STRX RFE coding standard

Rev. 0.8 — 10 November 2021 User manual

Document information

Information	Content
Keywords	STRX, RFE, coding standard
Abstract	This document describes the STRX RFE coding standard.

1 Introduction

This document defines the coding standard for the STRX RFE software. It incorporates the guidelines from the MISRA C 2012 coding standard including amendment 1.

Language features and constructs that are not explicitly described in this coding standard are not allowed to be used in the STRX RFE software.

All examples in this document are normative and describe the precise placement of spaces and parenthesis.

A separate API documentation template exists to specify the required Doxygen tags and structure.

2 Naming convention

This section describes the naming convention.

Variables defined and used inside a single function are explicetly not covered by this naming convention and it is allowed to use shorter names (e.g., counter or c). The same exception applies to the names of function parameters.

2.1 General naming convention

The general naming convention is rfe<layer><module>_<item-name> where <layer> specifies the layer of the this code, module is the module identifier, and <item-name> is the name of the item within the module (e.g., an enumerated type or a function). The <layer> part can be omitted in cases where the module applies to the user visible top level.

Camel case notation is used for the rfe<layer><module> and <item-name> sections. Multi-letter acronyms are written with a single upper case character followed by lower case characters (e.g., Can or Ahb).

In certain cases, an additional suffix is appended. Type aliases, defined by typedef, are appended with the _t suffix. Enumeration constants are appended with the _e suffix.

Variables that represent pointers shall start with the p prefix (e.g., pValue). The * character in pointers shall associate to the type and not to the identifier (e.g., uint32_t* pValue instead of uint32_t *pValue).

Examples:

```
rfeHwCan_mode_t         // a type alias in the hardware driver
                        // for the CAN peripheral
rfeHwCan_init           // a function identifier in the hardware driver
                        // for the CAN peripheral
rfeError_error_t        // a type alias in the general error handling module
rfeError_error_none_e   // an enumeration constant
rfeApi_startRadarCycle  // a top-level function identifier
```

2.2 Preprocessor macro naming convention

Preprocessor macros use the following naming convention RFE_<layer>_<module>_<item-name> where <layer>, <module>, and <item-name> are similar to the general naming convention described in Section 2.1. All items use the ALL_CAPS notation.

Examples:

```
RFE_HW_CAN_INTERRUPT_OVERFLOW   // a constant defined in the hardware driver
                                // for the CAN peripheral
RFE_ERROR_CREATE                // a preprocessor macro in the general error
                                // handling module
```

2.3 Hardware register definitions

Dedicated naming conventions are described in this section for defining hardware registers. These naming conventions are compliant with the preprocessor naming convention described in Section 2.2 .

Preprocessor macros used for peripheral base addresses use RFE_HW_<module>_BASE where <module> is the module identifier.

Preprocessor macros used for register offsets use `RFE_HW_<module>_<register>_REG` where `<module>` is the module identifier and `<register>` is the register identifier.

Preprocessor macros used for register content definitions use `RFE_HW_<module>_<type>_<register>_<item-name>` where `<module>` is the module identifier, `<register>` is the register identifier, `<item-name>` is the name of the item in the specified register, and `<type>` denotes the type. The following `<type>` literals are defined:

- `BIT` for individual bits
- `MSK` for masks
- `SHF` for shifts
- `VAL` for constant values

Examples:

```
// base address
#define RFE_HW_GPIO_BASE                ( ( rfeHwAbh_base_t ) ( 0x80040000ul ) )

// register offsets
#define RFE_HW_GPIO_CONTROL_REG         ( ( rfeHwAhb_offset_t ) ( 0x0000ul ) )
#define RFE_HW_GPIO_DIRECTION_REG       ( ( rfeHwAhb_offset_t ) ( 0x0004ul ) )

// register definition
#define RFE_HW_GPIO_CONTROL_ENABLE_BIT  ( ( uint8_t )  ( 0ul ) )
#define RFE_HW_GPIO_CONTROL_MODE_MSK    ( ( uint32_t ) ( 1ul<<3 | 1ul<<2 ) )
#define RFE_HW_GPIO_CONTROL_MODE_SHF    ( ( uint8_t )  ( 2ul ) )
#define RFE_HW_GPIO_CONTROL_DEFAULT_VAL ( ( uint32_t ) ( 0x00000008ul ) )
```

3 Files and folders

This section describe the rules related to files and folders.

3.1 Header files

The general naming convention for header files is `rfe<layer><module>.h` where `<layer>` specifies the layer of the this code and `module` specifies the module identifier. The `<layer>` part can be omitted in cases where the module applies to the user visible top level.

The contents of a header file shall be completely encapsulate in an include guard. This include guard shall use an identifier that corresponds to the file name converted to the ALL_CAPS notation. The formatting shall comply to the following example for file "rfeHwReg.h":

```
#ifndef RFE_HW_REG_H
#define RFE_HW_REG_H

// header file contents

#endif // !RFE_HW_REG_H
```

The last line of the header file shall be an empty line.

An #include directive shall not contain absolute paths.

An #include directive referring to standard library header files shall use angle brackets (e.g., `#include <stdint.h>`).

An #include directive not referring to standard library header files shall use double quotes (e.g., `#include "rfeHwReg.h"`).

The header file shall not contain executable code except for inline functions. Inline functions are preferred to function like macros due to the added type checking.

An example of an inline function is:

```
inline uint32_t rfeHwExample_max(
    uint32_t        a,
    uint32_t        b
)
{
    // implementation
}
```

3.1.1 C++ compatibility

If a header file is going to be used in an C++ environment, it is required to wrap the declarations in a `extern "C"` block as shown in the followin code example:

```
#ifdef __cplusplus
extern "C" {
#endif

// declarations

#ifdef __cplusplus
}
#endif
```

3.2 Source code files

The general naming convention for source code files is rfe<layer><module>.c where <layer> specifies the layer of the this code and module specifies the module identifier. The <layer> part can be omitted in cases where the module applies to the user visible top level.

Source code files shall be structured such that these contain #include directives, type definitions, variables, and functions in this order.

Internal function prototypes shall be placed in the optional internal header file when needed (see Section 3.4).

The last line of the source code file shall be an empty line.

3.3 Formatting

Tabs shall not be used. One or more sequences of four spaces shall be used for indentations.

Comments shall be added as either block (/* .. */) or line (// ..) comments. Block comments shall not be nested.

Code shall not be commented out. This include block comments (/* .. */), line (// ..) comments, and preprocessor constructs (e.g., #if 0 .. #endif).

Lines shall not exceed 120 columns.

Only the Unix style line ending (line feed, LF, \n, or 0x0A) shall be used.

The last line of each file shall be an empty line.

Digraphs (<:, :>, <%, %>, %:, and %:%:) shall not be used. Digraph literals can be used in string literals, but are discouraged.

Trigraphs (??(, ??), ??<, ??>, ??=, ??/, ??!, ??', and ??-) shall not be used. Trigraph literals are not allowed in string literals.

Refer to the sections on types (Section 4), expressions (Section 5), and statements (Section 6) for formatting rules specific to these items. Spacing and alignment in the examples in these sections are normative.

The API shall be documented using Doxygen as defined by the header file templates available in the STRX RFE SW repository.

3.4 Folder structure

The source code files for a module will be stored in a folder that corresponds to the name of the module. The main public header file uses the name of the module, with ".h" extenstion, and is placed in the "inc" subfolder. The main public source code file uses the name of the module, with ".c" extension, and is placed in the "src" subfolder.

An optional internal header file uses the name of the module, with "_internal.h" suffix, and is placed in the "src" subfolder. This internal header file can declarations and definitions that are only visible to module itself (e.g., constant definitions, static inline functions, function prototypes, type definitions, et cetera).

The hardware driver for an SPI controller would therefore have the following folder structure (where "rfeHwSpi_internal.h" is optional):

```
rfeHwSpi
    inc
        rfeHwSpi.h
    src
        rfeHwSpi_internal.h
        rfeHwSpi.c
```

4 Types

This section describes the rules concerning types.

4.1 Boolean types

The `bool` type from `<stdbool.h>` shall be used for boolean types.

Only the literals `true` and `false` from `<stdbool.h>` shall be used.

4.2 Integer types

The following integer types from `<stdint.h>` are allowed: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int8_t`, `int16_t`, `int32_t`, and `int64_t`.

The unsigned versions should be used, unless negative numbers are explicitly required.

The standard integer types (`char`, `short`, `int`, and `long`) shall not be used as these are inherently ambigiuous (size and signedness) making interoperability more difficult.

4.3 Floating-point types

The `float` and `double` types can be used for floating-point types.

Using floating-point types is discouraged and therefore their usage should be minimized. Otherwise, the type that matches the hardware accelerator is preferred.

Take care when using floating-point types in operations. For instance, do not use floating-point types for looping variables, consider the possibility of `NaN` inputs and results, consider the epsilon when comparing two floating-point values, et cetera.

4.4 Constant values

Constant values shall be specified as preprocessor macro definitions. These macro definitions will use the ALL_CAPS notation for the name of the macro. The replacement text of the macro shall contain an explicit cast to the desired type.

```
#define RFE_HW_EXAMPLE_MAGIC_NUMBER     ( (uint32_t) 42ul )
#define RFE_HW_EXAMPLE_MAGIC_BOOLEAN    ( (bool) true )
```

This notation shall also be used for bit fields.

```
typedef uint32_t                rfeHwSpi_int_t;
#define RFE_HW_SPI_INT_OVER     ( (rfeHwSpi_int_t) ( 1ul<<2 ) )
#define RFE_HW_SPI_INT_UNDER    ( (rfeHwSpi_int_t) ( 1ul<<1 ) )
#define RFE_HW_SPI_INT_DONE     ( (rfeHwSpi_int_t) ( 1ul<<0 ) )
```

4.5 Enumerated types

The enumerated type shall only be used for enumerations — a complete, ordered listing of all the items in a collection. Other constant values (e.g., hardware base addresses and bit fields) shall be implemented using constant values (see Section 4.4).

Enumerated types shall be always be declared within a typedef declaration and the tag name shall be omitted.

The identifier for each enumeration item shall have a prefix that matches the name of the enumerated type and shall use "_e" as suffix (e.g., "hwSpi_mode_test_e" for enumerated type "hwSpi_mode_t").

An enumerated type shall only list the functional items and shall not include items that are purely of interest to the implementation. This forbids the inclusion of "last" or "max" items that are used for range checking or looping and such implementation details shall be implemented using constant values (see Section 4.4).

Explicit constant values shall only be included when there is a need for these values (e.g., hardware register) and can be specified for either only the first enumeration item or for all enumeration items. All three possibilities are shown in the example below.

```
typedef enum
{
    rfeHwSpi_mode_idleLow_first_e,
    rfeHwSpi_mode_idleHigh_first_e,
    rfeHwSpi_mode_idleLow_second_e,
    rfeHwSpi_mode_idleHigh_second_e
} rfeHwSpi_mode_t;

typedef enum
{
    rfeHwSpi_mode_idleLow_first_e = 0ul,
    rfeHwSpi_mode_idleHigh_first_e,
    rfeHwSpi_mode_idleLow_second_e,
    rfeHwSpi_mode_idleHigh_second_e
} rfeHwSpi_mode_t;

typedef enum
{
    rfeHwSpi_mode_idleLow_first_e   = 0ul,
    rfeHwSpi_mode_idleHigh_first_e  = 1ul,
    rfeHwSpi_mode_idleLow_second_e  = 2ul,
    rfeHwSpi_mode_idleHigh_second_e = 3ul
} rfeHwSpi_mode_t;
```

4.6 Structure types

Structure types shall be always be declared within a typedef declaration and the tag name shall be omitted.

```
typedef struct {
    uint32_t            clockDivider;
    rfeHwSpi_mode_t     enumMember;
} rfeHwSpi_config_t;
```

The C programming language specification only specifies that the members of a structure are stored in memory in the order of their declaration and that the address of the first member is identical to the address of the structure object. Therefore, the placement of the other members in memory is compiler specific and care should be taken when a specific memory representation is required, especially when multiple compilers or processors are involved.

The representation of bit fields in memory is not specified by the C programming language specification and therefore bit fields shall not be used in situations where their memory representation matters (e.g., hardware registers).

Using flexible structure members — the final member is an array with flexible length — is discouraged and therefore their use should be minimized.

4.7 Pointer types

The C programming language defines the following four pointer qualifiers:

```
      uint32_t* pA         //          pointer to an         uint32_t
const uint32_t* pB         //          pointer to a constant uint32_t
      uint32_t* pC const   // constant pointer to an         uint32_t
const uint32_t* pD const   // constant pointer to a constant uint32_t
```

Pointer `pA` is the least restrictive pointer while pointer `pD` is the most restrictive pointer.

The most restrictive pointer qualifier — suitable for the situation — should be used.

Declarations should contain no more than two levels of pointer nesting.

5 Expressions

This section describes the rules concerning expressions.

5.1 Unused code

Unreachable code shall not be present. Unreachable code is code that cannot be executed as it cannot be called from a program. See MISRA C:2012 mandatory Rule 2.1.

Dead code shall not be present. Dead code is code that does not affect the behavior of a program when removed. See MISRA C:2012 advisory Rule 2.2.

Unused type declaration shall not be present. See MISRA C:2012 advisory Rule 2.3.

Unused preprocessor macro declaration shall not be present. See MISRA C:2012 advisory Rule 2.5.

Unused function parameters shall not be present. See MISRA C:2012 advisory Rule 2.7.

5.2 Variable declarations

Only a single variable shall be declared on a single line.

It is preferred to immediately define a (default) value for a variable when this is declared.

It is preferred to declare a variable at the moment it is used (e.g., a function returned).

Variables that represent pointers shall start with the `p` prefix (e.g., `pValue`). The `*` character in pointers shall associate to the type and not to the identifier (e.g., `uint32_t* pValue` instead of `uint32_t *pValue`).

Variables defined and used inside a single function are explicetly not covered by this naming convention and it is allowed to use shorter names (e.g., `counter` or `c`).

5.3 Operators

Operators shall be preceded and followed by spaces for readability.

The following code block depicts examples or expressions with operators:

```
uint32_t a = 1ul + 3ul;
42 <= 42 ? 1 : 0;
```

5.4 Casts

Only explicit casts shall be used. Implicit casts shall not be used.

Casts that narrow values shall have an explicit masking operation.

The use of casts shall be minimized as they are usually indications of problems with the type system. A comment must accompany each cast to explain its existence.

The entire `cast` expression must be enclosed in parenthesis in those cases where this expression is part of a larger expression and when used as the body of a `#define` preprocessor macro.

The following code block depicts the two templates for `cast` expressions:

```
( ( newType ) value )    // for use in complicated expressions
( newType ) value        // for use in simple statements
```

The following code block depicts an example where a cast is used to convert an `uint8_t` value to an `uint32_t` value:

```
uint8_t  value = 0ul;
uint32_t newValue = ( uint32_t ) value ;     // reason ...
```

The following code block depicts an example where a narrowing cast is used to convert an `uint32_t` value to an `uint8_t` value:

```
uint32_t value = 0ul;
uint8_t  newValue = ( uint8_t ) ( value & 0x000000fful );       // reason ...
```

The following code block depicts an example where a narrowing cast is used to convert an `uint32_t` value to an `uint8_t` value as part of a larger expression and therefore is enclosed in parenthesis:

```
uint32_t value = 0ul;
uint8_t  newValue = ( ( uint8_t ) ( value & 0x000000fful ) ) + 0x00001234ul;    //
 reason ...
```

The following code block depicts an example where a cast is used in a `#define` preprocessor macro:

```
#define RFE_HW_GPIO_BASE                ( ( rfeHwAbh_base_t ) ( 0x80040000ul ) )
```

5.5 Conditional expressions

Conditional expressions shall be kept as simple as possible.

Parenthesis should be used to improve readability by grouping together parts of the expression while not relying too much on the operator precedence rules.

Assignments are not allowed in conditional expressions. When dealing with comparisons to constants, it is advisable to use the form `( constant == variable )` as `( constant = variable )` will result in an error emitted by the compiler.

Use of negative logic should be limited as it is hard to understand by human readers.

Examples of valid and invalid conditional expressions are:

```
// assuming that these variables exist
bool         valid;
uint32_t     number;

// valid conditional expressions:
if ( valid ) ...
if ( true == valid ) ...
if ( 0 != number ) ...
if ( ( number >= 1 ) && ( number < 100 ) ) ...

// invalid conditional expressions:
if(true==valid) ...                     // incorrect spacing
if ( number ) ...                       // implicit cast to boolean
if ( number >= 1 && number < 100 ) ...  // relies on operator precedence
```

6 Statements

This section describes the rules concerning statements.

6.1 Selection statements

Selection statements are used to run specific code blocks based on certain conditions.

An `if` statement should be used when a binary decision needs to be made. A `switch` statement should be used when the decision is not binary. The conditional operator (`? :`) can be used in situations where a specific value is required, based on a binary decision.

See Section 5.5 for more information on the condition expression.

6.1.1 If statement

The `if` and `else` branches will both be present and consist of block statements. Each block statement should contain executable code or a comment that explains why there is no executable code in this block statement.

The following code block depicts the template for an `if` statement:

```
if ( condition )
{
    // code ...
}
else
{
    // code or comment ...
}
```

6.1.2 Switch statement

Each `switch` clause shall be terminated by an unconditional `break` statement. Fall throughs are only allowed when the clause is empty (see the `case 2:` label in the code block below). The `default` label is always required and must be the last label in the switch statement.

The following code block depicts the template for a `switch` statement:

```
switch ( condition )
{
    case 1:
        // code ...
        break;

    case 2:
    case 3:
        // code ...
        break;

    case 4:
    default:
        //code ...
        break;
}
```

It is not required to explicitly list all possible enumeration items when an enumerated type is used as the condition for a `switch` statement as the final `default` clause will apply to all remaining enumeration item

6.1.3 Conditional operator

The two expressions of the conditional operator should have the same type and this should match with the type expected by the code calling the conditional operator.

The following code block depicts the template for a conditional operator statement.

```
condition ? trueExpression : falseExpression;
```

6.2 Loops

Loops can be used to run a code block zero, one, or multiple times.

The `for` loop should be used if the required number of iterations is known beforehand. The endless loop is special variant of the `for` loop. The `while` loop can be used if the number of iterations is not known beforehand. The `do ... while` loop is a special variant of the `while` loop that can be used if the loop has to loop at least once.

Unbounded loops should be avoided as it can lead to undesired application behavior (e.g., missed deadlines).

6.2.1 For statement

The `expression1` shall only be concerned with declaring and initializing loop variable(s). The `expression2` shall only be concerned with checking the loop variable(s). The `expression3` shall only be concerned with updating the loop variable(s).

Preferably, looping variables should be declared in `expression1` such that these variables cannot be accessed outside of the `for` loop.

The following code block depicts the template for a `for` loop:

```
for ( expression1; expression2; expression3 )
{
    // code ...
}
```

Example:

```
for ( uint8_t c; c < 100ul; c++ )
{
    // code ...
}
```

6.2.2 Endless loop

The following code block depicts the template for an endless loop:

```
for ( ;; )
{
    // code ...
}
```

The `for ( ;; )` variant is preferred over `while` variants as this does not rely on a condition and implicit conversions to a boolean expression.

6.2.3 While statement

The loop variable(s) shall be declared and initialized before the `while` loop is called.

The condition expression shall only be concerned with checking the loop variable(s).

Updating the loop variable(s) shall be the last statement(s) of the while block.

The following code block depicts the template for a while loop:

```
while ( condition )
{
    // code ...
    // update loop variable
}
```

Example:

```
uint32_t value = 0ul;

while ( value < 10ul )
{
    value += 2ul;
}
```

6.2.4 Do … while statement

The loop variable(s) shall be declared and initialized before the do ... while loop is called.

The condition expression shall only be concerned with checking the loop variable(s).

Updating the loop variable(s) shall be the last statement(s) of the do ... while block.

The following code block depicts the template for a do ... while loop:

```
do
{
    // code ...
    // update loop variable
} while ( condition )
```

Example:

```
uint32_t value = 0ul;

do
{
    value += 2ul;
} while ( value < 10ul )
```

6.3 Functions

A function shall always have a proper function prototype defined in a header file.

Recursive functions shall not be used.

Variable arguments shall not be used.

A function that returns a value shall declare and intialize a variable of this type at the beginning of the function block and the last statement shall return this value. No other exit points are allowed. The caller of such a function must use this returned value or explicitly cast this to void.

A function that takes no parameters shall explicetly indicate this by using the void keyword in the parameter list. A function that takes multiple parameters shall list these on new lines and shall align the parameter names on a proper column.

The following code block depicts the template for a function that takes no parameters:

```
void rfeHwExample_reset( void )
{
    // code ...
}
```

The following code block depicts the template for a function that takes one or more parameters:

```
uint32_t rfeHwExample_max(
    uint32_t        a,
    uint32_t        b
)
{
    uint32_t result = 0ul;

    // code ...

    return result;
}
```

The function shall not modify its parameters. The code in the body of function `rfeHwExample_max` could assign new values to parameters `a` and `b` that will be visible only to the remainder of this body. This style might give the false impression that these new values remain visible to the caller after this function exits. See rule MISRA C:2012 rule 17.8.

Functions that should return multiple values can return a structure type or use pointers. Returning a structure type is prefered due to the better readability of this solution.

6.4 Unions

Unions are discouraged and shall not be used.

It can never be assured which specific type the value has when an union is used for polymorphic types. Using an union to access a single value in several ways (e.g., `uint32_t` and `uint8_t[4]`) is compiler dependent and therefore should also not be relied upon.

7 Error handling

This section describes the generalized error handling mechanism.

The main concept behind this error handling system is that functions will only perform their code parts if no prior error exists and will skip these code parts if a prior error exists.

A single enumerated type named `rfeError_error_t` is defined in file `rfeError.h` that lists all possible error conditions in the system. The first item in this list must be `rfeError_error_none_e`.

This file also defines several preprocessor macros that implement the error handling mechanism. These preprocessor macros should be treated as opaque items.

`RFE_ERROR_CREATE`	Creates the error variable. Should always be the first error handling macro used in a error handling context.
`RFE_ERROR_CLEAR`	Clears the current error value by setting this to `rfeError_error_none_e`.
`RFE_ERROR_SET( err )`	Sets the current error varlue to the specified error value.
`RFE_ERROR_GET`	Returns the current error value.
`RFE_ERROR_IS_NO_ERROR`	Returns `true` if the current error value does not contain an error. Otherwise, it returns `false`.
`RFE_ERROR_IS_ERROR`	Returns `true` if the current error value contains an error. Otherwise, it returns `false`.
`RFE_ERROR_GUARD( block )`	The code in the `block` argument is only executed if the current error value does not contain an error.
`RFE_ERROR_FUNCTION_PARAMETER`	Use as the last parameter in function definitions to pass through the error information.
`RFE_ERROR_FUNCTION_ARGUMENT`	Use as the last argument when calling error handling aware functions.

7.1 Defining error handling aware functions

Each error handling aware function requires an additional parameter as defined by the `RFE_ERROR_FUNCTION_PARAMETER` preprocessor macro. This parameter should be the last parameter in order to ensure a common look and feel. Code parts inside an error handling aware function that should only be executed if the current error value does not contain an error should be encapsulated in the `RFE_ERROR_GUARD( block )` macro. The `RFE_ERROR_SET( err )` macro should be used when a new error condition has been detected.

The following code block shows an example of an error handling aware function that incorporates all these macros:

```
uint32_t rfeHwExample_squareValue(
    uint32_t        value,
    RFE_ERROR_FUNCTION_PARAMETER
)
{
    // result value to return, initialized with default value.
    uint32_t result = 0;

    // only execute the code block if no prior error exists
    RFE_ERROR_GUARD(
        if ( value <= 0x0000FFFFUL )
        {
            result = value * value;
```

```
        }
        else
        {
            result = 0;
            RFE_ERROR_SET( rfeError_error_range_e );
        }
    );

    return result;
}
```

Example function `rfeHwExample_squareValue` takes one `uint32_t` value and returns a `uint32_t` value. The preprocessor macro `RFE_ERROR_FUNCTION_PARAMETER` is added to the end of the parameter list to make this function aware of the current error variable. The code within the `RFE_ERROR_GUARD( block )` will only execute if the current error value does not contain an error when this function is called. The code inside this guarded block will check the input value and sets variable `result` to the squared value of the input if the input value is valid. Otherwise, it calls the `RFE_ERROR_SET( err )` function like preprocessor macro with `rfeError_error_range_e` as an argument (it is assumed that `rfeError_error_range_e` is part of the `rfeError_error_t` enumerated type).

The following code block shows the same example, but in this case it returns a structure type consisting of the input value and the squared value:

```
typedef struct
{
    uint32_t    value;
    uint32_t    squared;
} rfeHwExample_squaredValue_t;


rfeHwExample_squaredValue_t rfeHwExample_squareValue2(
    uint32_t        value,
    RFE_ERROR_FUNCTION_PARAMETER
)
{
    // result value to return, initialized with default values.
    rfeHwExample_squaredValue_t result = { .value = 0, .squared = 0 };

    // only execute the code block if no prior error exists
    RFE_ERROR_GUARD(
        if ( value <= 0x0000FFFFUL )
        {
            result.value   = value;
            result.squared = value * value;
        }
        else
        {
            result.value   = 0;
            result.squared = 0;
            RFE_ERROR_SET( rfeError_error_1_e );
        }
    );

    return result;
}
```

7.2 Using error handling aware functions

The following code block depicts an example main function that uses the example error handling aware functions defined in the previous section.

```
void main( void )
{
    RFE_ERROR_CREATE;

    uint32_t valueA = rfeHwExample_squareValue(  2ul, RFE_ERROR_FUNCTION_ARGUMENT );
    uint32_t valueB = rfeHwExample_squareValue( 40ul, RFE_ERROR_FUNCTION_ARGUMENT );
    uint32_t valueC = rfeHwExample_squareValue( 42ul, RFE_ERROR_FUNCTION_ARGUMENT );

    uint32_t valueD;
    if ( RFE_ERROR_IS_NO_ERROR )
    {
```

```
        valueD = valueA + valueB + valueC;
    }
    else
    {
        rfeError_t error = RFE_ERROR_GET;
        valueD = 0ul;
    }

    RFE_ERROR_CLEAR;

    rfeHwExample_squaredValue_t valueE
            = rfeHwExample_squareValue2(     2ul, RFE_ERROR_FUNCTION_ARGUMENT );
    rfeHwExample_squaredValue_t valueF
            = rfeHwExample_squareValue2( 75000ul, RFE_ERROR_FUNCTION_ARGUMENT );
    rfeHwExample_squaredValue_t valueG
            = rfeHwExample_squareValue2(    42ul, RFE_ERROR_FUNCTION_ARGUMENT );

    uint32_t valueH;
    if ( RFE_ERROR_IS_NO_ERROR )
    {
        valueH = valueE.squared + valueF.squared + valueG.squared;
    }
    else
    {
        rfeError_t error = RFE_ERROR_GET;
        valueH = 0ul;
    }
}
```

The `RFE_ERROR_CREATE` preprocessor macro will create the error variable required before calling any other error handling macro or error handling aware function.

The three calls of the `rfeHwExample_squareValue` function will all succeed and therefore the values for `valueA`, `valueB`, and `valueC` will be `4`, `1600`, and `1764` respectively.

The first call to the `RFE_ERROR_IS_NO_ERROR` macro will return `true` and therefore `valueD` will be set to `3368`.

Macro `RFE_ERROR_CLEAR` will clear the current error value to `hwError_error_none_e`.

The first call of the `rfeHwExample_squareValue2` function will succeed and `.valueE.value` will be set to `2` and `valueE.squared` to `4`. The input value to the second call is out of range and this will cause the current error value to be set to `hwError_error_range_e`, `valueF.value` to `0`, and `valueE.squared` to `0`. The third call fails as a previous error condition has been encountered and this will set `valueG.value` to `0` and `valueG.squared` to `0`.

The second call to the `RFE_ERROR_IS_NO_ERROR` macro will return `false`. Therefore, `error` will be set to `rfeError_error_range_e` and `valueH` to `0`.

Contents

1 Introduction 2
2 Naming convention 3
2.1 General naming convention 3
2.2 Preprocessor macro naming convention 3
2.3 Hardware register definitions 3
3 Files and folders 5
3.1 Header files 5
3.1.1 C++ compatibility 5
3.2 Source code files 6
3.3 Formatting 6
3.4 Folder structure 6
4 Types 8
4.1 Boolean types 8
4.2 Integer types 8
4.3 Floating-point types 8
4.4 Constant values 8
4.5 Enumerated types 8
4.6 Structure types 9
4.7 Pointer types 10
5 Expressions 11
5.1 Unused code 11
5.2 Variable declarations 11
5.3 Operators 11
5.4 Casts 11
5.5 Conditional expressions 12
6 Statements 13
6.1 Selection statements 13
6.1.1 If statement 13
6.1.2 Switch statement 13
6.1.3 Conditional operator 14
6.2 Loops 14
6.2.1 For statement 14
6.2.2 Endless loop 14
6.2.3 While statement 14
6.2.4 Do … while statement 15
6.3 Functions 15
6.4 Unions 16
7 Error handling 17
7.1 Defining error handling aware functions 17
7.2 Using error handling aware functions 18

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 10 November 2021
Document identifier: STRX_RFE_CODING_STANDARD