

---

**Floating point unit demonstration on STM32 microcontrollers**

---

**Introduction**

This application note explains how to use floating-point units (FPUs) available in STM32 Cortex<sup>®</sup>-M4 and STM32 Cortex<sup>®</sup>-M7 microcontrollers, and also provides a short overview of: floating-point arithmetic.

The X-CUBE-FPUDEMO firmware is developed to promote double precision FPUs, and to demonstrate the improvements coming from the use of this hardware implementation.

Two examples are given in [Section 4: Application example](#).

# Contents

- 1 Floating-point arithmetic ..... 6**
  - 1.1 Fixed-point or floating-point ..... 6
  - 1.2 Floating-point unit (FPU) ..... 7
  
- 2 IEEE standard for floating-point arithmetic (IEEE 754) ..... 8**
  - 2.1 Overview ..... 8
  - 2.2 Number formats ..... 8
    - 2.2.1 Normalized numbers ..... 9
    - 2.2.2 Denormalized numbers ..... 10
    - 2.2.3 Zeros ..... 10
    - 2.2.4 Infinities ..... 10
    - 2.2.5 NaN (Not-a-Number) ..... 10
    - 2.2.6 Summary ..... 10
  - 2.3 Rounding modes ..... 11
  - 2.4 Arithmetic operations ..... 11
  - 2.5 Number conversions ..... 11
  - 2.6 Exception and exception handling ..... 11
  - 2.7 Summary ..... 12
  
- 3 STM32 Cortex®-M floating-point unit (FPU) ..... 13**
  - 3.1 Special operating modes ..... 14
  - 3.2 Floating-point status and control register (FPSCR) ..... 14
    - 3.2.1 Code condition bits: N, Z, C, V ..... 14
    - 3.2.2 Mode bits: AHP, DN, FZ, RM ..... 14
    - 3.2.3 Exception flags ..... 15
  - 3.3 Exception management ..... 15
  - 3.4 Programmers model ..... 15
  - 3.5 FPU instructions ..... 16
    - 3.5.1 FPU arithmetic instructions ..... 16
    - 3.5.2 FPU compare & convert instructions ..... 17
    - 3.5.3 FPU load/store instructions ..... 17
  
- 4 Application example ..... 18**

---

4.1	Julia set .....	18
4.2	Implementation on STM32F4 .....	19
4.3	Implementation on STM32F7 .....	20
4.4	Results .....	22
4.5	Mandelbrot set .....	25
4.6	Conclusion .....	28
<b>5</b>	<b>Reference documents .....</b>	<b>29</b>
<b>6</b>	<b>Revision history .....</b>	<b>30</b>

## List of tables

Table 1.	Integer numbers dynamic . . . . .	6
Table 2.	Floating-point numbers dynamic. . . . .	6
Table 3.	Normalized numbers range . . . . .	9
Table 4.	Denormalized numbers range . . . . .	10
Table 5.	Value range for IEEE.754 number formats . . . . .	10
Table 6.	FPU implementation within the STM32 Cortex®-M4/-M7 . . . . .	13
Table 7.	FPSCR register. . . . .	14
Table 8.	Some floating-point single-precision data processing instructions . . . . .	16
Table 9.	Some floating-point double-precision data processing instructions . . . . .	16
Table 10.	Cortex®-M4 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM™ tool-chain V5.17. . . . .	22
Table 11.	Cortex®-M7 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM™ tool-chain V5.17 . . . . .	23
Table 12.	Performance comparison HW DP FPU versus SW implementation FPU with MDK-ARM™ tool-chain V5.17. . . . .	24
Table 13.	Reference documents. . . . .	29
Table 14.	Document revision history . . . . .	30

## List of figures

Figure 1. IEEE.754 single and double precision floating-point coding . . . . . 9  
Figure 2. Julia set with value coded on 8 bpp blue ( $c=0.285+i.0.01$ ) . . . . . 19  
Figure 3. Julia set with value coded on an RGB565 palette ( $c=0.285+i.0.01$ ) . . . . . 20  
Figure 4. Configure FPU with MDK-ARM™ tool-chain V5.17 . . . . . 20  
Figure 5. Picture of Mandelbrot-set with zoom in =1 . . . . . 26  
Figure 6. Picture of Mandelbrot-set using Double precision FPU with zoom in 48 times. . . . . 27  
Figure 7. Picture of Mandelbrot-set using Single precision FPU with zoom in 32 times . . . . . 27

# 1 Floating-point arithmetic

Floating-point numbers are used to represent non-integer numbers. They are composed of three 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 FPU-less 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 the algorithm.

**Table 1. 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 hand-coded by the programmer.

**Table 2. 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 FPU-less 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
- four rounding modes
- five exceptions and their handling

### 2.2 Number formats

All values are composed of three fields:

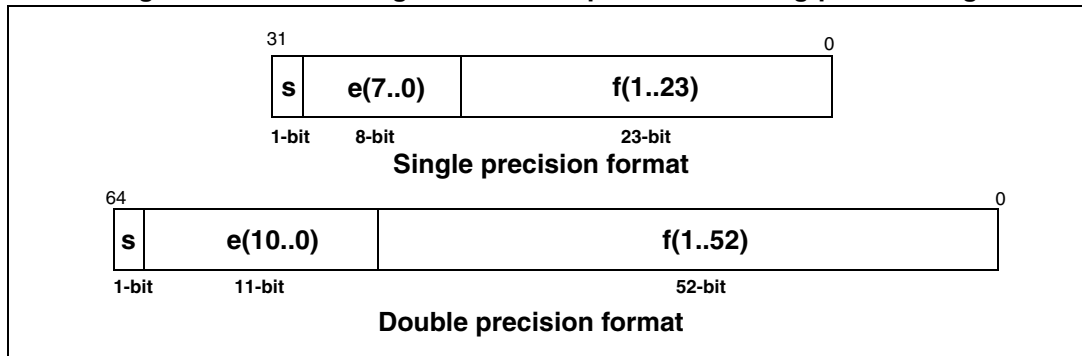
- Sign:  $s$
- Biased exponent:
  - sum of the exponent =  $e$
  - constant value = bias
- Fraction (or mantissa):  $f$

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:

$$\text{Normalized number} = (-1)^s \times (1 + \sum f_i \times 2^{-i}) \times 2^{e-bias} \quad (\text{with } i > 0)$$

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

Table 3. Normalized numbers range

Mode	Exponent	Exp. Bias	Exp. Range	Mantissa	Min. value	Max. Value
Half	5-bit	15	-14, +15	10-bit	$6,10 \cdot 10^{-5}$	65504
Single	8-bit	127	-126,+127	23-bit	$1,18 \cdot 10^{-38}$	$3,40 \cdot 10^{38}$
Double	11-bit	1023	-1022,+1023	52-bit	$2,23 \cdot 10^{-308}$	$1,8 \cdot 10^{308}$

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** =  $2^{-1} + 2^{-2} = 0b110000000000000000000000$
- **Binary value** = 0b 1 10000001 110000000000000000000000
- **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:

$$\text{Denormalized number} = (-1)^s \times (\sum f_i \times 2^{-i}) \times 2^{-bias} \quad (\text{with } i > 0)$$

**Table 4. Denormalized numbers range**

Mode	Min value
Half	$5,96 \cdot 10^{-8}$
Single	$1,4 \cdot 10^{-45}$
Double	$4,94 \cdot 10^{-324}$

## 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  $+\infty$  or  $-\infty$ . 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 5. Value range for IEEE.754 number formats**

Sign	Exponent	Fraction	Number
0	0	0	+0
1	0	0	-0
0	Max	0	$+\infty$
1	Max	0	$-\infty$
[0, 1]	Max	$\neq 0$ & MSB=1	QNaN
[0, 1]	Max	$\neq 0$ & MSB=0	SNaN
[0, 1]	0	$\neq 0$	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  $+\infty$
- Direct rounding toward  $-\infty$
- 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  $\pm\infty$  or  $\pm\text{Max}$  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 STM32 Cortex®-M floating-point unit (FPU)

*Table 6* shows the implementation of FPU for STM32 Cortex®-M4 and Cortex®-M7.

**Table 6. FPU implementation within the STM32 Cortex®-M4/-M7**

Features	Configurable options	STM32 Implementation	
		STM32F3xx STM32F4xx STM32F74x/5x STM32L4xx	STM32F76x/7x
FPU	No FPU	-	-
	Single Precision (SP) only	Yes	-
	SP and DP	-	Yes

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).

The Cortex®-M7 double precision FPU is an implementation of the ARM® FPv5 floating point. The FPv5 fully supports single-precision and double-precision, it also provides conversions between fixed-point and floating-point data formats, and floating-point constant instructions.

The FPU provides IEEE754-compliant operations on 32-bit single-precision and 64-bit double-precision floating-point values.

The FPU provides an extension register file containing 32 single-precision registers. These can be viewed as:

- Sixteen 64-bit double word registers, (D0-D15), which is the same as for the FPv4 with no additional registers.
- Thirty-two 32-bit single-word registers, (S0-S31), load/store instructions are identical to the supported instructions by the FPv4 which already includes support for 64-bit data types.

The FPv5 provides a hardware support for denormals and all IEEE Standard 754-2008 rounding modes.

### 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.

$$\text{Alternative Half-precision} = (-1)^s \times (\sum f_i \times 2^{-i}) \times 2^{16}$$

- 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
											r/w	r/w	r/w	r/w	r/w

#### 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.

Five 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)

[Table 8](#) shows some of the floating-point single-precision data processing instructions:

**Table 8. Some floating-point single-precision data processing instructions**

Instruction	Description	Cycles
VABS.F32	Absolute value	1
VADD.F32	Addition	1
VSUB.F32	Subtraction	1
VMUL.F32	Multiply	1
VDIV.F32	Division	14
VCVT.F32	Conversion to/from integer/fixed-point	1
VSQRT.F32	Square root	14

[Table 9](#) shows some of the floating-point double-precision data processing instructions:

**Table 9. Some floating-point double-precision data processing instructions**

Instruction	Description	Cycles
VADD.F64	Addition	3
VSUB.F64	Subtraction	3
VCVT.F<32 64>	Conversion to/from Integer/fixed-point	3

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

Two examples are given with this application note that show the benefit brought by the STM32 FPU.

The first example is Julia set, which highlights performances comparison between the hardware FPU versus the software one.

The second example is Mandelbrot set, which highlights the gain in precision with the hardware double precision FPU versus the single precision 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  $c = c_x + i.c_y$ :

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

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;
        }
    }
}
```

```

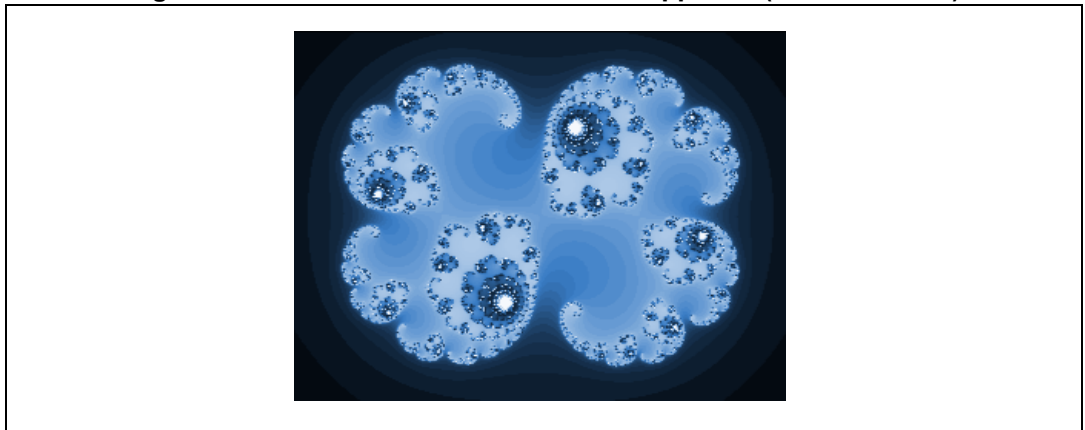
        radius = 0;
    while ((i<ITERATION-1) && (radius < 4))
    {
        tmp1 = num_real * num_real;
        tmp2 = num_img * num_img;
        num_img = 2*num_real*num_img + IMG_CONSTANT;
        num_real = tmp1 - tmp2 + REAL_CONSTANT;
        radius = tmp1 + tmp2;
        i++;
    }
    /* Store the value 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.

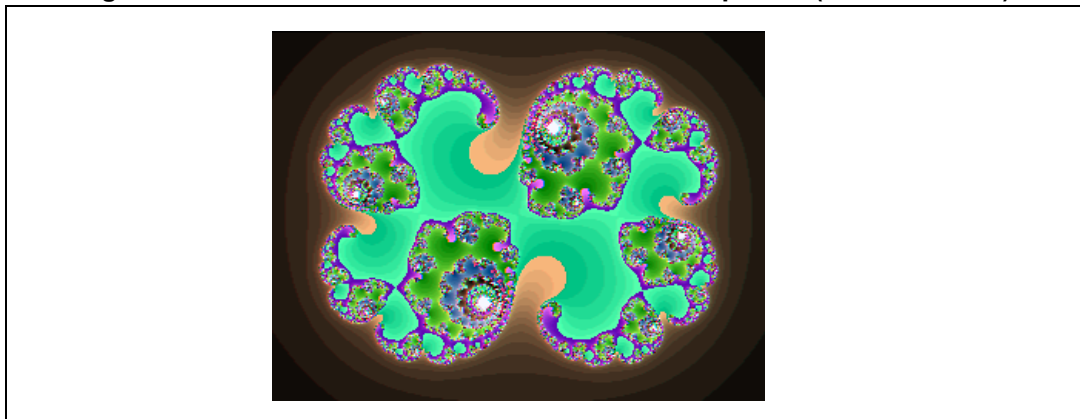
The maximum iteration value is set to 128. As a consequence, the color palette will have 128 entries. The circle radius is set to 2.

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.

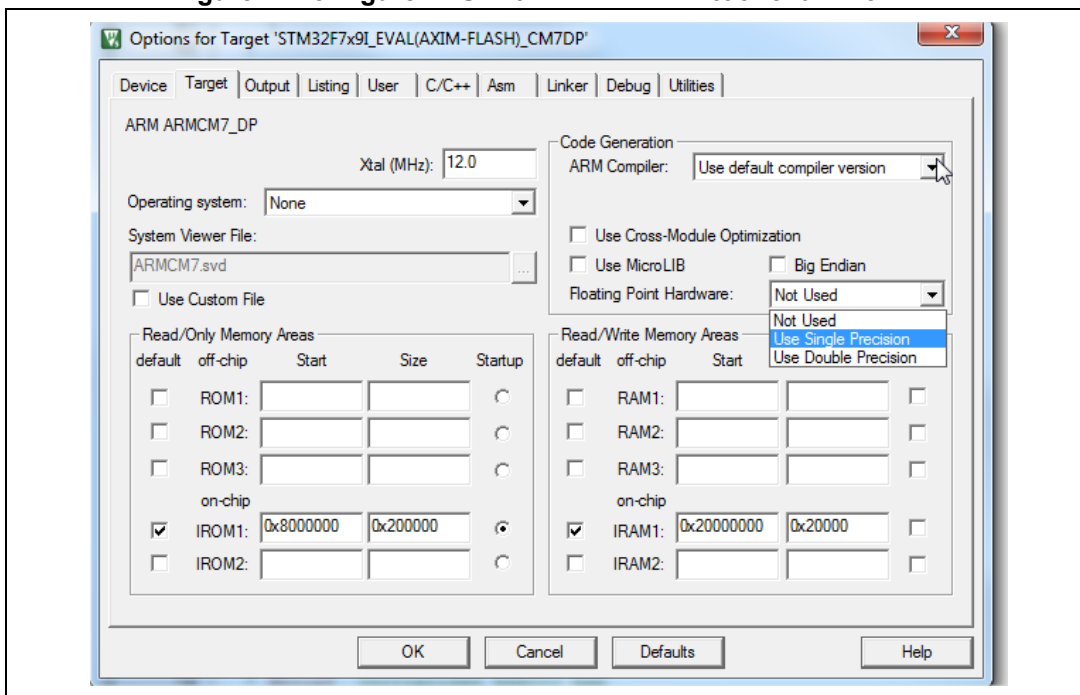
Figure 3. Julia set with value coded on an RGB565 palette ( $c=0.285+i.0.01$ )



### 4.3 Implementation on STM32F7

The same algorithm is implemented on the STM32F769i-Eval. The microcontroller is running at 216 MHz, with the following two configurations: FPU single precision enabled and FPU double precision enabled. This is done through the RealView Microcontroller Development Kit (MDK-ARM™) tool-chain V5.17 as shown in Figure 4.

Figure 4. Configure FPU with MDK-ARM™ tool-chain V5.17



Only the manual mode is available for the STM32F7, once a touch screen is detected, this will launch another calculation.

The algorithm has been changed too:

```
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)
{
    double      tmp1, tmp2;
    double      num_real, num_img;
    double      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;
        }
    }
}
```

## 4.4 Results

*Table 10* shows the time spent by the Cortex<sup>®</sup>-M4 based STM32F4 to calculate the Julia set, for several zooming factors, as shown in the demonstration firmware.

**Table 10. Cortex<sup>®</sup>-M4 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM<sup>™</sup> tool-chain V5.17**

Frame	Zoom	Duration with HW FPU [ms]	Duration with SW implementation FPU [ms]	Ratio
0	120	195	2426	12,44
1	110	170	2097	12,34
2	100	146	1782	12,21
3	150	262	3323	12,68
4	200	275	3494	12,71
5	275	261	3307	12,67
6	350	250	3165	12,66
7	450	254	3221	12,68
8	600	240	3038	12,66
9	800	235	2965	12,62
10	1000	230	2896	12,59
11	1200	224	2824	12,61
12	1500	213	2672	12,54
13	2000	184	2293	12,46
14	1500	213	2672	12,54
15	1200	224	2824	12,61
16	1000	230	2896	12,59
17	800	235	2965	12,62
18	600	240	3038	12,66
19	450	254	3221	12,68
20	350	250	3165	12,66
21	275	261	3307	12,67
22	200	275	3494	12,71
23	150	262	3323	12,68
24	100	146	1781	12,20
25	110	170	2097	12,34

Table 11 shows the time spent by the Cortex<sup>®</sup>-M7 based STM32F7 to calculate the Julia set with the same algorithm running on the Cortex<sup>®</sup>-M4 based STM32F4, for several zooming factors, as shown in the demonstration firmware.

**Table 11. Cortex<sup>®</sup>-M7 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM<sup>™</sup> tool-chain V5.17**

Frame	Zoom	Duration with HW FPU [ms]	Duration with SW implementation FPU [ms]	Ratio
0	120	134	1759	13,13
1	110	118	1519	12,87
2	100	102	1291	12,66
3	150	179	2407	13,45
4	200	187	2529	13,52
5	275	178	2396	13,46
6	350	171	2294	13,42
7	450	174	2335	13,42
8	600	165	2204	13,36
9	800	161	2150	13,35
10	1000	157	2101	13,38
11	1200	154	2048	13,30
12	1500	146	1936	13,26
13	2000	127	1661	13,08
14	1500	146	1936	13,26
15	1200	154	2048	13,30
16	1000	157	2101	13,38
17	800	161	2150	13,35
18	600	165	2204	13,36
19	450	174	2335	13,42
20	350	171	2294	13,42
21	275	178	2396	13,46
22	200	187	2529	13,52
23	150	179	2407	13,45
24	100	102	1291	12,66
25	110	118	1519	12,87

Table 12 shows the time spent by the Cortex®-M7 based STM32F7 to calculate the Julia set with the above described algorithm, for several zooming factors, as shown in the demonstration firmware.

**Table 12. Performance comparison HW DP FPU versus SW implementation FPU with MDK-ARM™ tool-chain V5.17**

Frame	Zoom	Duration with HW DP FPU [ms]	Duration with SW implementation FPU [ms]	Ratio
0	120	408	2920	7,16
1	110	355	2523	7,11
2	100	305	2145	7,03
3	150	550	3995	7,26
4	200	577	4197	7,27
5	275	547	3971	7,26
6	350	524	3799	7,25
7	450	533	3866	7,25
8	600	504	3643	7,23
9	800	492	3557	7,23
10	1000	481	3476	7,23
11	1200	470	3390	7,21
12	1500	446	3206	7,19
13	2000	386	2752	7,13
14	1500	446	3206	7,19
15	1200	470	3390	7,21
16	1000	481	3476	7,23
17	800	492	3557	7,23
18	600	504	3643	7,23
19	450	533	3866	7,25
20	350	524	3799	7,25
21	275	547	3971	7,26
22	200	577	4197	7,27
23	150	550	3995	7,26
24	100	305	2145	7,03
25	110	355	2523	7,11

We can observe that the ratio between using the hardware SP FPU versus the software implementation of the FPU is better than the ratio between using the hardware DP FPU versus the software implementation of the FPU, user should use “double” only when he needs to get more precisions. However if user seeks to get better performances and to reduce RAM usage, he should use the “float”.



## 4.5 Mandelbrot set

To generate a Mandelbrot set, we used the same iterative function as the Julia Set, the  $c$  variable will represent the position of the pixel and  $z$  will start at the position ( $x=0, y=0$ ).

```
void drawMandelbrot_Double(float centre_X, float centre_Y, float Zoom,
uint16_t IterationMax)
{
    double X_Min = centre_X - 1.0/Zoom;
    double X_Max = centre_X + 1.0/Zoom;
    double Y_Min = centre_Y - (YSIZE_PHYS-CONTROL_SIZE_Y) / (XSIZE_PHYS *
Zoom);
    double Y_Max = centre_Y + (YSIZE_PHYS-CONTROL_SIZE_Y) / (XSIZE_PHYS
*Zoom);

    double dx = (X_Max - X_Min) / XSIZE_PHYS;
    double dy = (Y_Max - Y_Min) / (YSIZE_PHYS-CONTROL_SIZE_Y) ;

    double y = Y_Min;

    double c;

    for (uint16_t j = 0; j < (YSIZE_PHYS-CONTROL_SIZE_Y); j++)
    {
        double x = X_Min;

        for (uint16_t i = 0; i < XSIZE_PHYS; i++)
        {
            double Zx = x;
            double Zy = y;
            int n = 0;
            while (n < IterationMax)
            {
                double Zx2 = Zx * Zx;
                double Zy2 = Zy * Zy;
                double Zxy = 2.0 * Zx * Zy;

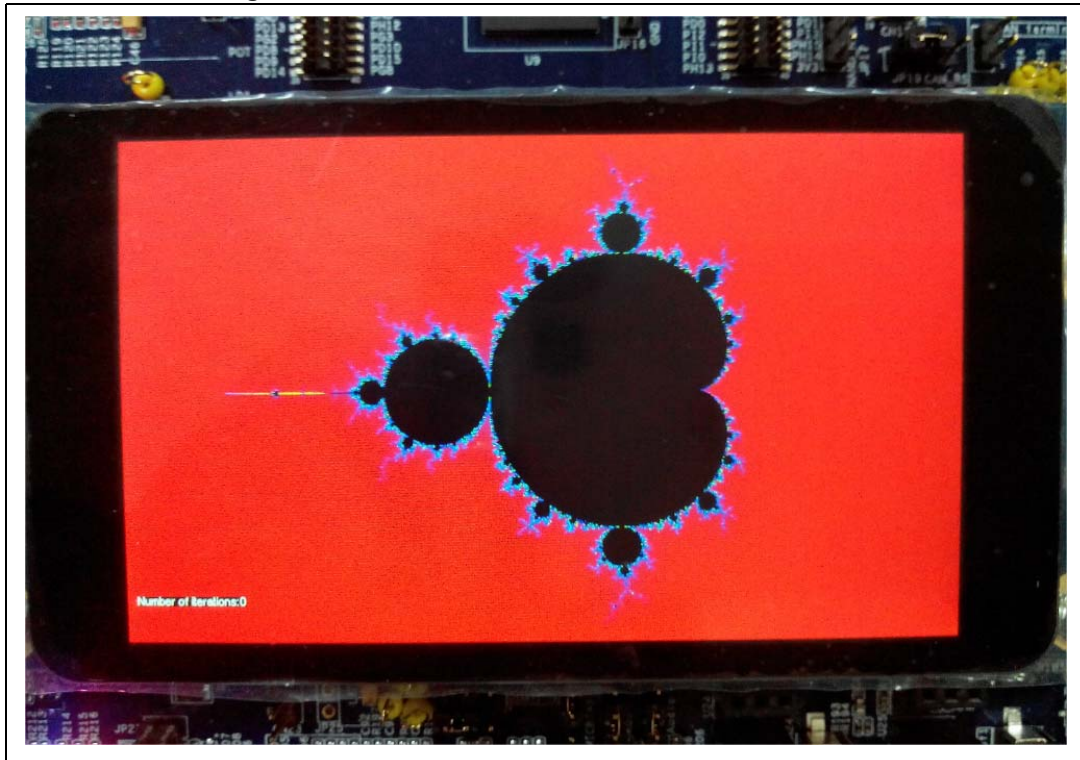
                Zx = Zx2 - Zy2 + x;
                Zy = Zxy + y;

                if(Zx2 + Zy2 > 16.0)
                {
                    break;
                }
                n++;
            }
            x += dx;
        }
    }
}
```

```
    }  
    y += dy;  
}  
}
```

*Figure 5* shows the image generated with zoom=1.

**Figure 5. Picture of Mandelbrot-set with zoom in =1**

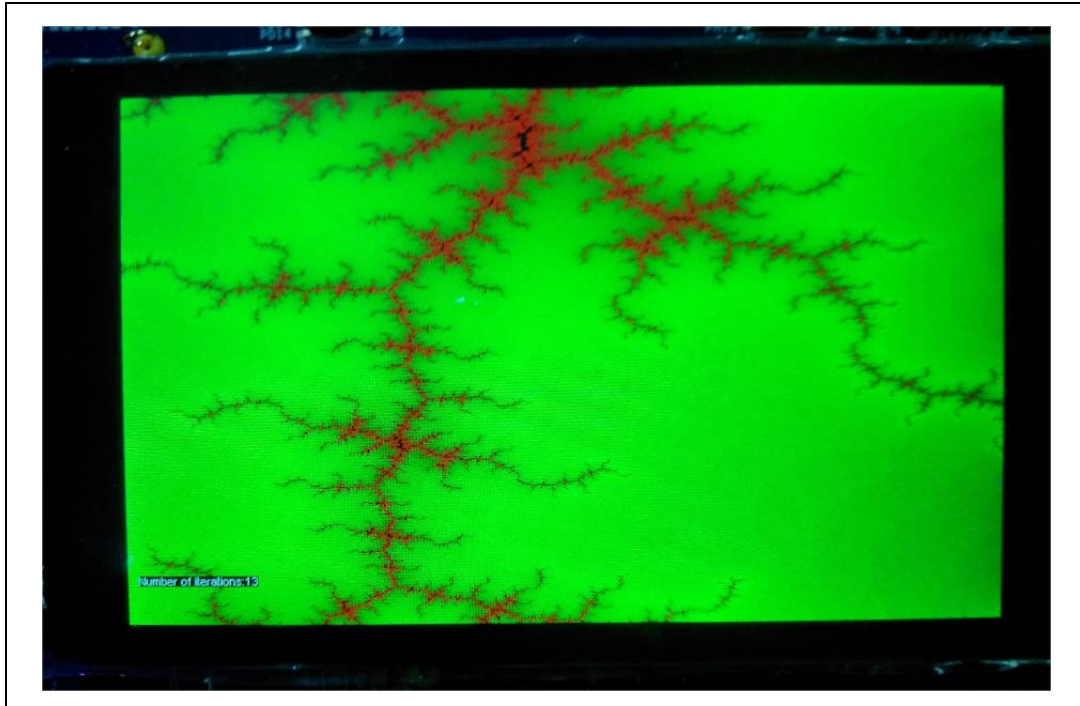


Every time the user touches the screen the picture will be zoomed in four times.

*Figure 6* shows a picture that was zoomed in so much, that the numerical limit of 64-bit floating point numbers was reached.

It starts looking blocky. It's zoomed in more than 48 times though.

Figure 6. Picture of Mandelbrot-set using Double precision FPU with zoom in 48 times



The same algorithm used with single precision FPU. [Figure 7](#) shows that the picture starts looking blocky after zooming in 32 times.

Figure 7. Picture of Mandelbrot-set using Single precision FPU with zoom in 32 times



Using the hardware double precision, FPU will not only allow us to gain in calculation time as we already seen when running Julia Set, but also it allows us to gain in precision.

## 4.6 Conclusion

The hardware FPU makes the Julia Set algorithm 12.5 times faster when we are using “float”, and 7.2 times faster when we are using “double”. No code modification is needed, the FPU is activated in the compiler options.

It also allows a wider range for precision.

The STM32 FPU allows very fast mathematical computation on float and double.

FPU 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 13. Reference documents**

<b>Title</b>	<b>Author</b>	<b>Editor</b>
The first computers History and Architectures	Raul Rojas Ulf Hashagen	MIT Press
IEEE754-2008 Standard	IEEE	IEEE
ARMv-7M Architecture Reference Manual	ARM	ARM
ARM <sup>®</sup> Cortex <sup>®</sup> -M7 Processor	ARM	ARM
RM0385 Reference manual	STMicroelectronics	STMicroelectronics
RM0090 Reference Manual	STMicroelectronics	STMicroelectronics

## 6 Revision history

**Table 14. Document revision history**

Date	Revision	Changes
16-Mar-2012	1	Initial release.
30-May-2016	2	<p>Updated:</p> <ul style="list-style-type: none"> <li>– <i>Introduction</i></li> <li>– <i>Section 3: STM32 Cortex®-M floating-point unit (FPU)</i></li> <li>– <i>Section 4.6: Conclusion</i></li> <li>– <i>Table 10: Cortex®-M4 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM™ tool-chain V5.17</i></li> <li>– <i>Table 13: Reference documents</i></li> </ul> <p>Added:</p> <ul style="list-style-type: none"> <li>– <i>Table 6: FPU implementation within the STM32 Cortex®-M4/-M7</i></li> <li>– <i>Table 7: FPSCR register</i></li> <li>– <i>Table 8: Some floating-point single-precision data processing instructions</i></li> <li>– <i>Table 9: Some floating-point double-precision data processing instructions</i></li> <li>– <i>Table 11: Cortex®-M7 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM™ tool-chain V5.17</i></li> <li>– <i>Figure 4: Configure FPU with MDK-ARM™ tool-chain V5.17</i></li> <li>– <i>Figure 5: Picture of Mandelbrot-set with zoom in =1</i></li> <li>– <i>Figure 6: Picture of Mandelbrot-set using Double precision FPU with zoom in 48 times</i></li> <li>– <i>Figure 7: Picture of Mandelbrot-set using Single precision FPU with zoom in 32 times</i></li> </ul> <p>Removed former <i>Table 1: Applicable products and tools.</i></p>

**IMPORTANT NOTICE – PLEASE READ CAREFULLY**

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2016 STMicroelectronics – All rights reserved