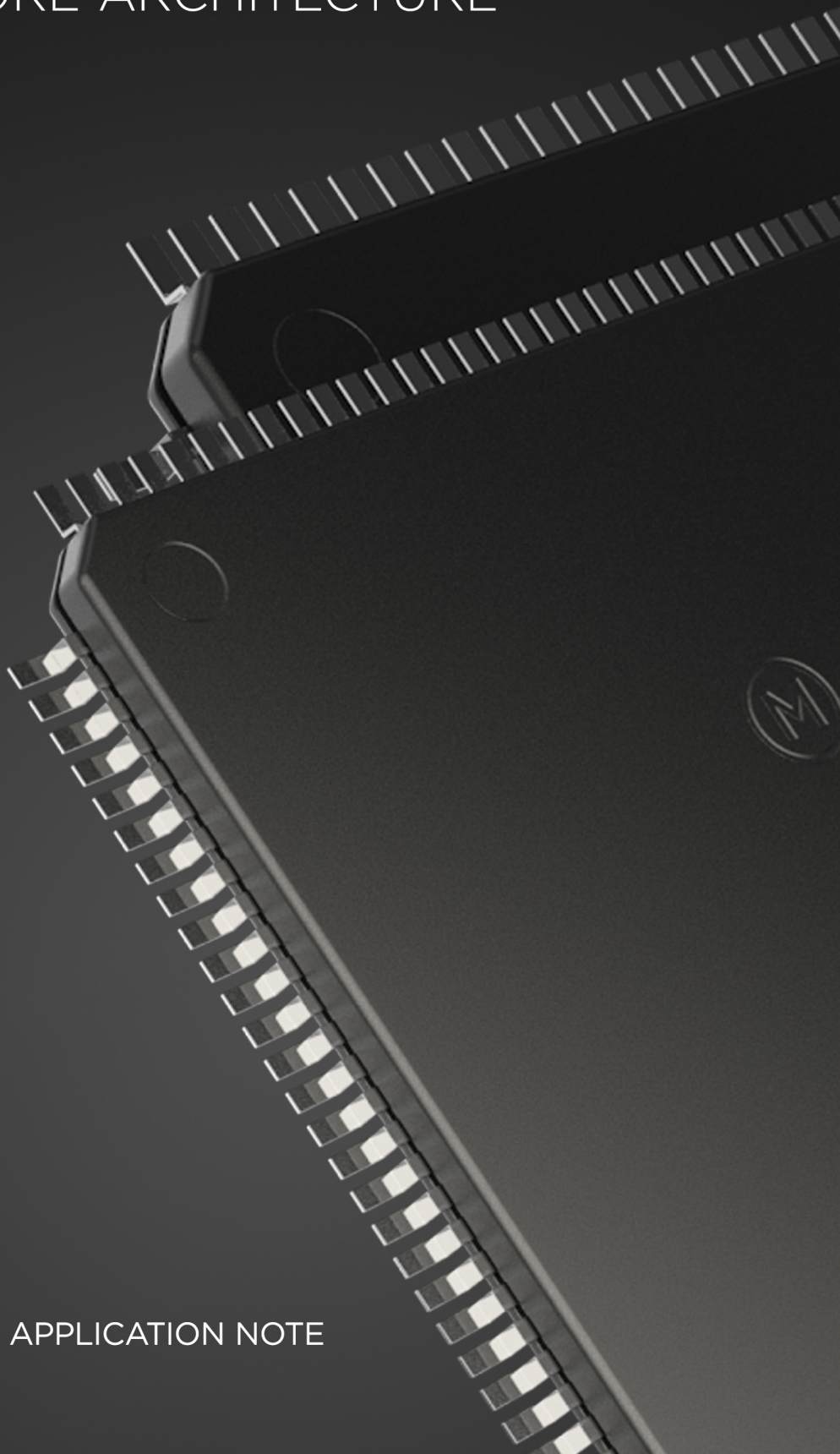
TASKING®

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE



APPLICATION NOTE

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE

The TriCore architecture comprises some constraints related to the alignment of certain data types, structs and struct members. This article describes the following alignment related topics:

- Alignment restrictions of the TriCore architecture
- Alignment restrictions of the Infineon EABI
- Alignment options provided by the C compiler
- Alignment options provided by the LSL linker script language
- Precautions

ALIGNMENT RESTRICTIONS OF THE TRICORE ARCHITECTURE

The TriCore architecture supports data access instructions which require a certain alignment. This is e.g. valid for:

- Core register accesses. MFCR / MTCR instructions used to access a Core Special Function Register (CSFR) require a word aligned address.
- Context store instructions require a halfword aligned address. LDUCX, LDLCX, STLCX, STUCX
- Load and store instructions for word or double word values require a halfword aligned address. ST.W, LD.W, ST.D, LD.D
- SWAP.W, LDMST and other Read-Modify-Write instructions require the address to be Word aligned (See the TriCore Architecture Manual).
- Pointers must always be word aligned.

The TASKING C compiler for TriCore ensures that those alignment restrictions are not violated.

ALIGNMENT RESTRICTIONS OF THE INFINEON EABI

The Infineon Embedded Application Binary Interface (EABI) specification is a set of interface standards that writers of compilers and assemblers and linker/locators must use when creating compliant tools for the TriCore architecture. A compiler vendor however may also offer alternative settings e.g. to achieve a more compact data placement to reduce unnecessary alignment gaps.

Characters must be stored on byte boundaries. Short integers must be two byte aligned. Data types with a size of four bytes or larger must be four byte aligned.

The EABI requires an array alignment depending on the array member alignment. `char` arrays are byte aligned, `short` arrays halfword aligned, `int`, `long`, `float`, `double` or `long long` arrays are word aligned.

In TASKING TriCore tools before version v6.2r1, when the EABI compliance option is enabled **(--eabi-compliant)**, the alignment of a struct is:

- A minimum alignment of 2 bytes when the struct size is larger than 1 byte.
  For a struct with an overall size of 8 bytes or more the TASKING tools use a word alignment and the overall struct size is set to a multiple of 4 bytes for efficient struct copy operations.
- A word alignment when the struct includes an `int`, `long`, `float`, `double` or l`ong long` data type member. The struct size is a multiple of the struct alignment. Padding is applied between struct members to fulfill this requirement. At the end of a struct padding is also applied to fulfill the struct size requirement.

  The TASKING tools apply a word alignment if the struct size is larger or equal than 8 bytes for a penalty free performance of double word copy instructions. Double-word read/write accesses for addresses starting not on a word address but on a halfword address are slower.

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE

In EABI v3.0 struct alignment requirements changed to:

- A structure having a size larger than 1 byte, containing only members with 1 byte alignment, has an alignment of 2 bytes. The overall struct size is a multiple of 2 bytes instead of a multiple of 4 bytes when the struct size is larger or equal than 8 bytes.
- The size of a union or structure must be an integral multiple of its alignment. A struct containing `char` and `short` type values only has a size of a multiple of 2 bytes.

This EABI change is incorporated in the TASKING TriCore tools since v6.2r1.

ALIGNMENT OPTIONS SUPPORTED BY THE TASKING TOOLS

The TASKING TriCore tools offer means to change the section alignment or data alignment. This might be useful to increase data density e.g. for struct member offset or access performance when an access to a data value following a certain alignment is more efficient (less stall cycles).

These are:

- `__align(n), __attribute__((__align(n))) , __attribute__((aligned(n))), #pragma align n`
  With the attributes you can increase the default alignment of variables, functions or since v6.3r1 also structure members. `__attribute__((aligned(n)))` was added in v6.3r1. With the pragma you can increase the default alignment of variables and functions. Structure member alignment is not affected by the pragma. n must be a power of two or 0.

  Changing the function alignment is supported for TriCore tools versions v6.1r1 and up.

  Changing the data alignment is applicable for all data types for v6.1r1 and up. For older versions the alignment change is only possible for a data type having a size of 4 bytes or more or an array / struct size of 4 bytes or more.

  Examples:

```
__align(8) int var_1;  /* This variable will have a 8 byte alignment. */

__align(32) void func(void) /* This function will be 32 byte aligned. */
{
}
```

- `__unaligned`

  With the type qualifier `__unaligned` (introduced in v6.3r1) you can specify to suppress the alignment of objects or structure members. This can be useful to create compact data structures. In this case the alignment will be one byte for objects and non-bit-field structure members.

  At the left side of a pointer declaration you can use the type qualifier `__unaligned` to mark the pointer value as potentially unaligned. This can be useful to access externally defined data. However the compiler can generate less efficient instructions to dereference such a pointer, to avoid unaligned memory access.

  You can always convert a normal pointer to an unaligned pointer. Conversions from an unaligned pointer to an aligned pointer are also possible. However, the compiler will generate a warning in this situation, with the exception of the following case: when the logical type of the destination pointer is `char` or `void`, no warning will be generated.

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE

Example:

```
struct
{
    char c;
    __unaligned  int i;   /* aligned at offset 1 ! */
} s;

__unaligned int * up = & s.i;
```

- `__packed__`

  To prevent alignment gaps in structures, you can use the attribute `__packed__` (introduced in v6.3r1). When you use the attribute `__packed__` directly after the keyword struct, all structure members are marked `__unaligned`. For example the following two declarations are the same:

```
struct __packed__
{
    char  c;
    int * i;
} s1;

struct
{
    char  __unaligned c;
    int * __unaligned i; /* __unaligned at right side of '*'
                            to pack pointer member        */
} s2;
```

  The attribute `__packed__` has the same effect as adding the type qualifier `__unaligned` to the declaration to suppress the standard alignment.

  You can also use `__packed__` in a pointer declaration. In that case it affects the alignment of the pointer itself, not the value of the pointer. The following two declarations are the same:

```
int * __unaligned p;
int * p __packed__;
```

  C compiler options **--align**, **--code-section-alignment** and **--data-section-alignment**

  To change the alignment of all sections within one C source file the C compiler option **--align** is applicable. The value must be a power of two or 0.

  For TriCore tools versions up to v6.0r1 the minimum size of a section whose section alignment can be changed using the **--align** option is 4 bytes. From v6.1r1 and up there is no minimum section size requirement anymore.

  To change the alignment of all code sections within one C source file the C compiler **option --code-section-alignment** is applicable. The value must be at least 2 and a power of two.

  To change the alignment of all data sections within one C source file the C compiler option **--data-section-alignment** is applicable. The value must be a power of two.

- C compiler option **--eabi**

  Some sub options of the EABI compliance option have an impact on the alignment. These are:

  half-word-align: When this option is enabled int values, and other basic types of four bytes, are aligned on halfword boundaries to reduce alignment gaps. This affects the alignment of struct members of 4-byte types too.

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE

Disabling this option is required for EABI v2.9 / v3.0 compliance.

Example:

```
struct {
    char c1;
    int i1;
} my_str;
```

Compiled using **--eabi=+half-word-align** will align the struct on a half word boundary and the struct member i1 will have an offset of two bytes. The struct size is 6 bytes.

Compiled using **--eabi=-half-word-align** will align the struct on a word boundary and the struct member i1 will have an offset of four bytes. The struct size is 8 bytes.

char-bitfield: This sub option was introduced in TriCore tools v6.1r1. When enabled a bit-field declared with base type char will be accessed using single-byte load and store instructions. This may result in additional padding to avoid crossing a byte boundary. Disabling this option is required for EABI v2.9 / v3.0 compliance.

Example:

```
struct bf1_t {
    int f1 : 3;
    char f2 : 8;
} bf1;
```

Compiled using **--eabi=+char-bitfields**: bit-fields f1 and f2 start at bit 0 and bit 3 respectively.

Compiled using **--eabi=-char-bitfields**: bit-fields f1 starts at bit 0 and f2 at bit 8.

word-struct-align: This sub option was introduced in TriCore tools v6.2r1. When enabled, a struct with a size larger than or equal 64 bits gets a minimum word alignment to guaranty penalty free performance of double word copy with instructions ld.d and st.d. This may result in additional padding to avoid crossing a word boundary. When disabled, the struct may start on a halfword address. Disabling this option is required for EABI v3.0 compliance.

bitfield-align: This sub option was introduced in TriCore tools v6.3r1. When enabled, a bit-field with a zero size will cause the next struct member to be aligned to a multiple of the size of the base type of the bit-field. When disabled, the next member will be aligned to the next byte boundary, as specified by the TriCore EABI.

ALIGNMENT OPTIONS PROVIDED BY THE LSL LINKER SCRIPT LANGUAGE

Instead of changing the section alignment by means of C compiler options or language extensions it is also possible to modify the alignment using the Linker Script Language (LSL). Some examples how to do this are listed below.

Change the alignment of all sections within one linker group

```
/* all individual sections selected in the group below will start on a 32-byte alignment */
group  (ordered, align=32)
{
    select ".bss.file_1.*";      /* select all sections starting with .bss.file_1. */
    select ".bss.file_2.*";      /* select all sections starting with .bss.file_2. */
    select ".bss.file_3.var_1";  /* select all sections with the name .bss.file_3.var_1 */
}
```

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE

To prevent the linker from filling gaps with other small sections which are not part of the group, the group definition can be changed into a sequential group by adding `contiguous` and `fill`.

```
group  (ordered, contiguous, align=32, fill) …
```

Change the alignment of the first section within one linker group

```
group (ordered, contiguous, fill)
{
    group ALIGNED_START (align=64) /* This section will be 64-byte aligned */
    {
        select ".bss.file_1.var_1";
    }
    group FOLLOW_ALIGN_GROUP
    {                              /* These sections will be placed behind the first section
                                      but do not have an alignment requirement */
        select ".bss.file_1.var_2";
        select ".bss.file_1.var_3";
    }
}
```

The same goal can be achieved using a sub group with a changed alignment:

```
group ALIGNED_GROUP (ordered, contiguous, fill, align=64)
{
    select ".bss.file_1.var_1";    /* This section will be 64-byte aligned */

    group FOLLOW_ALIGN_GROUP (align=0) /* align=0 switches back to the default alignment */
    {   /* These sections will have default alignment but follow the first section */
        select ".bss.file_1.var_2";
        select ".bss.file_1.var_3";
    }
}
```

Change the alignment of the first section within one output section

When an output section is used the alignment of the output section itself is defined by the align entry used for the group definition for the group which includes the output section. If the sections included in that output section do need to have a dedicated alignment a sub group can be created.

The example below is used to align the output section to start on a 64-byte boundary. All sections included in that output section will have a 8-byte alignment instead of the default alignment.

ALIGNMENT REQUIREMENTS
RESTRICTIONS FOR THE TRICORE ARCHITECTURE

```
group MY_DATA ( ordered, align=64 )
{
    section "aligned_output_section"(attributes=rw, size=0x1000)
    {
        group (align = 8)
        {
            select ".bss.file_1.*";  /* the sections included in this output section */
        }                             /* will have an 8-byte alignment */
    }
}
```

PRECAUTIONS (MIXED EABI ALIGNMENT SETTINGS, POINTER CASTS)

Prevent mixing EABI alignment settings

To prevent compatibility issues the **half-word-align** sub option setting of the **--eabi** option needs to be the same for all modules included in a project. Otherwise struct member accesses might go wrong when the struct is defined in a module with halfword alignment enabled and the access is made in a module where halfword alignment is disabled.

The same applies to the **char-bitfield**, **word-struct-align** and **bitfield-align** sub option settings of the **--eabi** option. The options should have the same values for the whole application.

Problems caused by using different settings for those options can be detected with global type checking, by specifying the C compiler option **--global-type-checking** or C compiler option **--debug-info** and the linker option --**global-type-checking** (or when you use MIL linking).

Also keep in mind that the option **--eabi-compliant** of the compiler is an alias for a set of **--eabi** option flags. To ensure compatibility with older toolset versions, the --eabi option flags char-bitfield (introduced in v6.1r1), **word-struct-align** (introduced in v6.2r1) and **bitfield-align** (introduced in v6.3r1) should not be disabled when you are cross-linking objects from older releases, neither directly, nor through the option--**eabi-compliant**.

Pointer casts

When a pointer to a `char` variable is casted to a pointer to an `int` this might go wrong when the `char` variable is located on an odd address.

```
unsigned char c_var;
unsigned int *uip;
unsigned int res;

void func(void)
{
    uip=(unsigned int*)&c_var;
    res = *uip;  /* This access might cause an odd address access bus trap when */
                 /* variable c_var is located at an odd address because the compiler uses */
                 /* ld.w/st.w instructions which can only access even addresses. */
}
```