LSL Sample Cases using the Control Program

TASKING™
Embedded software development from Altium™

The aim of this document is to demonstrate the usage of the *Linker Script Language* while using the TriCore toolchain to locate your application. It contains examples that deal with the most commonly used aspects of the language. It first lists all questions and continues with the proposed solutions. Studying this document will take 2 hours of your time. You may skip this document if you intend to use EDE instead. In that case document "*LSL Sample Cases using EDE*" will suffice for most cases.

Each solution consists of a proposed section layout definition - which you must place in a user module *layout.lsl* - and an invocation of the control program. The control program invocation will use *tc1796.lsl* for derivative definition and *extmem.lsl* for external memory definition. You can find both in the product folder include.lsl.

You are encouraged to verify the results of the linker by checking the map file. You can also test most examples with CrossView Pro's simulator execution environment using the following command line invocation:

```
xfwtc example.elf -C tc1796 -tcfg tsim2.cfg
```

THE QUESTIONS...

1. How do I locate a function within a specific memory definition?
2. How do I locate a function at an absolute memory offset address?
3. How do I locate a function at an absolute address?
4. I have several functions that must be located within a specific memory definition. Their order must be the order in which they were defined plus that they must be contiguous. Possible alignment gaps between these sections may not be replaced with existing sections but should simply remain gaps. How do I do this?
5. I have a function that I want to run from RAM memory. I would like the startup code to load this function in RAM prior to entering the main program. How do I do this?
6. How do I locate a variable at a fixed address without using the __at() memory qualifier?
7. I have battery-backed data. I do not want the startup code to clear it because then it will also be cleared in case of sudden reboots which I do not want. For this data I want the linker to disable the clear attribute but how?
8. I know how I can address linker labels for the stack or heap. But how can I change their size and/or start address?
9. We have a project that because of its size and complexity has been split into several logical blocks, each having their own group of engineers. All blocks are delivered to us as libraries before finally locating the entire application. We want to avoid one group of engineers accidentally using resources from other groups and therefore want a mechanism that restricts memory use to well defined memory blocks per group of engineers. No one may enter the application block of someone else nor may a block itself exceed a certain size. Each application block must be extended with a fill pattern to meet this size. Is there a way?
10. Within the memory map of our application we have specific memory parts that must be reserved. No data may be located within this range other than data that is located by means of an absolute address. Is there a way to do this in the Linker Script Language?
11. For a self-designed processor we need a way to define a Scratch-Pad RAM. Since it may only be used for executing code we want to prevent the linker from using this RAM memory for standard application data. In a way it needs to be 'disabled' and only those groups that we explicitly locate within it should be allowed. What internal memory definition should we use?
12. We have two groups with sections requiring contiguous ordering. The groups themselves must be contiguous but not necessarily ordered. What section layout definition do you propose we use?

THE PROPOSED SOLUTIONS...

QUESTION 1

How do I locate a function within a specific memory definition?

Solution: Assume the following C-module:

```
#include <stdio.h>

extern void _lc_ub__text_test_Sample[];
extern void _lc_ue__text_test_Sample[];

void Sample(void)
{
    printf("_lc_ub__text_test_Sample: 0x%p\n",
            _lc_ub__text_test_Sample);
    printf("_lc_ue__text_test_Sample: 0x%p\n",
            _lc_ue__text_test_Sample);
}

void main(void)
{
    Sample();
}
```

Suppose you want to locate function *Sample* and suppose the C-module is called *test.c* then according to TriCore's section naming convention (See section 3.10, *compiler generated sections*) *Sample* will be assigned to the following code section:

```
".text.test.Sample"
```

Suppose *Sample* must be located within memory *ext_c* such as defined in *extmem.lsl* then the following section layout definition will accomplish this:

```
section_layout ::linear
{
    group ( run_addr = mem:ext_c )
    {
        select ".text.test.Sample";
    }
}
```

The following control program command line invocation locates the application:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 2

How do I locate a function at an absolute memory offset address?

Solution: Using the same C-module as with Solution 1 and assuming a memory offset of 32k the section layout definition becomes:

```
section_layout ::linear
{
     group ( ordered, run_addr = mem:ext_c[32k] )
     {
          select ".text.test.Sample";
     }
}
```

Note that in addition to the memory offset a new keyword is introduced called `ordered`. This keyword is required because a fixed memory offset address implies a restricted address and by default groups are *unrestricted*. The required control program command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 3

How do I locate a function at an absolute address?

Solution: Using the same C-module as with Solution 1 and assuming an absolute address of 0xa000c000 the section layout definition becomes:

```
section_layout ::linear
{
     group ( ordered, run_addr = 0xA000C000 )
     {
          select ".text.test.Sample";
     }
}
```

The required control program command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 4

I have several functions that must be located within a specific memory definition. Their order must be the order in which they were defined plus that they must be contiguous. Possible alignment gaps between these sections may not be replaced with existing sections but should simply remain gaps. How do I do this?

Solution: Assume the following C-module called *test.c*:

```
#include <stdio.h>

extern void _lc_ub__text_test_SampleB[];
extern void _lc_ue__text_test_SampleE[];

void SampleB(void)
{
   printf("_lc_ub__text_test_SampleB: 0x%p\n",
           _lc_ub__text_test_SampleB);
}

void SampleE(void)
{
   printf("_lc_ue__text_test_SampleE: 0x%p\n",
           _lc_ue__text_test_SampleE);
}

void main(void)
{
    SampleB();
    SampleE();
}
```

Suppose you want to locate functions *SampleB* and *SampleE* then from Solution 1 it follows that these are assigned to the sections listed below:

```
".text.test.SampleB"
".text.test.SampleE"
```

Further suppose function *SampleB* must be located before *SampleE* then the section layout definition becomes:

```
section_layout ::linear
{
    group ( ordered, contiguous, fill=0x55, run_addr = mem:ext_c )
    {
        select ".text.test.SampleB";
        select ".text.test.SampleE";
    }
}
```

Note the use of the `fill` keyword. It prevents the linker from substituting alignment gaps with unrestricted sections/groups. The required control program command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 5

I have a function that I want to run from RAM memory. I would like the startup code to load this function in RAM prior to entering the main program. How do I do this?

Solution: Assume the following C-module called *test.c*:

```
#include <stdio.h>

extern void _c_init(void);
extern void _lc_ub__text_test_Add[];

inline void PatchAdd2Sub(void)
{
    ((int*)_lc_ub__text_test_Add)[0] = 0x2080540B;
}

inline void ReinstallAdd(void)
{
    _c_init();
}

__indirect int Add(const int a, const int b)
{
    return a+b;
}

void main(void)
{
    if ( Add(1,1) == 2 )
    {
        if ( PatchAdd2Sub(), Add(1,1) == 0 )
        {
            if ( ReinstallAdd(), Add(1,1) == 2 )
            {
                puts("success!");
            }
        }
    }
}
```

For this example the aim is to have *Add* installed in the on-chip scratch-pad memory defined as csram in the derivative definition for `tc1796`. Note the use of the `__indirect` keyword which - in this case - is required because within the `tc1796` memory map the gap between external memory `ext_c` and internal memory `csram` exceeds 16 MB and can therefore not be resolved with a standard `callg` or `jg` instruction. Not using `__indirect` here will result in linker error messages. The proposed section layout definition:

```
section_layout ::linear
{
    group ( run_addr = mem:spe:csram, copy )
    {
        select ".text.test.Add";
    }
}
```

The `copy` keyword instructs the linker to create a copy section for the sections within its group. In this case *.text.test.Add* spawns into copy section *[.text.test.Add]* and runtime section *.text.test.Add* which is installed in `csram`. The copy section itself is still *unrestricted* and will be freely located in ROM memory. Note the use of `spe:csram` to uniquely identify `csram` within a

possible multi-core environment or from possible external memory definitions with the same name. The required control program command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 6

How do I locate a variable at a fixed address without using the `__at()` *memory qualifier?*

Solution: Assume the following C-module called *test.c*:

```
extern void _lc_ub__bss_test_MMRDeviceOne[];
extern void _lc_ub__bss_test_MMRDeviceTwo[];

__far unsigned MMRDeviceOne;
__far unsigned MMRDeviceTwo;

void main(void)
{
     while ( _lc_ub__bss_test_MMRDeviceOne != ((void*)0xA0080100) );
     while ( _lc_ub__bss_test_MMRDeviceTwo != ((void*)0xA0080200) );
     puts ("memory mapped devices have been properly located"):
}
```

In this example memory mapped registers *MMRDeviceOne* and *MMRDeviceTwo* are supposed to reside at addresses *0xA0080100* and *0xA0080200* respectively. If they are not, the program will lock at either the 1st or 2nd while-statement. Locating both memory mapped registers to their designated addresses requires the following section layout definition:

```
section_layout ::linear
{
     group ( ordered, run_addr=mem:ext_d[0x100] )
     {
          select ".bss.test.MMRDeviceOne";
     }
     group ( ordered, run_addr=mem:ext_d[0x200] )
     {
          select ".bss.test.MMRDeviceTwo";
     }
}
```

Note that memory-relative addresses are used. Instead you may also use the following equivalent that implements absolute addresses:

```
section_layout ::linear
{
     group ( ordered, run_addr=0xA0080100 )
     {
          select ".bss.test.MMRDeviceOne";
     }
     group ( ordered, run_addr=0xA0080200 )
     {
          select ".bss.test.MMRDeviceTwo";
     }
}
```

The required control program command line invocation in both cases:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl
     --section-per-data-object --verbose test.c
```

Note the use of compiler command line option `--section-per-data-object` which is to make sure each variable is assigned a unique section name rather than one data section per module.

QUESTION 7

I have battery-backed data. I do not want the startup code to clear it because then it will also be cleared in case of sudden reboots which I do not want. For this data I want the linker to disable the clear attribute but how?

Solution: Assume the following C-module called *test.c*:

```
#include <string.h>
#include <stdio.h>

char securedString[35];
extern void _lc_ub__text_libc[];

void ColdReboot(void)
{
     ((void(*)(void))_lc_ub__text_libc)();
}

void main(void)
{
     printf("%35s\n", securedString);
     strcpy(securedString, "All are clear, I alone am clouded");
     ColdReboot();
}
```

With every cold reboot - and the example repeatedly keeps doing so - the main program begins by printing the contents of *securedString*. Under normal circumstances this means nothing is printed because *securedString* has been cleared by the startup code. In this case *securedString* resides in battery backed memory (for simplicity's sake this is assumed to be memory *ext_d*) and a sudden reboot will therefore unwantingly clear its contents. To prevent this from happening use the following section layout:

```
section_layout ::linear
{
     group ( ordered, run_addr=mem:ext_d, attributes=ws )
     {
          select ".bss.test.securedString";
     }
}
```

Where `attributes=ws` assigns the scratch attribute to all sections within that group which effectively disables clearing. The example requires the following control program command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl
     --section-per-data-object --verbose test.c
```

QUESTION 8

I know how I can address linker labels for the stack or heap. But how can I change their size and/or start address?

Solution: Assume the following C-module:

```
#include <stdio.h>

extern void _lc_ub_ustack[];
extern void _lc_ue_ustack[];
extern void _lc_ub_heap[];
extern void _lc_ue_heap[];

void main(void)
{
    printf("_lc_ub_ustack = 0x%p\n", _lc_ub_ustack);
    printf("_lc_ue_ustack = 0x%p\n", _lc_ue_ustack);
    printf("_lc_ub_heap = 0x%p\n", _lc_ub_heap);
    printf("_lc_ue_heap = 0x%p\n", _lc_ue_heap);
}
```

Further suppose this application requires a relatively small stack size of 256 bytes which must be offset to 32k of memory definition *ext_d*. The heap - though not used - should be 2k in size and located at an offset of 40k within the same memory. Listed below is the proposed section layout definition:

```
section_layout ::linear
{
    group ( ordered, run_addr=mem:ext_d[32k] )
    {
        stack "ustack" ( size=256 );
    }
    group ( ordered, run_addr=mem:ext_d[40k] )
    {
        heap "heap" ( size=2k );
    }
}
```

If on the other hand the stack or heap location is *unrestricted* then the following section layout definition will suffice:

```
section_layout ::linear
{
    group stack "ustack" ( size=256 );
    group heap "heap" ( size=2k );
}
```

The usage of the `stack` or `heap` keyword within the section layout definition is different from that of the architecture definition (*tc_arch.lsl*). Within the architecture definition attributes such as minimum stack or heap size, stack or heap growth, ballooning and alignment are defined. The section layout allows you to modify or extend these with group properties. For more information about the `stack` or `heap` keyword within the architecture definition please refer section 8.5.3, *Defining Address Spaces in chapter Semantics of the Architecture Definition* of the Reference Manual. The same keywords within the section layout definition are explained in section 8.8.3, *Creating or Modifying Special Sections* in chapter *Semantics of the Section Layout Definition* of the Reference Manual.

The control program command line invocation for this example:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 9

We have a project that because of its size and complexity has been split into several logical blocks, each having their own group of engineers. All blocks are delivered to us as libraries before finally locating the entire application. We want to avoid one group of engineers accidentally using resources from other groups and therefore want a mechanism that restricts memory use to well defined memory blocks per group of engineers. No one may enter the application block of someone else nor may a block itself exceed a certain size. Each application block must be extended with a fill pattern to meet this size. Is there a way?

Solution:

Since there must be a way to distinguish sections from one group of engineers from another group of engineers the module names for each group must conform to a naming convention. The naming convention will then also be in the section names which simplifies the `select` statements for each group. Suppose our fictive company has developer teams in Spain, Denmark and Germany and therefore use the first characters to identify the country code followed by a dash and finally the module name:

```
countryCode-moduleName.c
```

What follows is a simplefied example of the real thing. Team Spain writes library module *es-module.c* as listed below:

```
#include <stdio.h>

void blockA(void)
{
    puts("Team Spain");
}
```

Which they archived as below:

```
cctc --default-near-size=0 --create=object --verbose es-module.c
artc -crv libes.a es-module.o
```

Team Denmark on the other hand writes *dk-module.c* as listed below:

```
#include <stdio.h>

void blockB(void)
{
    puts("Team Denmark");
}
```

Which they archived as below:

```
cctc --default-near-size=0 --create=object --verbose dk-module.c
artc -crv libdk.a dk-module.o
```

Team Germany has the task of linking the libraries from teams Spain and Denmark in the overall application program being *test.c* as listed below:

```
extern void blockA(void);
extern void blockB(void);

void main(void)
{
    blockA();
    blockB();
}
```

It was agreed beforehand that each team should not cross an application block size of 1k. It was also decided beforehand that the library delivered by team Spain should reside at address *0xa0000400* and team Denmark *0xa0000800*. And finally it was agreed that gaps should be filled with *0x55* to meet the required block size. The proposed section layout definition as used by the German team:

```
section_layout ::linear
{
    group ( ordered, run_addr=0xA0000400 )
    {
        section "es.text" ( size=1k, attributes=rx, fill=0x55 )
        {
            select ".text.es-*.*";
            select ".rodata.es-*";
        }
    }
    group ( ordered, run_addr=0xA0000800 )
    {
        section "dk.text" ( size=1k, attributes=rx, fill=0x55 )
        {
            select ".text.dk-*.*";
            select ".rodata.dk-*";
        }
    }
}
```

The control program command line invocation finally used by the German team has been listed below:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl
     --library=es --library=dk --verbose test.c
```

Note the use of the `--library` command line option to automatically select the libraries from teams Spain and Denmark. The use of output sections is rare and really only applies to cases such as this one. In almost all other cases a standard section layout definition with only groups or nested groups will do.

QUESTION 10

Within the memory map of our application we have specific memory parts that must be reserved. No data may be located within this range other than data that is located by means of an absolute address. Is there a way to do this in the Linker Script Language?

Solution: Assume the following C-module called *test.c*:

```
#include <stdio.h>
#include <stdlib.h>

__far volatile unsigned DACSample __at(0xA0088000);

static inline unsigned NewSample(void)
{
    return DACSample = rand();
}

void main(void)
{
    while (1)
    {
        printf("Current DAC Noice Sample: 0x%04X\n", NewSample());
    }
}
```

Suppose that memory range *0xA0088000* through *0xA00880FF* is reserved memory and may not be used for standard relocatable application data. Then the following section layout definition applies:

```
section_layout ::linear
{
    group ( ordered, run_addr = 0xA0088000 )
    {
        reserved "mmio" ( alloc_allowed=absolute, size = 256 );
    }
}
```

Note the use of `alloc_allowed=absolute` which allows absolute sections to be located within the reserved address range. Omitting these keywords will result in error messages. The required control program command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl --verbose test.c
```

QUESTION 11

For a self-designed processor we need a way to define a Scratch-Pad RAM. Since it may only be used for executing code we want to prevent the linker from using this RAM memory for standard application data. In a way it needs to be 'disabled' and only those groups that we explicitly locate within it should be allowed. What internal memory definition should we use?

Solution:

For cases like these keywords reserved ram must be used while declaring the memory type. Listed below is an extract from the internal memory definition `csram` as used in `tc1796.lsl`.

```
memory csram
{
        mau = 8;
        size = 48k;
        type = reserved ram;
        map (dest=bus:tc:fpi_bus, dest_offset=0xD4000000, size=48k);
}
```

The above definition defines the Scratch-Pad RAM (SPRAM) such as it resides on tc1796's program local memory bus (PLMB). This local bus has been excluded from `tc_arch.lsl` to simplify it. For this reason csram is directly mapped to the `tc:fpi_bus` bus.

QUESTION 12

We have two groups with sections requiring contiguous ordering. The groups themselves must be contiguous but not necessarily ordered. What section layout definition do you propose we use?

Solution: Assume the following C-module called *test.c*:

```
#include <stdio.h>

__far int a;
__far char b;
__far int c;
__far char d;

void main(void)
{
    printf( "&a = %p\n"
            "&b = %p\n"
            "&c = %p\n"
            "&d = %p\n"
            , &a, &b, &c, &d );
}
```

Then the following section layout definition can be used:

```
section_layout ::linear
{
     group P ( contiguous )
     {
          group C1 ( contiguous, ordered )
          {
               select ".bss.test.d";
               select ".bss.test.c";
          }
          group C2 ( contiguous, ordered )
          {
               select ".bss.test.b";
               select ".bss.test.a";
          }
     }
}
```

The parent group P locates child groups `C1` and `C2` in a single address range but not necessarily in the order such as defined above. Child group `C1` locates sections `.bss.test.d` and `.bss.test.c` sequentially as does child group `C2` with sections `.bss.test.b` and `.bss.test.a`. The required command line invocation:

```
cctc --output=example.elf --debug-info --cpu=tc1796
     --lsl-file=layout.lsl --lsl-file=extmem.lsl
     --section-per-data-object --verbose test.c
```