OMAPL138 基于 SYSLINK 的双核通信 LED 实例

Revision History

Revision No.	Description	Draft Date	Remark
V1.0	1.初始版本	2014/2/25	

阅前须知

版权声明

广州创龙电子科技有限公司保留随时对其产品进行修改和完善的权利，同时保留在不作任何通告的情况下，终止其任何一款产品的供应和服务的权利。请用户在购买前向我司获取相关产品的最新信息，本文档一切解释权归广州创龙所有。

公司简介

德州仪器（TI）第三方技术合作伙伴——广州创龙电子科技有限公司（简称“广州创龙”，英文简称“Tronlong”），是杰出的嵌入式方案商，专业提供嵌入式开发平台、软硬件定制设计及技术支持等服务，专注于 DSP+ARM+FPGA 三核系统方案开发，和国内诸多著名企业、研究所和高校有密切的技术合作，如富士康、三一重工、中国科学院、清华大学、中国航空航天大学等国内龙头企业和院校。

TI 嵌入式处理业务拓展经理 Zheng Xiaolong 指出：“Tronlong 是国内研究 OMAP-L138 最深入的企业之一，Tronlong 推出 OMAP-L138+Spartan-6 三核数据采集处理显示解决方案，我们深感振奋，它将加速客户新产品的上市进程，带来更高的投资回报率，使得新老客户大大受益。”

经过近几年的发展，创龙产品已占据相关市场主导地位，特别是在电力、通信、数控、音视频处理等数据采集处理行业广泛应用。创龙致力于让客户的产品快速上市、缩短开发周期、降低研发成本。选择创龙，您将得到强大的技术支持和完美的服务体验。

产品保修

广州创龙所有产品保修期为一年，保修期内由于产品质量原因引起的，经鉴定系非人为因素造成的产品损坏问题，由广州创龙免费维修或者更换。

更多帮助

请浏览广州创龙官网：www.tronlong.com

公司总机：020-8998-6280

技术邮箱：support@tronlong.com

销售邮箱：sales@tronlong.com

目　　录

1 实例编译

光盘中 demo/syslink/ex10_led 实例实现了利用 MCSDK 的 SYSLINK 组件在 ARM 端控制 DSP 端来操作开发板外设 LED 执行跑马灯程序。本实例是基于 ex03_notify 增加 DSP 控制 LED 功能。

先按照广州创龙 OMAPL138 开发板的用户手册《基于 OMAPL138 的多核软件开发组件--MCSDK 开发教程.pdf》安装 MCSDK，配置、编译和安装 SYSLINK。然后将 ex10_led 文件夹拷贝到虚拟机/home/tl/ti/syslink_2_21_01_05/examples 目录下（该路径不可随意放置，否者无法包含到 SYSLINK 里面的头文件），然后进入 ex10_led 目录，如下图所示：

```
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples$ pwd
/home/tl/ti/syslink_2_21_01_05/examples
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples$ ls
archive          ex04_sharedregion  ex08_ringio     ex34_radar
ex01_helloworld  ex05_heapbufmp     ex09_readwrite  makefile
ex02_messageq    ex06_listmp        ex10_led        readme.txt
ex03_notify      ex07_gatemp        ex33_umsg       scripts
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples$ cd ex10_led/
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$ ls
dsp  host  makefile  products.mak  readme.txt  run.sh  shared
```

图 1

执行“sudo make clean”清除编译生成文件，执行“sudo make”命令重新编译该例程，如下图所示：

```
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$ sudo make clean
[sudo] password for tl:
rm -rf install
make -C host clean
make[1]: Entering directory `/home/tl/ti/syslink_2_21_01_05/examples/ex10_led/ho
st'
rm -rf bin
make[1]: Leaving directory `/home/tl/ti/syslink_2_21_01_05/examples/ex10_led/hos
t'
make -C dsp clean
make[1]: Entering directory `/home/tl/ti/syslink_2_21_01_05/examples/ex10_led/ds
p'
rm -rf configuro bin
make[1]: Leaving directory `/home/tl/ti/syslink_2_21_01_05/examples/ex10_led/dsp
```

图 2

```
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$ sudo make
#
# Making all ...
make -C host all
make[1]: Entering directory `/home/tl/ti/syslink_2_21_01_05/examples/ex10_led/ho
st'
#
# Making all ...
make PROFILE=debug app_host
make[2]: Entering directory `/home/tl/ti/syslink_2_21_01_05/examples/ex10_led/ho
st'
#
# Making bin/debug/obj/main_host.ov5T ...
/home/tl/arm-2009q1/bin/arm-none-linux-gnueabi-gcc -c -MD -MF bin/debug/obj/main
```

图 3

在该目录的 dsp/bin/debug/目录下生成.xe674 格式文件 server_dsp.xe674，如下图所示：

```
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$ ls dsp/bin/debug/
obj  server_dsp.xe674
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$
```

图 4

在该目录的 host/bin/debug/目录下生成 Linux 端可执行程序 app_host，如下图所示：

```
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$ ls host/bin/debug/
app_host  app_host.map  obj
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$
```

图 5

2 实例演示

执行此实例双核通信需要 4 个文件，syslink.ko、slaveloader、server_dsp.xe674 和 app_host。按照《基于 OMAPL138 的多核软件开发组件--MCSDK 开发教程.pdf》教程完成 SYSLINK 编译和安装后，syslink.ko 和 slaveloader 将位于开发板文件系统如下位置：

syslink.ko：/lib/modules/3.3.0/kernel/drivers/dsp/syslink.ko

slaveloader：开发板任意 example 的 debug 目录中，如/ex03_notify/debug/slaveloader。

以下为各个文件的作用：

syslink.ko：双核通信驱动。

slaveloader：用于 ARM 端启动 DSP 并加载.xe674 格式的 DSP/BIOS 文件，例如 server_dsp.xe674。

server_dsp.xe674：DSP 端应用程序。在此实例中，增加的 DSP 端控制 LED 流水灯功能的代码镜像就是 server_dsp.xe674。

app_host：ARM 端应用程序。

将以上编译出来的 slaveloader、server_dsp.xe674、app_host 和 ex10_led 中的 run.sh 拷贝到开发板同一个目录下，例如开发板的根目录：

```
root@tl:/# pwd
/
root@tl:/# ls
Settings            ex04_sharedregion  lib              sbin
app_host            ex05_heapbufmp     linuxrc          server_dsp.xe674
bin                 ex06_listmp        media            slaveloader
boot                ex07_gatemp        mnt              srv
dev                 ex08_ringio        nfs              sys
etc                 ex09_readwrite     opt              tmp
ex01_helloworld     ex33_umsg          proc             usr
ex02_messageq       ex34_radar         run.sh           var
ex03_notify         home               runall.sh
root@tl:/#
```

图 6

进入开发板的 Linux 文件系统后，执行如下命令安装双核通信驱动：

Targert# insmod /lib/modules/3.3.0/kernel/drivers/dsp/syslink.ko TRACE=1 TRACEFAILURE=1

```
root@tl:/# insmod /lib/modules/3.3.0/kernel/drivers/dsp/syslink.ko TRACE=1 TRACE
FAILURE=1
[  165.271635] SysLink version : 2.21.01.05
[  165.271666] SysLink module created on Date:Feb  5 2014 Time:14:34:59
[  165.292810] Trace enabled
[  165.295491] Trace SetFailureReason enabled
root@tl:/# _
```

图 7

然后执行“./run.sh”命令，观察发现 LED 会先闪烁两次，再依次点亮所有 LED，接着依次熄灭所有 LED。

Target# ./run.sh

```
root@tl:/# ./run.sh
+ ./slaveloader startup DSP server_dsp.xe674
Attached to slave procId 0.
Loading procId 0.
Loaded file server_dsp.xe674 on slave procId 0.
Started slave procId 0.
+ ./app_host DSP
--> main:
--> App_create:
--> App_exec:
<-- App_exec:
--> App_delete:
App_delete: Sending stop command
App_delete: Received---> Stop has been acknowledged
App_delete: Cleanup complete
<-- App_delete:
<-- main:
+ ./slaveloader shutdown DSP
Stopped slave procId 0.
Unloaded slave procId 0.
Detached from slave procId 0.
root@tl:/#
```

图 8

使用“cat run.sh”命令可以查看到 run.sh 脚本中的内容是：

```
set -x
./slaveloader startup DSP server_dsp.xe674
./app_host DSP
./slaveloader shutdown DSP
```

图 9

以下为脚本内容的解释：

./slaveloader startup DSP server_dsp.xe674：加载 DSP/BIOS 应用程序和启动 DSP 核。

./app_host DSP：启动 ARM 端 Linux 应用程序。

./slaveloader shutdown DSP：关闭 DSP 核。

3 实例解析

3.1 实例程序结构解析

在 ex10_led 目录中运行“tree -L 3”命令,可以看到实例程序目录的结构如下图所示：

```
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$ tree -L 3
.
├── dsp
│   ├── bin
│   │   ├── debug
│   │   └── release
│   ├── configuro
│   │   ├── compiler.defs
│   │   ├── compiler.opt
│   │   ├── config.bld
│   │   ├── custom.mak
│   │   ├── linker.cmd
│   │   ├── package
│   │   ├── package.bld
│   │   ├── package.mak
│   │   ├── package.xdc
│   │   └── package.xs
│   ├── Dsp.cfg
│   ├── main_dsp.c
│   ├── makefile
│   ├── Server.c
│   └── Server.h
├── host
│   ├── App.c
│   ├── App.h
│   ├── bin
│   │   ├── debug
│   │   └── release
│   ├── main_host.c
│   └── makefile
├── makefile
├── products.mak
├── readme.txt
├── run.sh
└── shared
    ├── AppCommon.h
    ├── config.bld
    └── SystemCfg.h

11 directories, 25 files
tl@tl-desktop:~/ti/syslink_2_21_01_05/examples/ex10_led$
```

图 10

dsp：DSP/BIOS 源代码。

host：ARM 端 Linux 应用程序。

shared：ARM 和 DSP 内存共享相关。

products.mak：makefile 调用的配置文件，用于识别编译的头文件和库文件路径。

3.2 实例 DSP/BIOS 应用程序解析

dsp/main_dsp.c 中创建了 smain 任务，smain 任务会先执行 Server_init()，如下图所示：

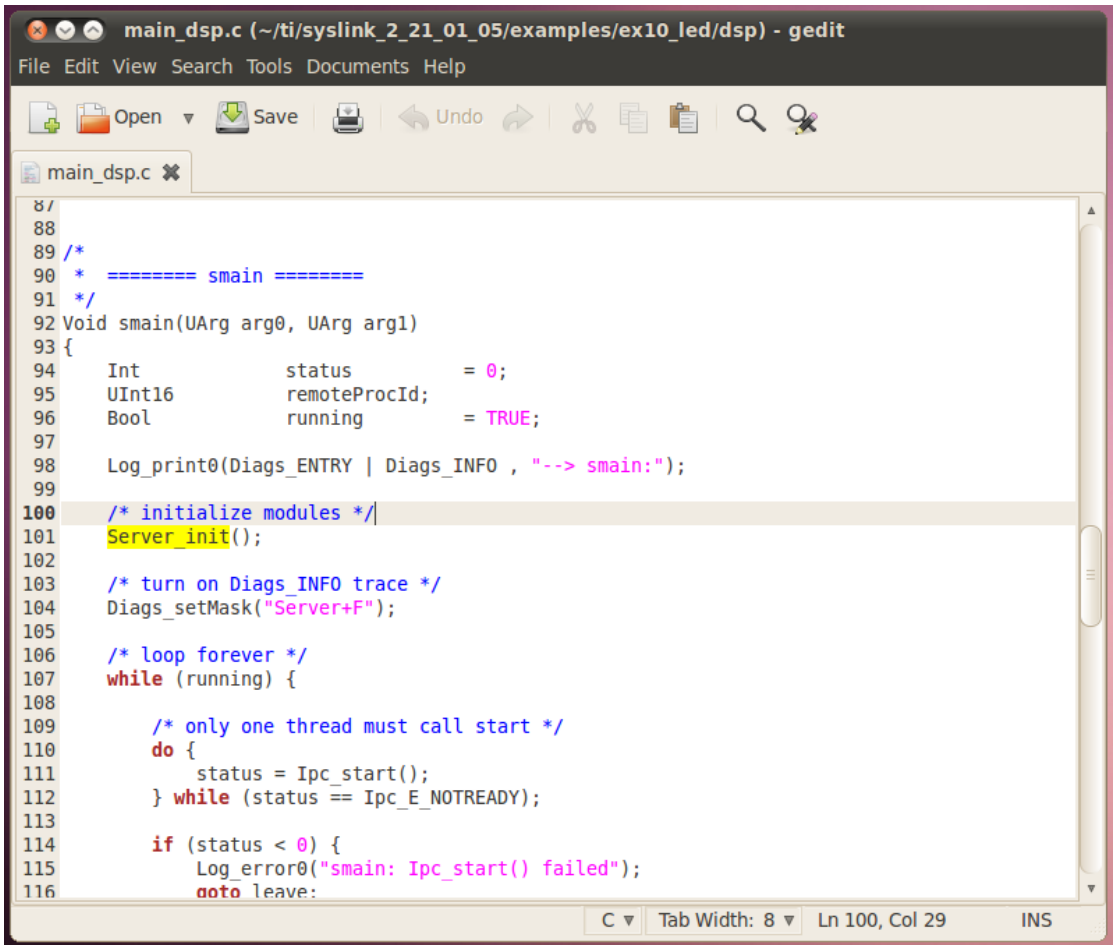

图 11

Server_init()在 dsp/Server.c 中定义，Server.c 是最常修改的 DSP/BIOS 文件。此实例在 Server.c 中增加了 LED 控制函数 led_init()，如下图所示：

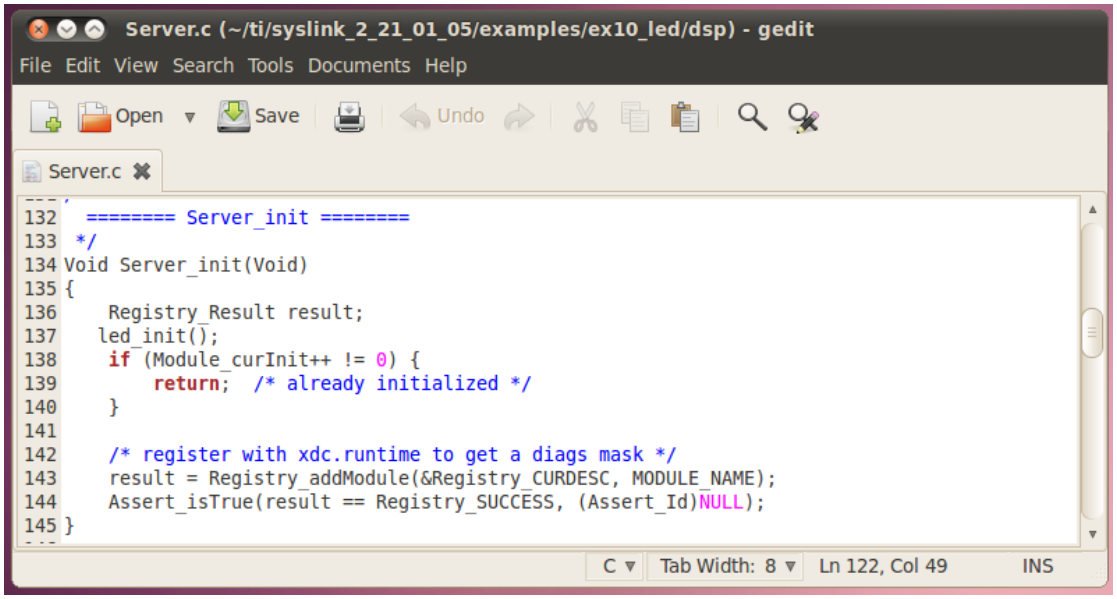

图 12

dsp/Server.c 中的 led_init()函数实现了 LED 对应的 GPIO 的基本配置。在初始化配置时让 4 个 LED 连续闪烁 2 次，如下图所示：

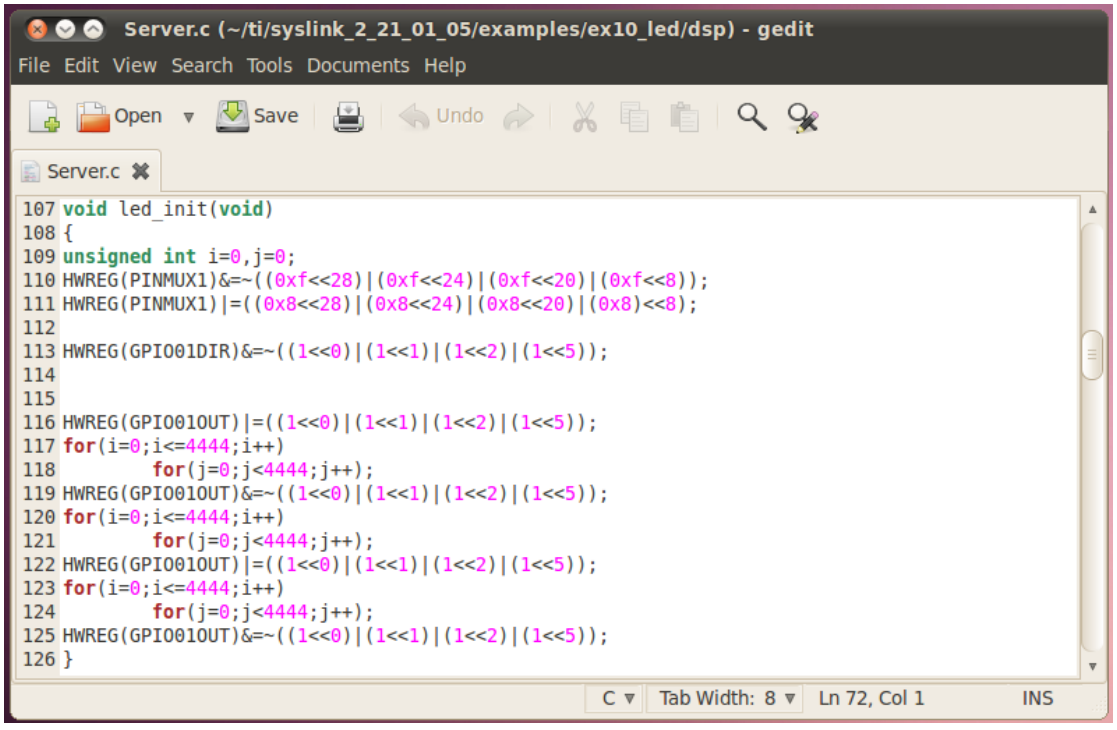

图 13

LED 对应的 GPIO 相关寄存器定义如下图所示：

图 14

DSP/BIOS 的 smain 任务完成后会执行 dsp/Server.c 中的 Server_create()函数。如下图所示：

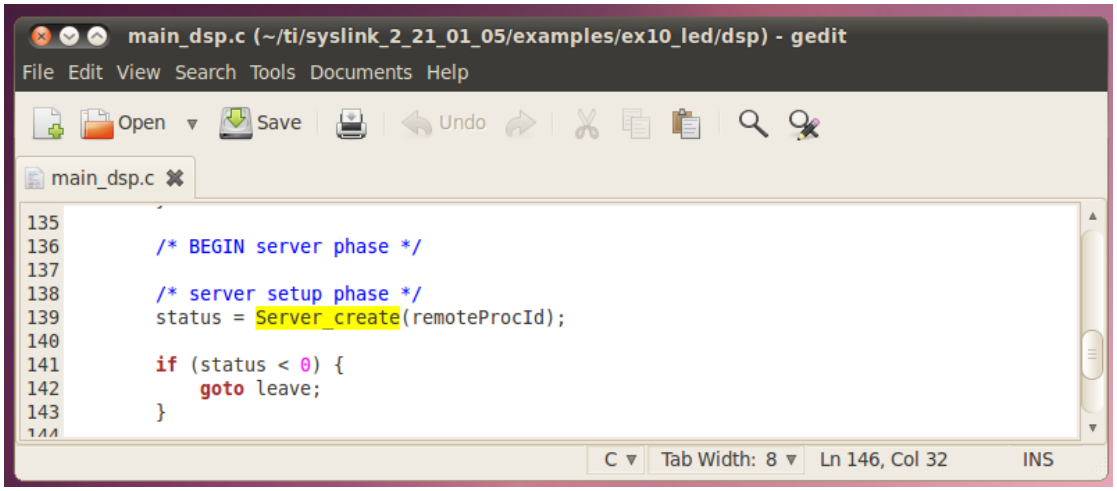

图 15

Server_create()函数在 dsp/Server.c 中定义，代码如下图所示：

Server.c (~/ti/syslink_2_21_01_05/examples/ex10_led/dsp) - gedit

File Edit View Search Tools Documents Help

Open Save Undo

Server.c

```
148 /*
149  *  ======== Server_create ========
150  *
151  *  1. create sync object
152  *  2. register notify callback
153  *  3. wait until remote core has also registered notify callback
154  */
155 Int Server_create(UInt16 remoteProcId)
156 {
157     Int                 status      = 0;
158     Semaphore_Params    semParams;
159
160     /* setting default values */
161     Module.eventQueue.head      = 0;
162     Module.eventQueue.tail      = 0;
163     Module.eventQueue.error     = 0;
164     Module.eventQueue.semH      = NULL;
165     Module.lineId               = SystemCfg_LineId;
166     Module.eventId              = SystemCfg_EventId;
167     Module.remoteProcId         = remoteProcId;
168
169
170     /* 1. create sync object */
171     Semaphore_Params_init(&semParams);
172     semParams.mode = Semaphore_Mode_COUNTING;
173     Semaphore_construct(&Module.eventQueue.semObj, 0, &semParams);
174     Module.eventQueue.semH = Semaphore_handle(&Module.eventQueue.semObj);
175
176     /* 2. register notify callback */
177     status = Notify_registerEvent(Module.remoteProcId, Module.lineId,
178             Module.eventId, Server_notifyCB,(UArg)  &Module.eventQueue);
179     if (status < 0)    return (status);
180
181
182     /* 3. wait until remote core has also registered notify callback */
183     do {
184         status = Notify_sendEvent(Module.remoteProcId,Module.lineId,
185                 Module.eventId, APP_CMD_NOP, TRUE);
186
187         if (status == Notify_E_EVTNOTREGISTERED) {
188             Task_sleep(100);
189         }
190     } while (status == Notify_E_EVTNOTREGISTERED);
191
192    return status;
193 }
```

C Tab Width: 8 Ln 155, Col 18 INS

图 16

Server_create()函数会注册 notify 事件。当 ARM 端 notify 事件注册时，DSP 会触发 Server_notifyCB 函数，接着执行 dsp/Server.c 中的 Server_exec()函数。如下图所示：

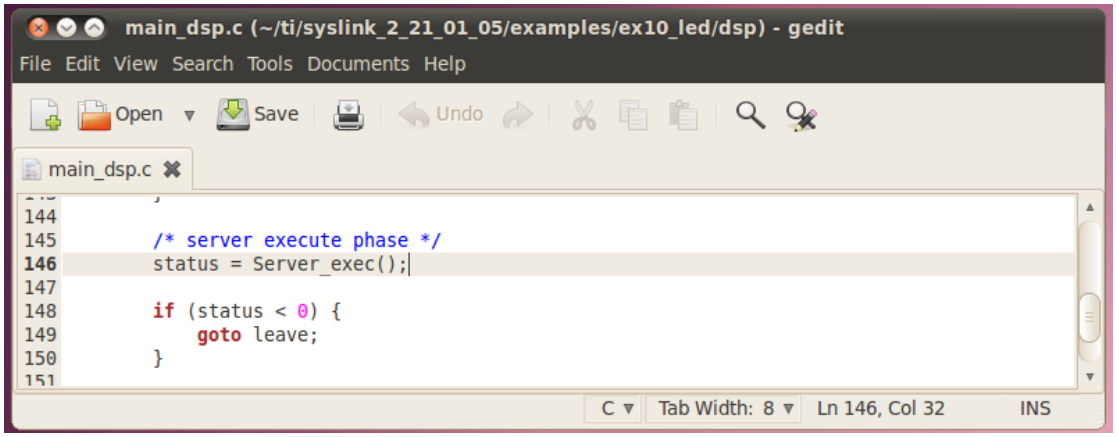

图 17

Server_exec()函数在 dsp/Server.c 中定义，该函数轮询等待 ARM 端发来的命令，其中 Server_waitForEvent()是一种信号量等待方式，当 ARM 端有命令传送过来时会解除等待，然后解析 ARM 端传入的命令，解析命令代码如下图所示：

```
/* retrieve only payload information */
num = (UInt16) (event & APP_PAYLOAD_MASK);

/* retrieve only command information */
event = (event & APP_CMD_MASK);


if (event == APP_CMD_ON_PAYLOAD) {
                switch(num){
                        case 0:HWREG(GPIO01OUT)|=(1<<0);break;
                        case 1:HWREG(GPIO01OUT)|=(1<<5);break;
                        case 2:HWREG(GPIO01OUT)|=(1<<1);break;
                        case 3:HWREG(GPIO01OUT)|=(1<<2);break;
                        }
                                for(i=0;i<=4444;i++)
                                        for(j=0;j<4444;j++);

}
else {
                switch(num)
                        {
                        case 0:HWREG(GPIO01OUT)&=~(1<<0);break;
                        case 1:HWREG(GPIO01OUT)&=~(1<<5);break;
                        case 2:HWREG(GPIO01OUT)&=~(1<<1);break;
                        case 3:HWREG(GPIO01OUT)&=~(1<<2);break;
                        }
                                for(i=0;i<=4444;i++)
                                        for(j=0;j<4444;j++);
}
```

图 18

从上图可以看出，ARM 传到 DSP 并解析出来的是 num 和 event 两个变量。APP_CMD_ON_PAYLOAD 将在下一章节解释。

3.3 实例 Linux 应用程序解析

host/main_host.c 功能和 dsp/main_dsp.c 类似，它初始化 SYSLINK，然后执行 host/App.c 中的 App_create()函数注册 notify 事件，等待 DSP 端创建 notify 事件后，接着执行 host/App.c 中 App_exec()函数。ARM 端在 App_exec()函数中向 DSP 发送控制 LED 的命令，代码如下：

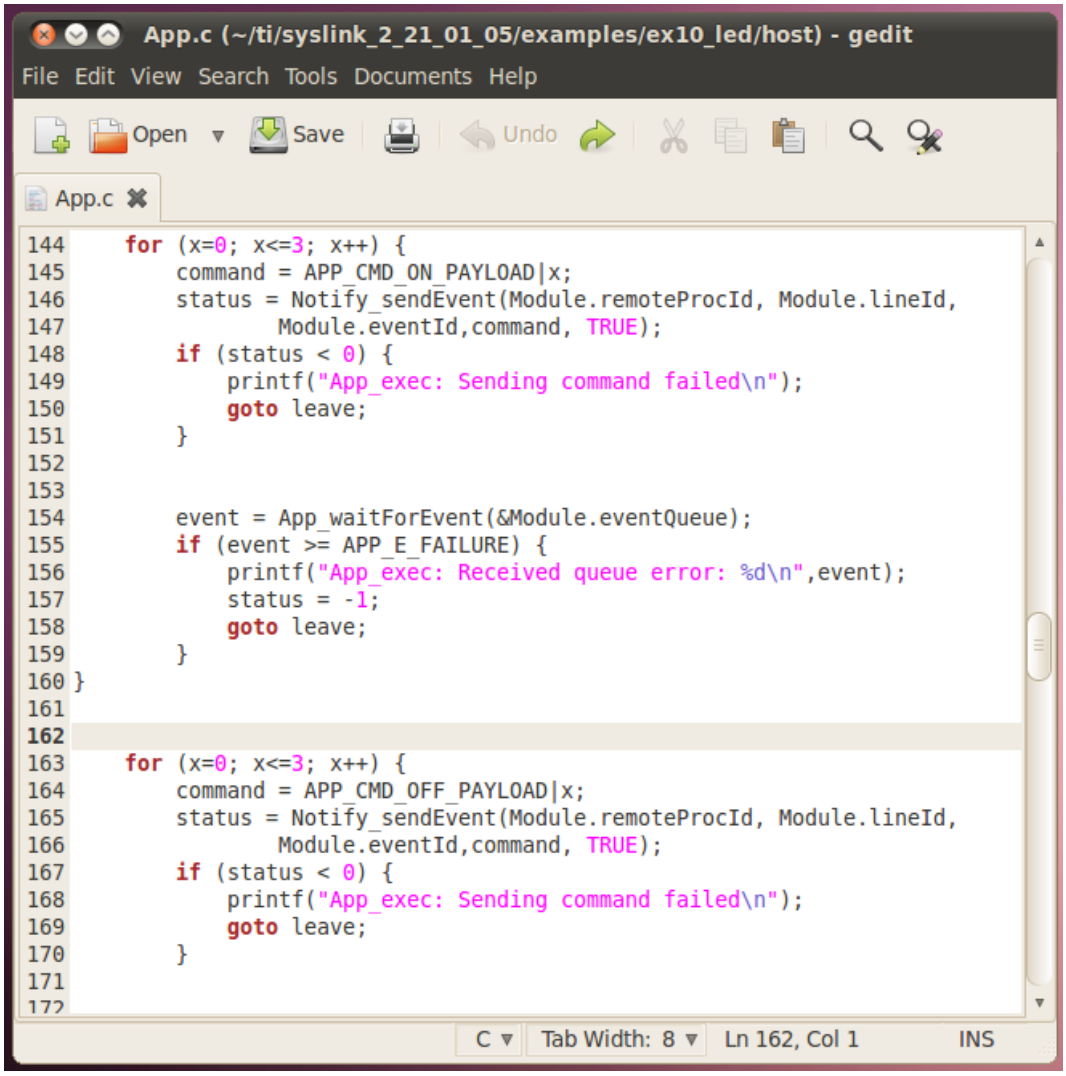

图 19

可以看出 ARM 端发送给 DSP 的命令有 8 个，分别是依次点亮 4 个 LED，再依次熄灭 4 个 LED。APP_CMD_ON_PAYLOAD 和 APP_CMD_OFF_PAYLOAD 分别表示控制 LED 亮和灭，x 分别为 4 个 LED 编号。控制状态和编号需要 DSP 端解析。所以 APP_CMD_ON_PAYLOAD 和 APP_CMD_OFF_PAYLOAD 是共享数据，其宏定义存放在 shared/AppCommon.h 中，如下图所示：

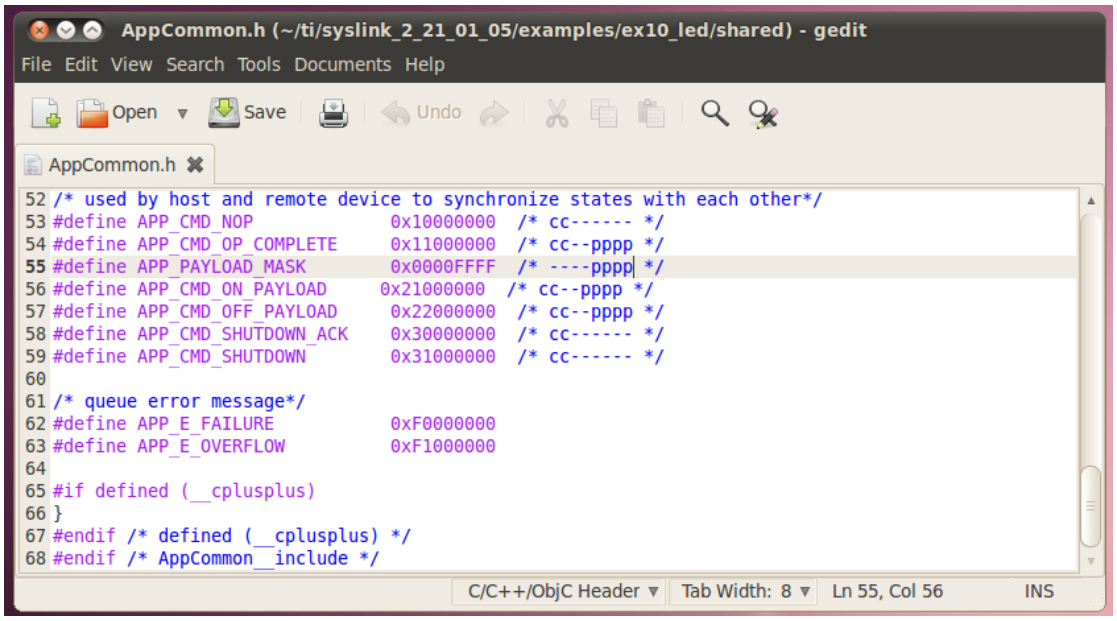

图 20

APP_CMD_ON_PAYLOAD 和 APP_CMD_OFF_PAYLOAD 宏是用户根据实际情况在 shared/AppCommon.h 中修改或者添加的，ARM 端和 DSP 端都会使用到。