
- 7) 错误管理模块
- 8) 源代码风格
- 9) 硬件抽象层（HAL）API 使用说明

2 工程模块结构

工程整体框图如图 2-1 所示。分为三个层次，分别为应用层、中间件、硬件层 MCU。

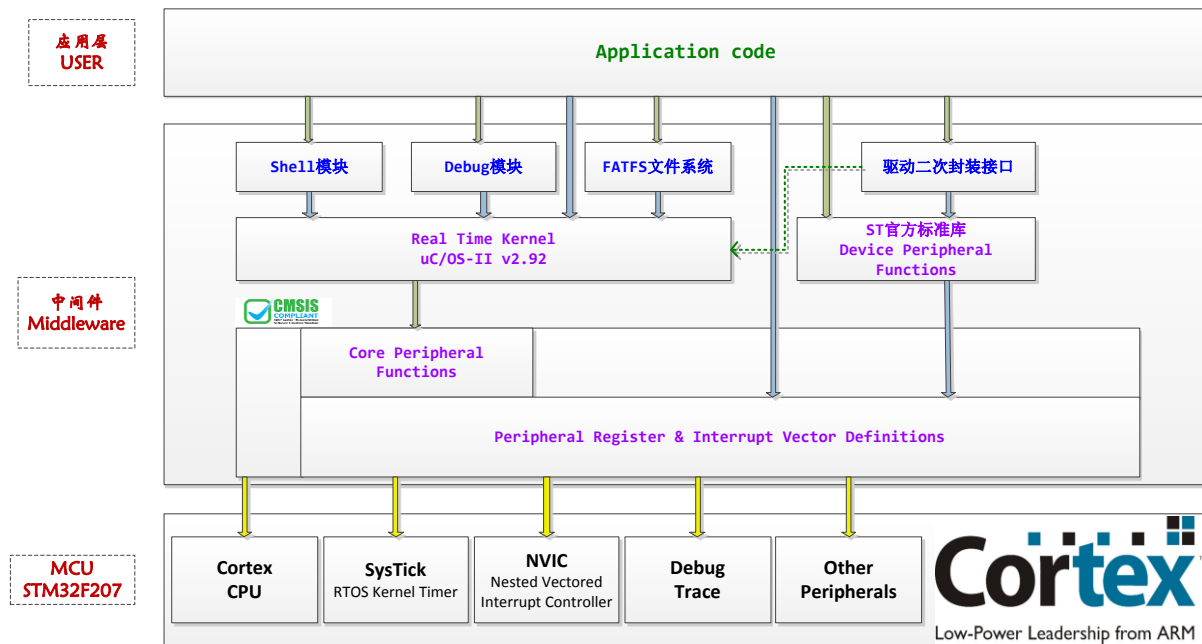


图 2-1 系统整体示意框图

根据以上框图，当前整个工程模块结构如图 2-2 所示（以中控板 ARM0 板为例），分为：

- | | |
|-------------------|---|
| ①APP | μC/OS-II 应用程序； |
| ②USER | 裸机程序（不带操作系统的应用程序）； |
| ③BSP | 板级支持包； |
| ④stdlib | ST 官网自带的库，包括标准库及 USB 库，后面根据工程的需要可能会再增加； |
| ⑤μC/OS-II | μC/OS-II 操作系统源代码； |
| ⑥FATFS | FAT 文件系统 |
| ⑦Hilscher_Toolkit | CANOpen 主站 Toolkit 工具包 |

一些复杂的子模块会有相应的文档进行更详细的说明。比如 FATFS 文件系统，CANOpen 驱动调试文档。

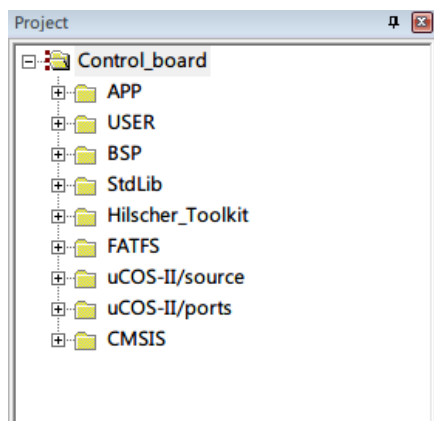


图 2-2 工程模块结构

2.1 APP 目录

此目录下为 μ C/OS-II 的应用程序，当前 APP 目录下有如下文件（图 2-2）。随着开发的进行，文件会持续添加。此文件夹中应用程序文件都是以“app_”作为头部，代表 application process。

名称	修改日期	类型	大小
 app.c	2013/8/1 13:33	C Source	10 KB
 app.h	2013/8/1 13:35	C/C++ Header	5 KB
 app_cfg.h	2013/8/1 14:44	C/C++ Header	6 KB
 app_comX100.c	2013/8/1 13:36	C Source	20 KB
 app_comX100.h	2013/8/1 13:36	C/C++ Header	5 KB
 app_shell.c	2013/8/1 16:14	C Source	10 KB
 app_shell.h	2013/8/1 14:40	C/C++ Header	5 KB
 os_cfg.h	2013/7/27 15:35	C/C++ Header	15 KB

图 2-2 APP 目录下文件

app_cfg.h 为系统配置文件，系统版本号、任务的优先级、任务栈大小都需要在这个文件中定义。

os_cfg.h 为 μ C/OS-II 的配置文件，一般不需要修改。

app.h 和 **app.c** 为整个系统的起始点，每个任务需要在 **app.c** 中添加任务启动代码（函数 **void app_start_task(void *p_arg)** 中）。具体的操作方法可以参考现有的代码。

每个应用程序建议使用一个单独的模块，命名为：**app_xxx.c** 和 **app_xxx.h**。这样每个人都可以并行工作，并且可以减少添加的功能模块对其他人模块的干扰，并且有利于系统的维护和更新。

2.2 USER 目录

该目录下包含的文件比较杂，主要是一些裸机下的程序。这些裸机程序也可以在 μ C/OS-II 的应用程序中使用。

名称	修改日期	类型	大小
IAP	2013/6/24 13:43	文件夹	
test_program	2013/7/8 17:09	文件夹	
can_test.h	2013/5/16 10:43	C/C++ Header	11 KB
common.c	2013/7/30 10:22	C Source	10 KB
common.h	2013/7/26 14:52	C/C++ Header	3 KB
includes.h	2013/7/31 8:34	C/C++ Header	6 KB
main.c	2013/7/26 14:01	C Source	7 KB
pid.c	2013/6/13 17:08	C Source	11 KB
pid.h	2013/5/31 15:07	C/C++ Header	6 KB
protocol.c	2013/6/9 7:25	C Source	9 KB
protocol.h	2013/6/3 16:35	C/C++ Header	5 KB
queue.c	2013/6/24 13:39	C Source	5 KB
queue.h	2013/7/20 9:10	C/C++ Header	5 KB
shell.c	2013/7/31 16:12	C Source	41 KB
shell.h	2013/7/31 16:05	C/C++ Header	21 KB
shell_cfg.h	2013/8/1 9:10	C/C++ Header	8 KB
stm32f2xx_conf.h	2013/7/26 14:15	C/C++ Header	8 KB
stm32f2xx_it.c	2013/8/1 14:33	C Source	13 KB
stm32f2xx_it.h	2013/5/16 10:44	C/C++ Header	7 KB

图 2-3 USER 目录下文件

IAP 文件夹中存放为在线升级程序。详细实现参考文档：《STM32 μ C/OS-II 稳定性测试子模块 IAP 实现.pdf》。

common.h 和 **common.c** 模块为一些常用的函数，字符串处理函数。大家有好的比较好用的函数可以添加到这个模块中。注意放到这个模块下的函数的依赖性。

main.c 为 **main** 函数所在文件，系统完成底层初始化后会调用 **main** 函数，裸机程序可以直接写在 **main** 函数中。

文件	文件功能描述
pid.h	PID 算法模块
pid.c	
protocol.h	电机驱动模块的上下位通信协议模块
protocol.c	
queue.h	一个简单的队列实现
queue.c	
shell.h	移植的一个 shell 应用程序，调试用
shell.c	
shell_cfg.h	

2.3 BSP 目录

BSP 目录下为 STM32F207 外设驱动二次封装模块以及一些外设模块的驱动（比如 comX100 CANopen Master, SRAM, Nor Flash 等等）。如图 2-4 所示（没有显示完全）。

名称	修改日期	类型	大小
GLCD	2013/5/16 10:43	文件夹	
USB	2013/6/6 21:08	文件夹	
adc_dac.c	2013/7/20 14:18	C Source	20 KB
adc_dac.h	2013/7/20 14:19	C/C++ Header	6 KB
can.c	2013/7/20 14:15	C Source	13 KB
can.h	2013/5/16 10:43	C/C++ Header	5 KB
comx100.c	2013/7/30 8:25	C Source	13 KB
comx100.h	2013/7/30 8:25	C/C++ Header	5 KB
delay.c	2013/5/16 10:43	C Source	8 KB
delay.h	2013/5/16 10:43	C/C++ Header	5 KB
ds18b20.c	2013/5/16 10:43	C Source	9 KB
ds18b20.h	2013/5/16 10:43	C/C++ Header	6 KB
encoder.c	2013/6/24 11:16	C Source	19 KB
encoder.h	2013/6/3 17:00	C/C++ Header	6 KB
flash_if.c	2013/6/22 15:23	C Source	8 KB
flash_if.h	2013/6/22 16:03	C/C++ Header	4 KB
fsmc_nor.c	2013/7/20 11:04	C Source	28 KB
fsmc_nor.h	2013/7/9 14:04	C/C++ Header	7 KB
gpio.c	2013/7/20 16:10	C Source	9 KB
gpio.h	2013/7/20 16:11	C/C++ Header	5 KB
motor.c	2013/7/20 14:25	C Source	11 KB
motor.h	2013/7/20 14:20	C/C++ Header	6 KB
pwm.c	2013/6/7 20:23	C Source	15 KB
pwm.h	2013/6/3 17:14	C/C++ Header	6 KB
fram.c	2013/7/1 14:16	C Source	15 KB

图 2-4 BSP 目录下文件

2.4 ST 目录

此目录下的源文件都是从 ST 官网上下载的库，ST 提供的库基本上都是开源的，具体的参考 ST 官方说明，这里不作介绍。

名称	修改日期	类型	大小
CMSIS	2013/5/16 10:43	文件夹	
STM32_USB_Device_Library	2013/6/6 15:18	文件夹	
STM32_USB_HOST_Library	2013/6/6 15:18	文件夹	
STM32_USB_OTG_Driver	2013/6/6 15:18	文件夹	
STM32F2xx_StdPeriph_Driver	2013/5/16 10:43	文件夹	

图 2-5 ST 目录结构

2.5 μ C/OS-II 目录

此目录下为 μ C/OS-II 系统的源代码以及和 STM32F207 相关的移植代码。

名称	修改日期	类型
 Ports	2013/5/16 10:43	文件夹
 Source	2013/5/16 10:43	文件夹

图 2-6 μ C/OS-II 系统源代码及移植代码

2.6 Doc 目录

此目录下包含在开发过程中形成的一些文档，版本信息及一些小工具。

名称	修改日期	类型	大小
 TOOL	2013/6/22 16:12	文件夹	
 版本说明.txt	2013/7/30 11:23	文本文档	5 KB
 版本说明.txt.bak	2013/7/30 11:23	BAK 文件	5 KB

图 2-7 Doc 包含的文件

3 系统服务

μC/OS-II 实时内核只包含任务调度及一些操作系统必须的功能（比如信号量、消息队列、块内存管理）。为了方便使用，我们在 μC/OS-II 的基础上添加了一些常见的系统服务，比如 SHELL 程序、TRACE 功能、ASSERT 功能、文件系统（FATFS）、硬件抽象层（HAL）等等。后面根据需要可能会添加 TCP/IP 模块，高级文件系统（支持 NAND FLASH）等等。

下面对已经实现的系统服务进行说明。

3.1 TRACE 功能

为了方便调试，本系统使用 STM32F207 的 USART1（可以换成其他 USART）作为调试串口，通过程序在运行过程中打印信息获取程序当前运行状态信息。有一点需要注意：打印信息是通过 USART1 的查询方式来工作的，也就是说程序如果打印信息就必须等待 USART 操作完成才能执行下面的程序。由于 USART 的速率远远没有 CPU 快，因此使用 TRACE 打印的效率会很低，平时在测试时显示一些 log 信息没有问题，但是程序完成准备发布时需要将无关的调试信息全部去掉。

目前使用了三个相关的宏另加一个 DUMP_DATA 宏：

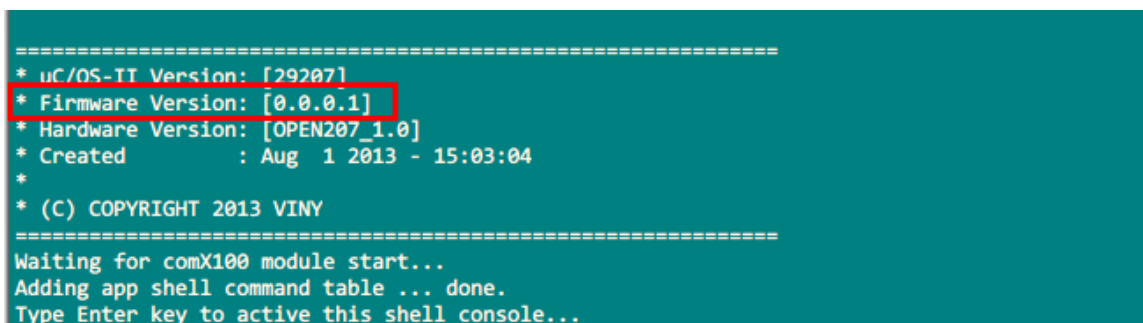
宏定义	说明
APP_TRACE(...)	在应用程序中调用的宏，加入 Mutex 防止数据打印混乱
TICK_TRACE(...)	在调试信息最前面打印系统 tick 数，方便应用程序估算程序执行时间
TRACE(...)	在中断服务程序中使用，没有 Mutex 保护，裸机程序也可使用
DUMP_DATA(...)	通过调试串口打印出整块数据

三个宏都使用可变参数，使用时注意不要打印浮点型变量，这个 BUG 目前还没解决。可能和 C 库有关。任何使用 C 库的地方都不能打印浮点型变量，比如：sprintf, printf...

EXAMPLE 1:

```
APP_TRACE(" * Firmware Version: [%s]\r\n", FIRMWARE_VERSION);
```

终端打印效果如图 3-1:



```
=====
 * uC/OS-II Version: [29207]
 * Firmware Version: [0.0.0.1]
 * Hardware Version: [OPEN207_1.0]
 * Created      : Aug  1 2013 - 15:03:04
 *
 * (C) COPYRIGHT 2013 VINY
=====
Waiting for comX100 module start...
Adding app shell command table ... done.
Type Enter key to active this shell console...
```

图 3-1 APP_TRACE()使用

EXAMPLE 2:

```
TICK_TRACE("Device Type autodetection: [%s]\r\n", FIRMWARE);
```

终端打印效果如图 3-2:

```
*
* (C) COPYRIGHT 2013 VINY
=====
Waiting for comX100 module start...
Adding app shell command table ... done.
Type Enter key to activate this shell console...
comX100 module init...
[ 1635]Device Type autodetection: Flash Based Device found!
[ 1641]Device Info:
[ 1643] - Device Number : 1531500
[ 1647] - Serial Number : 20052
```

图 3-2 TICK TRACE()

使用这个宏会在打印的调试信息前加上当前系统 tick 数，方便查看程序执行时间，每个 tick 间隔为 1 毫秒。

EXAMPLE 3:

```
TRACE("RECV DATA: [%d]\r\n", ch);
```

由于 APP_TRACE 和 TICK_TRACE 加了互斥量来防止其他应用程序抢占正在打印的任务，不加保护可能会造成打印的信息错乱。但是我们知道在中断服务程序不能使用 Mutex 或者等待信号量。因此我们提供了 TRACE 宏，这个宏直接将数据送往串口打印，没有互斥量保护。

另外，如果项目中有用到串口，请选择其他串口号，比如 UART2, UART3...

EXAMPLE 4:

```
DUMP_DATA(u8* p_data, u32 len);
```

函数 DUMP_DATA 能以十六进制打印数据, 特别适合打印数据包。下图 3-3 为使用 DUMP_DATA 函数打印的例子 (从第三行开始)。每行打印十六字节, 右边是对应的可显示的 ASCII 码。

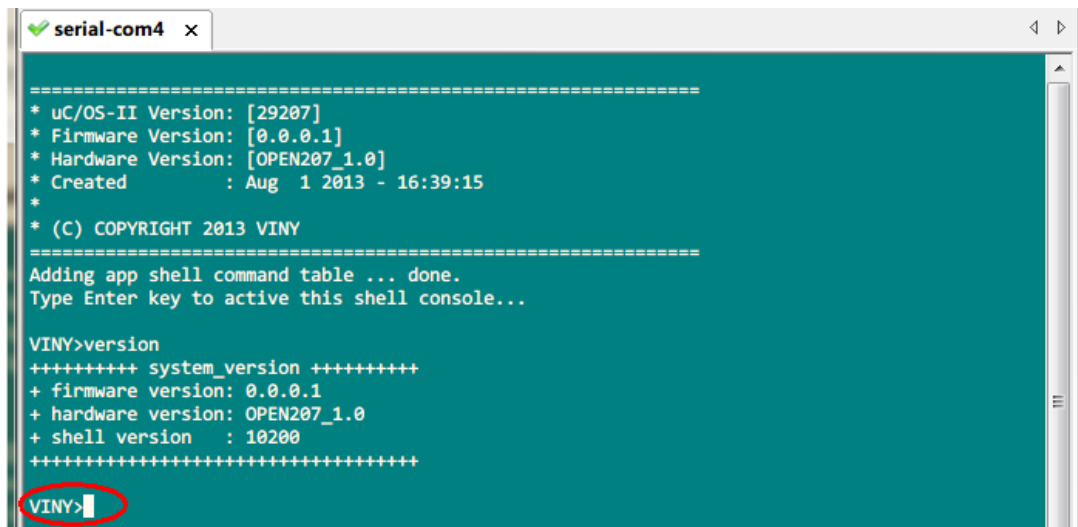
[illegible]

图 3-3 DumpData 打印示例

3.2 SHELL 功能

使用过 Linux 系统的同学可能都知道 **shell**（命令解析器），很强大的一个工具。我们移植了 Micrium 公司的 shell 代码实现了一个简单的 **shell**。

效果如图 3-4:



```

=====
* uC/OS-II Version: [29207]
* Firmware Version: [0.0.0.1]
* Hardware Version: [OPEN207_1.0]
* Created      : Aug  1 2013 - 16:39:15
*
* (C) COPYRIGHT 2013 VINY
=====
Adding app shell command table ... done.
Type Enter key to activate this shell console...

VINY>version
+++++++ system_version +++++++
+ firmware version: 0.0.0.1
+ hardware version: OPEN207_1.0
+ shell version   : 10200
+++++++
VINY>

```

图 3-4 SHELL 程序执行效果

开机后按“Enter”键激活 **SHELL** 程序，然后会出现 **VINY>** 等待命令的输入，作为参考上面截图中已经输入一个 **version** 命令打印系统的版本信息。

SHELL 程序其实也是一个任务，他一直在等待串口的数据，一旦有数据即根据当前数据和之前的数据进行处理。

3.2.1 添加专有 shell 命令

为什么要添加相应的命令？

如果不是处在调试模式下，程序执行时我们无法实时查看当前的值。我们可以对我们写的程序模块留一个后门，在这个后门程序中查看我们感兴趣的变量或者执行相应的功能。

添加方法如下：

在 **app_shell.c** 文件中的 **App_ShellAppCmdTbl[]** 数组中添加命令，命令格式为：

第一项：**SHELL_HEADER**+命令字符串。

第二项：然后加上对应这条命令的函数名函数根据自己的需要实现。本系统已经实现了三条命令，如图 3-5 所示：

```

00077: //=====
00078: // define your own shell commands here...
00079: //=====
00080: static SHELL_CMD App_ShellAppCmdTbl[] = {
00081:     // you can follow this command to create your own commands...
00082:     {SHELL_HEADER"test"          ,App_TestShellCmd  },
00083:     // show system version information on the console
00084:     {SHELL_HEADER"version"       ,get_system_version },
00085:     // show all the commands on the console
00086:     {SHELL_HEADER"help"          ,app_help           },
00087:
00088:     // Don't delete this command line!!!
00089:     {0                            ,0                  }
00090: };
00091:
00092:

```

图 3-5 添加命令数组

可以参数函数 **App_TestShellCmd** 来定义自己的 **shell** 命令，如图 3-6 所示：

```

00039: static CPU_INT16S App_TestShellCmd( CPU_INT16U i,
00040:                                     CPU_CHAR **p_p,
00041:                                     SHELL_OUT_FNCT p_f,
00042:                                     SHELL_CMD_PARAM* p)
00043: {
00044:     APP_TRACE("App_TestShellCmd\r\n");
00045:
00046:     return 0;
00047: }
00048:

```

图 3-6 函数实现

3.2.2 命令执行

按“ENTER”键出现 **VINY>** 提示符，在提示符后面输入命令字符，然后按“ENTER”键就会根据命令字符调用到指定的函数。输入 list 命令即可列出当前系统所支持的所有命令。

命令	说明	备注
app_test	测试命令	命令前面加了 app_，这是模块处理过程中需要的命令头。我们在输入命令时不需要加这个头部。
app_version	查看系统版本信息	
app_list	查看系统支持的命令	
app_top	查看 μ C/OS-II 系统状态信息	
app_restart	重启系统	
app_flash	spi flash 相关测试命令	
app_fatfs	FATFS 文件系统相关命令	
...	...	

图 3-7 为 list 和 version 两命令执行的例子。

```

VINY>list
----- support commands -----
app_test
app_version
app_list
app_top
app_restart
app_flash
app_fatfs

VINY>version
+++++++ system_version ++++++++
+ firmware version: 0.0.3.0      +
+ hardware version: OPEN207 1.0  +
+ shell version   : 10200        +
+ uC/OS-II version: 29207        +
+ build           : Sep 16 2013/11:16:27 +
+                                     +
+          (C) COPYRIGHT 2013 VINY  +
+++++++
VINY>

```

图 3-7 命令执行结果

3.3 函数返回值及参数检查

为了方便检查函数返回值以及函数参数的合法性，我们在调试过程中可以使用宏 `assert_param(expr)`，调试结束可以关闭以提高程序执行性能。

在文件中 `stm32f2xx_conf.h` 定义 `USE_FULL_ASSERT` 宏打开 `assert_param(expr)`，此函数在 `main.c` 中实现。如果参数有错误，`expr` 为 `false` 也就是 0 的话，会在终端打印图 3-8 所示的内容。45 是出错行号，后面紧跟文件名，这样我们就能很容易定位出错位置，方便调试。最新版的 `assert_param` 加入了是哪个任务调用这个宏，更容易确定出错位置。

```

=====
Adding app shell command table ... done.
Type Enter key to active this shell console...

VINY>
VINY>
VINY>test
App TestShellCmd
==> [45] - [..\APP\app_shell.c]

```

图 3-8 `assert_param(expr)` 调用

3.4 文件系统

文件系统我们采用了 FATFS 0.9c，FATFS 专门为小型嵌入式系统设计，开放源代码，代码尺寸小，支持多种存储媒介。详细的移植步骤参考《SPI FLASH 及 FATFS 文件系统调试.pdf》。

FATFS 提供的系统 API 说明在源代码目录下。

3.5 硬件抽象层

当前的硬件抽象层（HAL）只对 ST 提供的库作了二次封装，对调用者提供更简单的 API。因此不能算是真正意义上的硬件抽象层。

实现一个真正意义上的硬件抽象层还有很多事情要做，如果我们真的需要可以考虑其他操作系统。比如：RT-Thread、uCLinux、eCos 等等，这些系统的功能比 μ C/OS-II 强大的多，但是消耗的资源也相对较多。

3.6 添加更多系统服务

大家在工作和学习过程中肯定会遇到一些好的代码或者想法，可以直接将这些好想法做成一个系统服务，提供给大家使用。

4 APP 应用程序编写

先大概了解下本系统启动流程。开机后从内部 FLASH 取指运行（BOOT0 设置为 0），然后进行底层初始化（`startup_stm32f2xx.s` 文件中），最后跳转到 C 文件 `main` 函数中（`main.c`）。在 `main` 函数中我们初始化 $\mu\text{C}/\text{OS-II}$ 系统，启动一个任务 `app_start_task`（`app.c`）。如图 4-1:

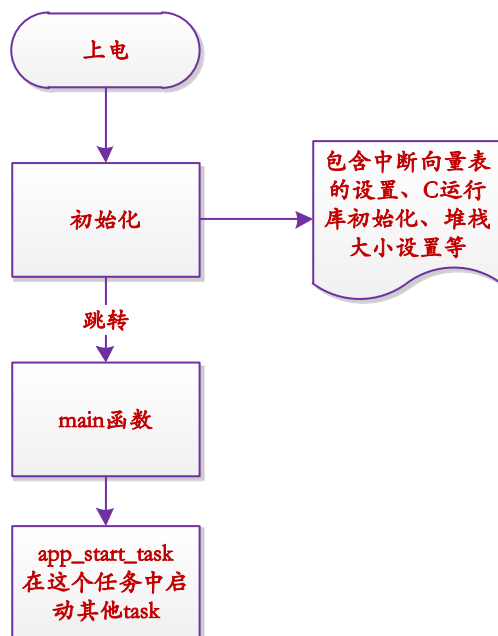


图 4-1 系统启动说明

4.1 添加新任务

以 `shell task` 为例说明添加新任务的方法。图 4-2 为启动代码:

```

00099:
00100:  /* shell module task *****/
00101:  os_err = OSTaskCreateExt((void (*)(void *)) shell_task,
00102:                          (void *) 0,
00103:                          (OS_STK *) &shell_task_stk[SHELL_TASK_STK_SIZE - 1],
00104:                          (INT8U *) SHELL_TASK_PRIO,
00105:                          (INT16U *) SHELL_TASK_PRIO,
00106:                          (OS_STK *) &shell_task_stk[0],
00107:                          (INT32U *) SHELL_TASK_STK_SIZE,
00108:                          (void *) 0,
00109:                          (INT16U *) (OS_TASK_OPT_STK_CLR | OS_TASK_OPT_STK_CHK));
00110:
00111:  assert_param(OS_ERR_NONE == os_err);
00112:
00113:  #if OS_TASK_NAME_EN > 0
00114:    OSTaskNameSet(SHELL_TASK_PRIO, (INT8U *) "shell module task", &os_err);
00115:  #endif
00116:

```

图 4-2 任务启动代码

① 首先在 `app_cfg.h` 中定义任务 `task` 的优先级和任务栈大小。如图 4-3:

注意: 因为每个栈元素定义为 `unsigned int`，定义 128 则栈大小为 $128 \times 4 = 512$ 字节。优先级不能有重复，要不然任务创建会失败。栈空间其实就是一个数组，本任务定义为:

`static OS_STK shell_task_stk[SHELL_TASK_STK_SIZE];` ;放在文件 `app.c` 中


```

00029: //=====
00030: // define your task priority and stack here...
00031: //=====
00032: /* COMX100 task priority */
00033: #define COMX100_TASK_PRIO          11u
00034: /* COMX100 task stack size */
00035: #define COMX100_TASK_STK_SIZE      1024u
00036:
00037:
00038: /* SHELL task priority */
00039: #define SHELL_TASK_PRIO            58u
00040: /* SHELL task stack size */
00041: #define SHELL_TASK_STK_SIZE        256u
00042:

```

图 4-3 任务优先级和栈大小定义

② shell_task 的编写

定义:

```

void shell_task(void *p_arg)
{
    while(1)
    {
        //任务代码...
        //在下次循环前必须调用 OS 相关 API 比如: 任务 delay,
        //信号量 pend 等
    }
}

```

注意: 每个任务不能无限制循环, 必须调用 OS 提供的相关 API, 具体可以参考第三讲视频《μC/OS-II 应用》。FTP 资料地址:

[/卫宁医疗/部门/软件部/学习/M3 学习资料专区/Cortex M3 学习报告视频/第三讲/](#)

4.2 LED 应用程序

LED 任务实现了一个简单 LED 亮灭功能, 每隔 50 毫秒亮灭一次。优先级比 watchdog 任务优先级高一级。通过观察 LED 灯即可大致看出当前系统的负载的情况, 后续可以更改指示灯的状态 (比如更改亮灭频率) 来指示系统的当前状态, 方便调试和观察。

4.3 Watchdog 应用程序

优先级为最低, 超过 3 秒没有“喂狗”系统将重启, 在重启后打印是因为 Watchdog 引起的重启。发生这样的情况一般是因为其他应用程序长时间占用 CPU, 使“喂狗”操作无法执行。注意, 调试阶段最好关闭 watchdog 功能, 防止干扰正常的调试功能。

5 系统参数配置

本系统基本参数已经配置好，本小节对这些参数的配置进行说明，方便大家了解整个系统以及今后可以根据需求进行修改。

5.1 malloc 堆配置

在文件 `startup_stm32f2xx.s` 中可以设置堆（Heap）大小，将 EQU 后面的值改为所需要的大小就可以了。当前为 `0x00002000`（8KB）。

注意： malloc 为线程不安全函数，在操作系统下调用需要加保护。

```
; <h> Heap Configuration
;   <o>   Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>
Heap_Size      EQU      0x00002000
                AREA     HEAP, NOINIT, READWRITE, ALIGN=3
```

5.2 μ C/OS-II 系统配置

μ C/OS-II 为高度可定制的 RTOS，我们根据需要在 `os_cfg.h` 文件中进行配置。

当前需要关注的配置项为：

① 最低优先级， μ C/OS-IIv2.92 支持 254 个任务，因此最低优先级可以设置为 254。目前设为 64，如果后面应用程序数量大于 64 可以调整 `OS_LOWEST_PRIO` 的值。

```
/* Defines the lowest priority that can be assigned ... */
/* ... MUST NEVER be higher than 254! */
#define OS_LOWEST_PRIO          64u
```

② 几个默认任务的栈大小

μ C/OS-II 默认情况下启动了三个任务，分别为定时器任务，统计任务，空闲任务：

```
/* ----- TASK STACK SIZE ----- */
#define OS_TASK_TMR_STK_SIZE    128u
#define OS_TASK_STAT_STK_SIZE   128u
#define OS_TASK_IDLE_STK_SIZE   128u
```

需要注意的是定时器任务的栈大小，如果后面使用到定时器函数，要注意栈大小不要超过 `128*4 = 512Bytes`。

③ 系统 tick，也就是每秒 tick 数，现在定义为 1000，1ms 一次 tick 中断。

```
/* Set the number of ticks in one second */
#define OS_TICKS_PER_SEC        1000u
```

5.3 应用程序栈大小设置

[见 4.1 节 添加新任务](#)

5.4 时钟配置

待完成

6 系统版本号定义

版本号随着开发的进行需要进行升级以示区别。目前有两个主版本号，一个是固件版本号 Firmware version，另一个是跟硬件相关的控制板版本号 board version。

目前定义如下：

```
#define FIRMWARE_VERSION          "0.0.4.0"

#if defined(CONTROL_MAIN_BOARD)
#define BOARD_VERSION             "CONTROL BOARD 0.1"
#elif defined(SLAVE_BOARD_1)
#define BOARD_VERSION             "SLAVE BOARD[1] 0.1"
#elif defined(SLAVE_BOARD_2)
#define BOARD_VERSION             "SLAVE BOARD[2] 0.1"
#elif defined(SLAVE_BOARD_3)
#define BOARD_VERSION             "SLAVE BOARD[3] 0.1"
#endif /* BOARD_VERSION */
```

7 MDK 工程管理

我们现有项目 Triam 分离机包含四块控制板。每块控制板都有一部分代码是相同的，比如 RTOS μ C/OS-II 内核，相关的系统服务等等。如果将每块板代码分开管理，有些公用的代码改一处的，必须同时更新四块板工程，这个增加了系统维护成本，出错的概率也大大提高。因此我们考虑在同一个源代码工程下通过不同的宏来区分不同的控制板。

实现方法如下：

在源代码目录下有 MDK 目录，在 MDK 目录下细分各个控制板的 MDK 工程。图 7-1 为当前 MDK 目录下的文件夹分类。

名称	修改日期	类型	大小
MDK_Control	2013/10/9 14:57	文件夹	
MDK_Slave_motor_test_project	2013/9/30 11:11	文件夹	
MDK_Slave1	2013/9/30 16:12	文件夹	
MDK_Slave2	2013/9/30 10:34	文件夹	
MDK_Slave3	2013/9/30 11:13	文件夹	

图 7-1 MDK 目录

7.1 宏定义

通过不同的宏定义来区分不同的工程。定义方法如下：

① 顶层宏

在 MDK (Microcontroller Development Kit) 工程的 C/C++ 选项卡下有 Define 文本框。在这里我们定义每个工程的顶层宏，下图为 Control board 的定义，此处定义为 **CONTROL_MAIN_BOARD**。

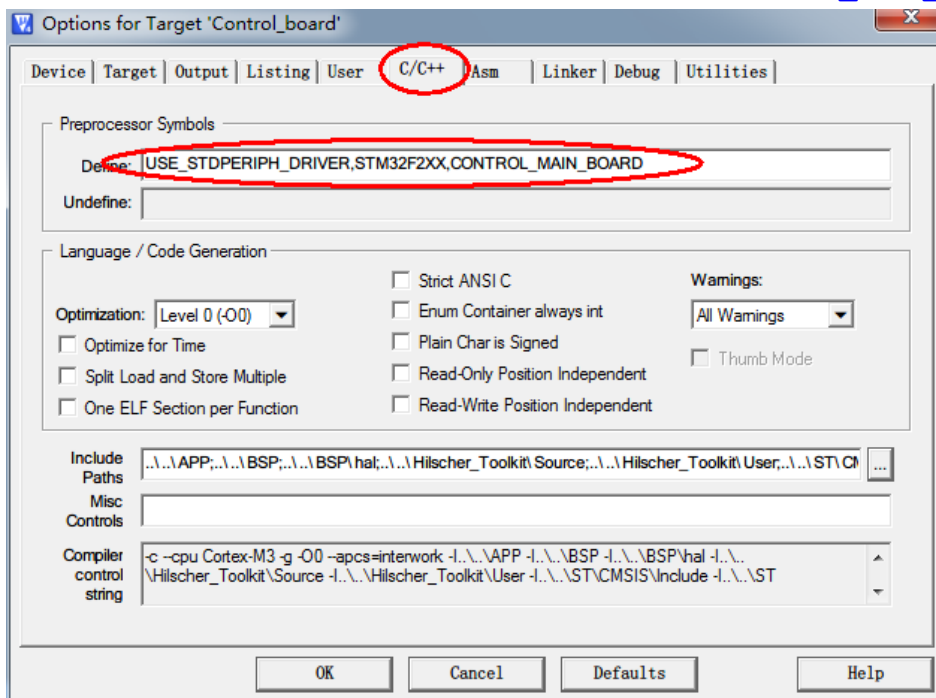


图 7-1 顶层宏定义

② 二级宏

根据每个工程的不同定义二级宏，定义在最顶层的头文件 `stm32f2xx_conf.h` 中。

定义方法

每个工程使用顶层宏来扩展相应的二级宏。图 7-2 所示为 control board 定义，control board 目前定义了许多 **FTR (feature)**。每个 **FTR** 最好都加上说明，方便使用。

```
00101: //
00102: // main control board with uCOS-II
00103: // This board use comX100 module for CANOpen master
00104: //
00105: #if defined(CONTROL_MAIN_BOARD)
00106:     #define _USE_UCOS_FTR_           /* use uCOS-II */
00107:     #define _COMX100_MODULE_FTR_     /* use comX100 module for CANOpen master */
00108:     //#define USE_STM32_CAN_DEBUG    /* use STM32 CAN to get CANBUS data from CANOpen */
00109:     #define CFX_TOOLKIT_HWIF         /* Toolkit */
00110:     //#define USE_UART3_TRACE_FTR    /* debug comX100 module with CAN and output CAN bus data to UART3 */
00111:     #define USE_SPI_FLASH_AT45DB_FTR /* SPI FLASH feature */
00112:     #define USE_FATFS_FTR            /* for FATFS file system */
00113:     #define USE_USART3_COMM_FTR      /* use USART3 to communication with IPC */
00114:
00115:     // SRAM definition, if you want to USE sram you need
00116:     // to uncomment the MACRO
00117:     #define USE_ISSI_SRAM_FTR
00118:
00119:     // RTC and BKPSRAM feature
00120:     #define USE_RTC_BKP_FTR
00121:
00122:     // debug message output serial definition
00123:     #define USE_USART1_TO_DEBUG
00124:     //#define USE_USART4_TO_DEBUG
00125:
00126:     // which board do you use?
00127:     // only one board can be selected
00128:     //#define USE_OPEN_207Z_BOARD
00129:     //#define USE_OPEN_207V_BOARD
00130:     #define USE_ARM0_REV_0_1
00131:
00132: #endif /* CONTROL_MAIN_BOARD */
00133:
00134: //
00135: // slave board 1 with uCOS-II
00136: //
00137: #if defined(SLAVE_BOARD_1)
00138:     #define _USE_UCOS_FTR_           /* use uCOS-II */
00139:     #define _NETX_MODBUS_FTR_        /* use NETX50 module for CANopen slave */
00140:     #define USE_USART3_COMM_FTR      /* use USART3 to communication with NETX50 module */
00141:     #define USE_USART1_TO_DEBUG
```

图 7-2 二级宏定义方法

7.2 工程管理

由于我们有多个工程对应不同的控制板，因此需要专门的人员对这些控制板进行单独管理。每人负责一块控制板。

我们采用相同功能代码由同一人负责的方法，也就是说假如一个功能项每块控制板都需要使用，那么就由你来负责各个控制板的代码编写。

每个工程的负责人需要对相应的工程整个代码都要熟悉，需要对负责的工程进行测试、确定出错位置并将问题让相关人员解决。

此部分具体实施细则则由刘工统筹。

8 错误管理系统

系统在运行中任何时刻都可能发生错误，我们一般都是通过打印 log 的方式来。考虑到我们的系统是医疗方面的，对安全性要求非常高。因此我们必须认真对待每一个错误。

8.1 错误分类

8.1.1 系统逻辑错误

系统逻辑错误是指程序员编写的错误功能代码，比如：程序运行到了不可能进入的路径。这种错误一般情况下都能在测试阶段发现，但有时因为程序路径很复杂，不可能全部测试到。或者就是程序员编写代码时有些路径没作处理。在我们的医疗系统中，每个程序的分支路径都必须处理，不能存在侥幸心理。一个好的程序往往是 20% 的正常路径代码加上 80% 的出错处理代码。因此我们要重视每一个程序分支路径。

8.1.2 硬件错误

我们使用的 MCU 中不仅仅包含 CPU，它还包括很多外设，每个外设都有一定的自我管理功能，我们也叫它自我管理 I/O 系统。这些自我管理 I/O 系统在运行过程中可能会出现一些错误，比如：CAN 总线通信错误，USART 通信错误，DMA 传输失败等等。

这些错误是怎么产生的呢？

我们以 CAN 总线错误作为例子。CAN 总线协议规定了一些标准，大部分和硬件相关的都由硬件帮我们处理了，但是如果 CAN 总线上出现了协议规定的不能出现的错误，那么硬件也不知道怎么办了。这时就需要 CPU 来介入处理了。

我们应该怎么处理这些硬件错误？

硬件错误都有相应的寄存器，初始化时我们打开发生错误的中断，这样在硬件错误发生时就会产生中断，我们在中断服务程序中读取错误寄存器并进行相应的处理。所以在编写相应的驱动程序时必须把硬件提供的错误都加到代码里，根据实际情况进行处理。

8.1.3 系统错误（不可恢复）

上面两种错误情况经过程序的适当处理是可以恢复的。但是如果运行过程中函数指针，变量指针指向错误的地址或者堆栈溢出，这些情况一旦发生系统即发生崩溃，我们认为这是不可恢复错误。好在大部分情况下我们可以通过严格的测试在开发阶段解决掉。

一些特殊的没有解决的错误可以通过 watchdog 来监控，但这是最后的办法。开发阶段和测试阶段的 watchdog 重启必须解决掉。

8.2 错误处理

创建一个监控 task 专门实时查看当前系统状态，一旦发生错误即根据错误的类别进行处理。实施细则待完成。。。

9.1.3 设置自动缩进

Source insight 默认的缩进不符合我们的要求，图 7-3 为修改方法。

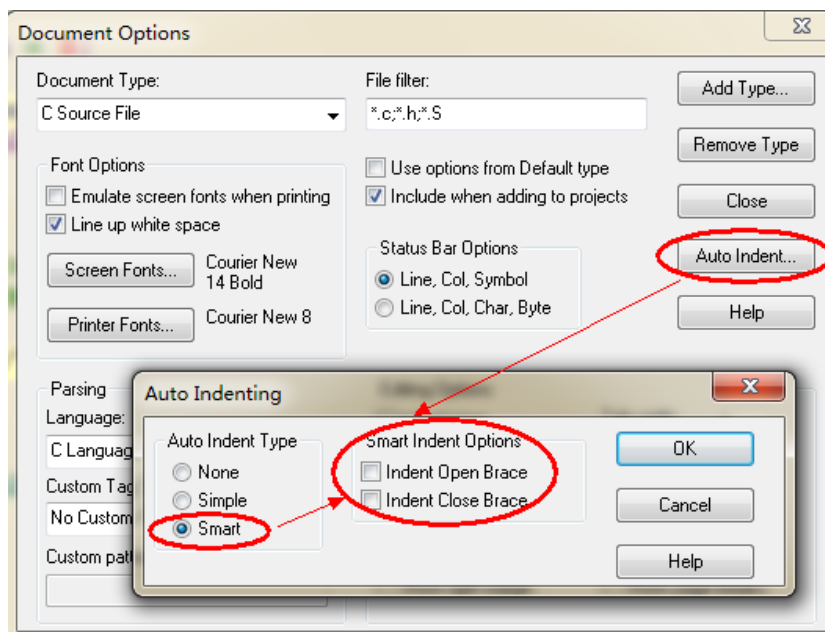


图 7-3 缩进设置选项

9.2 源代码编码例子

下图为一个函数的编码风格，作为参考。

```

00079: /**
00080:  * show all support commands
00081:  *
00082:  */
00083: static CPU_INT16S app_help( CPU_INT16U i, CPU_CHAR **p_p, SHELL_OUT_FNCT p_f, SHELL_CMD_PARAM* p)
00084: {
00085:     SHELL_MODULE_CMD* p_module_cmd;
00086:     SHELL_CMD* p_cmd;
00087:     uint32_t cmd_pos = 0;
00088:
00089:     APP_TRACE("==== support commands ====\r\n");
00090:     p_module_cmd = ShellModuleCmdUsedPoolPtr;
00091:     while(p_module_cmd != 0) {
00092:         cmd_pos = 0;
00093:         p_cmd = p_module_cmd->CmdTblPtr;
00094:         while(p_cmd[cmd_pos].Name)
00095:         {
00096:             APP_TRACE("%s\r\n", p_cmd[cmd_pos].Name);
00097:             cmd_pos++;
00098:         }
00099:         p_module_cmd = p_module_cmd->NextModuleCmdPtr;
00100:     }
00101:     return 0;
00102: }
00103:
00104: } ? end app_help ?
00105:

```

局部变量最好对齐

注意 != 两边都留有空格

一个TAB间隔，最好不要使用四个空格代替

代码后面不要留任何“看不见”的字符，比如空格，TAB

这个符号两边不要放空格

留一个空行

9.3 定义

上面对 source insight 的使用和例子有了一个基本了解，我们现在详细定义一下代码风格。具体请阅读《华为编码规范.pdf》

10 硬件抽象层（HAL）API 参考

本章节对 ST 标准库进行的二次封装进行说明。

ST 官方标准库（STLIB）只对寄存器级别的操作进行了封装，使用库函数操作相应的外设比直接读写寄存器方便了许多，但是每次操作还是需要调用很多函数，不方便使用。基于此，我们对 STLIB 进行了二次封装。封装的主要目的是屏蔽外设驱动使用的复杂性，对应用程序提供最简单的接口。

10.1 USART 串口

STM32F2xx 系列有 4 个全功能的串口（USART）和 2 个普通 UART，速度最高达到 7.5Mbit/s。为了使用方便，我们在 ST 官方库的基础上对串口驱动作了进一步的封装，使得使用更加简单。

USART 可以使用查询方式和 DMA 方式来进行数据接收和发送。查询方式简单但是效率差，大批量数据收发建议使用 DMA 方式。现在 USART 二次封装 API 支持 DMA 和 USART 两种方式。

10.1.1 查询方式

10.1.1.1 串口初始化 `init_usart`

```
uint8_t init_usart(COM_TypeDef COMx, u32 baud_rate, u8 word_length,  
parity_mode parity, usart_mode_type usart_mode);
```

参数

COMx	COM 口序号，COM1-COM7 可选
baud_rate	COM 口波特率
word_length	帧数据长度
parity	奇偶校验模式
usart_mode	配置串口模式，有 DMA 和中断方式可选

返回值

0: 成功
1: 失败

函数用例

例如，将 USART3 设置为波特率为 115200, 8 个数据位，1 个停止位，没有奇偶校验，使用中断模式。

```
init_usart(COM3, 115200, 8, PARITY_NONE, USART_MODE_INT);
```

10.1.1.2 发送数据 `usart_send_data`

```
void usart_send_data(COM_TypeDef COMx, uint8_t ch);
```

参数

COMx	COM 口序号，COM1-COM7 可选
ch	发送的一个字符

返回值



无

函数用例

发送字符'c'

```
usart_send_data (COM3, 'c');
```

还提供了一个发送字符串的函数 **usart_send_string**

```
void usart_send_string (COM_TypeDef COMx, uint8_t* str);
```

参数

COMx COM 口序号, COM1-COM7 可选

str 字符串, 以'\0'结尾

返回值

无

函数用例

```
usart_send_data (COM3, "Hello, world!\r\n");
```

10.1.1.3 接收数据

在 USART 接收数据的中断服务程序中接收字符数据, 将数据通过消息队列的形式发送到相应的应用程序中。

10.1.2 DMA 方式

DMA 相比查询方式复杂一些，初始化时需要打开相应的 DMA 通道，需要做到连续收发数据还要进行 memory 自动 reload。为此我们使用了两个 task 专门管理 USART DMA 收发。

串口数据收发采用 USART DMA 方式框图如下。

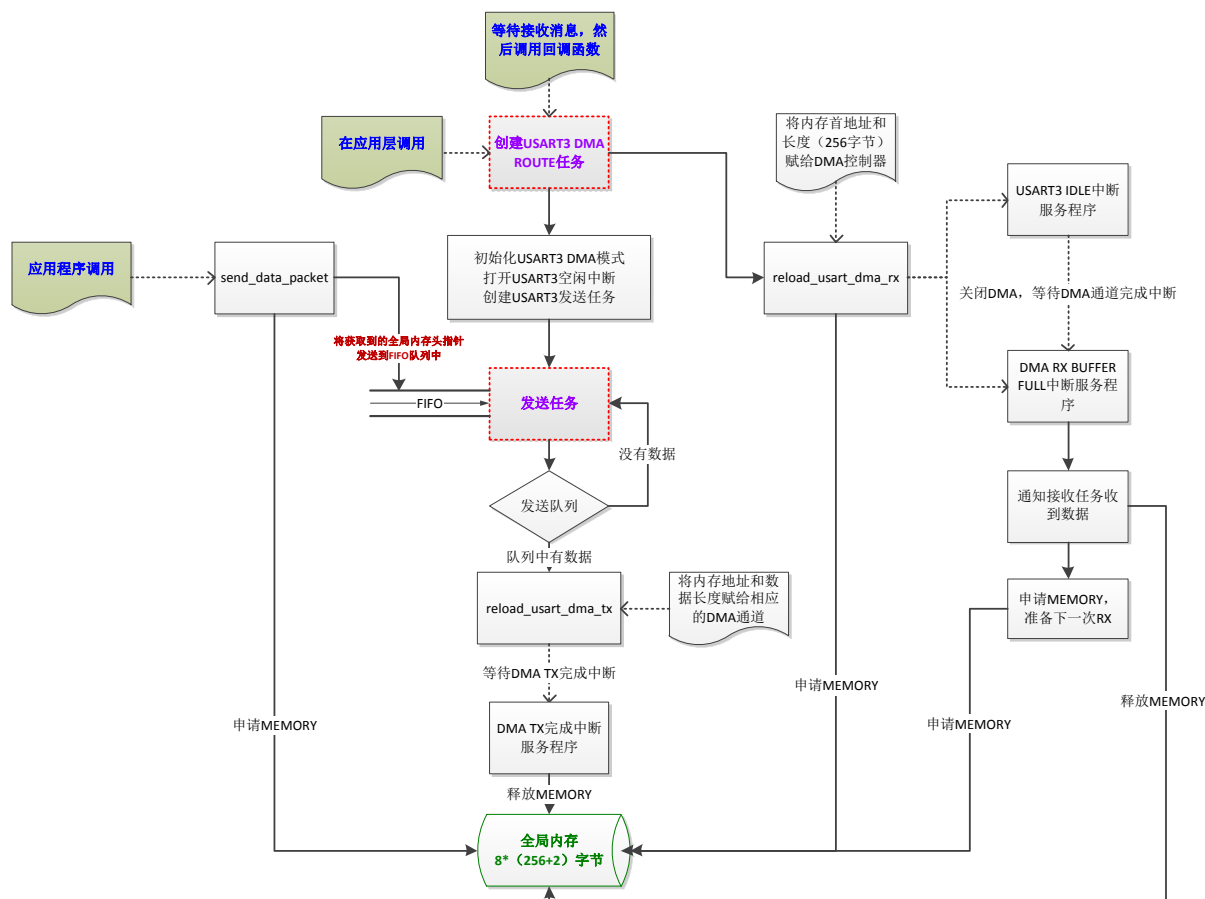


图 9-1 USART DMA 实现

10.1.2.1 初始化 init_usart

初始化和查询方式一样，只需要将 usart_mode 设置为 USART_MODE_DMA 即可。

函数用例：

```
init_usart(COM3, 115200, 8, PARITY_NONE, USART_MODE_DMA);
```

10.1.2.2 发送数据 send_data_packet

```
u8 send_data_packet(u8* p_cmd, u16 cmd_len);
```

参数

p_cmd 待发送数据头指针

cmd_len 待发送数据长度（不要大于 256 字节，此参数可以设置）

返回值

0: 数据成功加入 DMA 发送队列

1: 数据发送失败

函数用例

```
send_data_packet("Hello, world!\r\n", strlen("Hello, world!\r\n"));
```

10.1.2.3 接收数据 init_usart_recv_callback

接收数据采用回调函数的形式。即初始化时将回调函数指针传给 HAL, HAL 在接收到数据时自动调用回调函数, 用户只需要在回调函数中处理数据即可。

```
void init_usart_recv_callback(p_usart_dma_recv p_fun);
```

参数

p_fun 函数指针, 定义为 **p_usart_dma_recv**

```
typedef u8 (*p_usart_dma_recv)(u8* p_data, u16 len);
```

返回值

无

函数用例

```
init_usart_recv_callback(usart_recv_packet);
```

10.2 SPI

提供四个 API，初始化，去初始化，发送一个 byte，接收一个 byte

10.2.1 初始化 SPI

`void spi_init(void)`

参数：无

返回值：无

10.2.2 去初始化

`void spi_deinit(void)`

参数：无

返回值：无

10.2.3 发送一个字节

`u16 spi_send_byte(uint8_t byte)`

参数：uint8_t byte，发送的字节

返回值：u16，接收到的数据

10.2.4 接收一个字节

`u16 spi_read_byte(void)`

参数：无

返回：接收数据

10.3 SRAM

10.3.1 初始化

void sram_init(void)

参数：无

返回值：无

10.3.2 写操作（16 位）

void sram_write_buffer(uint16_t* pBuffer, uint32_t WriteAddr, uint32_t NumHalfwordToWrite)

参数： pBuffer, 16 位的 buffer 地址

WriteAddr, SRAM 偏移地址

NumHalfwordToWrite 写 buffer 长度

10.3.3 读操作（16 位）

void sram_read_buffer(uint16_t* pBuffer, uint32_t ReadAddr, uint32_t NumHalfwordToRead)

参数： pBuffer 16 位的 buffer 地址

ReadAddr SRAM 偏移地址

NumHalfwordToRead 读 buffer 长度

10.3.4 写操作（8 位）

void sram_write_byte(uint8_t* pBuffer, uint32_t WriteAddr, uint32_t NumHalfwordToWrite)

参数： pBuffer 8 位的 buffer 地址

WriteAddr SRAM 偏移地址

NumHalfwordToWrite 写 buffer 长度

10.3.5 读操作（8 位）

void sram_read_byte(uint8_t* pBuffer, uint32_t ReadAddr, uint32_t NumHalfwordToRead)

参数： pBuffer 8 位的 buffer 地址

ReadAddr SRAM 偏移地址

NumHalfwordToRead 读 buffer 长度

10.4 comX100 CANOpen 主站模块

10.4.1 初始化 comX100 task API

void init_comX100_module_task(control_pRPDO_function* p_control_callback, u8 call_num)

参数: p_control_callback, 接收到数据 callback 函数
call_num, RPDO 的数量

返回值: void

说明: 这个函数在系统启动时调用一次就行。接收到数据会调用 callback 函数, 注意: callback 函数运行在 comX100 task 环境下, 在内部处理时应简单快速, 否则可能会影响数据的接收。

10.4.2 发送 PDO API

u8 comX100_send_packet(void* p_data, u8 TPDO_offset, u8 TPDO_num)

参数: p_data, 要发送的数据 buffer 指针
TPDO_offset, TPDO 偏移地址, 这个参数需要和 comX100 模块配置的相对应。
TPDO_num, 一次发送的 PDO 数据包

返回值: u8, 0 成功, 1 失败

说明: 因为 comX100 模块的原因, 如果两次填写的数据完全一样, 那么, 数据不会被发送, 只有当两次发送的数据有差异时才会进行发送。

10.5 CAN 总线

10.5.1 初始化

void CAN_Config(USER_CAN_TypeDef CANx)

参数: CANx 选择 CAN 编号, USER_CAN_1 或者 USER_CAN_2

10.5.2 发送数据

void can_send_data(USER_CAN_TypeDef CANx, uint16_t ID, uint8_t* p_data, uint8_t len)

参数: CANx 选择 CAN 编号, USER_CAN_1 或者 USER_CAN_2

ID CAN ID

p_data 待发送的数据包

len 数据包长度

10.5.3 接收数据

void CAN1_RX0_IRQHandler(void)

通过中断来接收数据

10.6 RTC

RTC 模块包含 RTC 实时时钟和 4KB BKSRAM（掉电数据不丢失）。

10.6.1 初始化

u32 rtc_init(void)

参数： 无

10.6.2 清空 4KB SRAM

void clear_bkpsram(void)

参数： 无

10.6.3 写 4KB BKSRAM（8 位）

void write_byte_to_bkpsram(u32 add_offset, void* p_data, u32 len)

参数： add_offset BKSRAM 偏移地址

p_data buffer 指针

len 写长度

返回值： 无

10.6.4 读 4KB BKSRAM（8 位）

void read_byte_from_bkpsram(u32 add_offset, void* p_data, u32 len)

参数： add_offset BKSRAM 偏移地址

p_data 存数据 BUFFER 指针

len 读长度

返回值： 无

10.6.5 写 4KB BKSRAM（32 位）

void write_u32_to_bkpsram(u32 add_offset, void* p_data, u32 len)

参数： add_offset BKSRAM 偏移地址

p_data 存数据 BUFFER 指针

len 写长度

返回值： 无

10.6.6 读 4KB BKSRAM（32 位）

void read_u32_from_bkpsram(u32 add_offset, void* p_data, u32 len)

参数： add_offset BKSRAM 偏移地址

p_data 存数据 BUFFER 指针
len 读长度
返回值: 无

10.6.7 写备份寄存器

void write_backup_reg(u8 index, u32 data)
参数: index 写寄存器序号
data 写数据
返回值: 无

10.6.8 读备份寄存器

u32 read_backup_reg(u8 index)
参数: index 读寄存器序号
返回值: u32 寄存器数据

10.6.9 设置 RTC 时间

void set_rtc_time(u8 second, u8 minute, u8 hour)
参数: second 秒
minute 分钟
hour 小时
返回值: 无

10.6.10 读 RTC 时间

void get_rtc_time(u8* second, u8* minute, u8* hour)
参数: second 存秒地址
minute 存分钟地址
hour 存小时地址
返回值: 无

10.6.11 设置 RTC 日期

void set_rtc_date(u32 weekday, u32 date, u32 month, u32 year)
参数: weekday 周
date 日
month 月
year 年
返回值: 无

10.6.12读 RTC 日期

void get_rtc_date(u32* weekday, u32* date, u32* month, u32* year)

参数:	weekday	存周地址
	date	存日地址
	month	存月地址
	year	存年地址
返回值:	无	