AN14712

Advanced PowerQuad Operation Guide

Rev. 1.0 — 30 June 2025 Application note

Document information

Information	Content
Keywords	AN14712, MCX N series, PowerQuad, and FRDM-MCXN947
Abstract	This application note provides some advice to optimize PowerQuad based algorithm.

1 Introduction

This application note provides some information, code snippets, and tips to help users accelerate their calculations with MAU. The MCX N Series microcontrollers feature a powerful and efficient coprocessor called PowerQuad. It operates in parallel with the CPUs to offload intensive mathematical computations and enhance overall performance.

PowerQuad is a DSP accelerator designed to assist a Cortex-M CPU. It includes seven internal computation engines:

- Transform
- Transcendental function
- Trigonometry function
- Dual Biquad infinite impulse responses (IIR) filter
- Matrix accelerator
- Finite impulse responses (FIR) filter
- Coordinate rotation digital computer (CORDIC)

2 PowerQuad pre or post scaling

If the input or output data format is a fixed-point number except for FFT computation, a scaling factor applies for the PowerQuad AHB memory-mapped engine. This is useful when the users prefer fixed-point numbers in their algorithm. However, converting between floating-point and fixed-point numbers always consumes a significant amount of CPU time.

The following example code shows how to use the pre scaling feature of PowerQuad to reduce CPU loading.

```
// Convert floating point number into fixed point number.
    {
        float input[16] = {0};
        int32_t output[16] = {0};

        for (uint32_t i = 0; i < 16; i++) {
            input[i] = 100.0f;
        }

        pq_config_t pqCfg;
        PQ_GetDefaultConfig(&pqCfg);
        pqCfg.outputFormat = kPQ_32Bit;
        pqCfg.outputPrescale = 23; // Convert to Q8.23, 23 fractional bits.
        PQ_SetConfig(POWERQUAD, &pqCfg);
        PQ_MatrixScale(POWERQUAD, POWERQUAD_MAKE_MATRIX_LEN(4, 4, 4), 1.0f,
 (const void *)input, (void *)output);
        PQ_WaitDone(POWERQUAD);

        for (uint32_t i = 0; i < 16; i++) {
            // 100 in fixed point Q8.23
            assert(output[i] == 838860800);
        }
    }
```

```
// Convert fixed point number into floating point number
    {
        int32_t input[16] = {0};
        float output[16] = {0};
```

```
        for (uint32_t i = 0; i < 16; i++) {
            // 100 in fixed point Q8.23
            input[i] = 838860800;
        }

        pq_config_t pqCfg;
        PQ_GetDefaultConfig(&pqCfg);
        pqCfg.inputAFormat = kPQ_32Bit;
        pqCfg.inputAPrescale = -23; // Q8.23, 23 fractional bits.
        PQ_SetConfig(POWERQUAD, &pqCfg);
        PQ_MatrixScale(POWERQUAD, POWERQUAD_MAKE_MATRIX_LEN(4, 4, 4), 1.0f,
 (const void *)input, (void *)output);
        PQ_WaitDone(POWERQUAD);

        for (uint32_t i = 0; i < 16; i++) {
            assert(output[i] == 100.f);
        }
    }
```

The scaling feature enables PowerQuad to handle fixed-point numbers with different fractional bits. In typical use cases, such as sampling data from external sensors, users often want to extract features from the data. This process generally involves applying a window function to the original data, followed by other computation such as FFT. This requires an element-wise loop to apply the window and shift bits to align the fractional bits for subsequent calculation. This operation often consumes significant CPU time; however, with the acceleration of PowerQuad, it can be executed efficiently, reducing the computational load on the CPU.

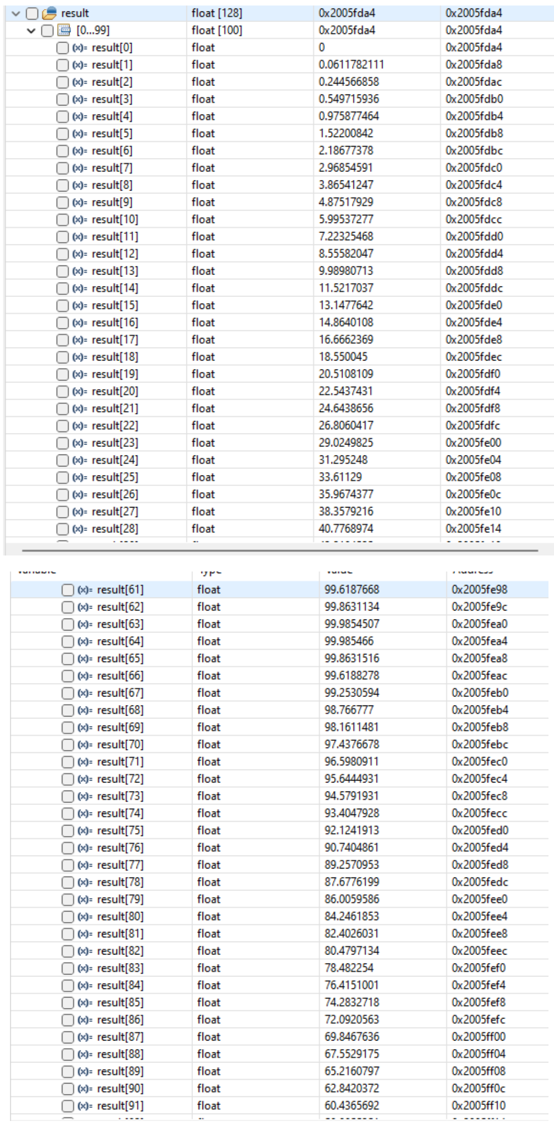
```
// Use Case:
    // Apply fixed-point window to signals from fixed-point external sensor or
 ADC.
    {
        int32_t inputA[128] = {0}; // Q8.23
        int32_t window[128] = {0}; // Q0.31
        float result[128] = {0};

        // Initialize HANN window
        for (uint32_t i = 0; i < 128; i++) {
            float w = 0.5 * (1 - cosf(2 * M_PI * i / (128 - 1)));
            window[i] = scalbnf(fmaxf(fminf(w, 0x0.FFFFFFp0F), -1.0F), 31);
        }
        // 100.0f in Q8.23 format.
        for (uint32_t i = 0; i < 128; i++) {
            inputA[i] = 838860800;
        }

        pq_config_t pqCfg;
        PQ_GetDefaultConfig(&pqCfg);
        pqCfg.inputAFormat = kPQ_32Bit;
        pqCfg.inputAPrescale = -23; // Q8.23, 23 fractional bits.
        pqCfg.inputBFormat = kPQ_32Bit;
        pqCfg.inputBPrescale = -31; // Q0.31, 31 fractional bits.
        PQ_SetConfig(POWERQUAD, &pqCfg);
        PQ_MatrixProduct(POWERQUAD, POWERQUAD_MAKE_MATRIX_LEN(16, 8, 8), (void
 *)inputA, (void *)window,
                         (void *)result);
        PQ_WaitDone(POWERQUAD);
    }
```

PowerQuad supports several matrix operations, including addition, subtraction, Hadamard product, standard matrix product, transpose, scaling, inversion, and vector dot product. It is important to note that users can encounter naming differences in the PowerQuad SDK: the standard matrix product is referred to as 'PQ_MatrixMultiplication', whereas the matrix Hadamard product is named as 'PQ_MatrixProduct'.

The output of the operation is illustrated in the Figure 1. To get a Q8.23 fixed-point number results, follow the previous example to set 'pqCfg.outputFormat = kPQ_32Bit' and 'pqCfg.outputPrescale = -23'.



result	float [128]	0x2005fda4	0x2005fda4
[0...99]	float [100]	0x2005fda4	0x2005fda4
result[0]	float	0	0x2005fda4
result[1]	float	0.0611782111	0x2005fda8
result[2]	float	0.244566858	0x2005fdac
result[3]	float	0.549715936	0x2005fdb0
result[4]	float	0.975877464	0x2005fdb4
result[5]	float	1.52200842	0x2005fdb8
result[6]	float	2.18677378	0x2005fdbc
result[7]	float	2.96854591	0x2005fdc0
result[8]	float	3.86541247	0x2005fdc4
result[9]	float	4.87517929	0x2005fdc8
result[10]	float	5.99537277	0x2005fdcc
result[11]	float	7.22325468	0x2005fdd0
result[12]	float	8.55582047	0x2005fdd4
result[13]	float	9.98980713	0x2005fdd8
result[14]	float	11.5217037	0x2005fddc
result[15]	float	13.1477642	0x2005fde0
result[16]	float	14.8640108	0x2005fde4
result[17]	float	16.6662369	0x2005fde8
result[18]	float	18.550045	0x2005fdec
result[19]	float	20.5108109	0x2005fdf0
result[20]	float	22.5437431	0x2005fdf4
result[21]	float	24.6438656	0x2005fdf8
result[22]	float	26.8060417	0x2005fdfc
result[23]	float	29.0249825	0x2005fe00
result[24]	float	31.295248	0x2005fe04
result[25]	float	33.61129	0x2005fe08
result[26]	float	35.9674377	0x2005fe0c
result[27]	float	38.3579216	0x2005fe10
result[28]	float	40.7768974	0x2005fe14

result[61]	float	99.6187668	0x2005fe98
result[62]	float	99.8631134	0x2005fe9c
result[63]	float	99.9854507	0x2005fea0
result[64]	float	99.985466	0x2005fea4
result[65]	float	99.8631516	0x2005fea8
result[66]	float	99.6188278	0x2005feac
result[67]	float	99.2530594	0x2005feb0
result[68]	float	98.766777	0x2005feb4
result[69]	float	98.1611481	0x2005feb8
result[70]	float	97.4376678	0x2005febc
result[71]	float	96.5980911	0x2005fec0
result[72]	float	95.6444931	0x2005fec4
result[73]	float	94.5791931	0x2005fec8
result[74]	float	93.4047928	0x2005fecc
result[75]	float	92.1241913	0x2005fed0
result[76]	float	90.7404861	0x2005fed4
result[77]	float	89.2570953	0x2005fed8
result[78]	float	87.6776199	0x2005fedc
result[79]	float	86.0059586	0x2005fee0
result[80]	float	84.2461853	0x2005fee4
result[81]	float	82.4026031	0x2005fee8
result[82]	float	80.4797134	0x2005feec
result[83]	float	78.482254	0x2005fef0
result[84]	float	76.4151001	0x2005fef4
result[85]	float	74.2832718	0x2005fef8
result[86]	float	72.0920563	0x2005fefc
result[87]	float	69.8467636	0x2005ff00
result[88]	float	67.5529175	0x2005ff04
result[89]	float	65.2160797	0x2005ff08
result[90]	float	62.8420372	0x2005ff0c
result[91]	float	60.4365692	0x2005ff10

Figure 1. Fixed-point multiplication results

The same operations are applicable to 16-bit fixed-point numbers as well.

```
// Use case:
    // Convert internal fixed-point ADC sample value to float.
    {
        int16_t input[128] = {0};
        float output[128] = {0};

        for (uint32_t i = 0; i < 128; i++) {
            input[i] = 26214; // 0.8 in Q0.15
        }
```

```
        pq_config_t pqCfg;
        PQ_GetDefaultConfig(&pqCfg);
        pqCfg.inputAFormat = kPQ_16Bit;
        pqCfg.inputAPrescale = -15; // 15 fractional bits.
        PQ_SetConfig(POWERQUAD, &pqCfg);
        PQ_MatrixScale(POWERQUAD, POWERQUAD_MAKE_MATRIX_LEN(16, 8, 8), 1.0f,
 (const void *)input, (void *)output);
        PQ_WaitDone(POWERQUAD);

        for (uint32_t i = 0; i < 128; i++) {
            assert(output[i] == 0.799987793f);
        }
    }
```

3 PowerQuad private memory

According to the *MCX Nx4x Reference Manual* (document MCXNX4XRM), PowerQuad has a built-in 128-bit wide RAM controller and 4×4 kB private RAM mapped to address, 0xE0000000. This address always contains private peripherals in an Arm Cortex-M system. So, the Cortex-M33 Core can only access its private peripheral like SCB, but not the private memory of PowerQuad.

To speed up the memory operation of PowerQuad, send one operand from system RAM and the other from private memory of PowerQuad. This allows PowerQuad to read two operands at the same time and improves bandwidth and performance.

To move data from system memory to PowerQuad private memory, users set the output address of a computation to the private memory address. They can also use the PowerQuad Matrix scale function to move data from system to PowerQuad private memory.

The following example shows how to use the Matrix Scale function to move data from system memory to PowerQuad private memory and perform calculations. A 16×16 matrix addition takes 566 cycles when one operand comes from system memory and the other from PowerQuad private memory. This method gives almost 40 % better performance than using system memory for both operands. The performance improves more when the algorithm calls more than one PowerQuad acceleration.

```
// PowerQuad private memory VS system memory.
    {
        float inputA[256] = {0};
        float inputB[256] = {0};
        float output[256] = {0};

        const uint32_t pqLen = POWERQUAD_MAKE_MATRIX_LEN(16, 16, 16);

        for (uint32_t i = 0; i < 256; i++) {
            inputA[i] = 1.0f * i;
            inputB[i] = 0.5f * i;
        }

        pq_config_t pqCfg;
        PQ_GetDefaultConfig(&pqCfg);
        PQ_SetConfig(POWERQUAD, &pqCfg);

        // Copy inputB to PQ's private memory.
        PQ_MatrixScale(POWERQUAD, pqLen, 1.0f, (void *)inputB, (void *)
(0xE0001000));
        PQ_WaitDone(POWERQUAD);

        // Matrix addition from system memory and PQ's private memory.
```

```
        // Takes 566 cycles under GCC O3 (MCUXpressoIDE v24.12)
        PQ_MatrixAddition(POWERQUAD, pqLen, (void *)inputA, (void *)
(0xE0001000), (void *)output);
        PQ_WaitDone(POWERQUAD);

        // Matrix addition from both system memory.
        // Takes 811 cycles under GCC O3 (MCUXpressoIDE v24.12)
        PQ_MatrixAddition(POWERQUAD, pqLen, (void *)inputA, (void *)inputB,
 (void *)output);
        PQ_WaitDone(POWERQUAD);

        __NOP();
    }
```

PowerQuad does not apply a scaling factor on its private memory during read and operation. It always keeps a floating-point number in its private memory, except PowerQuad FFT operation.

Users can dump PowerQuad private memory by a Matrix scaling function for debug purpose.

```
float tmpArea[1024 * 4] = {0};
void dump_powerquad_private_memory() {
    pq_config_t pqCfg;
    PQ_GetDefaultConfig(&pqCfg);
    PQ_SetConfig(POWERQUAD, &pqCfg);

    const uint32_t pqLen = POWERQUAD_MAKE_MATRIX_LEN(16, 16, 16);
    for (uint32_t i = 0; i < 4 * 1024; i += 256) {
        PQ_MatrixScale(POWERQUAD, pqLen, 1.0f, (const void *)(0xE0000000 + 4 *
 i), (void *)(&tmpArea[i]));
        PQ_WaitDone(POWERQUAD);
    }
}
```

This optimization is not noticeable when the operation times are short. So, moving one operand from AHB system memory to PowerQuad private memory before performing the calculation can be slower than computing directly with two operands present in AHB system memory. Moreover, users usually don’t need to move the data to PowerQuad’ private memory when using functions like 'PQ_MatrixScale'. It is more efficient to store the result in private memory during the first calculation.

The following is a simple PID controller example code. This simple PID controller enhances the performance of PowerQuad private memory. Using MCUXpressoIDE v24.12 with maximum optimization, the PowerQuad-based PID algorithm requires 3,290 CPU cycles per update. In comparison, the CM33 core-based algorithm takes 5,572 cycles, resulting in a 40 % performance improvement.

```
#include <cstdint>

#include "fsl_powerquad.h"

template <const std::uint8_t INSTANCE, const std::uint8_t INPUT_SIZE, const
 std::uint8_t OUTPUT_SIZE> class PQID {
    static_assert(INPUT_SIZE <= 16, "Input size must be less than 16");
    static_assert(OUTPUT_SIZE <= 16, "Output size must be less than 16");
    static_assert(INSTANCE < 2, "Instance must be less than 2");

    static constexpr std::uintptr_t KP = (0xE0001000 + 0x1000 * INSTANCE);
    static constexpr std::uintptr_t KI = (0xE0001000 + 0x1000 * INSTANCE +
 0x400);
    static constexpr std::uintptr_t KD = (0xE0001000 + 0x1000 * INSTANCE +
 0x800);
```

```
    static constexpr std::uintptr_t TMP0 = (0xE0000000);
    static constexpr std::uintptr_t TMP1 = (0xE0000000 + 4 * 16);
    static constexpr std::uintptr_t TMP2 = (0xE0000000 + 4 * 16 * 2);

  public:
    explicit PQID(float *Kp, float *Ki, float *Kd) : integral{0.0f},
prevError{0.0f} {
        PQ_Init(POWERQUAD);

        uint32_t len = POWERQUAD_MAKE_MATRIX_LEN(OUTPUT_SIZE, INPUT_SIZE, 0);

        void *KP = (void *)(this->KP);
        void *KI = (void *)(this->KI);
        void *KD = (void *)(this->KD);

        PQ_MatrixScale(POWERQUAD, len, 1.0f, (void *)Kp, KP);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, len, 1.0f, (void *)Ki, KI);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, len, 1.0f, (void *)Kd, KD);
        PQ_WaitDone(POWERQUAD);
    }

    void Update(float *setpoint, float *input, float *output, float dtime) {
        uint32_t len1 = POWERQUAD_MAKE_MATRIX_LEN(INPUT_SIZE, 1, 1);
        uint32_t len2 = POWERQUAD_MAKE_MATRIX_LEN(OUTPUT_SIZE, INPUT_SIZE, 1);

        void *KP = reinterpret_cast<void *>(this->KP);
        void *KI = reinterpret_cast<void *>(this->KI);
        void *KD = reinterpret_cast<void *>(this->KD);
        void *TMP0 = reinterpret_cast<void *>(this->TMP0);
        void *TMP1 = reinterpret_cast<void *>(this->TMP1);
        void *TMP2 = reinterpret_cast<void *>(this->TMP2);

        float error[INPUT_SIZE];
        float derivative[INPUT_SIZE];

        PQ_MatrixSubtraction(POWERQUAD, len1, (void *)setpoint, (void *)input,
error);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixAddition(POWERQUAD, len2, KP, (void *)error, TMP0);
        PQ_WaitDone(POWERQUAD);

        PQ_MatrixSubtraction(POWERQUAD, len1, error, (void *)prevError,
derivative);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, len1, 1.0f / dtime, derivative, derivative);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixMultiplication(POWERQUAD, len2, KD, (void *)derivative, TMP1);
        for (int i = 0; i < INPUT_SIZE; i++) {
            this->prevError[i] = error[i];
        }
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixScale(POWERQUAD, len1, dtime, (void *)error, TMP2);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixAddition(POWERQUAD, len1, TMP2, (void *)integral, (void
*)integral);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixMultiplication(POWERQUAD, len2, KI, (void *)integral, TMP2);
        PQ_WaitDone(POWERQUAD);
```

```
        PQ_MatrixAddition(POWERQUAD, len2, TMP0, TMP1, (void *)output);
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixAddition(POWERQUAD, len2, TMP0, TMP2, (void *)output);
        PQ_WaitDone(POWERQUAD);
    }

  private:
    float integral[INPUT_SIZE];  // Integral value
    float prevError[INPUT_SIZE]; // Previous error value
};
```

4 Pipeline operation

PowerQuad is a coprocessor which can run in parallel with the main CPU. Users can notice the function call, 'PQ_WaitDone(POWERQUAD)'; – this is a simple wait for event which is inefficient. Users can prepare the next PowerQuad calculation while waiting for the previous one to finish. For example, PowerQuad support up to 16×16 matrix product operation, which can be treated as up to 256 vector product operation.

```
// Calculate next operand during the previous PowerQuad calculation.
    {
        float input[1024] = {0};
        float window[1024] = {0};
        float result[1024] = {0};

        pq_config_t pqCfg;
        PQ_GetDefaultConfig(&pqCfg);
        pqCfg.inputAFormat = kPQ_Float;
        pqCfg.inputBFormat = kPQ_Float;
        pqCfg.outputFormat = kPQ_Float;
        PQ_SetConfig(POWERQUAD, &pqCfg);
        for (uint32_t i = 0; i < 1024; i++) {
         input[i] = 100.0f;
        }
        const uint32_t pqLen = POWERQUAD_MAKE_MATRIX_LEN(16, 16, 16);

        // Initialize HANN window Part0
        for (uint32_t i = 0; i < 256; i++) {
            float w = 0.5 * (1 - cosf(2 * M_PI * i / (1024 - 1)));
            window[i] = w;
        }
        PQ_MatrixProduct(POWERQUAD, pqLen, (void *)(&input[0]), (void *)
(&window[0]), (void *)(&result[0]));
        // Calculate next operand during the previous PowerQuad calculation.
        // Initialize HANN window Part1 During Matrix Calculation Part0
        for (uint32_t i = 256; i < 512; i++) {
            float w = 0.5 * (1 - cosf(2 * M_PI * i / (1024 - 1)));
            window[i] = w;
        }
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixProduct(POWERQUAD, pqLen, (void *)(&input[256]), (void *)
(&window[256]), (void *)(&result[256]));
        // Calculate next operand during the previous PowerQuad calculation.
        // Initialize HANN window Part2 During Matrix Calculation Part1
        for (uint32_t i = 512; i < 768; i++) {
            float w = 0.5 * (1 - cosf(2 * M_PI * i / (1024 - 1)));
            window[i] = w;
        }
```

```
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixProduct(POWERQUAD, pqLen, (void *)(&input[512]), (void *)
(&window[512]), (void *)(&result[512]));
        // Calculate next operand during the previous PowerQuad calculation.
        // Initialize HANN window Part3 During Matrix Calculation Part2
        for (uint32_t i = 768; i < 1024; i++) {
            float w = 0.5 * (1 - cosf(2 * M_PI * i / (1024 - 1)));
            window[i] = w;
        }
        PQ_WaitDone(POWERQUAD);
        PQ_MatrixProduct(POWERQUAD, pqLen, (void *)(&input[768]), (void *)
(&window[768]), (void *)(&result[768]));
        PQ_WaitDone(POWERQUAD);
    }
```

```
// Use PowerQuad's two MACs.
    {
        float input[128] = {0};
        float output[128] = {0};

        for (uint32_t i = 0; i < 128; i++) {
            input[i] = 1.0f * i;
        }

        for (uint32_t i = 0; i < 128; i += 2) {
            pq_float_t val0, val1;

            val0.floatX = input[i + 0];
            _pq_sqrt0(val0.integerX);
            val1.floatX = input[i + 1];
            _pq_sqrt1(val1.integerX);
            val0.integerX = _pq_readMult0();
            val1.integerX = _pq_readMult1();

            output[i + 0] = val0.floatX;
            output[i + 1] = val1.floatX;
        }
    }
```

5 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

6 Revision history

Table 1 summarizes the revisions to this document.

Table 1. Revision history

Document ID	Release date	Description
AN14712 v.1.0	30 June 2025	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents