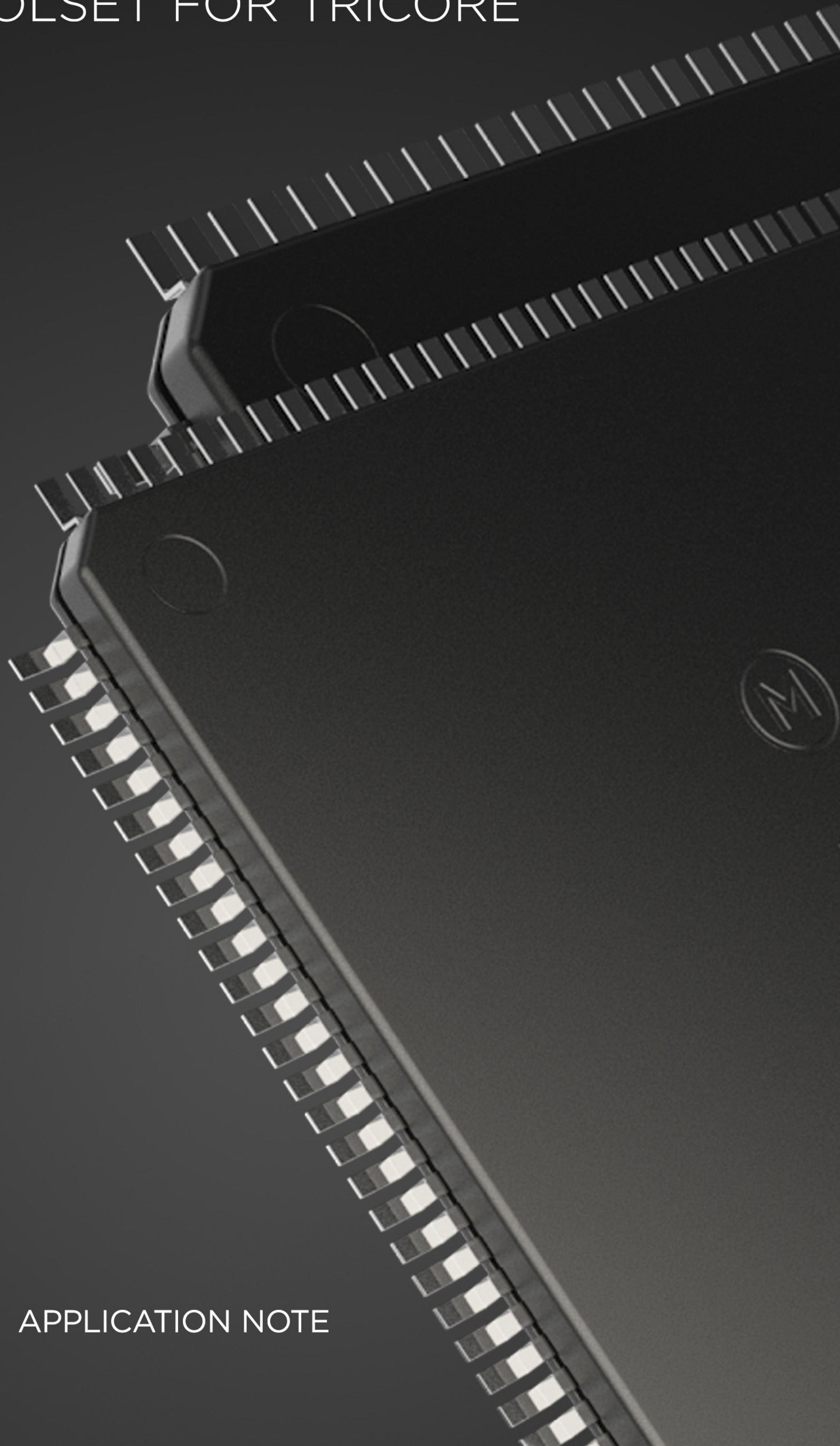
TASKING®

STACKS AND STACK SIZE ESTIMATION IN THE TASKING VX-TOOLSET FOR TRICORE



APPLICATION NOTE

STACKS AND STACK SIZE ESTIMATION IN THE TASKING VX-TOOLSET FOR TRICORE

The TriCore architecture defines two stacks: the user stack (`ustack`) and the interrupt stack (`istack`). Several TriCore devices have more than one TriCore core, each of which has its own pair of `ustack` and `istack`. In the TASKING VX-toolset for TriCore version v6.2r2 or older, the stack usage is calculated for a single stack for all cores. From TriCore toolset version v6.3r1 it is possible to calculate the stack usage for interrupt handlers and the stack usage for each core separately.

USER STACK AND INTERRUPT STACK

The TriCore architecture has one stack pointer register (A10). The user stack / interrupt stack switch is done by loading a different value into the stack pointer register.

To specify which stack is used, in the TriCore Eclipse IDE perform the following steps:

1. In an active project, from the **Project** menu, select **Properties for** <*project*>

2. Select **C/C++ Build » Startup Configuration**

3. Enable or disable the option **Use the user stack (clear PSW.IS)**

When a normal function is active bit `PSW.IS` is not set, A10 points to user stack. When an Interrupt Service Routine (ISR) is active bit `PSW.IS` is set, A10 points to the interrupt stack.

When an interrupt is taken and the interrupted task was using its private stack (`PSW.IS == 0`), the contents are saved with the upper context of the interrupted task and A[10](SP) is loaded with the current contents of the ISP.

When an interrupt or trap is taken and the interrupted task was already using the interrupt stack (`PSW.IS == 1`), then no pre-loading of A[10](SP) is performed. The ISR continues to use the interrupt stack at the point where the interrupted routine had left it.

Usually it is only necessary to initialize the ISP once during the initialization routine. However, depending on application needs, the ISP can be modified during execution. Note that there is nothing preventing an ISR or system service routine from executing on a private stack.

Note: Use of A[10](SP) in an ISR is at the discretion of the application programmer.

When the ISP is not initialized and an interrupt occurs, a bus trap is likely since it points to a range in segment zero.

STACK SIZE ESTIMATION IN THE LINKER SCRIPT LANGUAGE FILE

Because the compiler does not know what stack a function will use, nor on which cores a function will run, the linker must associate code with stack areas. This can be done through the linker script language (LSL).

In TASKING VX-toolset for TriCore version v6.3r1 and up, if a separate program is run on a specific core n, then the stack usage of this program can be computed separately by defining LSL macro `__USTACKn_ENTRY_POINTS` to the name of the symbol (between double quotes) that represents the main function for this program. `entry_points` statements are used for this. Multiple symbols can be specified by listing them between square brackets, separated by commas. Each symbol name must correspond to the caller name of a `.CALLS` directive as generated by the compiler.

STACKS AND STACK SIZE ESTIMATION IN THE TASKING VX-TOOLSET FOR TRICORE

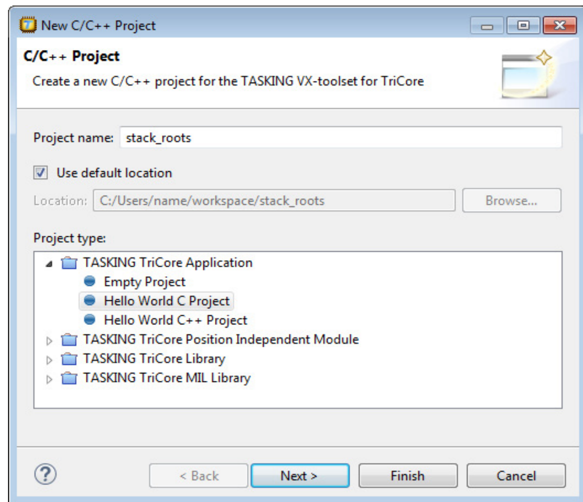
See the following snippet from `tc27xb.lsl` (located in directory `ctc\include.lsl`) for `ustack_tc0` and `istack_tc0`:

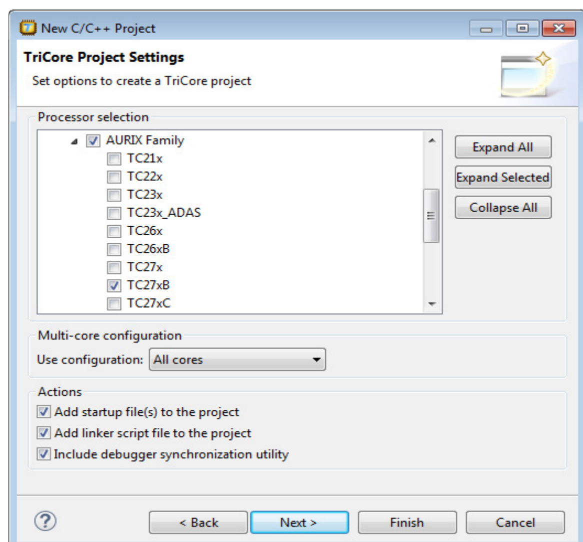
```
#ifndef USTACK_TC0
#define USTACK_TC0      16k                     /* user stack size tc0 */
#endif
#ifndef ISTACK_TC0
#define ISTACK_TC0      1k                      /* interrupt stack size tc0 */
#endif
#ifdef __USTACK0_ENTRY_POINTS
#define __USTACK0_ENTRY_POINTS_ATTRIBUTE ,entry_points=__USTACK0_ENTRY_POINTS
#else
#ifdef __NO_VTC
#define __USTACK0_ENTRY_POINTS_ATTRIBUTE ,entry_points=["_START" /* , group trap_tab, group int_
tab */ ]
#else
#define __USTACK0_ENTRY_POINTS_ATTRIBUTE ,entry_points=["_START" /* , group trap_tab_tc0, group
int_tab_tc0 */ ]
#endif
#endif
#ifdef __ISTACK0_ENTRY_POINTS
#define __ISTACK0_ENTRY_POINTS_ATTRIBUTE ,entry_points=__ISTACK0_ENTRY_POINTS
#else
#define __ISTACK0_ENTRY_POINTS_ATTRIBUTE
#endif

section_setup :tc0:linear
{
        stack "ustack_tc0"
        (
                min_size = (USTACK_TC0)
                ,fixed
                ,align = 8
                __USTACK0_THREADS_ATTRIBUTE
                __USTACK0_ENTRY_POINTS_ATTRIBUTE
        );
        stack "istack_tc0"
        (
                min_size = (ISTACK_TC0)
                ,fixed
                ,align = 8
                __ISTACK0_ENTRY_POINTS_ATTRIBUTE
        );
}
```

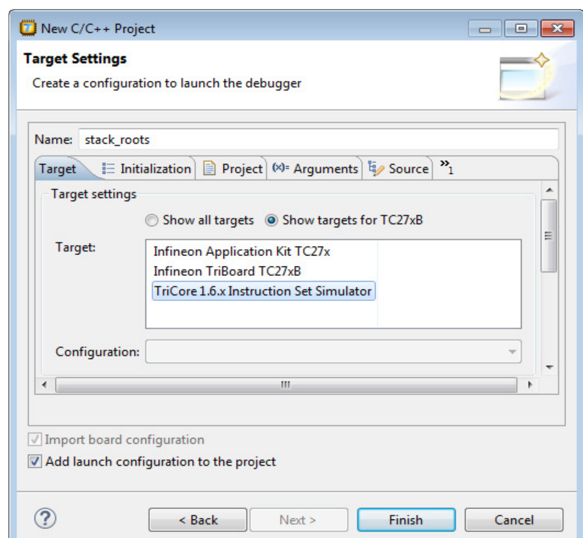
CREATE A MULTI-CORE PROJECT AND SPECIFY THE STACK ENTRY POINTS

The following example multi-core project shows you how to specify stack entry points for the core local user stacks `ustack_tc1` and `ustack_tc2`. In the TriCore Eclipse IDE v6.3r1 or higher, perform the following steps:

STACKS AND STACK SIZE ESTIMATION IN THE TASKING VX-TOOLSET FOR TRICORE







1. From the **File** menu, select **New » TASKING TriCore C/C++ Project**. The New C/C++ Project wizard appears.
2. Enter a name for your project, for example `stack_roots`.
3. In the Project type box, expand **TASKING TriCore Application** and select **Hello World C Project**. This creates the file `stack_roots.c` with a simple main function.
4. Click **Next**. The TriCore Project Settings page appears.
5. Select a multi-core processor. In this example we choose the TC27xB.
6. In the **Multi-core configuration** select **All cores**.
7. Enable all Actions checkboxes and click **Next**. The Target Settings page appears.
8. Select the simulator or a target board and click **Finish**.

Note that if you want to actually run a multi-core application, you need to select a target board, because the simulator has the restriction to only simulate core 0. For this example, we will not run the application, but only build it to demonstrate the stack usage.

9. Replace the contents of `stack_roots.c` with the following source:

```
#ifdef __CPU__
#include __SFRFILE__(__CPU__)
#endif

#define CORE __mfcr(CORE_ID)

int f1(int n)
{
     return n * 7 + 29;
}

void main_tc0(void)
{
     int     arr[28];
     int     i;

     arr[0] = 8;

     for (i = 1; i < 28; ++i)
     {
             arr[i+1] = f1(arr[i]);
     }
}
```

```
int f2(int n)
{
      return n * 5 + 83;
}


void main_tc1(void)
{
      int     arr[78];
      int     i;
      arr[0] = 194;
      for (i = 1; i < 78; ++i)
      {
              arr[i+1] = f2(arr[i]);
      }
}

int f3(int n)
{
      return n * 5 + 83;
}

void main_tc2(void)
{
      int     arr[23];
      int     i;
      arr[0] = 14;
      for (i = 1; i < 23; ++i)
      {
              arr[i+1] = f3(arr[i]);
      }
}

int main(int argc, char ** argv)
{
      switch (CORE)
      {
              case 0:
                      main_tc0();
                      break;
              case 1:
                      main_tc1();
                      break;
              case 2:
                      main_tc2();
                      break;
      }
      return 0;
}
```

STACKS AND STACK SIZE ESTIMATION IN THE TASKING VX-TOOLSET FOR TRICORE

10. Open file `stack_roots.lsl` and add the following two lines at the beginning of the file

```
#define __USTACK1_ENTRY_POINTS "main_tc1"
#define __USTACK2_ENTRY_POINTS "main_tc2"
```

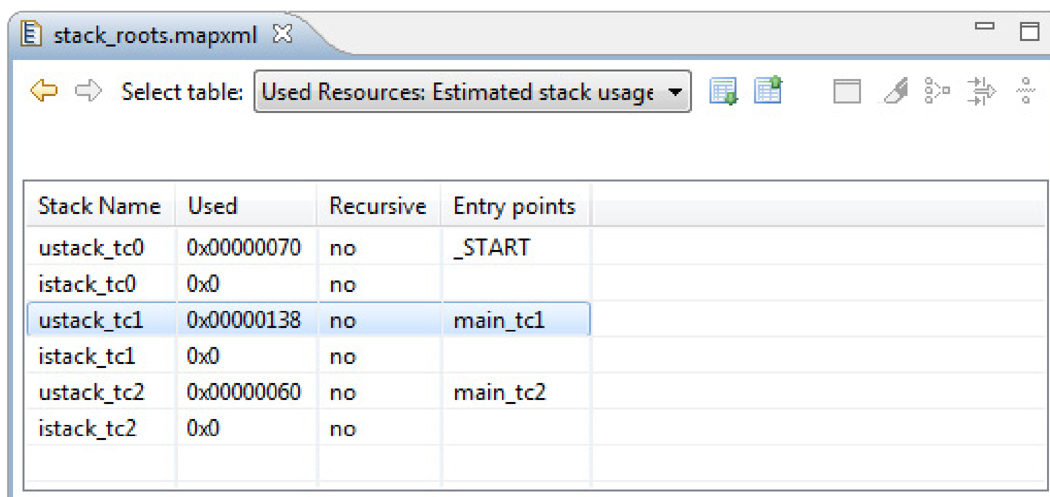
11. From the **Project** menu, select **Properties for stack_roots**, select **C/C++ Build » Startup Configuration**, and in the **core tc0** tab enable **Start TC1** and **Start TC2** and click **OK.** This will start the other cores from the main core 0.

Build the project

- From the **Project** menu, select **Rebuild stack_roots**. This creates files in the Debug folder of your project.

STACK SIZE ESTIMATION IN THE LINKER MAP FILE

1. From the `Debug` folder in your project, double-click on `stack_roots.mapxml` to open the map file.
2. From the **Select table** list, select **Used Resources: Estimated stack usage**, you will see results similar to this.



Stack Name	Used	Recursive	Entry points
ustack_tc0	0x00000070	no	_START
istack_tc0	0x0	no	
ustack_tc1	0x00000138	no	main_tc1
istack_tc1	0x0	no	
ustack_tc2	0x00000060	no	main_tc2
istack_tc2	0x0	no	

As you can see, apart from the default `ustack_tc0` and `istack_tc0`, there are now also stack estimations for the stacks `ustack_tc1` and `ustack_tc2`. Their entry points are now also visible in the call graph as root functions. This is a feature of the TriCore toolset v6.3r1 and up.

The **Used** column contains an estimation of the stack usage. The linker calculates the required stack size by using information (`.CALLS` directives) generated by the compiler. If for example recursion is detected, the calculated stack size is inaccurate; therefore this is an estimation only. The calculated stack size is supposed to be smaller than the actual allocated stack size. If that is not the case, then a warning is given.

The **Entry Points** column contains a list of entry points used for estimation of the stack usage.

Note: When you use an RTOS, the RTOS takes care of the stack handling and the calculated stack details the TASKING linker provides are not relevant anymore. Consult your RTOS documentation about the stack size you need to reserve in this situation.

STACKS AND STACK SIZE ESTIMATION IN THE TASKING VX-TOOLSET FOR TRICORE

STACKS IN OTHER ARCHITECTURES

The other architectures in the TASKING VX-toolset for TriCore use the following stacks:

- **C51:** 3 stacks: "stack", "vstack_xdata", and "vstack_pdata", but none of them are linked to stack usage info (static stack may be used)
- **PCP:** no stacks (static stack is used)
- **MCS:** "stack_0" .. "stack_7"
- **ARM:** "stack", "stack_fiq", "stack_irq", "stack_svc", "stack_abt", "stack_und"

For MCS the LSL macro `GTM_MCScore_STACK_channel_ENTRY_POINTS` specifies the entry points for stack estimation of the stack for the specified *channel* of the MCS *core*. By default the macro for the main channel is set to "`__START`".

For ARM the LSL macro `__STACK_ENTRY_POINTS` specifies the entry points for the stack. By default the value is set to "`__START`".