Application Note

将软件从 8 位（字节）可寻址 CPU 迁移至 C28x CPU

Veena Kamath

摘要

在开发应用程序时，您可能会假设器件在 8 位可寻址器件上运行，并且在将代码移植到 16 位可寻址器件时会遇到问题。本应用手册讨论了此类常见场景，并提供了如何开发应用程序而不考虑可寻址性的指南。

内容

插图清单

表格清单

商标

1 引言

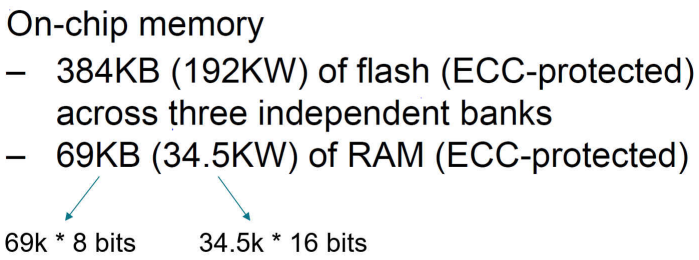
TMS320C28x 是 TMS320™ 系列中多个定点 CPU 之一。与 Arm® CPU 等 8 位可寻址架构相比，它具有 16 位可寻址架构。

在 8 位可寻址架构中，每 8 位数据都有一个唯一的地址，在 16 位可寻址器件中，每 16 位数据都有一个唯一的地址。因此，在 16 位架构中，存储器的最小可寻址单元和支持的最小数据类型为 16 位。

2 字节与字术语

一直以来，字节被定义为存储器的最小可寻址单元。因此，从技术上讲，字节的大小取决于硬件：C28x 器件为 16 位，Arm 器件为 8 位。但如今，字节用作 8 位的同义词，因为大多数器件架构都是 8 位可寻址的。为避免混淆，我们使用术语 8 位字节和 16 位字

器件特性集中的字节和字用法文档：



从 C28x 编译器的角度来看，存储器大小始终以 16 位的最小可寻址单元表示。这包括存储器长度、链接器 cmd 文件、.map 文件中提供的代码/数据大小等。标准 sizeof() 函数还返回以 16 位为单位的大小。

- 器件特定数据表中的存储器映射表：

表 2-1. 存储器映射表

存储器	大小	起始地址	终止地址
LS0 RAM	2K × 16	0x0000 8000	0x0000 87FF
LS1 RAM	2K × 16	0x0000 8800	0x0000 8FFF

- 链接器命令文件：

```
RAMLS0     : origin = 0x00008000,  length = 0x0800
RAMLS1     : origin = 0x00008800,  length = 0x0800
```

- Code Composer Studio™ (CCS) 生成的 .map 文件：

```
MEMORY CONFIGURATIONS

      name           origin       length       used         unused       attr
-------------      --------     ------       ----         ------       ----
  RAMLS0           00008000     00000800     00000112     000006ee     RWIX
  RAMLS1           00008800     00000800     00000194     0000066c     RWIX
```

- sizeof(uint32_t) = 2 → 2 * 16 位

3 需要考虑的要点

3.1 不支持 8 位数据类型

在基于 C28x CPU 的项目中，不支持 8 位数据类型。char 为 16 位宽，uint8_t 和 INT8_t 类型不是由 C28x 编译器定义的。C2000Ware 将这些数据重映射到 uint16_t 和 int16_t 数据类型。有关数据类型的更多信息，请参阅 *TMS320C28x 优化 C/C++ 编译器 v22.6.0.LTS 用户指南*。

但是，C28x 编译器为字节访问提供了内在 __byte()。有关详细信息，请参阅 https://software-dl.ti.com/ccs/esd/documents/c2000_byte-accesses-with-the-c28x-cpu.html。

- 在将应用从 8 位可寻址架构（例如 Arm）移植到 C28x 时，由于这种差异，您可能会发现存储器要求有所增加。

  示例：

```
struct
{
   uint8_t a;
   uint8_t b;
   uint16_t c;
} myStruct;
```

  在 Arm 器件中，myStruct 的大小为 8 + 8 + 16 = 32 位。而在 C28x 中，大小为 16 + 16 + 16 = 48 位。
- C28x 中的 int8 或 char 数据类型为 16 位宽，因此编译器在执行算术或移位运算时不会在 0xFF 处执行回绕。

  示例：

```
uint8_t a = 0xFF;
a += 1;
if (a == 0)
{
     //Condition is true for Arm and false for C28x
}
```

- 在将较大的数据类型转换为较小的数据类型或将较小的数据类型转换为较大的数据类型时，应注意正确类型转换、使用掩码并注意器件字节序。基于 C28x 的器件是小字节序器件。

```
uint8_t a[4] = {0x1, 0x2, 0x3, 0x4};

uint32_t b1 = *(uint32_t *)a;   ✘ The result varies with endianness as well. C28x is little endian device and
                                  b1 = 0x00020001. In ARM little endian device, b1 = 0x04030201

uint32_t b2 = (a[0] << 0)  |    ✘ Shifting beyond the data type size. The behavior is compiler specific. In the C28x
              (a[1] << 8)  |      compiler, the value is truncated if shifted beyond its size. It also throws a warning
              (a[2] << 16) |
              (a[3] << 24);

uint32_t b2 = ((uint32_t)a[0] << 0)  |   ✓ Recommended usage of typecasts and shift operators.
              ((uint32_t)a[1] << 8)  |     b3 = 0x04030201 in both C28x and ARM-LE
              ((uint32_t)a[2] << 16) |
              ((uint32_t)a[3] << 24);


uint32_t a = 0x12345678;

uint8_t b1 = a;          ✘ b1 = 0x5678 in C28x, 0x78 in ARM

uint8_t b2 = (uint8_t)a; ✘ b2 = 0x5678 in C28x, 0x78 in ARM

uint8_t b3 = a & 0xFF;   ✓ Recommended usage of masks. b3 = 0x78 in both C28x and ARM
```

- 需要重新讨论使用联合体的问题。

```
union
{
    struct
    {
        uint8_t byte1;
        uint8_t byte2;
        uint8_t byte3;
        uint8_t byte4;
    };
    uint32_t word;
}test;
```

✘ The total size of the union will be 64 bits in C28x and 32 bits in ARM
b1=1, b2=1, b3 =1, b4 =1 → word = 0x00010001 in C28x
→ word = 0x01010101 in ARM

```
union
{
    struct
    {
        uint8_t byte1 : 8;
        uint8_t byte2 : 8;
        uint8_t byte3 : 8;
        uint8_t byte4 : 8;
    };
    uint32_t word;
}test;
```

✓ Recommended usage of bitfields
b1=1, b2=1, b3 =1, b4 =1 → word = 0x01010101 in C28x and ARM

3.2 存储器大小用 16 位表示

链接器 cmd 文件、.map 文件中提到的存储器大小都用 16 位表示。sizeof() 函数始终返回相对于最小可寻址存储器单元的大小（Arm 为 8 位，C28x 为 16 位）。

请注意不要在应用程序中对尺寸信息进行硬编码。一种常见的情况是使用 memset/cpy/cmp 函数时，这些函数的 size 参数是以最小可寻址单元表示的。

```
struct
{
    uint16_t a;
    uint16_t b;
    uint32_t c;
} myStruct;
```

The total size of the struct is 64 bits in C28x and in ARM.
But sizeof(myStruct) = 4 in C28x, and 8 in ARM

```
memset(&myStruct, 0xA, 8);
```

✘ Hardcoded memory size – In the case of C28x, corrupts the adjacent memory

```
memset(&myStruct, 0xA, sizeof(myStruct));
```

✓ Recommended usage of sizeof() instead of assuming it as 8

此外，请注意在使用 memset 后数据打包的差异：

C28x

```
myStruct
000A000A 000A000A
```

Arm

```
myStruct
0A0A0A0A 0A0A0A0A
```

3.3 数组和结构：各个元素偏移量是不同的

- 在 C28x 中，每 16 位有一个唯一地址，而 Arm 器件中为 8 位。

  示例：

```
uint32_t  Array32[4]   = {1,2,3,4};
uint16_t  Array16[4]   = {1,2,3,4};
uint8_t   Array8[4]    = {1,2,3,4};
```

Expression	Type	Value	Address	
Array32	unsigned long[4]	[1,2,3,4]	0x0000A800@Data	
[0]	unsigned long	1	0x0000A800@Data	Address increments by 2 for a 32-bit array
[1]	unsigned long	2	0x0000A802@Data	
[2]	unsigned long	3	0x0000A804@Data	
[3]	unsigned long	4	0x0000A806@Data	
Array16	unsigned int[4]	[1,2,3,4]	0x0000A80E@Data	
[0]	unsigned int	1	0x0000A80E@Data	Address increments by 1 for a 16-bit array
[1]	unsigned int	2	0x0000A80F@Data	
[2]	unsigned int	3	0x0000A810@Data	
[3]	unsigned int	4	0x0000A811@Data	
Array8	unsigned int[4]	[1,2,3,4]	0x0000A812@Data	
[0]	unsigned int	1	0x0000A812@Data	uint8_t = uint16_t Address increments by 1
[1]	unsigned int	2	0x0000A813@Data	
[2]	unsigned int	3	0x0000A814@Data	
[3]	unsigned int	4	0x0000A815@Data	

图 3-1. C28x 内核中的存储器分配

Expression	Type	Value	Address	
Array32	unsigned int[4]	[1,2,3,4]	0x2000C000	
[0]	unsigned int	1	0x2000C000	Address increments by 4 for a 32-bit array
[1]	unsigned int	2	0x2000C004	
[2]	unsigned int	3	0x2000C008	
[3]	unsigned int	4	0x2000C00C	
Array16	unsigned short[4]	[1,2,3,4]	0x2000C010	
[0]	unsigned short	1	0x2000C010	Address increments by 2 for a 16-bit array
[1]	unsigned short	2	0x2000C012	
[2]	unsigned short	3	0x2000C014	
[3]	unsigned short	4	0x2000C016	
Array8	unsigned char[4]	[1 '\x01',2 '...	0x2000C018	
[0]	unsigned char	1 '\x01'	0x2000C018	Address increments by 1 for a 8-bit array
[1]	unsigned char	2 '\x02'	0x2000C019	
[2]	unsigned char	3 '\x03'	0x2000C01A	
[3]	unsigned char	4 '\x04'	0x2000C01B	

图 3-2. Arm 内核中的存储器分配

- 使用数组时，基于索引的访问和基于指针增量的访问将在两个器件中产生相同的结果。但是，如果您尝试使用硬编码偏移，则会产生不同的结果。

```
uint32_t read1 = Array32[2];                                  ✓ Index-based access - compiler takes care of using the correct offset
uint32_t read2 = *(Array32 + 2);                              ✓ Pointer increment – compiler takes care of incrementing by the right size
uint32_t read3 = *((uint32_t *)(baseAddr + 8));               × Hardcoded offset value – Gives different results in C28x and ARM
uint32_t read4 = *((uint32_t *)(baseAddr + 2*sizeof(uint32_t))); ✓ Recommended usage of sizeof() instead of assuming it as 4
```

- 这同样适用于结构体。struct.element 在两个器件中提供相同的结果，而使用 (baseaddr + offset) 访问结构体。

3.4 标准数据类型宽度的差异

与任何 8 位可寻址架构不同，C28x 器件中 int 和 char 的大小不同。为了实现更好的可移植性，强烈建议使用基于宽度的数据类型，例如 uint16_t、int16_t、uint32_t、int32_t、uint64_t、int64_t、float32_t、float64_t 等。这些数据类型在 C28 编译器头文件 stdint.h 中定义。

请注意，uint8_t 和 int8_t 数据类型不是由 C28x 编译器定义。C2000Ware 会分别将这些数据重映射到 uint16_t 和 int16_t 数据类型。

类型	大小
char	16 位
_Bool	16 位
short	16 位
int	16 位
long	32 位
long long	64 位
float	32 位
double(COFF)	32 位
double(EABI)	64 位
long double	64 位
指针	32 位

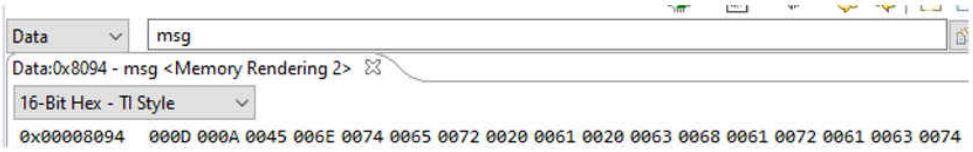
有关数据类型的更多信息，请参阅 *TMS320C28x 优化 C/C++ 编译器 v22.6.0.LTS 用户指南*。

3.5 处理 8 位通信协议

C28x 器件中的控制器局域网 (CAN)、串行通信接口 (SCI) 和通用异步接收器/发送器 (UART) 等通信外设支持 8 位数据传输。但请注意，数据将以 16 位的形式存储在存储器中。C28x 编译器不支持 pragma pack(1)。

示例：

```
unsigned char *msg;
msg = "\r\nEnter a character: \0";
SCI_writeCharArray(SCIA_BASE, (uint16_t*)msg, 22);
```



Memory Browser view
1 character = 16 bits

The SCI TX pin captured in a logic analyzer.
1 character = 8 bits

在此示例中，该函数将 uint16_t * 作为输入数据参数。但是，它仅需要每个 uint16_t 类型有 8 位数据。uint16_t 的上半部分被忽略，仅传输低 8 位。

4 参考文献

- *TMS320C28x 优化 C/C++ 编译器 v22.6.0.LTS 用户指南*
- https://software-dl.ti.com/ccs/esd/documents/c2000_byte-accesses-with-the-c28x-cpu.html

5 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from Revision * (March 2023) to Revision A (April 2023)	Page
• 更新了 节 2	2
• 更新了 节 3.1	3
• 更新了 节 3.2	4

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265