./

Learning Report – Embedded C - Hardware + Programming + Testing

Course Code: <CODE>





**Document History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver.Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **Approved By** | **Remarks/Revision Details** |
| 1. | 29/09/2020 | DEEKSHA P |  | Dr. Vivek K and Bhargav N | Linker scripts, Semi hosting technique, SPI interface |
| 2. | 30/09/2020 | DEEKSHA P |  | Dr. Vivek K and Bhargav N | Types of debugging techniques |
| 3. | 5/10/2020 | DEEKSHA P |  | Dr. Vivek K and Bhargav N | Driver development files and MISRA Standards |
|  |  |  |  |  |  |

Table of Contents

[ACTIVITY 1 - LINKER SCRIPTS 5](#_Toc52868309)

[ACTIVITY 2 - SEMI HOSTING 7](#_Toc52868310)

[ACTIVITY 3 – SPI 8](#_Toc52868311)

[ACTIVITY 4 - DEBUGGING TECHNIQUES 9](#_Toc52868312)

[Activity 5 - MCU Specific Files (Driver Development) 14](#_Toc52868313)

[Activity 6 - Code quality- MISRA C standards 16](#_Toc52868314)

[References 19](#_Toc52868315)

**List of Figures**

[Figure 1: Semi hosting technique code and result 8](#_Toc52869803)

[Figure 2: main.c for SPI interface 9](#_Toc52869804)

[Figure 3: SPE bit set in SPI register 9](#_Toc52869805)

[Figure 4: Serial wire viewer debugger 10](#_Toc52869806)

[Figure 5: Stepping Technique 12](#_Toc52869807)

[Figure 6: Debug using breakpoint 14](#_Toc52869808)

[Figure 7: Variable Window 15](#_Toc52869809)

[Figure 8: Disassembly Window 16](#_Toc52869810)

# ACTIVITY 1 - LINKER SCRIPTS

**1.1 Code Snippet**

File : stm32\_flash.ld

\*\*

\*\* Abstract : Linker script for STM32F407VG Device with

\*\* 1024KByte FLASH, 128KByte RAM

\*\*

\*\* Set heap size, stack size and stack location according

\*\* to application requirements.

\*\*

\*\* Set memory bank area and size if external memory is used.

\*\*

\*\* Target : STMicroelectronics STM32

\*\*

\*\* Distribution: The file is distributed "as is", without any warranty

\*\* of any kind.

\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

**/\* Entry Point \*/**

|  |  |
| --- | --- |
| ENTRY(Reset\_Handler) | |
|  | |
| **/\*Memory listing\*/** | |  | |
|  | | MEMORY | |
|  | { | |
|  | FLASH(rx) : ORIGIN = 0X08000000, LENGTH = 1024K; | |
|  | SRAM1(rwx): ORIGIN = 0X20000000, LENGTH = 116K; | |
|  | }  **/\*Section creation\*/** | |
|  |  | |
|  |  | |
|  | SECTIONS | |
|  | { | |
|  |  | |
|  |  | |
|  | .text : | |
|  | { | |
|  | \*(.isr\_vector) | |
|  | \*(.text) |
|  | \*(.rodata) |
|  | } >FLASH AT> FLASH |
|  |  |
|  | .data : |
|  | { |
|  | \_sdata= . ; |
|  | \*(.data) \_edata= .; |
|  | } >SRAM AT >FLASH |
|  |  |
|  | .bss : |
|  | { |
|  | \_sbss = .; |
|  | \*(.bss) .ebss= .; |
|  | } >SRAM |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |

# ACTIVITY 2 - SEMI HOSTING

Semi hosting technique is one of the debugging techniques where we can use printf statements to print the values on the same console which we use to write command.

**2.1 Code Snippet & Output**

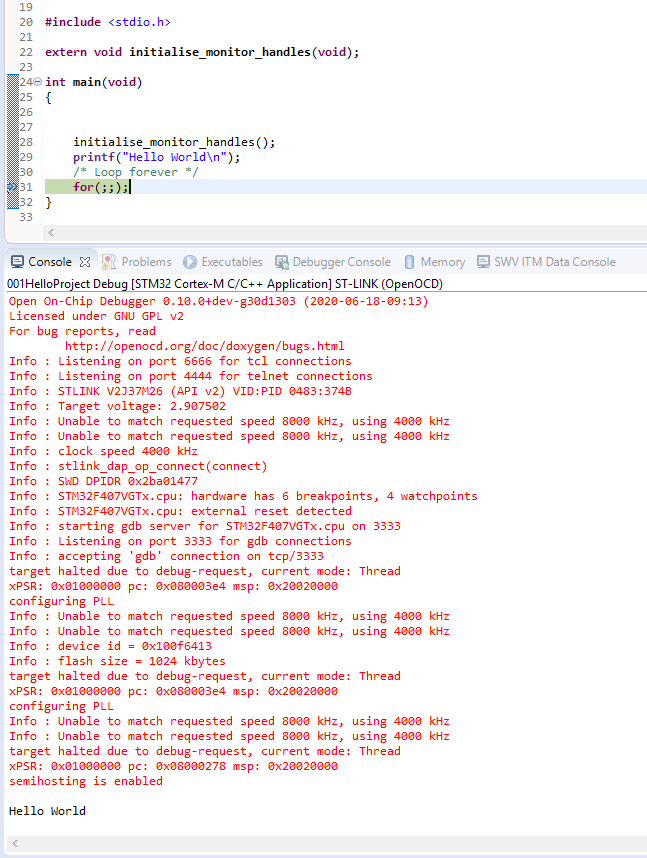


Figure 1: Semi hosting technique code and result

# ACTIVITY 3 – SPI

Enable the SPE bit position of SPI control register 1 (SPI\_CR1) by writing embedded C code.

**3.1 Code Snippet**

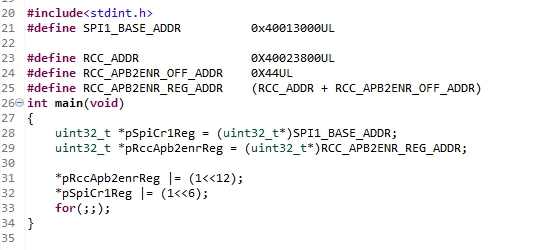
****

Figure 2: main.c for SPI interface

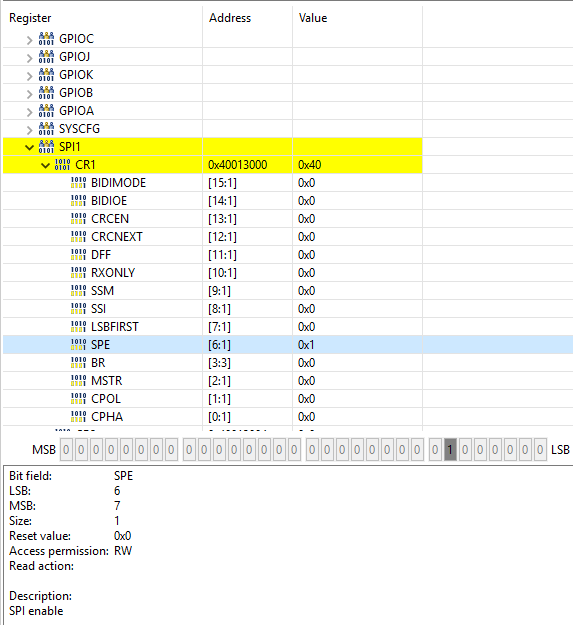
****

Figure 3: SPE bit set in SPI register

# ACTIVITY 4 - DEBUGGING TECHNIQUES

* Serial wire viewer and data tracing (printf style debugging)
* Single stepping, stepping over and stepping out.
* Breakpoints (inserting, deleting and skipping breakpoints).
* Disassembly
* Call stack
* Expression and variable window
* Memory browser
* Data watch points
  1. **Serial wire viewer and data tracing (printf style debugging)**

Serial Wire Viewer is a real-time trace technology that uses the Serial Wire Debugger (SWD) port and the Serial Wire Output (SWO) pin. Serial Wire Viewer provides advanced system analysis and real-time tracing without the need to halt the processor to extract certain types of debug information.

Serial Wire Viewer (SWV) can provide the following types of target information:

* Periodic samples of program counter (PC) values
* Event notification on memory accesses (such as reading/writing C variables)
* Event notification on exception entry and exit
* Event counters
* Timestamp and CPU cycle information

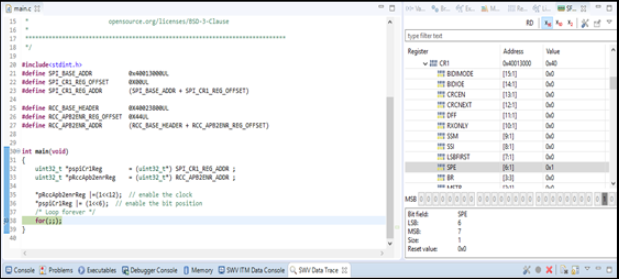


Figure 4: Serial wire viewer debugger

* 1. **Breakpoints and single-stepping**

Create a project in Eclipse:

This is a short program that uses a helper method to calculate the total cost for an order of donuts and coffees. See the javadoc for an explanation. There are couple of bugs. The errors may or may not be obvious to you, but let us illustrate how a debugger might be used to find them quickly. First, run the program and see that the outputs are incorrect. (52 donuts is four dozen plus four singles, which should be a total of 4 \* 8.00 plus 4 \* .75, or 35.00, and we should get 8 free coffees, so the two additional coffees are 3.00, for a grand total of 38.00.)

**Setting a breakpoint**

1. Find the first line of the helper method donutHelper() and double-click in the left margin. You should see a small blue circle appear to the left of the line.
2. Now, start the program again, but start it in "Debug mode". That means that instead of using the "Play" button to run it, use the button with the little beetle-like icon. Alternatively, right-click on the project and choose "Debug As... Java Application".
3. A dialog comes up asking you whether you want to switch to "Debug perspective". Click Yes.

You'll see an editor window with your code, along with some other unfamiliar panes.

* + The pane at the top left, labeled "Debug" is the *call stack*. At the top of the stack you can see donutHelper, the method currently being executed, and just below that you see main, the method that called it.
  + The pane at the top right has two tabs, labeled "Variables" and "Breakpoints". The Variables tab lists all the local variables that are currently in scope. You see two variables, namely the parameters donuts and coffees. There are other local variables defined later in this method, but at this point in the code, they haven't been defined yet.

**4.3 Single-stepping with Step Over**

1. At the top of the Debug pane you'll see some buttons with funny-looking arrows. If you hover your mouse over them, you'll see that they are called "Step Into", "Step Over", and "Step Return" and that they are also available using the F5, F6, and F7 keys, respectively.
2. Press the Step Over button (or the F6 key) once. This causes the current line of code to be executed. Notice in the Variables pane, the variable dozens is now defined. The green highlight in the editor pane is normally the line of code that is about to be executed.
3. Use Step Over or F6 to step through the next line. Notice that now singles is defined too. At this point you can identify the first error in the program. Look at the values of the two incoming parameters, donuts and coffees. Weren't we trying to test this for 52 donuts and 10 coffees? We're calling it with the arguments mixed up. No wonder it's giving us incorrect values!
4. Continue to step through the remaining lines of the method. When it reaches the end, you'll be back in the main method. Notice that the method donutHelper is no longer on the call stack.
5. Press the square red button to terminate the program. Let's fix the error we found: in the main method, switch the order of the arguments to donutHelper. Note you can stay in the Debug perspective to edit your code and start it again. Remember to start it in Debug mode.

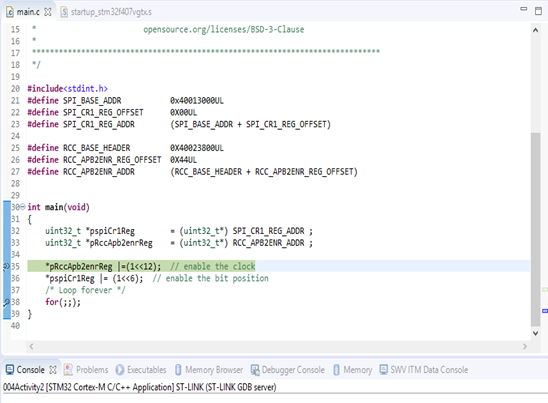


Figure 5: Stepping Technique

**Managing breakpoints**

1. After fixing the code, let's set a breakpoint in main this time. Double-click in the left margin by the first line of main. Now, let's also disable the breakpoint we set before. Click on the Breakpoints tab, next to Variables. You'll probably see two breakpoints listed. Un-check the breakpoint in donutHelper to disable it - the debugger remembers where the breakpoint is, but won't stop execution there.
   * Eclipse will list all the breakpoints you've set in all your projects, not just the one that's open. To clean things up, you can *remove* a breakpoint by right-clicking on it and choosing Remove.
   * If you want to see the code where the breakpoint is set, just double-click on the listed breakpoint. (Use the back-arrow in the main toolbar to return.)
2. After you've examined the breakpoints, click on the Variables tab again to make sure it's visible.

**Step Into**

1. Now, run the program again in debug mode. It should stop on the first line of main. Press Step Over or F6 now and see what happens. Notice it executes the current line of code completely, *including the method call*.
2. Step through the next line, which includes the call to System.out.println. You should see the correct output this time.
3. For the next line, that is, the second call to donutHelper, let us try something different. In this case, we want to step *into* the method call so we can see what it's doing. Use the Step Into button or F5.
4. Step through the first few lines. (When you see the coffees variable updated to a negative value, you should have a clue about what's going wrong. Note that variables that have *changed* are highlighted in yellow.)

**Step Return**

1. Now, suppose we are done examining a method and want to just return to the point where it was called. That's what Step Return is for. Click the Step Return button or F7 and you'll be back on the line containing the call to donutHelper.
2. Press Step Over or F6 to finish executing the current line. The green highlight will be on the println statement.
3. Now we're going to do something odd that frequently happens by accident. Press Step Into again. Depending on exactly how your JDK is set up, you will either see a message saying "the source code is not available", or you might see the actual source code from some class in the Java libraries. Either way, *we don't want to debug the Java libraries!* Use your Step Return button to get back to your own code.
4. Before you leave this page, fix the bug in the donutHelper method, and make sure you get the right output.

**Switching perspectives**

For regular code editing, you probably don't want your workspace cluttered with the extra panes in the Debug perspective. You can easily switch between perspectives.

* Under the main Window menu, go to Open Perspective and select "Java".
* When in Java perspective, you can use the same menu to switch to "Debug"
* There are usually some shortcut buttons on the upper right side of the toolbar as well.

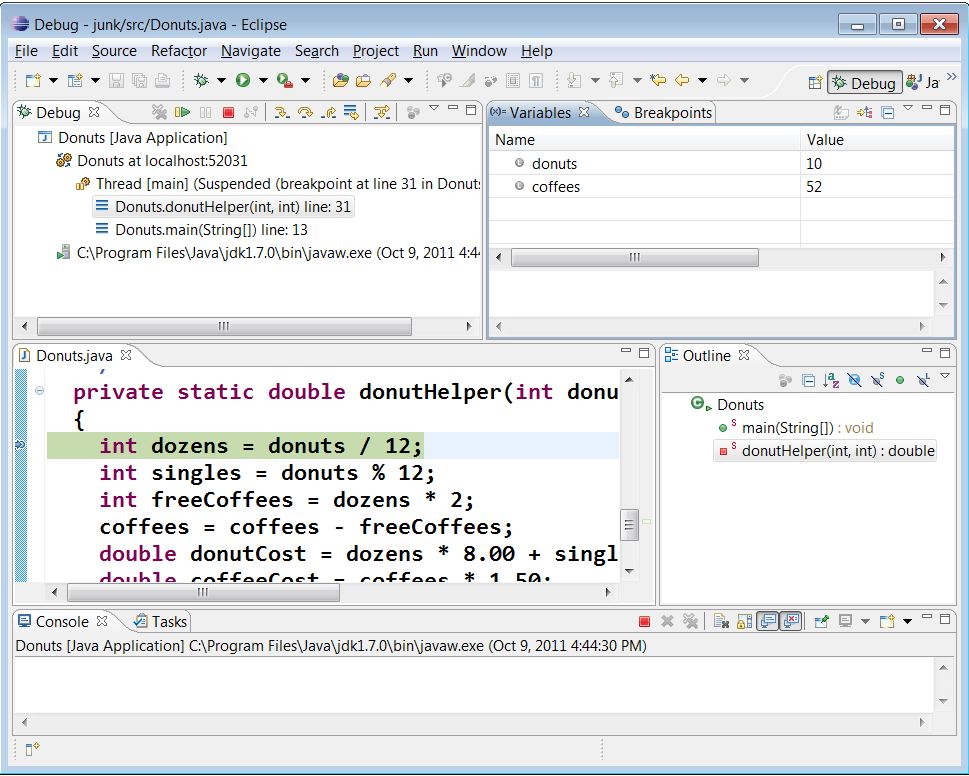


Figure 6: Debug using breakpoint

**4.4 Call Stack**

The call stack is a list of all the active functions that have been called to get to the current point of

execution. The call stack includes an entry for each function called, as well as which line of code will be

returned to when the function returns.

**4.5 Expression and variable window**

To watch a variable, right-click a variable in the editor or from the variables view and select Watch or Add Watch Expression. The Expressions view will open and the variable will be added to it.

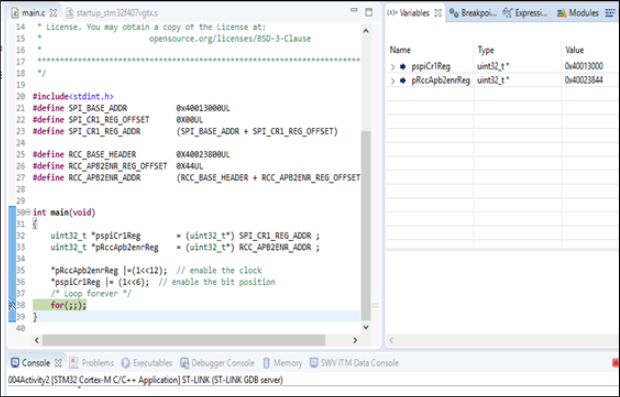


Figure 7: Variable Window

The variables information will be updated as the debugging process continues.

* Memory browser**:**To make browsing the internet faster, almost all browsers store copies of webpages you've visited and open destinations from memory. From time to time, this can cause an error as the browser tries to access a page that has been changed. Clearing your Cache/browser memory will reset the browser so it loads with the most current information.

* Watch points**:** The STM32 have four Watch points. Watch points can be thought of as conditional breakpoints. The Logic Analyzer uses the same comparators as Watch points in its operations and they must be shared. This means in µVision you must have two variables free in the Logic Analyzer to use Watch points. Watch points are also referred to as Access Breakpoints

**4.6 Disassembly**

The Debug Disassembly Window gives the user access to debugging in assembly language for project written in C. The Debug Disassembly Window allows the user to perform all the normal debug operations including single stepping and setting breakpoints on the individual assembly instructions generated from C code.

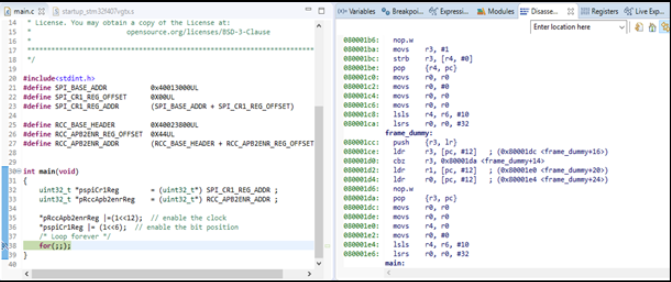


Figure 8: Disassembly Window

# Activity 5 - MCU Specific Files (Driver Development)

MCU Specific Header file

File name: STM32F407xx.h

Link: [link1](https://lnttsgroup.sharepoint.com/sites/GEA/Global%20Engineering%20Academy/GEA%20Insights/Genesis/Shared%20Documents/Forms/AllItems.aspx?originalPath=aHR0cHM6Ly9sbnR0c2dyb3VwLnNoYXJlcG9pbnQuY29tLzpmOi9zL0dFQS9HbG9iYWwlMjBFbmdpbmVlcmluZyUyMEFjYWRlbXkvR0VBJTIwSW5zaWdodHMvR2VuZXNpcy9FaW9RZnd0YkxmSkNqX2ZQdk8wbUVsVUJYaVMxLXc2X2FWT2tnd1ZwX2JtanVBP3J0aW1lPURJUk5GYVZwMkVn&id=%2Fsites%2FGEA%2FGlobal%20Engineering%20Academy%2FGEA%20Insights%2FGenesis%2FShared%20Documents%2FSubmission%2FMYSORE%2F2009MYSEMB%2FFoundation%2FApplied%20SDLC%20with%20Software%20Testing%2F99002690%2FEmbedded%20C%20%2D%20Hardware%20%2B%20%20Programming%20%2B%20Testing%2F99002664%2FMCU%20Specific%20Files%2FDriver%2Finc%2FSTM32F407xx%2Eh&parent=%2Fsites%2FGEA%2FGlobal%20Engineering%20Academy%2FGEA%20Insights%2FGenesis%2FShared%20Documents%2FSubmission%2FMYSORE%2F2009MYSEMB%2FFoundation%2FApplied%20SDLC%20with%20Software%20Testing%2F99002690%2FEmbedded%20C%20%2D%20Hardware%20%2B%20%20Programming%20%2B%20Testing%2F99002664%2FMCU%20Specific%20Files%2FDriver%2Finc)

GPIO header file

File name: stm32f407xx\_GPIO\_driver.h

Link: [link2](https://lnttsgroup.sharepoint.com/sites/GEA/Global%20Engineering%20Academy/GEA%20Insights/Genesis/Shared%20Documents/Forms/AllItems.aspx?originalPath=aHR0cHM6Ly9sbnR0c2dyb3VwLnNoYXJlcG9pbnQuY29tLzpmOi9zL0dFQS9HbG9iYWwlMjBFbmdpbmVlcmluZyUyMEFjYWRlbXkvR0VBJTIwSW5zaWdodHMvR2VuZXNpcy9FaW9RZnd0YkxmSkNqX2ZQdk8wbUVsVUJYaVMxLXc2X2FWT2tnd1ZwX2JtanVBP3J0aW1lPURJUk5GYVZwMkVn&id=%2Fsites%2FGEA%2FGlobal%20Engineering%20Academy%2FGEA%20Insights%2FGenesis%2FShared%20Documents%2FSubmission%2FMYSORE%2F2009MYSEMB%2FFoundation%2FApplied%20SDLC%20with%20Software%20Testing%2F99002690%2FEmbedded%20C%20%2D%20Hardware%20%2B%20%20Programming%20%2B%20Testing%2F99002664%2FMCU%20Specific%20Files%2FDriver%2Finc%2Fstm32f407xx%5FGPIO%5Fdriver%2Eh&parent=%2Fsites%2FGEA%2FGlobal%20Engineering%20Academy%2FGEA%20Insights%2FGenesis%2FShared%20Documents%2FSubmission%2FMYSORE%2F2009MYSEMB%2FFoundation%2FApplied%20SDLC%20with%20Software%20Testing%2F99002690%2FEmbedded%20C%20%2D%20Hardware%20%2B%20%20Programming%20%2B%20Testing%2F99002664%2FMCU%20Specific%20Files%2FDriver%2Finc)

Source file

File name: stm32f407xx\_GPIO\_driver.c

Link: [link3](https://lnttsgroup.sharepoint.com/sites/GEA/Global%20Engineering%20Academy/GEA%20Insights/Genesis/Shared%20Documents/Forms/AllItems.aspx?originalPath=aHR0cHM6Ly9sbnR0c2dyb3VwLnNoYXJlcG9pbnQuY29tLzpmOi9zL0dFQS9HbG9iYWwlMjBFbmdpbmVlcmluZyUyMEFjYWRlbXkvR0VBJTIwSW5zaWdodHMvR2VuZXNpcy9FaW9RZnd0YkxmSkNqX2ZQdk8wbUVsVUJYaVMxLXc2X2FWT2tnd1ZwX2JtanVBP3J0aW1lPURJUk5GYVZwMkVn&id=%2Fsites%2FGEA%2FGlobal%20Engineering%20Academy%2FGEA%20Insights%2FGenesis%2FShared%20Documents%2FSubmission%2FMYSORE%2F2009MYSEMB%2FFoundation%2FApplied%20SDLC%20with%20Software%20Testing%2F99002690%2FEmbedded%20C%20%2D%20Hardware%20%2B%20%20Programming%20%2B%20Testing%2F99002664%2FMCU%20Specific%20Files%2FDriver%2Fsrc%2Fstm32f407xx%5FGPIO%5Fdriver%2Ec&parent=%2Fsites%2FGEA%2FGlobal%20Engineering%20Academy%2FGEA%20Insights%2FGenesis%2FShared%20Documents%2FSubmission%2FMYSORE%2F2009MYSEMB%2FFoundation%2FApplied%20SDLC%20with%20Software%20Testing%2F99002690%2FEmbedded%20C%20%2D%20Hardware%20%2B%20%20Programming%20%2B%20Testing%2F99002664%2FMCU%20Specific%20Files%2FDriver%2Fsrc)

# Activity 6 - Code quality- MISRA C standards

Originally, MISRA (Motor Industry Software Reliability Association) was founded for the purpose of designing a set of guidelines for development of software for microcontrollers used in road vehicles. Since then it has been adopted in every reliability and safety critical field including the automotive industry, medical devices, aerospace and defense, and so on.

Writing a safe program in C not only means that the program runs as the programmer expected it to. It must also run correctly when ported to a different environment. More importantly, when other people read the source code the meaning must be crystal clear.

The rules can be grouped into the following categories:

* Mandatory
* Required – Deviations from the rules are permitted (but recommended to be documented)
* Advisory – Not obligatory

Certain key aspects:

* Assignment operators shall not be used in expressions which return Boolean value;
* All functions that are not void should return a value;
* A loop counter shall not have essentially floating type;
* A function should have a single point of exit at the end;

Few of the rules are listed below:

**Rule 1 (required):** All code shall conform to ISO 9899 standard C, with no extensions permitted.

When I first read this, I laughed out loud. Since the C standard was not originally written with embedded systems in mind, it is impossible to write a non-trivial embedded program without resorting to a variety of compiler extensions. For example, ISO C provides no way to specify that a function is an interrupt service routine. (MISRA frowns upon assembly language, so you can't use that as a way to skirt the issues.)

Fortunately, in reading the notes associated with rule 1, one finds the following comment: It is recognized that it may be necessary to raise deviations (as described in section 5.3.2) to permit certain language extensions, for example to support hardware specific features.

The deviations mentioned in the comment are, arguably, the strongest feature of MISRA C. In putting together the rules, the authors recognized that they could not possibly foresee all the circumstances associated with all embedded systems. Thus, they put in place a mechanism by which an organization can formally deviate from a given rule. Note that this does not mean that the program is free to violate the rules on a whim. Rather it means that all violations must be carefully considered, justified, and documented. MISRA also recommends that deviations be localized as much as possible-such as, all I/O operations be performed in one file. Notwithstanding MISRA C, this is good programming practice anyway.

**Rule 20 (required):** All object and function identifiers shall be declared before use.

Any programmer who has a problem with this rule probably needs to find a new line of work. Some of the rules are more stylistic in their intent and, as such, are more likely to offend individual sensibilities. This is particularly true of the advisory rules, such as:

**Rule 49 (advisory):** Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. This rule goes against my own preferred style of coding. However, I see where the authors are going and wouldn't exactly be compromising deeply held beliefs by conforming to their advice.

While rule 49 isn't going to change much in the way most people write code, rule 104 could have a bigger impact.

**Rule 104 (required):** Non-constant pointers to functions shall not be used.

The use of pointers to functions is a favored technique of many experienced embedded programmers. The bottom line: regardless of your experience level, if you choose to go down the MISRA C path, expect to have

to change the way you code in both big and small ways.

**Rule 13 (advisory):** This rule states that an application should not directly use basic types such as char, int, float etc. Instead specific-length equivalents should be typed fed for the specific compiler, and these type names should be used in the code. The reason is that different compilers could use different underlying representation for the basic types. The most common case is the type int that could be seen as 16 bit wide by one compiler and 32 bit wide by another. My personal experience with specific-length types is that it is that I take greater care when selecting the appropriate size and signees. Instead of just picking an int I now more carefully select, say, uint16\_t whenever I need to represent something that never will be negative. One effect of such code is that it is easier to read and understand as the reader doesn't have to consider a situation where expressions could be negative. The C standard, as of the 1999 version, contains a header file stdint.h that provides type names on the form int8\_t and uint32\_t. IAR Embedded Workbench supports this header file in the DLib standard library.

**Rule 23 (advisory):** All declarations at file scope should be static where possible As you probably know, by declaring a variable or a function static it cannot be seen or used directly by other modules in the application.

This rule:

• Prevents you from unintentionally exposing internal help functions and file-local variables

• Forces you to really design the interface of new modules

• Makes the interface more clear, something that is rely important as applications grow older and the person who wrote the code may not be around any more.

When using IAR Embedded Workbench this rule will be checked by the linker since it has got access to the entire application.

**Rule 33 (required):** The right hand side of a "&&" or "||" operator shall not contain side effects There is nothing in the C language that prevents you from writing code that looks like the following: if ((x == y) || (\*p++== z)) { /\*do something \*/ } In this example, the right hand side of the || operator is only evaluated (and its side-effects executed) if the expression on the left-hand side is false—that is, if x and y are not equal. In this example, the side effect is to increase the pointer p.

However, even if this behavior is specified by the standard it is still easy to get it wrong when writing the code. And even if you manage to get it right, everyone that will ever read or maintain the code must also understand the rules and your intentions. As a side note, the code above could be rewritten into the much more straightforward sequence instead.

int doSomething = 0;

if (x == y)

{

doSomething = 1;

}

else if (\*p++ == z)

{

doSomething = 1;

} i

f (doSomething)

{

/\* do something \*/

}

**Rule 59 (required):** The statement forming the body of an "if", "else if", "else", "while", "do ... while", or "for" statement shall always be enclosed in braces Basically, this says that from now you must clean up your act, you can't write sloppy things like the else clause in following example.

if (x == 0)

{

y = 10;

z = 0;

}

else

y = 20;

The idea of this rule is to avoid a classical mistake. In the example below the line z = 1; was added. It looks as though it's part of the else clause but it's not! In fact, it is placed after the if statement altogether, which means that the assignment will always take place.

If the original else clause would have contained the braces from the beginning this problem would never have

occurred.

if (x == 0)

{

y = 10;

z = 0;

}

else

y = 20;

z = 1;

# References

1. <https://www.youtube.com/watch?v=B7oKdUvRhQQ&list=PLERTijJOmYrDiiWd10iRHY0VRHdJwUH4g&index=4>
2. <https://www.youtube.com/watch?v=5aafG5mjZ_Y&list=PLERTijJOmYrDiiWd10iRHY0VRHdJwUH4g&index=5>
3. <http://web.cs.iastate.edu/~smkautz/cs227s13/labs/lab6/page04.html>
4. <https://www.keil.com/appnotes/files/discovery_m4_lab.pdf>
5. <https://www.digikey.com/en/maker/projects/getting-started-with-stm32-introductiontostm32cubeide/6a6c60a670c447abb90fd0fd78008697>
6. <https://signuphelp.zendesk.com/hc/en-us/articles/202715400-How-do-you-Clear-your-Cache-or-yourBrowserMemory>
7. <https://www.embedded.com/tutorial-achieve-reliable-embedded-code-with-misra-c/>