DOI: 10.35598/mcfpga.2022.012

Practical Aspects of Software Optimization for MCUs with RTOS

Ivan Shevtsov
ORCID 0000-0003-0597-1589
dept. Microprocessor Technologies and Systems
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
ivan.shevtsov@nure.ua

Iryna Svyd
ORCID 0000-0002-4635-6542
dept. Microprocessor Technologies and Systems
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
iryna.svyd@nure.ua

Valeriia Chumak
ORCID 0000-0002-2403-020X
dept. Microprocessor Technologies and Systems
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
valeriia.chumak@nure.ua

Anton Sierikov
ORCID 0000-0002-3917-2008
dept. Microprocessor Technologies and Systems
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
anton.sierikov1@nure.ua

***Abstract*—This paper is focused on some practical aspects of optimization of MCU software written in C programming language using RTOS. Machine-specific optimizations and RTOS specific optimizations are described.**

Keywords—optimization, RTOS, MCU, periphery.

I. Introduction

The task of optimizing MCU software often appears in the process of developing and extending existing software. Although there is a lot of information on C programming language software optimization, the area of MCU software optimization remains largely undescribed. RTOS software optimizations are also extremely under-documented.

II. Before Optimization

The optimization process begins with searching for bottlenecks. Bottlenecks are searched by profiler integrated into debugger or by means of RTOS [1]. There are also IDE plugins for RTOS profiling [2]. It is also possible to perform profiling on your own.

When the bottlenecks are detected, it is necessary to determine what place consumes so much memory resource or processor time for a certain code fragment. In order to do this, a more detailed profiling is performed, determining how much time it takes to execute each function in a given code section, until you get to the clean code and library functions or system calls.

Understanding assembly code and architecture is necessary to analyse why normal code (without system and library calls) takes a long time to execute. For example, some calculations can be done using floating point numbers, which will take much longer than the same code using integer calculations.

Bottleneck analysis in system and library calls is more difficult because the source code is often unavailable. Generally, this is solved by a detailed inspection of the documentation. In some cases, it is possible to try to analyse the assembler code. When the bottleneck is the usage of libraries, there are the following solutions:

- rewrite the code without using third-party libraries, or system calls;
- to use more optimized versions of libraries;
- to use calls which use fewer resources but give the same or similar result.

III. Optimization Methods

A lot of information on optimization of programs for MCU written in C can be found in open access [3, 4]. There are also user's guides for program optimization for particular compilers [5] and particular MCU [6]. Here will be described some optimization techniques which are rarely described and which give significant performance gain.

A. Machine-specific optimizations

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

1) Copy code and constants to RAM or other "fast" memory: depending on the MCU architecture, this can improve the code execution time. For example, in many TMS devices of F28xxx family the speed of code execution is lower from flash memory than from RAM. Also, the speed of reading the constants from the flash memory, may have latency in the absence of prefetch data mechanisms [3]. In this case, can be recommended the placement of the most frequently executed or time-critical functions, as well as the constants associated with them in the RAM, additionally to methods of code optimization. For various devices and clock frequencies, this can give performance gains from 2.5% to 40% [7].

2) Intrinsic functions: Most microcontrollers architectures provide intrinsic functions for simplifying and increasing the performance of the most commonly used

tasks. For example, the Corterx-M4 kernel contains intrinsic functions for byte and bit reordering, which are useful in communication tasks [8]. There are also internal functions for frequently occurring DSP addition/subtraction with saturation.

3) Using more peripherals: One of the ways of offloading the CPU is to use more peripherals. Modern MCUs have a lot of peripherals which can do completely different things for the CPU. It is also possible to change the program, or algorithm, to make more usage of the peripherals.

a) Copying data using DMA instead of CPU resources: The C memcpy function in RTOS does not use DMA, so in order to copy large amounts of data frequently, it would be better to configure DMA.

b) Check-sum calculation: Many controllers have built-in peripherals to calculate error detection codes - for example CRC, or SHA-256. In case of need to increase performance of controller, it is recommended to use them.

c) Usage of timers, or the built-in RTOS delay functions, instead of using the C delay function: Often there is a requirement of implementation delays in MCU programming. For this purpose, you can use the function delay, which implements the delay by means of CPU. It is preferably to use RTOS facilities to implement delays (e.g. vTaskDelay in FreeRTOS [9]), or the MCU's timers. To implement delay accurately, it is necessary to use only the MCU timer with interrupts at the end of the time interval.

4) Using fixed-point arithmetic instead of float-point: Using fixed-point arithmetic (IQmath for example) instead of floating-point arithmetic in many architectures can give a performance gain, despite the fact that floating-point computation in many MCUs is handled by the peripheral coprocessor. At first look, it seems to contradict the previous point, but with floating point math the CPU is forced to stay idle while waiting for the operation to complete and for the result to be available. Additionally, to this time is added the time of copying values into the coprocessor registers and copying the result. Often, this time is longer than performing the same operation using fixed point arithmetic. Thus, it is recommended to use integer math in heavily loaded parts of the program. If it is impossible to refuse the floating-point arithmetic (for example, the required functions are missing in the library), it is possible to use optimized computation libraries (such as Fast RTS for the C2000 architecture by TI) or extension of the FPU - like TMU in C2000.

B. RTOS specific optimization

1) System calls cutting: One of the ways of optimizing time-critical interrupts is to remove RTOS code from interrupt begin and end. RTOS monitors the interrupt to check if higher priority tasks that may have been released during the interrupt operation are needed. If the interrupt does not use RTOS calls then it can bypass RTOS and interrupt hooks, profiling and other functions provided by OC will not be available. In case of need to profile such interrupts, it will be possible to use built in timers of MC by implementing profiling functions yourself.

2) Place more code into task and only necessary code into ISR: The main idea when placing code into interrupt handlers is: "Place only necessary code into ISR". Moving code between interrupts and tasks does not free up additional CPU resources, but allows the device to be more responsive to external impacts. However, if an interrupt is executed more often than a task, it will free up CPU resources. Also, it is worth actively using memoization, with the transfer of rare calculations to low-priority RTOS tasks.

3) Refuse usage of FPU in interrupts, or tasks: The usage of FPU coprocessor in applications with RTOS involves the need of saving FPU registers in the task stack and recovery (in addition to the ALU registers). For example, in STM32F4 MCU FPU has 32 single precision registers [10] (each 32-bits) that also have to be stacked in addition to 12 general-purpose registers and other registers. Such storing and restoring is performed by the CPU, and could be done every time tasks are switched and interrupts are entered. As an intermediate optimization, it is possible to allocate usage of FPU only in one handler of the program - a task, or interrupt. In such case there is no need to save and restore FPU registers, because other parts of the program will not use the coprocessor. In addition to the previous point, it is possible to move all resource-consuming calculations to low-priority task and do them with FPU using floating point arithmetic.

CONCLUSION

In this paper was described some optimization techniques for MCU software which give significant performance gain. Attention was focused on the optimizations related to the use of RTOS.

REFERENCES

[1] Run Time Statistics [Online]. Available: https://www.careeraddict.com/soft-skills [Accessed: 8- June- 2022]

[2] Stateviewer Plugin [Online]. Available: https://www.highintegritysystems.com/tools/stateviewer/ [Accessed: 8-June-2022]

[3] Optimization of Computer Programs in C [Online]. Available: http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html [Accessed: 8-June-2022]

[4] Optimizing C For Microcontrollers [Online]. Available: http://events17.linuxfoundation.org/sites/events/files/slides/Optimizing%20C%20For%20Microcontrollers.pdf [Accessed: 8-June-2022]

[5] TMS320C28x Optimizing C/C++ Compiler [Online]. Available: https://www.ti.com/lit/ug/spru514y/spru514y.pdf?ts=1657715409563 [Accessed: 8-June-2022]

[6] Optimizing C Code for Size With MSP430™ MCUs: Tips and Tricks [Online]. Available: https://www.ti.com/lit/an/slaa801/slaa801.pdf [Accessed: 8-June-2022]

[7] Running an Application from Internal Flash Memory on the TMS320F28xxx DSP [Online]. Available: https://www.ti.com/lit/an/spra958l/spra958l.pdf [Accessed: 8-June-2022]

[8] Intrinsic Functions for CPU Instructions [Online]. Available: https://www.keil.com/pack/doc/CMSIS/Core/html/group__intrinsic__CPU__gr.html [Accessed: 8-June-2022]

[9] vTaskDelay [Online]. Available: https://www.freertos.org/a00127.html [Accessed: 8-June-2022]

[10] Floating point unit demonstration on STM32 microcontrollers [Online]. Available: https://www.st.com/resource/en/application_note/dm00047230-floating-point-unit-demonstration-on-stm32-microcontrollers-stmicroelectronics.pdf [Accessed: 8-June-2022].