文章编号: 1002-0640-(2012)增刊-0107-05

VxWorks下MPC8280网络驱动的开发

周 文, 张日飞, 蒋秀波, 蔡志鹏
(北方自动控制技术研究所, 太原 030006)

摘 要: 介绍了一种在Wind River Workbench 3.0的开发环境下开发MPC8280芯片网络驱动的方法。首先简要介绍了嵌入式操作系统VxWorks的网络体系的架构, 尔后介绍了END开发的流程, 重点落在了对END开发的核心内容和方法的介绍上。按此方法所开发的END经验证可以驱动相应的硬件使其正常工作, 本开发方法较为简便, 具有实用价值。

关键词: 嵌入式系统, 网络栈, 风河, 增强型网络驱动
中图分类号: TP393 **文献标识码**: A

Exploitation of MPC8280 Network Device Driver Based on VxWorks

ZHOU Wen, ZHANG Ri-fei, JIANG Xiu-bo, CAI Zhi-peng
(*North Automatic Control Technology Institute, Taiyuan* 030006, *China*)

Abstract: A method in which the END is designed in Wind River Workbench 3.0 is introduced in this paper. At first, the structure of the embedded OS VxWorks's network system is introduced briefly; then the process of END development is discussed; finally, the key point of this paper is the content and the way of END development. The END introduced here has been tested to be able to drive hardware successfully, and the method introduced here is simple and valuable.

Key words: embedded system, network stack, wind river, END

引 言

随着近年来嵌入式技术在国防和民用各个领域得到了越来越广泛的应用, 各种嵌入式操作系统也被越来越多的被开发人员所熟悉。目前国内比较常用的几种嵌入式操作系统有: Linux、μC-OS、winCE、VxWorks等等。其中由美国Wind River System (风河) 公司开发的VxWorks以其卓越的实时性和良好的可靠性而著称于世, 同时它也兼具其他嵌入式操作系统普遍具有的一些优点: 如体积小巧, 具有可裁剪性等等, 正因为如此, VxWorks被广泛地用于很多高新技术领域, 典型范例包括: F-16战斗机、B-2隐形轰炸机和NASA的火星探测器上。为便于广大开发人员更好地利用VxWorks开发出高效的应用系统, 风河公司先后推出了两种集成开发环境: Tornado和Workbench, 目前前者已基本被后者取代。本文所要介绍的正是在Workbench环境下增强型网络驱动 (Enhanced Network Device, 简称为END) 的开发方法和技巧。

收稿日期: 2011-09-19 **修回日期**: 2011-12-08
作者简介: 周 文 (1986-), 男, 安徽宿州人, 硕士研究生, 研究方向: 系统工程。

1 VxWorks的网络体系结构

用在计算机网络中被大家所熟知的OSI模型来说明, 网络协议应该对应的是传输层和网络层——如TCP协议用面向流的方式实现了端到端的可靠连接, 应属于传输层, 而IP协议则给要发送的数据添加了路由等信息, 故属于网络层。网络协议负责具体的服务, 那么负责与底层硬件打交道的就应该是网络接口驱动了。VxWorks下网络驱动有两种模式, 本文只涉及END, 及增强型网络驱动。网络驱动在OSI模型中对应的是数据链路层: 因为它在此层对数据进行处理, 从而使数据能够在物理层进行传输。VxWorks的网络体系如下页图1所示。

实际上在VxWorks的网络体系中, 协议并不直接调用下层的驱动, 这两者之间还有一个MUX层。上层的协议只能通过图示的MUX层中的各种接口函数来与下层的驱动程序进行绑定, 但使用VxWorks默认的网络协议栈和使用自己开发的非标准的网络协议栈在调用方式上是有很大不同的。默认的网络协议栈实际上就是指VxWorks BSD 4.3标准协议栈, 它和通用的BSD 4.3是完全兼容的。实际在VxWorks的网络体系下进行开发工作时, 有时需要根据用

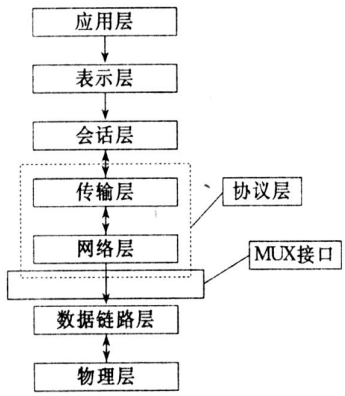

图 1 VxWorks 的网络体系结构

户的需求来开发一些非标准的网络协议栈。协议通过 muxBind() 等函数来绑定下层的 END，以此来实现与 MUX 接口的通信。当协议通过 MUX 接口与 END 驱动实现了绑定后，就可以调用 MUX 层的函数为自己提供服务了。如果使用的协议是默认协议，系统则会用已封装好的函数 ipAttach() 来调用 muxBind() 的功能，以便与底层驱动捆绑。

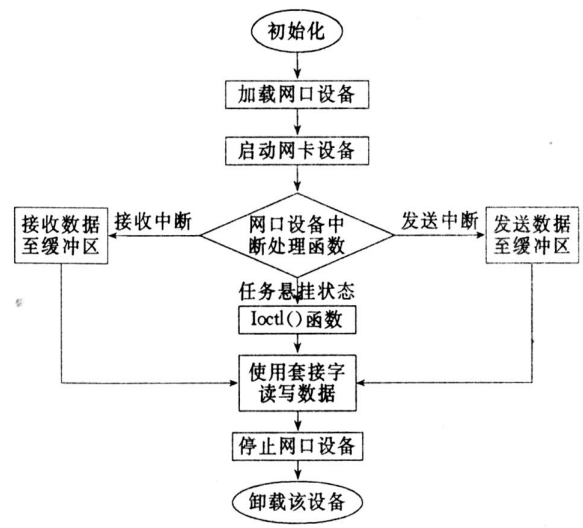

图 2 END 程序的流程

2 VxWorks 下 END 程序的流程

在 VxWorks 下开发网口设备的 END 程序，一般流程如图2 所示，这里可以把整个流程分为8 个部分。第1 部分——初始化部分，网口设备首先要在硬件上初始化，这通常由 END 中的 xxxInit() 函数来完成。第2 部分——设备加载，初始化后，还需要在程序中加载此设备，使用的是 xxxLoad() 函数。第3 部分是网络设备启动。加载函数中通常要发起任务，最重要的是中断处理任务——这就是第 4 部分，值得注意的是，这里的中断处理任务由于要避免阻塞函数的使用，要分成两部分——第一个函数只是简单地进入中断，做标记，这是第 5 部分；同时释放信号灯以同步真正的中断服务程序，后者中需要完成所有中断服务功能这是第 6 部分。中断服务程序根据不同的中断请求将数据或收发到不同的缓冲区，或将任务挂起；网络数据进入相应的缓冲区内就可以使用套接字的方式进行读写了。完成数据读写后，网络设备就可以进行第 7、8 部分——停止并卸载了。

3 VxWorks 下 END 程序的构成

在 VxWorks 下进行驱动程序的开发，程序的结构和流程往往大同小异，但不同的硬件平台下具体的实现细节还是会有所不同。MPC8280 的 END 开发也有其自己的特色，具体表现为：在 MPC8280 中集成的以太网控制器可工作于 FCC 模式，而工作于 FCC 模式的以太网控制器在收发网络数据帧的实现方式上都与一般方式不同。MPC8280 中集成的以太网控制器可以工作为两种模式：FCC 和 SCC。FCC 是 Fast Communication Controls 的缩写，而 SCC 是 Serial Communication Controls 的缩写，这两者间的区别是前者的网络连接速度更快，对网络数据的收发的管理和对网络数据缓冲区的管理更可靠。这两种不同模式的实现首先需要将 MPC8280 以不同方式与网络驱动的物理层芯片相连接（管脚连接不同），其次还需要对寄存器进行不同的设置。选定了不同工作模式后，程序中收发数据、管理缓冲区的方式就不同，本设计中 MPC8280 工作在 FCC 模式下。下面就具体讲下程序实现的细节。

3.1 END 中重要的数据结构

首先要详细地了解驱动所针对的特定的硬件的技术资料，如该设备的数据收发流程等。本文所进行的驱动程序开发是针对一块以 MPC8280 为处理器板为例，MPC8280 支持 100M 的网络传输速率，驱动 3 个物理层芯片与外界进行数据交互，实际上网口驱动就是用来驱动 MPC8280 芯片中的以太网控制模块工作，从而使板子能完成正常的网络数据收发工作的。

驱动中首先要根据网络接口硬件的特点来确定网络数据帧的封装结构，本文中所开发的 END 中定义了一个数据结构来完成对网络接口设备所收发的数据帧进行封装，这个数据结构在这里是这样定义的：

```
typedef struct
    {
      int len;
      unsigned char type; /* DL_DATA: data from end; DL_COMMAND: command from console */
      unsigned char data[DL_FRAME_MSG_SIZE];
    } DL_InterfaceMsg;
```

从数据结构“DL_InterfaceMsg”（DL 代表 DataLink）可见在这里本 END 设备实际上封装了在数据链路层需要收发的数据，并且结构体本身就存放了数据。所要传输的数据在这里分为两种类型：数据型和控制命令型。而宏 DL_FRAME_MSG_SIZE 则定义了可存放一帧数据的单个缓冲区的大小，在这里赋值为 1 600，这个值的赋予是有原因的：从硬件手册中查得，MPC8280 网络数据帧的最大长度是 1 526 个字节。网络数据帧就盛放在 DL_InterfaceMsg 的 unsigned char data[DL_FRAME_MSG_SIZE] 中，它的详细结构在硬件

手册中也可以查到:由7部分构成,分别是同步头、帧起始标志字节、源/目的地址、类型、用户数据和校验和。此外,在开发适于MPC8280的驱动时,还要掌握8280中的网络数据收发缓冲区的结构。图3分别列出了数据接收和发送的缓冲区结构。

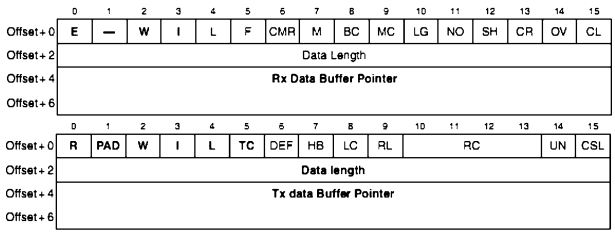

图3　MPC8280网络数据收发缓冲区结构

可以看到接收缓冲区长为8个字节,开始的两个字节长度用来存放各种设置的值,紧接其后的两个字节是数据长度,最后的4个字节用来存放所需接收数据的指针。这里简要介绍头两个字节的各标志位:第0位"E"(empty),是用来表明缓冲数据是否为空;第1位留待扩展;第2位"W"(wrap),用来表明当前缓冲区是否为缓冲区链表的最后一个;第3位"I"(interrupt),表明此缓冲区使用后是否有中断产生;第4位"L"(last in frame);第5位"F"(first in frame);第6位"CMR"(CAM match result for the frame);第7位"M"(miss);第8位"BC"(broadcast address);第9位"MC"(Multicast address);第10位"LG"(Rx frame length violation);第11位"NO"(Rx nonoctet aligned frame);第12位"SH"(Short frame);第13位"CR"(Rx CRC error);第14位"OV"(Overrun);第15位"CL"(Collision)。发送缓冲区长也为8个字节,结构也和接收缓冲区相同,只是前两个字节的标志位的定义不同,这里就不再一一介绍了。

在本文中,笔者使用的是Wind River Workbench 3.0(集成了VxWorks 6.6)这一开发环境,它提供了很丰富的资源给开发者,避免了一切都从零开始的繁琐劳动,其中最重要的就是一系列的END开发模板。这些模板被放在安装路径下:\target\src\drv\end中。里面有很多常见芯片的END模板,但最具通用性的是templateEnd.c,从END模板中很容易发现,构建一个基本的END也需要很多重要的数据结构,但是倘若只需简单的收发功能,模板中的数据结构大多不需修改。这里重点介绍一种需要根据自己需求进行修改的数据结构:END_DEVICE。这个结构在templateEnd.c中可以见到,实际应用中常将其置于头文件中,内部成员也和模板中的不尽相同,名字可根据驱动的名字将其命名为XXX_DEVICE。本文中在原有的END_DEVICE的模板的基础之上添加了一些新成员:用以完成数据的收发。现将它们列出如下:

```
int mtu;
    M_BLK_ID pRecvMblk;
    char * pRecvCluster;
    CL_BLK_ID pRecvCIBlk;
    int adfTaskId;
    char * pDrvName;
    char * pNodeName;
    DL_INTERFACE_ID DLid;
    int bindPort[ComMaxUnit];
```

其中的DLid正是前文中所述的用来对设备收发的数据帧进行封装的数据结构。DLid是网络设备硬件操作时对数据的封装,除此之外,为了便于内核代码对网络数据帧进行解析,VxWorks内核对网络数据的封装也有明确的要求,见上述代码中的pRecvMblk、pRecvCluster、pRecvCIBlk这三者代表了3个不同层次的内核数据帧封装结构mBlk、clBlk以及Cluster。底层驱动将数据帧传给上层前,要先将其封装进一个叫M_BLK_ID的内核数据结构中,而M_BLK_ID实际是一种mBlk的衍生数据结构——是mBlk类型的指针,所有相关的结构体的定义都在安装目录下h/netbufLib.h文件中。Cluster是指底层缓冲区,每个clBlk对应其中的一块,缓冲区地址由该结构体中的成员clNode.pClBuf来指向;mBlk与clBlk相比是一种比较上层的结构,此结构体中的*pCIBlk用于指向对应的clBlk,为了避免内核数据复制的麻烦,一个clBlk可以同时被几个mBlk共享。END_DEVICE中其余的几个新增结构体作用由名称都可以方便的得知,在这里不再多说。

另外还有一个END与MUX接口共享的非常重要的数据结构数据结构:END_OBJ需要在此介绍一下。这个数据结构用来封装网络接口设备的相关参数,同时它也是很多与END开发有关的数据结构的基础,如前文中的END_DEVICE中就有END_OBJ类型的数据成员。END驱动中的一个非常重要的函数即数据装载函数(xxxLoad()函数)的返回值为END_OBJ型的,这是因为END_OBJ结构体中包含了END提供给MUX接口层的所有的设备信息,还有指向END标准进入配给函数的指针(所有标准进入配给函数都封装在一个名为NET_FUNC的结构体中,指针即指向这个结构体)。

3.2　END驱动中的主要函数

END驱动函数的主体是END的标准进入配给函数,这些函数被MUX接口层调用来为上层协议服务。标准进入配给函数共有14个,在模板文件中通常会有一个NET_FUNC类型的结构体会将它们罗列出,读者可参见模板了解其详细信息。其中不少函数在实际驱动的编写中作用不大,且在原模板的基础上无需多大改动,故在此只对其中的少数几个重要函数进行介绍,分别是:网络设备加载函数xxxLoad(),网络设备启动函数xxxStart(),网络数据帧接收任务入口函数xxxRcvTask(),数据帧发送中断服务函数xxxSend(),中断服务任务和内存的相关的初始化函数。

加载函数xxxLoad()的实现。实际上MUX层的函数muxDevLoad()将两次调用这个函数。第一次调用时,初始化字符串initString是一个空字符串,此时的xxxLoad()只是将驱动对应的名称复制到initString指向的地址中,这样当第二次调用加载函数时,初始化字符串才是指定的那个字符串

(在configNet.h 中指定的)。程序首先要判断加载函数是第几次被调用,这是通过查看初始化字符串来实现的,当初始化字符串不为空时,一方面要对它进行解析,另一方面也是最重要的操作——将底层驱动函数集合安装到END OBJ结构中,供MUX 层使用,这样才能实现驱动函数的加载,这部分的代码在下文给出,并以粗体体现。

```
if (END OBJ INIT (&pDrvCtrl->end, NULL, "usrAdfEND",
    pDrvCtrl->unit, &usrAdfENDFuncTable,
    "A kind of adf END.") == ERROR
||END MIB INIT (&pDrvCtrl->end, M2 ifType ethernet csmacd,
    &pDrvCtrl->enetAddr[0], 6, ADFEND BUFSIZ,
    ADFEND SPEED)
== ERROR)
goto errorExit;
```

此处的驱动函数集合由usrAdfENDFuncTable 来表示,其中成员皆为END 的标准进入配给函数。加载后,网络设备的启动由函数xxxStart()来实现,启动函数的思想主要就是注册与使能中断,设置END OBJ 中的相关标志位,使设备开始正常工作。在这里由于网络设备的正常工作主要是数据的收发,而数据收发通过发起任务实现,故这里的启动函数内容涉及任务的发起,比较重要,这里给出部分源码:

```
STATUS result= OK;
UINT32 val;
char * taskName;
char * string;
result = intConnect (pDrvCtrl, usrAdfENDInt, (int) pDrvCtrl); //注册一个函数usrAdfENDInt 为中断响应函数,稍后会实现
SYS INT ENABLE (pDrvCtrl);
sprintf (string," tDL%s%dIntS ", pDrvCtrl->DLid->devName, pDrvCtrl->DLid->devUnit);
if ((pDrvCtrl->DLid->sendTaskId = taskSpawn ("DlSendTask", 140, 0, 10000, (FUNCPTR) dlInterfaceSendTask, (int)pDrvCtrl->DLid, 0, 0, 0, 0, 0, 0, 0, 0, 0))== ERROR)
    {
      DRV LOG (DRV DEBUG LOAD,"Can't creat a datalink! \n", 1, 2, 3, 4, 5, 6);
      return ERROR;
    }
sprintf (string," tDL%s%dIntR ", pDrvCtrl->DLid->devName, pDrvCtrl->DLid->devUnit);
if ((pDrvCtrl->DLid->recvTaskId = taskSpawn ("DlRecvTask", 140, 0, 10000, (FUNCPTR) dlInterfaceRecvTask, (int)pDrvCtrl->DLid, 0, 0, 0, 0, 0, 0, 0, 0, 0))== ERROR)
    {
      DRV LOG (DRV DEBUG LOAD,"Can't creat a datalink! \n", 1, 2, 3, 4, 5, 6);
      return ERROR;
    }
sprintf(taskName,"endDev%s",pDrvCtrl->unit);
if (pDrvCtrl->endTaskId = taskSpawn (taskName, usrAdfENDTaskPriority, usrAdfENDTaskOptions, usrAdfENDTaskStackSize, (FUNCPTR)usrAdfENDRcvTask, (int) pDrvCtrl, 0, 0, 0, 0, 0, 0, 0, 0, 0)== ERROR)
{/* 新增了一函数 */
  DRV LOG (DRV DEBUG LOAD,"Can't creat a receive task!", 1, 2, 3, 4, 5, 6);
  return ERROR;
}
```

在程序中共发起了3 个任务,其中2 个是数据链路层的任务,入口函数也不在驱动中实现,仅有1 个是网络设备的数据帧接受任务,它的实现是在本驱动中。网络接受任务的优先级usrAdfENDTaskPriority 的值设定为151,比数据链路层的任务优先级低,这样是符合实际的网络结构模型的。具体的接受任务的入口函数xxxRcvTask()的实现和通常情况下从网络接收数据的方法是一样的:建立缓冲区,再通过使用系统提供的缓冲队列来进行数据的收发。

启动后的网络设备要真正实现提供稳定的数据收发服务,就需要用到硬件的中断响应机制。驱动中使用xxxInt()作为中断服务程序,xxxHandleRcvInt()作为中断服务任务的入口函数。在xxxInt()中通过读MPC8280 的状态寄存器中RINT 和RXON 这两位的值,来加载实际提供中断服务的xxxHandleRcvInt()。当以上两位都为1 时,就调用netJobAdd()来加载xxxHandleRcvInt(),这样网络控制模块就可以接受网络中传送的数据帧了。在xxxStart()中发起的数据链路层任务是在数据链路层进行数据帧的收发,而网络层的数据帧接收任务xxxRcvTask()就是从数据链路层接收数据而后将其转到IP 层,xxxHandleRcvInt()的实现也需要调用它。

网络设备进行数据帧的收发必然会使用一定量的系统内存来满足网络数据缓冲的需要,因此,对相应的芯片进行系统内存的初始化是必不可少的操作,本文中针对MPC8280 进行内存初始化的函数是xxxMemInit()。初始化8280 内存可分为两个步骤,第一步是设置网络存储区的3 种数据封装结构mBlk、clBlk 以及Cluster 的大小及数目,同时设置了管理整个存储区的内存池的大小。对不同层次的内核数据封装结构进行初始化的时候,其值通过MPC8280 的数据手册得到,如clBlk 的数目为32,mBlk 的数目为64——这里要说明的是:mBlk、clBlk 是不同层次上对内核结构的封装(见前文),所以mBlk 的数目一般要比clBlk 大。由于内存池是系统结构体end 中的一个数据成员,对其设置一方面为其分配了一定内存,另一方面是用netClPoolIdGet()为其分配ID,分配ID 的过程中要指明内存池大小,这可通过查阅MPC8280 的数据手册得知为1 540。第二步是计算数所有的mBlk、clBlk 所占用的系统存储区的总大小,为其分配内存,同时初始化内存池。

网络数据帧的发送函数xxxSend()。发送函数首先要判定续传送的数据帧的大小是否超出所允许的一帧的最大值,

若超过则报错; 是否低于所允许的一帧的最小值, 若低于则补零至最小值后再发送。函数主要用 END_TX_SEMTAKE 这个系统自定义的宏来获取放置在 END_OBJ 结构中的信号灯启动发送工作(用后再用 END_TX_SEMGIVE 来释放信号灯), 启动后主要使用两个系统函数 netMblkToBufCopy() 和 netMblkClChainFree() 来执行数据从底层缓冲区复制到用户指定缓冲区的操作和释放内存缓冲区的操作。数据到达用户缓冲区后, 就需要用到双缓冲队列进行数据的发送, 这和使用套接字进行通信并无本质区别, 在这里不再进行说明。

3.3　与 END 开发相关的 BSP 中的修改

除了动手编写 END 的代码之外, 还应该考虑的是如何让系统加载上一个额外的网络设备, 这需要在 END 自身的代码外实现。在 VxWorks 中的 BSP 中定义了一个网络设备分组表: endDevTbl[]—— 开发者在安装目录: \target\config\bspname 中的 configNet.h 中可见到其源码, 这其中的 bspname 是指相应板卡的 BSP 的名字。要想让系统正确加载 END 设备, 开发者必须手动修改 configNet.h 中的代码, 简单一点说就是在 endDevTbl[] 为该设备中添加一行以便系统的初始化时会自动加载该设备。为了能直观地显示出具体的操作内容, 特给出部分代码作参考:

```
#define ADFEND_LOAD_FUNC usrAdfENDLoad
#define ADFEND_LOAD_STRING "0:100:1:1"
#define ADFEND_BUFF_LOAN 0
IMPORT END_OBJ* ADFEND_LOAD_FUNC(char*, void*);
END_TBL_ENTRY endDevTbl[] =
{
#ifdef INCLUDE_ADFEND
{0, ADFEND_LOAD_FUNC, ADFEND_LOAD_STRING, ADFEND_BUFF_LOAN, NULL, FALSE}
#endif
}
```

上面的代码中的 endDevTbl[] 表中仅一行, 这说明此类型的 END 设备仅有一个。而且实际上, END 设备的类型也只有一个, 因此, 只有一行"#ifdef INCLUDE_ADFEND", 如果需要添加其他类型的设备则应该在前面定义, 形式是: #define INCLUDE_XXX。来观察表中这一行的值, 其中的"0"是该类型设备的序号, ADFEND_LOAD_FUNC 用来指定 END 的进入配给函数, ADFEND_LOAD_STRING 用来指定函数 muxDevLoad() 中的参数 initString 的初始化字符串。每个条目后的 FALSE 表明该条目对应的设备未被装载, 此时必须为 FALSE, 待系统正确加载该设备后, 其值将自动变为 TRUE。

4　结　论

END 的开发对于从事网络方面的嵌入式开发的人员来说, 是一项有必要掌握的技术。本文探索和总结了一些在 MPC8280 下 END 开发相关的方法和技巧, 但考虑到除去硬件相关的设置, END 的开发方法具有一定的通用性, 因此, 对其他情况下的 END 开发也会有一定的参考价值。

参考文献:

[1]　周启平, 张　杨. VxWorks 下设备驱动程序及 BSP 开发指南[M]. 北京: 中国电力出版社, 2004.

[2]　程敬原. VxWorks 软件开发项目实例完全解析[M]. 北京: 中国电力出版社, 2005.

[3]　陈智育, 温彦军, 陈　琪. VxWorks 程序开发实践[M]. 北京: 人民邮电出版社, 2004.

[4]　曹桂平. VxWorks 设备驱动开发详解[M]. 北京: 电子工业出版社, 2011.

(上接第 106 页)

炮塔信息、弹药信息四屏。车辆信息包括车辆位置、车辆速度、发动机水温和车辆状态。炮塔信息包括炮塔和瞄线角度、目标位置、目标角速度、测距方式和瞄准方式。弹药信息包括气温、横风、药温、炮弹初速、弹种和剩余弹量等。

图 2 是车辆信息界面, 显示信息包括车辆位置(海拔、经纬、倾斜角)、车辆速度(航向、速率)、发动机水温(水温、沸点、气压)和车辆状态(正常、故障)等。用户可以设置水温预警温度, 超过该温度时会发出接近沸点警报。

6　结　论

基于 ADS512101 嵌入式开发板, 完成了针对装甲火控系统的中文图形用户界面的开发, 实现了 CAN 总线数据处理和中文显示功能。由图 2 可以看出图形的显示表现出明显的锯齿, 需要合适的反走样方法, 图形外观也可以进一步美化。另外由于火控系统对实时性要求较高, 图形渲染的质量和速度还需要平衡取舍。

参考文献:

[1]　邢丽娟, 杨世忠. 虚拟仪器的原理及发展[J]. 山西电子技术, 2006, 30(1): 90-91.

[2]　薛　娟, 李旭勇. 基于 VxWorks 的 OpenGL 图形界面开发[J]. 计算机工程, 2010, 36(24): 269-271.

[3]　Sunmoon. 终于能在 OPENGL-ES 上贴中文字体了!! [EB/OL]. http://hi.baidu.com/s_rlzheng/blog/item/b53227169f6aaa4c21a4e97d.html, 2008-10-17.

[4]　阮一峰. 字符编码笔记: ASCII, Unicode 和 UTF-8 [EB/OL]. http://www.ruanyifeng.com/blog/2007/10/ascii_unicode_and_utf-8.html, 2007-10-28.