Coding Standard for STM32-SDR

The following is the suggested coding style for new code being added to the STM32-SDR project.

It is always best to write simple, clear code that is easy to understand, debug and maintain.

Use Meaningful Names

All identifiers must have meaningful, human-readable, English names. Avoid cryptic abbreviations such as "dspl()", "cntStd()", or "stdRegYYYYMMDD".

Instead use:

```
void display();
int countStudents();
char dateStudentRegistered[30];
```

Exception 1: loop variables may be i, j or k:

```
for (int i = 0; i < 10; i++) {
   ...
}
```

Exception 2: variables with very limited scope (<20 lines) may be shortened if the purpose of the variable is clear.

```
void swapMoney(float *pMoney1, float *pMoney2)
{
   float tmp = *pMoney1;
   *pMoney1 = *pMoney2;
   *pMoney2 = tmp;
}
```

Naming Conventions

Constants must be all upper case, with multiple words separated by '_':

```
const int DAYS_PER_WEEK = 7;
```

"Public" functions exported from a module in a .h file should start with the module name, such as:

```
double LCD_ClearScreen();
_Bool Keyboard_IsKeyboardConnected();
```

"Private" functions which are local to a module (not in the .h file) should be made static (to prevent other code from calling them and to avoid linking errors). These functions should start with a lower-case letter and use camelCase. Functions should be named in terms of an action:

```
double calculateTax();
_Bool verifyInput();
```

Global constants are acceptable and should be placed in the most restrictive scope possible. For example, if it is used in only one .c file, then define the constant inside that file. If a constant is needed in multiple files, place it in a .h file (and make it static).

Global variables must be avoided as much as possible. When *absolutely* required, their names should be prefixed with g_:

```
int g_bad;
double g_doNotUseGlobals;
```

Module variables should be placed in a .c file and not exported to other modules (don't put them in the .h file!). Make these module variables static and their names should be prefixed with s_:

```
int s_isKeyboardConnected;
double s_countErrors;
```

Boolean (_Bool) variables should be named so that they make sense in an if statement:

```
if (isOpen) {
    ...
}
while (!isEndOfFile && hasMoreData()) {
    ...
}
```

Pointers should be prefixed with 'p' in order to remind the programmer that it is a pointer.

```
int *pNumStudents;
```

Constants must be all upper case, with words separated by '_':

```
const int MONTHS_PER_YEAR = 12
#define MONTHS_PER_YEAR 12
```

Use named constants instead of literal numbers (magic numbers). It is often acceptable to use 0 and 1; however, it must be clear what they mean:

```
// OK:
int i = 0;
i = i + 1;

// Bad: What are 0 and 1 for?!?
someFunction(x, 0, 1);

// Terrible code:
int a = 215;
int b = a * 5 % 502;
```

Indentation and Braces {...}

There's **always** time for **perfect indentation**.

Tab size is 4; indentation size is 4. Do not replace tabs with spaces. Use tabs at the start of a line to indent code to the current level of indentation. Use spaces to indent code beyond the current level of indentation. This format should make the code readable in all editors, but you will have to configure your editor to maintain this formatting.

```
if (j < 10) {
→   counter = getStudentCount(lowIndex,
→   ·····················highIndex);
→   if (x == 0) {
→   →   if (y != 0) {
→   →   →   x = y;·····// Insightful comment here
→   →   }
→   }
}
```

Braces follow the K&R style. Opening brace is at the end of the enclosing statement; closing brace is on its own line, lined up with the start of the opening enclosing statement. Statements inside the block are indented one tab.

```
for (int i = 0; i < 10; i++) {
→   ...
}

while (i > 0) {
→   ...
}

do {
→   ...
} while (x > 1);

if (y > 500) {
→   ...
} else if (y == 0) {
→   ...
} else {
→   ...
}
```

Exception 1 (as per K&R): Functions have their opening braces on their own line:

```
void foo()
{
→   ...
}
```

"Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are _right_ and (b) K&R are right.", Linux kernel coding style.

Exception 2: *If* statements with multi-line conditions have the starting brace aligned on the left to make it easier to spot the block in the *if* statement.

```
if (someBigBooleanExpression
····&& !someOtherExpression)
{
→   ...
}
```

All *if* statements and loops should include braces around their statements, even if there is only one statement in the body:

```
if (a < 1) {
    a = 1;
} else {
    a *= 2;
}
while (count > 0) {
    count--;
}
```

Statements and Spacing

Declare each variable in its own definition, rather than together ("int i, j"). This is because pointer declarations can be confusing:
"int* p1, p2;" makes a pointer and an int.

```
int *p1;
int p2;
```

Each statement should be on its own line:

```
// Good:
i = j + k;
l = m * 2;

// Bad (what are you hiding?):
if (i == j) l = m * 2;
    printf("Can ya read this?\n");
```

All binary (2 argument) operators (arithmetic, bitwise and assignment) and ternary conditionals (?:) must be surrounded by one space. Commas must have one space after them and none before. Unary operators (!, *, &, - (ex: -1), + (ex: +1), ++, --) have no additional space on either side of the operator.

```
i = 2 + (j * 2) + -1 + k++;
if (i == 0 || j < 0 || !k) {
    arr[i] = i;
}
```

Add extra brackets in complex expressions, even if operator precedence will do what you want. The extra brackets increase readability and reduce errors.

```
if ((!isReady && isBooting)
    || (x > 10)
    || (y == 0 && z < (x + 1)))
{
    ...
}
```

However, it is often better to simplify complex expressions by breaking them into multiple sub-expressions that are easier to understand and maintain:

```
_Bool isFinishedBooting = (isReady || !isBooting);
_Bool hasTimedOut = (x > 10);
_Bool isOldFirmware = (y == 0 && z < (x + 1));
if (!isFinishedBooting
    || hasTimedOut
    || isOldFirmware)
{
    ...
}
```

Comments

Comments which are on one line should use the // style. Comments which are on a couple lines may use either the //, or /* ... */ style. Comments which are many lines long should use /* ... */.

Each file must have a descriptive comment describing the general purpose of the module, and the standard GPL license (or the license the code is released under). Recommended format is shown below:

```
/*
 * Code for SSB operations screen
 *
 * STM32-SDR: A software defined HAM radio embedded system.
 * Copyright (C) 2013, STM32-SDR Group
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * 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.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
 */
 ...
```

Each function listed in a .h file may have a comment describing what it does. However, the name of the function, combined with an understanding of the module, should be enough to tell the user what it does.

Comments should almost always be the line before what they describe and be placed at the same level of indentation as the code. Only very short comments should appear in-line with the code:

```
// Display final confirmation message box.
callSomeFunction(
      0,            // Parent.
      "My App",      // Title
      "test");       // Message
```

Files

Header files have the extension .h; source files have the extension .c. Treat file names as being case sensitive: if the file is named Car.h, use #include "Car.h", not #include "car.h".

All header files must use include guards of the form:

```
#ifndef FILENAME_H
#define FILENAME_H
...
#endif
```

Header files must be self-contained. For example, if it uses uint8_t, then "#include <stdint.h>" in the .h file. The client code must be able to include the .h file without having to first include other .h files.

Place all #include directives at the top of the file, instead of distributing them throughout your code.

Function prototypes must include full parameter names instead of (the valid, but poor style) "void printResult(int, char*);". The parameter names in the prototype must match the function definition (for style, not for compiler).

```
void printResult(int numStudents, float avgShoeSize);
```

Order your files so that the most general function is at the top, and the sub-routines are below it. For example, your main file should have main() at the top, and then the functions which it calls below it. This will require prototypes but allows the reader to read your code top-down.

Other

Either post-increment or pre-increment may be used on its own:

```
i++;
++j;
```

Avoid using goto. If possible, design your loops to not require the use of "break" or "continue".

All switch statements should include a "default" label. If the "default" case seems impossible, place an "assert(false);" in it. Comment any intentional fall-through's in switch statements:

```
switch(buttonChoice) {
case YES:
   // Fall through
case OK:
   printf("It's all good.\n");
   break;
case CANCEL:
   printf("It's over!\n");
   break;
default:
   assert(false);
}
```

Use plenty of assertions. Any time you can use an assertion to check that some condition is true which "must" be true, you can catch a bug early in development. It is especially useful to verify function pre-conditions for input arguments or the object's state.

Never let an assert have a side effect such as "assert(i++ > 1);". This may do what you expect during debugging, but when you build a release version, the asserts get compiled out (they are macros). Therefore, the i++ won't happen in the release build.

When changing the code, change the version number #define named VERSION_STRING in code/main/src/main.c.

References:

1. Kernighan and Ritchie's "The C Programming Language", 1988.
2. Linux kernel coding style.