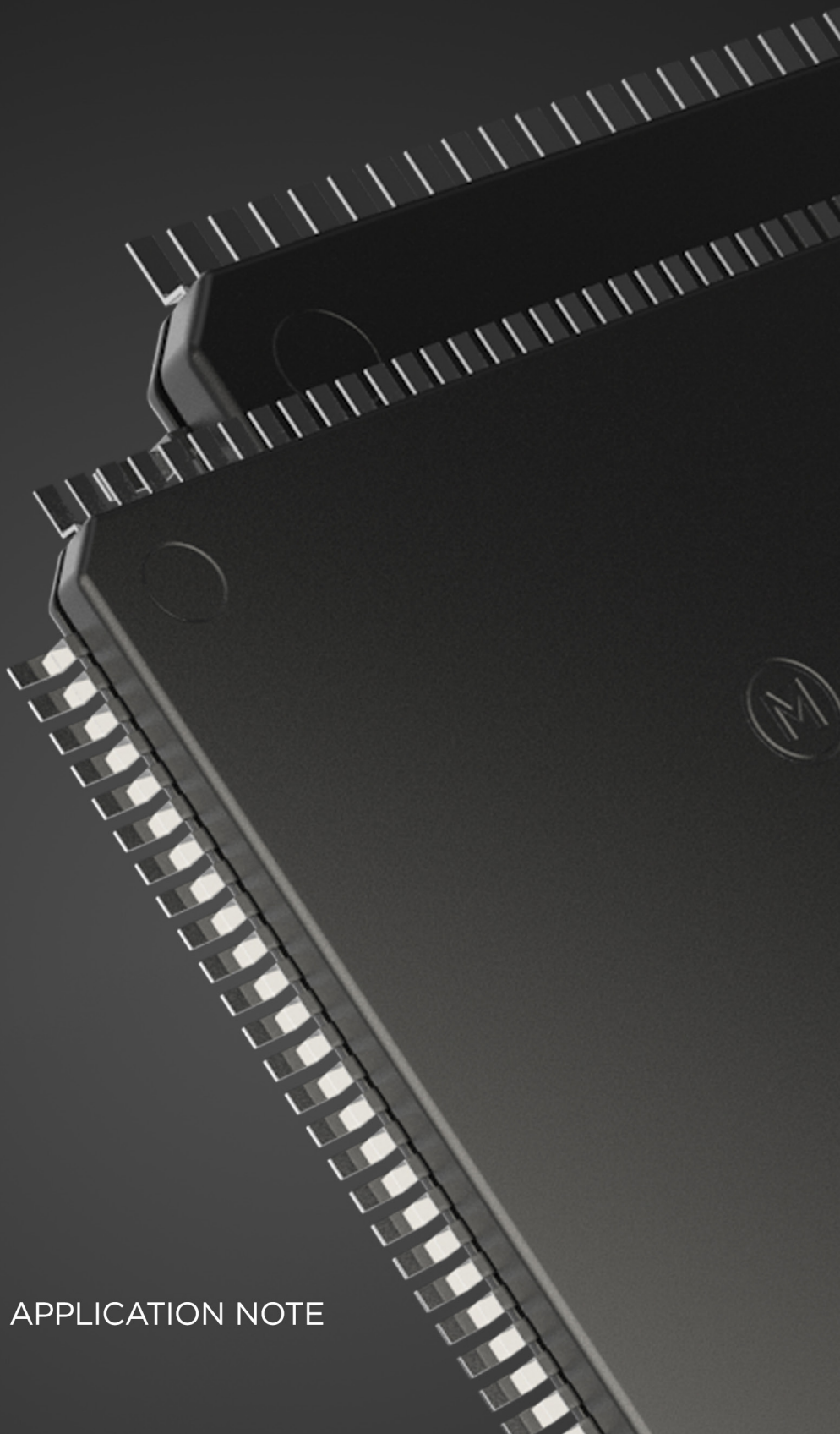
TASKING®

LINKER SCRIPT LANGUAGE
TIPS & TRICKS FOR TASKING TRICORE TOOLSET



APPLICATION NOTE

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

INTRODUCTION

The TASKING® Linker/Loader in the VX-toolset for TriCoreTM is a very feature rich tool, offering many helpful functionalities to create an efficient application. For example, it provides optimization options, it supports multi-core development and incremental linking, and it can be steered through an advanced Linker Script Language (LSL). This LSL is a very powerful language, used by many of our customers to optimize their application to their specific needs.

This document provides several helpful tips and tricks to help you benefit from the opportunities that LSL provides. Our support staff has worked with many of our users on LSL tuning and collected frequently used hints for this tips and tricks overview. We believe you can also learn from these hints and become an LSL expert. If you have any questions or maybe other suggestions/hints, please don't hesitate to contact us through support@tasking.com.

For detailed information about the linker and LSL we refer to the User Manual, chapter 7 Using the Linker, 10.5 Linker Options and 16 Linker Script Language (LSL).

LSL CODE CORE ASSOCIATION AND DATA CORE ASSOCIATION

The TASKING TriCore tools feature the language extensions `__share`, `__private0`, `__private1`, `__private2` and `__clone` to assign a code section or data section to core local memory. You can achieve the same goal with the following pragmas:

```
#pragma code_core_association share | private{012} | clone
#pragma data_core_association share | private{012} | clone
```

This method influences the section name that the linker uses to assign the section appropriately. To do so, the linker adds the core association name to the section name. E.g. `.bss.private1.`*module_name.variable_name* for a non initialized variable which must be placed in core 1 local memory.

When the C source code is not available (e.g. because a 3rd party library is used or an object file is already certified and thus the C source may not be modified) the core assignment using the above mentioned C language extensions or the pragma is no longer possible. To assign a section to a different core, you can use the linker LSL language keyword 'modify input'.

For the following C code, compiled with default options but with **-N0** for no default near allocation of variables, the variable `my_var_1` will be placed in section `.bss.file_1.my_var_1` and the code for `init_func` will be placed in section `.text.file_1.init_func`.

file_1.c

```
int my_var_1;
void init_func(void);
void main(void)
{
	init_func();
	while(1)
	{
		__nop();
	}
}
void init_func(void)
{
	my_var_1 = 0x1234;
}
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

To move the function init_func from flash memory to core 0 local RAM memory to be executed by core 0 only, you can use the following LSL file entries. The startup code takes care of copying the flash image of the function code to the core 0 local RAM:

```
// section_setup entry for the source location (here virtual linear memory, vtc)
section_setup mpe:vtc:linear
{
        // Link time code core association private to assign the section to
        // core0 memory
        modify input ( space = mpe:tc0:linear )
        {
                select ".text.file_1.init_func";
        }
}

// section_layout entry for the placement of the section in TC0 local PSPR0 memory
// and the 'copy' keyword for the linker to create a ROM copy section
// for the initialized code
section_layout :tc0:linear
{
      group TC0_FUNCTIONS ( ordered, run_addr=mem:mpe:pspr0, copy )
      {
              select ".text.file_1.init_func";
       }
}
```

The linker generated map file shows the result.

ROM copy of the initialized section in flash memory:

```
+ Space mpe:vtc:linear (MAU = 8bit)

+-------------------------------------------------------------------------------------------------------------------------------+
| Chip                | Group         | Section                          | Size (MAU) | Space addr | Chip addr  | Alignment  |
|===============================================================================================================================|
...
| mpe:pflash0         |               | [.text.file_1.init_func] (367)   | 0x0000000e | 0x800000ac | 0x000000ac | 0x00000002 |
...
```

Placement in core local PSPR0 memory:

```
+ Space mpe:tc0:linear (MAU = 8bit)

+-------------------------------------------------------------------------------------------------------------------------------+
| Chip                | Group         | Section                          | Size (MAU) | Space addr | Chip addr  | Alignment  |
|===============================================================================================================================|
...
| mpe:pspr0           | TC0_FUNCTIONS | .text.file_1.init_func (169)     | 0x0000000e | 0x70100000 | 0x0        | 0x00000002 |
...
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

OVERFLOW KEYWORD FOR DISTRIBUTED OUTPUT SECTIONS

Output sections are used to reserve a dedicated memory range for selected sections. Suppose, for example, that you have different memory regions available like DSPR0, DSPR1 RAM for an AURIX CPU and the selected sections might not fit into one output section. Then you can define an overflow section that is only created when the 'main' section size is not large enough to include all assigned sections. The size of the overflow output section can be static or adaptive using a blocksize entry, which increases the size of the section by the blocksize value if needed to store all sections.

You can apply the the size keyword to an output section as well as to an overflow output section. But the blocksize keyword can be applied only to a non-overflow output section.

```
/* Create some arrays with an overall size larger than the output section to provoke the need for
the overflow section. */
unsigned int i_arr_1[0x800];
unsigned int i_arr_2[0x800];
unsigned int i_arr_3[0x800];
group ( ordered, run_addr = mem:mpe:dspr0 )
{
      /* Define output section BSS_DATA with a size of 16kB in memory DSPR0 and
       * assign the arrays defined in file_1.c to this section.
       * Use overflow output section BSS_DATA_OVERFLOW for sections
       * that do not fit in BSS_DATA. */
      section "BSS_DATA" (size=16k, attributes=rw, overflow =

"BSS_DATA_OVERFLOW")
      {
            select ".bss.file_1.i_arr*";
      }
}
group ( ordered, run_addr = mem:mpe:dspr1 )
{
      /* If the available space of 16kB is not enough for all sections starting
       * with the name .bss.file_1.i_arr
       * this overflow section with a size
       * of 8 kB is used for the remaining sections. */
      section "BSS_DATA_OVERFLOW" (size=8k, attributes=rw)
      /* Instead of using a fixed absolute size for the output section you can
       * use the blocksize keyword to specify an adaptive size.
       * Then the size of the output section increases to a multiple of
       * the blocksize value. */
/*    section "BSS_DATA_OVERFLOW" (blocksize=1k, attributes=rw) */
      {
      }
}
```

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-----------------------------------------------------------------------------------------------------------------------+
| Chip          | Group | Section                      | Size (MAU) | Space addr | Chip addr  | Alignment  |
|=======================================================================================================================|
| mpe:dspr1     |       | BSS_DATA_OVERFLOW (371)      | 0x00002000 | 0x60000000 | 0x0        | 0x00000002 |
| mpe:dspr0     |       | BSS_DATA (370)               | 0x00004000 | 0x70005000 | 0x00005000 | 0x00000002 |
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

OVERLAY KEYWORD FOR RUNNING CODE FROM RAM MEMORY

When you have a function that needs to be executed in RAM memory, a RAM-to-ROM copy loop is required to copy the function code from flash to the run-time address range. The startup code can do this automatically. For this purpose the 'copy' keyword is applied to the LSL group definition:

```
section_layout :vtc:linear

{
// generate a ROM copy section and a copy table entry for the startup code
// in order to copy the code from ROM to RAM during startup
       group INIT_CODE ( ordered, run_addr=mem:mpe:lmuram, copy )
       {
              select ".text.file_1.func_1";
       }
}
```

Based on this keyword the linker creates a copy table entry for the initialized function and this is processed by the init function, called by the startup code, to copy the code from ROM to RAM memory.

If the startup code does not perform the initialization, it should be done later by the application code, changing the 'copy' keyword to the 'overlay' keyword. The initialization process itself is achieved using so-called linker labels to determine the start and end address of the ROM copy group. Those labels are used in a copy loop executed by the application code.

The 'overlay' keyword is also used for a different purpose. If you need to have multiple sections share the same RAM memory region, use 'overlay' for this purpose. That use case is not addressed in this section.

```
section_layout :vtc:linear

{
        /* generate a ROM copy section for initialized code */
        group INIT_CODE (overlay, ordered, run_addr=mem:mpe:dspr0)
        {
        select ".text.file_1.func_1";
        }
        /* assign the ROM copy of the initialized section to a group to determine start and end address */
        group ROM_COPY_INIT_CODE (ordered, load_addr)
        {
              select ".text.file_1.func_1";
        }
}
```

To use linker labels in the C source code, extern declarations are added:

```
/* use a linker label to determine the start address of the group in flash */
extern __far unsigned short _lc_gb_ROM_COPY_INIT_CODE;
/* use a linker label to determine the end address of the group in flash */
extern __far unsigned short _lc_ge_ROM_COPY_INIT_CODE;
/* use a linker label to determine the start address of the overlay group in RAM memory */
extern __far unsigned short _lc_gb_INIT_CODE;
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

The copy loop to initialize the RAM function can look like:

```
/* load the start address of the ROM copy group into source pointer */
source = &_lc_gb_ROM_COPY_INIT_CODE;
/* load the start address of the RAM group into destination pointer */
dest = &_lc_gb_INIT_CODE;
/* initialize the RAM function using a copy loop */
for(loopcount=0; loopcount <
(&_lc_ge_ROM_COPY_INIT_CODE-&_lc_gb_ROM_COPY_INIT_CODE); loopcount++)
{
       *dest++ = *source++;
}
```

For the map file the overlay group shows up like a section, thus the group name is not listed in the group column:

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-------------------------------------------------------------------------------------------------------------------------------+
| Chip               | Group              | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment  |
|===============================================================================================================================|
| mpe:dspr0          |                    | INIT_CODE (364)                | 0x0000000e | 0x70000000 | 0x0        | 0x00000002 |
| mpe:pflash0        | ROM_COPY_INIT_CODE | [.text.file_1.func_1] (365)    | 0x0000000e | 0x80000024 | 0x00000024 | 0x00000002 |
```

PRECAUTIONS WHEN USING SECTION_SETUP KEYWORD

When you use an LSL group to select sections, e.g. for a dedicated placement in memory, it could happen that a section is not assigned as expected. Although group names are optional, consider to use them as a best practice, because the group name is shown in the map file when the assignment was successful. For example:

```
section_layout :vtc:linear
{
// group entry to place a non initialized far addressed data section in LMURAM memory
       group MY_DATA ( ordered, run_addr=mem:mpe:lmuram )
       {
              select ".bss.file_1.my_var_2";
       }
}

+ Space mpe:vtc:linear (MAU = 8bit)
+-------------------------------------------------------------------------------------------------------------------------------+
| Chip               | Group        | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment  |
|===============================================================================================================================|
|
...
| mpe:lmuram         | MY_DATA      | .bss.file_1.my_var_2 (170)     | 0x00000004 | 0x90000000 | 0x0        | 0x00000002 |
```

If you do not specify a group name, the section might be placed in the group's range by accident and this will most likely change after you have made modifications to the project's C sources. It is important to keep in mind that the section_layout entry determines which sections will be selected. For example:

```
section_layout :vtc:linear
{
       group MY_DATA ( ordered, run_addr=mem:mpe:lmuram )
       {
              select ".zbss.file_1.my_var_1";
       }
}
```

will not work because a `.zbss` section is a near-addressable section (abs18) whereas the `section_layout` is for the linear (far-addressable) range. Chapter '7.9.5. The Architecture Definition' of the *TriCore User Guide* includes a table about the address space relations.

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

To solve this you need to adapt the `section_layout` entry:

```
section_layout :vtc:abs18
{
// group entry to place a non initialized near addressable data section
// in LMURAM memory
       group MY_NEAR_DATA ( ordered, run_addr=mem:mpe:lmuram )
       {
              select ".zbss.file_1.my_var_1";
       }
}
```

```
+ Space mpe:vtc:abs18 (MAU = 8bit)
+-----------------------------------------------------------------------------------------------------------------------+
| Chip              | Group         | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment  |
|===================================================================================================================|
| mpe:lmuram        | MY_NEAR_DATA  | .zbss.file_1.my_var_1 (169)    | 0x00000004 | 0x90000000 | 0x0        | 0x00000002 |
```

USING LSL SYMBOLS

If an address of a variable or a function is not known by the application, because the variable or function is part of another application which is linked separately, you can use LSL symbols to specify those values during link time. Then object files do not need to be modified for this purpose.

```
/* var_1 defined in another application located at address 0x70001000 */
extern unsigned int var_1;
/* func_1 defined in another application located at address 0x80007000 */
void func_1(void);
void main(void)
{
       var_1 = 0x1234;
       func_1();
}
```

E.g.: Linker LSL file entry to assign var_1 and func_1 to the addresses:

```
section_layout :vtc:linear
{
       /* variable var_1 is included in another project and located at address
          0x70001000 */
       "var_1"  = 0x70001000;

       /* function func_1 is included in another project and located at address
          0x80007000 */
       "func_1" = 0x80007000;
}
```

You can make a symbol conditional. Then it is created only when it is referenced in an object file. For this purpose, use ':='. For example:

```
"func_1" := 0x80007000;
```

This example defines func_1 only when it is referred in an object file.

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

When you are debugging there is no debug information for those LSL file symbols. Thus you cannot add the variable to a 'Variables watch' view. But you can use a pointer dereference to address 0x70001000 to show the variable in a debugger, like this:

```
*(unsigned int *)0x70001000
```

in the 'Expressions' view of the TASKING C/C++ debugger.

USING THE RESERVED KEYWORD TO PREVENT SECTION PLACEMENT

By default, the linker places sections according to the 'priority' value entered for the memory block. Memories with a higher priority value will be filled up first. The memories will be loaded from lower to upper addresses. If a certain memory range should not be used, you can apply the 'reserved' keyword to prevent a section placement in that memory range. E.g.:

```
section_layout :vtc:linear
{
        group ( ordered, run_addr = mem:mpe:pflash0[0x100] )
        {
                reserved "MY_RESERVE" ( size = 16k );
        }
}
```

The above listed LSL entry reserves 16 kB in pflash0 memory starting at offset 0x100 in this memory range.

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-------------------------------------------------------------------------------------------------------------------+
| Chip               | Group        | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment|
|===================================================================================================================|
|
...
| mpe:pflash0        |              | MY_RESERVE (354)               | 0x00004000 | 0x80000100 |0x00000100  |0x00000001|

...
```

If you need to place a section or some sections in a reserved range, you can add an `alloc_allowed` entry to the reserve group. To allow absolute-placed sections use `alloc_allowed=absolute`. To allow placement of sections which are placed in a memory range use `alloc_allowed=ranged`.

```
section_layout :vtc:linear
{
        group ( ordered, run_addr = mem:mpe:pflash0[0x100] )
        {
                reserved "MY_RESERVE" ( alloc_allowed=absolute, size = 16k );
        }

        group SPECIAL_FUNCTIONS ( ordered, contiguous, run_addr = mem:mpe:pflash0[0x110] )
        {
                select ".text.file_1.func_1";
                select ".text.file_1.func_2";
        }
}
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

Here the `SPECIAL_FUNCTIONS` group is placed at the absolute start address offset 0x110 in memory pflash0 which is within the reserved range.

```
section_layout :vtc:linear
{
	group ( ordered, run_addr = mem:mpe:pflash0[0x100] )
	{
		reserved "MY_RESERVE" ( alloc_allowed=ranged, size = 16k );
	}

	group SPECIAL_FUNCTIONS ( ordered, contiguous, run_addr =
mem:mpe:pflash0[0x110..0x200] )
	{
		select ".text.file_1.func_1";
		select ".text.file_1.func_2";
	}
}
```

Here the `SPECIAL_FUNCTIONS` group is placed in an address range, starting at offset `0x110` and ending at offset `0x200` in memory `pflash0` which is within the reserved range.

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-----------------------------------------------------------------------------------------------------------------------+
| Chip       | Group            | Section                          | Size (MAU) | Space addr | Chip addr  |  Alignment  |
|=======================================================================================================================|
...
| mpe:pflash0 |                  | MY_RESERVE (371)                 | 0x00004000 | 0x80000100 | 0x00000100 | 0x00000001  |
| mpe:pflash0 |SPECIAL_FUNCTIONS | .text.file_1.func_1 (168)        | 0x00000008 | 0x80000110 | 0x00000110 | 0x00000002  |
| mpe:pflash0 |SPECIAL_FUNCTIONS | .text.file_1.func_2 (169)        | 0x00000008 | 0x80000118 | 0x00000118 | 0x00000002  |

| mpe:pflash0 |                  |.text._c_init_entry.libcs_fpu (222)| 0x00000120 | 0x80004100 | 0x00004100 | 0x00000002  |
```

DEDICATED PLACEMENT OF ROM COPY SECTIONS FOR INITIALIZED DATA

When a variable is initialized like:

```
int var_1 = 10;
```

the initialization value for this variable needs to be placed in flash memory. The copy table processed by the init function which is called by the C startup code includes information about:

a) Where this ROM copy section is located
b) To which address in RAM memory it needs to be copied
c) The number of bytes that need to be copied

More details about the C startup code are included in chapter '4.3. The C Startup Code' of the *TriCore User Guide* .

If you need to place those ROM copy sections in a dedicated memory range, the LSL language offers different approaches to solve this. One solution makes use of the `run_addr` keyword. The other uses `load_addr` instead.

The name of the ROM copy section is equal to the name of the RAM section, but the ROM copy section name is offset by square brackets [ ]. To select a ROM copy section you need to use the full name including the square brackets in the select line for an LSL group. In order to have them accepted by the linker you need to escape them using '\'.

For the above listed C source line, suppose the line is included in a file named `file_1.c` and the default near allocation value is set to zero to prevent any near addressable sections without using the `__near` qualifier, the default section name is `.data.file_1.var_1` which consists of:

```
<section name prefix>.<module name>.<variable name>
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

The section name prefixes used for the different types of sections are listed in chapter '1.12. Compiler Generated Sections' of the *TriCore User Guide.*

To select this section in an LSL group which is placed in ROM memory (for example, starting at offset 0x100 in `pflash0` memory) and to have the RAM section placed (for example: starting at offset 0x200 in DSPR0 RAM) you can use the following syntax:

```
section_layout :vtc:linear

{
        group MY_ROM_COPY_SECTIONS ( ordered, run_addr = mem:mpe:pflash0[0x100] )
        {
                select "\[.data.file_1.var_1\]";
        }
        group MY_INITIALIZED_SECTIONS ( ordered, run_addr = mem:mpe:dspr0[0x200] )

        {
                select ".data.file_1.var_1";
        }
}
```

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-------------------------------------------------------------------------------------------------------------------------------+
| Chip       | Group                   | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment  |
|===============================================================================================================================|
| mpe:dspr0  | MY_INITIALIZED_SECTIONS | .data.file_1.var_1 (170)       | 0x00000004 | 0x70000200 | 0x00000200 | 0x00000002 |
...
| mpe:pflash0| MY_ROM_COPY_SECTIONS    | [.data.file_1.var_1] (363)     | 0x00000004 | 0x80000100 | 0x00000100 | 0x00000002 |

…
```

As an alternate approach, instead of using run_addr in the group definition you can use the keyword `load_addr`. This informs the linker to select the ROM copy of initialized sections. When doing this, you need to remove the square brackets to select the section:

```
section_layout :vtc:linear

{
        group MY_ROM_COPY_SECTIONS ( ordered, load_addr = mem:mpe:pflash0[0x100] )
        {
                select ".data.file_1.var_1";
        }
        group MY_INITIALIZED_SECTIONS ( ordered, run_addr = mem:mpe:dspr0[0x200] )
        {
                select ".data.file_1.var_1";
        }
}
```

CHANGING SECTION ATTRIBUTES AT LINK STAGE / PREVENT INITIALIZATION OF SECTIONS

The C compiler adds section attributes to all code and data sections it creates. The available section attributes are:

r	readable sections
w	writable sections
x	executable sections
i	initialized sections
b	sections that should be cleared at program startup
s	scratch sections (not cleared and not initialized)
p	protected sections

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

When the C source code of a module is not available or may not be changed anymore by re-compiling it, but you need to change the section attributes or to disable the initialization of sections, you can solve this during the link stage. To prevent the initialization of a section you can use the group entry keyword 'nocopy'.

Possible link stage operations are:

- Prevent cleanup of non initialized variables by adding the 's' (scratch) attribute. Then the linker will not add a copy table entry for those sections and the startup code will not touch them.
- Prevent initialization of initialized variables by adding the 'nocopy' keyword. In this case the linker will not add a copy table entry for those sections and the startup code will not touch them. Furthermore, the linker will not create ROM copy sections including the initialization values.
- Prevent unreferenced sections from automated removal by adding the 'p' (protect) attribute. This is applicable when the linker optimization 'Delete unreferenced sections from the output file' is enabled, and although the section is not referenced, it should not be removed from the linker output.

For these purposes you can add an 'attributes' entry to a group definition in the LSL file. Listed below are two examples demonstrating the use of the 'nocopy' keyword and the 's' (scratch) attribute.

```
section_layout :vtc:linear
{
#ifdef DO_NOT_INITIALIZE_CALIB_VALUE
// nocopy is added here to remove the ROM copy section for this initialized data
// and the copy table entry
      group CALIBRATION_VALUE ( ordered, run_addr = mem:mpe:dspr0, nocopy )
      {
            select ".data.file_1.calib_value";
      }
#else
      group CALIBRATION_VALUE ( ordered, run_addr = mem:mpe:dspr0 )
      {
            select ".data.file_1.calib_value";
      }
#endif

#ifdef DO_NOT_CLEAR_VAR_1
// the scratch attribute 's' is added here to prevent a copy table entry
// to clear this section
      group VAR_1 ( ordered, run_addr = mem:mpe:dspr0, attributes = rws )
      {
            select ".bss.file_1.var_1";
      }
#else
      group VAR_1 ( ordered, run_addr = mem:mpe:dspr0 )
      {
            select ".bss.file_1.var_1";
      }
#endif
}
```

LINKER SCRIPT LANGUAGE (LSL)
TIPS & TRICKS FOR TASKING TRICORE TOOLSET

The following output shows part of the map file when the non initialized section `.bss.file_1.var_1` is cleared. The size of the copy table section 'table' is 0x90 bytes:

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-------------------------------------------------------------------------------------------------------------------------+
| Chip          | Group             | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment  |
|=========================================================================================================================|
...
| mpe:dspr0     | VAR_1             | .bss.file_1.var_1 (171)        | 0x00000004 | 0x70000004 | 0x00000004 | 0x00000002 |
...
| mpe:pflash0   |                   | table (368)                    | 0x00000090 | 0x8000002c | 0x0000002c | 0x00000004 |

…
```

The following output shows part of the map file when the non initialized section `.bss.file_1.var_1` is not cleared. This is done by defining the `DO_NOT_CLEAR_VAR_1` macro in the LSL file.

The size of the copy table section 'table' is 0x80 bytes which is smaller due to the fact that the clear entry for section `.bss.file_1.var_1` is not included anymore:

```
+ Space mpe:vtc:linear (MAU = 8bit)
+-------------------------------------------------------------------------------------------------------------------------+
| Chip          | Group             | Section                        | Size (MAU) | Space addr | Chip addr  | Alignment  |
|=========================================================================================================================|
...
| mpe:dspr0     | VAR_1             | .bss.file_1.var_1 (171)        | 0x00000004 | 0x70000004 | 0x00000004 | 0x00000002 |
...
| mpe:pflash0   |                   | table (368)                    | 0x00000080 | 0x8000002c | 0x0000002c | 0x00000004 |

…
```

Also check the map file about what happens to the ROM copy section for the initialized section `.data.file_1.calib_value` when the initialization is switched off. Then the ROM copy section `[.data.file_1.calib_value]` will disappear.

More Help?

Hopefully, you were able to learn some new tips to using the TASKING Linker/Loader and benefit from the opportunities that LSL provides. Once again, if you have any questions or maybe other suggestions/hints, please don't hesitate to contact us through support@tasking.com.