TASKING®

EMPOWERING YOUR CODE, PULSE BY PULSE.

Application Note

Technical Content Benchmarking Guideline

Table of Content

Technical Content Benchmarking Guideline

1. Introduction

1.1 Benchmarks

A benchmark is a standardized test used to evaluate and compare the performance of various computing systems, including processors and memory subsystems. Benchmarks are designed to emulate a set of operations or workloads providing quantifiable metrics that offer insights into the efficiency, speed, and capabilities of the system under test. They play a crucial role in performance analysis, enabling researchers, engineers, and developers to make the best decisions regarding hardware and software optimizations, system design, and purchasing. Benchmarks can be synthetic, focusing on specific types of operations, or application-based, simulating entire application scenarios. Their results aid in identifying bottlenecks, validating improvements, and ensuring that systems meet the required performance standards for intended applications.

1.2 The Dhrystone Benchmark

The Dhrystone benchmark is a synthetic computing benchmark program developed by Reinhold P. Weicker in 1984. It is widely utilized to measure and compare the performance of computing systems, particularly the CPU and memory subsystems. The Dhrystone benchmark operates by running a program that predominantly focuses on integer arithmetic, control statements, and string operations. The performance metric is based on the number of iterations per second of the main loop in the executed code. The results of the benchmarking are typically reported in DMIPS (Dhrystone Millions of Instructions per Second). 1 DMIPS was originally defined as the performance of one VAX 11/780 machine. By further dividing the DMIPS score by the CPU's clock speed (DMIPS/MHz), the resulting metric allows for a better comparison between processors running at different frequencies. Although considered outdated and often replaced by other benchmarks like CoreMark, Dhrystone is still commonly utilized in the industry as a preliminary measure of performance.

1.3 Ready benchmarks for SmartCode

A set of pre configured benchmarks is available to be instantly usable by SmartCode. The list of available benchmarks and the access to this benchmarks can be obtained with this link:

https://tasking.com/benchmark

2. Building an Executable - TASKING SmartCode toolset

2.1 Import the Dhrystone Project

Preconfigured Eclipse projects are available for various TC4x Starter Kits. To import a project into an Eclipse workspace please execute the following steps:

1. Open menu:

File >> Import >> General >> Existing Projects into Workspace

and select the project matching your available TC4x Starter Kit.

Ensure the checkbox 'Copy projects into workspace' is enabled when the project is imported.

2. After a successful import, you can decide which configuration you would like to build and execute it on the given target board. The active build configuration is selected using a right mouse click at the project name.

Within the menu that opens select:

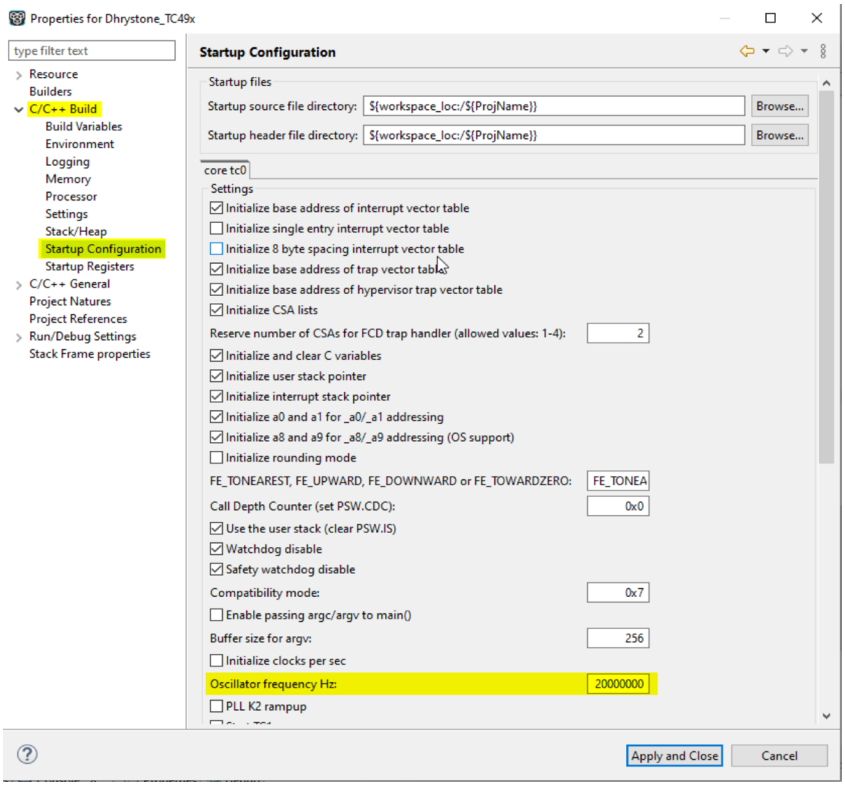
Build Configurations >> Set active

and chose the configuration to build.

3. Depending on the Target board you use a different value for the oscillator frequency needs to be filled in. The COM board uses an oscillator frequency of 25 MHz and the STD board uses 20 MHz. The oscillator frequency needs to be specified in menu:

Project >> Properties >> C/C++ Build >> Startup Configuration

under 'Oscillator frequency Hz':



4. Select:

Project >> Build Project

to build the project.

2.2 Preconfigured Build Configurations

The following setup is applicable to all use cases:

- The application is a single core application executed on core 0.
- The application code is in the cached memory region (segment 8) and the program and data cache is enabled.
- The writeable sections of the application (variables, stacks) are in the core local DSPR0 memory.
- The placement of the sections which belong to the benchmark code and data as well as the code and data sections of the utility functions (timer, startup code) and the library functions can be verified by reviewing the content of the map file. The 'group' column in the 'Locate result' section of the map file shows group names like APPLICATION_CODE, UTILITY_CODE to identify the sections which include functions of the application/benchmark code and the utility functions. The LSL file of the application (file extension .lsl) which is located in the root folder of the Eclipse project, is including the group definitions for those assignments.

The following build configurations do exist for different use cases:

2.2.1 Ground rule

This is the configuration according to the original Dhrystone benchmark requirements. Benchmark functions may not be inlined, and each C source file must be compiled separately. Important C compiler command line options used for this configuration:

-O3l	maximum optimization but no automatic inlining of small functions
-t0	size/speed tradeoff set to max speed
-N=0x2800	all data is accessed using near data accesses

2.2.2 Ground rule + inlining:

For this configuration, function calls within the same C source module are inlined.

Important C compiler command line options used for this configuration:

-O3	maximum optimization
-t0	size/speed tradeoff set to max speed
-N=0x2800	all data is accessed using near data accesses
--inline	Inline function calls within the same C source module

2.2.3 Ground rule + inlining + application wide optimizations:

For this configuration, all function calls are inlined and application wide optimizations are applied.

Important C compiler command line options used for this configuration:

-O3	maximum optimization
-t0	size/speed tradeoff set to max speed
-N=0x2800	all data is accessed using near data accesses
--inline	Inline function calls across the whole project
--mil	enable application wide optimizations

2.2.4 Ground rule + static + application wide optimizations:

For this configuration, application wide optimizations are applied and functions are defined as static which permits fast calls (fcall) instead of normal function calls.

Important C compiler command line options used for this configuration:

-O3l	maximum optimization but no automatic inlining of small functions
-t0	size/speed tradeoff set to max speed
-N=0x2800	all data is near
--mil	enable application wide optimizations
--static	all functions are static, use fcall instead of call instruction

3. Flash and Execute the Benchmark - winIDEA

3.1 Physical Connection

The host PC, running winIDEA, connects to the BlueBox via USB or Ethernet. The connection between the BlueBox and the target board varies based on the Debugger model. The IC7mini and IC7pro connect to the TC4x board using a ribbon cable and a DAP Debug Adapter (see Fig.1), while the IC7max requires an Active Probe for the connection (Fig.2).

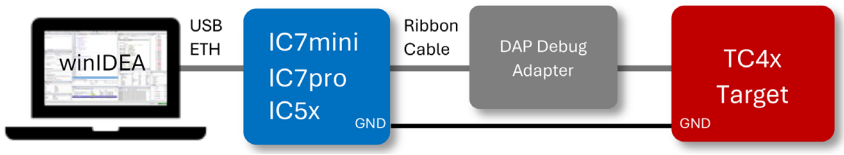


Fig.1: Target connection via DAP Debug Adapter

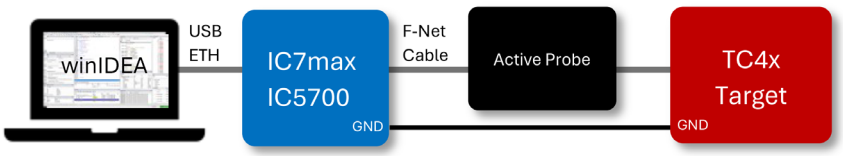


Fig.2: Target connection via Active Probe

3.2 Workspace Configuration

For each device (TC4Dx and TC49x), a separate winIDEA workspace has been prepared in the corresponding folder. After opening the workspace, the type of Blue Box used must be configured via the drop-down menu:

Hardware >> Debugger Hardware >> Hardware Type

winIDEA accesses the Blue Box either via USB or Ethernet. The configuration and communication test are performed by:

Hardware >> Debugger Hardware

If the communication test is successful, the Dhrystone executable can be downloaded to the target. By default, the workspace is configured to download the executable built for the ground rule configuration. The workspace can be reconfigured to download another executable, such as „Ground rule + inlining," to the target by:

Debug >> Configure Session >> Symbol file

Note: The Dhrystone Benchmark is not provided with any executable. Each configuration must be built by the user. A new subfolder will be created for every build configuration.

3.3 Executing the Benchmark

Once the executable is fully downloaded, CPU 0 should stop at its entry point.

Note: winIDEA attempts to locate the source files as they are referenced in the ELF file. If the source files are moved, winIDEA might have difficulty finding them. If winIDEA cannot locate a source file for any reason, you can manually specify the location by right-clicking on the missing file and selecting the option to locate it manually. For further information, please refer to our help document, "How to locate the source code"

By clicking the RUN button, the CPU will start running and execute the Dhrystone benchmark. The result is stored in the char array printf_buffer[ ] and can be read via the memory window.

This entire process is automated by a Python script, which starts the core, runs the benchmark, reads the result from the RAM, and stores it in a text file within the winIDEA workspace folder. The script is in the winIDEA workspace folder and can be executed via:

Tools >> External Scripts >> Dhrystone

The execution is performed by the interpreter provided with winIDEA. If another Python interpreter shall be used, the winIDEA SDK for Python must be installed. Please follow the instructions on our website:

Help >> SDK >> Download Python SDK

4. Tricore Compiler Configuration Hints To Maximize Execution Performance

4.1 Introduction

This abstract is about best practice hints to maximize the execution performance of an application or benchmark project. Its focus is on the most important tool settings which do help to increase the execution speed. Possible side effects which need to be considered are also referred to.

The execution performance of an application depends on multiple constraints like:

- Compiler optimization settings
- Placement of the application's code and data in selected memory regions of the CPU
- Alignment of code/data sections and jump labels for loops
- Compiler options used to build the Standard C and run-time libraries
- Access speed configuration of the target CPUs memory / program and data cache usage

4.2 Compiler Optimization Settings

The C compiler offers options to change the optimization level and specify a preference in favor of code size or speed optimization. The optimizations settings are defined using option

--optimize / -O

Four predefined optimization levels do exist. For maximum performance, optimization level 3 can be applied.

--optimize=3 / -O3

This does enable all supported optimizations.

Caveat: Using a high optimization level will make debugging of the application code less convenient. E.g. the C compiler might not generate debug information for all local variables. This can be depending on the usage and lifetime duration of a local variable. For the user, the generated assembly code might be harder to read when a high optimization level is used. Disabling all optimizations, however, is also decreasing the readability e.g. due to unnecessary data movement, which is removed by an optimization. The suggested optimization setting for debugging is optimization level 1

--optimize=1 / -O1

Independent of the optimization option, the toolset supports another option which is about changing the optimization focus to size or speed optimization. For maximum speed optimization, option:

--tradeoff=0 / -t0

can be applied.

To reduce function call / return overhead, the compiler may use function inlining. Automatic function inlining is active, when optimization level 3 is enabled. This automatic function inlining optimization is scalable, by defining a threshold value for the maximum size of (smaller) functions that shall be inlined and another threshold value which is set to specify a percentage value for the overall code size increase added due to automatic function inlining. The settings are defined using options:

--inline-max-size and --inline-max-incr

Ultimately for small applications like benchmarks it can be considered to inline all function calls.
C compiler option

--inline

is used for this purpose.

Within the TASKING Eclipse environment, those settings are configured in properties menu:

C/C++ Build >> Settings >> C/C++ Compiler >> Optimization

To further increase function inlining, it is possible to have the compiler prefer adding inline code instead of a run-time library function call, using the C compiler option:

--no-rtlib-calls

Within the TASKING Eclipse environment, this option needs to be added under 'Additional options' in menu:

C/C++ Build >> Settings >> C/C++ Compiler >> Miscellaneous

4.3 Project-Wide Optimization with MIL Linking

Optimizations on application scope can be enabled using MIL-split or MIL linking. If MIL linking is enabled, the C compiler will generate MIL files for every C source file. Those files include the so-called Medium Level Intermediate Language information. After all MIL files have been generated, they are used in a single C compiler invocation together with the MIL versions of the Standard and run-time libs to create one assembly language source file which includes the application code. The assembly language file is converted into an object file by the assembler which is finally processed by the linker.

For MIL linking, it is possible to specify the C compiler option --static. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module. Calling a static function permits additional optimizations including the usage of a fcall (fast call) assembly instruction when a static function is called.

MIL-split linking has a smaller impact on the build time of the application, compared to MIL linking when a single assembly code file is generated which includes the application code. Using MIL-split, an object file will be generated for each C source file instead. But the information included in the MIL files of the other C source files is utilized when the C compiler is processing a C source file to generate an object.

Simply spoken, this approach enables the C compiler to peek into the other source files which are included in the application to conduct frontend optimizations on application scope instead of file scope only.

If the content of a C source file changes, the file needs to be recompiled to generate an updated object file. But there is no need to recompile the other C source files.

4.4 Data Access Optimizations

To optimize access time for data read/write instructions, near accesses or address register indexed accessing may be used. The benefit is that two assembly instructions are needed to access near, or address register indexed accessed data instead of using three instructions for a far data access.

Example:

```
__near int var_1;
__far  int var_2;
```

Generated assembly code for data access:

```
; file_1.c              6     var_1 = 10;
        mov     d15,#10
        st.w    var_1,d15
; file_1.c              7     var_2 = 20;
        movh.a          a15,#@his(var_2)
        mov     d15,#20
        st.w    [a15]@los(var_2),d15
```

The access mode can be specified when defining the data and it needs to be included in the extern declaration of the data too. Qualifier __near is used to have the compiler generate assembly code for a near access to the selected variable. For access via e.g. address register A0 it is qualifier __a0 that needs to be applied to the data definition and extern declaration.

The available amount of near addressable data is limited by the TriCore architecture design. Near access is only permitted for data located in the first 16kB of a 256MB segment. This is the range:

X0000000h to X0003FFFh

For Ax address register accessed data, the amount is 64kB per address register. Technically the address registers A0, A1, A8 and A9 can be used for this access method. If an RTOS is used, some of those registers might be reserved for RTOS tasks and thus they are not available for application data.

For a small application which is not using a large amount of data, a default near allocation threshold value may be defined using:

--default-near-size=<max size of the largest variable/struct/array included in the application> / -N<max size of variable/struct/array included in the application>

Then the C compiler will use near accesses for all data with a size equal to or less the given threshold value, without the need to add the __near qualifier to the source code.

As an alternative it is possible to use this option without any threshold value filled in. Then all data will be near accessed:

--default-near-size / -N

Within the TASKING Eclipse environment, this option needs to be added under 'Additional options' in menu:

C/C++ Build >> Settings >> C/C++ Compiler >> Allocation

4.5 Application Placement Hints for Optimized Performance

The TriCore AURIX CPU includes different memory types like program flash memory, scratch pad memory, local memory unit (LMU) memory. The different memory types do have different access speeds. Each TriCore core can access its core local scratch pad memory (DSPR/PSPR) as well as the scratch pad memory of other cores. But the access speed to the local memory is faster compared to accessing the scratch pad memory of another core. For optimized performance, the data shall be placed in the core local memory of the core which is processing the data. This is achieved by defining groups within the linker LSL file. Those groups include select statements for data / code sections that shall be placed in the specified memories / memory ranges.

4.6 Cache Usage

For some memory types, cached access is possible which increases the execution performance.
To benefit from the cache usage, the SFR register configuration of the application for registers PCON and DCON needs to ensure the cache is enabled (not bypassed) and the code or data sections need to be in the cacheable memory range. For the TriCore CPU, the segments 0x8 and 0xA which include flash memory ranges are physically the same. But cache usage is only possible if the code is executed in segment 0x8. For the LMU RAM memory, segment 0x9 and 0xB are mirrored. Segment 0x9 is used for cached accesses.

Within the TASKING Eclipse environment, The PCON / DCON SFR register configuration is possible in menu:

C/C++ Build >> Startup Registers

4.7 Alignment of Code and Data Sections

The execution speed of application code depends on the alignment of the function entry address and the alignment of jump labels included in a function e.g. used for conditional code execution or in a for- or do-while loop. To ensure that the execution speed of a function is not changing when the content of the function itself is not edited, the function start address should have a defined alignment. Technically, assembly instructions typically have a minimal alignment of two bytes. When all functions are placed without any alignment requirement, they can end up in a contiguous block. The drawback here is: if the content of a function located at a lower address in memory is changed, this will shift the start address of all functions following that function in the memory range. This change can have an impact on the alignment of the following functions and the alignment of jump labels included in those functions. This can result in a changed execution speed for those functions although their content did not change at all. To prevent this, an alignment of e.g. 8 bytes for all functions which belong to the application code can be specified in the linker LSL file for the LSL group selecting the sections of those functions.

Example:

```
group MY_APP_CODE (ordered, align=8, run_addr=mem:mpe:pflash00 )
{
   select ".text.appcode.*";
}
```

Hint: To further improve the performance of an application, profiling can be conducted to locate the functions which have the biggest share in the application execution time. After those functions have been identified, the alignment of those functions can be changed by selecting the section including a function in individual LSL groups with a changed alignment. After the alignment change, another profiling run is required to measure the impact to the function execution time.

In a special use case, when the function execution time has been measured and for a selected function it turns out that a halfword alignment shows the better performance, a reserved entry needs to be added to the LSL group definition to ensure the function will always start on a halfword address (and not a word address).

Example: If section .text.appcode.func_1 needs to start on a halfword address:

```
group func_1_CODE (ordered, align=4)
{
   reserved "ALIGN_2" (size = 2 );
   select ".text.appcode.func_1";
}
```

4.8 Rebuilding Standard C and Floating-Point Run-Time Libraries

The Standard C and floating-point run-time libraries included in the compiler installation package are compiled using optimization level 2 and size speed tradeoff value 4 (maximum optimization for size). To increase the execution performance of library functions, the libraries can be rebuilt e.g. adapting the C compiler options for speed optimization (--tradeoff=0 / -t0). Details about rebuilding the libraries are included in the compiler user guide, TASKING SmartCode - TriCore User Guide (Chapter 14, p. 938).

After a recompiled library has been created, this can be copied into the lib folder which includes the default version of the library to replace it.

4.9 Access Speed Configuration of Target CPU Memory

Check the configuration of wait states to access flash memory. The calculation formulas are included in the Infineon User Manual.

Additional technical details, especially for the TriCore V1.6 Architecture are included in the Infineon application note named:

AP32168 Application Performance Optimization for TriCore V1.6 Architecture