Application Note

Migrating Software From 8-Bit (Byte) Addressable CPU's to C28x CPU

Veena Kamath

ABSTRACT

While developing applications, you may assume that the device runs on a 8-bit addressable device and can face issues while porting the code to a 16-bit addressable device. This application note discusses such common scenarios and provides a guide on how to develop application irrespective of the addressability.

Table of Contents

List of Figures

List of Tables

Trademarks

TMS320™ and Code Composer Studio™ are trademarks of Texas Instruments.
Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
All trademarks are the property of their respective owners.

1 Introduction

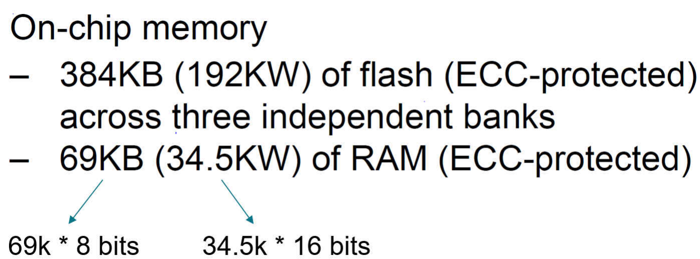
The TMS320C28x is one of several fixed-point CPUs in the TMS320™ family. It has a 16-bit addressable architecture compared to 8-bit addressable architecture like Arm® CPUs.

In a 8-bit addressable architecture, every 8 bits of data has a unique address, where as in a 16-bit addressable device , every 16 bits of data has a unique address. Hence, in a 16-bit architecture, the smallest addressable unit of memory and the smallest data type supported is 16 bits.

2 Byte vs Word Terminology

Historically, byte is defined as the smallest addressable unit of memory. Hence, technically, the size of a byte is hardware dependent: 16 bits for C28x devices and 8 bits for Arm devices. But nowadays, byte is used as a synonym of 8 bits, since majority of the device architectures are 8 bit addressable. To avoid the confusion, let us use the terminology 8-bit bytes and 16-bit words

Byte and Word Usage in the device feature set Documentation:



From the C28x compiler's perspective, the memory size is always expressed in smallest addressable units, which is 16 bits. This includes the memory length, code/data size provided in the linker cmd files, .map file, and so forth. The standard sizeof() function also returns the size in 16 bits.

- Memory Map Table in device-specific data sheet:

Table 2-1. Memory Map Table

Memory	Size	Start Address	End Address
LS0 RAM	2K × 16	0x0000 8000	0x0000 87FF
LS1 RAM	2K × 16	0x0000 8800	0x0000 8FFF

- Linker command file:

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

- .map file generated by Code Composer Studio™ (CCS):

```
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 bits

3 Key Points to Consider

3.1 8-Bit Data Types are not Supported

In C28x CPU-based projects, 8-bit data types are not supported. char is 16 bits wide and the types uint8_t and int8_t are not defined by the C28x compiler. C2000Ware remaps these to uint16_t and int16_t data types. For more details on the data types, see the *TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS User's Guide*.

However, the C28x compiler provides intrinsic __byte() for byte accesses. For more details, see the https://software-dl.ti.com/ccs/esd/documents/c2000_byte-accesses-with-the-c28x-cpu.html.

- While porting an application from an 8-bit addressable architecture, such as Arm to C28x, you may find an increase in memory requirement due to this difference.

  Example:

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

  In an Arm device, the size of myStruct would be 8 + 8 + 16 = 32 bits. Whereas, in C28x, the size would be 16 + 16 + 16 = 48 bits.
- Since the int8 or char data types in C28x is 16 bit wide, the compiler will not do a wraparound at 0xFF while performing arithmetic or shift operations.

  Example:

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

- While converting larger data types to smaller ones or vise versa, care should be taken to properly typecast, use masks, and be aware of the device endianness. The C28x-based devices are little endian devices.

```
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)  |
              (a[1] << 8)  |
              (a[2] << 16) |
              (a[3] << 24);
```

✘ Shifting beyond the data type size. The behavior is compiler specific. In the C28x compiler, the value is truncated if shifted beyond its size. It also throws a warning

```
uint32_t b2 = ((uint32_t)a[0] << 0)  |
              ((uint32_t)a[1] << 8)  |
              ((uint32_t)a[2] << 16) |
              ((uint32_t)a[3] << 24);
```

✓ Recommended usage of typecasts and shift operators. b3 = 0x04030201 in both C28x and ARM-LE

```
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

- Usage of unions needs to be revisited.

```
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 Memory Size is Expressed in 16 Bits

The memory sizes mentioned in the linker cmd files, .map files are all expressed in 16 bits. The sizeof() function always returns the size with respect to smallest addressable memory units (8 bits for Arm and 16 bits for C28x).

Be careful to not hardcode the size information in the application. One common scenario is while using memset/cpy/cmp functions, the size parameter to these functions are in terms of smallest addressable units.

```
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

Also, note the difference in data packing after memset usage:

C28x

```
myStruct
000A000A 000A000A
```

Arm

```
myStruct
0A0A0A0A 0A0A0A0A
```

3.3 Arrays and Structures: Individual Element Offsets are Different

- In C28x, there is a unique address for every 16 bits, as opposed to 8 bits in an Arm device.

  Example:

```
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
(x)= [0]	unsigned long	1	0x0000A800@Data
(x)= [1]	unsigned long	2	0x0000A802@Data
(x)= [2]	unsigned long	3	0x0000A804@Data
(x)= [3]	unsigned long	4	0x0000A806@Data
Array16	unsigned int[4]	[1,2,3,4]	0x0000A80E@Data
(x)= [0]	unsigned int	1	0x0000A80E@Data
(x)= [1]	unsigned int	2	0x0000A80F@Data
(x)= [2]	unsigned int	3	0x0000A810@Data
(x)= [3]	unsigned int	4	0x0000A811@Data
Array8	unsigned int[4]	[1,2,3,4]	0x0000A812@Data
(x)= [0]	unsigned int	1	0x0000A812@Data
(x)= [1]	unsigned int	2	0x0000A813@Data
(x)= [2]	unsigned int	3	0x0000A814@Data
(x)= [3]	unsigned int	4	0x0000A815@Data

Address increments by 2 for a 32-bit array

Address increments by 1 for a 16-bit array

uint8_t = uint16_t
Address increments by 1

Figure 3-1. Memory Allocation in C28x Core

Expression	Type	Value	Address
Array32	unsigned int[4]	[1,2,3,4]	0x2000C000
(x)= [0]	unsigned int	1	0x2000C000
(x)= [1]	unsigned int	2	0x2000C004
(x)= [2]	unsigned int	3	0x2000C008
(x)= [3]	unsigned int	4	0x2000C00C
Array16	unsigned short[4]	[1,2,3,4]	0x2000C010
(x)= [0]	unsigned short	1	0x2000C010
(x)= [1]	unsigned short	2	0x2000C012
(x)= [2]	unsigned short	3	0x2000C014
(x)= [3]	unsigned short	4	0x2000C016
Array8	unsigned char[4]	[1 '\x01',2 '...	0x2000C018
(x)= [0]	unsigned char	1 '\x01'	0x2000C018
(x)= [1]	unsigned char	2 '\x02'	0x2000C019
(x)= [2]	unsigned char	3 '\x03'	0x2000C01A
(x)= [3]	unsigned char	4 '\x04'	0x2000C01B

Address increments by 4 for a 32-bit array

Address increments by 2 for a 16-bit array

Address increments by 1 for a 8-bit array

Figure 3-2. Memory Allocation in Arm Core

- While using arrays, the index-based accesses and pointer increment-based access would yield the same results in both the devices. But if you try to use hardcoded offsets, it would produce different results.

`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

- Same is applicable for structs. While struct.element provides the same results in both devices, the usage of (baseaddr + offset) to access struct.

3.4 Difference in Standard Data Type Widths

As opposed to any 8-bit addressable architecture, the sizes of int and char are different in C28x devices. For better portability, it is highly recommended to use the width-based data types such as uint16_t, int16_t, uint32_t, int32_t, uint64_t, int64_t, float32_t, float64_t, and so forth. These data types are defined in the C28 compiler header file stdint.h.

Note that the data types uint8_t and int8_t are not defined by the C28x compiler. C2000Ware remaps these to uint16_t and int16_t data types, respectively.

Type	Size
char	16 bits
_Bool	16 bits
short	16 bits
int	16 bits
long	32 bits
long long	64 bits
float	32 bits
double(COFF)	32 bits
double(EABI)	64 bits
long double	64 bits
Pointers	32 bits

For more details on the data types, see the *TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS User's Guide* .

3.5 Dealing With 8-Bit Communication Protocols

Communication peripherals like the Controller Area Network (CAN), Serial Communications Interface (SCI) and Universal Asynchronous Receiver/Transmitter (UART) in C28x devices support 8-bit data transfers. But note that the data would be stored as 16 bits in the memory. The C28x compiler does not support pragma pack(1).

Example:

```
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

In this example, the function takes in uint16_t * as the input data parameter. But it only expects 8 bits of data per uint16_t type. The upper half of the uint16_t is ignored and only the lower 8 bits are transferred.

4 References

- *TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS User's Guide*
- https://software-dl.ti.com/ccs/esd/documents/c2000_byte-accesses-with-the-c28x-cpu.html

5 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (March 2023) to Revision A (April 2023)	Page
• Updated the numbering format for tables, figures and cross-references throughout the document	2
• Updated Section 2	2
• Updated Section 3.1	3
• Updated Section 3.2	4