Embedded systems software, possibly even more than other kinds of software, is time critical and has cost-sensitive size constraints. Literally every bit of the microcontroller software costs precious code space and cycles. Even the most minute software weakness can lead to system-debilitating resource problems. Writing efficient C++ code for microcontrollers mandates command of the language and solid development practices. This chapter aids this endeavor by providing a selection of helpful tips for optimized C++ microcontroller programming.

6.1 Use Compiler Optimization Settings

Compiler optimization settings allow for flexible tuning of the compiler’s code generation. It is possible to optimize with emphasis on space, speed or a combination thereof. GNU compilers have a particularly rich set of command-line optimization settings. See van Hagen [3] Chap. 5 and Appendix A for further information on optimization settings in GCC.

When researching microcontroller optimization techniques for this book, a computationally intensive code sequence rich in 32-bit operations implementing a CRC32 cyclic redundancy check [4] was benchmarked. There are numerous well-known types of CRC calculations with various bit widths ranging from 4 to 64-bits. In this benchmark, we use a CRC32 / MPEG–2 algorithm, also commonly used for data-integrity verification in MPEG–2 program streaming [2, 5]. For the investigation here, the code has been optimized and specially designed for reliable porting to 8, 16, and 32-bit microcontrollers.

After being prepared for efficient use with microcontrollers, the CRC32 code was compiled three times for our target with the 8-bit microcontroller, the first two times optimized for speed and the third time optimized for space.

When benchmarking the CRC32 program for speed, two runs were made with optimization settings -O2 and -O3. The space optimized run used the optimization setting -Os. For the GNU C++ compiler, optimization setting -O2 performs most
available optimizations that do not strongly increase code size. Optimization setting 
−O3 performs all the optimizations of level −O2 plus additional potentially expen-
sive optimizations such as inline functions and loop distribution. See also [1, 3] for 
 further details on GCC optimization settings.

Table 6.1 shows the benchmark results for computing the CRC32 of the 8-bit 
ASCII characters representing the nine digits 1 to 9 (in other words: 0x31, 0x32, 
0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39). Both the space-optimized ver-
sion as well as each of the two speed-optimized versions obtain the correct result 
for the CRC32. In particular,

\[
\text{CRC32} (0x31 \ldots 0x39) = 0x0376E6E7. \quad (6.1)
\]

The space-optimized version of the algorithm results in a code size about 20% 
smaller than the version optimized for speed with −O2, whereas, the version opti-
mized for speed with −O2 runs approximately 10% faster than the space-optimized 
one. In general, space and speed are opposing optimization goals. Improvements in 
speed are usually obtained at the cost of larger code size. The benchmark results 
shown above confirm this tendency.

Differences between speed and space optimization can be strongly pronounced 
if inline-depth control, loop unrolling and common subexpression elimination are 
available. In particular, the size and speed of template-intensive code can be signif-
ically influenced by the compiler optimization settings.

The impact of these factors can be observed in Table 6.1. Consider the code sizes 
and the runtime characteristics resulting from optimization settings −O2 and −O3.
The version fully optimized with −O3 runs about 10% faster than the version opti-
mized with −O2, as expected. The code size, however, significantly increases with 
optimization setting −O3. In fact, the resulting code size with optimization setting 
−O3 is about a five times larger than the code size resulting from optimization set-
ting −O2. Does a 10% improvement in runtime justify a five-fold increase in code 
size? This depends on the characteristics and design goals of the application.

---

1 Calculating the CRC of the ASCII characters representing the nine digits 1 − 9 has evolved into 
a standard test for CRC checksum algorithms.

2 This is another testament to the quality and language standards adherence capabilities of GCC. 
GCC correctly compiles this 32-bit computationally intensive CRC32 calculation with ease and 
absolute correctness—even for an 8-bit platform.