Document Number XXX

Firmware Development Guidelines for External Vendors

(Meta, RL Accessories, FW)

Revision History

Rev / ECO	Date	Author	Description of Change
0.1	7/1/2023	M. Sompel	Initial Draft

Table of Contents

Purpose

The purpose of this document is to provide requirements for firmware development for external vendors. The RL Accessories team utilizes third party vendors in a joint development model (JDM) to produce our products. The JDM partners are sometimes required to write firmware which needs to meet Meta and industry standards.

Scope

This document shall act as a reference for Accessory JDM partners' firmware development. The firmware to be reviewed will be the actual updates performed by JDM. Libraries and SDKs from third party microcontrollers will not be covered in this document. All libraries shall be used as-is with no changes.

Reference Documents

MISRA C:2012

Development Tools

All of the tools required for the development, compilation, debug, and programming of firmware will be provided by the vendor with detailed instructions on how to use them. Meta will review, compile, and compare binary image against that of the vendor.

These can be provided as links to the tools and hardware.

Special fixtures designed for programming and debugging will be provided unless otherwise agreed upon.

Firmware Releases

Firmware releases will be done on a weekly basis or an agreed upon schedule. They will be formalized. Any firmware going into a release must:

- All code will reside in the GitHub created by Meta
- Tagged with revision number
- Compile without errors
- go through a peer and Meta code review

The review will require any design docs such as block diagrams, flow charts, state machine diagrams to be provided.

Code will be reviewed for following:

- coding guidelines
- design implementation
- issues resolved

Once the reviewers approve the code, it can be added to the release.

Firmware Testing/Debugging

Test procedures will be provided by the vendor with the steps required for each test, plus the equipment required for testing. Any special fixtures will be provided by the vendor. All test procedures will be reviewed and approved by Meta before each build.

All bugs shall be documented inside GitHub bug tracker. Follow all GitHub rules and conventions for bug tracking.

Firmware Documentation

All firmware developed by Meta Accessories partner is required to have full documentation delivered to Meta for every build. Each document must contain revision control. This includes, but is not limited to, the following documents.

- System design (flow diagrams)
    - Full system flow chart
    - Entering/Exiting power states (deep sleep, run, charging)
    - Error States (FOD, OTP, OCP, SCP)
    - Calibration
    - Authentication
- Production/Test commands and usage
- Special cases in FW
- Battery charging/use cases (if applicable)
- UX
- Initialization
- Test Procedures
    - ERS HW validation
    - Functional validation
    - System validation
    - FW regression validation

Firmware Design Practices

Design

Don't immediately start writing code. Think about what needs to be implemented. Map out the design, think through issues, block out the state machines, algorithms, design components, then start on the code.

Design Reviews

Once a design is completed, run that through a design review. Make changes as necessary.

Code Reviews

Before code is released, it runs through a peer review process. This is to check for errors, coding conventions, etc.

Modularity

Divide your code into logical modules or functions to improve readability, reusability, and maintainability.

Layered Architecture

Implement a layered architecture with clear separation of concerns, such as separating hardware-specific code from application logic.

Event-Driven Design

Employ event-driven design to respond to asynchronous events and improve responsiveness. Limited busy loops allowed. Create event timers instead.

Power Management

Design your firmware to optimize power consumption, including sleep modes, clock scaling, and efficient use of peripherals.

Design the power management functions into the modules.

C Coding Practices

Readability

Write clear, well-structured code with meaningful variable and function names. Use appropriate indentation and formatting conventions. Be consistent with the style.

Commenting

Include comments to explain complex logic, assumptions, and rationale behind your code. Well-documented code helps others understand and maintain it.

Code Reusability

Promote code reusability by designing functions and modules with a single responsibility and avoiding unnecessary code duplication.

Portability

Write portable code that can be easily adapted to different hardware platforms or operating systems by minimizing platform-dependent code and using standard C libraries whenever possible.

Dead Code

No unreachable or commented out code should exist.

Code Simplification

Keep your code concise and avoid unnecessary complexity. Simplify control flow, reduce nested conditions, and eliminate redundant code. Keep functions small, e.g. avoid 1200 line functions. Break it down.

Code Organization

- Group related functions and data structures together in logical modules or files
- Utilize header files to declare interfaces and provide clear separation of concerns
- Organize functions within a file in a consistent manner (e.g., alphabetically, by functionality)

Error Handling

Implement appropriate error handling mechanisms, such as, return codes or error flags, and handle errors gracefully. **All errors must have a unique identifier.**

- Handle errors properly and consistently throughout the codebase
- Return error codes or use appropriate error handling mechanisms (e.g., return NULL for pointer errors)
- Provide clear and meaningful error messages or logging where appropriate
- Success should never be assumed- all error conditions must be checked.
- Log all errors with a unique identifier to a capturable sink or retrievable storage location when feasible

Memory Management

Be mindful of memory usage, avoiding memory leaks and excessive stack usage. Use dynamic memory allocation judiciously and ensure proper deallocation.

Avoid malloc/free as much as possible.

Type Safety

Make effective use of C's strong type system to catch type-related errors at compile time. Avoid casting excessively and ensure compatibility between different data types.

Compiler Warnings

Enable and address compiler warnings to catch potential issues and ensure adherence to standards.

Released code must be error and warning free.

Optimization

Optimize critical sections of code for speed or size, but balance it with code readability and maintainability. Profile your code to identify bottlenecks before optimizing.

Configuration

Use files that contain configuration information.

E.g. A state machine that triggers off of various voltages and timings should have those values defined in a configuration file.

ISRs

ISR handlers should be simple and quick. Grab the items that are needed, set appropriate events, then get out. No looping in ISRs allowed.

Single Entry/Exit

- Try to have a single entry/exit in functions
- If necessary, use early return statements to avoid deep nesting of if statements

C Coding Conventions

These coding conventions will be used for writing C code.

C Standard

Follow the C11 standard

Naming Conventions

- Use meaningful and descriptive names for variables, functions, files, directories, and constants.
- Use **snake_case**.
- Use uppercase letters for enums, and macros (**MAX_VALUE**).
- Static Constants (**c_constant**)

Function Organization

- Organize functions in a file in a consistent way.
- Alphabetically, functionality, …

Data Types

Use size-specific data types (e.g., int32_t, uint64_t) for portable code.

Constants vs Macros

Use named constants, enums instead of hardcoding values (magic numbers) to enhance code maintainability.

Prefer const variables over macros when possible to benefit from type checking and scope limitations.

Preprocessor

Avoid unnecessary use of preprocessor directives.

Indentation

Spaces are used for indenting. 4 spaces per indent.

Line Width

Try to keep it at 80 characters. Use discretion.

Spaces and Parentheses

- Use parentheses to clarify complex expressions and operator precedence.
- Spaces after all commas and semicolons (if not at the end of a line)
    for (int i = 0; i < number; i++)
- Space after reserved words (except sizeof)
- Space before parenthesis for control structures.
    if (foo), while (foo)
- Space around math and binary operators.
    (2 + 7 / x) not (2+7/x)

Blank Lines

Use them to separate logical blocks inside functions.

Braces Style

Use linesaver mode for opening braces

```
if (foo) {
```

```
while (true) {
```

Control Structures (if, while, for, …)

- Use clear and concise control flow structures (e.g., if-else, switch-case) to improve code readability.
- Avoid deep nesting of control flow structures by breaking down complex logic into smaller functions or using early return statements.
- Always use braces. Even for single statements.
- Always put single statements on the next line.

```
    if (verbose) {
        log("There are too many discussions about style guides!");
    }
```

- Use **if (count != 0)** or **if (count > 0)** instead of if (count) for variables which are counters
- The if () conditional should always be an explicit boolean expression. For instance use **if (ptr != NULL)** instead of if (ptr).
  On the other hand, if the variable is already a boolean, you should do if (x) instead of if (x == true) or if (!x) instead of if (x == false).
- Use if (0 == x) instead of if (x == 0). The compiler will throw an error if there is a typo of If (0 = x)
- Forever loops are done with while(true)
  while (true) {}
- Avoid modifying loop variables in a loop.

Switch-case

Use FALLTHROUGH and NOTREACHED comments to mark the relevant situations. lint(1) can understand them and so do some code checkers (coverity) and it allows human readers to identify when it's done explicitly rather than by mistake. Do not, however, be excessively clever. Fall-through should be reserved for cases where there is clear runtime benefit.

```
switch (ch) {               /* Indent the switch. */
    case 'a':               /* Indent the case. */
        aflag = 1;
        /* FALLTHROUGH */
    case 'b':
        bflag = 1;
        Break;
    case '?':
    default:
        usage();
        /* NOTREACHED */
}
```

Variables

Use expressive variable names. Do not abbreviate unnecessarily and always optimize for readability. For instance use rectangle instead of rect, VirtualAddress instead of VA and so on.

Do not declare more than one variable on a line (this helps accomplish the other guidelines, promotes commenting, is good for tagging).

Initialize variables at the time of declaration if possible for instance if constants are assigned to variables.

Avoid patterns like **int* a = (int*)param; int b = a->field;** in initializations because they often lead to accidental NULL pointer dereferences. Just declare a and b in this case and defer the initialization.

Declare your variables in the smallest scope possible. Avoid global variables.

Declare your variables as close to the usage point as possible within the given scope. Put variable declarations before code but close to the scope they are being used in, i.e. don't group all your variables at the function scope.

If the variable or function parameter is a pointer type, group the * with the type
```
int* number;
```

Zero, NULL, Constants, and Magic Numbers#

Use `const` whenever possible for variables and arguments. Use it in pointer and pointed-at positions. Examples: `foo_t const* const param`, `static uint32_t const c_num_pages = 1024;`. Rationale: Communicate intent to the reader, catch bugs, and give the compiler information for better code.

Do not use variables, even `const` ones, in array definitions. Instead use a macro or enum. Rationale: Using variables defines a variable-length array, which is not permitted by this style guide. This is true even if the variables are `const`.

```
static const uint32_t c_num_foo = 32;

void
```

```
function(void)
{
  foo_t many_foo[c_num_foo];   // DON'T DO THIS --variable-length
array
}
enum {
  NUM_FOO = 32,
};

void
function(void)
{
  foo_t many_foo[NUM_FOO];   // OK -- constant-lengtharray

}
```

Use `NULL` if you need a constant that represents a null pointer. Use `0` to represent the numeric value.

Do not use unnecessary casts. Use `NULL` instead of `(type*)0` or `(type*)NULL` in contexts where the compiler knows the type, e.g., in assignments. Use `(type*)NULL` in other contexts, in particular for all function args. (Casting is essential for variadic args and is necessary for other args if the function prototype might not be in scope.)

Do not use magic numbers in the code (no literals of any numbers other than `0`, `1`, `-1` except as the initializer for a constant). In some cases, if there's no chance the value will be used elsewhere, it can be more readable to use a magic number inline if accompanied by a comment explaining the value. E.g. multiply by two in a loop to try to find best fit, etc.

For floating point literals, never omit the initial `0` before the decimal point (always `0.5f`, not `.5` or `.5f`). Do not omit the trailing `0` postfix (always `1.0f` not `1` or `1.0`). Do not omit the trailing `f` when defining floats (always `1.0f` not `1.0`). When defining doubles use `1.0`.

Function Definitions

Function definitions shall contain the return type followed by a new line followed by the function name.

Function name and parameters should be on the same line unless the parameters don't fit into the 80 character limit. If parameters don't fit into the 80 character limit each parameter shall be on a new line indented by two blocks (4 spaces) from the function name.

Beginning and end curly braces shall be on a new line

If using a prototype, static functions should have the static label on both prototype and implementation.

Examples

```
int
initialize_rectangle(uint32_t top_x, uint32_t top_y)
{
  // Function body goes here
}

int
map_vmo_into_address_space(
    virtual_memory_object_t* vmo,
    virtual_address_region_specifier_t* vmars,
    map_vmo_flags_t* flags)
{
  // Function body goes here
}

int
initialize_rectangle(
    uint32_t top_x,    // X coordinate of the top left corner
    uint32_t top_y,    // Y coordinate of the top left corner
    uint32_t bottom_x, // Y coordinate of the bottom right corner
    uint32_t bottom_y) // Y coordinate of the bottom right corner
{
  // Function body goes here
}

static const xr_ipc_service_handler_t*
echo_ipc_handler(flatbuffers_thash_t type_hash, const void* context);

// some other code....

static const xr_ipc_service_handler_t*
echo_ipc_handler(flatbuffers_thash_t type_hash, const void* context)
{
  // Function body goes here
}
```

Function Declarations in Header Files

- Function declarations in header files shall contain the return type, function name and parameters on the same line.
- If the declaration exceeds the 80 character limit then the parameters shall continue on the next line indented by two blocks.
- Do not declare functions inside other functions; ANSI C says that such declarations have file scope regardless of the nesting of the declaration.

Examples

```
int initialize_rectangle(int top_x, int top_y, int bottom_x, int
bottom_y);

int map_vmo_into_address_space(virtual_memory_object_t* vmo,
    virtual_address_region_specifier_t* vmars, map_vmo_flags_t*
flags);
```

Type Definitions

- Use single lines for type definitions for literal types
- Use multi-lines for type definitions for structs and enums
- All type definitions are post-fixed with _t.
- Major structures should be declared at the top of the file in which they are used
- Every member of a struct or enum shall be on a new line
- Use typedefs for structs and enums. Use anonymous structs and enums unless it’s impossible to avoid.
- Do not overload identifiers for struct/enum and other identifiers i.e. don’t use the same name for a say a struct and an enum and have the only differentiator be that the type of a variable is declared as struct Foo vs enum Foo.
- Make the struct name match the typedef, e.g. typedef struct foo foo_t if a struct is not anonymous.
- Use comments liberally. Comments should generally explain why something is done, not what is being done, unless it’s not obvious. Avoid i++; // increment i.
- Use bool (stdbool.h) and true/false for boolean types.
- Use int64_t, uint64_t, int32_t, uint32_t and other exact width integer types instead of int (stdint.h).
- Each element in a structure should have a comment if not very obvious.
- If the lines are long, put the comment before each entry.

- Some people seem to leave blank lines between elements which have comments above them.

Examples

```
typedef uint64_t virtual_address_t;
typedef struct {
  int top_x; // X coordinate of the top left corner
  int top_y; // Y coordinate of the top left corner
  int bottom_x; // X coordinate of the bottom right corner
  int bottom_y; // Y coordinate of the bottom right corner
} rectangle_t;

/// A single line Doxygen comment about the structure.
typedef struct {

  /// A Doxygen comment about the following line.
  int really_long_name_that_leaves_no_room;

  /// A Doxygen comment about the second element.
  int another_really_long_name_that_leaves_even_less_room;

  /**
   * A multi line Doxygen comment about following element
   * that gives a lot of information.
   */
  int another_really_long_name_that_leaves_no_room;
} my_example_t
```

Macros

Avoid function-style macros. Static inline functions are preferable if at all possible. Rationale: Static inline functions get type-checking and avoid double-evaluation and syntax issues caused when macro arguments are expanded.

Definitions of expression-like macros must be parenthesized unless the macro cannot work correctly with the parentheses. Rationale: Surrounding the definition with parentheses can prevent unexpected interactions with surrounding code. However, it may not be possible to parenthesize some macros, such as those that expand to types or those whose results are used with the stringize (#) operator.

Example:

```
#define NUM_VPNS 0x1000
```

```
// CORRECT -- With surrounding parentheses
#define END_VPN (VPN + 1)
release_vpns(NUM_VPNS - END_VPN);    // Releases (0x1000 - VPN - 1)
VPNs

// INCORRECT - Without surrounding parentheses
#define END_VPN VPN + 1
release_vpns(NUM_VPNS - END_VPN);    // Releases (0x1000 - VPN + 1)
VPNs -- two more than intended
```

However, in this example, the best solution is to use `static const` variables instead of macros. `static const` variables are safer and should be used in preference to macros when possible. See . With `static const`, the example may be rewritten to the following:

```
// Use static const variables instead of macros.
static uint64_t const c_num_vpns = 0x1000;
static uint64_t const c_end_vpn = VPN + 1;
release_vpns(c_num_vpns - c_end_vpn); // Releases (0x1000 - VPN - 1)
VPNs
```

Macro arguments used in the definition must be parenthesized unless the macro cannot work correctly with the parentheses. Rationale: This prevents unexpected results when expanding an argument consisting of multiple tokens. However, macros using the stringize operator `#`, the concatenation operator `##`, or implicit concatenation of adjacent string literals cannot use parenthesized arguments.

Example:

```
/*
 * CORRECT -- all parameter uses are parenthesized
 *
 * XR_USEC(100 + 50) expands to XR_DURATION(UINT64_C(1000) * (100 +
50)), which is 150 usec.
 */
#define XR_USEC(_n) XR_DURATION(UINT64_C(1000) * (_n))

/*
 * INCORRECT -- a parameter use is not parenthesized
 *
 * XR_USEC(100 + 50) expands to XR_DURATION(UINT64_C(1000) * 100 +
50), which is 100.050 usec.
 */
#define XR_USEC(_n) XR_DURATION(UINT64_C(1000) * _n)
```

Put a single space character between the #define and the macro name. Rationale: Consistent code style improves readability.

If a macro is an inline expansion of a function, the function name is all in lowercase and the macro has the same name all in uppercase. Rationale: Consistent code style improves readability.

To define a no-op or empty function-like macro, define it as `((void)0)`. Do not use an empty macro definition. Rationale: Defining the macro as `((void)0)` makes it behave more like a real function, including requiring a semicolon as a statement terminator, thus making it more likely the macro will be used correctly when it is not empty.

In multi-line macro definitions, vertically align the line-continuation backslashes. Rationale: This makes the macro easier to read.

In blocks defining registers, addresses or other closely related things, right justify the values for readability. Sort the registers by their address or group them by function if these registers belong to the same logical entity. Do not leave them in a random order.

```
#define PERI_CRG_RSTEN4    0x90
#define PERI_CRG_RSTDIS4   0x94
#define PERI_CRG_ISODIS    0x148
```

If the macro expands to a compound statement, enclose it in a statement expression (https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html) if it is supported by the compiler (e.g., gcc and clang). Rationale: This allows the macro to be safely used in `if` statements and loops without full bracing and works around a clang limitation. The common idiom of `do { .. } while(0)` has a similar purpose, but clang cannot correctly analyze unreachable code when it is used.

```
#define ASSERT(cond)                         \
  ({                                         \
     if (unlikely(!(cond))) {                \
       PANIC("ASSERT FAILED at: %s", #cond); \
     }                                       \
  })
```

Macro definitions shall not end with a semicolon. The statement-termination semicolon should be supplied at the macro invocation point rather than by the macro. Rationale: Including a semicolon causes the macro definition to behave syntactically differently from a function call and can cause suprising results if the macro invocation is followed by a semicolon within an unbraced loop or conditional.

```
#define MACRO(x, y)          \
  ({                          \
     variable = (x) + (y);   \
     (y) += 2;               \
  })
```

Sizeof

Prefer `sizeof(type)` over `sizeof(*var)`.

Rationale: using `sizeof(type)` follows the principle of optimizing for reading the code instead of writing it as it does not require the reader to find the type of the variable.

Unused Variables

XROS development requires that the master branch both compiles and does not accrue static analysis errors on a few different builds. This causes some issues for variables that are introduced to help with debugging in the development builds. The two following sections describe how to deal with the main sources of valid unused variables: assertions, and everything else.

Assertions

Assertions are a recommended mechanism to validate invariants of the code for internal call path. They help document assumptions a function makes. However, they are not a mechanism to avoid arguments checking of public APIs (e.g. library, syscall). Assertions should not be used to verify that an argument is not NULL for instance, instead public APIs should return the appropriate error code. Assertions can be used to validate that an internal state of the API is valid, though.

Do not wrap expressions with side effects in assertions, i.e. `ASSERT(important_function())` or `ASSERT(important = vImportant())`. This causes different behavior on release and developer builds.

```
void important_callback(xr_important_object_t* modifiable)
{
  xr_error_t assert_error;
  assert_error = modify_input_with_side_effects_one(modifiable);
  ASSERT_XR_OK(assert_error);
  assert_error = modify_input_with_side_effects_two(modifiable);
  ASSERT(assert_error == XR_ERR_NOT_FOUND);
}
```

If a variable is only used by the message of an `ASSERT_MSG`, the variable must be declared as `__ASSERT_ONLY` to mark it as unused in non-debug builds.

```
void verify_header(header_t* header)
{
  __ASSERT_ONLY xr_uuid_formatted_t uuid_read;
  ASSERT_MSG(header->magic == c_stress_crc_magic &&
    header->data_length == length && xr_uuid_equals(uuid,
&header->uuid),
    "header: magic=%lx length=%lx uuid=%s", header->magic,
    header->data_length, xr_uuid_format(&header->uuid, &uuid_read));
}
```

VLAs (Variable Length Arrays)

Variable length arrays (https://clang.llvm.org/compatibility.html#vla) are not permitted. Statically sized arrays or heap allocation should be used wherever a variable-length array would have been used. The rationale for this is fully documented in T57940410, but boils down to:

1. VLAs are by far the easiest way to accidentally cause a stack overflow which can, at worst, create an avenue for an attacker to crash your code or leak memory.
2. Any “safe” VLA must have a statically determined maximal size, so if the array must be allocated on the stack instead of the heap, allocating a maximal size array instead of a variable-length array is always possible.
3. VLAs will generate code at least as slow as or slower than a statically sized array.

Flexible Array Members

Flexible array members are permitted. In a struct with more than one member, the last member of the struct can have an incomplete array type. Such a member is called a flexible array member of the struct, and enables one to mimic dynamic type specification in C in the sense that you can defer the specification of the array size to runtime. A flexible array member must be the last member, and must be specified as `a[]`, not `a[0]`, so the compiler can detect misuse. Here is an example struct with a flexible array member, `a`:

```
typedef struct {
    int len;
    char a[];
} my_struct_t;
```

Allocating memory can be then be done via:

```
my_struct_t *s = xr_malloc(offsetof(my_struct_t, a[n]));
s->len = n;
```

Copyright

We are moving away from the historical practice of using copyright notices. Use the following confidentiality notice in the source code.

```
/*
 * This software contains information and intellectual property
 * that is confidential and proprietary to Facebook, Inc. and its
affiliates.
 */
```

When significant changes are made to files that have 3rd party copyright, add the Facebook copyright to the original copyright notice:

```
// Copyright 2017 The Fuchsia Authors. All rights reserved.
// Copyright (c) Facebook, Inc. and its affiliates. All Rights
Reserved.
// Use of this source code is governed by a BSD-style license that can
be
// found in the LICENSE file.
```

When contributing new code based on 3rd party code, add a Facebook copyright to the existing block. For example:

```
/*
 * Copyright (c) Facebook, Inc. and its affiliates. All Rights
Reserved.
 * Copyright (C) 2017-2018 Hilisicon Electronics Co., Ltd.
 *              http://www.huawei.com
 *
 * Authors: Yu Chen <chenyu56@huawei.com>
 *
 * This program is free software: you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License version 2  of
 * the License as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */
```

In all these cases the linter should detect and propose fixes. The relative position of the Facebook copyright is not important, although there is a weak preference to have it first for readability purposes.

Go to https://our.intern.facebook.com/intern/wiki/Copyright_Headers/ to see all corner cases.

Comments

Code is NOT self documenting.

Include comments to explain the purpose, behavior, and assumptions of functions, blocks of code, and complex logic. Especially if there is a reason for doing something a specific way.

Document function prototypes, input/output parameters, return values, and any error conditions.

Avoid excessive or redundant comments that don't add meaningful information.

For single line code comments, always use `//` (with a space separator before the comment).

```
// This is a single-line comment
```

For multi-line code comments use `/* ... */`. The comment has to be on different lines than the beginning and end tokens.

```
/*
 * This is a
 * multi-line
 * comment.
 */
```

Documentation

Headers should use Doxygen (with at-sign prefix) to document function, type and interface declarations and definitions. The opening `/**` and end `*/` of a multi-line comment shall stand on lines of their own.

Use `///` for single line comments. Rationale: Consistency with other single-line comments.

Use `/** ... */` for multi-line comments. Rationale: Consistency with other multi-line comments.

```
/**
 * Brief description which ends with a '.'.
 * Detailed part of the comment started here.
 */
```

Use the `@`-notation for Doxygen directives. The `\`-notation shall not be used. For example, use `@file` and `@param`, not file and param.

File Documentation

Source files must include the `@file` tag, placed immediately after the copyright notice. `@file` should be left unspecified by default, and should only be specified if the source file name is not unique in the codebase. Rationale: Doxygen requires the `@file` tag, and by default uses the name of the given source file if the directive is blank.

Example file documentation, positioned directly after the copyright notice:

```
/** @file [file name, with partial directory if needed]
 * This is a brief description of my module.
 *
 * The detailed description starts here. It contains a link to design
 * docs created that cover the implementation in this module.
 */
```

For any program, the file documentation for the source file containing its main routine (entry point) must describe what the program does and how to invoke it. Rationale: This is essential documentation for understanding and using a program.

Function Documentation

All functions must have Doxygen-format documentation at their point of definition. Function declarations in public header files must also have Doxygen-format documentation. Other function declarations shall not have documentation blocks. Rationale: in most cases, documentation at only one point is adequate. However, public header files need to document the API for use by developers who might not have access to our source code. Furthermore, public and internal documentation for the same function may have a different level of detail, particularly when the public documentation stresses an abstraction and the internal documentation discusses the implementation details of that abstraction.

Example function documentation:

Example 1:

```
/**
 * Create a uni-directional IPC receiver channel.
 *
 * Setup the kernel representation of a uni-directional IPC channel.
 * The pages belonging to the IPC channel are getting mapped into the
 * kernel address space.
 * The IPC channel acquires references to the underlying physical
pages
 * used to back the channel, and to the port. The channel needs to get
 * destroyed before those resources can be freed.
 *
 * To be able to guarantee that the kernel can post a notification to
the
 * port to signal that user space should check the IPC channel for new
 * messages, the IPC channel takes a reservation on the port.
 *
 * @note To close the provided port, the IPC channel needs to be
closed
 *       first.
 *
 * @param[in] setup Struct to initialize IPC channel.
 * @param[in] produce_mem_object_header Object header to produce
memory.
 * @param[in] consume_mem_object_header Object header to consume
memory.
 * @param[in] msg_buffer_mem_object_header Object header to message
buffer
 * memory.
 * @param[in] control_mem_object_header Object header to control
information
 * memory.
 * @param[out] ipc_channel_object_header Object header for new IPC
channel.
 *
 * @retval XR_OK On success.
 * @retval XR_ERR_INVALID_ARGS One of the input arguments is invalid.
 * @retval XR_ERR_NO_VIRTUAL_ADDRESS_SPACE No virtual address space is
 * available to map in pages.
 * @retval XR_ERR_NO_MEMORY A memory allocation failed to allocate the
data
 * structure which tracks the IPC channel.
 * @retval XR_ERR_UNAVAILABLE An associated memory region is intended
to be
 * freed.
 */
 xr_error_t ipc_channel_create(const xr_ipc_channel_setup_t* setup,
    object_header_t* produce_mem_object_header,
    object_header_t* consume_mem_object_header,
    object_header_t* msg_buffer_mem_object_header,
    object_header_t* control_mem_object_header,
    object_header_t** ipc_channel_object_header);
```

Example 2:

```
/**
 * Check whether the message buffer can store the additional message.
 *
 * A message may span across multiple buffer split regions. However, a
message
 * may not wrap around, and must always be virtually contiguous.
 * If a new message would wrap around, we restart it at offset 0 of
the buffer.
 * If a message doesn't start at the beginning of a split buffer area,
we know
 * that the split region must have been fully read at the time when
the initial
 * message was added to the buffer, which means we can just add a new
message
 * into the split region.
 * If a message crosses into a new split region, it may only do so, if
the
 * corresponding counter is zero, which means that user space has
consumed
 * all messages previously associated with that split region.
 *
 * @param[in] channel IPC channel for which to enqueue a new message.
 * @param[in] msg_length  Length of the message including handle
space.
 * @param[out] buf_pos [optional] Buffer position from which the new
message
 * starts from. Can be set to 0 if the message would have wrapped
around the
 * buffer.
 *
 * @return True if the message can be enqueued, false otherwise.
 */
static bool
ipc_channel_check_msg_buffer(
    ipc_channel_t* channel,
    uint32_t msg_length,
    uint32_t* buf_pos)
```

Function documentation consists of the following elements:

1. A description of the function's effect. Rationale: This is the most basic essential documentation.
2. The range limitations of all function arguments, if those arguments are not valid over their entire usual range. Rationale: This is essential information for using the function safely, especially for internal functions that may not be range-checked.
3. All function parameters and their meanings. Rationale: This is important information for using the function.
4. The meaning of return values from the function. Rationale: Important information for using the function.
5. Any notes or warnings required to use the function correctly or safely. These should document any behavior that may be considered surprising by other engineers using the function. Rationale: Whenever possible, design functions to have no surprising behavior. When such behavior cannot be avoided, give clear advance notice to minimize surprise.

The `@brief` directive shall not be used for function descriptions. Rationale: `@brief` does not affect the rendering of the detailed documentation for a function. When present, it merely adds the annotated paragraph (regardless of the number of lines it spans or sentences it contains) to the the table of contents at the top of the page. Avoiding the use of `@brief` reduces extra labor and visual distraction.

When a function declaration is documented, the documentation must include all information necessary for correct use of the function, but may omit implementation details unnecessary for using the function.

All function documentation in public headers, and all documentation for functions that are published or intended for use across teams or externally, shall include the function description. For other functions, the description of effect is highly recommended. However, it may be omitted if the function's name makes its effect crystal clear and unambiguous. Rationale: Require the description where it is needed and avoid labor where it is not.

The function description, if present, must outline the major details that are required to build an understanding of the function, and any possible gotchas or other quirks. Rationale: Require the basic level of documentation.

If the function implements a complex, tricky, or far-out-of-the-ordinary algorithm, the function description must document what a programmer calling the function needs to know about the algorithm in order to use it safely and correctly. Such a function must also have enough documentation to allow engineers to read and modify the code; this documentation may be placed either in the function description or as comments appropriately placed within the function. Rationale: Provide the needed levels of understanding to clients and maintainers.

For function documentation not in a public header file and not intended for external use, if a published book, paper, etc. was used in writing the function and the algorithm used is outside the knowledge of a typical engineer, a citation to that book, paper, etc., shall be included in the function description. Rationale: Provide needed levels of understanding for maintainers.

Function parameters must be documented unless the function signature makes the meanings of all parameters crystal-clear and unambiguous. Rationale: Require documentation where needed and avoid labor where it is not.

If any parameter of a function is documented, all of that function's arguments shall be documented. Rationale: Avoid ambiguity and incompleteness.

Function parameters shall be documented with the `@param` directive. The parameter documentation shall also include the units and/or range of the parameter if these are not crystal-clear and unambiguous. The @param directive shall indicate the direction of data flow for each parameter, as described below.

- `in` - parameter is read by the function and scalar/struct/etc is never modified.
    - e.g. `str` in `size_t strlen(const char *str);`
- `out` - parameter is only written by the function.
    - e.g. `dest` in `void* memcpy(void *dest, const void *src, size_t n);`
- `in,out` - parameter is read by the function and possibly modified.

  - e.g. `size` in `compute_initial_stack_pointer(uintptr_t base, size_t* size)` as size is the initial size of the stack area, which is reset to reflect the actual size after the computation.
  - `in,out` should not be used in places where functions are taking in a parameter and changing the internal state
- `optional` - Doxygen doesn't support optional arguments. Hence add the `[optional]` keyword prior to the description of the parameter.

The `@param` lines shall have no space between `@param` and the direction brackets, a single space between the brackets and the parameter name, and a single space between the parameter name and its description. Additional space for vertical alignment should not be added. If a parameter description spawns multiple lines, the parameter description should continue at the beginning of the next line following standard doxygen formatting. Existing code may not follow these rules; it does not have to be updated until it is touched. Rationale: This formatting is adequately readable and less time-consuming than maintaining vertical alignment.

In the event of disagreement on whether anything that might be omitted is crystal clear and unambiguous, it is by definition unclear and shall not be omitted. Rationale: Err in favor of over-documenting.

If return values are explicitly called out, the individual return values need to be on their own line(s) and must start with `@retval`. The description should not start with "If …" but just state under which condition the return value could be observed. If no explicit return values are called out, or form a simple sentence `@return` should be used. Examples for `@return`:

```
/**
 ...
 * @return Pointer to an ipc_channel_t on success, NULL otherwise.
 */

/**
 ...
 * @return True if the message can be enqueued, false otherwise.
 */
```

The use of other markup to enhance the usability of function documentation, such as using `@ref` to cross-reference related documentation, is permitted but not required.

Struct and Union Documentation

Every struct or union declaration must be documented. The documentation must describe its purpose or use. Rationale: When reading code, it is helpful to understand the overall meaning of a struct or union. Documentation of the struct or union's members is usually not sufficient.

Every member of a struct or union must be documented. However, documentation may be omitted for a member if its purpose and use is crystal-clear and unambiguous from its declaration. Rationale: Enable use, understanding, and modification of the struct.

In the event of disagreement on whether anything that might be omitted is crystal clear and unambiguous, it is by definition unclear and shall not be omitted. Rationale: Err in favor of over-documenting.

Header Documentation

Doxygen interface comments are also mandatory in exported .h headers consumed by developers (e.g. headers that will be in an SDK), but not in internal or private header files. The identification of these files shall be unambiguous per the parts of this document defining library & build system conventions.

Documentation in exported headers should be written for public consumption, with a level of detail up to a man page. It shall include examples of use. Because the audience is different, this documentation is distinct from and is expected to be different from documentation for the same functions in C files.

Due to the labor required to keep these headers in-sync, including professionally written content from tech writers, it is acceptable to defer public header file documentation during development. All files for which documentation is deferred shall include a file-specific TODO task comment like: `// TODO(T123456): Add public header Doxygen comments`

Special Comments

Use `// TODO(<task id>): <description>` comments when something needs to be fixed. Do not use your username instead of a task id. Include a brief description to give sufficient context to a reader.

Code Organization

- `.h` files should be in same directory with corresponding `.c` files

- Unittest files should be under a subdirectory called "test" in the same directory.

Include Files

- Avoid using unnecessary includes.
- Use double quotes for local includes, and angle brackets for system-wide includes.
- When including headers, sort headers by scope so system include files appear first, project-wide include files second, and local include files last. For headers at the same scope, sort by name.
- Headers must be self-contained, so that they can be included without including other headers first. If they depend on opaque types, e.g. for argument types of functions declared in the header, they should include the header declaring the opaque types, not a header with full type definitions.
- Use `#pragma once` for all new headers to prevent duplicate definitions.

```
% cat hikey960-hw.h

#pragma once

...

...
```

- `...`
- Do not use relative paths. For instance do not write `#include "../../foo.h"`