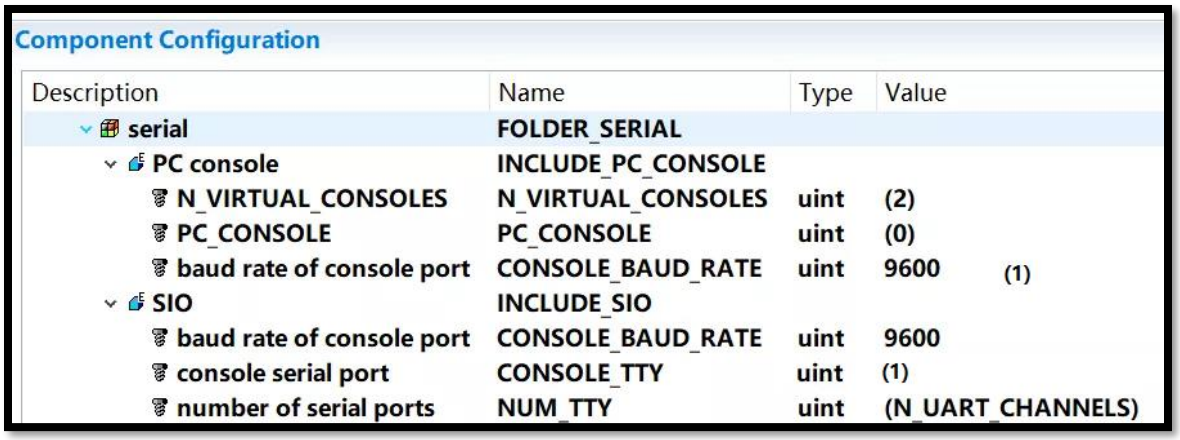
VxWorks 下的 kprintf 调用

调试程序时，最常用的一个手段是打印一些调试语句，而最常用的打印函数应该就是 printf()了。printf()的作用是向标准输出设备输出格式化的调试语句。这个标准输出设备默认是 PC Console 或串口。



也可以修改，例如改为文件或管道

```
02: #include <vxWorks.h>
03: #include <pipeDrv.h>
04: #include <stdio.h>
05: #include <ioLib.h>
06: #include <memPartLib.h>
07:
08: void test(char *data)
09:     {
10:     int fd;
11:     char *buf;
12:
13:     /* 建一个管道 */
14:     pipeDevCreate("/myPipe1", 10, 20);
15:
16:     /* 打开它 */
17:     fd = open("/myPipe1", 2, 0);
18:
19:     /* 将当前任务的输出设备指向管道 */
20:     ioTaskStdSet(0, 1, fd);
21:
22:     /* 输出字符 */
23:     printf(data);
24:
25:     /* 读取管道数据 */
26:     buf=calloc(1, 20);
27:     read(fd, buf, 20);
28:
29:     /* 恢复输出设备 */
30:     ioTaskStdSet(0, 1, 1);
31:
32:     /* 查看读取的数据 */
33:     printf("read from pipe: %s\n", buf);
34:     free(buf);
35:     close(fd);
36:     }
```

```
->
-> test "hello"
read from pipe: hello
value = 0 = 0x0
->
-> sp test,"world"
Task spawned: id = 0x88a7860, name = t1
value = 143292512 = 0x88a7860 = '''
-> read from pipe: world
```

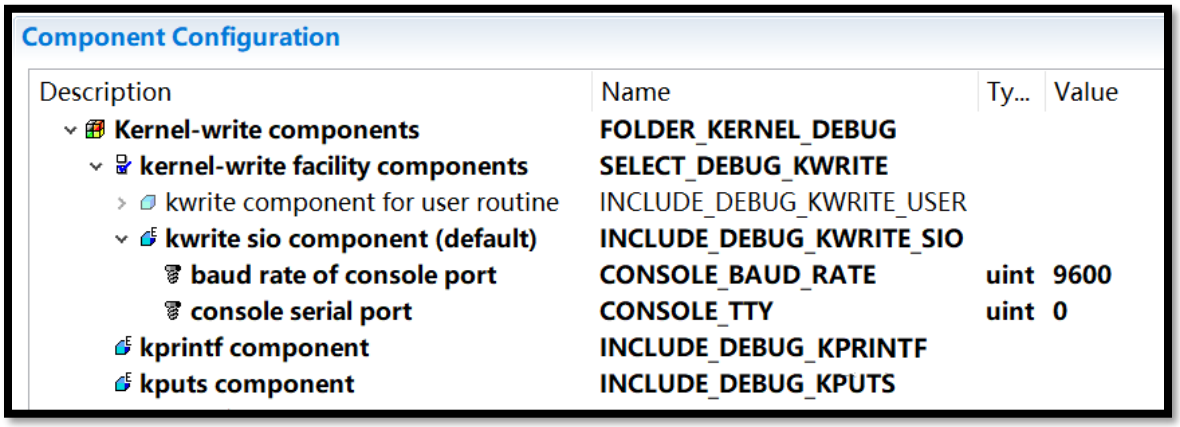
不过 printf()这种 IO 操作默认是**阻塞**模式的，**因此不能在 ISR 中使用**。那调试中断的时候，可以使用 logMsg()来打印调试语句

```
int printf
    (
    const char *fmt,
    ...
    );
int logMsg
    (
    char *fmt,
    int   arg1,
    int   arg2,
    int   arg3,
    int   arg4,
    int   arg5,
    int   arg6
    );
```

logMsg()在 ISR 中执行时，通过底层的
msgQSend(logMsgQId, msg, sizeof(msg), NO_WAIT,MSG_PRI_NORMAL)
将调试语句发送给优先级为 0 的任务 tLogTask
不过要想使用 logMsg()，需要在 usrRoot()调用 logInit()之后；使用 printf()，需要在 usrRoot()调用 usrSerialInit()或 usrPcConsoleInit()之后。

```
void usrRoot()
    {
    ...
    void usrIosCoreInit()
        {
        ...
        usrSerialInit ();
        usrPcConsoleInit ();
        ...
        }
    ...
    void usrIosExtraInit()
        {
        ...
        logInit ();
        ...
        }
    ...
    }
```

那在这些初始化之前如何打印调试语句呢？
vx6 新加了一个组件(好像是从 vx67 开始的)：Kernel Write



这个组件只是给开发人员提供了两个函数 kputs()和 kprintf()。这两个函数的声明分别类似于 ANSI 定义的 puts 和 printf()，其作用也差不多，它俩就是打印语句，用来调试的。

```
typedef int ssize_t;
ssize_t kputs
    (
    char *buffer
    );
int kprintf
    (
    const char *fmt,
    ...
    );
```

那么，区别呢？kputs/kprintf()默认只能输出到串口，而且使用的是串口的轮询模式。这样做的好处是：在中断使能和 IO 系统初始化之前的内核启动阶段就可以使用它了！

```
0272: void usrInit (int startType)
0273:     {
0274:     sysStart (startType);                    /* clear BSS
0275:     usrBootHwInit ();                        /* call usrBo
0276:     cacheLibInit (USER_I_CACHE_MODE, USER_D_CACHE_MOI
0277:     excShowInit ();                          /* exception
0278:     excVecInit ();                           /* exception
0279:     vxCpuLibInit ();                         /* Enable the
0280:     sysHwInit ();                            /* call the I
0281: kprintf("kprintf in %s %d\n",__FUNCTION__,__LINE__);
0282:     usrDebugKWriteInit ();                   /* Kernel pri
0283: kprintf("kprintf in %s %d\n",__FUNCTION__,__LINE__);
0284:     usrCacheEnable ();                       /* optionally

kprintf in usrInit 283
```

可以看到，在 usrInit()阶段的 sysHwInit()之后、usrCacheEnable()之前，就可以使用 kprintf()了，这可比 printf()/logMsg()的可用时机提前了很多。
另外，logMsg()在调用 msgQSend()时用的 NO_WAIT 方式，因此有可能会丢弃部分消息

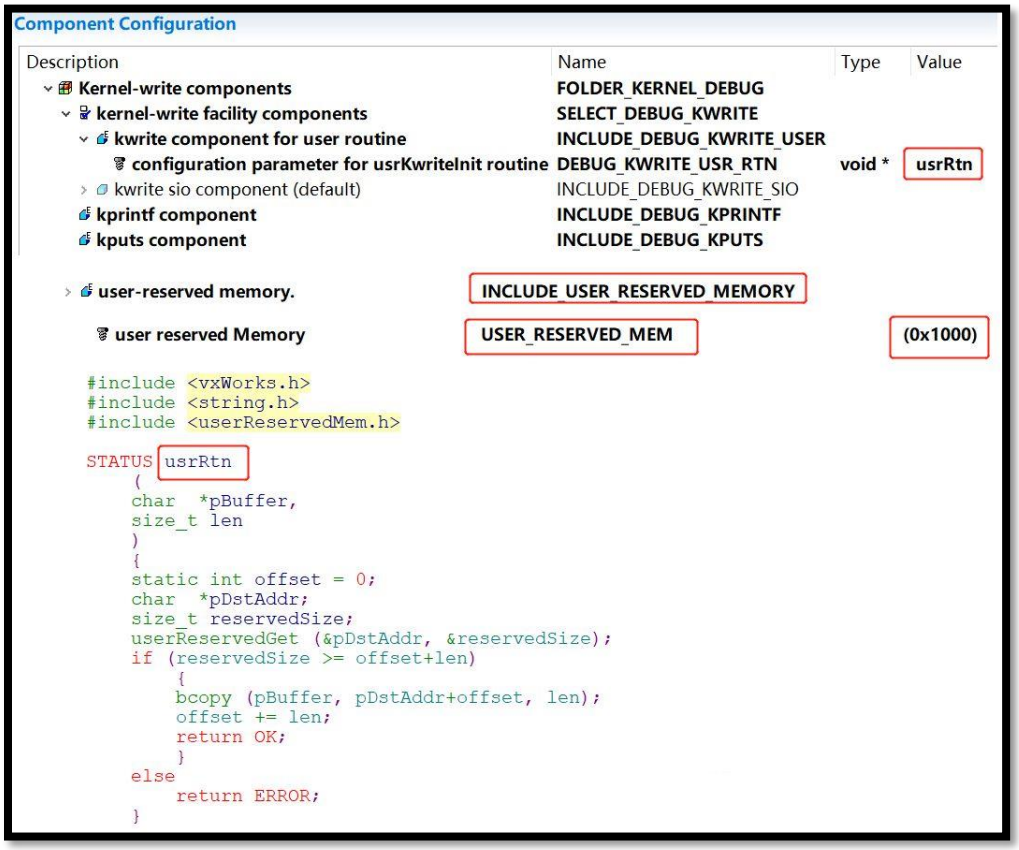
```
476: void usrIosExtraInit (void)
477:     {
478:     jobTaskLibInit (JOB_TASK_STACK_SIZE); /* task level wor
479:     excInit (MAX_ISR_JOBS);                  /* interrupt-level
480:     erfLibInit (ERF_MAX_USR_CATEGORIES, ERF_MAX_USR_TYPES);
481: logMsg("logMsg in %s %d\n",__FUNCTION__,__LINE__,0,0,0,0);
482:     logInit (consoleFd, MAX_LOG_MSGS, LOG_MSG_UNBREAKABLE_T
483: logMsg("logMsg in %s %d\n",__FUNCTION__,__LINE__,0,0,0,0);
484:     pipeDrv (PIPE_MAX_OPEN_FDS);             /* pipes */

0x45726e0 (tRootTask): logMsg in usrIosExtraInit 483
logTask: 5 log messages lost.
```

kput()/kprintf()就不存在这个问题了，它们既不会阻塞，也不会丢弃消息
如果 Target 的串口不能用，这个 Kernel Write 组件还可以通过自定义函数 DEBUG_KWRITE_USR_RTN 来输出调试语句。它可以将语句直接输出到 RAM 区域，或者 Flash 等 NVRAM 里。

Component Configuration

Description	Name	Type	Value
Kernel-write components	FOLDER_KERNEL_DEBUG		
kernel-write facility components	SELECT_DEBUG_KWRITE		
kwrite component for user routine	INCLUDE_DEBUG_KWRITE_USER		
configuration parameter for usrKwriteInit routine	DEBUG_KWRITE_USR_RTN	void *	NULL
kwrite sio component (default)	INCLUDE_DEBUG_KWRITE_SIO		
kprintf component	INCLUDE_DEBUG_KPRINTF		
kputs component	INCLUDE_DEBUG_KPUTS		

来看一个输出到 RAM 保留区的例子



再次调用 kprintf()后，调试语句就是记录到这个 User Reserved Memory 里了

```
-> kprintf "hello "
value = 6 = 0x6
-> kprintf "world\n"
value = 6 = 0x6
-> d sysMemTop(),12,1
NOTE: memory values are displayed in hexadecimal.
0x3ffff000:  68 65 6c 6c 6f 20 77 6f 72 6c 64 0a  *hello world.....*
value = 0 = 0x0
->
```


三者对比：

printf()

- 最常用
- 阻塞模式，不丢消息
- 不能用于中断
- 在 usrRoot()初始化 IO 系统后才能调用

logMsg()

- 可用于中断
- 可能丢消息
- 在初始化 IO 系统和 logging 机制后才能调用

kprintf()

- 在 usrInit()调用 sysHwInit()之后即可调用
- 可用于中断
- 可输出到串口或 Memory 设备
- 不丢消息