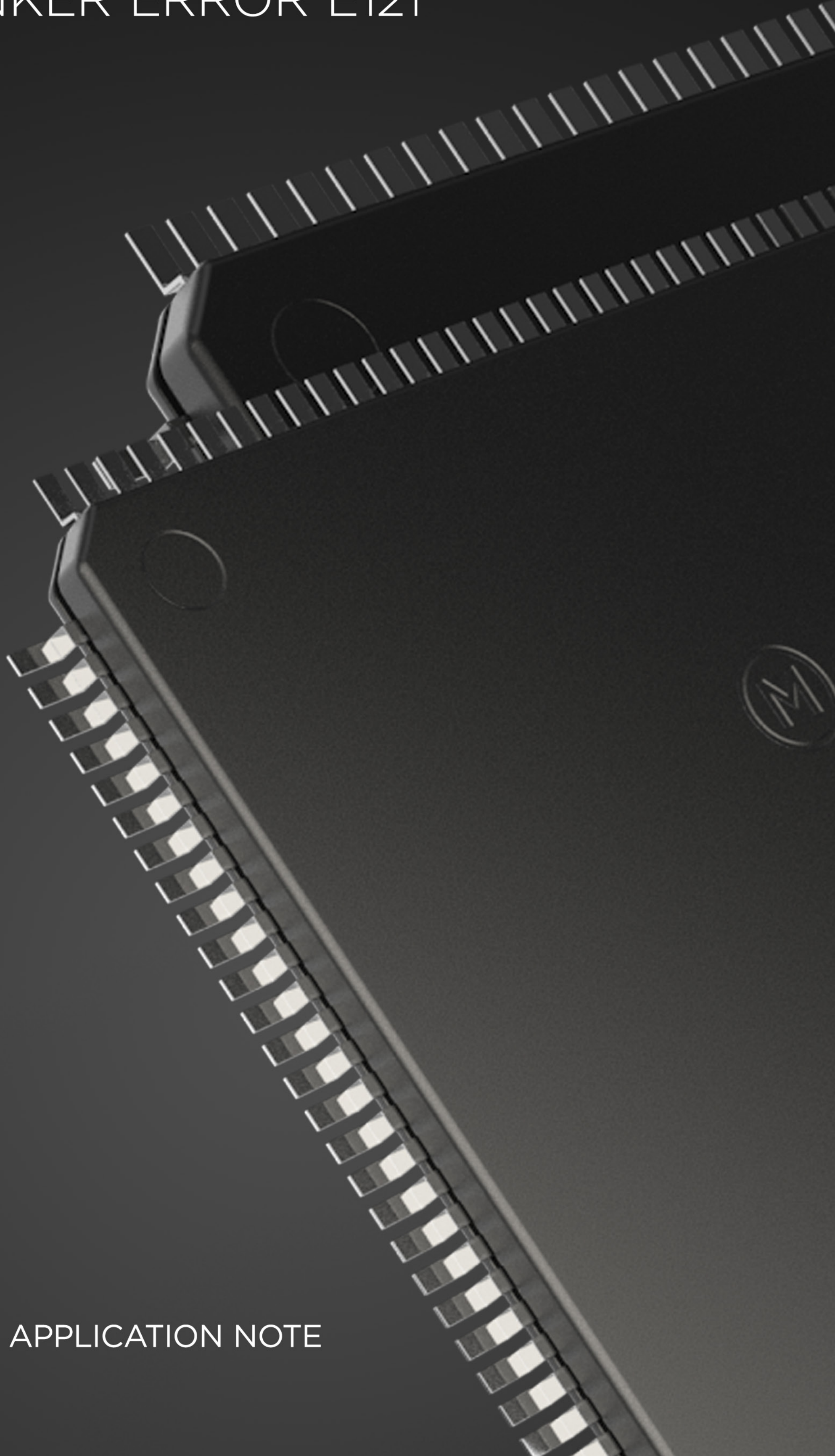
TASKING®

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121



APPLICATION NOTE

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

The linker error E121 is described as follows in the linker diagnostics documentation:

```
E121: relocation error in "<task>": <cause>
```

```
An error occurred while patching the result of the evaluated relocation
information into the data in the object file, for task <task>. The <cause>
explains the reason of the error.
```

This application note explains the most common reasons for E121 linker errors and how they can be mitigated.

If your use case is not listed in this application note, or you need more technical assistance, please contact our support staff at https://www.tasking.com/support.

E121 error caused by a mixture of near and far addressing

In the TriCore CPU near addressable data must be located within the first 16 kB of a 256 MB segment (offset 0x0 to 0x00003FFF within a segment). The E121 error is provoked when a variable is defined as a far addressable variable but an `extern` declaration is made for a near addressable variable instead. Then the linker may place the variable anywhere in memory. In the C modules where the variable is declared `extern` a near access is made which requires that the variable is placed within the first 16 kB.

Example:

```
/* file_1.c */
void func(void);
__far char var_1 __at(0xD0005000); /* __at is used to ensure the variable is placed
                                      outside of the near addressable range */

int main (void)
{
    func();
    return 0;
}

/* file_2.c */
extern __near char var_1;   /* __near is used, while it should have been __far */

void func(void)
{
    var_1 = 10;
}
```

Invocation:

```
cctc file_1.c file_2.c -o result.elf -t -v -s
```

Linker error:

```
ltc E121: relocation error in "task1": relocation value 0xd0005000, type abs18 space, offset 0x2,
section ".text.file_2.func" at address 0x8000036c is not a valid address in R_TRICORE_18ABS. Hint:
check the mapfile for a section that occupies this address.
```

Although in the above listed example the error looks obvious the problem can also be hidden. This is valid for a situation when different default near size options are used (C compiler option **-N**). This option is used to specify the maximum size of a data object for automatic near allocation. The default value is 8, which means that all data with a size lower or equal than 8 bytes are near addressed and as a result must be located in a near addressable memory range.

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

The problematic situation shows up when the C source file wherein the variable is defined is e.g. compiled using the option **-N0**. This forces the compiler to access all data which is not explicitly defined / declared using a `__near` qualifier using a far addressing mode. When the C source files which include the 'extern' declarations for that variable are compiled using the option **-N8** (or the option is not used at all which is equal to using **-N8**) then the access to the variable will be a near access. A modified example is:

```
/* file_1.c */
void func(void);
char var_1 __at(0xD0005000); /* __at is used to ensure the variable is placed
                                outside of the near addressable range */

int main (void)
{
    func();
    return 0;
}


/* file_2.c */
extern char var_1;

void func(void)
{
    var_1 = 10;
}
```

Invocation:

```
cctc file_1.c -o file_1.o -N0 -t -v -s -co
cctc file_2.c file_1.o -o result.elf -N8 -t -v -s
```

Here `__near` and `__far` are not used in the source code, but the problem is the same due to the different default near size values used with the **-N** option.

It is possible to overrule the **-N** option locally using a pragma. E.g.:

```
#pragma for_extern_data_use_memory __far
```

Then all `extern` declared variables following that pragma will be far accessed, independent of the **-N** value used.

This kind of E121 error can also show up when instead of a near/far address mixture a different, non- far address method like A0, A1, A8 or A9 addressing is used. Using the address register, indexed access requires the data to be located within +/- 32 kB of the address the address register points to. E.g. when the A0 register content is 0xD0018000 then the address range where all A0 addressed variables must be placed in is 0xD0010000 to 0xD001FFFF. When an A0 access is made to a variable placed outside of the range the linker error E121 is generated:

```
ltc E121: relocation error in "task1": relocation value 0xd0005000, type R_TRICORE_16SM, offset 0x0,
section ".text.file_2.func" at address 0x8000036c is not within a 16-bit signed range from the value
of A0 as defined by the symbol _SMALL_DATA_
```

TASKING

APPLICATION NOTE

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

E121 error when a function is called in a different segment starting at an address above a 2 MB segment offset

The TriCore CPU supports different function call instructions like absolute and relative calls or an indirect call. An indirect call is only used when the `__indirect` language extension keyword is applied in the function definition/declaration or when the C compiler option **--indirect** is used. An indirect call is less efficient because it requires two instructions instead of one.

When a linker error like:

```
ltc E121: relocation error in "task1": relocation value 0x80616ec2, type rel24 or abs24, offset 0xc2,
section ".text.file_1.func_1" at address 0x50100cb4 is not a valid address in R_TRICORE_24REL. Hint:
check the mapfile for a section that occupies this address.
```

is generated, this indicates a problem related to the placement of the called function. When the caller is located in a different 256 MB segment than the callee, this can work only when an absolute call (`calla`) is used and the callee is located within the first 2 MB of a segment. This is a limitation of an absolute call which can only access functions located within the first 2 MB. A relative call is not an option in this situation because this can only be used for calls within one segment, since the call distance is +/- 16 MB.

For the error message shown above the caller is located in segment 5 and the function call instruction itself which causes the trouble is located at offset 2 in section `.text.file_1.func_1` at the absolute address 0x50100cb4. The callee is located in segment 8 at address 0x80616ec2 and this is outside of the 2 MB offset which is in the range from 0x80000000 to 0x801FFFFF.

Possible mitigations to prevent this linker error:

1. Use an indirect call for the function located at address 0x80616ec2:

   ```
   __indirect void func(void) {…}
   ```

2. Determine the section name of the function which is placed at 0x80616ec2 and introduce a linker LSL file group wherein this section is selected and assigned to a group which must be placed within the first 2 MB of the segment. E.g.:

   ```
   group FUNCTIONS_IN_2MB_OFFSET ( run_addr=mem:mpe:pflash0[0x0..0x1FFFFF], ordered )
   {
       select ".text.file_1.func_2";
   }
   ```

   This will place the section named `.text.file_1.func_2` somewhere in the first 2 MB of `pflash0` memory.

3. Enable the C compiler option ‘long branch veneers’. This is supported since v6.3r1 of the TASKING TriCore tools. The option is described as follows:

   Long Branch Veneers (a.k.a. Trampolines)

   With the new linker option **--long-branch-veneers** the linker generates a so-called veneer (a.k.a. trampoline) if the target of a 24-bit PC-relative call instruction is out-of-range. The call instruction is replaced by an absolute call to the veneer. The veneer makes an indirect call to the original call target. Veneers will be located in the "abs24" address space. The locating process of the linker may become less efficient when this feature is used, even if no trampolines are required. Therefore it is better to first see if out-of-range calls are in the code (unlikely) before switching on this option. Out-of-range calls are reported by the linker by means of an error message.

   To control the location of veneer code a new LSL syntax element is supported: `veneer_layout`. The specification of the address space to which a `veneer_layout` definition applies works in the same way as the specification of the address space for a `section_layout` definition.

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

This type of E121 problem can also show up when the startup code is located in a different segment than the `main` function and the `main` function starts at an offset above 2 MB.

Example: the startup code is located in cached segment 0x8 and the main function is located starting at address 0xa0200000. Then the linker error:

```
ltc E121: relocation error in "task1": relocation value 0xa0200000, type rel24 or abs24, off-
set 0x178, section ".text.cstart._start" at address 0x80000330 is not a valid address in R_TRI-
CORE_24REL. Hint: check the mapfile for a section that occupies this address.
```

is generated. A similar situation is when the reset vector is e.g. placed at 0xa0000020 and the startup code is placed in segment 0x8 starting at an offset above 2 MB. Then the error message is:

```
ltc E121: relocation error in "task1": relocation value 0x802461b6, type rel24 or abs24, offset 0x0,
section ".text.libc.reset" at address 0xa0000020 is not a valid address in R_TRICORE_24REL. Hint:
check the mapfile for a section that occupies this address
```

To mitigate this without being forced to place the startup code within the first 2 MB you can slightly modify the C startup code (`cstart.c`):

```
#ifdef NO_PFLASH0
    static void __noinline__ __noreturn__ __jump__ __used__ __init_sp( void );
#else
    static void __noinline__ __noreturn__ __jump__ __init_sp( void );
#endif
...
#ifdef NO_PFLASH0
    static void __noinline__ __noreturn__ __jump__ __used__ __init_sp( void )
#else
    static void __noinline__ __noreturn__ __jump__ __init_sp( void )
#endif
...
void _START( void )
{
#ifdef NO_PFLASH0
    __asm(" movh.a a15,#@his(__init_sp) \n"
          " lea a15,[a15]@los(__init_sp) \n"
          " ji a15 \n");
#else
    __init_sp();
#endif
}
```

E121 error after migrating from a version 5.0 or below to v6.0 or up

After a project migration to a newer version v6.x of the TriCore tools a linker error like:

```
ltc E121: relocation error in "task1": relocation value 0x90003000, type R_TRICORE_16SM, offset
0x104, section ".text.file_1.func_1" at address 0x80020db0 is not within a 16-bit signed range from
the value of A0 as defined by the symbol _SMALL_DATA_
```

appears.

This is caused by a change in the C compiler for an extended functionality for base registers A0 and A1, which is a result of an EABI v2.9 change. See EABI v2.9 chapter 2.2.1.4 and 4.2.3 for details. A corresponding change in the project LSL file is missing.

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

The section names for variables declared with `__a0` and `__a1` have changed. As well as the fact that `__a1` can be declared without `const`.

Register	Example	Before EABI v2.9 (old)	Since EABI v2.9 (new)
A0	`int const __a0 var;`	`.sbss`	`.bss_a0`
A0	`int const __a0 var = 1;`	`.sdata`	`.rodata_a0, rom`
A0	`int       __a0 var;`	`.sbss`	`.bss_a0`
A0	`int       __a0 var = 1;`	`.sdata`	`.data_a0`
A1	`int const __a1 var;`	`.ldata`	`.bss_a1`
A1	`int const __a1 var = 1;`	`.ldata, rom`	`.rodata_a1, rom`
A1	`int       __a1 var;`	error	`.bss_a1`
A1	`int       __a1 var = 1;`	error	`.data_a1`

The `#pragma section` now supports for all 4 base registers the type: `a<n>data`, `a<n>rom` and `a<n>bss`.
The behavior is conform the above table.

To mitigate this problem you can search the LSL file used in your project for the string `"(.sdata|.sdata.*)"`.
This should show up in the `group a0` definition. For v6.0 and up compliance, you can change the group definition to:

```
group a0 (ordered, contiguous)
{
    select "(.sdata|.sdata.*)";
    select "(.sbss|.sbss.*)";
    select "(.data_a0|.data_a0.*)";
    select "(.bss_a0|.bss_a0.*)";
    select "(.rodata_a0|.rodata_a0.*)";
}
```

To adapt the a1 group you can search for `"(.ldata|.ldata.*)"` and change the `group a1` entry to:

```
group a1 (ordered, contiguous)
{
    select "(.ldata|.ldata.*)";
    select "(.data_a1|.data_a1.*)";
    select "(.bss_a1|.bss_a1.*)";
    select "(.rodata_a1|.rodata_a1.*)";
}
```

Then all possible section prefixes for a0/a1 addressed data sections are assigned to the a0/a1 group.

E121: relocation error when a function is placed in an initialized output section without fill

This is a known problem in the TASKING tools v4.0r1 to v6.2r2. It is fixed in v6.3r1. The issue description is:

TCVX-43335 Linker error E121: relocation error when a function is placed in an initialized output section without fill

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

DESCRIPTION

When a function is placed in an initialized output section, the linker stops with an E121 relocation error when this function calls a function in flash memory and the initialized output section does not use the 'fill' qualifier to fill the non-occupied bytes with a pattern.

Example:

```
/* file_1.c */
int var_1;

int func_1(void);
void func_2(void);

/* func_1 is located in PSPR0 RAM */
int func_1(void)
{
    func_2(); /* this function call fails due to a call instead of a calla used */
    return 1;
}

/* func_2 is located in pflash */
void func_2(void)
{
}
int main(void)
{
    var_1 = func_1();
    func_2();
    return 1;
}
```

Linker LSL file:

```
/* link.lsl */
section_layout :vtc:linear
{
    group CORE0_FUNCTION (contiguous, ordered, run_addr = mem:mpe:pspr0, align=4, copy)
    {
        section "CORE0_FUNCTION_RAMCODE" (size=0x20, attributes=rwx)
        {
            select ".text.file_1.func_1";
        }
    }
}
```

Invocation:

```
cctc file_1.c -Ctc27x -Wl-dlink.lsl -t -v
```

HOW TO MITIGATE THE TASKING TRICORE TOOLS LINKER ERROR E121

Generated linker error:

```
ltc E121: relocation error in "task1": relocation value 0x80000024, type rel24 or abs24, offset 0x0, section ".text.file_1.func_1" at address 0x70100000 is not a valid address in R_TRICORE_24REL. Hint: check the mapfile for a section that occupies this address.
```

MITIGATION

Add the 'fill' entry to the output section entry in the LSL file:

```
section "CORE0_FUNCTION_RAMCODE" (size=0x20, attributes=rwx, fill=0xAA)
```

Note: The mitigation is only applicable when a 'size' instead of a 'blocksize' entry is used for the output section. When a 'blocksize' entry is used with a value smaller than the overall output section size a different linker error shows up which is related to issue TCVX-43233.