Coding Guideline

Coding Standards for C language

File Organisation

Files may contain items in the following order.

File Banner

```
/*
 * If not stated otherwise in this file or this component's Licenses.txt file the
 * following copyright and licenses apply:
 *
 * Copyright 2020 RDK Management
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
*/
```

File Guards

In all header files, do the following (using the exact file names) to prevent an header file from being included more than once: For example in C do the following

```
#ifndef FILENAME_H

#define FILENAME_H

// File content

#endif // Last line of code in file
```

This is to avoid multiple definitions, and to ensure that the ifndef checks a unique identifier.

File Includes

```
/** @addtogroup XXX_API XXX Public API
* Description.
* @{
*/

/*****************************************************************************
* STANDARD INCLUDE FILES
*****************************************************************************/
#include <stdint.h>

/*****************************************************************************
* PROJECT-SPECIFIC INCLUDE FILES
*****************************************************************************/
//#include "xxx_misc.h"

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

Functions

Every function should have a function banner as follows :

/* XXX_function name : . */

```
/**
* Description. Briefly describe what the function does
*  Parameters : List all function parameters and provide a description of each parameter along * with the type
of the parameter.

* (A table with parameter Name, Description, And * Type may be created for *clarity.)

* @param[in] arg1 Some argument.
*
* @return The status of the operation.
* @retval XXX_OK if successful.
* @retval XXX_ERR_NOTINIT if the module was not initialised.
* @retval XXX_ERR_BADPARAM if a bad parameter was supplied.
*
* @execution Synchronous.
* @sideeffect None.
*
* @note This function must not suspend and must not invoke any blocking system
* calls. It should probably just send a message to a driver event handler task.
*
* @see XXX_SomeOtherFunction.
*/
```

Visual Impact

Size

A printed page should be no more than 72 columns.

Number of statements in a file should not exceed 1000 and the number of lines 2000.

Number of statements in a function should not exceed 100 in a function and the number of lines 200.

Indentations

Use spaces rather than tabs as printers as users might have different tab settings

Use single line spacing between logical blocks of code.

Use double line spacing between functions.

Comments

Strategic Comments

These comments give a generic overview of what is going on. they appear at the top of files and before function codes in file and function banners. These are to be included in the descriptive section of file and function banners

Tactical Comments

These comments explain what parameters do, control flow etc.. These comments should be embedded in the code and tend to be shorter and simpler and in bullet format

Closing Comments

Comment the end of function (for e.g. in C the curly bracket "}") with a // end function name - for Clarity and consistency.

Comment the last line of each file with // End FileName - so that in printed output, you know when you have the last page.

Comment the closing block (for e.g. in C the curly bracket "}") in all conditional blocks and loops with // end if, // end for, etc.
It is not necessary to comment intermediate brackets in an if - else construct - for Clarity .

Declarations and Definitions

Declare/Define every variable on a separate line.

Put a comment at the right of variable.

Every variable that is declared must be used.

Explicitly initialize all variables before use.

Initialize all pointers to 0 or to an object before use.

It is not recommended to declare variables local to a block of statements. This makes code difficult to maintain and avoids declarations from being made anywhere in the code.

Naming Conventions

All identifiers (variables, constants, Classes etc.) declared should have meaningful names.

Have naming conventions to differentiate between local and global data.

Identifiers may have their types attached to their names for clarity and consistency.

In case where the language has support for header file, ensure all user defined header file should have the same name as the source file that is referenced in.

Names should be readable and self documenting. Abbreviations and contractions are to be discouraged. Abbreviations are allowed when they follow common usage within the domain.

Identifiers should not exceed 31 characters.

Data Type	Prefix	Sample Variable Names
int32	i	iSequenceNo
unsigned int32	ui	uiSequenceNo
int16	n	nSequenceNo
unsigned int16	un	unSequenceNo
BOOL	b	bTransmitted
Char	ch	chSequenceNo
unsigned char	uch	uchSequenceNo
long	l	lSequenceNo
unsigned long	ul	ulSequenceNo

enumerated data type	e	eCapabilityMode
Arrays	a	int32 aiSequenceNo[10] etc
Pointers	p	int32 piSequenceNo etc …
Globals	g	giSequenceNo, gpTimer etc

Typedef	All the type names shall be suffixed with _t.	Typedef struct _IPAddessInfo
		{
		}IPAddressInfo_t;

Statements

Only one statement should exist per line.

Avoid use of numeric values in code; use symbolic values instead.

Many small, simple statements are better than fewer, large and complex statements. Do not write "Clever" code - it should be simple and straight forward.

Use a space before an after an operator (for e.g. in C use - if (value == 0) and not - if (value==0))

Conditional statements found in if, while, do etc. should be explicit based on the data type of variable tested for.

for e.g. in C++

```
if ( value == 0 ) // right
{
doso() ;
}
if ( !value ) //wrong
{
dontdoso();
}
```

Inaccessible code should be avoided. Care should be taken when using GOTO or return statement .

Multiple assignments are not recommended. e.g. a=b=c ;

Maximum 4 Levels of Control Structure Nesting : Nesting of statements, "if", "for", "while", etc., should go no more than 4 levels. If more levels appear to be needed,
consider use of a function at one of the higher levels.

Null statements must include a comment line. For example, if the "default:" case in a switch statement does nothing,
put in a "default:" label followed by a comment to the effect: /* no action */ followed by the break statement.

The null body of a "for" or "while" loop should be alone on a line and commented so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
{
; /* VOID */
}
```

All control statements should be followed by an indented code block enclosed with braces, even if it only contains one statement. This makes the code consistent and allows the block to be
easily expanded in the future.

For example in C++ use :

```
if ( value == 0 )
{ // right
doSomething();
}
if ( value == 0 ) doSomething(); // wrong - no block, not indented
if (value == 0)
doSomething(); // wrong - no block
```

Automatically Generated Code

Document the name of the automatic code generation product and add a high level description of the options/settings in the "*Description*" section of File banners (from this guideline) to all
code that is created by automatic code generation products such as Visual C++ Wizards.

Document any manual modifications to the automatically generated code in the function abstract and code comments. If you modify an automatically generated function describe the modification.

Reason: To minimize the re-work of automatically generated code, while providing enough documentation for code maintenance.

Copied, cut and paste, or "borrowed" code : Format all code that is copied from other sources so that it conforms to this Coding guideline. Comment the copied code with a block that states
the source of the cut and paste, such as a Software Developers Kit (SDK), tutorial example, or commercial package. All hard-coded constants must be redefined with symbols
that have meaningful names.

Reason: to minimize the re-work of "borrowed" code, while providing enough documentation for code maintenance.

Coding standards for Python

Resources : Documents/Tools

- PEP8 - coding conventions SI team use to write / format SI python scripts.
- PEP257 - documentation conventions;
- Google Python Style Guide - python practices;
- Pylint - default code analysis tool SI team use to verify python code There are 2 more tools that may complement the checking:

    https://github.com/jcrocholl/pep8 - enforces PEP8 checks;

    http://pychecker.sourceforge.net/ - Google recommends in its doc;

Rule #0

The main rule that takes precedence in any language.

Google coding style: **BE CONSISTENT**.

- If you're editing code, take a few minutes to look at the code around you and determine its style.
- If they use spaces around all their arithmetic operators, you should too.
- If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.
- The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're saying it.
- If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

Naming

Type	Public	Internal
Packages	`lower_with_under`	
Modules	`lower_with_under`	`_lower_with_under`
Classes	`CapWords`	`_CapWords`
Exceptions	`CapWords`	
Functions	`lower_with_under()`	`_lower_with_under()`
Global/Class Constants	`CAPS_WITH_UNDER`	`_CAPS_WITH_UNDER`
Global/Class Variables	`lower_with_under`	`_lower_with_under`

Instance Variables	`lower_with_under`	`_lower_with_under (protected) or __lower_with_under (private)`
Method Names	`lower_with_under()`	`_lower_with_under() (protected) or __lower_with_under() (private)`
Function/Method Parameters	`lower_with_under`	
Local Variables	`lower_with_under`	

Names to avoid

- single character names except for counters or iterators
- dashes (`-`) in any package/module name
- `__double_leading_and_trailing_underscore__` names (reserved by Python)

Line Length

- Maximum line length is 100 characters.
- This deviates from PEP8 / Google recommendations.
- Rationale: we don't have devices that may need use 80 characters per line. 100 is a reasonable length that fit line on 14-17" monitors.
- Use wrapping for really long lines as it is described in http://www.python.org/dev/peps/pep-0008/#maximum-line-length
- Try to not use slash to split the lines.
- But in contrast to google guide mixing both styles is okay for SI scripts. The main thing is to avoid long lines forcing reader to scroll code back and forth.

Indentation

- Indent code blocks with 4 spaces. Never mix spaces and tabs.

PyLint:use code checkers

- Install and use pylint for the code written as much as possible. Catch problems earlier. Please note that this is just a tool.
- Always review warnings / errors, and if some of them can be safely ignored or tool just confusing - suppress them.
- Don't leave warnings in code - suppress. If there are many warnings that may be ignored, important things can be missed due to this noise.

Documentation

- All public classes, methods, variables should be documented. Refer to PEP257 for more details.

Code documentation guidelines

General Guidelines

Only one statement per line.

Indentation- K&R style for opening and control statement.
Braces must be always apply to control statements even if the control block only has a single statement.

- Opening statements - brace under function name
- control statement - brace same line as control statement
- tab = 4 spaces.

Array initialization - array items on separate lines.

Feature Specific Support

- Each component must include a configuration file for enabling or disabling a specific RDK feature.
- Every feature must be capable of enabled or disabled using a configuration.

Naming Conventions

- Folder names in lower case separated by underscores where appropriate to communicate meaning.
- Class names in UpperCamelCase.
- Function, variables and constant names in lowerCamelCase.

Documentation Conventions

- Code documentation must comply with Drupal Documentation Standards.which is based on the Doxygen documentation standard.