J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_12

12. Interfaces, Contracts, and Assertions

Jacob Beningo[1]
(1) Linden, MI, USA

An essential feature of every object, module, package, framework, and so forth is that it has a well-defined and easy-to-use interface. Perhaps most important is that those interfaces create a binding contract between the interface and the caller. If the caller were to break that contract, a bug would be injected into the code. Without a software mechanism to verify that the contract is being met, the bugs will not just be injected but also run rampant and perhaps become challenging to find and detect.

This chapter will look at developing contracts between interfaces and the calling code. Creating the contract will allow us to introduce and explore the use of assertions available in both C and C++. Assertions in real-time systems can potentially be dangerous, so once we understand what they are and how we can use them to create an interface contract, we will also explore real-time assertions.

Note I wrote extensively in my book *Reusable Firmware Development* about how to design interfaces and APIs. For that reason, I'll briefly introduce interfaces and dive into contracts and assertions. For additional details on interfaces, see *Reusable Firmware Development*.

Interface Design

An interface is how an object, person, or thing interacts with another item. For example, developers will often work with several different types of interfaces, such as an application programming interface (API) or a graphical user interface (GUI). These interfaces allow someone to request an action to be performed, and the interface provides a mechanism to re-

quest, perform, and respond to that action.

The interfaces we are most interested in for this chapter are APIs. APIs are the functions and methods we use as developers to expose features in our application code. For example, Listing 12-1 shows an example interface for a digital input/output (DIO) driver. It includes interactions such as initialization, reading, and writing. The driver also has an interface to extend the API's functionality through low-level interactions like reading and writing registers and defining callback functions for managing interrupt service routine behavior.

```
void          Dio_Init(DioConfig_t const * const Config);
DioPinState_t Dio_ChannelRead(DioChannel_t const Channel);
void          Dio_ChannelWrite(DioChannel_t const Channel,
                               DioPinState_t const State);
void          Dio_ChannelToggle(DioChannel_t const Channel);
void          Dio_ChannelModeSet(DioChannel_t const Channel,
                                 DioMode_t const Mode);
void          Dio_ChannelDirectionSet(DioChannel_t Channel,
                                      PinModeEnum_t Mode);
void          Dio_RegisterWrite(uint32_t const Address,
                                TYPE const Value);
TYPE          Dio_RegisterRead(uint32_t const Address);
void          Dio_CallbackRegister(DioCallback_t Function,
                           TYPE (*CallbackFunction)(type));
```

Listing 12-1 An example digital input/output interface used to interact with a microcontroller's general-purpose input/output peripheral. (Note: TYPE is replaced with uint8_t, uint16_t, or uint32_t depending on the microcontroller architecture.)

Interfaces can come in many shapes and sizes and can be found at all layers of a system, such as in the drivers, middleware, operating system, and even the high-level application code. The interface abstracts low-level details the developer doesn't need to know and packages them behind an easy-to-use interface. The best interfaces are designed and implemented to create a binding contract between the interface and the caller. Personally, I prefer to limit the interface for any module, package, or object to around a dozen or so methods if possible. I find that doing so makes it easier to remember the interface. (Most humans can only retain information in chunks of 7–12 items, as we've discussed before. So if you keep any given interface to around 7–12 methods, it's more likely you can commit it to memory and therefore minimize the time needed to review documentation.)

An interesting technique that developers should be interested in but that I often see a lack of support for in many designs and programming languages is Design-by-Contract (DbC).

Design-by-Contract

The idea behind Design-by-Contract programming is that every interface should be clearly defined, precise, and verifiable. This means that every interface should have clearly stated

- Preconditions
- Side effects[1]
- Postconditions
- Invariants[2]

By formally defining these four items, the interface creates a fully specified contract that tells the developer

- What needs to happen before calling the interface for it to operate properly
- What the expected system state will be after calling the interface
- The properties that will be maintained upon entry and exit
- Changes that will be made to the system

Design-by-Contract creates the obligations, the contract, that a developer using the interface must follow to benefit from calling that interface.

Utilizing Design-by-Contract in C Applications

At first glance, Design-by-Contract might seem like it is yet another overweight process to add to an already overloaded development cycle. However, it turns out that using Design-by-Contract can be pretty simple and can improve your code base documentation and decrease the defects in the software, which has the added benefit of improving software quality. Design-by-Contract doesn't have to be super formal; instead, it can be handled entirely in your code documentation. For example, look at the Doxygen function header for a digital input/output (DIO) initialization function shown in Listing **12-2**.

As you can see from the example, the comment block has a few additional lines associated with it that call out the function's preconditions and postconditions. In this case, for Dio_Init, we specify that

- A configuration table that specifies pin function and states needs to be populated.
- The definition for the number of pins on the MCU must be greater than zero.
- The definition for the number of pin ports must be greater than zero.
- The GPIO clock must be enabled.

If these conditions are all met, when a developer calls the Dio_Init function and provides the specified parameters, we can expect that the output from the function will be configured GPIO pins that match the predefined configuration table.

```
/*************************************************************
* Function : Dio_Init()
*//**
* \b Description:
*
* This function is used to initialize the Dio based on the
* configuration table defined in dio_cfg module.
*
* PRE-CONDITION: Configuration table needs to populated
* (sizeof > 0) <br/>
* PRE-CONDITION: NUMBER_OF_CHANNELS_PER_PORT > 0 <br/>
* PRE-CONDITION: NUMBER_OF_PORTS > 0 <br/>
* PRE-CONDITION: The MCU clocks must be configured and
* enabled.
*
* POST-CONDITION: The DIO peripheral is setup with the
* configuration settings.
*
* @param[in]  Config is a pointer to the configuration
* table that contains the initialization for the peripheral.
*
* @return        void
*
* \b Example:
* @code
*    const DioConfig_t *DioConfig = Dio_ConfigGet();
```

```
*    Dio_Init(DioConfig);
* @endcode
*
* @see Dio_Init
* @see Dio_ChannelRead
* @see Dio_ChannelWrite
* @see Dio_ChannelToggle
* @see Dio_ChannelModeSet
* @see Dio_ChannelDirectionSet
* @see Dio_RegisterWrite
* @see Dio_RegisterRead
* @see Dio_CallbackRegister
************************************************************/
```

Listing 12-2 The Doxygen documentation for a function can specify the conditions in the Design-by-Contract for that interface call

Documentation can be an excellent way to create a contract between the interface and the developer. However, it suffers from one critical defect; the contract cannot be verified by executable code. As a result, if a developer doesn't bother to read the documentation or pay close attention to it, they can violate the contract, thus injecting a bug into their source code that they may or may not be aware of. At some point, the bug will rear its ugly head, and the developer will likely need to spend countless hours hunting down their mistake.

Various programming languages deal with Design-by-Contract concepts differently, but for embedded developers working in C/C++, we can take advantage of a built-in language feature known as assertions.

Assertions

The best definition for an assertion that I have come across is

"An *assertion* is a Boolean expression at a specific point in a program that will be true unless there is a bug in the program."[3]

There are three essential points that we need to note about the preceding definition, which include

- An assertion evaluates an expression as either true or false.

- The assertion is an *assumption* of the state of the system at a specific point in the code.
- The assertion is validating a system assumption that, if not true, *reveals a bug* in the code. (It is not an error handler!)

As you can see from the definition, assertions are particularly useful for verifying the contract for an interface, function, or module.

Each programming language that supports assertions does so in a slightly different manner. For example, in C/C++, assertions are implemented as a macro named `assert` that can be found in assert.h. There is quite a bit to know about assertions, which we will cover in this chapter, but before we move on to everything we can do with assertions, let's first look at how they can be used in the context of Design-by-Contract. First, take a moment to review the contract that we specified in the documentation in Listing 12-2.

Assertions can be used within the Dio_Init function to verify that the contract specified in the documentation is met. For example, if you were to write the function stub for Dio_Init and include the assertions, it would look something like Listing 12-3. As you can see, for each precondition, you have an assertion. We could also use assertions to perform the checks on the postcondition and invariants. This is a bit stickier for the digital I/O example because we may have dozens of I/O states that we want to verify, including pin multiplexing. I will leave this up to you to consider and work through some example code on your own.

```
void Dio_Init(DioConfig_t const * const Config)
{
    assert(sizeof(Config) > 0);
    assert(CHANNELS_PER_PORT > 0);
    assert(NUMBER_OF_PORTS > 0);
    assert(Mcu_ClockState(GPIO) == true);
    /* TODO: Define the implementation */
}
```

Listing 12-3 An example contract definition using assertions in C

Defining Assertions

The use of assertions goes well beyond creating an interface contract. Assertions are interesting because they are pieces of code developers add to their applications to verify assumptions and detect bugs directly. When

an assertion is found to be true, there is no bug, and the code continues to execute normally. However, if the assertion is found to be false, the assertion calls to a function that handles the failed assertion.

Each compiler and toolchain tend to implement the assert macro slightly differently. However, the result is the same. The ANSI C standard dictates how assertions must behave, so the differences are probably not of interest if you use an ANSI C–compliant compiler. I still find it interesting to look and see how each vendor implements it, though. For example, Listing 12-4 shows how the STM32CubeIDE toolchain defines the assert macro. Listing 12-5 demonstrates how Microchip implements it in MPLAB X. Again, the result is the same, but how the developer maps what the assert macro does if the result is false will be different.

```
/* required by ANSI standard */
#ifdef NDEBUG
# define assert(__e) ((void)0)
#else
# define assert(__e) ((__e) ? (void)0 : __assert_func \
                      (__FILE__, __LINE__, __ASSERT_FUNC,
#__e))
```

Listing 12-4 The assert macro is defined in the STM32CubeIDE assert.h header

Looking at the definitions, you'll notice a few things. First, the definition for macros changes depending on whether NDEBUG is defined. The controversial idea here is that a developer can use assertions during development with NDEBUG not defined. However, for production, they can define NDEBUG, changing the definition of assert to nothing, which removes it from the resultant binary image. This idea is controversial because you should ship what you test! So if you test your code with assertions enabled, you should ship with them enabled! Feel free to ship with them disabled if you test with them disabled.

```
#ifdef NDEBUG
#define    assert(x) (void)0
#else
#define assert(x) ((void)((x) || (__assert_fail(#x,
__FILE__,\
                                  __LINE__, __func__),0)))
#endif
```

Listing 12-5 The assert macro is defined in the MPLAB X assert.h header

Next, when we are using the full definition for assert, you'll notice that there is an assertion failed function that is called. Again, the developer defines the exact function but is most likely different between compilers. For STM32CubeIDE, the function is `__assert_func`, while for MPLAB X the function is `__assert_fail`.[4] Several key results occur when an assertion fails, which include

- Collecting the filename and line number where the assertion failed
- Printing out a notification to the developer that the assertion failed and where it occurred
- Stopping program execution so that the developer can take a closer look

The assertion tells the developer where the bug was detected and halts the program so that they can review the call path and memory states and determine what exactly went wrong in the program execution. This is far better than waiting for the bug to rear its head in how the system behaves, which may occur instantly or take considerable time.

> **Caution** It is important to note that just stopping program execution could be dangerous in a real-time application driving a motor, solenoid, etc. The developer may need to add extra code to handle these system activities properly, but we will discuss this later in the chapter.

When and Where to Use Assertions

The assert macro is an excellent tool to catch bugs as they occur. That also preserves the call stack and the system in the state it was in when the assertion failed. This helps us to pinpoint the problem much faster, but where does it make sense to use assertions? Let's look at a couple proper and improper uses for assertions.

First, it's important to note that we can use assertions anywhere in our program where we want to test an assumption, meaning assertions could be found just about anywhere within our program. Second, I've found that one of the best uses for assertions is verifying function preconditions and postconditions, as discussed earlier in the chapter. Still, they can also be "sprinkled" throughout the function code.

As a developer writes their drivers and application code, in every function, they analyze what conditions must occur before this function is executed for it to run correctly. They then develop a series of assertions to test those preconditions. The preconditions become part of the documentation and form a contract with the caller on what must be met for every-

thing to go smoothly.

A great example of this is a function that changes the state of a system variable in the application. For example, an embedded system may be broken up into several different system states that are defined using an `enum`. These states would then be passed into a function like `System_StateSet` to change the system's operational state, as shown in Listing 12-6.

```
void System_StateSet(SystemState_t const State)
{
    SystemState = State;
}
```

Listing 12-6 A set function for changing a private variable in a state machine

What happens if only five system states exist, but an application developer passes in a state greater than the maximum state? In this case, the system state is set to some nonexistent state which will cause an unknown behavior in the system. Instead, the software should be written such that a contract exists between the application code and the `System_StateSet` function that the state variable passed into the function will be less than the maximum state. If anything else is passed in, that is a defect, and the developer should be notified. We can do this using assert, and the updated code can be seen in Listing 12-7.

```
void System_StateSet(SystemState_t const State)
{
    assert(State < SYSTEM_STATE_MAX);
    SystemState = State;
}
```

Listing 12-7 A set function for changing a private variable in a state machine with an assertion to check the preconditions

Now you might say that the assertion could be removed and a simple if statement used to check the parameter. This would be acceptable, but that is error handling! In this case, we are constraining the conditions under which the function executes, which means we don't need error handling but a way to detect an improper condition (bug) in the code. This brings us to an important point; **assertions should NOT be used for error handling**! For example, a developer should NOT use an assertion to check that a file exists on their file system, as shown in Listing 12-8. In this case, creating an error handler to create a default version of UserData.cfg makes a lot more sense than signaling the developer that there is a bug in the software. There is not a bug in the software but a file that is missing from the file system; this is a runtime error.

```
FileReader = fopen("UserData.cfg", 'r');
assert(FileReader != NULL);
```

Listing 12-8 An INCORRECT use of assertions is to use them for error handling

Does Assert Make a Difference?

There is a famous paper that was written by Microsoft[5] several years ago that examined how assertion density affected code quality. They found that if developers achieved an assertion density of 2–3%, the quality of the code was much higher, which meant that it had fewer bugs in the code. Now the paper does mention that the assertion density must be a true density which means developers aren't just adding assertions to reach 2–3%. Instead, they are adding assertions where it makes sense and where they are genuinely needed.

Setting Up and Using Assertions

Developers can follow a basic process to set up assertions in their code base. These steps include

1. Include <assert.h> in your module.
2. Define a test assertion using `assert(false);`.
3. Compile your code (this helps the IDE bring in function references).
4. Right-click your assert and select "goto definition."
5. Examine the assertion implementation.
6. Implement the assert_failed function.
7. Compile and verify the test assertion.

I recommend following this process early in the development cycle. Usually, after I create my project, I go through the process before writing even a single line of code. If you set up assertions early, they'll be available to you in the code base, which hopefully means you'll use them and catch your bugs faster.

The preceding process has some pretty obvious steps. For example, steps 1–4 don't require further discussion. We examined how assertions are implemented in two toolchains earlier in the chapter. Let's change things up and look at steps 5–7 using Keil MDK.

Examining the Assert Macro Definition

After walking through steps 1–4, a developer would need to examine how

assertions are implemented. When working with Keil MDK, the definition for assert located in assert.h is slightly different from the definitions we've already explored. The code in Figure 12-1 shows the definitions for assert, starting with the NDEBUG block. There are several important things we should notice.

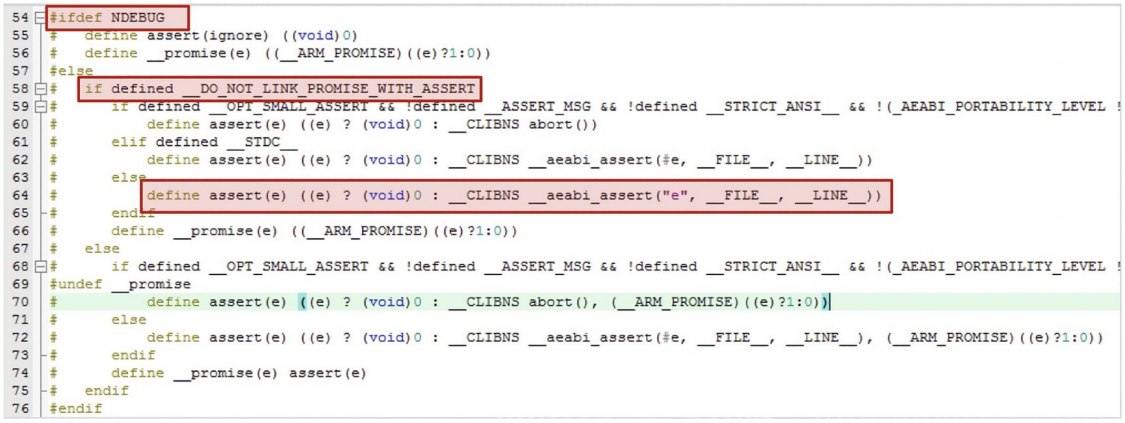

Figure 12-1 The assert.h header file from Keil MDK defines the implementation for the assert macro

First, we can control whether the assert macro is replaced with nothing, basically compiled out of the code base, or we can define a version that will call a function if the assertion fails. If we want to disable our assertions, we must create the symbol NDEBUG. This is usually done through the compiler settings.

Next, we also must make sure that we define `__DO_NOT_LINK_PROMISE_WITH_ASSERT`. Again, this is usually done within the compiler settings symbol table. Finally, at this point, we can then come to the definition that will be used for our assertions:

```
define assert(e) (e ? (void)0 : __CLIBNS __aeabi_assert("e",
\
                                                __FILE__, __LINE__))
```

As you can see, there is a bit more to defining assertions in Keil MDK compared to (GNU Compiler Collection) GCC-based tools like STM32CubeIDE and MPLAB X. Notice that the functions that are called if the assert fails are similar but again named differently.[6] Therefore, to define your assertion function, you must start by reviewing what our toolchain expects. This is important because we will have to define our assert_failed function, and we need to know what to call it so that it is properly linked to the project.

Implementing assert_failed

Once we have found how the assertion is implemented, we need to create the definition for the function. assert.h makes the declaration, but nothing useful will come of it without defining what that function does. There are four things that we need to do, which include

- Copy the declaration and paste the declaration into a source module[7]
- Turn the new declaration into a function definition
- Output something so that the developer knows the assertion failed
- Stop the program from executing

For a developer using Keil MDK, their assertion failed function would look something like the code in Listing 12-9.

```
void __aeabi_assert(const char *expr, const char *file,
                            int line)
{
    Uart_printf(UART1, "Assert failed in %s at line %d\n",
                                                    file,
line);
}
```

Listing 12-9 The implementation for the "assert failed" function in Keil MDK

In Listing 12-9, we copied and pasted the declaration from assert.h and turned it into a function definition. (Usually, you can right-click the function call in the macro, which will take you to the official declaration. You can just copy and paste this instead of stumbling to define it yourself.) The function, when executed, will print out a message through one of the microcontrollers' UARTs to notify the developer that an assertion failed. A typical printout message is to notify the developer whose file the assertion failed in and the line number. This tells the developer exactly where the problem is.

This brings us to an interesting point. You can create very complex-looking assertions that test for multiple conditions within a single assert, but then you'll have to do a lot more work to determine what went wrong. I prefer to keep my assertions simple, checking a single condition within each assertion. There are quite a few advantages to this, such as

- First, it's easier to figure out which condition failed.
- Second, the assertions become clear, concise documentation for how the function should behave.

- Third, maintaining the code is more manageable.

Finally, once we have notified the developer that something has gone wrong, we want to stop program execution similarly. There are several ways to do this. First, and perhaps the method I see the most, is just to use an empty while (true) or for(;;) statement. At this point, the system "stops" executing any new code and just sits in a loop. This is okay to do, but from an IDE perspective, it doesn't show the developer that something went wrong. If my debugger can handle flash breakpoints, I prefer to place a breakpoint in this function, or I'll use the assembly instruction __BKPT to halt the processor. At that point, the IDE will stop and highlight the line of code. Using __BKPT can be seen in Listing **12-10**. (Great, Scott! Yes, there are still places where it makes sense to use assembly language!)

```
void __aeabi_assert(const char *expr, const char *file,
                            int line)
{
    Uart_printf(UART1, "Assert failed in %s at line %d\n",
                                                    file,
line);
    __asm("BKPT");
}
```

Listing 12-10 The complete implementation for the "assert failed" function in Keil MDK includes using an instruction breakpoint to stop code execution and notify the developer

Verifying Assertions Work

Once the assertion is implemented, I will always go through and test it to ensure it is working the way I expect it to. The best way to do this is to simply create an assertion that will fail somewhere in the application. A great example is to place the following assertion somewhere early in your code after initialization:

```
assert(false);
```

This assertion will never be true, and once the code is compiled and executed, we might see serial output from our application that looks something like this:

- "Assertion failed in Task_100ms.c at line 17"

As you can see, when we encounter the assertion, our new assert failed function will tell us that an assertion failed. In this case, it was in the file Task_100ms.c at line number 17. You can't get more specific about where a defect is hiding in your code than that!

Three Instances Where Assertions Are Dangerous

Assertions, if used properly, have been proven to improve code quality. Still, despite code quality improvements, developers need to recognize that there are instances where using assertions is either ineffective or could cause serious problems. Therefore, before using assertions, we must understand the limits of assertions. There are three specific instances where using an assertion could cause problems and potentially be dangerous.

Instance #1 – Initialization

The first instance where assertions may not behave or perform as expected is during the system initialization. When the microcontroller powers up, it reads the reset vector and then jumps to the address stored there. That address usually points to vendor-supplied code that brings up the clocks and performs the C copy down to initialize the runtime environment for the application. At the same time, this seems like a perfect place to have assertions; trying to sprinkle assertions throughout this code is asking for trouble for several reasons.

First, the oscillator is starting up, so the peripherals are not properly clocked if an assertion were to fire. Second, at this point in the application, printf will most likely not have been mapped, and whatever resource it would be mapped to would not have been initialized. Stopping the processor or trying to print something could result in an exception that would prevent the processor from starting and result in more issues and debugging than we would want to spend.

For these reasons, it's best to keep the assertion to areas of the code after the low-level initializations and under the complete control of the developer. I've found that if you are using a library or autogenerated code, it's best to leave these as is and not force assertion checks into them. Instead, in your application code, add assertions to ensure that everything is as expected after they have "done their thing."

Instance #2 – Microcontroller Drivers

Drivers are a handy place to use assertions, but, again, we need to be careful which drivers we use assertions in. Consider the case where we get through the start-up code. One of the first things many developers do is initialize the GPIO pins. I often use a configuration table that is passed into the Gpio_Init function. If I have assertions checking this structure and something is not right, I'm going to fire an assertion, but the result of that assertion will go nowhere! Not only are the GPIO pins not initialized, but printf and the associated output have not yet been mapped! At this point, I'll get an assertion that fails silently or, worse, some other code in the assertion that fails that then has me barking up the wrong tree.

A silent assertion otherwise is not necessarily a bad thing. We still get the line number of the failed assertion and information about what the cause could be stored in memory. The issue is that we don't realize that the assertion has fired and just look confounded at our system for a while, trying to understand why it isn't running. As we discussed earlier, we could set an automatic breakpoint or use assembly instructions to make it obvious to us. One of my favorite tricks is to dedicate an LED as an assert LED that latches when an assertion fires. We will talk more about it shortly.

Instance #3 – Real-Time Hardware Components

The first two instances we have looked at are truthfully the minor cases where we could get ourselves into trouble. The final instance is where there is a potential for horrible things to happen. For example, consider an embedded system that has a motor that is being driven by the software. That motor might be attached to a series of gears lifting something heavy, driving a propulsion mechanism, or doing other practical work. If an operator was running that system and the motor suddenly stopped being driven, a large payload could suddenly give way! That could potentially damage the gearing, or the payload could come down on the user![8]

Another example could be if an electronic controller were driving a turbine or rocket engine. Suppose the engine was executing successfully, and the system gained altitude or was on its way to orbit, and suddenly an assertion was hit. In that case, we suddenly have an uncontrolled flying brick! Obviously, we can't allow these types of situations to occur.

For systems that could be in the middle of real-time or production operations, our assertions need to have more finesse than simply stopping our code. The assertions need to be able to log whatever data they can and then notify the application code that something has gone wrong. The application can then decide if the system should be shut down safely or terminated or if the application should just proceed (maybe even attempt recovery). Let's now look at tips for how we can implement our assertions to be used in real-time systems.

Getting Started with Real-Time Assertions

A real-time assertion is

"An assertion in a *real-time* program that, if evaluated as false, will identify a bug without breaking the real-time system performance."

As discussed earlier, you want to be able to detect a bug the moment that it occurs, but you also don't want to break the system or put it into an unsafe state. Real-time assertions are essentially standard assertions, except that they don't use a "BKPT" instruction or infinite loop to stop program execution. Instead, the assertion needs to

1) Notify the developer that an assertion has failed
2) Save information about the system at the failed assertion
3) Put the system in a safe state so the developer can investigate the bug

There are many ways developers can do this, but let's look at four tips that should aid you when getting started with real-time assertions.

Real-Time Assertion Tip #1 – Use a Visual Aid

The first and most straightforward technique developers can use to be notified of an assertion without just halting the CPU is to signal with a visual aid. In most circumstances, this will be with an LED, but it is possible to use LED bars and other visual indicators. The idea here is that we want to print out the message, but we also want to get the developers' attention. Therefore, we can modify the assert failed function like Listing 12-11, where we remove the BKPT instruction and instead place a call to change the state of an LED.

```
void __aeabi_assert(const char expr, const char *file,
                                                int line)
{
   Uart_printf(UART1, "Assertion failed in %s at line %d\n",
                                                    file,
line);
   // Turn on the LED and signal an assertion.
   LED_StateWrite(LED_Assert, ON);
}
```

Listing 12-11 When the assertion occurs, a modification to the assert function turns on an LED, LED_Assert

The latched LED shows that the assertion has occurred, and the developer can then check the terminal for the details. While this can be useful, it isn't going to store more than the file and line, which makes its use just a notch up from continuously watching the terminal. Unfortunately, as implemented, we also don't get any background information about the system's state. You only know which assertion failed, but if you crafted the assertion expression correctly, this should provide nearly enough information to at least start the bug investigation.

Real-Time Assertion Tip #2 – Create an Assertion Log

Assertions usually halt the CPU, but if we have a motor spinning or some other reason we don't want to stop executing the code, we can redirect the terminal information to a log. Of course, logs can be used in the development, but they are also one of the best ways to record assertion information from a production system without a terminal. If you look at the DevAlert product from Percepio, they tie assertions into their recording system. They then can push the information to the cloud to notify developers about issues they are having in the field.[9]

There are several different locations to which we could redirect the assertion log. The first location that we should consider, no matter what, is to log the assertion data to RAM. I will often use a circular buffer that can hold a string of a maximum length and then write the information to the buffer. I'll often include the file, line number, and any additional information that I think could be important. I sometimes modify the assertion failed function to take a custom string that can provide more information about the conditions immediately before the assertion was fired. Finally, we might log the data doing something like Listing 12-12.

```
void __aeabi_assert(const char expr, const char *file,
                                                int line)
{
#if ASSERT_UART == TRUE
   Uart_printf(UART1, "Assertion failed in %s at line %d\n",
                                                    file,
line);
 #elif
   Log_Append(ASSERT, Assert_String, file, line);
#endif
   // Turn on the LED and signal an assertion.
   LED_StateWrite(LED_Assert, ON);
}
```

Listing 12-12 An assert function can map where the output from assert will be placed

You'll notice in Listing 12-12 that I've also started adding conditional compilation statements to define different ways the assertion function can behave. For example, if ASSERT_UART is true, we just print the standard assertion text to the UART. Otherwise, we call Log_Append, which will store additional information and log the details in another manner. Once the log information is stored in RAM, there would be some task in the main application that would periodically store the RAM log to nonvolatile memory such as flash, an SD card, or other media.

Real-Time Assertion Tip #3 – Notify the Application

Assertions are not designed to be fault handlers; a real-time assertion can't just stop the embedded system. Of course, we may still want to stop the system, but to do so, we need to let the main application know that a defect has been detected and that we need to move into a safe mode as soon as possible (at least during development). We can create a signaling mechanism between the assertion library and the main application.

For example, we may decide that there are three different types of assertions in our real-time system:

1) Critical assertions that require the system to move into a safe state immediately.

2) Moderate assertions don't require the system to stop, but the developer is immediately notified of an issue so that the developer can decide how to proceed.

3) Minor assertions don't require the system to be stopped and may not even need to notify the application. However, these assertions would be logged.

We might add these assertion severity levels into an enum that we can use in our code. For example, Listing 12-13 demonstrates the enumeration.

```
typedef enum
{
   ASSERT_SEVERITY_MINOR,
   ASSERT_SEVERITY_MODERATE,
   ASSERT_SEVERITY_CRITICAL,
   ASSERT_SEVERITY_MAX_COUNT
}AssertSeverity_t
```

Listing 12-13 An example assertion severity level enumeration

Our assert failed function would then require further modifications that would allow it to notify the main application. We might even need to move away from the C standard library assert functions. For example, we might decide that the developer should specify if the assertion failed and what severity level the assertion is. We may want to redefine our assertion failed function to look something like Listing 12-14.

```
void assert_failed(const char expr, const char *file, int
                           line, AssertSeverity_t Severity)
{
#if ASSERT_UART == TRUE
   Uart_printf(UART1, "Assertion failed in %s at line %d\n",
                                                  file,
line);
 #elif
   Log_Append(ASSERT, Assert_String, file, line);
#endif
   // Turn on the LED and signal an assertion.
   LED_StateWrite(LED_Assert, ON);
   App_Notify(Severity)
}
```

Listing 12-14 A custom assertion that can specify the severity level of the assertion failure

In the modified assertion function, we allow an assertion severity to be passed to the function and then have a custom function named

App_Notify that passes that severity to a function that will behave how the application needs it to based on the severity level. After all, each application may have its requirements for handling these things. So, for example, App_Notify can decide if the assertion is just logged or if some custom handler is executed to put the system into a safe state.

Real-Time Assertion Tip #4 – Conditionally Configure Assertions

Assertions are a simple mechanism for detecting defects, but developers can create as sophisticated a mechanism as they decide will fit their application. If you plan to use assertions in both development and production, it can be helpful to create a series of conditions that determine how your assertions will function. For example, you might create conditions that allow the output to be mapped to

- UART
- Debug Console
- Serial Wire Debug
- A log file
- Etc.

There can also be conditions that would disable assertion functionality altogether or the information that would gather and be provided to the log. There may be different capabilities that a developer wants during development vs. what they would like in production. What's important is to think through what you would like to get from your assertion capabilities and design the most straightforward functionality you need. The more complex you make it, the greater the chances that something will go wrong.

A Few Concluding Thoughts

Interfaces can be designed to create a contract between the interface designer and the developer using that interface. The designer can carefully craft what the developer would expect from their interface by defining preconditions and postconditions. These conditions can then be verified by using executable instructions and assertions.

Assertions provide you a mechanism to verify the interface contract but also verify at any point in an application that developer assumptions are met. Assertions can be a powerful tool to detect bugs the moment they occur. A potential problem with assertions arises if you build real-time embedded systems. You can't have a system just stop! The result could damage the machine, the operator, or the user. To prevent an issue, real-time assertions can keep the system safe while still catching the bug in the act.

Designers and developers do need to be careful when they start to use real-time assertions. The inherent temptation to turn them into fault and error handlers is an entirely different exercise. Faults and errors are used to manage runtime issues, while assertions are meant to catch bugs. Be careful to make sure that you understand and adhere to the differences!

CautionDon't allow your assertion functions to become runtime fault and error handlers!

Action Items
To put this chapter's concepts into action, here are a few activities the reader can perform to start finding and using the right tools for their job:

- Review the interfaces in your application code. Ask yourself the following questions:
  - Are they consistent?
  - Memorable?
  - Does a contract exist to enforce behavior and use?
- Review Listings 12-2 and 12-3. If you have a microcontroller with two ports with eight I/O channels, what might the postcondition contract look like using assertions? (Hint: You'll need a for loop and the Dio_ChannelRead() function.)
- In your development environment, review how the assert macro is implemented. Identify differences between its implementation and what we've seen in this chapter.
- Enable assertions and write a basic assertion failed function.
- Pick a small interface in your code base and implement a contract using assertions. Then, run your application code and verify that it is not violating the contract!
- Explore the use of real-time assertions. Build out some of the real-time

assertion capabilities we explored in this chapter.
- Going forward, make sure that you define enforceable contracts on your interfaces.
- Jack Ganssle has also written a great article on assertions that can be found at
  - www.ganssle.com/item/automatically-debugging-firmware.htm

These are just a few ideas to go a little bit further. Carve out time in your schedule each week to apply these action items. Even small adjustments over a year can result in dramatic changes!

Footnotes

1 The interface modifies, interacts, or changes the state of the system outside the interface scope.

2 A set of assertions that must always be true during the lifetime of the interface.

3 Sadly, I committed this definition to memory and have been unable to find the original source!

4 Sigh. The devil is always in the details. If you use assert, you literally must check your compiler implementation.

5 http://laser.cs.umass.edu/courses/cs521-621.Fall10/documents/ISSRE_000.pdf

6 I wish the function name was included in the standard so that assert failed functions could be more easily ported!

7 Yes, we must be careful with copy/paste to ensure that we don't inject any bugs! For this operation, the risk is minimal.

8 This is an example that Jean Labrosse brought to my attention several years ago while we were chatting at Embedded World. I had done some webinars or wrote a blog at the time, and it led to some ideas and topics that we will discuss later in the chapter.

9 https://percepio.com/devalert/