Linux 64 位系统适配 FAQ

文档版本 00B03

发布日期 2018-03-15

商标声明

 、HISILICON、海思和其他海思商标均为深圳市海思半导体有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

深圳市海思半导体有限公司

地址：深圳市龙岗区坂田华为基地华为电气生产中心　　邮编：518129

网址：http://www.hisilicon.com

客户服务电话：+86-755-28788858

客户服务传真：+86-755-28357515

客户服务邮箱：support@hisilicon.com

前 言

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3559A	V100ES
Hi3559A	V100
Hi3559C	V100

说明
未有特殊说明，Hi3559CV100 与 Hi3559AV100 内容一致。

修订记录

修订记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

文档版本 00B03 (2018-03-15)

2.3.8 和 2.3.9 小节涉及修改

文档版本 00B02 (2018-01-10)

2.3.9 小节涉及修改

文档版本 00B01 (2017-04-28)

第 1 次临时发布。

目 录

1 概述

Linux ARM 定义的 ARMv8，支持 AARCH32 模式和 AARCH64 模式。

- AARCH32 模式地址用 32 位表示，最大 4GB；
- AARCH64 模式地址用 64 位表示，但最多只有后 48 位是有效的地址位，即最大为 2^48 = 256TB。

本文以 Hi3559AV100ES/Hi3559AV100 芯片为例，简述客户在 Linux ARMv8 架构中移植 C 语言代码遇到的问题及解决方法。

2 FAQ

2.1 Linux ARMv8 架构业务部署模式

Linux ARMv8 架构支持以下几种业务部署模式：

- 内核态 64 位，用户态 64 位。
- 内核态 64 位，用户态 32 位。
- 内核态 32 位，用户态 32 位。

注意

Hi3559AV100ES/Hi3559AV100 仅支持内核态 64 位，用户态 64 位。

2.2 编译器对数据类型长度的修改

ARMv8 针对 aarch64 与 aarch32 环境，使用不同的编译器。aarch64 环境使用 64 位编译器 aarch64-hisiv610-linux-，aarch32 环境使用 32 位编译器 arm-hisiv610-linux-。64 位和 32 位的编译器对数据类型位宽差异的影响如下表 2-1 所示：

表2-1 64 位和 32 位的编译器对数据类型位宽差异的影响

Programming Type	Size in 32bit Compiler	Size in 64bit Compiler
char	8 bit	8 bit
short	16 bit	16 bit
int	32 bit	32 bit
pointer	32 bit	64 bit
long	32 bit	64 bit
long long	64 bit	64 bit

float	32 bit	32 bit
double	64 bit	64 bit
size_t	32 bit	64 bit

从表 2-1 可知，64 位编译器对指针和 long，size_t 类型的位宽都升级为 64bit，int 类型保持不变。

2.3 C 语言移植修改注意事项

基本原则：同一份 C 代码需要能够支持 32bit 模式编译和 64bit 模式编译。32bit 和 64bit 的版本代码是归一的。实在无法归一的代码，Linux 内核态处理统一采用 64bit 的宏：CONFIG_64BIT，用户态则需要用户根据选择的编译器位宽自定义相关的宏。

2.3.1 kernel 配置选项 CONFIG_COMPAT 和 compat_ioctl

kernel 配置选项 CONFIG_COMPAT 和 compat_ioctl。

- 内核态 64 位，用户态 64 位，用户态到内核态的系统调用 ioctl，在底层 kernel 驱动中最终调用到 unlocked_ioctl。
- 内核态 64 位，用户态 32 位，用户态到内核态的系统调用 ioctl，在底层 kernel 驱动中最终调用到 compat_ioctl。
- 内核态 32 位，用户态 32 位，用户态到内核态的系统调用 ioctl，在底层 kernel 驱动中最终调用到 unlocked_ioctl。

因此，在 kernel 驱动中除了提供原有的 unlocked_ioctl 外，还需提供 compat_ioctl。

代码示例：

```
static struct osal_fileops vdec_fops =
{
    .open         = vdec_open,
    .release      = vdec_close,
    .mmap       = NULL,
    .poll         = vdec_poll,
    .unlocked_ioctl  = vdec_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl  = vdec_compat_ioctl,
#endif
};
```

2.3.2 谨慎使用可变位宽的数据类型

内核态和用户态共用的变量类型定义不能使用 long 和 size_t。

从表 2-1 可知，如果内核态、用户态共用的变量类型定义有 long 和 size_t，那么当内核态 64 位、用户态 32 位模式下，它们的数据位宽是不一样的，当有数据需要在用户态、内核态之间相互传递时由于数据位宽不一样导致拷贝数据时无论是按照用户态长度拷贝还是按照内核态长度拷贝都是错误的。

- 内核态和用户态共用的变量类型定义不能使用 long 和 size_t，使用 long 和 size_t 定义的变量统一修改为 long long。
- Hisilicon SDK 对以下数据类型进行了重定义。

```
typedef unsigned long long   HI_U64;
typedef long long           HI_S64;
typedef unsigned long        HI_UL;
typedef signed long          HI_SL;
```

2.3.3 指针的处理

如果内核态和用户态共用的结构体、联合体里面定义了指针变量，那么在内核态 64 位、用户态 32 位模式下，这些结构体、联合体的长度由于有指针变量的存在，在内核态和用户态的长度是不一样的。

可以把整个系统里面内核态和用户态共用的结构体、联合体里面的指针变量替换为 long long 变量，在具体使用时再强制类型转换为指针变量，但是这种修改方法工作量大，而且使用 long long 记录地址不如直接使用指针直观。

下面介绍一种修改工作量小的指针处理方法。

- 使用 GCC 编译器指令__attribute__ ((aligned (8)))对指针和紧挨着指针后面的变量单独进行首地址 8 bytes 对齐。

  注意：__attribute__ ((aligned (8)))只能对单个指针进行首地址 8 bytes 对齐，不能对指针数组的每一个指针元素进行首地址 8 bytes 对齐。指针数组需要拆成每一个独立的指针变量或者使用 long long 数组替换。

- compat_ioctl 从用户态把结构体数据 copy_from_user 到内核态之后，在进行正常业务处理之前把指针的高 32bit 赋 0。

  注意：结构体里面的指针变量是用户态地址传递到内核态的才需要把指针的高 32bit 赋 0，内核态地址传递到用户态不需要处理。

代码示例：

头文件对相关的宏、结构体定义：

```
#define ALIGN_NUM 8
#define ATTRIBUTE __attribute__((aligned (ALIGN_NUM)))
#define COMPAT_POINTER(ptr, type)\
do {\
    HI_UL ulAddr   =  (HI_UL)ptr;\
    HI_U32 u32Addr =  (HI_U32)ulAddr;\
    ptr = (type)(HI_UL)u32Addr;\
} while(0)

typedef struct hiVDEC_STREAM_S
```

```
{
    HI_U8* ATTRIBUTE pu8Addr;
    HI_U32  ATTRIBUTE u32Len;
    HI_U64  u64PTS;
    HI_BOOL bEndOfFrame;
    HI_BOOL bEndOfStream;
}VDEC_STREAM_S;
```

用户态代码：

```
HI_S32 HI_MPI_VDEC_SendStream(VDEC_CHN VdChn, const VDEC_STREAM_S
*pstStream)
{
    ......
    return ioctl(g_s32Vdecfd[VdChn],VDEC_CHN_SENDSTREAM_CTRL,pstStream);
}
```

内核态代码：

```
static long vdec_compat_ioctl (unsigned int cmd, unsigned long arg, void
*private_data)
{
    void __user * pArg = (void __user *)arg;
    ……
    switch(cmd)
    {
    ……
        case VDEC_CHN_SENDSTREAM_CTRL:
        {
            VDEC_STREAM_S stStream;
            if(copy_from_user(&stStream, pArg, sizeof(VDEC_STREAM_S)))
            {
                s32Ret = -EFAULT;
                break;
            }
            COMPAT_POINTER(stStream.pu8Addr, HI_U8*);
            s32Ret = VDEC_SendStream(s32ChnID, &stStream);
            break;
        }
        ……
    default:
        {
            s32Ret = HI_FAILURE;
```

```
            break;
    }
    }
    ……
   return s32Ret;
}
static long vdec_ioctl (unsigned int cmd, unsigned long arg, void
*private_data)
{
void __user * pArg = (void __user *)arg;
    ……
   switch(cmd)
{
    ……
       case VDEC_CHN_SENDSTREAM_CTRL:
       {
          VDEC_STREAM_S stStream;
           if(copy_from_user(&stStream, pArg, sizeof(VDEC_STREAM_S)))
           {
               s32Ret = -EFAULT;
              break;
           }
          s32Ret = VDEC_SendStream(s32ChnID, &stStream);
           break;
       }
        ……
        default:
        {
           s32Ret = HI_FAILURE;
       break;
    }
}
……
return s32Ret;
}
```

2.3.4 不同类型数据混合运算

有符号数和无符号数混合运算，长整型和短整型混合运算，需要强制统一各个运算变量的类型。64 位编译器的计算结果是按以下的类型自动转换。

- int + long = long
- unsigned + signed = unsigned

在这种原则下，下列代码：

```
long long a;
int b;
unsigned int c;
b = -2;
c = 1;
a = b + c;
```

a 的值是 0x00000000ffffffff，而不是-1。原因是 b+c 是 unsigned+signed，结果是个无符号数，再转成 long long，只能前面填充 0。

而如下代码：

```
long long a;
int b;
unsigned int c;
b = -2;
c = 1;
a = (long) b + c;
```

a 的结果就是 0xffffffffffffffff （-1），因为 b 转为 long 后就是 0xfffffffffffffffe，与 unsigned int c 相加，就是上述结果。

2.3.5 常量表达式指定数据类型

如果用户想使一个常量为 long，必须清楚地在数值后跟上 L，否则，编译器有可能把它当作 int 来对待。

- 各种常量类型定义如下：

1L // (long)

1LL // (long long)

1U // (unsigned)

1UL // (unsigned long)

1ULL // (unsigned long long)

错误代码示例：

```
long long SetBitN(long long value, unsigned bitNum)
{
    long long mask;
    mask = 1 << bitNum;
    return value | mask;
}
```

在 64 位下乍一看以为没问题，实际上 1 当成 int 来处理，如果 bitNum 为超过 32 的数，超过的部分被截断了，结果就是 0。

正确代码示例：

```
long long SetBitN(long long value, unsigned bitNum)
 {
     long long mask;
     mask = 1L << bitNum;
     return value | mask;
}
```

2.3.6 指针和 int、指针和 long long 之间不能直接强转

指针和整型之间强制类型转换必须是同等长度的，不然编译器会报 warning。长整型和短整型之间强制转换编译器不会报 warning，短整型强转为长整型时编译器会截断高位，保留低位。

- 64 位系统上指针和 int 之间直接强转编译器会报 warning；
- 32 位系统上指针和 long long 之间直接强转编译器会报 warning；

因此，为了代码的归一性，指针和 int、指针和 long long 之间不能直接转换，需要通过 unsigned long 间接转换。

代码示例：

```
HI_S32 value;
HI_U64 addr;
addr = (HI_U64)(HI_UL)&value;
```

2.3.7 调用外部函数一定要对其进行声明

C89 有一个隐式声明规则(implicit declaration)，当需要调用一个函数但找不到函数原型时，编译器会提供一个隐式声明，该隐式声明会假定函数返回值类型为 int，C99 已经去掉了这一规则，要求函数调用必须有函数声明，但 GCC 可能为了兼容老代码，并没有强制执行 C99，只是给出了一个警告。

```
unsigned long long addr;
addr = Get_PhyAddr(1);
```

比如上面应用程序的代码调用了外部函数 unsigned long long Get_PhyAddr(int id)，但是没有对函数 Get_PhyAddr 进行声明，GCC 找不到函数 Get_PhyAddr 的原型，于是假定函数 Get_PhyAddr 的返回值类型是 int，把函数 Get_PhyAddr 返回值赋给变量 addr 就相当于把一个 32 位的有符号数赋值给一个 64 位的无符号数。C 语言规定当赋值表达式两边类型不相同时，等号右边的类型会转成等号左边的类型，于是 32 位的有符号 int 被转换转成 64 位的无符号数，编译器生成符号扩展指令，该指令导致的结果是 addr 的高 32bit 值被置为全 1 或全 0（取决于转换之前 int 最高位的值）。另外，64 位系统上如果函数的返回值是指针，指针是 64 位，int 是 32 位，如果未对调用的返回值为指针的外部函数进行声明也会产生类似错误。

2.3.8 标准系统函数 mmap 和 mmap64 的使用

```
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t
off);
void *mmap64(void *addr, size_t len, int prot, int flags, int fildes,
off64_t off);
```

```
int munmap(void *addr, size_t length);
```

因为 size_t 是 unsigned long 的的重定义，而 long 的位宽跟随系统位宽改变，因此

- 之前在 32 位系统上一般使用标准系统函数 mmap 在用户态对物理地址映射得到虚拟地址，然后用虚拟地址进行各种业务操作，但是 32 系统上的函数 mmap 只能映射不超过 32bit 的物理地址。
- 当内核态是 64 位，用户态是 32 位时，如果需要映射的物理地址超过 32bit 时，就一定要使用函数 mmap64。特别需要注意的是使用函数 mmap64，必须要在编译选项中加上以下三个宏定义：
  - D_FILE_OFFSET_BITS=64
  - D_LARGEFILE_SOURCE
  - D_LARGEFILE64_SOURCE
- 打开以上 3 个宏定义之后，C 库会把 mmap 重定义为 mmap64，因此为了代码书写的归一性，代码中可以不显示调用 mmap64，而是统一写为调用 mmap，但是要注意最后一个参数传参的长度必须以 long long 类型传入才能映射超过 32bit 的物理地址。64 位系统中可以使用 mmap 对物理地址得到虚拟地址。

代码示例：参见 2.3.9 device 类型内存和 DC ZVA 指令的冲突中的代码示例。

2.3.9 device 类型内存和 DC ZVA 指令的冲突

64 位系统上使用 memset、memcpy 等批处理函数访问 device 属性的虚拟地址有 8Bytes 对齐的限制。

【现象描述】

内核态 64 位，用户态 64 位系统上，使用 memset、memcpy 访问设备节点/dev/mem 对应的 mmap 函数映射出来的虚拟地址，当首地址或者长度非 8Bytes 对齐时出触发对齐异常。

Unhandled fault: alignment fault (0x92000061) at 0x0000007f89ed7128

【原因分析】

标准设备节点/dev/mem 对应的 mmap 函数在分配虚拟内存时使用的是 device 类型的内存，这种内存是不可 cache、不可 reorder 的 IO 地址。ARMv8 架构的 64 位系统由于性能的需要在 memset、memcpy 等批处理函数中新增了 DC ZVA 指令，但是 DC ZVA 指令不能用于 device 类型的内存，否则就会触发对齐异常。

【解决办法】

Hisilicon SDK 提供两个不同的接口分别用于映射 device/非 device 类型地址。

- 用户自定义接口 MPI_USER_IOMmap 用于映射 device 类型（不可 cache、不可 reorder）的 IO 地址，比如寄存器地址，此接口 open 的是标准内核节点 dev/mem。用此接口映射的地址不能使用 memset、memcpy 等批处理函数。由于 dev/mem 节点是带 root 权限的，网络安全风险较高，因此此接口不宜在 Hisilicon SDK 中提供，在用户程序中封装成带 static 前缀的接口仅限当前文件使用较为安全。
- Hisilicon SDK 提供接口 HI_MPI_SYS_Mmap open 节点/dev/mmz_userdev，在 MMZ 中实现映射非 device 类型（不可 cache、可 reorder）的 DDR 地址。

- MPI_USER_IOMmap 和 HI_MPI_SYS_Mmap 对应的解除映射接口都是 HI_MPI_SYS_Munmap。

代码示例：

```
static HI_VOID * MPI_USER_IOMmap(HI_U64 u64PhyAddr, HI_U32 u32Size)
{
    HI_U32 u32Diff;
    HI_U64 u64PagePhy;
    HI_U8 * pPageAddr;
    HI_UL   ulPageSize;

  if (s_s32MemDev <= 0)
    {
        s_s32MemDev = open("/dev/mem", O_RDWR|O_SYNC);
        if (s_s32MemDev < 0)
        {
            perror("Open dev/mem error");
            return NULL;
        }
    }

    if (!u32Size)
    {
        printf("Func: %s u32Size can't be 0.\n", __FUNCTION__);
        return NULL;
    }

    /* The mmap address should align with page */
    u64PagePhy = u64PhyAddr & 0xfffffffffffff000UL;
    u32Diff    = u64PhyAddr - u64PagePhy;

    /* The mmap size shuld be mutliples of 1024 */
    ulPageSize = ((u32Size + u32Diff - 1) & 0xfffff000UL) + 0x1000;

    pPageAddr   = mmap ((void *)0, ulPageSize, PROT_READ|PROT_WRITE,
                                    MAP_SHARED, s_s32MemDev, u64PagePhy);

    if (MAP_FAILED == pPageAddr )
    {
        perror("mmap error");
        return NULL;
    }
    return (HI_VOID *) (pPageAddr + u32Diff);
}
```

```
HI_VOID * HI_MPI_SYS_Mmap(HI_U64 u64PhyAddr, HI_U32 u32Size)
{
    HI_U32 u32Diff;
    HI_U64 u64PagePhy;
    HI_U8 * pPageAddr;
    HI_UL   ulPageSize;

    if (g_s32MmzFd <= 0)
    {
        g_s32MmzFd = open("/dev/mmz_userdev", O_RDWR);
        if (g_s32MmzFd < 0)
        {
            perror("open mmz_userdev");
            return NULL;\
        }
    }

    if (!u32Size)
    {
        printf("Func: %s u32Size can't be 0.\n", __FUNCTION__);
        return NULL;
    }

    u64PagePhy = u64PhyAddr & 0xfffffffffffff000UL;
    u32Diff    = u64PhyAddr - u64PagePhy;
    ulPageSize = ((u32Size + u32Diff - 1) & 0xfffff000UL) + 0x1000;

    pPageAddr   = mmap ((void *)0, ulPageSize, PROT_READ|PROT_WRITE,
                                        MAP_SHARED, g_s32MmzFd, u64PagePhy);

    if (MAP_FAILED == pPageAddr )
    {
        perror("mmap error");
        return NULL;
    }
    return (HI_VOID *) (pPageAddr + u32Diff);
}

HI_S32 HI_MPI_SYS_Munmap(HI_VOID* pVirAddr, HI_U32 u32Size)
{
    HI_U64 u64PageAddr;
    HI_U32 u32PageSize;
```

```
    HI_U32 u32Diff;

    u64PageAddr = (((HI_UL)pVirAddr) & 0xfffffffffffff000UL);
    u32Diff     = (HI_UL)pVirAddr - u64PageAddr;
    u32PageSize = ((u32Size + u32Diff - 1) & 0xfffff000UL) + 0x1000;

    return munmap((HI_VOID*)(HI_UL)u64PageAddr, u32PageSize);
}
```

2.3.10 内核相关 ioremap 函数的区别

- Ioremap：device 类型，一般用于映射 non-cacheable、不可 reorder 的 IO 地址，比如寄存器地址。
- ioremap_nocache：实现与 ioremap 完全一样。
- ioremap_wc：非 device 类型，一般用于映射 non-cacheable、可 reorder 的 memory 地址。
- ioremap_cache：非 device 类型，一般用于映射 cacheable 的 memory 地址。