
深入剖析 *printf* 函数

Copyright © 2013 Long Luo.

Revision History

版本历史

Version 版本	Editing Note 修订历史	Reviser/Date 修订人/日期	Confirm/Date 确认/日期	Approval/ Date 批准/日期	Implement Date 实施/日期
1.0	创建	罗龙	12/25/2013		

Table Of Contents

Revision History.....	2
Table Of Contents.....	3
一、引子.....	4
二、如何不借助第三方库实现 printf 函数？	5
1.1 内核的诱惑.....	5
1.2 用 printf 撕开一个小小的口子.....	5
1.3 Linux 系统调用.....	7
1.4 如何使用 Linux 系统调用？	7
1.5 用汇编语言实现 "Hello World".....	8
1.6 胜利的果实.....	11
三、形参列表和格式化输出是如何实现的？	12
3.1 格式化输出.....	12
3.2 形参列表的读入.....	12
3.3 参数格式化输出.....	15
3.4 应用层的使用.....	27
结束语.....	30

一、引子

---"你为什么要去登珠穆朗玛?"

当美国《纽约时报》记者问英国登山家乔治·马洛里。

--- "Because it is there(因为山在那里)。"

--题记

二、如何不借助第三方库实现 printf 函数？

知识是一种快乐，而好奇则是知识的萌芽。

---（英国）弗朗西斯·培根

1.1 内核的诱惑

会当凌绝顶，一览众山小。

内核，是一个操作系统的核心。它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

几十年来，内核以它那深深的魅力吸引着无数的码农为之倾倒，一代又一代的码农们从青青葱葱走向硕果累累，从风华正茂走向耄耋之年，也走出了现在多姿多彩的世界。

内核就像一位风姿卓约的美女，多少码农欲一亲芳泽而不得。Linux 内核是庞大复杂的，超过 600 万行的代码，就如同珠穆朗玛峰一样那样让人望而生畏。初学者一踏入，绝大多数会不自觉地迷失在这座庞大的迷宫里。

1.2 用 printf 撕开一个小小的口子...

作为一名内核小白，我也期望着那天能登上 Linux 内核这座高峰，一览其风采，但高原反应可不是闹着玩的。

既然暂时攀登不了珠穆朗玛峰，那就先试试攀登莲花山吧...

每一位初学者都学习过下面这个例子：

---（报告）没看过？

---拖出去，XX 了

```
1.  /*****
2.  ** File: - Z:\code\c\LLprintf\print0.1\LLapp.c
3.  **
4.  ** Copyright (C), Long.Luo, All Rights Reserved!
5.  **
6.  ** Description:
7.  **      LLapp.c
8.  **
9.  ** Version: 0.1
10. ** Date created: 21:30:00,10/01/2013
11. ** Author: Long.Luo
12. **
13. ** ----- Revision History: -----
14. ** <author> <data>          <desc>
15. **
16. *****/
17.
18. #include <stdio.h>
19.
20. int main(void)
```

```

21. {
22.     printf("Hello, World!\n");
23.
24.     return 0;
25. }

```

我们通过调用 `printf` 就可以实现在屏幕上输出一段字符？

为什么呢？

但是假如我们不用 `printf`，怎么做呢？

`printf` 里面蕴含着什么样的秘密呢？

.....

许许多多未知等待我们去深入分析和研究...

我们先看看 `LLapp.c` 文件经过预处理之后发生了什么？

```

root@long-desktop:/work/code/c/LLprintf/print0.1# cat LLapp.cpp | grep printf
extern int _IO_vfprintf (_IO_FILE *__restrict, const char *__restrict,
extern int fprintf (FILE *__restrict __stream,
extern int printf (__const char *__restrict __format, ...);
extern int sprintf (char *__restrict __s,
extern int vfprintf (FILE *__restrict __s, __const char *__restrict __format,
extern int vprintf (__const char *__restrict __format, __gnuc_va_list __arg);
extern int vsprintf (char *__restrict __s, __const char *__restrict __format,
extern int snprintf (char *__restrict __s, size_t __maxlen,
    __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3, 4)));
extern int vsnprintf (char *__restrict __s, size_t __maxlen,
    __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3, 0)));
extern int vdprintf (int __fd, __const char *__restrict __fmt,
    __attribute__ ((__format__ (__printf__, 2, 0)));
extern int dprintf (int __fd, __const char *__restrict __fmt, ...)
    __attribute__ ((__format__ (__printf__, 2, 3)));
printf("Hello, printf!\n");
root@long-desktop:/work/code/c/LLprintf/print0.1# █

```

图 1.1 PrintfCpp

可见经过预处理之后引入了很多其他函数，正是经过这一系列调用实现了我们想要的功能。

我们再来看看 `printf` 的定义：

```

1. int printf(const char *fmt, ...)
2. {
3.     int i;
4.     char buf[256];
5.
6.     va_list arg = (va_list)((char*)&fmt + 4);
7.     i = vsprintf(buf, fmt, arg);
8.     write(buf, i);
9.
10.    return i;
11. }

```

事实上，`printf` 是作为 C 语言的标准输入输出库里面的一个函数提供给我们

的，我们已经对它习以为常了。

这些 C 语言函数库是随 Linux 核心提供给我们的，这些库对系统调用进行了一些包装和扩展。

实际上，很多已经被我们习以为常的 C 语言标准函数，在 Linux 平台上的实现都是靠系统调用完成的，所以如果想对系统底层的原理作深入的了解，必须先掌握各种系统调用。

1.3 Linux 系统调用

Linux 内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。

从某种角度来看，系统调用和普通的函数调用非常相似。区别仅仅在于，系统调用由操作系统核心提供，运行于核心态；而普通的函数调用由函数库或用户自己提供，运行于用户态。二者在使用方式上也有相似之处，在下面将会提到。

1.4 如何使用 Linux 系统调用？

好了，了解到这些之后，来看下我们不使用 C 语言标准库完成的 app.c 1.0 版：

```
1.  /*****
2.  ** File: - Z:\code\c\LLprintf\print1.0\app.c
3.  **
4.  ** Copyright (C), Long.Luo, All Rights Reserved!
5.  **
6.  ** Description:
7.  **     app.c
8.  **
9.  ** Version: 1.0
10. ** Date created: 21:48:18,10/01/2013
11. ** Author: Long.Luo
12. **
13. ** ----- Revision History: -----
14. ** <author> <data>          <desc>
15. **
16. *****/
17.
18. void LLprint(char *msg, int len);
19.
20. int main(void)
21. {
22.     LLprint("Hello, LLprint!\n", 16);
23.
24.     return 0;
25. }
```

从代码可以看出，这一次我们没有包含任何头文件，只是调用了一个 `void LLprint(char *msg, int len)` 函数来实现在屏幕上输出的工作，那么 `void LLprint(char *msg, int len)` 的实现呢？

当然，按照我们的要求：

---我们必须实现一个纯正的 `printf`，不借助任何第三方库，完全调用 Linux 的系统调用，甚至直接自己写....

---NO!

完全实现 `printf` 的功能，目前来说还是一个 Mission Impossible，但是我们可以完全降低难度，**只是在屏幕上输出一段我们自己想要的字符**，这完全是可以办到的。

1.5 用汇编语言实现 "Hello World"

我们可以用纯汇编语言来实现它， Look:

```

1.  ;*****
2.  ;   File: - Z:\code\c\LLprintf\print1.0\LLprint.asm
3.  ;
4.  ;   Copyright (C), Long.Luo, All Rights Reserved!
5.  ;
6.  ;   Description:
7.  ;       LLprint.asm
8.  ;
9.  ;   Version: 1.0
10. ;   Date created: 21:45:04,10/01/2013
11. ;   Author: Long.Luo
12. ;
13. ; ----- Revision History: -----
14. ;   <author>   <data>           <desc>
15. ;
16. ;*****
17.
18. extern main
19.
20. [section .data] ; Data start here
21.
22. [section .text] ; Code start here
23.
24. global _start  ; import ENTRY: _start, for the LD.
25. global LLprint ; import the function for app.c.
26.
27. _start:
28.     call main      ; main
29.     add esp, 8      ;
30.

```



```

31.      mov ebx, 0          ; 参数一: 退出代码
32.      mov eax, 1          ; 系统调用号(sys_exit)
33.      int 0x80           ; 调用内核功能
34.
35. ; =====
36. ;                      void LLprint(char* msg, int len);
37. ; =====
38. LLprint:
39.      mov edx, [esp + 8]   ; 参数三: 字符串长度
40.      mov ecx, [esp + 4]   ; 参数二: 要显示的字符串
41.      mov ebx, 1          ; 参数一: 文件描述符(stdout)
42.      mov eax, 4          ; 系统调用号(sys_write)
43.      int 0x80           ; 调用内核功能
44.      ret                ; 退出程序
45.

```

代码中的注释已经写的挺详细的了，简单来说，

程序启动之后，首先会进入 **main** 函数执行，然后调用 **LLprint** 函数，在 **LLprint** 函数里面会调用 **sys_write** 的系统调用来实现在屏幕上输出字符的功能，最后返回，在 **main** 函数里面调用 **sys_exit** 来退出。

有了上面的 **app.c** 和 **LLprint.asm** 2 个主要文件就算是大功告成了，不过，万事俱备只欠东风，我们还需要将分别将它们编译链接起来，最终生成一个可执行文件才行。

不过呢，由于编译的命令比较长，我们写代码经常需要修改查错，每次都输入命令，未免太长了，不符合低碳环保的要求。

要记住，懒人才是推动这个世界进步的力量！

一个勤奋的程序员不是一个好的程序员。

我们的前辈们，正是看到了这个弊端，所以发明了 **Makefile** 来代替这种重复劳动，实现自动化编译链接工作。

最终完成的 **Makefile** 如下所示：

```

1.  *****
2.  # File: - Z:\code\c\LLprintf\print1.0\Makefile
3.  #
4.  # Copyright (C), Long.Luo, All Rights Reserved!
5.  #
6.  # Description:
7.  #      Makefile
8.  #
9.  # Version: 1.0
10. # Date created: 21:49:10,10/01/2013
11. # Author: Long.Luo
12. #
13. # ----- Revision History: -----
14. # <author> <data>          <desc>

```

```
15. #
16. #####
17.
18. # Programs, flags, etc.
19. ASM      = nasm
20. CC       = gcc
21. LD       = ld
22. ASMFLAGS = -f elf
23. CFLAGS   = -c
24. #CFLAGS  = -m32 -c
25. LDFLAGS   = -s
26. #LDFLAGS  =-m elf_i386 -s
27.
28. # This Program
29. BIN      = AppPrint
30. OBJS     = LLprint.o app.o
31.
32. # All Phony Targets
33. .PHONY : everything final image clean realclean disasm all building
34.
35. # Default starting position
36. everything : $(BIN)
37.
38. all : realclean everything
39.
40. final : all clean
41.
42. clean :
43.     rm -f $(OBJS)
44.
45. realclean :
46.     rm -f $(OBJS) $(BIN)
47.
48. $(BIN) : $(OBJS)
49.     $(LD) $(LDFLAGS) -o $(BIN) $(OBJS)
50.
51. LLprint.o : LLprint.asm
52.     $(ASM) $(ASMFLAGS) -o $@ $<
53.
54. app.o: app.c
55.     $(CC) $(CFLAGS) -o $@ $<
56.
```

这样我们每次都只需要在命令行输入 **make all** 就会自动化编译连接了。

1.6 胜利的果实

来看看我们最终的成果：

```
root@long-desktop:/work/code/c/LLprintf/print1.0# ll
total 32
drwxrwxrwx 2 nobody nogroup 4096 2013-01-10 21:51 ./
drwxrwxrwx 4 nobody nogroup 4096 2013-01-10 21:24 ../
-rwxr--r-- 1 nobody nogroup 633 2013-01-10 21:48 app.c*
-rw-r--r-- 1 root root 860 2013-01-10 21:51 app.o
-rwxr-xr-x 1 root root 496 2013-01-10 21:51 AppPrint*
-rwxr--r-- 1 nobody nogroup 1409 2013-01-10 21:47 LLprint.asm*
-rw-r--r-- 1 root root 640 2013-01-10 21:51 LLprint.o
-rwxr--r-- 1 nobody nogroup 1147 2013-01-10 21:51 Makefile*
root@long-desktop:/work/code/c/LLprintf/print1.0# ./AppPrint
Hello, LLprint!
root@long-desktop:/work/code/c/LLprintf/print1.0#
```

图 1.1 PrintfResult

我们只是完成了使用汇编语言来实现在屏幕上输出我们想要的字符，
But, printf 那么复杂的功能到底是如何实现的呢？

预知后事如何，且听下回分解！

三、形参列表和格式化输出是如何实现的？

在上一章里我们论述了“如何不借助第三方库在屏幕上输出“Hello World”？”。我们已经实现了用汇编语言在屏幕上输出了“Hello World”，迈出了万里长征的第一步，但是我们知道实际的 `printf` 的功能是十分强大的，它和 `scanf` 一样属于标准输入输出的一种 **格式化函数**，我们一般是这样使用它的：

`printf()`的基本形式： `printf("格式控制字符串", 变量列表);`

3.1 格式化输出

`printf()`函数是格式输出函数，请求 `printf()`打印变量的指令取决与变量的类型。

例如，在打印整数是使用 `%d` 符号，在打印字符是用 `%c` 符号。这些符号被称为转换说明。因为它们指定了如何不数据转换成可显示的形式。

下列列出的是 ANSI C 标准 `printf()`提供的各种转换说明。

转换说明及作为结果的打印输出

<code>%a</code>	浮点数、十六进制数字和 <code>p</code> -记数法（ <code>C 9 9</code> ）
<code>%A</code>	浮点数、十六进制数字和 <code>p</code> -记法（ <code>C 9 9</code> ）
<code>%c</code>	一个字符
<code>%d</code>	有符号十进制整数
<code>%e</code>	浮点数、 <code>e</code> -记数法
<code>%E</code>	浮点数、 <code>E</code> -记数法
<code>%f</code>	浮点数、十进制记数法
<code>%g</code>	根据数值不同自动选择 <code>%f</code> 或 <code>%e</code> 。
<code>%G</code>	根据数值不同自动选择 <code>%f</code> 或 <code>%e</code> 。
<code>%i</code>	有符号十进制数（与 <code>%d</code> 相同）
<code>%o</code>	无符号八进制整数
<code>%p</code>	指针
<code>%s</code>	字符串
<code>%u</code>	无符号十进制整数
<code>%x</code>	使用十六进制数字 <code>0 f</code> 的无符号十六进制整数
<code>%X</code>	使用十六进制数字 <code>0 f</code> 的无符号十六进制整数
<code>%%</code>	打印一个百分号

3.2 形参列表的读入

`printf` 函数的参数列表是如下的形式：

`int printf(const char *fmt, ...)`

类似于上面参数列表中的 `token: ...`，介个是可变形参的一种写法。当传递参数的个数不确定时，就可以用这种方式来表示。

但是电脑比程序员更笨，函数体必须知道具体调用时参数的个数才能保证顺利执行，那么我们必须寻找一种方法来了解参数的个数。

让我们先回到代码中来:

```
1.  /*****
2.  ** File: - Z:\code\c\LLprintf\print2.1\LLprintf.c
3.  **
4.  ** Copyright (C), Long.Luo, All Rights Reserved!
5.  **
6.  ** Description:
7.  **     LLprintf.c
8.  **
9.  ** Version: 2.0
10. ** Date created: 23:56:33,24/01/2013
11. ** Author: Long.Luo
12. **
13. ** ----- Revision History: -----
14. ** <author> <data>          <desc>
15. **
16. *****/
17.
18.
19. #include "LLprintf.h"
20.
21.
22. // Lprintf
23. int Lprintf(const char *fmt, ...)
24. {
25.     int i;
26.     char buf[256];
27.
28.     va_list arg = (va_list)((char*)&fmt + 4); /*4 是参数 fmt 所占堆栈中的大小*/
29.     i = vsLprintf(buf, fmt, arg);
30.     buf[i] = 0;
31.     LLprint(buf, i);
32.
33.     return i;
34. }
```

如上面代码中的:

```
va_list arg = (va_list)((char*)&fmt + 4);
```

而 va_list 的定义:

```
typedef char *va_list
```

这说明它是一个字符指针。

其中的: (char*)&fmt + 4 表示的是...中的第一个参数。

大家肯定很迷惑，不急，再详细解释：

C 语言中，参数压栈的方向是从右往左。也就是说，当调用 printf 函数的适合，先是最右边的参数入栈。

fmt 是一个指针，这个指针指向第一个 const 参数 (const char *fmt)中的第一个元素。
fmt 也是个变量，它的位置，是在栈上分配的，它也有地址。

对于一个 char *类型的变量，它入栈的是指针，而不是这个 char *型变量。

换句话说：

你 sizeof(p) (p 是一个指针，假设 p=&i,i 为任何类型的变量都可以)

得到的都是一个固定的值。(我的计算机中都是得到的 4)

当然，我还要补充的一点是：栈是从高地址向低地址方向增长的。

现在我想你该明白了：为什么说(char*)&fmt + 4 表示的是...中的第一个参数的地址。

为毛我还是不明白啊？？？

不急，我给你更直观的解释：

```

1.  /*****
   *
2.          可变参数函数调用原理（其中涉及的数字皆为举例）
3.  =====
   =
4.
5.  i = 0x23;
6.  j = 0x78;
7.  char fmt[] = "%xd";
8.  printf(fmt, i, j);
9.
10.      push    j
11.      push    i
12.      push    fmt
13.      call    printf
14.      add     esp, 3 * 4
15.
16.
17.      |      HIGH      |      |      HIGH      |
18.      |      ...      |      |      ...      |
19.      |-----|      |-----|
20.      |      |      |      0x32010 |      '\0'      |
21.      |-----|      |-----|
22.      0x3046C |      0x78      |      0x3200c |      d      |
23.      |-----|      |-----|

```

```

24.   arg = 0x30468 |      0x23 |      0x32008 |      %      |
25.           |-----|           |-----|
26.   0x30464 |      0x32000 |      0x32004 |      x      |
27.           |-----|           |-----|
28.           |           |      0x32000 |      %      |
29.           |-----|           |-----|
30.           |      ...      |           |      ...      |
31.           |      LOW      |           |      LOW      |
32.
33. 实际上, 调用 vsprintf 的情形是这样的:
34.
35.      vsLprintf(buf, 0x32000, 0x30468);
36.
37.  ****
    /

```

下面我们来看看下一句:

```
i = vsLprintf(buf, fmt, arg);
```

这句起什么作用呢?

让我们进入下一节: 对参数进行格式化处理。

3.3 参数格式化输出

让我们来看看 vsLprintf(buf, fmt, arg)是什么函数:

```

1.  int vsLprintf(char *buf, const char *fmt, va_list args)
2.  {
3.      char *p;
4.      int m;
5.      char inner_buf[STR_DEFAULT_LEN];
6.      char cs;
7.      int align_nr;
8.
9.      va_list p_next_arg = args;
10.
11.     for (p=buf; *fmt; fmt++) {
12.         if (*fmt != '%') {
13.             *p++ = *fmt;
14.             continue;
15.         }
16.         else { /* a format string begins */
17.             align_nr = 0;
18.         }
19.
20.         fmt++;
21.

```

```
22.     if (*fmt == '%') {
23.         *p++ = *fmt;
24.         continue;
25.     }
26.     else if (*fmt == '0') {
27.         cs = '0';
28.         fmt++;
29.     }
30.     else {
31.         cs = ' ';
32.     }
33.     while (((unsigned char)(*fmt) >= '0') && ((unsigned char)(*fmt) <= '9')) {
34.         align_nr *= 10;
35.         align_nr += *fmt - '0';
36.         fmt++;
37.     }
38.
39.     char * q = inner_buf;
40.     memset(q, 0, sizeof(inner_buf));
41.
42.     switch (*fmt) {
43.     case 'c':
44.         *q++ = *((char*)p_next_arg);
45.         p_next_arg += 4;
46.         break;
47.     case 'x':
48.         m = *((int*)p_next_arg);
49.         i2a(m, 16, &q);
50.         p_next_arg += 4;
51.         break;
52.     case 'd':
53.         m = *((int*)p_next_arg);
54.         if (m < 0) {
55.             m = m * (-1);
56.             *q++ = '-';
57.         }
58.         i2a(m, 10, &q);
59.         p_next_arg += 4;
60.         break;
61.     case 's':
62.         strcpy(q, *((char**)p_next_arg));
63.         q += strlen(*((char**)p_next_arg));
64.         p_next_arg += 4;
65.         break;
```



```
66.         default:
67.             break;
68.     }
69.
70.     int k;
71.     for (k = 0; k < ((align_nr > strlen(inner_buf)) ? (align_nr - strlen(inner_buf)) :
    0); k++) {
72.         *p++ = cs;
73.     }
74.     q = inner_buf;
75.     while (*q) {
76.         *p++ = *q++;
77.     }
78. }
79.
80. *p = 0;
81.
82. return (p - buf);
83. }
```

这个函数起什么作用呢？

我们回想下 printf 起什么作用呢？

哦，printf 接受一个格式化的命令，并把指定的匹配的参数格式化输出。

好的，我们再看看 `i = vsLprintf(buf, fmt, arg);`

vsLprintf 返回的是一个长度值，那这个值是什么呢？

会不会是打印出来的字符串的长度呢？

没错，返回的就是要打印出来的字符串的长度。

其实看看 printf 中后面的一句：`LLprint(buf, i);`

介个是干啥的？

什么，你不知道，那赶紧看上一篇文章。

总结：vsLprintf 的作用就是格式化。

它接受确定输出格式的格式字符串 `fmt`。用格式字符串对个数变化的参数进行格式化，产生格式化输出。

我们也可以看看一个串口的 printf 的实现：

```
1.  //*****
2.  //
3.  //! A simple UART based printf function supporting %c, %d, %p, %s, %u,
4.  //! %x, and %X.
5.  //!
6.  //! \param pcString is the format string.
```

```
7.  //!
```

```
51.    //
52.    // Start the varargs processing.
53.    //
54.    va_start(vaArgP, pcString);
55.
56.    //
57.    // Loop while there are more characters in the string.
58.    //
59.    while(*pcString)
60.    {
61.        //
62.        // Find the first non-% character, or the end of the string.
63.        //
64.        for(ulIdx = 0; (pcString[ulIdx] != '%') && (pcString[ulIdx] != '\0');
65.            ulIdx++)
66.        {
67.        }
68.
69.        //
70.        // Write this portion of the string.
71.        //
72.        UARTwrite(pcString, ulIdx);
73.
74.        //
75.        // Skip the portion of the string that was written.
76.        //
77.        pcString += ulIdx;
78.
79.        //
80.        // See if the next character is a %.
81.        //
82.        if(*pcString == '%')
83.        {
84.            //
85.            // Skip the %.
86.            //
87.            pcString++;
88.
89.            //
90.            // Set the digit count to zero, and the fill character to space
91.            // (i.e. to the defaults).
92.            //
93.            ulCount = 0;
94.            cFill = ' ';
```

```
95.
96.      //
97.      // It may be necessary to get back here to process more characters.
98.      // Goto's aren't pretty, but effective.  I feel extremely dirty for
99.      // using not one but two of the beasts.
100.     //
101. again:
102.
103.     //
104.     // Determine how to handle the next character.
105.     //
106.     switch(*pcString++)
107.     {
108.         //
109.         // Handle the digit characters.
110.         //
111.         case '0':
112.         case '1':
113.         case '2':
114.         case '3':
115.         case '4':
116.         case '5':
117.         case '6':
118.         case '7':
119.         case '8':
120.         case '9':
121.         {
122.             //
123.             // If this is a zero, and it is the first digit, then the
124.             // fill character is a zero instead of a space.
125.             //
126.             if((pcString[-1] == '0') && (ulCount == 0))
127.             {
128.                 cFill = '0';
129.             }
130.
131.             //
132.             // Update the digit count.
133.             //
134.             ulCount *= 10;
135.             ulCount += pcString[-1] - '0';
136.
137.             //
138.             // Get the next character.
```

```
139.          //
140.          goto again;
141.      }
142.
143.      //
144.      // Handle the %c command.
145.      //
146.      case 'c':
147.      {
148.          //
149.          // Get the value from the varargs.
150.          //
151.          ulValue = va_arg(vaArgP, unsigned long);
152.
153.          //
154.          // Print out the character.
155.          //
156.          UARTwrite((char *)&ulValue, 1);
157.
158.          //
159.          // This command has been handled.
160.          //
161.          break;
162.      }
163.
164.      //
165.      // Handle the %d command.
166.      //
167.      case 'd':
168.      {
169.          //
170.          // Get the value from the varargs.
171.          //
172.          ulValue = va_arg(vaArgP, unsigned long);
173.
174.          //
175.          // Reset the buffer position.
176.          //
177.          ulPos = 0;
178.
179.          //
180.          // If the value is negative, make it positive and indicate
181.          // that a minus sign is needed.
182.          //
```

```
183.         if((long)ulValue < 0)
184.         {
185.             //
186.             // Make the value positive.
187.             //
188.             ulValue = -(long)ulValue;
189.
190.             //
191.             // Indicate that the value is negative.
192.             //
193.             ulNeg = 1;
194.         }
195.         else
196.         {
197.             //
198.             // Indicate that the value is positive so that a minus
199.             // sign isn't inserted.
200.             //
201.             ulNeg = 0;
202.         }
203.
204.         //
205.         // Set the base to 10.
206.         //
207.         ulBase = 10;
208.
209.         //
210.         // Convert the value to ASCII.
211.         //
212.         goto convert;
213.     }
214.
215.     //
216.     // Handle the %s command.
217.     //
218.     case 's':
219.     {
220.         //
221.         // Get the string pointer from the varargs.
222.         //
223.         pcStr = va_arg(vaArgP, char *);
224.
225.         //
226.         // Determine the length of the string.
```

```
227.         //
228.         for(u1Idx = 0; pcStr[u1Idx] != '\0'; u1Idx++)
229.         {
230.         }
231.
232.         //
233.         // Write the string.
234.         //
235.         UARTwrite(pcStr, u1Idx);
236.
237.         //
238.         // Write any required padding spaces
239.         //
240.         if(u1Count > u1Idx)
241.         {
242.             u1Count -= u1Idx;
243.             while(u1Count--)
244.             {
245.                 UARTwrite(" ", 1);
246.             }
247.         }
248.         //
249.         // This command has been handled.
250.         //
251.         break;
252.     }
253.
254.     //
255.     // Handle the %u command.
256.     //
257.     case 'u':
258.     {
259.         //
260.         // Get the value from the varargs.
261.         //
262.         u1Value = va_arg(vaArgP, unsigned long);
263.
264.         //
265.         // Reset the buffer position.
266.         //
267.         u1Pos = 0;
268.
269.         //
270.         // Set the base to 10.
```

```
271.          //
272.          ulBase = 10;
273.
274.          //
275.          // Indicate that the value is positive so that a minus sign
276.          // isn't inserted.
277.          //
278.          ulNeg = 0;
279.
280.          //
281.          // Convert the value to ASCII.
282.          //
283.          goto convert;
284.      }
285.
286.      //
287.      // Handle the %x and %X commands. Note that they are treated
288.      // identically; i.e. %X will use lower case letters for a-f
289.      // instead of the upper case letters it should use. We also
290.      // alias %p to %x.
291.      //
292.      case 'x':
293.      case 'X':
294.      case 'p':
295.      {
296.          //
297.          // Get the value from the varargs.
298.          //
299.          ulValue = va_arg(vaArgP, unsigned long);
300.
301.          //
302.          // Reset the buffer position.
303.          //
304.          ulPos = 0;
305.
306.          //
307.          // Set the base to 16.
308.          //
309.          ulBase = 16;
310.
311.          //
312.          // Indicate that the value is positive so that a minus sign
313.          // isn't inserted.
314.          //
```



```
315.         ulNeg = 0;
316.
317.         //
318.         // Determine the number of digits in the string version of
319.         // the value.
320.         //
321. convert:
322.         for(ulIdx = 1;
323.             (((ulIdx * ulBase) <= ulValue) &&
324.              (((ulIdx * ulBase) / ulBase) == ulIdx));
325.             ulIdx *= ulBase, ulCount--)
326.         {
327.         }
328.
329.         //
330.         // If the value is negative, reduce the count of padding
331.         // characters needed.
332.         //
333.         if(ulNeg)
334.         {
335.             ulCount--;
336.         }
337.
338.         //
339.         // If the value is negative and the value is padded with
340.         // zeros, then place the minus sign before the padding.
341.         //
342.         if(ulNeg && (cFill == '0'))
343.         {
344.             //
345.             // Place the minus sign in the output buffer.
346.             //
347.             pcBuf[ulPos++] = '-';
348.
349.             //
350.             // The minus sign has been placed, so turn off the
351.             // negative flag.
352.             //
353.             ulNeg = 0;
354.         }
355.
356.         //
357.         // Provide additional padding at the beginning of the
358.         // string conversion if needed.
```

```
359.          //
360.          if((ulCount > 1) && (ulCount < 16))
361.          {
362.              for(ulCount--; ulCount; ulCount--)
363.              {
364.                  pcBuf[ulPos++] = cFill;
365.              }
366.          }
367.
368.          //
369.          // If the value is negative, then place the minus sign
370.          // before the number.
371.          //
372.          if(ulNeg)
373.          {
374.              //
375.              // Place the minus sign in the output buffer.
376.              //
377.              pcBuf[ulPos++] = '-';
378.          }
379.
380.          //
381.          // Convert the value into a string.
382.          //
383.          for(; ulIdx; ulIdx /= ulBase)
384.          {
385.              pcBuf[ulPos++] = g_pchHex[(ulValue / ulIdx) % ulBase];
386.          }
387.
388.          //
389.          // Write the string.
390.          //
391.          UARTwrite(pcBuf, ulPos);
392.
393.          //
394.          // This command has been handled.
395.          //
396.          break;
397.      }
398.
399.      //
400.      // Handle the %% command.
401.      //
402.      case '%':
```

```
403.         {
404.             //
405.             // Simply write a single %.
406.             //
407.             UARTwrite(pcString - 1, 1);
408.
409.             //
410.             // This command has been handled.
411.             //
412.             break;
413.         }
414.
415.         //
416.         // Handle all other commands.
417.         //
418.         default:
419.         {
420.             //
421.             // Indicate an error.
422.             //
423.             UARTwrite("ERROR", 5);
424.
425.             //
426.             // This command has been handled.
427.             //
428.             break;
429.         }
430.     }
431. }
432. }
433.
434. //
435. // End the varargs processing.
436. //
437. va_end(vaArgP);
438. }
```

写的很精彩，是不是？

3.4 应用层的使用

通过上面的工作，我们已经实现了一个自己的 printf 函数：LLprintf。LLprintf 的功能和我们标准库的 printf 一样强大，我们可以在上层如此使用 LLprintf：

```
1.  /*****
2.  ** File: - Z:\code\c\LLprintf\print2.1\app.c
3.  **
4.  ** Copyright (C), Long.Luo, All Rights Reserved!
5.  **
6.  ** Description:
7.  **     app.c --- The Application Level.
8.  **
9.  ** Version: 2.1
10. ** Date created: 23:53:41,24/01/2013
11. ** Author: Long.Luo
12. **
13. ** ----- Revision History: -----
14. ** <author> <data>          <desc>
15. **
16. *****/
17.
18. #include "LLprintf.h"
19.
20. int main(void)
21. {
22.     char *welcome = "    A Tiny Demo show the LLprintf    ";
23.     char *program_name = "LLprintf";
24.     char *program_author = "Long.Luo";
25.     char *date = "Jan. 24th, 2013";
26.
27.     float program_version = 2.1;
28.
29.     Lprintf("%s\n\n", welcome);
30.     Lprintf("\t\t%s, version %f \n\n", program_name, program_version);
31.     Lprintf("\tCreated by %s, %s.\n\n", program_author, date);
32.
33.     return 0;
34. }
```

make 一下，我们再来看看输出结果：

```
root@long-desktop:/work/code/c/LLprintf/print2.1# ll
total 64
drwxrwxrwx 2 nobody nogroup 4096 2013-01-25 00:07 ./
drwxrwxrwx 6 nobody nogroup 4096 2013-01-24 22:01 ../
-rwxrwxrwx 1 nobody nogroup 953 2013-01-25 00:07 app.c*
-rw-r--r-- 1 root root 1128 2013-01-25 00:07 app.o
-rwxr-xr-x 1 root root 1724 2013-01-25 00:07 AppPrintf*
-rwxrwxrwx 1 nobody nogroup 1409 2013-01-24 23:52 LLprint.asm*
-rwxrwxrwx 1 nobody nogroup 2773 2013-01-25 00:21 LLprintf.c*
-rwxrwxrwx 1 nobody nogroup 740 2013-01-24 23:59 LLprintf.h*
-rw-r--r-- 1 root root 868 2013-01-25 00:07 LLprintf.o
-rw-r--r-- 1 root root 640 2013-01-25 00:07 LLprint.o
-rwxrwxrwx 1 nobody nogroup 1333 2013-01-24 23:55 Makefile*
-rwxrwxrwx 1 nobody nogroup 3745 2013-01-24 23:58 string.asm*
-rwxrwxrwx 1 nobody nogroup 1013 2013-01-24 23:57 string.h*
-rw-r--r-- 1 root root 800 2013-01-25 00:07 string.o
-rwxrwxrwx 1 nobody nogroup 3022 2013-01-24 23:59 vsLLprintf.c*
-rw-r--r-- 1 root root 1700 2013-01-25 00:07 vsLLprintf.o
root@long-desktop:/work/code/c/LLprintf/print2.1# ./AppPrintf
    A Tiny Demo show the LLprintf

        LLprintf, version

        Created by Long.Luo, Jan. 24th, 2013.
root@long-desktop:/work/code/c/LLprintf/print2.1#
```

至此这样我们就了解了 **printf** 函数的前因后果，聪明的你，弄明白了吗？

^_^

结束语

这篇文档最初去年发表在我的个人博客上：

1. 深入剖析 printf 函数(上)：如何不借助第三方库在屏幕上输出 "Hello World"?

<http://blog.csdn.net/tcpipstack/article/details/8490811>

第一篇主要讲述 printf 函数的具体调用过程、系统调用及如何使用汇编语言实现一个简单的 printf 函数；

2. 深入剖析 printf 函数(下)：---形参列表和格式化输出是如何做到的？

<http://blog.csdn.net/tcpipstack/article/details/8279584>

第二篇讲述的是 printf 的详细形参列表和格式化输出是如何做到的，并列举了一个串口的 printf 函数的详细实现来进行说明。

最近几天看见了，然后整理了出来，希望对大家有所帮助，如果发现有任何疑问或者错误之处，请及时与我联系，我的 E-mail: uniquelulong@gmail.com

Have Fun:-)

Long Luo Version 2.0 @PM12:35~13:00 December 25th, 2013 at Shenzhen, China.