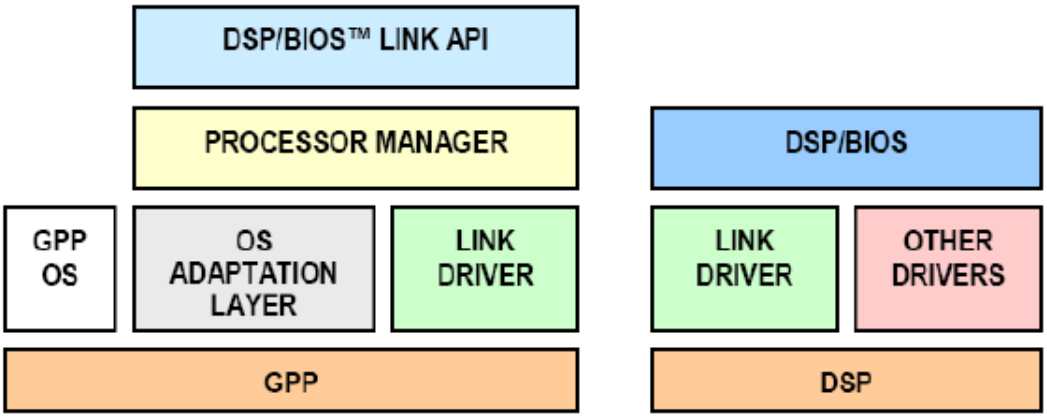
双核通信组件 DSPLINK 开发和例程演示教程

1 引言

DSPLINK 即 DSP/BIOS™ LINK，是通用处理器 GPP 与 DSP 端进行通信的驱动程序。本文将讲述基于 OMAPL138 平台下的 GPP 端(即 ARM 端)与 DSP 端双核通信组件 DSPLINK 内容。DSPLINK 提供了一套通用的 API，从应用层抽象出 ARM 与 DSP 的物理连接特性，从而降低用户开发程序的复杂度。

2 DSPLINK 介绍

DSPLINK 的软件架构如下图示：



2.1 GPP 端

GPP(General Purpose Processor)是指和 DSP 端通信的通用处理器，在本文特指 OMAPL138 平台的 ARM 端。

GPP OS：指通用处理器上的操作系统，本文特指 OMAPL138 平台 ARM 端 Linux 操作系统。

OS ADAPTATION LAYER：指操作系统抽象层，包含了 DSPLINK 需要的一些通用的 OS 服务部件，提供了一套通用的 API 接口，与 OS 的其他组件隔离。其他组件通过 API 访问，而不直接访问 DSPLINK，此特性使 DSPLINK 可以方便的被移植到不同操作系统中。

LINK DRIVER：指连接驱动层，包含了基于 GPP 与 DSP 的物理连接的底层控制操作，负责 GPP 与 DSP 之间的数据传输和 DSP 的运行等操作。

PROCESSOR MANAGER：指进程管理层，它维护一个针对所有模块的 Book-Keeping 信息，通过 API 给用户提供通过 LINK DRIVER 的控制操作。

DSP/BIOS™ LINK API：指提供给 GPP 端的接口，是非常轻小型的组件，API 层可以认为是基于 PROCESSOR MANAGER 和 LINK DRIVER 之上的层。

2.2 DSP 端

DSP 端在本文特指 OMAPL138 平台的 DSP 端。

DSP/BIOS：指 DSP 端的 BIOS 轻量级操作系统，同时它是一个实时操作系统。本文提到的 DSPlink 在 ARM 端是基于 Linux 系统，而在 DSP 端时基于 DSP/BIOS™。也就是说，如果使用 DSPlink 调用 DSP，那么编写 DSP 程序时必须要选择操作 DSP/BIOS™支持。

LINK DRIVER：LINK DRIVER 是 DSP/BIOS 中驱动的一部分，该部分驱动只负责基于物理连接之上与 GPP 之间的交互。DSP 端没有 DSPLINK API，和 GPP 之间的通信是基于 DSP/BIOS™中的 SIO、GIO、MSGQ 模块来实现的。

OTHER DRIVERS：指 DSP 端其他驱动程序。

2.3 DSPLINK 关键组件

GPP 端的 DSPLINK 关键组件有 PROC、CHNL、MSGQ、POOL、NOTIFY、MPCS、MPLIST、RINGIO。

2.3.1 PROC

PROC 表述应用空间的 DSP 处理器，提供以下服务：

- 初始化 DSP，使之能从 GPP 端进行访问
- 从 GPP 端加载 DSP 的代码至 DSP 处理器
- 从 DSP 代码制定的地址运行 DSP 程序
- 读写 DSP 端的地址空间
- 停止 DSP 端程序的运行
- 其他一些平台专有的控制操作

2.3.2 CHNL

CHNL 是 channel 的缩写，该组件表述应用空间的一个逻辑数据传输通道，负责 GPP 与 DSP 之间的数据传输。通道 channel 的概念如下述：

- 一种 GPP 端与 DSP 端传输数据的方式
- 一个基于 GPP 与 DSP 的物理连接上的逻辑实体映射
- 唯一的数字标识，标记连接到 DSP 的一组 channel 中的某个 channel
- 无方向性，通道方向可以配置

DSP 端与 GPP 端的一条物理连接被多通道复用，被传输的数据中不包含目的地和源地址的任何信息，数据发送与接收端的数据通路需要在应用前进行显式的建立，采用 issue-reclaim 模式进行数据传输。

2.3.3 MSGQ

MSGQ 是 message queue 的缩写，表述基于 message 的消息队列，负责 GPP 与 DSP 端的可变长度的短消息交互，基于 DSP/BIOS™ 的 MSGQ 模块实现。

message 的发送接收都通过消息队列实现，消息接收者从消息队列接收信息，而消息发送者将数据写入到消息队列中，一个消息队列只可以有一个接收者，但可以有多个发送者。一个任务可以读写多个消息队列。

2.3.4 POOL

此模块提供了 API 用于配置共享内存区域，同时还提供两个 CPU 间的缓存数据同步的 API 接口。此模块提供如下功能：

- 通过调用打开（open）和关闭（close）配置共享内存区域
- 在共享内存区域内分配或释放缓存
- 分配的内存地址可以在不同的地址空间内转换
- 在不同 CPU 核之间实现内存数据的同步

2.3.5 NOTIFY

此组件允许应用程序为发生在远程处理器上的事件通知（Notification ）注册，并发送事件通知给远程的处理器。允许应用程序为远程处理器上的事件注册一个带事件回调函数；使能应用程序发送事件通知到远程处理器；同时，应用程序也可以发送一个事件处理的选项值。

Notify 组件为事件通知定义了优先级，优先级通过事件编号来实现，低编号的事件享有更高的优先级。如果事件通知不再需要使用，应用程序也可以实时注销其相应的回调函数。

2.3.6 MPCS

应用程序 MPCS 实现 GPP 和 DSP（Multi-Processor Critical Section）互斥访问共享的数据结构。应用程序有时候需要定义属于自己的，并能够被多个处理器访问的数据结构，用于多个处理器之间信息的通信。但是，应用程序必须保证多个处理器，或者每个处理上的

各个任务之间互斥地访问这些数据结构，从来保证数据的连贯性。MPCS 被用来实现这个功能。

在拥有可以共同访问的内存区域的多处理器系统中，可以实现 GPP 和 DPS 之间的 MPCS 。而为了防止没有共享存储区域的情况出现，该模块内部实现了 MPCS 组件要求的带保护的同步。

MPCS 组件提供了 APIs 接口来创建和删除 MPCS 实体，每个 MPCS 实体都通过一个系统唯一的字符串名字来标识。每一个需要使用 MPCS 的客户端都需要调用 API 打开函数来获取句柄。当不再需要使用 MPCS 时，通过相应的 API 函数来关闭句柄，同时也提供了进入和离开 MPCS 对象句柄指定的临界区域的 API 函数。

如果 MPCS 对象要求的存储空间由用户提供的话，它必须位于所有处理器都可以访问

到的池（Pool）中；如果在创建对象的时候不提供存储空间，则指定池的 ID 号将被 MPCS 对象用来内部分配空间。

2.3.7 MPLIST

此组件提供 GPP 和 DSP 之间传输机制的双重循环连接列表。在 GPP 和 DSP 之间存在共享存储空间的设备上，该模块实现共享存储空间的连接列表。对于不存在共享存储空间的设备，该模块内部保持和远程处理器上连接列表的一致。

该组件提供了创建和删除 MPLIST 实体的 APIs 函数，每个 MPLIST 实体都通过一个系统唯一的字符串名字来标识。每一个需要使用 MPLIST 的客户端都需要调用 API 打开函数来获取句柄。当不再需要使用 MPLIST 时，通过相应的 API 函数来关闭句柄。

MPLIST 组件提供的 API 函数，可以在列表最后加入一个新的元素，或者删除列表最前面的一个元素。也允许应用程序在某个已有的列表元素前插入一个缓冲，删除列表中任意一

个指定的元素，并提供了检查列表是否为空的 API 函数。另外，还有 API 函数通过获取列表首元素的指针和指定元素后的一个元素，横断列表。

2.3.8 RING IO

该组件提供基于数据流的循环缓冲区。该组件允许在共享存储空间创建循环缓冲区,不同的处理都能够读取或者写入循环缓冲区。RINGIO 组件允许通过写指针来获取数据缓冲区的空存储空间，当该存储空间被释放之后，相应存储空间可以被再次写入。

RINGIO 组件允许读指针获取缓冲区中读取空间的有效数据。当被释放之后，相应存储空间的数据被标记为无效。每个 RINGIO 实体拥有一个读指针和一个写指针。RINGIO 组件也有 API 函数，可以使能数据属性的同步传输。如：EOS(End Of Stream)、事件戳、流偏移地址等，也可能伴随着循环缓冲区的偏移值。

3 DSPLINK 配置

将光盘中的 tools 目录下的以下 4 个文件拷贝到虚拟机共享目录。

备注：此版本 DSPLINK 仅适用于 linux-2.6.33 内核。即需用将光盘中 2.6.33 版本的内核启动开发板。

（1）	bios_5_41_10_36.tar.gz	//DSP/BIOS 源码
（2）	dsplink_linux_1_65_00_03.tar.gz	//dsplink 源码
（3）	ti_cgt_c6000_7.3.0.tar.gz	//dsp 编译工具链
（4）	xdctools_3_22_01_21.tar.gz	//实时调试工具

执行以下命令分别解压以上四个文件到/home/tl/omapl138 目录：

```
tl@tl-desktop:~$ cd /home/tl/omapl138/

tl@tl-desktop:~/omapl138$ tar zxvf /mnt/hgfs/shareVM/bios_5_41_10_36.tar.gz -C ./
```

```
tl@tl-desktop:~/omapl138$ tar zxvf
/mnt/hgfs/shareVM/dsplink_linux_1_65_00_03.tar.gz -C ./

tl@tl-desktop:~/omapl138$ tar zxvf /mnt/hgfs/shareVM/ti_cgt_c6000_7.3.0.tar.gz
-C ./

tl@tl-desktop:~/omapl138$ tar zxvf /mnt/hgfs/shareVM/xdctools_3_22_01_21.tar.gz
-C ./
```

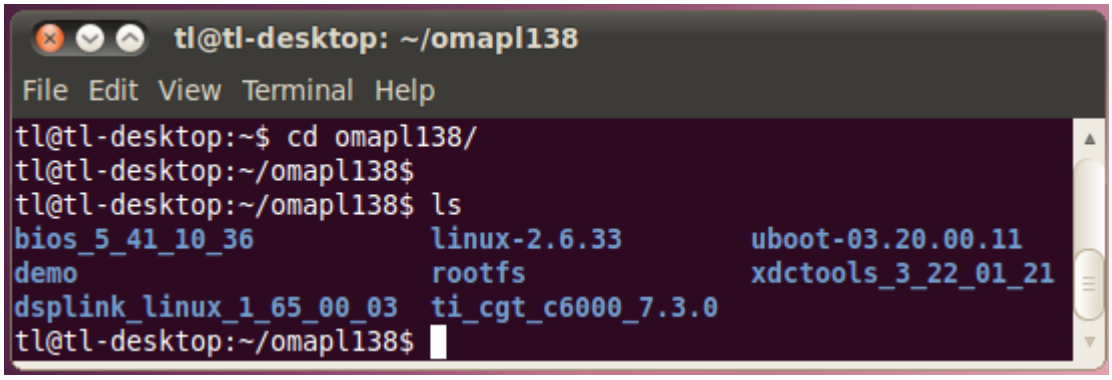
解压完成后，当前目录下就有了以下 4 个目录：

（1） bios_5_41_10_36

（2） dsplink_linux_1_65_00_03

（3） ti_cgt_c6000_7.3.0

（4） xdctools_3_22_01_21

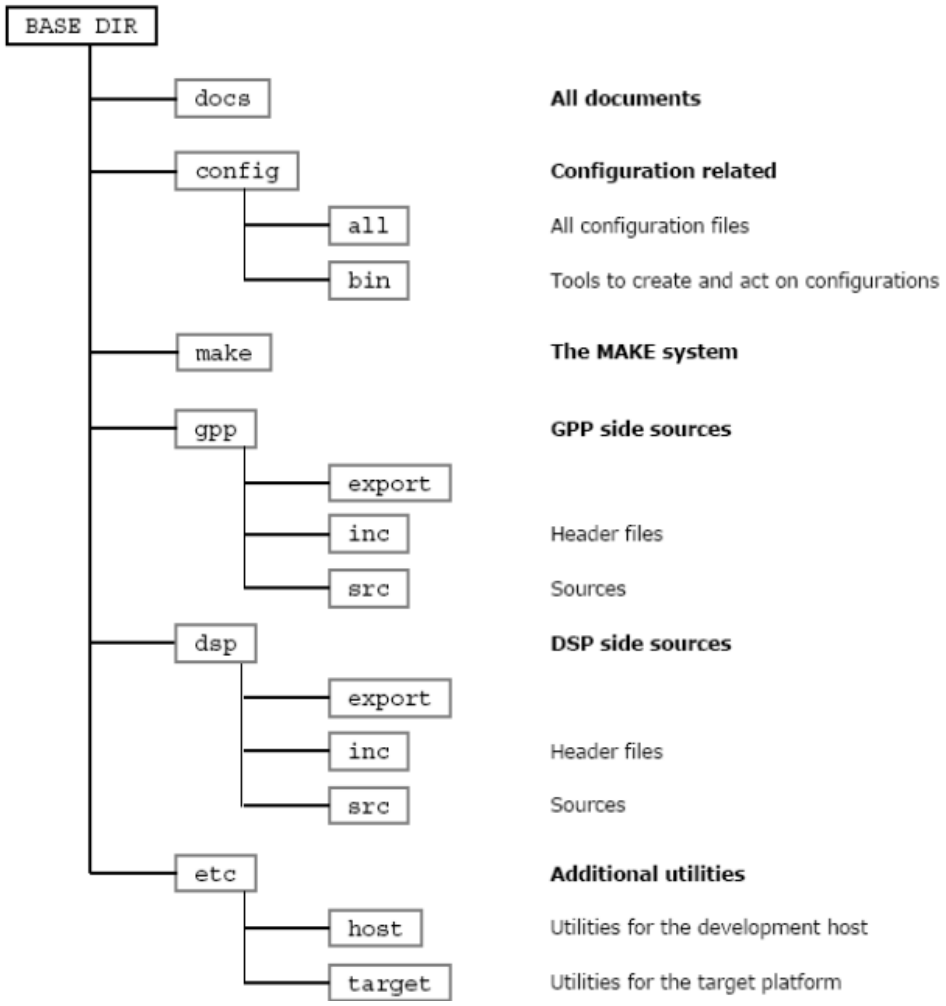


3.1 DSPLINK 源码

进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink 目录，可以看到 DSPLINK 源码的目录如下图：

```
tl@tl-desktop: ~/omapl138/dsplink_linux_1_65_00_03/dsplink
File  Edit  View  Terminal  Help
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink$ pwd
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink$ ls
config  doc  dsp  etc  gpp  make
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink$
```

展开其各个目录，其源码框架如下图：



3.1.1 GPP 端源码

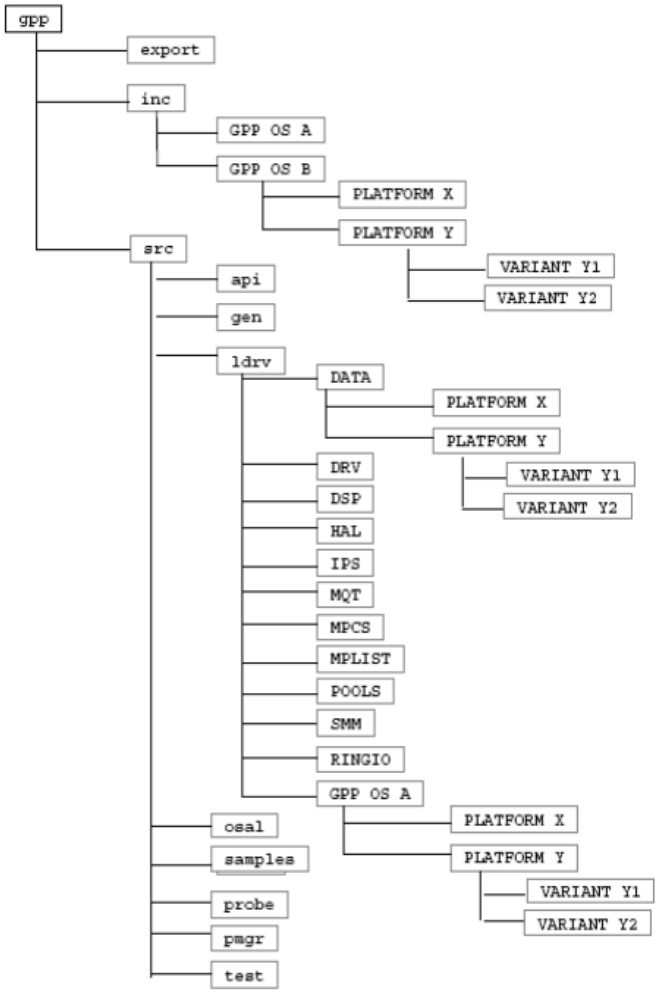
进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp 目录，可以看到 gpp 端的源码目录如下图：

```
tl@tl-desktop: ~/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp
File  Edit  View  Terminal  Help
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp$ pwd
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp$ ls
config.bld  Global.xs  package.bld  package.xs
export      inc        package.xdc  src
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp$
```

inc：相关的头文件。

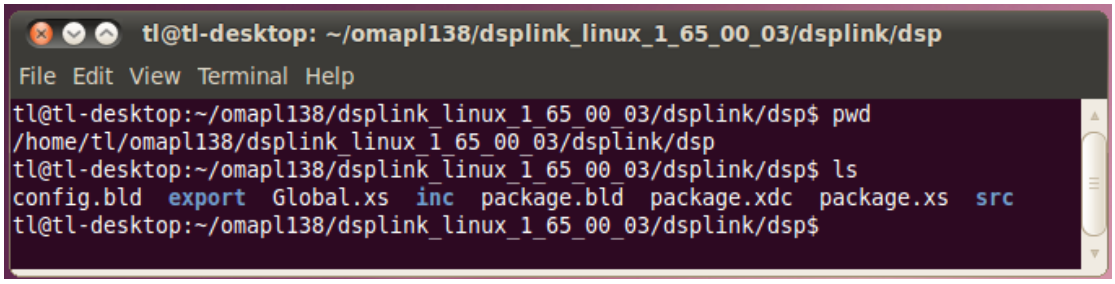
src：dsplink 源码和 sample 例程源码，用户可以编译和测试。

展开其各个目录，其源码框架如下图：



3.1.2 DSP 端源码

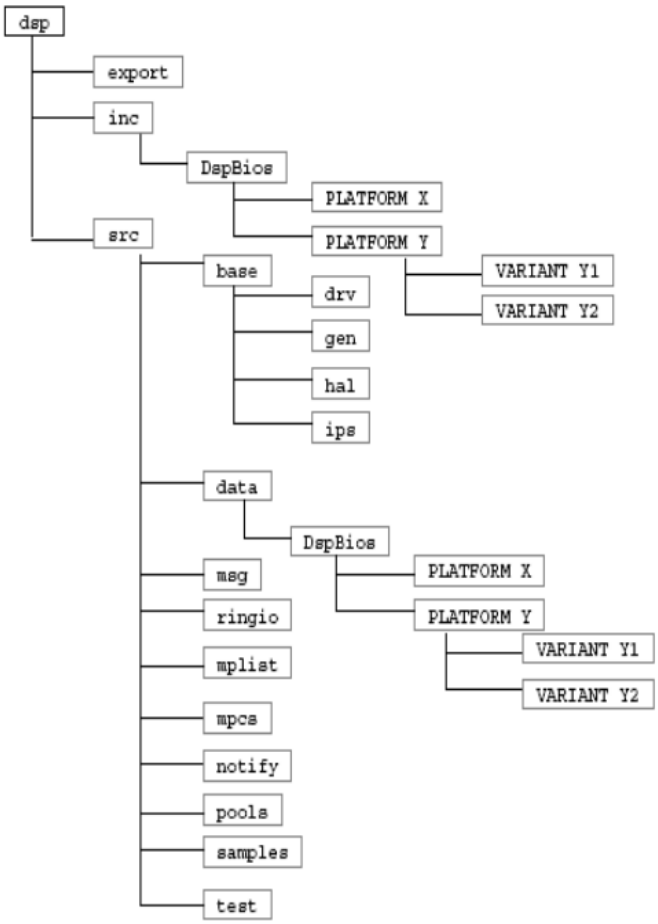
进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp 目录，可以看到 dsp 端的源码目录如下图：



inc：相关的头文件。

src：dsplink 源码和 sample 例程源码，用户可以编译和测试。

展开其各个目录，其源码框架如下图：



3.2 DSPLINK 配置

注意：本小结所有操作步骤务必在同一个窗口，否则会无法编译。

（1）修改环境变量“DSPLINK”

执行以下命令进入 dsplink 环境变量配置文件所在目录：

```
tl@tl-desktop:~$ cd
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/etc/host/scripts/Linux/
```

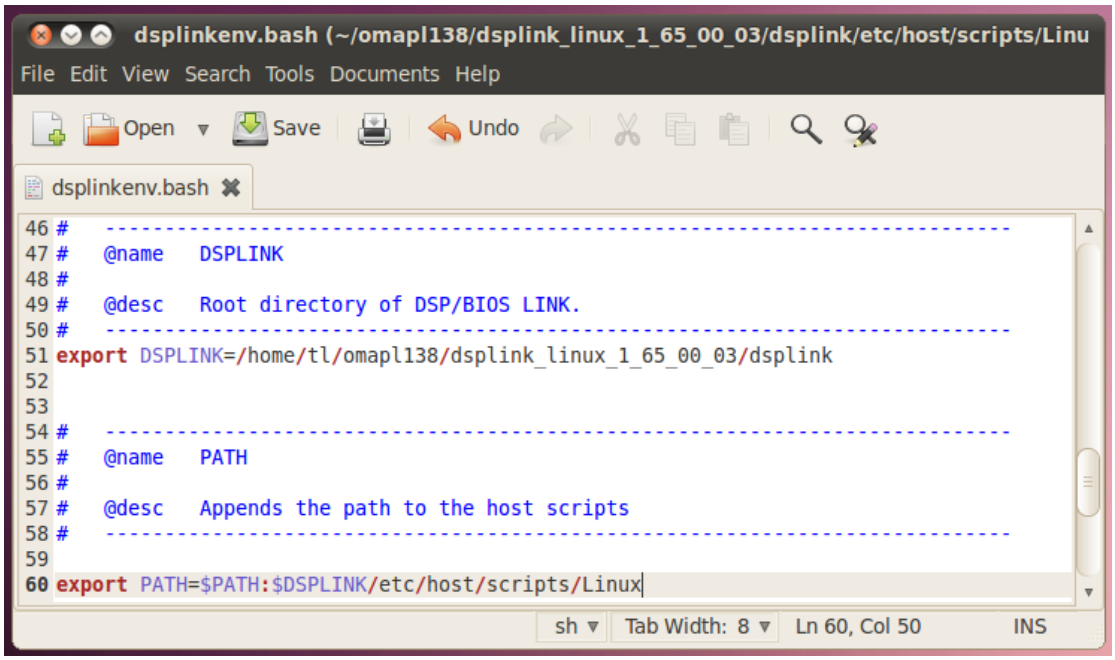
在当前目录运行命令“gedit dsplinkenv.bash”打开配置文件 dsplinkenv.bash。

将文件中的第 51 行：

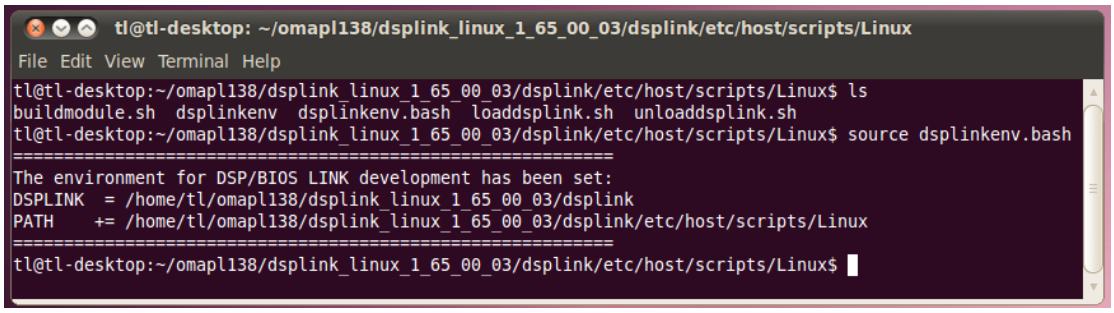
“export DSPLINK=$HOME/dsplink_1_65_00_03/dsplink”改为

“export DSPLINK=/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink”

修改如下图：



保存退出文件编辑，在当前目录执行命令“source dsplinkenv.bash”会提示环境变量“DSPLINK”、“PATH”的值，如下图所示：



（2）配置编译参数

进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/config/bin 目录，查看到当前目录下有 dsplinkcfg.pl 编译配置文件，在当前目录下执行以下命令：

tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/config/bin$ perl dsplinkcfg.pl --platform=OMAPL138 --nodsp=1 --dspcfg_0=OMAPL138GEMSHMEM --dspos_0=DSPBIOS5XX --gppos=ARM --comps=ponslrmc

如下图：

以下为各个参数的解释：

perl：使用 perl 工具配置参数

--platform=OMAPL138：编译的平台是 OMAPL138

--nodsp=1：只有一个 dsp（dsplink 支持一个 arm 跟多个 dsp 通信）

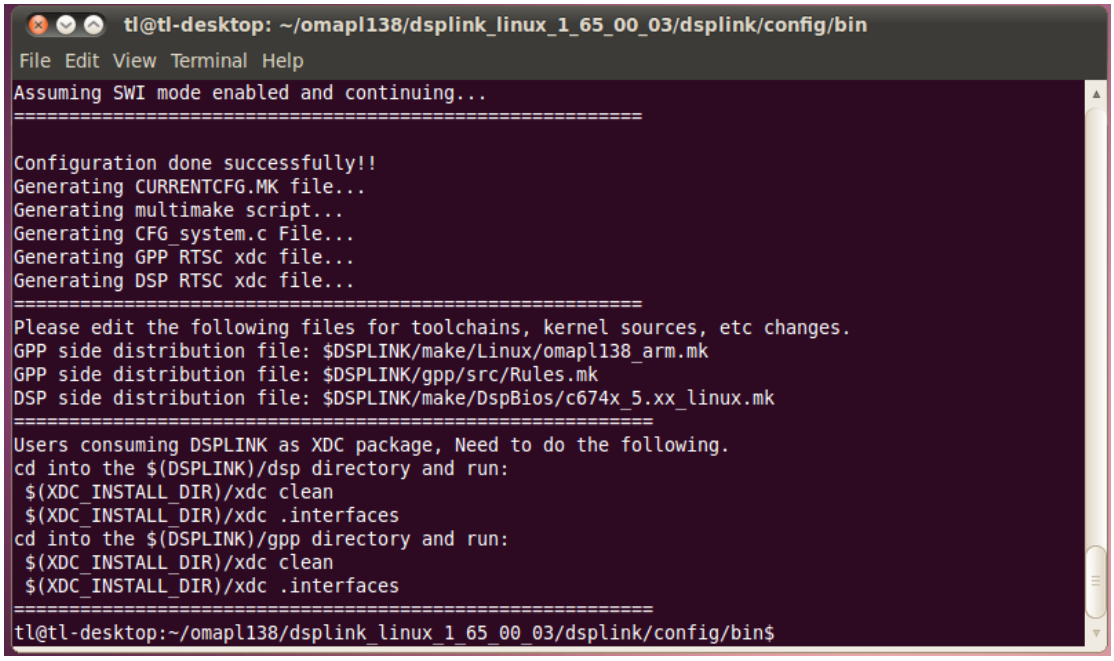
--dspcfg_0=OMAPL138GEMSHMEM：使用 OMAPL138GEMSHMEM 共享内存

--dspos_0=DSPBIOS5XX：dsp 端使用的是 bios5

--gppos=ARM：gpp 端是 ARM 平台

--comps=ponslrmc：编译所有组件

配置完成后提示如下图：



若出现如下图的错误，请执行命令如下：

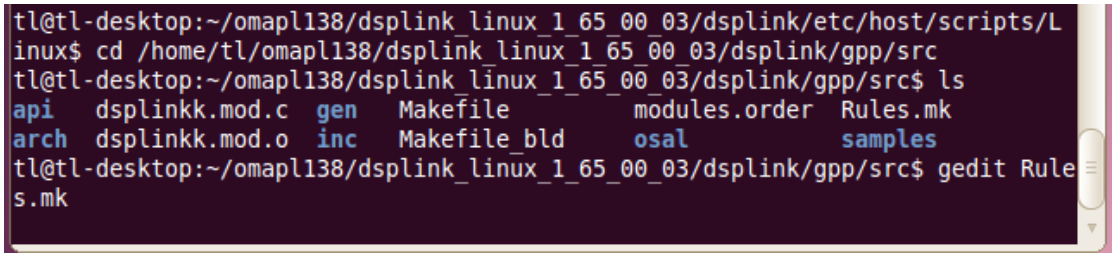
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/config/bin$ source /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/etc/host/scripts/Linux/dspli

nkenv.bash

错误的原因在于通过“source dsplinkenv.bash”设置环境变量仅在当前窗口有效。在另外一个窗口操作时需要重新 source。

（3）修改编译时 GPPOS 的路径和编译器路径

进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp/src 目录，打开配置文件 Rules.mk。



将第 56~59 行的内容：

```
ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAPL138")

KERNEL_DIR     :=
${HOME}/DaVinci-PSP-SDK-03.20.00.11/src/kernel/linux-03.20.00.11

TOOL_PATH      := ${HOME}/toolchains/git/arm-2009q1-203/bin
```

```
endif #ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAPL138")
```

修改为以下内容：

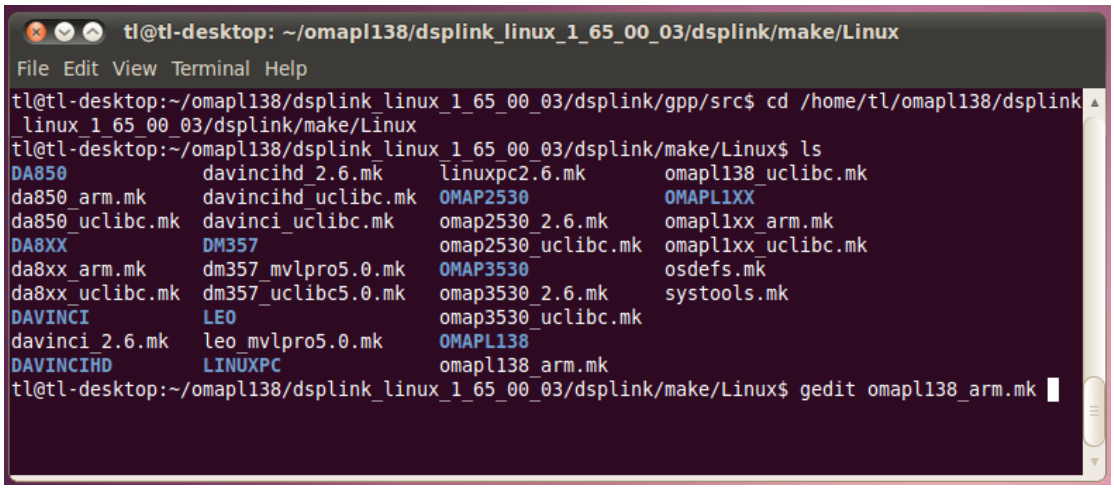
```
ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAPL138")

KERNEL_DIR     := /home/tl/omapl138/linux-2.6.33

TOOL_PATH      := /home/tl/arm-2009q1/bin

endif #ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAPL138")
```

以上修改务必对应内核源码和交叉编译工具链源码目录所在路径，修改结果如下图：

```
56 ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAPL138")
57 KERNEL_DIR     := /home/tl/omapl138/linux-2.6.33
58 TOOL_PATH      := /home/tl/arm-2009q1/bin
59 endif #ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAPL138")
```

（4）修改 gpp 端配置文件

进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/make/Linux 目录，打开当前目录下的 gpp 端配置文件 omapl138_arm.mk。



配置以下参数：

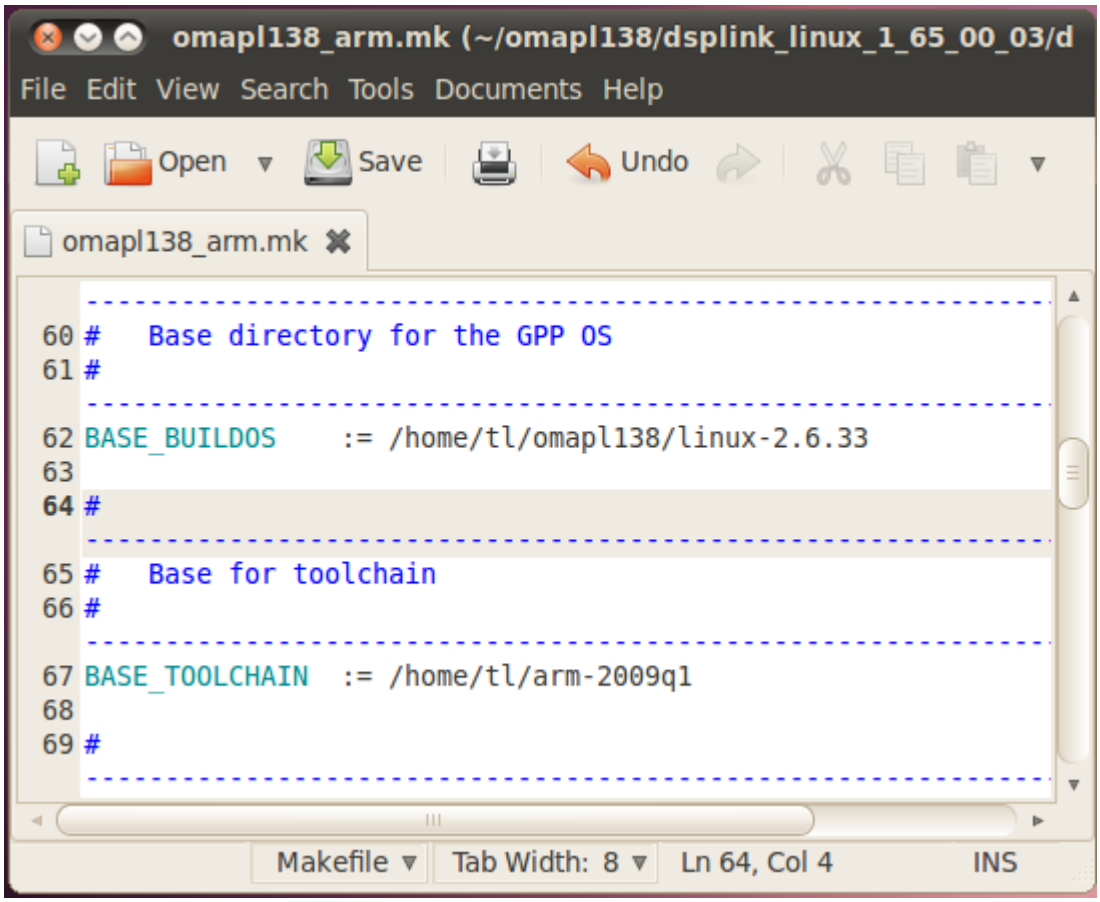
BASE_BUILDOS := /home/tl/omapl138/linux-2.6.33

（内核存放在位置，根据内核源码存放的目录而改动）

BASE-TOOLCHAIN := /home/tl/arm-2009q1

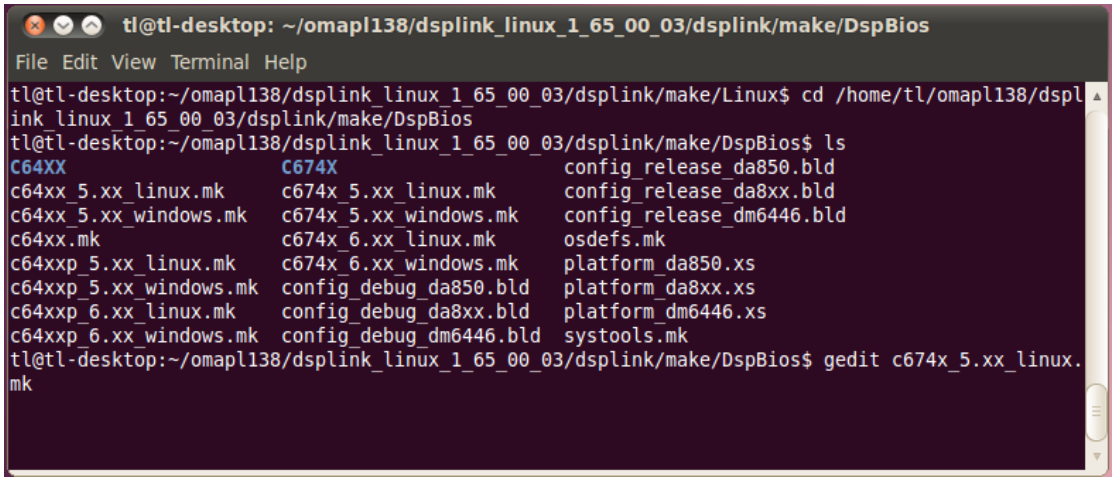
（交叉编译工具链存放在位置，根据工具链存放的目录而改动）

以上修改务必对应内核源码和交叉编译工具链源码目录所在路径，修改如下图：



（5）修改 DSP 端配置文件

进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/make/DspBios 目录，打开当前目录下的配置文件 c674x_5.xx_linux.mk。



修改 63~84 行，配置以下参数：

```
BASE_INSTALL          := /home/tl/omapl138

BASE_SABIOS           := /home/tl/omapl138/bios_5_41_10_36

XDCTOOLS_DIR          := /home/tl/omapl138/xdctools_3_22_01_21

BASE_CGTOOLS      := /home/tl/omapl138/ti_cgt_c6000_7.3.0
```

以上修改务必对应 bios_5_41_10_36、xdctools_3_22_01_2、ti_cgt_c6000_7.3.0 目录所在路径，修改如下图所示：

```
#   ----------------------------------------------------------------------------
#   Base directory for the DSP OS
#   ----------------------------------------------------------------------------
BASE_INSTALL    := /home/tl/omapl138
ifeq ($(BASE_INSTALL), "")
BASE_INSTALL    := /opt/ti-tools
endif
BASE_SABIOS     := /home/tl/omapl138/bios_5_41_10_36
ifeq ($(BASE_SABIOS), "")
BASE_SABIOS     := $(BASE_INSTALL)/bios
endif
BASE_BUILDOS    := $(BASE_SABIOS)/packages/ti/bios

#   ----------------------------------------------------------------------------
#   Base directory for the XDC tools
#   ----------------------------------------------------------------------------
XDCTOOLS_DIR    := /home/tl/omapl138/xdctools_3_22_01_21
ifeq ($(XDCTOOLS_DIR), "")
XDCTOOLS_DIR    := $(BASE_SABIOS)/xdctools
endif

#   ----------------------------------------------------------------------------
#   Base for code generation tools - compiler, linker, archiver etc.
#   ----------------------------------------------------------------------------
BASE_CGTOOLS    := /home/tl/omapl138/ti_cgt_c6000_7.3.0
ifeq ($(BASE_CGTOOLS), "")
BASE_CGTOOLS    := $(BASE_INSTALL)/c6000/cgtools
endif
BASE_CGTOOLSBIN := $(BASE_CGTOOLS)/bin

```

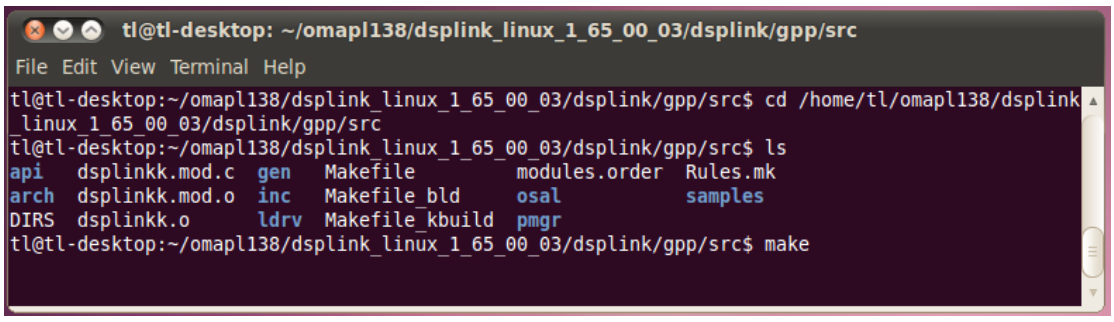
4 DSPLINK 编译

注意：本小结所有步骤应在上小结的终端窗口下继续操作，否则会出现编译错误现象。

4.1 GPP 端 DSPLINK 编译

4.1.1 DSPLINK 源码编译

切换到/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp/src 目录，这是 GPP 端源码所在目录，并运行“make”命令进行编译，操作如下图：



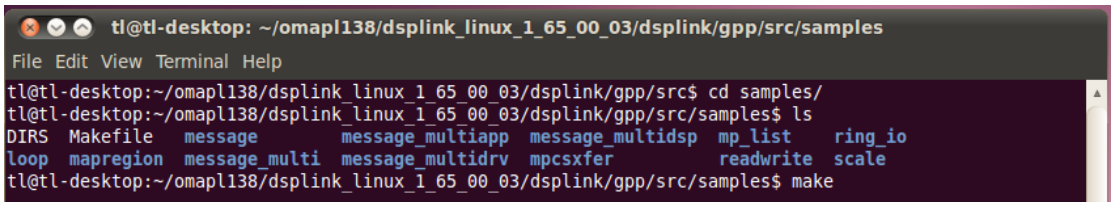
编译完成如下图所示：

编译完成后，将会在以下目录产生 gpp 端的驱动程序镜像 dsplinkk.ko 文件：

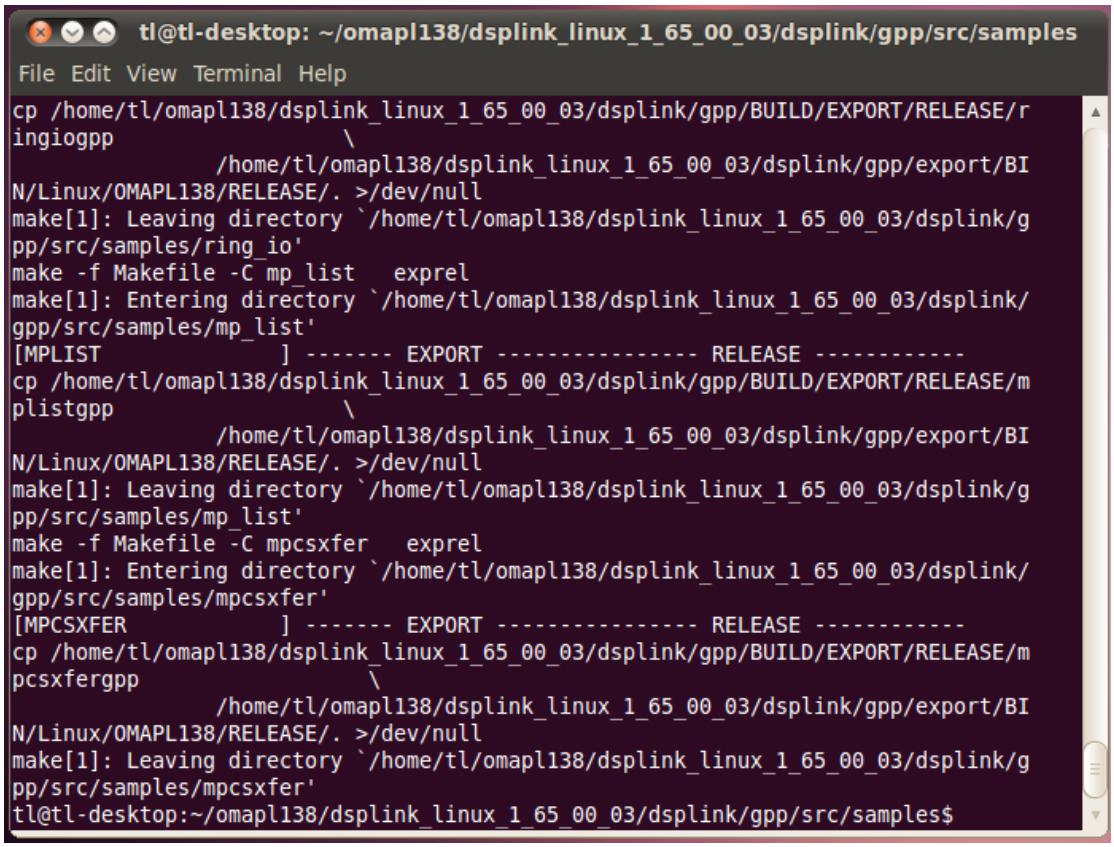
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp/export/BIN/Linux/OMAPL138/RELEASE

4.1.2 示例程序编译

本小节编译出来的示例程序是在 gpp 端运行的，首先切换到当前目录下的 samples 示例代码目录，再在运行“make”命令编译示例程序源码，操作如下图：

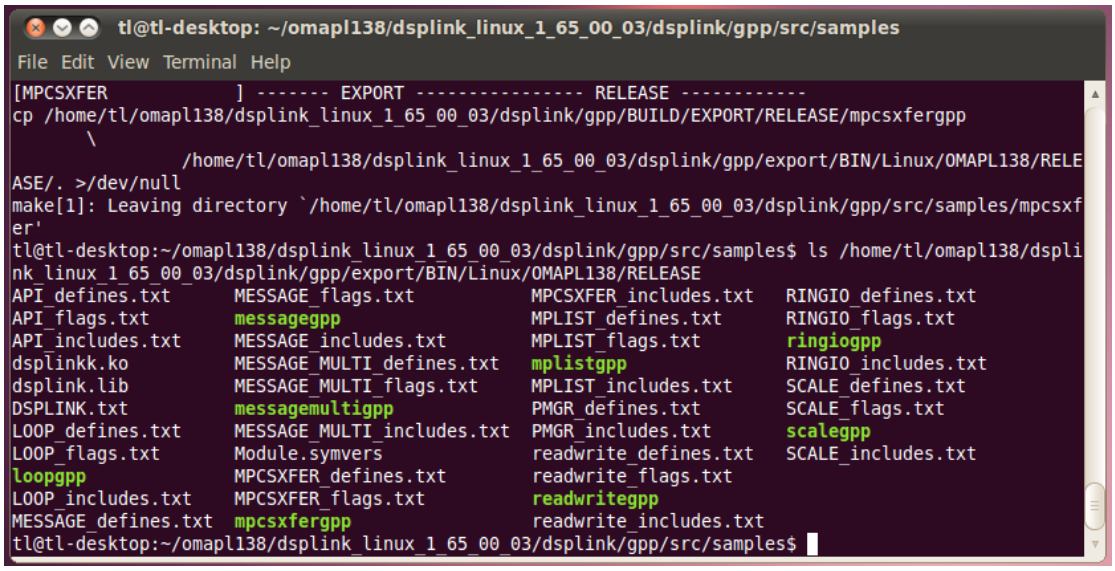


编译成功如下图所示：



编译出来的 gpp 端可执行文件位于以下目录：

/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp/export/BIN/Linux/OMAPL138/RELEASE，使用“ls”命令查看，如下图所示，8 个绿色图标*gpp 文件就是 gpp 端的可执行文件：



4.2 DSP 端 DSPLINK 编译

4.2.1 DSPLINK 源码编译

进入/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/目录，这是 DSP 端库文件源码所在目录，并运行“make”命令进行编译，操作如下图：

编译成功后如下图所示：

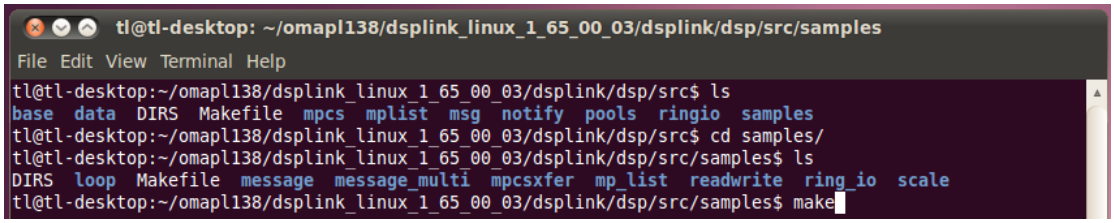
```
tl@tl-desktop: ~/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src
File  Edit  View  Terminal  Help
ata.lib                  \
               /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OM
APL138GEM_0/RELEASE/. >/dev/null
make[1]: Leaving directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/data'
make -f Makefile -C notify   exprel
make[1]: Entering directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/notify'
[NOTIFY              ] ------- EXPORT ---------------- RELEASE ------------
cp /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/BUILD/OMAPL138GEM_0/EXPORT/RELEASE/dsplinkn
otify.lib                \
               /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OM
APL138GEM_0/RELEASE/. >/dev/null
make[1]: Leaving directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/notify'
make -f Makefile -C ringio   exprel
make[1]: Entering directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/ringio'
[RINGIO              ] ------- EXPORT ---------------- RELEASE ------------
cp /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/BUILD/OMAPL138GEM_0/EXPORT/RELEASE/dsplinkr
ingio.lib                \
               /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OM
APL138GEM_0/RELEASE/. >/dev/null
make[1]: Leaving directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/ringio'
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src$
```

编译出来的库文件位于以下目录：/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OMAPL138GEM_0/RELEASE，使用“ls”命令查看，如下图所示，以.lib 结尾的文件就是库文件：

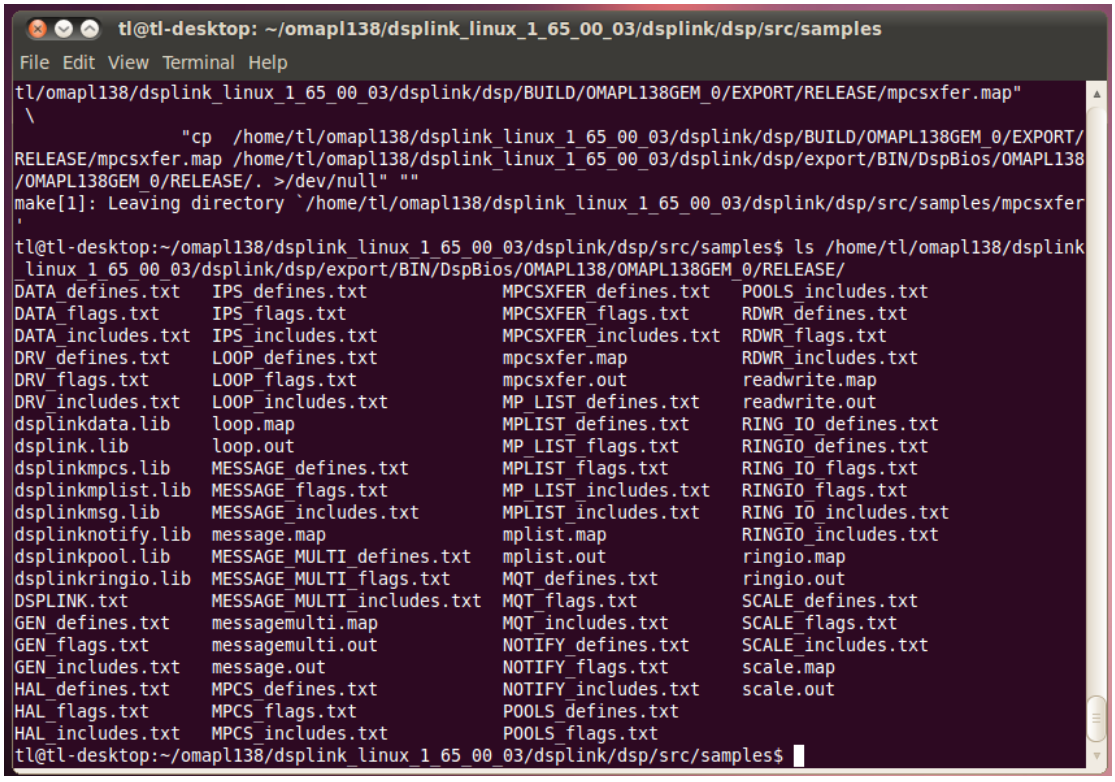
```
tl@tl-desktop: ~/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src
File  Edit  View  Terminal  Help
               /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OMAP
L138GEM_0/RELEASE/. >/dev/null
make[1]: Leaving directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/notify'
make -f Makefile -C ringio   exprel
make[1]: Entering directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/ringio'
[RINGIO              ] ------- EXPORT ---------------- RELEASE ------------
cp /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/BUILD/OMAPL138GEM_0/EXPORT/RELEASE/dsplinkrin
gio.lib                  \
               /home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OMAP
L138GEM_0/RELEASE/. >/dev/null
make[1]: Leaving directory `/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src/ringio'
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src$ ls /home/tl/omapl138/dsplink_linux_1
_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OMAPL138GEM_0/RELEASE
DATA_defines.txt   dsplinkmplist.lib  HAL_defines.txt    MPLIST_defines.txt   POOLS_defines.txt
DATA_flags.txt     dsplinkmsg.lib     HAL_flags.txt      MPLIST_flags.txt     POOLS_flags.txt
DATA_includes.txt  dsplinknotify.lib  HAL_includes.txt   MPLIST_includes.txt  POOLS_includes.txt
DRV_defines.txt    dsplinkpool.lib    IPS_defines.txt    MQT_defines.txt      RINGIO_defines.txt
DRV_flags.txt      dsplinkringio.lib  IPS_flags.txt      MQT_flags.txt        RINGIO_flags.txt
DRV_includes.txt   DSPLINK.txt        IPS_includes.txt   MQT_includes.txt     RINGIO_includes.txt
dsplinkdata.lib    GEN_defines.txt    MPCS_defines.txt   NOTIFY_defines.txt
dsplink.lib        GEN_flags.txt      MPCS_flags.txt     NOTIFY_flags.txt
dsplinkmpcs.lib    GEN_includes.txt   MPCS_includes.txt  NOTIFY_includes.txt
tl@tl-desktop:~/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/src$
```

4.2.2 示例程序编译

首先切换到当前目录下的 samples 示例代码目录，再在运行“make”命令编译示例程序源码，操作如下图：



编译出来的 dsp 端可执行文件位于以下目录：/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OMAPL138GEM_0/RELEASE/，使用“ls”命令查看，如下图所示， 以*.out 结尾的文件就是 dsp 端的可执行文件：



5 DSPLINK 例程演示

5.1 演示程序准备

程序演示需要将编译完成的 GPP 端和 DSP 端的程序和数据文件复制到开发板的文件系统。

（1） 在虚拟机的建立存放 GPP 端和 DSP 端的程序的文件夹 dsplink

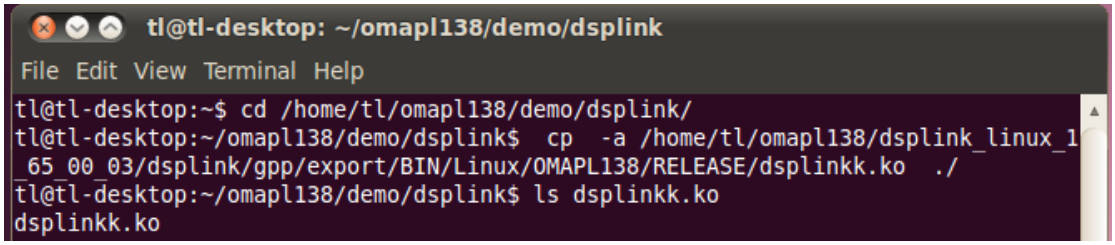
```
tl@tl-desktop:~$ mkdir  -p /home/tl/omapl138/demo/dsplink
```

（2） 将 dsplink 相关程序放到新建立的 dsplink 文件夹

拷贝 dsplinkk.ko:

```
tl@tl-desktop:~$ cd  /home/tl/omapl138/demo/dsplink/
```

```
tl@tl-desktop:~$ cp  -a
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp/export/BIN/Linux/OMAPL13
8/RELEASE/dsplinkk.ko  ./
```



拷贝所有以 gpp 后缀结尾的 gpp 端可执行程序：

```
tl@tl-desktop:~/omapl138/demo/dsplink$ cp  -a
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/gpp/export/BIN/Linux/OMAPL13
8/RELEASE/*gpp  ./
```

```
tl@tl-desktop:~/omapl138/demo/dsplink$ cp -a /home/tl/omapl138/dsplink_linux_1_6
5_00_03/dsplink/gpp/export/BIN/Linux/OMAPL138/RELEASE/*gpp ./
tl@tl-desktop:~/omapl138/demo/dsplink$ ls
dsplinkk.ko  messagegpp       mpcsxfergpp  readwritegpp  scalegpp
loopgpp      messagemultigpp  mplistgpp    ringiogpp
```

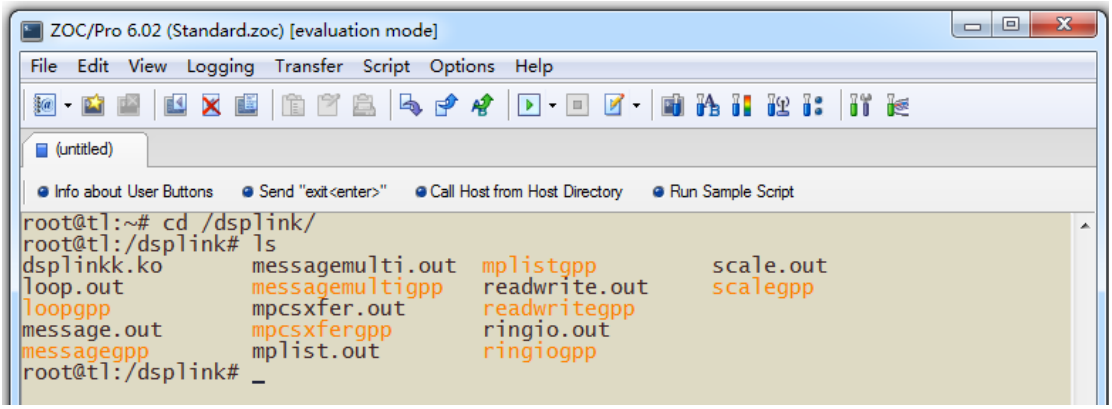
拷贝所有以.out 后缀结尾的 dsp 程序可执行程序：

```
tl@tl-desktop:~/omapl138/demo/dsplink$ cp  -a
/home/tl/omapl138/dsplink_linux_1_65_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL
138/OMAPL138GEM_0/RELEASE/*.out  ./
```

```
tl@tl-desktop:~/omapl138/demo/dsplink$ cp -a /home/tl/omapl138/dsplink_linux_1_6
5_00_03/dsplink/dsp/export/BIN/DspBios/OMAPL138/OMAPL138GEM_0/RELEASE/*.out ./
tl@tl-desktop:~/omapl138/demo/dsplink$ ls
dsplinkk.ko  messagemultigpp   mpcsxfer.out  readwrite.out  scale.out
loopgpp      messagemulti.out  mplistgpp     ringiogpp
loop.out     message.out       mplist.out    ringio.out
messagegpp   mpcsxfergpp       readwritegpp  scalegpp
tl@tl-desktop:~/omapl138/demo/dsplink$
```

（3） 将存放 dsplink 文件夹拷贝到开发板

将 dsplink 文件夹拷贝到开发板的根目录下，完成后，进入开发板的 dsplink 目录，显示如下图：



5.2 运行 dsplink 演示程序

在运行所有演示程序之前，请先安装 dsplink 驱动程序 dsplinkk.ko。在开发板的/dsplink 目录中运行以下命令：

```
root@tl:/dsplink# insmod dsplinkk.ko
```

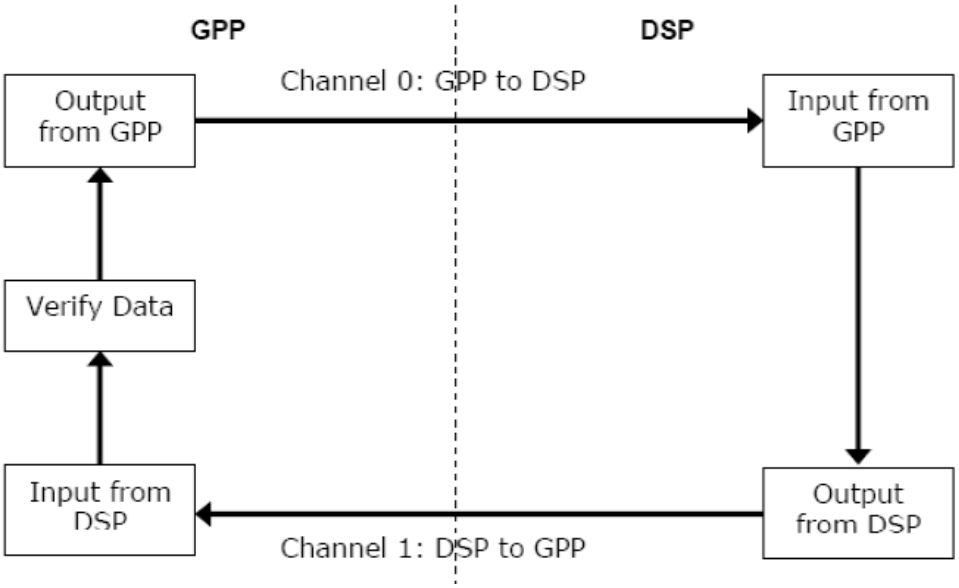
提示如下图：

```
root@tl:/dsplink# insmod dsplinkk.ko
DSPLINK Module (1.65.00.03) created on Date: Aug 30 2013 Time: 12:06:52
root@tl:/dsplink# _
```

5.2.1 LOOP

LOOP 示例阐明了在 DSP/BIOS™ LINK 中的基本数据流的概念。它实现了在 GPP 端的任务与 DSP 端任务的数据传输。DSP 端应用程序采用 SIO 和 GIO 实现了 TSK（任务）与 SWI（软中断）。

LOOP 例程中数据流向如下图：



LOOP 例程演示操作的命令格式：./loopgpp <absolute path of DSP executable> <Buffer Size> <number of transfers> < DSP Processor Id >

absolute path of DSP executable：指 DSP 端可执行文件的路径

Buffer Size：表示缓冲区大小；

number of transfers：表示执行次数，值为 0 时执行无数次；

DSP Processor Id：DSP 处理器的 Id 号，值为 0 指 DSP 0，值为 1 指 DSP 1，当只有一个 DSP 时，此处可不填。

执行“./loopgpp”可以看到例程用法和参数的英文解释（其他例程英文解释查看方法一样），如下图：

```
root@tl:/dsplink# ./loopgpp
Usage : ./loopgpp <absolute path of DSP executable> <Buffer Size> <number of trans
fers> < DSP Processor Id >
For infinite transfers, use value of 0 for <number of transfers>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0
        use value of 1  if sample needs to be run on DSP 1
        For single DSP configuration this is optional argument
root@tl:/dsplink# _
```

LOOP 例程演示命令如下：

root@tl:/dsplink# ./loopgpp loop.out 1024 1000

执行 1000 次，每次传输 1Kbyte 的数据量，运行命令后的结果如下图：

```
root@tl:/dsplink# ./loopgpp loop.out 1024 1000
=============== Sample Application : LOOP ==========
==== Executing sample for DSP processor Id 0 ====
Entered LOOP_Create ()
Leaving LOOP_Create ()
Entered LOOP_Execute ()
Transferred 1000 buffers
Leaving LOOP_Execute ()
Entered LOOP_Delete ()
Leaving LOOP_Delete ()
================================================
```

传输的内容对应在 samples/loop/loop.c 中已经定义好了，定义如下图：

```
83 #define XFER_CHAR   (Char8) 0xE7
```

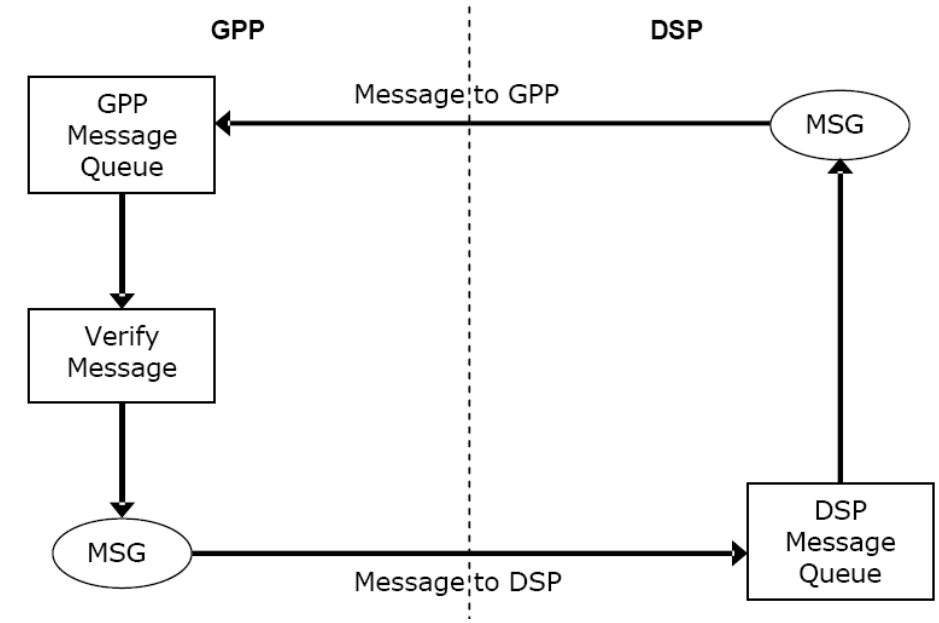
```
356     /*
357      *  Initialize the buffer with valid data.
358      */
359     if (DSP_SUCCEEDED (status)) {
360         temp = LOOP_Buffers [0] ;
361
362         for (i = 0 ; i < LOOP_BufferSize ; i++) {
363             *temp++ = XFER_CHAR ;
364         }
365     }
366
367     LOOP_0Print ("Leaving LOOP_Create ()\n") ;
368
369     return status ;
```

从上图可以看到填充的数据是 0xE7。

5.2.2 MESSAGE

MESSAGE 示例阐明了在 DSP/BIOS™ LINK 中的基本信息传递的概念。它实现了 GPP 端任务与 DSP 端任务之间的信息传递。DSP 端应用程序采用 MSGQ 实现了 TSK 与 SWI。

MESSAGE 例程中信息流向如下图：



MESSAGE 例程演示操作的命令格式如下：./messagegpp <absolute path of DSP executable> <number of transfers> <DSP Processor Id>

执行“./messagegpp”可以看到例程的用法和参数的英文解释，如下图：

```
root@tl:/dsplink# ./messagegpp
Usage : ./messagegpp <absolute path of DSP executable> <number of transfers> <DSP Processor Id>
For infinite transfers, use value of 0 for <number of transfers>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0
        use value of 1  if sample needs to be run on DSP 1
        For single DSP configuration this is optional argument
root@tl:/dsplink#
root@tl:/dsplink#
root@tl:/dsplink# _
```

MESSAGEGPP 例程演示命令如下：

root@tl:/dsplink# ./messagegpp message.out 10000

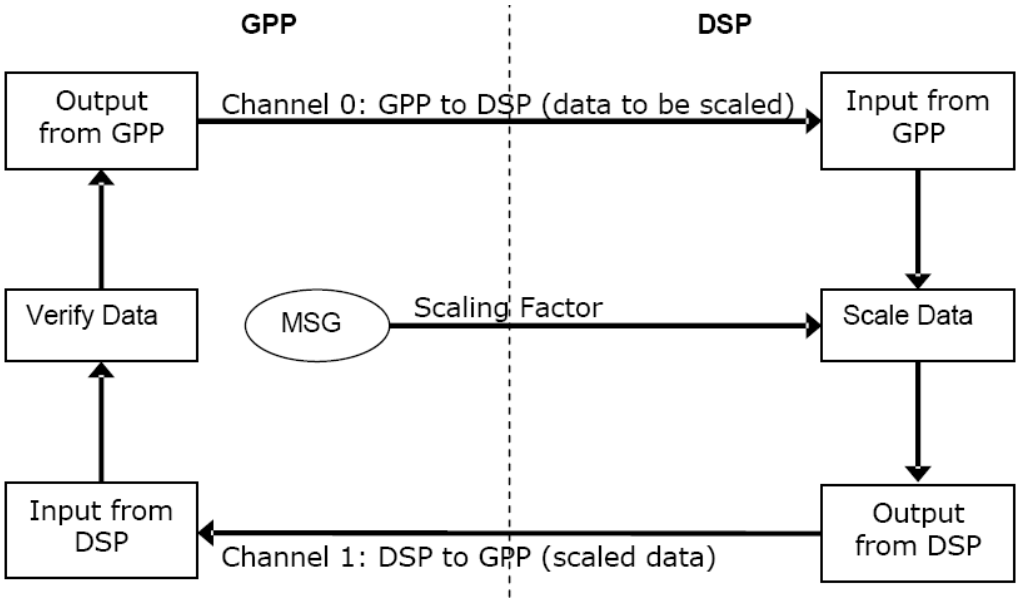
传输 10000 次，运行命令后的结果如下图：

```
root@tl:/dsplink# ./messagegpp message.out 10000
========== Sample Application : MESSAGE ==========
Entered MESSAGE_Create ()
Leaving MESSAGE_Create ()
Entered MESSAGE_Execute ()
Transferring 10000 iterations took 2 seconds 381400 microseconds.
RoundTrip Time for 1 message is 238 microseconds.
Leaving MESSAGE_Execute ()
Entered MESSAGE_Delete ()
Leaving MESSAGE_Delete ()
================================================
```

5.2.3 SCALE

SCALE 示例阐明了在 DSP/BIOS™ LINK 的数据流和信息的组合概念。它不仅实现了 GPP 端任务与 DSP 端任务之间的数据传递，而且从 GPP 端发送信息到 DSP 端。

DSP 端应用程序采用 SIO&MSGQ 和 GIO&MSGQ 实现了 TSK 与 SWI。SCALE 例程中数据与信息流向图如下：



SCALE 例程演示操作的命令格式如下：./scalegpp <absolute path of DSP executable> <Buffer Size> <number of transfers> <DSP Processor Id>

执行“./scalegpp”可以看到例程的用法和参数的英文解释，如下图：

```
root@tl:/dsplink# ./scalegpp
Usage : ./scalegpp <absolute path of DSP executable> <Buffer Size> <number of transfers> <DSP Processor Id>
For infinite transfers,use value of 0 for <number of transfers>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0

        use value of 1  if sample needs to be run on DSP 1
        For single DSP configuration this is optional argument
root@tl:/dsplink#
```

SCALE 例程演示命令如下：

```
root@tl:/dsplink# ./scalegpp scale.out 1024 100
```

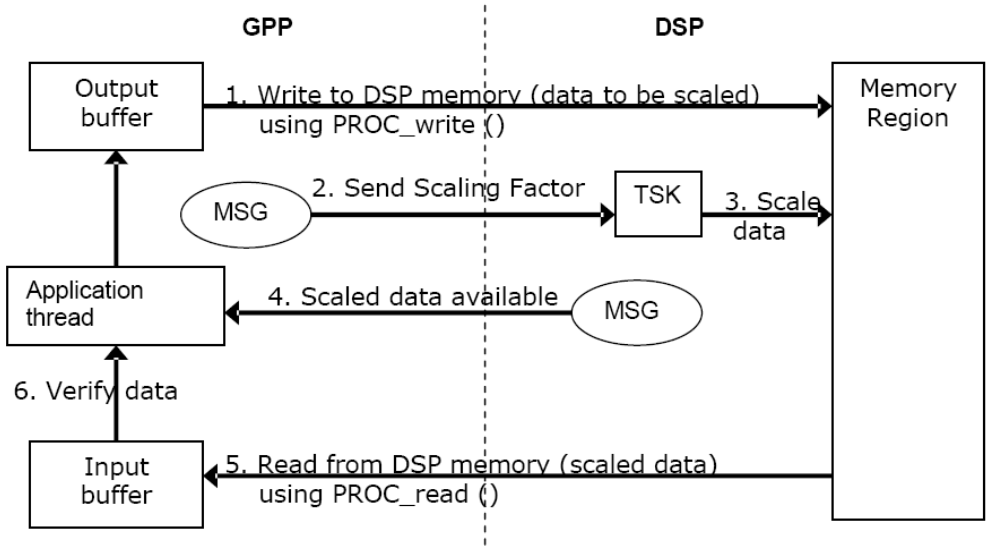
执行 100 次，每次传输 1Kbyte 的数据量，运行命令后的结果如下图：

```
root@tl:/dsplink# ./scalegpp scale.out 1024 100
=============== Sample Application : SCALE ==========
Entered SCALE_Create ()
Leaving SCALE_Create ()
Entered SCALE_Execute ()
Changed the scale factor to: 2
Changed the scale factor to: 3
Changed the scale factor to: 4
Changed the scale factor to: 5
Changed the scale factor to: 6
Changed the scale factor to: 7
Changed the scale factor to: 8
Changed the scale factor to: 9
Changed the scale factor to: 10
Transferred 100 buffers
Leaving SCALE_Execute ()
Entered SCALE_Delete ()
Leaving SCALE_Delete ()
=====================================================
```

5.2.4 READWRITE

READWRITE 示例阐明了大缓冲区通过直接读写 DSP 内部 RAM 来进行传输的概念。它实现了在 GPP 端和使用 PROC_Read()和 PROC_Write() API 的 DSP 端以及两个 DSP 端之间的大尺寸数据缓冲器之间的数据与信息的传递和转换。DSP 端应用程序采用 MSGQ 实现了 TSK。

READWRITE 例程中数据与信息流向图如下：



READWRITE 例程演示操作的命令格式如下：./readwritegpp <absolute path of DSP executable> <DSP address> <buffersize> <number of transfers><DSP ProcessorId>

执行“./readwritegpp”可以看到例程的用法和参数的英文解释，如下图：

```
root@tl:/dsplink# ./readwritegpp
Usage : ./readwritegpp <absolute path of DSP executable> <DSP address> <buffersize> <number of
transfers><DSP ProcessorId>
For infinite transfers, use value of 0 for <number of transfers>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0
        use value of 1  if sample needs to be run on DSP 1
        For single DSP configuration this is optional argument
```

READWRITE 例程演示命令如下：

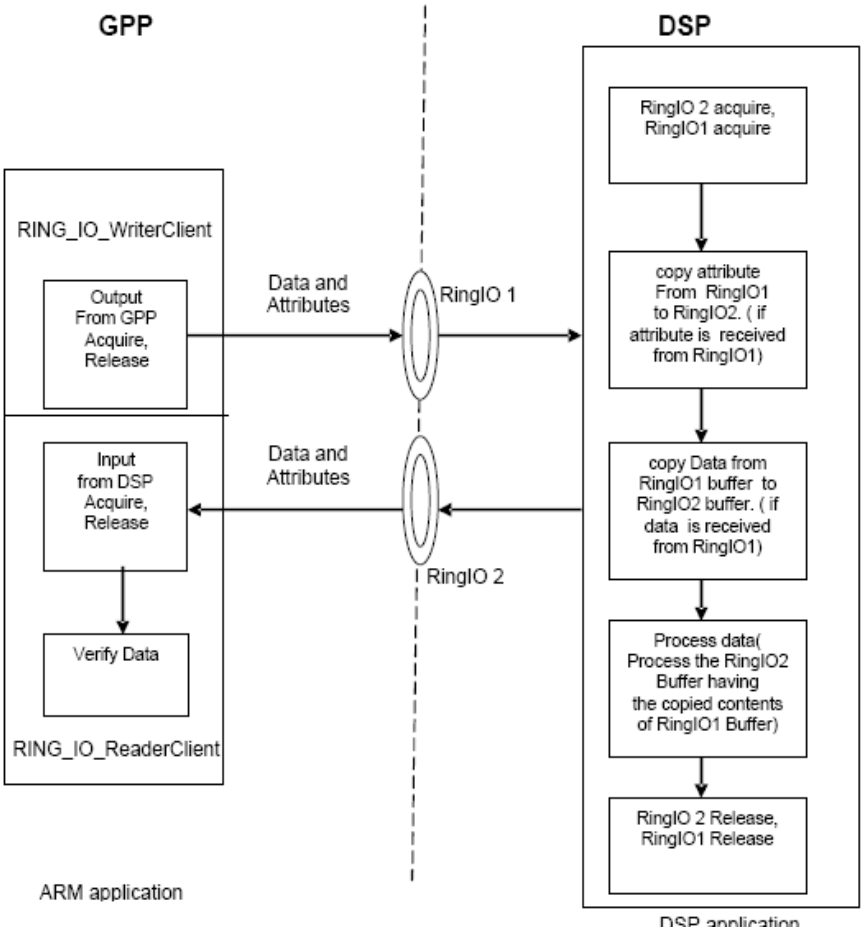
root@tl:/dsplink# ./readwritegpp readwrite.out 293601280 1024 1000

293601280 是 DSP 的内存地址，执行 1000 次，每次传输 1Kbyte 的数据量，运行命令后的结果如下图：

```
root@tl:/dsplink# ./readwritegpp readwrite.out 293601280 1024 1000
============= Sample Application : READWRITE ==========
Entered RDWR_Create ()
Leaving RDWR_Create ()
Entered RDWR_Execute ()
Verified   100 Iterations of Correct Data Read/ Write
Verified   200 Iterations of Correct Data Read/ Write
Verified   300 Iterations of Correct Data Read/ Write
Verified   400 Iterations of Correct Data Read/ Write
Verified   500 Iterations of Correct Data Read/ Write
Verified   600 Iterations of Correct Data Read/ Write
Verified   700 Iterations of Correct Data Read/ Write
Verified   800 Iterations of Correct Data Read/ Write
Verified   900 Iterations of Correct Data Read/ Write
Verified  1000 Iterations of Correct Data Read/ Write
Leaving RDWR_Execute ()
Entered RDWR_Delete ()
Leaving RDWR_Delete ()
=======================================================
```

5.2.5 RING_IO

RING_IO 示例阐明了如何使用 DSP/BIOS™ LINK 中的 RingIO 部件以及在 GPP 与使用两个 RingIO 实例的 DSP 之间的数据流的方法。它实现了数据在 GPP 端运行的线程/进程的应用程序和 DSP 端之间的传递与转换。在 Linux 中，这个应用程序在某个过程中通过进程或者线程来运行。在 PrOS 中，它通过一系列任务来运行。在随后的部分应用程序中的每个线程/进程/任务被视为客户端。在 DSP 端，此应用程序阐述了使用 RingIO 来实现 TSK 的用法。



RING_IO 例程演示操作的命令格式如下：./ringiogpp <absolute path of DSP executable> <RingIO data Buffer Size in bytes> <number of bytes to transfer><DSP Processor Id>

执行“./ringiogpp”可以看到例程的用法和参数的英文解释，如下图：

```
root@tl:/dsplink# ./ringiogpp
Usage : ./ringiogpp <absolute path of DSP executable> <RingIO data Buffer Size in byte
s> <number of bytes to transfer><DSP Processor Id>
For infinite transfers, use value of 0 for <number of bytes to transfer>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0
        use value of 1  if sample needs to be run on DSP 1
        For single DSP configuration this is optional argument
```

RING_IO 例程演示命令如下：

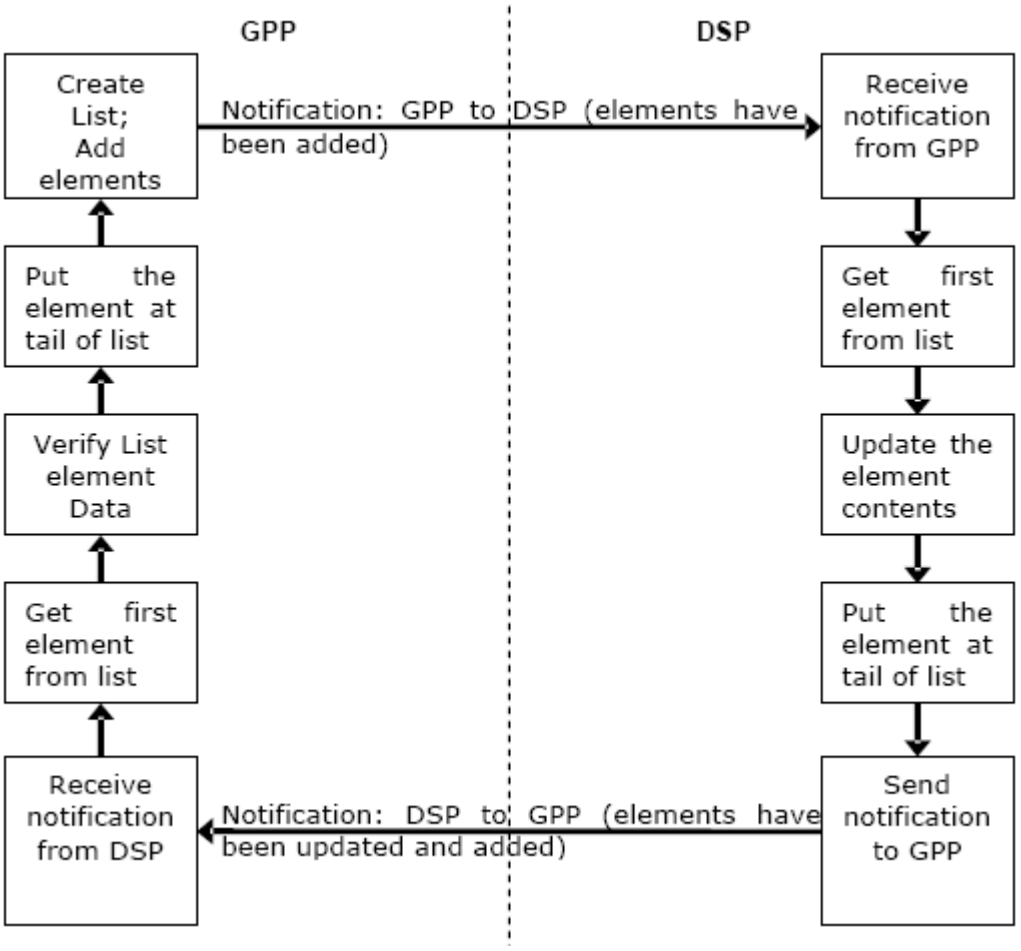
```
root@tl:/dsplink# ./ringiogpp ringio.out 10240 1000
```

执行 1000 次，数据缓冲区的大小是 10KByte，运行命令后的结果如下图：

```
root@tl:/dsplink# ./ringiogpp ringio.out 10240 1000
========== Sample Application : RING_IO ==========
Bytes to transfer :1024
Data buffer size  :10240
Entered RING_IO_Create ()
Leaving RING_IO_Create ()
Entered RING_IO_ReaderClient ()
Entered RING_IO_WriterClient ()
GPP-->DSP:Sent Data Transfer Start Attribute
GPP-->DSP:Sent Data Transfer Start Notification
GPP<--DSP:Received Data TransferStart Attribute
GPP-->DSP:Total Bytes Transmitted  1024
RingIO_setAttribute succeeded to set the  RINGIO_DATA_END. Status = [0x8100]
GPP-->DSP:Sent Data Transfer End Attribute
GPP-->DSP:Sent Data Transfer End Notification
GPP<--DSP:Received Data TransferEnd Attribute
GPP<--DSP:Bytes Received 1024
GPP<--DSP:Received Data Transfer End Notification
Leaving RING_IO_ReaderClient ()
Leaving RING_IO_WriterClient ()
Entered RING_IO_Delete ()
Leaving RING_IO_Delete ()
================================================
```

5.2.6 MP_LIST

MPLIST 示例阐明了如何使用 DSP/BIOS™ LINK 中的 MPLIST 的部件以及在 GPP 与使用多个处理器列表的 DSP 之间的数据流的方法。它实现了 GPP 端和 DSP 端之间的数据转换与传递。在 DSP 端，应用程序通过 MPLIST 实现了 TSK。



MPLIST 例程演示操作的命令格式如下：./mplistgpp <absolute path of DSP executable> <number of iterarions for which to run the sample> <number of elements to add at end of list> <DSP Processor Id>

执行“./mplistgpp”可以看到例程的用法和参数的英文解释，如下图：

```
root@tl:/dsplink# ./mplistgpp
Usage : ./mplistgpp <absolute path of DSP executable> <number of iterarions for which
to run the sample>
<number of elements to add at end of list> <DSP Processor Id>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0
        use value of 1  if sample needs to be run on DSP 1
        For single DSP configuration this is optional argument
```

MPLIST 例程演示命令如下：

```
root@tl:/dsplink# ./mplistgpp mplist.out 1000 100
```

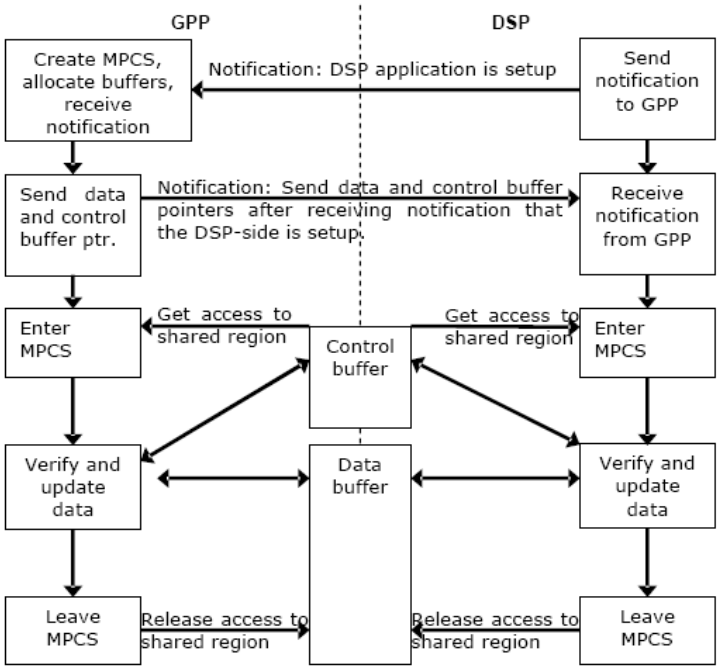
执行 1000 次，每次在队列最后添加 100 个数据，运行命令后的结果如下图：

```
root@tl:/dsplink# ./mplistgpp mplist.out 1000 100
========== Sample Application : MP_LIST ==========
Entered MP_LIST_Create ()
Leaving MP_LIST_Create ()
Entered MP_LIST_Execute ()
Verified 100 list based transfer
Verified 200 list based transfer
Verified 300 list based transfer
Verified 400 list based transfer
Verified 500 list based transfer
Verified 600 list based transfer
Verified 700 list based transfer
Verified 800 list based transfer
Verified 900 list based transfer
Verified 1000 list based transfer
Leaving MP_LIST_Execute ()
Entered MP_LIST_Delete ()
Leaving MP_LIST_Delete ()
================================================
```

5.2.7 MPCSXFER

MPCSXFER 示例阐明了通过一个拥有互斥访问保护的共享内存缓冲区的基本机制实现了数据在 GPP 端与 DSP 端之间的传递与转换。它通过使用 MPCS 的部件来为分配使用 POOL 组件的共享缓冲器提供一个访问保护机制。GPP 与 DSP 两端的应用程序同步化是通过使用 NOTIFY 的部件来实现的。

DSP 端应用程序通过使用 MPCS，POOL 和 NOTIFY 的部件来实现 TSK。其原理图如下所示：



MPCSXFER 例程演示操作的命令格式如下：./mpcsxfergpp <absolute path of DSP executable> <Buffer Size> <number of transfers> <DSP ProcessorId>

执行“./mpcsxfergpp”可以看到例程的用法和参数的英文解释，如下图：

```
root@tl:/dsplink# ./mpcsxfergpp
Usage : ./mpcsxfergpp <absolute path of DSP executable> <Buffer Size> <number of trans
fers> <DSP ProcessorId>
For infinite transfers,use value of 0 for <number of transfers>
For DSP Processor Id,
        use value of 0  if sample needs to be run on DSP 0
        use value of 1  if sample needs to be run on DSP 1
```

MPCSXFER 例程演示命令如下：

root@tl:/dsplink# ./mpcsxfergpp mpcsxfer.out 1024 1000

执行 1000 次，缓冲区的大小是 1KByte，运行命令后的结果如下图：

```
root@tl:/dsplink# ./mpcsxfergpp mpcsxfer.out 1024 1000
========== Sample Application : MPCSXFER ==========
Entered MPCSXFER_Create ()
Leaving MPCSXFER_Create ()
Entered MPCSXFER_Execute ()
GPP->DSP: Transferred 100 buffers
DSP->GPP: Transferred 100 buffers
GPP->DSP: Transferred 200 buffers
DSP->GPP: Transferred 200 buffers
GPP->DSP: Transferred 300 buffers
DSP->GPP: Transferred 300 buffers
GPP->DSP: Transferred 400 buffers
DSP->GPP: Transferred 400 buffers
GPP->DSP: Transferred 500 buffers
DSP->GPP: Transferred 500 buffers
GPP->DSP: Transferred 600 buffers
DSP->GPP: Transferred 600 buffers
GPP->DSP: Transferred 700 buffers
DSP->GPP: Transferred 700 buffers
GPP->DSP: Transferred 800 buffers
DSP->GPP: Transferred 800 buffers
GPP->DSP: Transferred 900 buffers
DSP->GPP: Transferred 900 buffers
GPP->DSP: Transferred 1000 buffers
DSP->GPP: Transferred 1000 buffers
Leaving MPCSXFER_Execute ()
Entered MPCSXFER_Delete ()
Leaving MPCSXFER_Delete ()
====================================================
```