# AN4044
# Application note

## Using floating-point unit (FPU) with STM32F405/07xx and STM32F415/417xx microcontrollers

## Introduction

This application note explains how to use floating-point units (FPU) with STM32F405/07xx and STM32F415/417xx microcontrollers and provides a short overview of:

- Floating-point arithmetic
- STM32F405/07xx and STM32F415/417xx family floating-point unit

An application example is given at the end of this application note.

*Table 1* lists the microcontrollers and development tools concerned by this application note.

**Table 1. Applicable products and tools**

| Type | Applicable products |
|---|---|
| Microcontrollers | STM32F405/07xx and STM32F415/417xx high-performance MCUs with DSP and FPU instructions |
| Development tools | STM3240G-EVAL evaluation board |

# Contents


1 **Floating-point arithmetic . . . . . 5**

1.1 Fixed-point or floating-point . . . . . 5

1.2 Floating-point unit (FPU) . . . . . 6

2 **IEEE standard for floating-point arithmetic (IEEE 754) . . . . . 7**

2.1 Overview . . . . . 7

2.2 Number formats . . . . . 7

2.2.1 Normalized numbers . . . . . 8

2.2.2 Denormalized numbers . . . . . 9

2.2.3 Zeros . . . . . 9

2.2.4 Infinites . . . . . 9

2.2.5 NaN (Not-a-Number) . . . . . 9

2.2.6 Summary . . . . . 9

2.3 Rounding modes . . . . . 10

2.4 Arithmetic operations . . . . . 10

2.5 Number conversions . . . . . 10

2.6 Exception and exception handling . . . . . 10

2.7 Summary . . . . . 11

3 **STM32F4 floating-point unit (FPU) . . . . . 12**

3.1 Special operating modes . . . . . 12

3.2 Floating-point status and control register (FPSCR) . . . . . 12

3.2.1 Code condition bits: N, Z, C, V . . . . . 13

3.2.2 Mode bits: AHP, DN, FZ, RM . . . . . 13

3.2.3 Exception flags . . . . . 13

3.3 Exception management . . . . . 13

3.4 Programmers model . . . . . 14

3.5 FPU instructions . . . . . 14

3.5.1 FPU arithmetic instructions . . . . . 14

3.5.2 FPU compare & convert instructions . . . . . 14

3.5.3 FPU load/store instructions . . . . . 14

4 **Application example . . . . . 15**

4.1 Julia set . . . . . 15
4.2 Implementation on STM32F4 . . . . . 16
4.3 Results . . . . . 17
4.4 Conclusion . . . . . 18

**5 Reference documents . . . . . 19**

**6 Revision history . . . . . 20**

# List of tables

Table 1. Applicable products and tools . . . 1
Table 2. Integer numbers dynamic . . . 5
Table 3. Floating-point numbers dynamic . . . 5
Table 4. Normalized numbers range . . . 8
Table 5. Denormalized numbers range . . . 9
Table 6. Value range for IEEE.754 number formats . . . 9
Table 7. FPSCR register . . . 13
Table 8. Performance comparison with MDK-ARM version 4.22 . . . 17
Table 9. Reference documents . . . 19
Table 10. Document revision history . . . 20

# 1 Floating-point arithmetic

Floating-point numbers are used to represent non-integer numbers. They are composed of 3 fields:

- the sign
- the exponent
- the mantissa

Such a representation allows a very wide range of number coding, making floating-point numbers the best way to deal with real numbers. Floating-point calculations can be accelerated using a Floating-point unit (FPU) integrated in the processor.

## 1.1 Fixed-point or floating-point

One alternative to floating-point is fixed-point, where the exponent field is fixed. But if fixed-point is giving better calculation speed on FPUless processors, the range of numbers and their dynamic is low. As a consequence, a developer using the fixed-point technique will have to check carefully any scaling/saturation issues in its algorithm.

**Table 2. Integer numbers dynamic**

| Coding | Dynamic |
|---|---|
| **Int8** | 48 dB |
| **Int16** | 96 dB |
| **Int32** | 192 dB |
| **Int64** | 385 dB |

The C language offers the **float** and the **double** types for floating-point operations. At a higher level, modelization tools, such as matlab or scilab, are generating C code mainly using float or double. No floating-point support means modifying the generated code to adapt it to fixed-point. And all the fixed-point operations have to be handcoded by the programmer.

**Table 3. Floating-point numbers dynamic**

| Coding | Dynamic |
|---|---|
| **half precision** | 180 dB |
| **single precision** | 1529 dB |
| **double precision** | 12318 dB |

When used natively in code, floating-point operations will decrease the development time of a project. It is the most efficient way to implement any mathematical algorithm.

## 1.2 Floating-point unit (FPU)

Floating-point calculations require a lot of resources, as for any operation between two numbers. For example, we need to:

- align the two numbers (have them with the same exponent)
- perform the operation
- round out the result
- code the result

On an FPUless processor, all these operations are done by software through the C compiler library and are not visible to the programmer; but the performances are very low.

On a processor having an FPU, all of the operations are entirely done by hardware in a single cycle, for most of the instructions. The C compiler does not use its own floating-point library but directly generates FPU native instructions.

When implementing a mathematical algorithm on a microprocessor having an FPU, the programmer does not have to choose between performance and development time. The FPU brings reliability allowing to use directly any generated code through a high level tool, such as matlab or scilab, with the highest level of performance.

# 2 IEEE standard for floating-point arithmetic (IEEE 754)

The usage of the floating-point arithmetic has always been a need in computer science since the early ages. At the end of the 30's, when Konrad Zuse developed his Z series in Germany, floating-points were already in. But the complexity of implementing a hardware support for the floating-point arithmetic has discarded its usage for decades.

In the mid 50's, IBM, with its 704, introduced the FPU in mainframes; and in the 70's, various platforms were supporting floating-point operations but with their own coding techniques. The unification took place in 1985 when the IEEE published the standard 754 to define a common approach for floating-point arithmetic support.

## 2.1 Overview

The various types of floating-point implementations over the years led the IEEE to standardize the following elements:

- number formats
- arithmetic operations
- number conversions
- special values coding
- 4 rounding modes
- 5 exceptions and their handling

## 2.2 Number formats

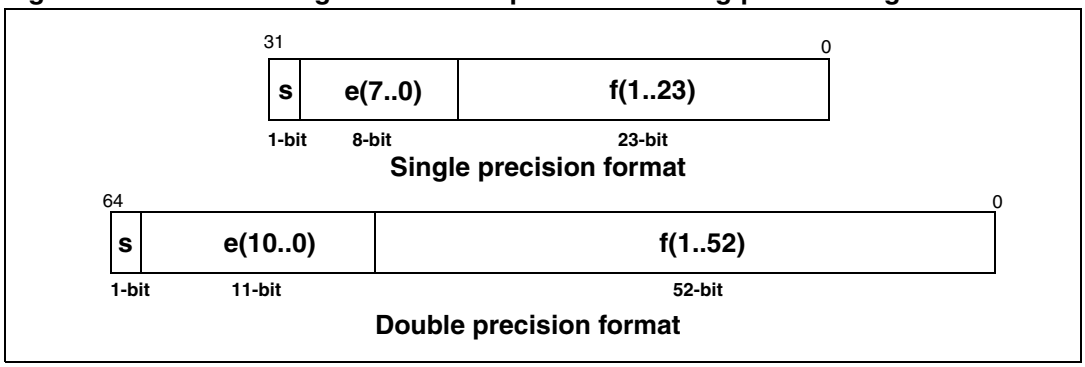
All values are composed of three fields:

- Sign: s
- Biased exponent:
  - sum of the exponent = e
  - constant value = bias
- Fraction (or mantissa): f

The values can be coded on various lengths:

- 16-bit: half precision format
- 32-bit: single precision format
- 64-bit: double precision format

**Figure 1. IEEE.754 single and double precision floating-point coding**




Five different classes of numbers have been defined by the IEEE:

- Normalized numbers
- Denormalized numbers
- Zeros
- Infinites
- NaN (Not-a-Number)

The different classes of numbers are identified by particular values of those fields.

### 2.2.1 Normalized numbers

A normalized number is a “standard” floating-point number. Its value is given by the above formula:

$$\text{Normalized Number} = (-1)^{s} \cdot (1 + \Sigma f_i.2^{-i}) \cdot 2^{e\text{-bias}} \text{ (with } i > 0)$$

The bias is a fixed value defined for each format (8-bit, 16-bit, 32-bit and 64-bit).

**Table 4. Normalized numbers range**

| Mode | Exponent | Exp. Bias | Exp. Range | Mantissa | Min. value | Max. Value |
|---|---|---|---|---|---|---|
| **Half** | 5-bit | 15 | -14, +15 | 10-bit | $6{,}10.10^{-5}$ | 65504 |
| **Single** | 8-bit | 127 | -126,+127 | 23-bit | $1{,}18.\ 10^{-38}$ | $3{,}40.10^{38}$ |
| **Double** | 11-bit | 1023 | -1022,+1023 | 52-bit | $2{,}23.10^{-308}$ | $1{,}8.10^{308}$ |

**Example:** single-precision coding of -7

- **Sign bit** = 1
- $7 = 1.75 \times 4 = (1 + 1/2 + 1/4) \times 4 = (1 + 1/2 + 1/4) \times 2^2$
- **Exponent** = 2 + bias = 2 + 127 = 129 = 0b10000001
- **Mantissa** = $2^{-1} + 2^{-2}$ = 0b11000000000000000000000
- **Binary value =** 0b 1 10000001 11000000000000000000000
- **Hexadecimal value =** 0xC0E00000

### 2.2.2 Denormalized numbers

A denormalized number is used to represent values which are too small to be normalized (when the exponent is equal to 0). Its value is given by the formula:

$$\text{Denormalized number} = (-1)^{s} \cdot (\Sigma f_i.2^{-i}) \cdot 2^{-bias} \text{ (with } i > 0)$$

**Table 5. Denormalized numbers range**

| Mode | Min value |
|---|---|
| **Half** | $5{,}96.10^{-8}$ |
| **Single** | $1{,}4.10^{-45}$ |
| **Double** | $4{,}94.10^{-324}$ |

### 2.2.3 Zeros

A Zero value is signed to indicate the saturation (positive or negative). Both exponent and fraction are null.

### 2.2.4 Infinites

An Infinite value is signed to indicate +oo or -oo. Infinite values are resulting of an overflow or a division by 0. The exponent is set to its maximum value, whereas the mantissa is null.

### 2.2.5 NaN (Not-a-Number)

A NaN is used for an undefined result of an operation, for example 0/0 or the square root of a negative number. The exponent is set to its maximum value, whereas the mantissa is not null. The MSB of the mantissa indicates if it is a Quiet NaN (which can be propagated through the next operations) or a Signaling NaN (which generates an error).

### 2.2.6 Summary

**Table 6. Value range for IEEE.754 number formats**

| Sign | Exponent | Fraction | Number |
|---|---|---|---|
| 0 | 0 | 0 | +0 |
| 1 | 0 | 0 | -0 |
| 0 | Max | 0 | +oo |
| 1 | Max | 0 | -oo |
| [0, 1] | Max | !=0 & MSB=1 | QNaN |
| [0, 1] | Max | !=0 & MSB=0 | SNaN |
| [0, 1] | 0 | !=0 | Denormalized Number |
| [0, 1] | [1, Max-1] | [0, Max] | Normalized Number |

## 2.3 Rounding modes

Four main rounding modes are defined:

- Round to nearest
- Direct rounding toward +oo
- Direct rounding toward -oo
- Direct rounding toward 0

Round to nearest is the default rounding mode (the most commonly used). If the two nearest are equally near, the selected one is the one with the LSB equal to 0.

The rounding mode is very important as it changes the result of an arithmetic operation. It can be changed through the FPU configuration register.

## 2.4 Arithmetic operations

The IEEE.754 standard defines 6 arithmetic operations:

- Add
- Subtract
- Multiply
- Divide
- Remainder
- Square root

## 2.5 Number conversions

The IEEE standard also defines some format conversion operations and comparison:

- Floating-point and integer conversion
- Round floating-point to integer value
- Binary-Decimal
- Comparison

## 2.6 Exception and exception handling

5 exceptions are supported:

- Invalid operation: the result of the operation is a NaN
- Division by zero
- Overflow: the result of the operation is +/-oo or +/-Max depending on the rounding mode
- Underflow: the result of the operation is a denormalized number
- Inexact result: caused by rounding

An exception can be managed in two ways:

- A trap can be generated. The trap handler returns a value to be used instead of an exceptional result.
- An interrupt can be generated. The interrupt handler cannot return a value to be used instead of an exceptional result.

## 2.7 Summary

The IEEE.754 standard defines how floating-point numbers are coded and processed.

An FPU implemented in hardware accelerates IEEE 754 floating point calculations. Thus, it can implement the whole IEEE standard or a subset. The associated software library manages the unaccelerated features.

For a “basic” usage, floating-point handling is transparent to the user, as if using `float` in C code. For more advanced applications, an exception can be managed through traps or interrupts.

# 3 STM32F4 floating-point unit (FPU)

The Cortex™M4 FPU is an implementation of the ARM™ FPv4-SP single-precision FPU.

It has its own 32-bit single precision register set (S0-S31) to handle operands and result. These registers can be viewed as 16 double-word registers (D0-15) for load/store operations.

A Status & Configuration Register stores the FPU configuration (rounding mode and special configuration), the condition code bits (negative, zero, carry and overflow) and the exception flags.

Some of the IEEE.754 operations are not supported by hardware and are done by software:

- Remainder
- Round floating-point to integer-value floating-point number
- Binary-to-decimal and decimal-to-binary conversions
- Direct comparison of single-precision and double-precision values

The exceptions are handled through interrupts (traps are not supported).

## 3.1 Special operating modes

The Cortex™M4 FPU is fully compliant with IEEE.754 specifications. However, some non-standard operating modes can be activated:

- Alternative Half-precision format (AHP control bit)
  - Specific 16-bit mode with no exponent value and no denormalized number support. Alternative Half-precision = (-1)s . (1 + Σfi.2-i) . 216
- Flush-to-zero mode (FZ control bit)
  - All the denormalized numbers are treated as zeros. A flag is associated to input and output flush.
- Default NaN mode (DN control bit)
  - Any operation with a NaN as an input, or which generates a NaN, returns the default NaN (Quiet NaN).

## 3.2 Floating-point status and control register (FPSCR)

The FPSCR stores the status (condition bit and exception flags) and the configuration (rounding modes and alternative modes) of the FPU.

As a consequence, this register may be saved in the stack when the context is changing.

FPSCR is accessed with dedicated instructions:

- Read: VMRS Rx, FPSCR
- Write: VMSR FPSCR, Rx

**Table 7. FPSCR register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Z | C | V | Res. | AHP | DN | FZ | RM | | Reserved | | | | | |
| | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | IDC | Reserved | | IXC | UFC | OFC | DZC | IOC |
| | | | | | | | | | | | rw | rw | rw | rw | rw |

### 3.2.1 Code condition bits: N, Z, C, V

They are set after a comparison operation.

### 3.2.2 Mode bits: AHP, DN, FZ, RM

They are configuring the alternative modes (AHP, DN, FZ) and the rounding mode (RM).

### 3.2.3 Exception flags

They are raised when an exception occurs in case of:

- Flush to zero (IDC)
- Inexact result (IXC)
- Underflow (UFC)
- Overflow (OFC)
- Division by zero (DZC)
- Invalid operation (IOC)

*Note: The exception flags are not reset by the next instruction.*

## 3.3 Exception management

Exceptions cannot be trapped. They are managed through the interrupt controller.

5 exception flags (IDC, UFC, OFC, DZC, IOC) are ORed and connected to the interrupt controller. There is no individual mask and the enable/disable of the FPU interrupt is done at the interrupt controller level.

The IXC flag is not connected to the interrupt controller and cannot generate an interrupt as its occurrence is very high. If needed, it must be managed by polling.

When the FPU is enabled, its context can be saved in the CPU stack using one of the three methods:

- No floating-point registers saving
- Lazy saving/restoring (only space allocation in the stack)
- Automatic floating-point registers saving/restoring

The stack frame consists of 17 entries:

- FPSCR
- S0 to S15

## 3.4 Programmers model

When the MCU is coming out of reset, the FPU has to be enabled specifying the access level of the code using the FPU (denied, privilege or full) in the Coprocessor Access Control Register (CPACR).

The FPSCR can be configured to define alternative modes or the rounding mode.

The FPU also has 5 system registers:

- FPCCR (FP Context Control Register) to indicate the context when the FP stack frame has been allocated, together with the context preservation setting.
- FPCAR (FP Context Address Register) to point to the stack location reserved for S0.
- FPDSCR (FP Default Status Control Register) where the default values for the Alternative half-precision mode, the Default NaN mode, the Flush-to-zero mode and the Rounding mode are stored.
- MVFR0 & MVFR1 (Media and VFP Feature Registers 0 and 1) where the supported features of the FPU are detailed.

## 3.5 FPU instructions

The FPU supports instructions for arithmetic operation, compare, convert and load/store.

### 3.5.1 FPU arithmetic instructions

The FPU offers arithmetic instructions for:

- Absolute value (1 cycle)
- Negate of a float or of multiple floats (1 cycle)
- Addition (1 cycle)
- Subtraction (1 cycle)
- Multiply, multiply accumulate/subtract, multiply accumulate/subtract then negate (3 cycles)
- Divide (14 cycles)
- Square root (14 cycles)

All the MAC operations can be standard or fused (rounding done at the end of the MAC for a better accuracy).

### 3.5.2 FPU compare & convert instructions

The FPU has compare instructions (1 cycle) and a convert instruction (1 cycle).

Conversion can be done between integer, fixed point, half precision and float.

### 3.5.3 FPU load/store instructions

The FPU follows the standard load/store architecture:

- Load and store on multiple doubles, multiple floats, single double or single float
- Move from/to core register, immediate of float or double
- Move from/to control/status register
- Pop and push double or float from/to the stack

# 4 Application example

The example given with this application note is showing the benefit brought by the STM32F4 FPU.

## 4.1 Julia set

The target is to compute a simple mathematical fractal: the Julia set.

The generation algorithm for such a mathematical object is quite simple: for each point of the complex plan, we are evaluating the divergence speed of a define sequence. The Julia set equation for the sequence is:

$$z_{n+1} = z_n^2 + c$$

For each x + i.y point of the complex plan, we compute the sequence with $c = c_x + i.c_y$:

$$x_{n+1} + i.y_{n+1} = x_n^2 - y_n^2 + 2.i.x_n.y_n + c_x + i.c_y$$

$$x_{n+1} = x_n^2 - y_n^2 + c_x \text{ and } y_{n+1} = 2.x_n.y_n + c_y$$

As soon as the resulting complex value is going out of a given circle (number's magnitude greater than the circle radius), the sequence is diverging, and the number of iterations done to reach this limit is associated to the point. This value is translated into a color, to show graphically the divergence speed of the points of the complex plan.

After a given number of iterations, if the resulting complex value remains in the circle, the calculation stops, considering the sequence is not diverging:

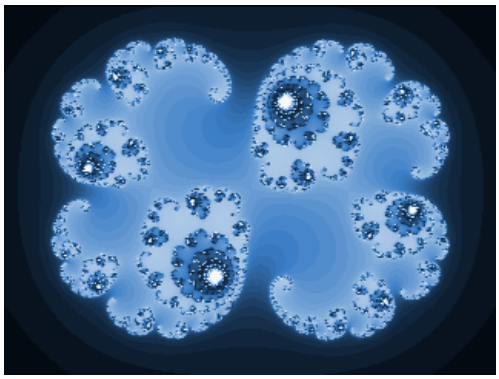
```
void GenerateJulia_fpu(uint16_t size_x, uint16_t size_y, uint16_t
offset_x, uint16_t offset_y, uint16_t zoom, uint8_t * buffer)
{
      float       tmp1, tmp2;
      float       num_real, num_img;
      float       radius;

      uint8_t           i;
      uint16_t        x,y;

      for (y=0; y<size_y; y++)
      {
            for (x=0; x<size_x; x++)
            {
                  num_real = y - offset_y;
                  num_real = num_real / zoom;
                  num_img = x - offset_x;
                  num_img = num_img / zoom;
                  i=0;
                  radius = 0;
                while ((i<ITERATION-1) && (radius < 4))
                  {
                        tmp1 = num_real * num_real;
                        tmp2 = num_img * num_img;
                       num_img = 2*num_real*num_img + IMG_CONSTANT;
```

```
                        num_real = tmp1 - tmp2 + REAL_CONSTANT;
                        radius = tmp1 + tmp2;
                        i++;
                    }
                    /* Store the value in the buffer */
                    buffer[x+y*size_x] = i;
                }
            }
        }
```

Such an algorithm is very efficient to show the benefits of the FPU: no code modification is needed, the FPU just needs to be activated or not during the compilation phase.

No additional code is needed to manage the FPU, as it is used in its default mode.

**Figure 2. Julia set with value coded on 8 bpp blue (c=0.285+i.0.01)**



## 4.2 Implementation on STM32F4

To have a better rendering on the RGB565 screen of the STM3240G-EVAL evaluation board, we are using a special palette to code the color values.

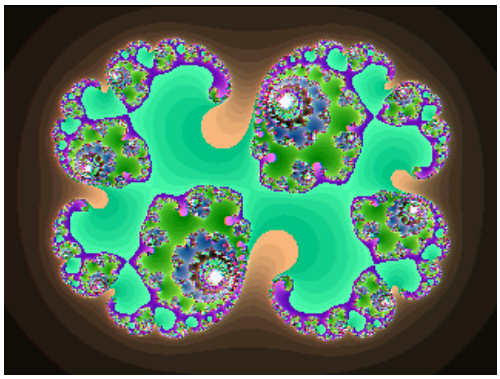
The maximum iteration value is set to 128. As a consequence, the color palette will have 128 entries. The circle radius is set to 2.

The main routine calls all the initialization functions of the board to set up the display and the buttons.

- The WAKUP button switches from automatic mode (continuous zoom in and out) to manual mode.
- In manual mode, the KEY button is used to launch another calculation, alternatively with and without an FPU, with performance comparison in between.

The whole project is compiled with the FPU enabled, except for GenerateJulia_noFPU.c which is compiled forcing the FPU off.

**Figure 3. Julia set with value coded on an RGB565 palette (c=0.285+i.0.01)**



## 4.3 Results

*Table 8* shows the time spent for the calculation of the Julia set, for several zooming factors, as shown in the demonstration firmware.

**Table 8. Performance comparison with MDK-ARM version 4.22**

| Frame | Zoom | Duration with FPU | Duration without FPU (microlib) | Ratio | Duration without FPU (no microlib) | Ratio |
|---|---|---|---|---|---|---|
| 0 | 120 | 222 | 3783 | 17.04 | 2597 | 11.70 |
| 1 | 110 | 194 | 3276 | 16.89 | 2243 | 11.56 |
| 2 | 100 | 167 | 2794 | 16.73 | 1906 | 11.41 |
| 3 | 150 | 298 | 5156 | 17.30 | 3558 | 11.94 |
| 4 | 200 | 312 | 5412 | 17.35 | 3740 | 11.99 |
| 5 | 275 | 296 | 5124 | 17.31 | 3540 | 11.96 |
| 6 | 350 | 284 | 4905 | 17.27 | 3389 | 11.93 |
| 7 | 450 | 289 | 4989 | 17.26 | 3448 | 11.93 |
| 8 | 600 | 273 | 4705 | 17.23 | 3251 | 11.91 |
| 9 | 800 | 267 | 4592 | 17.20 | 3173 | 11.88 |
| 10 | 1000 | 261 | 4485 | 17.18 | 3100 | 11.88 |
| 11 | 1200 | 255 | 4374 | 17.15 | 3023 | 11.85 |
| 12 | 1500 | 242 | 4138 | 17.10 | 2860 | 11.82 |
| 13 | 2000 | 210 | 3555 | 16.93 | 2455 | 11.69 |
| 14 | 1500 | 242 | 4138 | 17.10 | 2860 | 11.82 |
| 15 | 1200 | 255 | 4374 | 17.15 | 3023 | 11.85 |
| 16 | 1000 | 261 | 4485 | 17.18 | 3100 | 11.88 |
| 17 | 800 | 267 | 4592 | 17.20 | 3173 | 11.88 |

Table 8. Performance comparison with MDK-ARM version 4.22 (continued)

| Frame | Zoom | Duration with FPU | Duration without FPU (microlib) | Ratio | Duration without FPU (no microlib) | Ratio |
|---|---|---|---|---|---|---|
| 18 | 600 | 273 | 4705 | 17.23 | 3251 | 11.91 |
| 19 | 450 | 289 | 4989 | 17.26 | 3448 | 11.93 |
| 20 | 350 | 284 | 4905 | 17.27 | 3389 | 11.93 |
| 21 | 275 | 296 | 5123 | 17.31 | 3540 | 11.96 |
| 22 | 200 | 312 | 5412 | 17.35 | 3740 | 11.99 |
| 23 | 150 | 298 | 5156 | 17.30 | 3558 | 11.94 |
| 24 | 100 | 167 | 2794 | 16.73 | 1906 | 11.41 |
| 25 | 110 | 194 | 3276 | 16.89 | 2243 | 11.56 |

## 4.4 Conclusion

The FPU makes this very simple algorithm 11.5 to 17 times faster. No code modification has been done, just activating the FPU in the compiler options.

The STM32F4 FPU allows very fast mathematical computation on float and is a key benefit for many applications needing floating-point mathematical handling such as loop control, audio processing or audio decoding or digital filtering.

It makes the development faster and safer, from high level design tools to software generation.

# 5 Reference documents

**Table 9. Reference documents**

| Title | Author | Editor |
|---|---|---|
| The first computers<br>History and Architectures | Raul Rojas<br>Ulf Hashagen | MIT Press |
| IEEE754-2008 Standard | IEEE | IEEE |
| ARMv-7M Architecture Reference Manual | ARM | ARM |
| RM0090 Reference Manual | STMicroelectronics | STMicroelectronics |

# 6 Revision history

Table 10. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 16-Mar-2012 | 1 | Initial release. |