# C compilers for ARM benchmark

**AN0052**

**Document version**
11 June 2013

# Contents

# 1. Introduction

## 1.1 Purpose of this manual

This benchmark evaluates the GNU C Compiler in comparison to other commonly used C compilers that have been developed around ARM7TDMI embedded system designs, to see how the GNU output measures up in terms of code size and speed of execution.

## 1.2 Scope of this manual

This benchmark is not intended to be an exhaustive comparison of C compilers for ARM core-based microcontrollers. Should you wish to reproduce these results, or run this analysis with another C compiler for ARM, this document and the files used in this test are available from our microcontroller support team.

## 1.3 Additional help or information

If you want additional help or information, if you find any errors or omissions, or if you have suggestions for improving this manual, go to the KEOLABS' site for Raisonance microcontroller development tools www.raisonance.com, or contact the microcontroller support team.

| | |
|---|---|
| Microcontroller website: | www.raisonance.com |
| Support extranet site: | support-raisonance.com (software updates, registration, bugs database, etc.) |
| Support Forum: | forum.raisonance.com/index.php |
| Support Email: | support@raisonance.com |

## 1.4 Raisonance brand microcontroller application development tools

January 1, 2012, Raisonance became the brand under which the company KEOLABS sells its microcontroller hardware and software application development tools.

All Raisonance branded products regardless of their date of purchase or distribution are licensed to users, supported and maintained by KEOLABS in accordance with the companies' standard licensing maintenance and support agreements for its microcontroller application development tools. For information about these standard agreements, go to:

| | |
|---|---|
| Support and Maintenance Agreement: | http://www.raisonance.com/warranty.html |
| End User License Agreement: | http://www.raisonance.com/software-license.html |

# 2. Overview

A number of ARM development tool providers base their offer on the GNU C Compiler, even though this tool set has not been developed specifically for designers of low and medium complexity embedded systems such as those that run on ARM7TDMI core-based microcontrollers.

This begs the question – is a compiler adapted to developing applications that run under complex file management systems, equally adapted to developing microcontroller applications? To respond to this question, this benchmark tests the GNU C Compiler against other C Compilers that have been developed around embedded system design to see how the GNU output measures up in terms of code size and speed of execution.

The results show that the GNU C Compiler performs very well against the other tested compilers, and in many cases it out performs its competitors. In addition, the results illustrate how the GNU function libraries (specifically *printf*) can constitute a handicap in embedded system design, as they are not specifically adapted to the requirements of low and medium complexity embedded systems. However, these test also show how this handicap can be overcome with relative ease with the use of simplified function libraries.

**Important note** (February 2006): On demand from the ARM company, their compiler, RVCT (RealView Compilation Tools), was removed from this benchmark.

### Testing in the context of embedded system design

Code size optimizations and speed optimizations can tend to have an inverse relationship – optimizing code size can result in slower execution, whereas optimizing for speed can result in larger code. For example, 'inlining' (including code of specified functions in the code of calling functions to reduce function call overhead) is a speed optimization that tends to increase code size because the code of 'inline' functions is replicated in all calling functions. As a result, application developers may often have to choose between size and speed of execution.

In low and medium complexity embedded systems, the memory resources of target microcontrollers (such as the STR7 with ARM7TDMI core that are used to measure speed of execution in this benchmark) can impose significant limits on code size. When the target microcontrollers have relatively limited memory resources, increases in code size translate rapidly to larger devices and higher cost. For this reason, evaluating code size is given priority over speed of execution in our evaluation of compiler results.

# 3. Test Environment

## 3.1 ARM-specific considerations

All of the tested compilers support compilation in both ARM (Normal) and Thumb Modes. In the rest of this document "ARM Mode" is referred to as "Normal Mode" to avoid confusion with the ARM C compiler.

In Normal Mode, instructions are coded in 32 bits, whereas they are coded in 16 bits for the Thumb Mode. Tests were run and results are reported for both modes. Generally speaking, in this study, compiling in Thumb Mode generated smaller code, where as compiling in Normal Mode generated code that executed faster

Constants and addresses have been included in the calculation of code size because, for ARM devices, loading of 32-bit addresses or constants is not otherwise possible within a single instruction. Some of the differences in compiler results are caused by the compiler's treatment of constants and address. Some compilers store all the constants in a single table at the end of the file, whereas others store the required constants for each function locally. We tried to take into account the size of the storage of constants as fairly as possible in the calculation of code size.

## 3.2 The source files

Ten files were used to run the tests in this benchmark:

| | |
|---|---|
| • mars.c – | Encryption algorithm |
| • lucifer.c – | Encryption algorithm |
| • playfair.c – | Encryption algorithm |
| • rijndael.c – | Encryption algorithm |
| • serpent.c – | Encryption algorithm |
| • sha.c – | Encryption algorithm (famous for smart cards) |
| • dhry.c – | Well known Dhrystone benchmark. Handle integer and memory blocks. |
| • des.c – | Encryption algorithm (famous for smart cards) |
| • towers.c – | Short solution for "towers of Hanoi" |
| • *whets.c* – | Well known Whetstone benchmark. |

All of these files except, *dhry.c* (Dhrystone) and *whets.c* (Whetstone), were selected randomly from a sample of C source files having a cryptographic or algorithmic orientation. "Randomly" means that we didn't perform any analysis before making our choice. Files with a cryptographic or algorithmic orientation have the notable particularity of generating redundant treatment that can help point out the optimizations made by the compilers.

*Dhry.c* and *whets.c* were selected because they are well known standards in benchmarking. These files have been modified slightly to be better adapted to an embedded environment. They were modified to disregard "time" functions, which are highly dependent on the target device and hardware architecture when testing speed of execution. They were also modified to ignore *printf* during speed measures so that speed of execution would not be dependant on speed of transmission.

Code size and Speed of execution without floating point numbers were tested using the *mars.c, lucifer.c, playfair.c, rijndael.c, serpent.c, sha.c, dhry.c, des.c* and *towers.c* files. *Serpent.c* was excluded from the Speed test because of a compilation anomaly that resulted with the IAR compiler. *Serpent.c* results are reported in the Code size results, but are not used to calculate the totals and averages.

Speed of execution with floating point numbers was tested using the Whetstone test, *whets.c*.

## 3.3 The C compilers

This benchmark compares three C compilers for ARM (as indicated previously,  the ARM RVCT compiler was removed from the benchmark.):

- **GNU** – GNU C Compiler for ARM, version 4.0
- **IAR** – 32K code-size limited version of the C Compiler delivered with the Embedded Workbench, version 4.20A
- **KEIL** – C Compiler delivered with µVision3, version 3.12a

The table below shows the configurations used when compiling with each toolset.
Maximum optimization was chosen for all tests:

- size optimization for Code size measurements
- speed optimization for speed of execution measurements.

“Interworking” was selected for IAR, but we found that the effect of this option on the results was negligible (always less than 0.5% on the measurements). The code size without interworking was virtually identical to the code size with interworking (difference of less than 0.5%).

Table 3.1 Configurations

| | **GNU** | **IAR** | **KEIL** |
|---|---|---|---|
| **CPU** | ARM7TDMI | ARM7TDMI | ARM7TDMI |
| **Target processor/board** | STR711FR2/REva | STR711FR2/REva | STR711FR2/REva |
| **Code size optimization** | -Os | Size, level: HIGH | Emphasis on size, level: 7 |
| **Speed optimization** | -O3 | Speed, level: HIGH | Emphasis on speed Level: 7 |
| **Signed char** | Enabled | Enabled | Enabled |
| **Interworking** | Disabled | Enabled (default) | Not available |
| **Inline/Auto inline** | | | |

## 3.4 Measuring and reporting

### 3.4.1 Calculating and reporting code size

The following measures of code size were used and are discussed in the analysis and conclusions:

- Pure C Code Size: the total size of compiled code only, not taking into account libraries
- Total Code Size: total size of compiled code including libraries
- Code Size Ratio: a factor determined by dividing the resulting code size for a compiler and a given file by the best result of the four tested compilers

In the result tables, code sizes are reported in parentheses. Code Size Ratios are reported in bold.

**Method**

Different procedures for obtaining code size had to be employed in some cases, depending on the features of each compiler, or the supporting integrated development environment.

- GNU: The code size of each function was determined from the resulting *.map*. Total code size including libraries was provided automatically by RIDE7.
- IAR: Code size calculations were complicated because the compiler appears to generate a "Data Table" (the equivalent of the label described previously in section ARM-specific considerations). This was not taken into account in the size provided in the *.lst*. For this reason, 4 bytes (size of an address) were added each time the function used a different "Data Table." The compiler also applied optimizations to the data access, creating "subroutines" when there was redundant code. As a consequence, the calculated total C code is the sum of the function sizes, taking into account the "Data Table." To this total, the size of each subroutine was added once. As a result, the size of each function equals the size of the function reported in the *.lst*, plus 4 bytes for each "Data Table" used by the function, plus the size of called subroutines. Total code size is the sum of the Const block and the Code block furnished by the *.map*.
- KEIL: The code size of each function was determined from the resulting *.map*. Total code size was calculated by adding the size of each Const section and Code section reported in *.map*.

Note: During our tests we noticed that size of each function calculate from the file *.lst* was different from the length gave in the *.map*. We decided to use those provided in the *.map* because they matched those we found in the executable.

### 3.4.2 Measuring execution speed

**Method**

To calculate execution times, code was added to the main function to generate three pulses and a final rising edge on one of the outputs of the microcontroller.

For all files, the main function calls a single other function (func). The falling edge of the last pulse indicates the beginning of func, and the last following rising edge indicates the end. As a consequence the duration of func, that is to say of the entire test can be calculated using these bounds. The execution times provided in the results for this benchmark are equal to the duration between the two last edges.

This additional code that was added to the main function was written in assembly language so that all the compilers generated the pulses based on the same code. As all the compilers had the same code, it was possible to confirm that the duration of the pulses was the same for all compilers. The three initial pulses generated, made it possible to confirm that startups initialized the CPU in the same way (in particular for the Core Clock).

**Hardware environment**

Applications were run on a REva mother board with an STR711FR2 ARM7TDMI core-based microcontroller from STMicroelectronics. Signals for time measurement were captured using a Philips PM3580 Logic Analyzer.

# 4. Code Size Comparison without Floating Point Numbers

## 4.1 Code size test introduction

Pure C code is that which results from compilation of the source files' instructions– it does not include libraries (notably *printf*) and the startup file. The *printf* functions are particularly large and, while pertinent to operating systems, they serve little purpose in low and medium complexity embedded systems. Measuring the size of Pure C code is of interest because it is measures the compiler's treatment of the coded instructions and not the size of supporting libraries.

On the other hand, *printf* functions could be used in an embedded environment and the impact of a compiler's *printf* library, for some developers, cannot be ignored. In this case using a *printf* library that is adapted to the requirements of the application and the embedded environment is in the developer's interest.

For these reasons we have run all of the following tests:

- Code size without *printf* libraries (Pure C code size)
- Code size with simplified *printf* libraries (when available)
- Code size with the full *printf* libraries

## 4.2 Comparison of pure C code size

When considering the pure C code size (code compiled without *printf* libraries), the GNU C compiler yields the best results (best in 8 cases).

While the IAR and GNU yield similar results in terms of code size, the results with the KEIL compiler stand out as being significantly worse that the other tested compilers. To better understand these results, we looked at the disassembled code to see how the compilers treated the code.

Note: When compiling *serpent.c* with the IAR C Compiler in Normal Mode, the compiler entered into an infinite loop. Compilation was never successfully completed and we have no explanation for this anomaly. The results with this file are reported for the other compilers, but are not included in the calculation of averages.

Table 4.1 Ratios and Pure C Code size

| | KEIL | | IAR | | GNU | | BEST | |
|---|---|---|---|---|---|---|---|---|
| **Mode**<br>**File compiled** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Mars.c** | 1.6<br>(12236) | 1.3<br>(8184) | 1<br>(7668) | 1<br>(6402) | 1.1<br>(8764) | 1<br>(6244) | IAR | GNU |
| **Lucifer.c** | 1.7<br>(1912) | 1.3<br>(1112) | 1<br>(1112) | 1<br>(846) | 1.1<br>(1232) | 1<br>(832) | IAR | GNU |
| **Playfair.c** | 1.8<br>(2028) | 1.5<br>(1164) | 1<br>(1156) | 1.1<br>(852) | 1.1<br>(1184) | 1<br>(772) | IAR | GNU |
| **Rijndael.c** | 1.4<br>(17044) | 1.1<br>(10584) | 1<br>(11964) | 1<br>(10080) | 1.3<br>(15556) | 1<br>(10140) | IAR | IAR |

| | KEIL | | IAR | | GNU | | BEST | |
|---|---|---|---|---|---|---|---|---|
| **Mode**<br><br>**File compiled** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Serpent.c** | 2.4<br>(37664) | 1.9<br>(22550) | *<br>* | 1<br>(12020) | 1<br>(15998) | 1<br>(11760) | GNU | GNU |
| **Sha.c** | 2.2<br>(13176) | 1.6<br>(6824) | 1.5<br>(9054) | 1.3<br>(5458) | 1<br>(6068) | 1<br>(4308) | GNU | GNU |
| **Dhry.c** | 1.6<br>(1764) | 1.4<br>(1032) | 1.1<br>(1240) | 1.1<br>(816) | 1<br>(1156) | 1<br>(736) | GNU | GNU |
| **Des.c** | 1.6<br>(1852) | 1.5<br>(1250) | 1<br>(1192) | 1<br>(838) | 1<br>(1136) | 1<br>(824) | GNU | GNU |
| **Towers.c** | 1.5<br>(908) | 1.3<br>(544) | 1.1<br>(652) | 1<br>(446) | 1<br>(620) | 1<br>(428) | GNU | GNU |
| | | | | | | | | |
| **Average of Code Size** | 1.5<br>(6365) | 1.3<br>(3836.8) | 1<br>(4254.8) | 1.1<br>(3217.3) | 1<br>(4464.5) | 1<br>(3035.3) | IAR | GNU |

**Analysis**

The results described in the preceding section led us to question why one compiler performs significantly worse than another. Upon analysis of the disassembled code we discovered that the tested compilers used different approaches, notably regarding:

- Calculations (made with different instructions)
- Data storage and access

The main differences can be seen in files such as *rijndael.c, serpent.c* or *mars.c.* These are all cryptographic programs with two major functions; encrypt and decrypt. The repetitive nature of these functions tends to amplify the advantage of the best compiler and the differences in the results. For this analysis, we looked at the disassembled code for the encrypt function from *mars.c*.

The compilers that performed best benefited from efficient data access. For example some compilers create a table whose address is kept in a register for the entire function, so to access these values, only one instruction using an addressing mode with an offset is needed. For other compilers, two instructions are used, so to load or to store a value the code is twice as large.

Note: mem(Address) means the value stored at the address Address in the memory.

For example, for the following C code:

```
a = l-key[0] ;

m = l_key[1] ;

c = l_key[2] ;
```

The resulting code for one compiler is:

```
LDR R4,0x90a0           : R4 = Address of l_key

LDR R2,[R4,#0]          : R2 = mem(value in R4+0)=l_key[0]
LDR R2,[R4,#4]              : R2 = mem(value in R4 + 4) =l_key[1]

LDR R2,[R4,#8]              : R2 = mem(value in R4 + 8) =l_key[2]
```

Whereas for KEIL, the result is:

```
LDR R0,[PC,#0x0EF8]         : R0 = Address of l_key[0]

LDR R0,[R0]                 : R0 = mem( value in R0)

LDR R0,[PC,#0x0EF4]         : R0 = Address of l_key[1]

LDR R0,[R0]                 : R0 = mem( value in R0)

LDR R0,[PC,#0x0EF0]         : R0 = Address of l_key[2]

LDR R0,[R0]                 : R0 = mem( value in R0)
```

Moreover, non-optimized code is used to calculate offset for some compilers.
For example, for the instruction:

```
b = s_box[ a & 255];
```

The code for GNU is:

```
AND R3, R7, 0xFF            : R3= a &255

LDR R1, [PC,#0xEA0]         : R1 = Address of S_Box

LDR R3, [R1,+R3,LSL #2]     : R3 =mem(Address) with address=R1+R3*2^2
```

The code for KEIL is:

```
AND R1,R1,#0x000000FF       : R1= a &255

MOV R1,R1,LSL #2            : R1 = a * 2^2

LDR R0,[PC,#0x0EA4] : R0 = S_Box

LDR R0,[R0,R1]              : R0 = mem(address) with address=R0+R1
```

Because functions such as Encrypt repeatedly use routines for calculation and load/storage of values, differences (like those illustrated in this example) are amplified and rapidly increase the size of the compiled code.

On a more general note, the results also show that Thumb Mode yields better results in terms of code size than Normal (ARM) Mode, as is illustrated by the ratios in table 4.1:

Table 4.2 Code Size Ratio – Normal/Thumb Mode

| | KEIL | IAR | GNU |
|---|---|---|---|
| **C Code Size Ratio (Normal/Thumb)** | 1.6 | 1.3 | 1.5 |

GNU performed as well as the other tested compilers (in terms of pure C Code Size) and in several cases yielded the best results. However, the difference between the IAR and GNU is relatively insignificant when compiling in both ARM and Thumb Modes.

The KEIL compiler produced notably and consistently larger code (pure C Code Size) than the other tested compilers. Further analysis of two cases of disassembled code shows that the difference in performance can be explained by the number of instructions the compiler used to interpret data access routines and calculations.

## 4.3 Comparison of Simplified and Full *printf* libraries

The preceding test and analysis was of “Pure C Code Size”, it did not include the *printf* functions, which serve little purpose in embedded applications and are very penalizing in terms of Code Size.

To demonstrate the impact of *printf* on the compiled applications, the same tests were run with full *printf* libraries and simplified *printf* libraries.

Compiling with a full *printf* library increases code size for all the tested compilers.

The GNU compiler is very heavily penalized by *printf*. On average, Code Size for GNU increased by a factor of 3.6 in Normal Mode and 3.9 in Thumb Mode.

However, using simplified versions of the IAR and GNU *printf* libraries significantly improves the results, bringing them into closer alignment with the best result.

Table 4.3 on page 13 shows the resulting Total Code Size when using full *printf* libraries.

Table 4.4 on page 14 shows the Code Size results when using simplified *printf* libraries for IAR and GNU.

### Analysis

The results in this section illustrate the impact of *printf* libraries on Code Size.

Why does the use of *printf* libraries have such a heavy impact on the GNU results? It is important to note the GNU, unlike the other tested compilers, has not been developed specifically for microcontroller-based embedded systems. This is reflected in its *printf* libraries, developed around files and file management and more appropriate to operating systems (Linux, Windows CE, etc.).

Generally speaking, compilers aimed at microcontroller–based embedded systems implement a *printf* function that uses putchar directly. With the GNU compiler and full *printf* library, a function containing a call to *printf* requires an additional 30K of *printf* libraries, of which 15K are due to file management (*printf* calls f*printf*).

Using simplified *printf* libraries that are better adapted to the requirements of a microcontroller-based embedded system, it is possible to attain dramatic improvements in Code Size. This is demonstrated by the results with simplified *printf* libraries for IAR and GNU.

Table 4.3 Ratios and Total Code Size with Full printf Libraries

| | KEIL | | IAR | | GNU | | BEST | |
|---|---|---|---|---|---|---|---|---|
| **Mode**<br><br>**File compiled** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Mars.c** | 1<br>(16328) | 1<br>(12264) | **1.3**<br>*(20948)* | 1.4<br>(17228) | 2.6<br>(42656) | 2.7<br>(32804) | KEIL | KEIL |
| **Lucifer.c** | 1<br>(4100) | 1<br>(3300) | 3.2<br>(13224) | 2.9<br>(9468) | 8.6<br>(35408) | 8.3<br>(27448) | KEIL | KEIL |
| **Playfair.c** | 1<br>(4204) | 1<br>(3332) | 2.9<br>(12184) | 2.8<br>(9388) | 8.7<br>(36372) | 8.4<br>(27832) | KEIL | KEIL |
| **Rijndael.c** | 1<br>(18896) | 1<br>(12436) | 1.2<br>(22824) | 1.5<br>(18500) | 2.6<br>(49652) | 3.0<br>(36708) | KEIL | KEIL |
| **Serpent.c** | 1<br>(39512) | 1.2<br>(24400) | *<br>* | 1<br>(20459) | 1.2<br>(49382) | 1.9<br>(38224) | KEIL | IAR |
| **Sha.c** | 1<br>(15684) | 1<br>(9332) | 1.3<br>(20032) | 1.6<br>(14480) | 2.6<br>(40528) | 3.4<br>(31396) | KEIL | KEIL |
| **Dhry.c** | 1.4<br>(2285) | 1.4<br>(1553) | 1<br>(1636) | 1<br>(1112) | 2.5<br>(4104) | 2.9<br>(3216) | IAR | IAR |
| **Des.c** | 1.1<br>(4368) | 1.2<br>(3776) | **1.1**<br>*(4280)* | 1.2<br>(3744) | 1<br>(3868) | 1<br>(3108) | GNU | GNU |
| **Towers.c** | 1<br>(2720) | 1<br>(2356) | 4.2<br>(11540) | 3.8<br>(8876) | 13.2<br>(35916) | 11.8<br>(27716) | KEIL | KEIL |
| | | | | | | | | |
| **Average of Code Size** | 1<br>(8573.1) | 1<br>(6043.63) | 1.6<br>(13333.5) | 1.7<br>(10349.5) | 3.6<br>(31063) | 3.9<br>(23778.5) | KEIL | KEIL |

Table 4.4 Ratios and Total Code size with simplified printf libraries

| | KEIL | | IAR | | GNU | | BEST | |
|---|---|---|---|---|---|---|---|---|
| **Compilation Mode**<br><br>**File compiled** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Mars.c** | 1.4<br>(16328) | 1.2<br>(12264) | 1<br>(11804) | 1<br>(10124) | 1.4<br>(16708) | 1.2<br>(12632) | IAR | IAR |
| **Lucifer.c** | 1<br>(4100) | 1.4<br>(3300) | 1<br>(4104) | 1<br>(2412) | 2.4<br>(9636) | 3.1<br>(7592) | IAR | IAR |
| **Playfair.c** | 1.4<br>(4204) | 1.4<br>(3332) | 1<br>(3100) | 1<br>(2356) | 3.3<br>(10300) | 3.4<br>(7952) | IAR | IAR |
| **Rijndael.c** | 1.4<br>(18896) | 1.1<br>(12436) | 1<br>(13704) | 1<br>(11444) | 1.7<br>(23748) | 1.5<br>(16752) | IAR | IAR |
| **Serpent.c** | 1.7<br>(39512) | 1.8<br>(24400) | *<br>* | 1<br>(13376) | 1<br>(23944) | 1.4<br>(18152) | GNU | IAR |
| **Sha.c** | 1.6<br>(15684) | 1.3<br>(9332) | 1<br>(10088) | 1<br>(7396) | 1.5<br>(14784) | 1.5<br>(11336) | IAR | IAR |
| **Dhry.c** | 1.4<br>(2285) | 1.4<br>(1553) | 1<br>(1636) | 1<br>(1112) | 2.5<br>(4104) | 2.9<br>(3216) | IAR | IAR |
| **Des.c** | 2<br>(4368) | 2.1<br>(3776) | 1.9<br>(4280) | 2.1<br>(3744) | 1.7<br>(3868) | 1.7<br>(3108) | IAR | IAR |
| **Towers.c** | 1.1<br>(2720) | 1.3<br>(2356) | 1<br>(2396) | 1<br>(1792) | 3.6<br>(8540) | 3.8<br>(6792) | IAR | IAR |
| | | | | | | | | |
| **Average of Code Size** | 1.3<br>(8573.1) | 1.2<br>(6043.6) | 1<br>(6389) | 1<br>(5047.5) | 1.8<br>(11461) | 1.7<br>(8672.5) | IAR | IAR |

## 4.4 Overall code size optimization conclusions

When evaluating GNU's treatment of C source code (independently of its *printf* libraries), we find that the "Pure C Code Size" attained is very similar to that produced by IAR compiler. In a significant number of cases, GNU out-performs the other tested compilers.

The subsequent tests illustrate how GNU is penalized by the implementation of *printf* libraries that are better adapted to applications functioning under an operating system.

Moreover, these results show that adapting *printf* libraries (if *printf* is used in your application) to meet the functional requirements of an embedded application can directly improve Code Size results.

In addition, it is a reasonable assumption that other GNU libraries such as scanf (when used by your application) could be simplified to meet functional requirements and avoid adverse effects on Code Size.

# 5. Speed Comparison with/without Floating Point Numbers

## 5.1 Introduction of speed tests

Even though the priority of this benchmark is to evaluate C compiler performance in terms of output code size, the speed of execution of the resulting executable cannot be overlooked as a measure of performance.

To measure speed without floating point numbers, tests were run with 8 of the 9 files in the test sample. These files were modified to generate signals indicating the start and end of the functions. Files were compiled with the highest speed optimization and simplified *printf* libraries.

The Whetstone test (*whets.c*) was used to measure speed of execution with floating point numbers. Full *printf* libraries for IAR and GNU had to be used in order to support floating point numbers. For this test Pure C Code Size, Code Size and Speed of Execution are reported.

The CALDP.LIB for floating point math was not available for KEIL (at the moment, KEIL apparently supports only 32-bit floating point numbers), so the KEIL compiler was excluded from this test, and only the compilers using the same standard format (64-bit IEEE754) were compared.

On the first run of the Whetstone tests, the GNU results were exceptionally good (it outperformed the others by a ratio of nearly 5.). However, further evaluation of the disassembled code and the execution time in each module of the function *whets_main* (in *whets.c*) showed that some calculations had not been compiled. The GNU compiler had optimized the code to avoid calculations that were not used at run time. For example, when *printf* is removed from *POUT*, GNU optimizes the code to avoid doing any calculations that were made unnecessary by the removal of *printf*.
To avoid this kind of optimization, *whets.c* was divided into two files *whets.c* and *pout.c*. In *whets.c* the function main and *whets_main* were retained. *PA, POUT, Proc0* and *Proc3* were put in the *pout.c* file.

As a result, when compiling *whets_main*, the content of *POUT* is now hidden from the GNU optimizer and cannot be optimized so dramatically as the compiler can no longer determine whether or not the results of the intermediate calculation will be useful.

## 5.2 Speed of Execution Results without Floating Points

In this first test without floating point numbers, GNU produces excellent results that are equal to (in Thumb Mode), or better than (in Normal Mode) its competitors. When looking at total execution time for all the compiled files, IAR and GNU produce consistent results.

Table 5.1 Ratios and Execution Time (in ms)

| | KEIL | | IAR | | GNU | | BEST | |
|---|---|---|---|---|---|---|---|---|
| **Compilation Mode**<br><br>**File compiled** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Mars.c** | 4.2<br>(23.8) | 3.6<br>(25 2) | 2<br>(11.6) | 2.3<br>(16.4) | 1<br>(5.7) | 1<br>(7.0) | GNU | GNU |
| **Lucifer.c** | 2.0<br>(360) | 1.6<br>(423) | 1<br>(183) | 1.3<br>(342) | 1.1<br>(208) | 1<br>(270) | IAR | GNU |
| **Rijndael.c** | 3.8<br>(158) | 1.4<br>(165) | 2.1<br>(87.4) | 1.2<br>(146) | 1<br>(41.8) | 1<br>(120) | GNU | GNU |
| **Playfair.c** | 2.4<br>(13.6) | 1.7<br>(13.8) | 1<br>(5.7) | 1<br>(8.3) | 1.1<br>(6.3 ) | 1.5<br>(12.5) | IAR | IAR |
| **Sha.c** | 3.6<br>(14.9) | 2.7<br>(15) | 2.2<br>(9.4) | 2.1<br>(11.6) | 1<br>(4.2) | 1<br>(5.6) | GNU | GNU |
| **Dhry.c** | 2.5<br>(45.7) | 2.1<br>(46.3) | 1<br>(18) | 1<br>(21.9) | 1.3<br>(24.2) | 1.3<br>(28.1) | IAR | IAR |
| **Des.c** | 1.5<br>(358) | 1.7<br>(492) | 1.2<br>(280) | 1.1<br>(338) | 1<br>(237) | 1<br>(297) | GNU | GNU |
| **Towers.c** | 2.3<br>(1.69) | 1.7<br>(1.72) | 1.4<br>(1) | 1.2<br>(1.2) | 1<br>(0.7) | 1<br>(1) | GNU | GNU |
| **Sum of Execution Times** | 1.8<br>(975.7) | 1.6<br>(1181) | 1.1<br>(596.1) | 1.2<br>(885.5) | 1<br>(527.9) | 1<br>(741.2) | GNU | GNU |

## 5.3 Speed of Execution Result with Floating Point Numbers

In the second test with floating point numbers, even though the GNU code is larger, its performance in terms of speed of execution is close to the result attained by the best compiler. Whereas IAR produces smaller code, its speed of execution lags behind the competition by a factor of about 2.

The results of this test represent well the trade offs that designers may be faced with when forced to choose between speed of execution and code size. In this case the GNU compiler offers a good compromise with the best result for execution speed and an increase in code size by a factor of 1.8 in Normal Mode and Thumb Mode. Once more, this compiler is very good for the generated code (Pure C Code), but very handicapped by its libraries that are tailored for the huge Operating Systems.

Table 5.2 Code Size and Ratios of each function of the Whetstone Test and Total Code size

| | **IAR** | | **GNU** | | **BEST** | |
|---|---|---|---|---|---|---|
| **Mode** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Pure C Code Size** | 1 (*2216*) | 1 (*1494*) | 1 (2200) | 1 (1532) | GNU | IAR |
| **Total Code Size** | 1 (12544) | 1 (9692) | 1.8 (22516) | 1.8 (17768) | IAR | IAR |

Table 5.3 Ratios and Execution Time of the Whetstone Test and Total Code size

| | **IAR** | | **GNU** | | **BEST** | |
|---|---|---|---|---|---|---|
| **Compilation Mode** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| **Execution Time (in seconds)** | 2 (1.8) | 2 (1.9) | 1 (0.9) | 1 (0.97) | GNU | GNU |

**Analysis**

In the Whetstone test, floating point numbers (float libraries of approximately 15K) and their treatment by *printf* are costly in terms of Total Code Size for both tested compilers.
GNU is particularly penalized – larger than IAR by a factor of almost 2.

However, in speed of execution, we see that GNU outperforms IAR by a factor of about 2!

# 6. Overall conclusion

These results show clearly that the GNU C Compiler is a good choice in comparison with the other compilers tested here in terms of both code size and speed optimizations. However, when code size is an issue in your embedded application, users need to be aware of the impact GNU libraries such as *printf* can have on code size.

The Code Size tests run in this benchmark demonstrate the extent to which the GNU Compiler can be handicapped by a very complete *printf* library that serves little purpose in embedded applications. Excluding *printf*, IAR and GNU Compilers produced similar, consistent results in terms of Pure C Code Size, which we have considered a better measure for embedded applications than Code Size including *printf*. In addition, for embedded applications that use *printf*, or other GNU libraries, the results demonstrate that using a simplified version of the GNU libraries can improve results so that they are in line with the other tested compilers.

In speed tests without floating point numbers, GNU attained the best results when compiling in Normal Mode and in Thumb Mode. In addition, for speed tests with floating point numbers, GNU's speed of execution result is the best one.

The following table summarizes the main results in this comparison.

Table 6.1 Summary of Results

| | KEIL | | IAR | | GNU | | BEST | |
|---|---|---|---|---|---|---|---|---|
| | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** | **Normal** | **Thumb** |
| *Summary A: Ratios for "Pure C Code size" and Speed for 10 programs. The Code size is measured without the libraries (printf removed).* | | | | | | | | |
| **1. Code Size (no FP)** | 1.5 (6365) | 1.3 (3836.8) | 1 (4254.8) | 1.1 (3217.3) | 1 (4464.5) | 1 (3035.3) | IAR | GNU |
| **2. Speed (no FP)** | 1.8 (975.7) | 1.6 (1181) | 1.1 (596.1) | 1.2 (885.5) | 1 (527.9) | 1 (741.2) | GNU | GNU |
| *Summary B: Ratios for "Pure C Code size" and Speed for "Whetstones" (calculation with 64-bit floating point numbers)* | | | | | | | | |
| **3. Code Size (with FP)** | | | 1 (*2216*) | 1 (*1494*) | 1 (2200) | 1 (1532) | GNU | IAR |
| **4. Speed (with FP)** | | | 2 (1.8) | 2 (1.9) | 1 (0.9) | 1 (0.97) | GNU | GNU |

# 7. Conformity

**ROHS Compliance (Restriction of Hazardous Substances)**

KEOLABS products are certified to comply with the European Union RoHS Directive (2002/95/EC) which restricts the use of six hazardous chemicals in its products for the protection of human health and the environment.

The restricted substances are as follows: lead, mercury, cadmium, hexavalent chromium, polybrominated biphenyls (PBB), and polybrominated diphenyl ethers (PBDE).

**CE Compliance (Conformité Européenne)**

**KEOLABS products are certified to comply with the European Union CE Directive.**

In a domestic environment, the user is responsible for taking protective measures from possible radio interference the products may cause.

**FCC Compliance (Federal Communications Commission)**

KEOLABS products are certified as Class A products in compliance with the American FCC requirements. In a domestic environment, the user is responsible for taking protective measures from possible radio interference the products may cause.

**WEEE Compliance (The Waste Electrical & Electronic Equipment Directive)**

KEOLABS disposes of its electrical equipment according to the WEEE Directive (2002/96/EC).

Upon request, KEOLABS can recycle customer's redundant products.

For more information on conformity and recycling, please visit the KEOLABS website *www.keolabs.com*

# 8. Glossary

| Term | Description |
|---|---|
| RBuilder | Application builder that allows users to configure device peripherals and out put the required C code automatically for their applications. Code is based on libraries provided by the manufacturer. |
| REva | Raisonance evaluation platform – modular evaluation boards with main evaluation board (motherboard) and daughter boards featuring different microcontrollers |
| RFlasher | Raisonance Flasher: Programming interface for user-friendly flash programming |
| Ride7 | Raisonance Integrated Development Environment |

# 9. Index

# 10. History

| Date | Description |
|---|---|
| 02/02/06 | RCVT compiler from ARM removed |
| 17/07/13 | Modified template |