Cite this chapter as:
Elmenreich W., Wolf A., Rosenblattl M. (2009) Providing Standardized Fixed-Point Arithmetics for Embedded C Programs. In: Martínez Madrid N., Seepold R.E. (eds) Intelligent Technical Systems. Lecture Notes in Electrical Engineering, vol 38. Springer, Dordrecht

Chapter 16

PROVIDING STANDARDIZED FIXED-POINT ARITHMETICS FOR EMBEDDED C PROGRAMS

Fixed-point Library based on ISO/IEC TR 18037

Wilfried Elmenreich[1], Andreas Wolf[2] and Maximilian Rosenblattl[2]
[1]Lakeside Labs, Mobile Systems Group, Institute of Networked and Embedded Systems, University of Klagenfurt, 9020 Klagenfurt, Austria; [2]Vienna University of Technology, 1040 Vienna, Austria

Abstract: The ISO/IEC Standard TR 18037 defines the syntax and semantics for fixed-point operations for programming embedded hardware in C. However, there are currently only few compilers available that support this standard. Therefore, we have implemented a stand-alone library according to the standard that can be compiled with standard C compilers. The library is available as open source and written in plain C, thus can be used in various target architectures as long as a C compiler is available. This book chapter presents a brief description of the ISO/IEC standard and the library implementation followed by an evaluation of code size and performance of the fixed-point operations on the Atmel AVR architecture. A comparison with the standard floating-point library (which is machine code-optimized to the target architecture) shows that simple fixed-point functions such as addition, subtraction and multiplication are more efficient, while more complicate functions can only compete in the worst case behavior. The fixed-point approach provides a smaller memory foot print, for typical applications where only a small subset of functions is used. This is especially of interest for the big market of embedded microcontrollers with only a few Kbytes of program memory.

Key words: Fixed-point Arithmetics, C Programming Language, Embedded C, CORDIC

† This work was supported in part by the Austrian FWF project TTCAR under contract No. P18060-N04.

1. INTRODUCTION

The C language standard[1] specifies two data types for expressing fractional numbers, the float and the double data type. Both data types are in a floating-point format consisting of sign, exponent and mantissa according to the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)[2].

Since the regular integration of the floating-point coprocessor in the processors for personal computers (in the x86 world, this became reality when the 486DX replaced the 486SX in 1992), floating-point arithmetic was available with high performance and became a ubiquitous feature for computer programs.

On systems without a dedicated hardware module for floating-point operations, these have to be emulated by a series of integer operations in software. This is typically the case for embedded microcontrollers, which in most cases do not come with a floating-point module since die size and, therefore, the cost of a microcontroller is strongly increased when a floating-point unit has to be added in hardware. Also, many embedded applications do well without hardware floating-point support. The NIOS II soft core processor[3], for instance, requires around 700 Logic Elements when synthesized as a 32 bit integer processor onto an FPGA (Field-Programmable Gate Array). When adding a floating-point unit, the final design requires around thrice the size of the plain integer design.

Without compiler support, a programmer either has to code the operations manually, use one of the many libraries available for fixed-point operations or use tools like Matlab[4] that are able to generate C code that simulates fixed-point arithmetic.

Existing solutions for fixed-point libraries suffer from one of the following deficiencies: (i) the data types are not standardized, thus it is not possible to reuse code with a different library, (ii) not all required functions are supported, e. g., missing support for trigonometric functions, (iii) if the library is rather complete, the overhead on linking the library to the final program creates a large memory footprint, and (iv) the library is not written in C but in C++ or an architecture-specific assembly language.

Unfortunately, until recently, there was not much compiler support for fixed-point data types and no standard for implementing fixed-point data types. In 2008, ISO issued a standard that is describing the syntax and the data types for fixed-point arithmetic as an extension to the programming language C[5]. However, for particular embedded target systems there is still a lack of compilers that support this standard. Therefore, we have implemented the data types and functions of this standard as a software library that can be used with any standard C compiler.

It is the purpose of this chapter to describe a high-level language implementation of the main parts of the ISO/IEC Standard TR 18037 and evaluate the results by comparison to the standard software floating-point library of the avr-gcc compiler, a compiler for the embedded AVR 8-bit microcontroller series. Our intention is to provide a very generic implementation that can act as a transitional solution for systems where compiler support for the new standard is not yet available as well as a solution for applications with moderate performance requirements. Moreover, the timing behavior of the functions in our library has been thoroughly analyzed on the AVR architecture so that these data supports static Worst Case Execution Time analysis methods[6] for real-time systems on that hardware.

The rest of the chapter is structured as follows:

Section 2 reviews some basic properties of fixed-point and floating-point arithmetic. Section 3 gives a short introduction to the ISO/IEC 18037 standard. Section 4 describes our implementation of a library. Section 5 depicts the evaluation results for our library on the AVR architecture. Section 6 compares the results to floating-point operation. Section 7 concludes the chapter.

2. A CLOSER LOOK ON FIXED-POINT ARITHMETICS

Most programming languages offer only floating-point arithmetic in order to express fractional numbers. Being noticeable exceptions, ADA and COBOL are one of a few programming languages that also natively support fixed-point data types. The reason for this is that floating-point arithmetic comes with the following advantages over fixed-point numbers:

- They approximate real numbers over a relatively wide data range. For example, the float data type in C allows to express numbers between $1.175 \cdot 10^{-38}$ and $3.403 \cdot 10^{38}$. The double data type even supports numbers between $2.225 \cdot 10^{-308}$ and $1.798 \cdot 10^{308}$.
- They provide, except for special situations, like for example very small numbers near zero, a constant relative precision for approximating a real number. The float data type has a precision of $2^{-24} = 5.960 \cdot 10^{-8}$, the double data type has a precision of $2^{-53} = 1.110 \cdot 10^{-16}$.
- The number format is standardized by IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) to four different precisions: single, double, single-extended, and double-extended. Typically, programming languages and most floating-point hardware support single and double

precision[7]. There exists another standard, IEEE 854[8] that also supports specifying floating-point numbers on the basis 10, which is aligned to representation of numbers by humans. However, unlike IEEE 754, IEEE 854 does not specify how a binary format of a number should look like so that is less applicable in computer programming.

Therefore, most applications will do well in using floating-point numbers. However, there are some points in favor of fixed-point numbers:

- Operations on fixed-point numbers are less complicated than floating-point numbers. This argument is especially of weight, if the target hardware has no special support for floating-point calculations, which is often the case for embedded hardware. Therefore, using fixed-point arithmetic can yield a performance benefit.
- When the numbers to be operated with are in an a priori known order of magnitude, a fixed-point data type that has the radix in an appropriate position can store a number more efficiently than the floating-point data type, since the latter has also to store the flexible exponent. Again, this is an issue for embedded systems with a limited amount of RAM memory.
- Due to the reduced complexity of the fixed-point operations, the resulting code size is by a few kbytes smaller when using fixed-point instead of floating-point numbers. Accordingly to the previous argument, this is an issue for embedded systems with a limited amount of program memory.

If an application requires floating-point arithmetic or does better with fixed-point depends mainly on the data set the application has to deal with. Frantz and Simar[9] discuss this on the example of video and audio processing: While discrete cosine transformations and quantization operations as they appear in video signal processing can be effectively handled using integer operations while audio processing typically uses cascaded filters where each filtering state propagates the error of previous stages. Furthermore, audio signals must retain accuracy even if the signal approaches zero due to the sensitivity of the human ear, which makes audio applications less suitable for fixed-point arithmetic.

Therefore, the employment of fixed-point arithmetic can be advantageous for some embedded applications where fractional numbers are required, but floating-point operations would be too expensive in terms of hardware cost or processing time.

3. FLOATING-POINT EXTENSIONS ACCORDING TO ISO/IEC TR 18037:2008

Because fixed-point operations are commonly used for microcontrollers, the ISO/IEC has summarized some guidelines and suggestions in a technical report named ”Extensions for the programming language C to support embedded processors”[5], which defines some guidelines for including fixed-point data type support into C compilers. This includes data types, #pragma directives, constants, function names, and some mathematical conventions.

Although the standard is intended to describe fixed-point extensions for C compilers, we have decided to use those guidelines for the implementation of an external C library.

Having the real time and code size limitations in mind, we decided to implement a reasonable subset of the data types defined by the standard. The overall set of data types and the implemented types are shown in Table 1. The numbers indicate the bits that are available for representing the integer part and the fractional part. Note that the definition of our implementation exceeds the number of specified bits, which is still in conformance with the standard, since the given values specify the minimum number of bits for the data types.

Table 16-1. Fractional data types according to ISO/IEC TR 18037

ISO/IEC Definition		Implemented	
signed short _Fract	s.7		
signed _Fract	s.15		
signed long _Fract	s.23		
signed short _Accum	s4.7	_sAccum	s7.8
signed _Accum	s4.15	_Accum	s15.16
signed long _Accum	s4.23	_lAccum	s7.24

FX_ACCUM_OVERFLOW defines the overflow behavior of the Accum data types. When set to SAT, saturation is enabled; this means that when an overflow occurs, the result is either the minimal or the maximal possible value of the data type. This behavior often means a significant loss of speed and further increases code size, so the #pragma is normally set to DEFAULT, which is the other possible value. Since we cannot implement #pragmas in a library, this behavior can be set with a #define FX_ACCUM_OVERFLOW before the library header file is included.

FX_FRACT_OVERFLOW is the same for the Fract data types. Since we have none of them implemented, this #pragma is not used in our library.

FX_FULL_PRECISION forces the implementation to gain maximum precision, by allowing a maximum error of one ULP (Unit in the Last Place)

of the result. However, for particular problems, a precision of 2 ULPs on multiplication and division operations is sufficient, which enables optimizations towards execution speed and code size. For our implementation the FX_FULL_PRECISION switch is not implemented, instead the precision has been predicted separately for each function.

Almost any meaningful data type handling and some low level arithmetic functions are defined through naming conventions and behavior descriptions. There is one version for each data type respectively several for conversion and mixed type functions. Except for their parameters they differ also by some trailing and/or leading characters which describe the type of the parameters respectively the result.

4. LIBRARY IMPLEMENTATION

The primary goal of this work was to provide a fixed-point library especially for use with Atmel 8 bit processors in combination with real time applications. So, performance and performance predictability were strong requirements for the design. Also flash memory was very limited, therefore small code size was desired.

To optimize code for size and speed, every function was implemented as accurate as possible, trying to keep it mathematically fast and simple (thus reducing code size). Apart from optimizing the overall code size of the library, each function has been compiled into a separated object file that is only linked to the final program if the function was used.

A main decision that was made refers to the data types. The ISO/IEC paper recommends both the _Fract and the _Accum type. The difference between those two types is only the lack of integral bits in the _Fract type while not increasing the number of fractional bits, so we decided to implement the _Accum type. To further limit the complexity of the implementation, we only implemented two subtypes of the _Accum type. Although the two data types should be named _Accum and long _Accum, there is a problem with the name of the second type. As we use typedef to define the type, the name of the data type must not have blanks in it. So we decided to call it _lAccum, which should be kept in mind when comparing AVRfix with the ISO/IEC specification.

Both types are signed and held in a 32 bit container (signed long). While _Accum has 15 integral and 16 fractional bits, _lAccum has only 7 integral bits but therefore 24 fractional bits. Because we use the long data type as container, addition and subtraction are working implicitly by using integer arithmetic as long as _Accum and _lAccum are not mixed. Therefore, addition and subtraction require no additional function in the library.

Overloading of operators is not supported in ANSI-C, so for example a multiplication needs to be done by a function call or a macro. While function calls produce some overhead on runtime, the use of macros increases code size. Most operations except conversion functions are implemented as functions. Comparison functions are working as long as the data types are the same; casting has no effect for _Accum and _lAccum. If a comparison between a long and an _Accum is needed, one (or both) of the variables needs to be converted before the comparison can be done. The same approach is needed for assignments.

To meet requirements of code size and execution speed, the FX_FULL_PRECISION switch has not been implemented. Instead, the expected precision has been evaluated and documented separately for each function in the project documentation[10]. This evaluation includes also sophisticated math functions such as trigonometric functions and square root. The library is completely written in C and has been tested and evaluated with the established compilers avr-gcc 3.3.2 and the Microsoft Visual Studio IDE 6.0.

In reference to the ISO/IEC report, the naming conventions are used accordingly whenever possible, meaning that for _Accum a *k*, for _lAccum *lk* and for _sAccum *sk* is used as suffix to the function name to indicate the type of the parameters. The type of the return value is indicated by a letter before the function name. No letter suggests _Accum, an *l* means that the return value is of type _lAccum, and an *s* refers to _sAccum.

For example, the multiplication function that multiplies two _Accum values and returns an _Accum value, is named mulk. The multiplication function that multiplies two _lAccum values and returns an _lAccum value, is named lmullk.

Furthermore, the ISO/IEC paper specifies the FX_ACCUM_OVERFLOW flag, which defines the behavior if an overflow occurs. If it is set to saturation (SAT), the value will be either the maximum or minimum possible value if an overflow occurs. By default, an overflow will give an undefined result. While in the ISO/IEC paper this flag is defined as a #pragma directive, we needed to use a #define for the FX_ACCUM_OVERFLOW flag. Independent from this flag the behavior can be achieved by calling the respective version of the function directly. If a function provides both behaviors, there exist two functions which have either S for saturation or D for default behavior as trailing character after the function name. So one can attach an S to a function name to force saturation behavior or a D to force the default behavior (resulting in e.g. mulkD or mulkS for the two versions of mulk) if the function provides two different behaviors.

Apart from the four arithmetic basic operations, the library support also operations such as square root, logarithmic and trigonometric functions. For the latter we have decided to use the CORDIC approximation[11] instead of a Taylor series, because it saves considerable program memory when requiring sine and cosine (or, consequently, tangens) in the same program while achieving almost the performance of the Taylor version.

5. EVALUATION

For tests and benchmarks we used an evaluation board equipped with an Atmel ATMEGA 16, providing 16 MHz clock, 16 Kb flash memory and 2 Kb SRAM. For evaluating the correctness of the calculations done by the library, we tried to cover all meaningful calculations. To speed up this brute force approach, we mainly did this on a PC and compared the result with either results from 64 bit integer calculations or precalculated reference results. The reference results have been created with the statistical computing environment R for a meaningful range. For example, the meaningful range for sine and cosine is from zero to two times Pi, meaning for an _Accum parameter, that 411774 calculations and comparisons had to be done. For functions that have no fixed execution time, the execution time over parameter is recorded and visualized via gnuplot.

5.1 Evaluation on the Microcontroller

To measure execution time and verify the calculation results, we wrote a small microcontroller program. To measure execution speed, we use the 16 bit timer. The counter is reset to zero, the function is called and the counter value is fetched afterwards. The execution time, parameters and result is then transmitted via UART. To speed up transmission, a high bit rate is used and the data is sent binary, so a conversion was needed to plot the data in gnuplot.

5.2 Accuracy Test

To test the accuracy of the implemented functions, we compared the output values with precise 64-bit calculations for the _Accum and _lAccum data type (respectively with precise 64-bit calculations for the _sAccum data type) for addition/subtraction, multiplication and division. For higher mathematical operation pre-calculated reference values have been used. The accuracy test was done on a PC as we use regular C code and the execution is much faster as on the microcontroller. We assumed equality of the output

after some calculations done on both, the PC and the microcontroller. The established Microsoft Visual C++ 98 environment was used as the reference compiler.

5.2.1 Multiplication and Division

To test multiplication and division, we simply treated the _Accum and _lAccum values as signed 64-bit integer values, repeated the calculations with 64-bit accuracy and compared the results. Testing has been done completely for the _sAccum type. For the other data types, extensive testing including the critical value pairs has been performed.

For a multiplication $x \cdot y$, all values $|x| > (2^{i+f} - 1) \cdot 2^{-f} / y$ will lead to an overflow for $y > 1$, with i being the number of integral bits an f being the number of fractional bits of the data type. For a division x/y, all values $x > (2^{i+f} - 1) \cdot 2^{-f} \cdot y$ will lead to an overflow. According to the data type, this is only a limitation for x if $y < 1$.

Extensive testing of the functions showed the following results:

- The _sAccum functions have a maximum error of 0 for multiplication and division, both tested with default and saturation behavior.
- The _Accum functions have a maximum error of 2^{-16} for multiplication and well-defined division calculations, both tested with default and saturation behavior.
- The _lAccum functions have a maximum error of 2^{-24} for multiplication and well-defined division calculations with default behavior. For saturation behavior, the maximum error was 2^{-23}.

5.2.2 Extended and Trigonometric Functions

For extended and trigonometric functions (e.g. sine/cosine, logarithm etc.), the comparison values were provided by R from the R Foundation, a statistical calculation environment. Most sophisticated functions have an error behavior that strongly depends on the input. Thus regarding the error over the whole input range makes sense. For example, Figure 1 depicts the error function for the _Accum square root function. An exhaustive evaluation of all functions can be found in the project documentation[10].

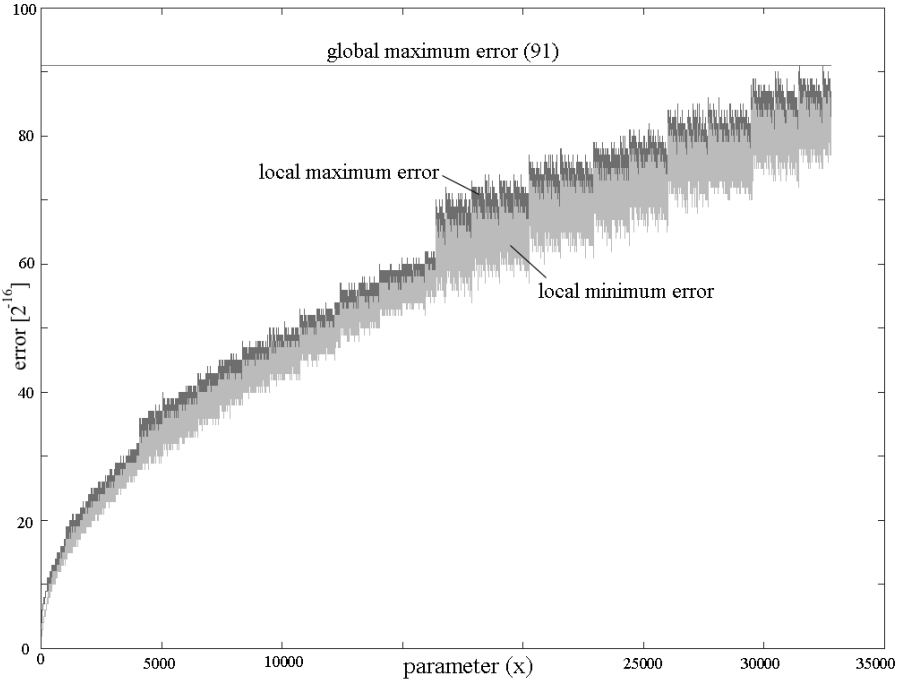

Figure 16-1. Accuracy distribution for *sqrk*

5.3 Performance Testing

Our first attempt to test performance of our implementation was to use the destination device, an Atmel ATMEGA 16, but as its maximum speed is 16 MHz and the serial port is a very slow transmission system, we decided to go a different way. We implemented a very simple simulator to test the performance on a PC.

5.3.1 The Disassembler & Simulator Creator (DsimC

The *Disassembler & Simulator Creator (DsimC)* is a little Java Application that disassembles an .srec-file for an Atmel ATMEGA16 and transforms each instruction into a piece of C code. This code can be compiled and executed on a PC instead of downloading and executing the original code on the microcontroller.

This was possible, because the ATMEGA16 has no caches or other elements that make code execution times indeterministic, but only a two-stage pipeline with very low effect on execution time. So each hardware instruction is expanded to a group of C code instructions which performs an

equivalent operation, maintains the virtual status register flags and increments a tick counter which furthermore can be used to determine the performance of the library functions. In addition, every write to the UART Data Register (UDR) results in a file output operation, which gives us a very high speedup. As assumed, the maintenance of the status register flags turned out to be most expensive, resulting in a simulation speed of only about 25 to 30 times faster than on the ATMEGA16 when using a Pentium-M with 2 GHz. This seems to be a good speedup, but most of it comes from the serial port implementation.

When we compared the performance values calculated by our simulations with values we determined on the microcontroller, we noticed a slight drift. It turned out that the simulator counts too many ticks under certain conditions, resulting in a few ticks more per function call, if ever. But when we tried to isolate the operations causing this drift, it turned out to be very tricky because of lack of an in-circuit debugger for the microcontroller we would have to flash the target many times to reduce the code range in which the drift appears. In our analysis we have noticed that the drift is only in one direction, if ever. Fortunately, the simulator never gives fewer ticks than it would take on the microcontroller, so this is sufficient to get guaranteed worst case execution time values. However, by taking the discrepancies between simulator and real hardware into account, tighter WCET values would be possible.

6. COMPARISON

The traditional way for fractional computing is the usage of floating-point operations, for which a various number of libraries exist. We compared our fixed-point library with the floating-point library (libm) that comes with the avr-gcc bundle.

6.1 Accuracy

All data types compared here (float, double, _Accum and _lAccum) reside in a 32-bit container. But the floating-point data types have to separate the container for exponent and mantissa, while the fixed-point data types have the whole container for the sign bit, the integral bits and the fractional bits. So, within the fixed-point range $(2^{31}\text{-}1) \cdot 2^{-16}$ and $(2^{31}\text{-}1) \cdot 2^{-24}$,

respectively, the _Accum and _lAccum types are more accurate as long as the value to be expressed matches the fixed-point range in its order of magnitude. The _sAccum data type has clearly a lower accuracy than the floating-point data types since it resides in a 16-bit container only.

6.2 Addition and Subtraction

The fixed-point addition and subtraction operations use the same instructions as normal integer operations, so they really make the cut over floating-point addition and subtraction (as shown in Table 2).

Table 16-2. Performance comparison for addition operations

Data type	Execution time in ticks
double	74 to 80
_sAccum	14
_Accum and _lAccum	23

6.3 Multiplication

Compared to the multiplication functions mulkD, mulkS, lmullkD and lmullkS the average performance of the double operation is higher, while the WCET of the fixed-point multiplication functions is better. This is due to the fact that the double data type is only implemented in 32 bit by the avr-gcc, thus a double multiplication involves a 23 bit multiplication of the mantissa and an addition of the exponent. In addition, the floating-point library has been optimized at assembly code level for average performance, which explains the results.

Table 16-3. Performance comparison of multiplication operations

	Execution time in ticks	
Data type	Default	Saturated
double	53 to 2851	-
_sAccum	79 to 82	92 to 95
_Accum	337 to 350	215 to 359
_lAccum	594 to 596	198 to 742

6.4 Division

Compared to the division functions divkD, divkS, ldivlkD and ldivlkS the minimum, average and maximum performance of the double operation is in general better, which can be seen in the overview given in Table 4. As a consequence, if possible, fixed-point divisions should be avoided for performance reasons.

Table 16-4. Performance comparison of division operations

	Execution time in ticks	
Data type	Default	Saturated
double	66 to 1385	-
_sAccum	634 to 711	650 to 727
_Accum	820 to 1291	853 to 1386
_lAccum	876 to 1405	862 to 1416

6.5 Floating-point Code Size

We have measured the code size for using the floating-point functions provided by the compiler. Using a simple addition, for example adds about 1740 bytes in code size. To cover the basic arithmetic operations about 3k of Flash ROM are needed. In contrast, when using AVRFix and the datatype _Accum with default behavior, only 758 bytes are needed. For _lAccum 848 bytes and for _sAccum only 260 bytes are needed. Thus, AVRFix has a clear advantage in code size compared to floating-point operations.

7. CONCLUSION

The contributions in this chapter are the implementation and evaluation of a generic fixed-point library based on the ISO/IEC Standard TR 18037. The documentation[10] and the source code is available as open source.

The fixed-point library contains not only basic mathematical functions and conversions but also more sophisticated operations such as square root, logarithmic and trigonometric functions. The linking model allows having only the used functions in the final assembler code, which saves considerable program memory over monolithic libraries.

We have performed exact performance measurements for a specific target architecture, the Atmel AVR with avr-gcc compiler. The results from this analysis can be used for static WCET analysis and optimization of execution time and code size, which is of special interest for embedded application on low-cost microcontrollers with few resources.

Addition and subtraction are generally by a factor of 3 to 5 faster than floating-point operations. The fixed-point multiplication and division have worse average performance, but a better WCET than the floating-point operations for most data types. Moreover, since the library has been written in C, there is also room for hardware-specific optimizations of the library, e.g., by using inline assembler functions for time-critical parts. Regarding code size the fixed-point operations are clearly in favor.

If only addition, subtraction and multiplication is needed or a small data type like _sAccum is sufficient, the use of fixed-point operations can clearly be favored from the viewpoints of speed, WCET, and code size. Sophisticated functions like trigonometric and exponential functions are slower than the fixed-point versions, but require less program memory, which makes the fixed-point implementation attractive for projects on small embedded microcontrollers where program memory becomes the main limiting factor. The optional saturation behavior is a nice feature which cannot easily be reproduced by floating-point calculations and small code size may be a decisive advantage.

In the future, we expect the fixed point standard to be supported by embedded compilers. The current version of gcc v4.3.1 supports the fixed point extensions only for the MIPS target. Being integrated into the compiler, we expect an increase in performance for compiler-supported fixed point arithmetic in comparison to our library. Until there is sufficient compiler support, our library can be a transitional solution that allows developers to use fixed-point arithmetic.

REFERENCES

1. ISO/IEC, Programming Languages -- C, approved by ANSI Accredited Standards Committee, ISO/IEC 9899:1999, December, 1999
2. IEEE, Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985; IEC-60559:1989, 1985
3. Altera Corporation, San Jose, CA, USA. Nios II Processor Reference Handbook Version 7.0, March 2007
4. D. P. Magee, Matlab extensions for the development, testing and verification of real-time DSP software, In Proceedings of the 42nd Annual Conference on Design Automation, pages 603–606, San Diego, CA, USA, 2005.
5. ISO/IEC, Programming languages -- C -- Extensions to support embedded processors, ISO/IEC TR 18037:2008, JTC 1/SC 22, 2008
6. P. Puschner, Worst-case execution time analysis at low cost, Control Engineering Practice, 6:129–135, January 1998.
7. D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys, 23(1):5--48, March, 1991
8. IEEE, Standard for Radix-independent Floating-point Arithmetic, ANSI/IEEE Std 854-1987, October 1987
9. G. Frantz and R. Simar, Comparing fixed- and floating-point DSPs. Texas Instruments, Dallas, TX, USA, 2004. White paper available at http://ocus.ti.com/lit/ml /spry061/spry061.pdf.
10. M. Rosenblattl and A. Wolf, Fixed-point library according to ISO/IEC standard DTR 18037 for Atmel AVR processors, Bachelor's thesis, Vienna University of Technology, Vienna, Austria, 2007. http://sourceforge.net/projects/avrfix.
11. J. E. Volder, The CORDIC trigonometric computing technique. IRE Transactions on Electronic Computers, Volume EC-8(3), 9 1959.