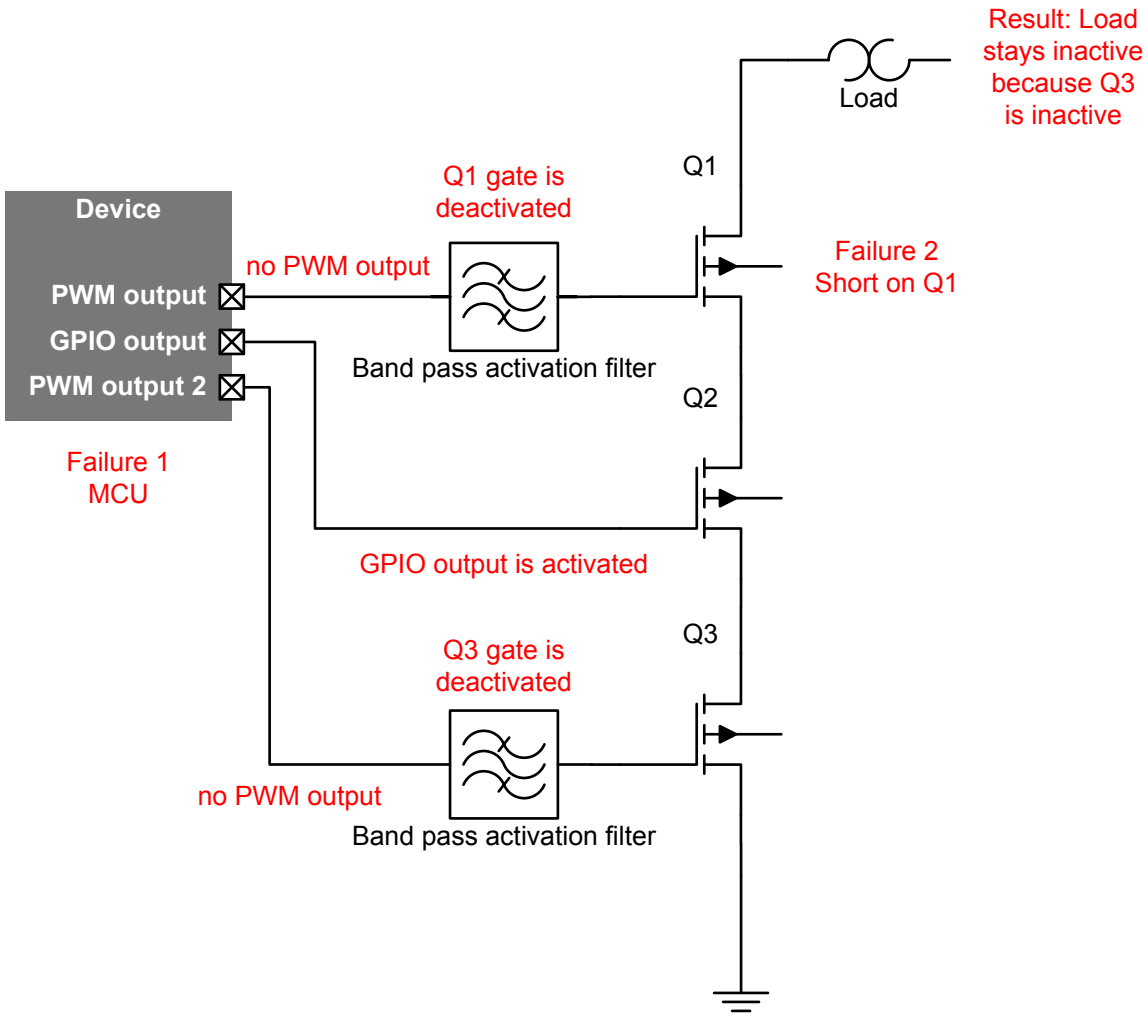
AN1074: EFM8 Family IEC60730 Library

The IEC60730 library allows OEMs to quickly integrate functional safety into their EFM8 based devices, per IEC 60730 software requirements as found in Annex H, Edition 5.1 2015-12, available at https://webstore.iec.ch/publication/23952.

This application note discusses configuring and using those routines, as well as other suggestions to meet IEC60730 certification. It will also cover integration testing requirements for certification.

KEY FEATURES

- Certified library for IEC60730
- Compatible with certified library for Capacitive Touch Sensing (CSLIB_IEC)
- Instructions for integration in OEM design
- Tips for integration testing when going through certification



1. Device Compatibility

This application note supports multiple device families, and some functionality is different depending on the device.

The following EFM8 devices are supported:

- EFM8 Busy Bee (EFM8BB1/EFM8BB2/EFM8BB3)
- EFM8 Laser Bee (EFM8LB1)
- EFM8 Sleepy Bee (EFM8SB1/EFM8SB2)
- EFM8 Universal Bee (EFM8UB1/EFM8UB2)

EFM8UB3 devices are not supported by this library.

2. Introduction

2.1 Functional Safety

What is functional safety? Functional safety is not the same thing as reliability or durability. Functional safety is what is required once reliability or durability has failed. Is a system in such a failed state still safe for a consumer? Does the potential exist for harm to the consumer or the environment? IEC 60730 is focused on the use of consumer white-box devices, typically items found in a home. Even within a home, systems exist that control potentially dangerous situations for ovens, microwaves, dishwashers, washing machines, hot water heaters, etc. Even a refrigerator can potentially cause serious damage with a water dispenser or an unsafe situation with an ice-maker screw turning with the door open.

IEC60730 covers all aspects of system safety, from the cables used to power a system to the software used to monitor inputs and drive the controls. This library contains software written for the EFM8 family to meet the requirements found in IEC60730, Annex H: Requirements for electronic controls.

Within Annex H, there are three types of electronic controls (taken from IEC60730-1 H.2.22):

- Class A — control functions which are not intended to be relied upon for the safety of the application
  - Examples are: room thermostats, temperature control
- Class B — control functions which are intended to prevent an unsafe state of the appliance
  - Examples are: thermal limiter, pressure limiter
- Class C — control functions which are intended to prevent special hazards such as explosion or whose failure could directly cause a hazard in the appliance
  - Examples are: burner control systems, thermal cut-outs for closed water systems (without vent protection)

The EFM8 IEC60730 library meets the requirements for Class B controls. This covers the controls most commonly found in household white-box goods.

2.2 Example System

For this document, an example system of a self-cleaning oven with stovetop will be used. This system contains the following parts:

1. High temperature oven, capable of reaching self-cleaning temperatures. When the oven is at self-cleaning temperatures, a hazardous condition will occur if the oven door opens because the air is hot enough to cause burns.
2. Temperature sensor for the oven, used to control cook temperature and self-clean temperature.
3. Latch for the oven, used during self-cleaning. This latch is slow-opening. When engaged to open, the open signal must be received for at least one second before the latch is fully opened and the door can open. If the open signal is terminated before the latch is fully opened, the door cannot be opened.
4. Light within the oven and a manual switch to activate it.
5. Touch controls and indicators for the oven.
6. Stovetop with four induction coils. Induction coils do not directly provide heat. Instead, heat is induced in a pot (or pan) via magnetic induction.
7. Temperature sensors for the stovetop induction coils. Because the stovetop cannot sense the amount of material in the pot, the amount of energy the induction coil provides must be controlled by the heat of the stovetop conducted from the pot. The pot must not be heated excessively, or else the contents may catch fire or the pot itself be damaged/too hot to handle.
8. Touch controls and indicators for the induction coils.
9. Clock and timer with audible buzzer and touch controls. The clock and timer do not directly control or interact with other parts.

2.2.1 Classification of the Controls in the Example System

Within the example system given above, the controls are classified as follows:

- Class A — These controls are not relied upon for the safety of the system. A failure in one of these controls does not cause an unsafe state. The controls are classified as Class A; however, they could be designed to meet Class B requirements.
  - Light within the oven and the switch to activate it.
  - Clock and timer, including controls to set them, and buzzer control.
- Class B — These controls prevent an unsafe state of the appliance. A failure in one of these controls could cause an unsafe state.
  - High temperature oven, controls, and temperature sensor, including the door latch.
  - Stovetop induction coils, controls, and temperature sensor.

2.2.2 EFM8 Devices within the Example System

For this application note, the EFM8 is used in the following locations in the example system:

- Touch controls for the oven and stovetop, using CSLIB_IEC (touch controls EFM8).
- Activation and temperature monitoring of the stovetop induction coils (stovetop EFM8).
- Activation and temperature monitoring of the oven, including controlling the door latch (oven EFM8).
- Clock and timer touch controls and display (clock EFM8).

A separate host MCU is responsible for controlling and monitoring the overall system. It receives input from the touch controls EFM8 and forwards commands to the stovetop EFM8 and oven EFM8. The clock EFM8 is independent from the other MCUs.

2.2.3 Failure Modes and Effects Analysis

Failure modes and effects analysis is required to understand the effects on the safety-related functions of any hardware component failures. For example, the failure of the touch controls will prevent a user from turning off the oven or induction coils, the failure of the stovetop control may cause unwanted activation of the induction coils or prevent shutting off coils, failure of the temperature sensors may allow excessive temperatures to occur, etc.

This analysis must be done for each hardware component in the system.

2.2.4 Fault Reaction Time

One of the key quantities for IEC60730 certification is the Fault Reaction Time of the system. Fault Reaction Time is defined as *time between the occurrence of a fault and the point where the control has reached a defined state*. For our example system, there are two primary hazards that must be addressed: the oven door opening during a self-clean cycle, and a pot on the induction coils reaching an unsafe temperature.

The switch for the oven door is specifically designed to be slow-opening, requiring 1 second before opening fully. Any time during the opening process if the open signal is terminated, the switch will halt, keeping the system in a defined, safe state.

For the second situation, it will take many seconds of induction heating before a pot will reach a dangerous temperature; temperature sensors on the stovetop will detect the pot temperature exceeding the safe range.

From the software controls, these two requirements set a maximum fault reaction time of 1 second. In other words, the software can take up to 1 second before entering a safe state.

2.2.5 Safe State

The Safe State is a defined state of the control where the output terminals ensure a safe situation under all circumstances.

For the example above, a Safe State will prevent the oven self-clean latch from opening and disable all induction heating coils. An OEM may also choose to notify the user that the system has entered a Safe State due to a malfunction, but this is not a requirement.

3. Annex H Measures to Address Software Faults/Errors

The table below lists the measures provided by the IEC60730 to address software faults/errors. This table is taken from the IEC60730 documentation available at http://devtools.silabs.com/studio/doc/EFM8/software/Lib/group__efm8__iec60730.html.

Table 3.1. Measures to Address Software Faults/Errors

Component	Measure Used	Notes
1.1 Registers	Periodic self-test using a Static memory test	Provided by library
1.2 Instruction decoding and execution	None	Not required for Class B
1.3 Program counter / Watchdog	Logical monitoring of the program sequence. Code execution sets flag that is periodically checked. Watchdog timer prevents runaway program execution.	Provided by library, see example for integration sample
1.4 Addressing	None	Not required for Class B
1.5 Data paths instruction decoding	None	Not required for Class B
2 Interrupt handling and execution	Time-slot monitoring (upper and lower limit)	Provided by library, see example for integration sample
3 Clock	Reciprocal Comparison. Use separate oscillator to monitor SYSCLK – Calculate ratio and determine range based on accuracy	Provided by library, see example for integration sample
4.1 Invariable Memory	Periodic 16-bit CRC	Provided by library, see example for integration sample
4.2 Variable Memory	Periodic static memory test using March-C & stack guard	Provided by library, see example for integration sample
4.3 Addressing (variable and invariable memory)		4.1 and 4.2 provide coverage for this component
5.1 Internal data path		4.1 and 4.2 provide coverage for this component
5.2 Internal addressing		4.1 and 4.2 provide coverage for this component
6 External Communications		Provided by OEM, UART example in library
6.1 External Communications – Data	16 bit CRC	CRC check provided by library
6.2 External Communications – Addressing	16 bit CRC including the address	OEM must include in protocol proper address verification – see UART example in library
6.3 External Communications – Timing (UART example)	Scheduled transmission	OEM must include in protocol proper timing measures – see UART example in library
7 Input/Output periphery/ 7.1 Digital I/O	Plausibility check	Provided by OEM
7.2 Analog I/O/ 7.2.1 A/D and D/A convertor	Plausibility check	Provided by OEM
7.2.2 Analog Multiplexer	Plausibility check	Provided by OEM
8 Monitoring device and comparators	None	Not needed for Class B
9 Custom chips	None	Not Applicable

4. Library Integration

The IEC60730 library is complex software. It requires OEMs to have a deep understanding of their system, the EFM8, and the IEC60730 standard. This section covers integrating the IEC60730 library, including configuration options, into a project under development.

4.1 `iec60730_SafeState()`

When the library detects a failure, it must enter a defined state. OEMs must update the function `iec60730_SafeState()`, found in `iec60730_oem_functions.c` in the example, so that it puts the system into a defined state. This may include setting outputs into a safe state, notifying the user a failure has occurred, or sending a notification to a host device. The variable failure passed into `iec60730_SafeState()` allows OEMs to determine the type of error. Activity within `iec60730_SafeState()` must be kept minimal, to make sure the error doesn't cause additional failures.

While in `iec60730_SafeState()`, interrupts must be disabled and the watchdog refreshed.

4.2 POST (Power On Self Test)

`iec60730_Post()` is normally called after system initialization is complete, but before beginning primary operating mode in the system main() loop. It includes a Watchdog test that resets the system to verify Watchdog operation.

OEMs must expect initialization code before `iec60730_Post()` to execute twice. Initialization code execution time must be short enough that the watchdog can be refreshed in `iec60730_Post()` before expiration.

4.3 BIST (Built In Self Test)

`iec60730_Bist()` is executed as part of the system `main()` loop, typically at the end. Systems with long execution times will require manual watchdog refresh, possibly adjustment of `iec60730_TestClockTick()` frequency to ensure `iec60730_programmeCounterTest()` passes, or calling `iec60730_Bist()` at multiple locations in the main system loop.

Interrupts should remain globally enabled. If interrupts must be disabled for a critical section of firmware, the critical section should follow best practices and be as short as possible. The time in the critical section must be shorter than the fastest clock timer interrupt (usually 10 ms), or `iec60730_timerTestControl` must be used to disable the timer tests.

4.4 Programme Counter

The Programme Counter check uses the bitfield variable `iec60730_programmeCounterCheck` to verify that all checks have completed in the required fault reaction time.

OEM code must set the bits `IEC60730_GPIO_COMPLETE` and `IEC60730_ANALOG_COMPLETE` in `iec60730_programmeCounterCheck` when each of these tests have completed.

OEMs can use the bits `IEC60730_OEM0_COMPLETE` - `IEC60730_OEM7_COMPLETE` in `iec60730_programmeCounterCheck` to verify their own test algorithms are executing at least once per every call to `iec60730_programmeCounterTest()`. Unused flags must be set to 1.

`iec60730_Bist()` can take several ms to execute. See Execution Time in the IEC60730 documentation for details. If the execution time is longer than the system can tolerate, the code in `iec60730_Bist()` can be split up into individual parts within the `system main()` loop, as long as all parts of `iec60730_Bist()` are included. For example, split it so that the Invariable Memory Check is in one part of the main loop, and the rest of the checks in a different part. As long as all tests complete before the Programme Counter Check executes, the system will continue normal operation.

4.5 Watchdog

`iec60730_WatchdogPost()` configures the Watchdog; do not use `enter_DefaultMode_from_RESET()` to setup the internal watchdog. `iec60730_WatchdogPost()` triggers a watchdog reset to validate that they can occur properly. Code before the POST is run twice, once during power on reset, and again after the watchdog reset.

When in a long latency loop, use `iec60730_RestartWatchdog()` to prevent a watchdog reset. Minimize the time in the loop as much as possible.

`iec60730_DisableWatchdog()` must be used on PCA-based watchdogs during flash programming because flash erase times are longer than the maximum watchdog timeout. Call `iec60730_RestartWatchdog()` to re-enable the watchdog once the flash erase has completed. During the flash erase time, any control signals must remain in a safe configuration. See Watchdog Test in the IEC60730 documentation for details.

4.6 Interrupt Handling and Execution

OEMs must update the array `iec60730_IRQFreqBounds` for their system use. Interrupt service routines must include code for incrementing `iec60730_IRQExecCount`; for an example, see `iec60730_oem_timer.c`. OEMs must choose and enumerate the index values for `iec60730_IRQExecCount`.

Interrupts are disabled during portions of the Invariable Memory Test and Variable Memory Test. Interrupts must be tolerant of the latency times these tests introduce to servicing an interrupt.

4.7 Clock

OEMs can modify the default System Clock and Timer Clock configurations in `iec60730_oem_timer.c` according to their system requirements.

4.8 Invariable Memory

OEMs must modify the `iec60730_Invariable` structure to align with their memory usage. `iec60730_Invar_Crc` or similar must be modified to store the CRC values. During the CRC generation for a block of memory, the CPU is halted and unable to respond to interrupt requests. For a typical system, the amount of invariable memory checked will determine the fault reaction time.

4.9 Variable Memory

More safety critical variables should be placed in IDATA, since IDATA is fully validated every call to `iec60730_VmcBist()`.

If not all of XDATA is used, `iec60730_XdataNumPartitions` can be decreased to cover only the used portions of memory. This will decrease the time to check the used XDATA space, but the time for `iec60730_Bist()` will not change.

Note: The USB XRAM in UB1/UB2 is not validated as part of the Variable Memory Check. Do not use this XRAM space as system memory.

5. Creating an IEC 60730 Class B Communications Interface

The example provides a sample Class B interface for UART0. An OEM designing their own serial interface must meet the requirements given in the IEC60730 standard. The library provides a 16-bit CRC routine that can be used by OEMs as part of their protocol.

5.1 Example Interface between EFM8s and a Host CPU

For the example system, an I2C serial interface is connected between the EFM8s in the system and the host CPU.

The protocol uses a 16-bit CRC to protect each packet, to guard against any corruption during transmission.

Multiple devices on the serial interface require unique addresses within the protocol.

Because a class A device, the timer/buzzer control EFM8, is also on the serial bus, the protocol must guarantee that the class A device cannot accidentally emulate class B traffic. To prevent the class A device from playing back a class B packet, the protocol includes a sequence number with each packet. The host controller and class B devices use the sequence number to verify the packet is received in the correct order. If the class A device played back a class B packet, the receiving device would detect that the sequence number did not increment, and enter Safe State.

The host CPU uses scheduled transmissions to determine if one of the other devices has entered Safe State. The host CPU expects a packet at least every 100 ms. If it does not receive a packet from a device, the host CPU will enter Safe State.

6. Digital I/O Plausibility Check

6.1 Outputs

For safety-related outputs, multiple outputs must be used together to verify that the target device is activated. For the oven example, activating the induction coils on the stovetop must use at least two outputs. One of the outputs must use a PWM waveform generated via software, along with external filtering hardware to generate an activation. Figure 6.1 Device with Two Output Redundancy on page 9 shows an example with output redundancy. If one of the transistors fails, the load does not activate. If the MCU fails, the PWM output will halt, and Q1 will not activate. This way if a single failure occurs, there is no accidental activation of the target device.

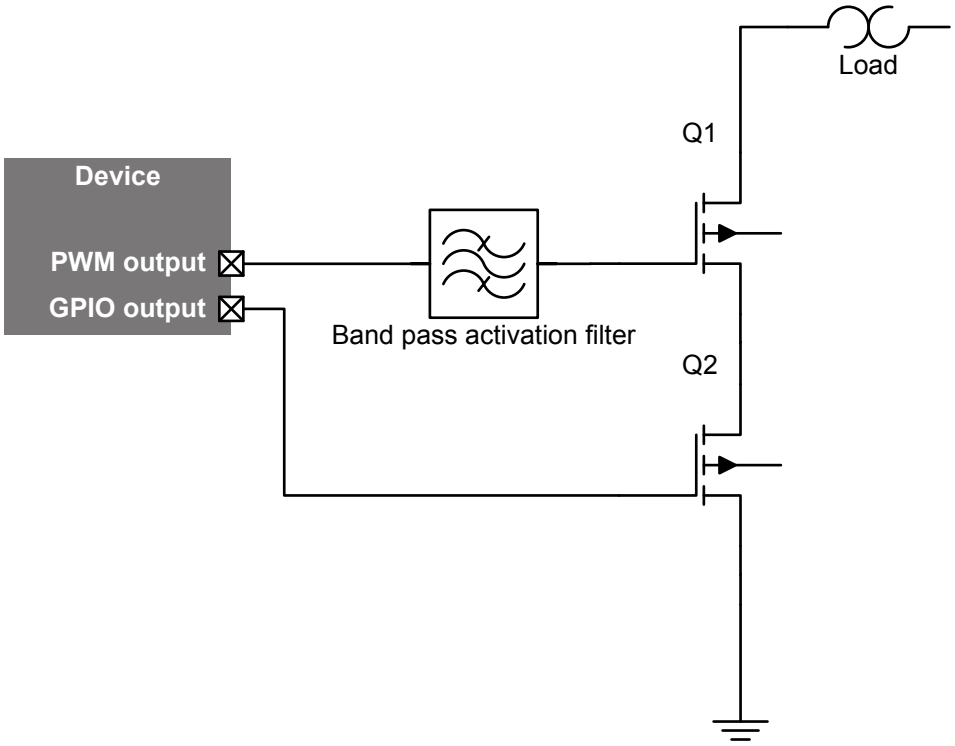


Figure 6.1. Device with Two Output Redundancy

However, for IEC60335-1 (this is a typical additional certification for safety) a second failure must be considered after the first failure occurs. In this case, Figure 6.1 Device with Two Output Redundancy on page 9 is insufficient. Say that the system fails as in Figure 6.2 Device with Two Output Redundancy, Software Failure Activates GPIO Output, Additional Failure Activates Q1 on page 10. Transistor Q2 is activated, but Q1 is not. However, if a second failure occurs across Q1, the target device is incorrectly activated.

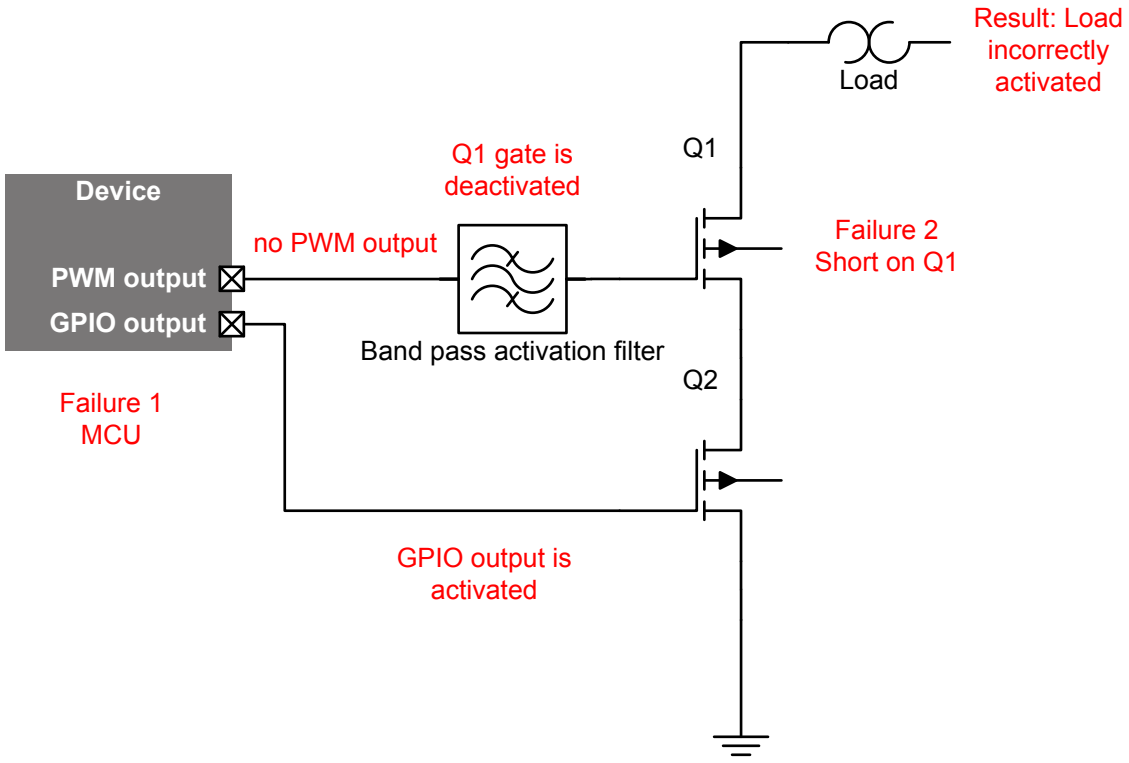


Figure 6.2. Device with Two Output Redundancy, Software Failure Activates GPIO Output, Additional Failure Activates Q1

To solve this requirement for IEC60335-1, a second PWM interface, using a different frequency than the first interface, must also activate the target device, as in Figure 6.3 Device with Three Output Redundancy on page 11.

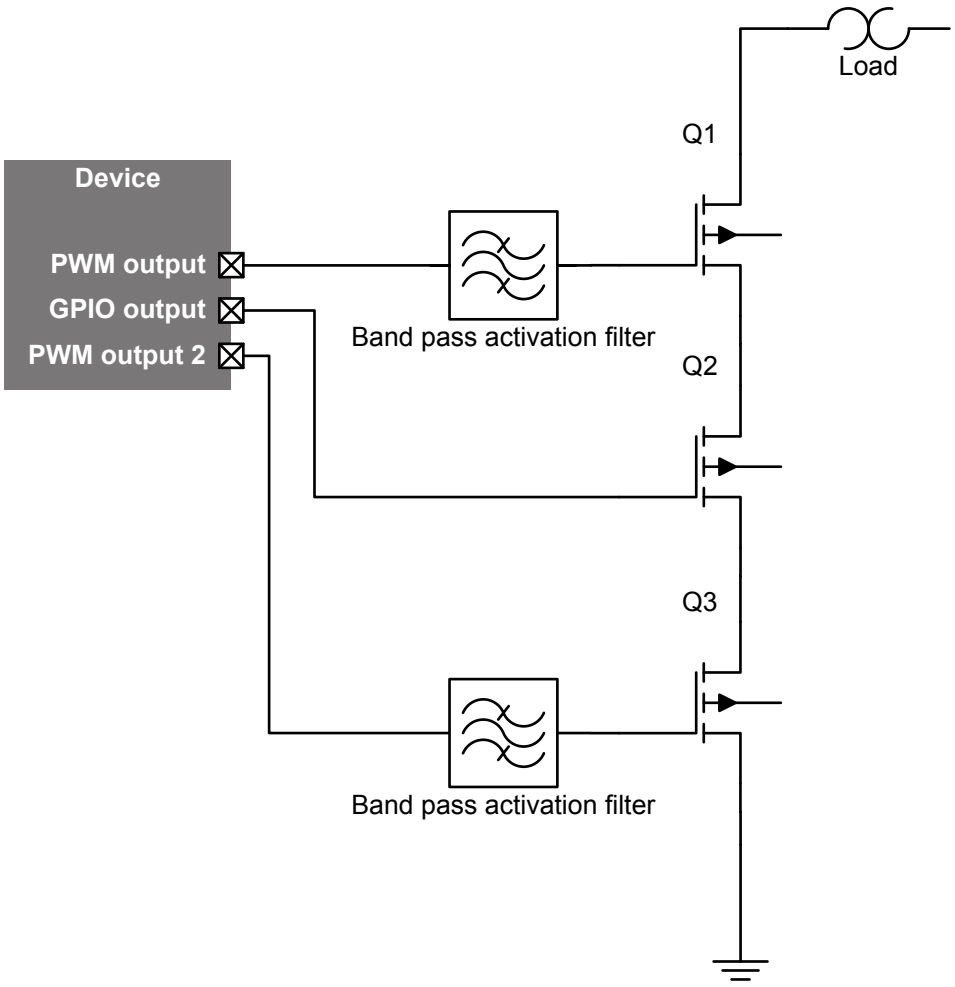


Figure 6.3. Device with Three Output Redundancy

As shown in Figure 6.4 Device with Three Output Redundancy, Software Failure Activates GPIO Output, Additional Failure Activates Q1 on page 12, a system failure leaving Q2 activated will prevent any second failure from activating the target device.

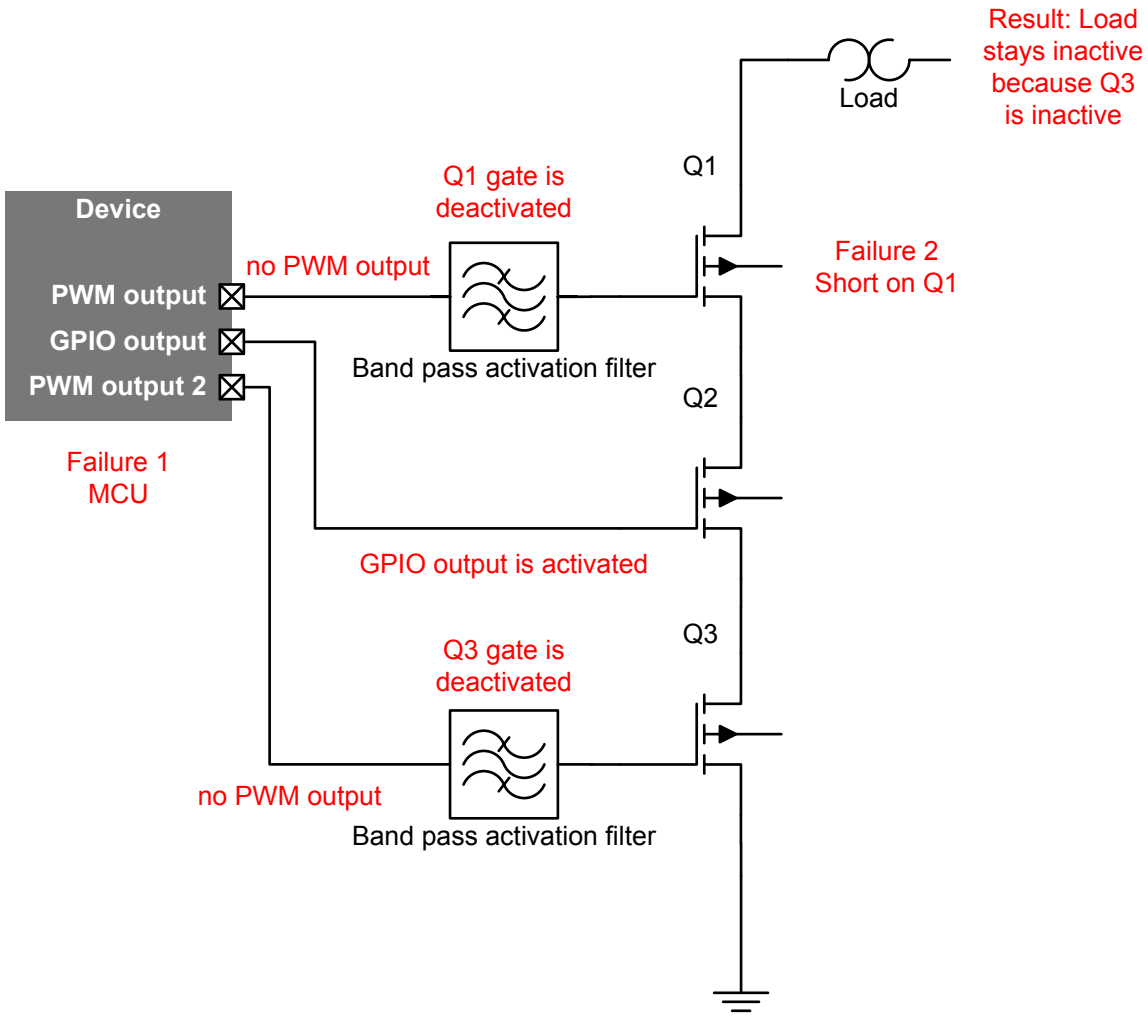


Figure 6.4. Device with Three Output Redundancy, Software Failure Activates GPIO Output, Additional Failure Activates Q1

6.2 Inputs

For safety-related inputs, redundancy or dynamic plausibility are common solutions.

6.2.1 Redundancy

Redundancy prevents a single failure from providing invalid data to the system. For example, the oven door may use two switches that detect when the door is closed. A discrepancy in the returned values would cause entry into safe state.

6.2.2 Dynamic Plausibility

Dynamic plausibility requires an input that can be controlled by the MCU, either directly or indirectly. For example, the oven door may use one switch that ties the input to ground via a small external resistor when the door is closed – otherwise the input is tied to VCC via the large internal pullup resistor. An output from the MCU is connected to the input near the small external resistor. During a plausibility check, the MCU shorts the output to VCC. The check confirms that the input reads VCC, verifying the MCU could detect if the door is open. Checking the MCU could detect the door is closed is not necessary, since the unsafe condition exists only when the door is open.

7. ADC Plausibility Check

Analog I/O plausibility checks require verifying that the ADC or DAC can sample or output the correct value, and also that any MUX on the ADC correctly reads from the requested input line.

7.1 ADC Value Validation

To verify the ADC hardware requires separate input(s) from the main sensor input. If this input is close to the expected value for the sensor, then a single separate input to validate the ADC hardware is sufficient. For example, if the temperature sensor for the induction coil hits the hazardous value at 1.8 V, then a separate input near 1.8 V is sufficient. If the sensor inputs have a large range, then two separate inputs at the low/high end of the ADC hardware is required. For example, if there are two sensors, one with a hazardous value cutoff at 1 V and another at 3 V, then two separate inputs at 0.5 V and 2.5 V can validate that the ADC hardware sample range is operating correctly. The plausibility test would verify that the separate inputs are measured correctly by the ADC.

For small systems, it may be easier to have a redundant sensor chain. For example, using two independent temperature sensors with two ADC inputs. A plausibility test would require the two inputs to track closely together.

7.2 Analog Mux Validation

Separately, the system must also validate that any ADC mux is functioning correctly. If ADC input 0x7 is requested, there must be confidence that ADC input 0x7 is actually what is measured by the ADC.

The largest concern is that the ADC mux control signals are shorted together internally. To validate this hardware, a simple test of two complementary addresses is sufficient. IE, address 0x5 (binary 0101) and 0xA (binary 1010) on a 16 input ADC. The separate input signals required for analog value validation can be placed on these inputs to verify the ADC mux at the same time as the ADC hardware. In this case, the analog value plausibility check will also validate the analog mux.

If additional voltage levels are not available, then dynamically change the ADC input value to validate that the ADC mux is correctly choosing the input. For example, if a thermistor is placed on ADC input 0x7 to measure temperature with a range of 1V to 3V, then a transistor controlled by a GPIO that shorts ADC input 0x7 to ground can be used to verify that the ADC mux is properly choosing the input.

8. Debugging with the Library

This section covers how to debug issues with the library and OEM firmware. It assumes experience with Simplicity Studio and common firmware debug areas such as setting breakpoints, stepping through code, and using watch expressions.

8.1 Pointing to the Library Source Files

Since the library was built using an automated build environment, Simplicity Studio will not find the source for the library when debugging a program. See Figure 8.1 Breaking within the Library on page 14 for a picture of what Simplicity Studio shows when breaking within the library.

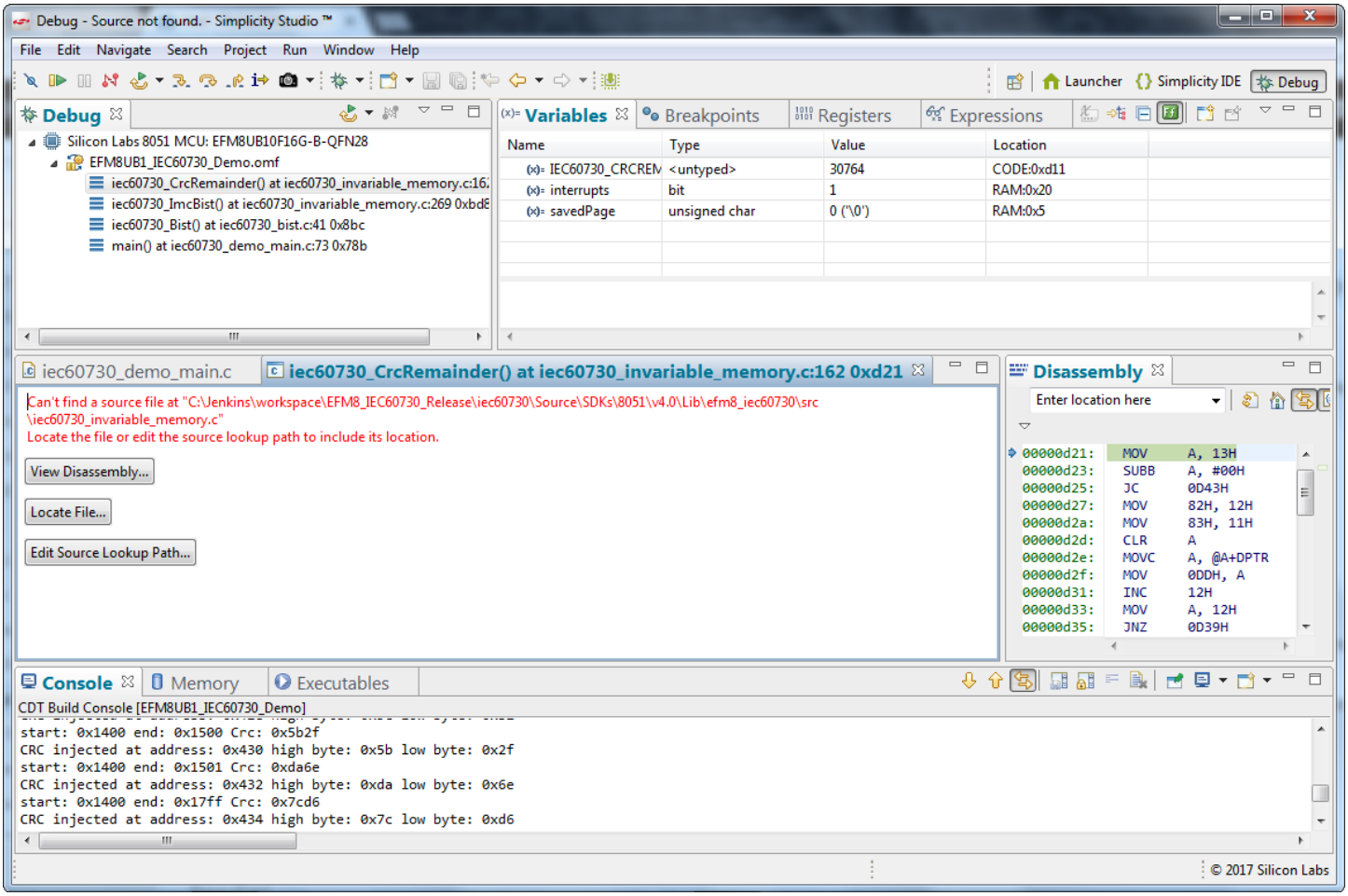


Figure 8.1. Breaking within the Library

To show the library source code during a debug session, Simplicity Studio must be pointed to the new location of the library source code within the SDK directory tree. While debugging, click [**Locate File**] and browse to the location of the source file, as shown in Figure 8.2 Browsing to the IEC60730 Library Source Files on page 15.

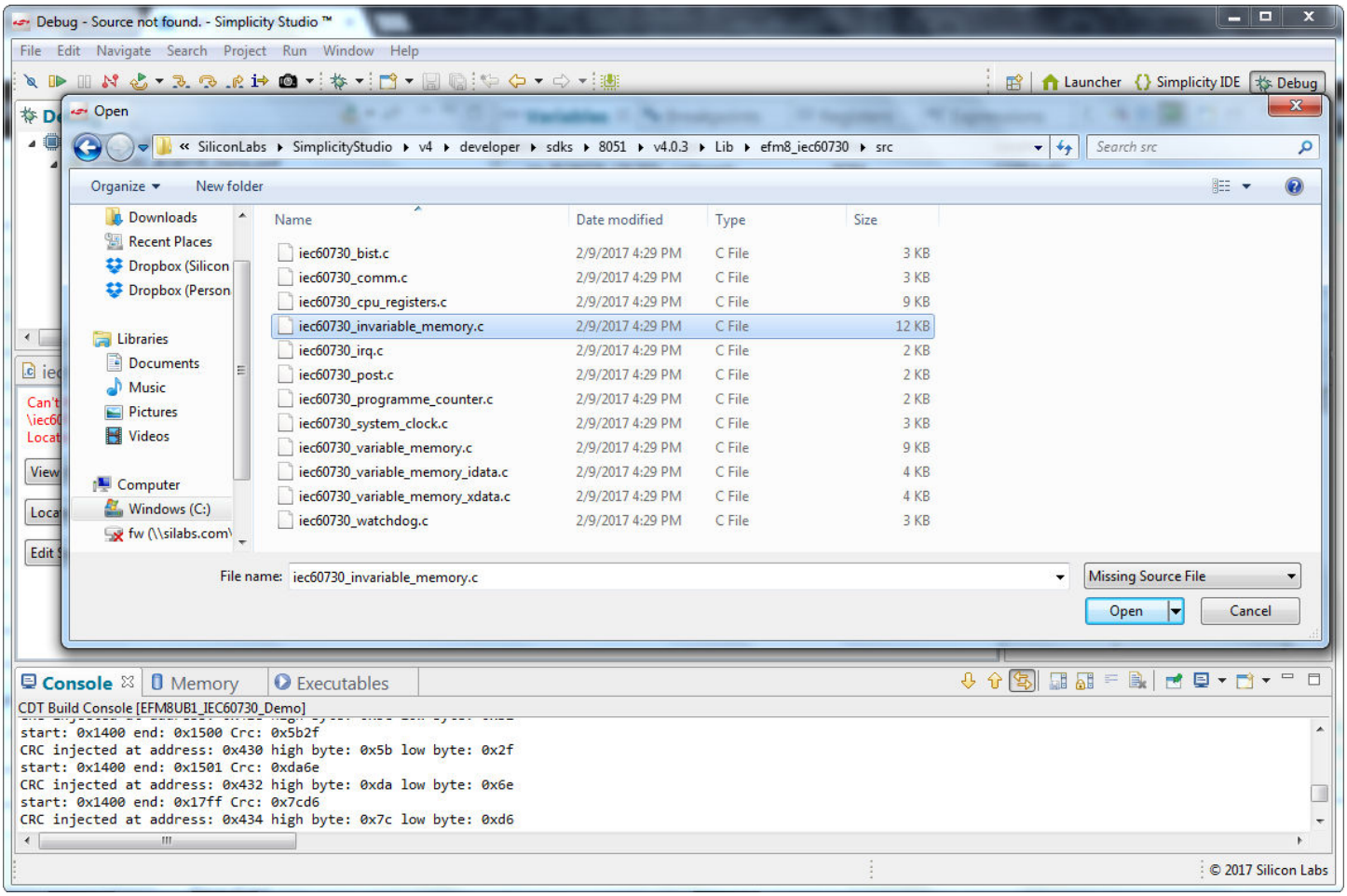


Figure 8.2. Browsing to the IEC60730 Library Source Files

Once Simplicity Studio is pointed to the library source files, it will correctly show source for any other library files except for the CPU registers check. Halting execution within the CPU registers check will point to a source file that does not exist, `iec60730_cpu_registers.SRC`. This file is not part of the IEC60730 source code, as it is generated during compilation from `iec60730_cpu_registers.c`. Addresses for labels within this file can be found in the `.M51` output file by searching for `_ASM_BKPT`. These labels can be used to set breakpoints, and execution can be manually compared against the original source file `iec60730_cpu_registers.c`.

Simplicity Studio will remember the location of the library source files for any new debug sessions with that EFM8 version of the device library. New projects with other EFM8 devices will require pointing Simplicity Studio to the library source files.

8.2 Setting Breakpoints

Breakpoints set at the labels located within the library give locations that variables or registers can be modified to validate operation.

To locate the address for a label, use the `.M51` file generated during the compile. For this example, search for the label `IEC60730_CPU_REGS_CORE_ACC_ASM_BKPT` within your project's `.M51` file. Note the last four digits of the address field. Figure 8.3 Last 4 Digits of the Address Field for `IEC60730_CPU_REGS_CORE_ACC_ASM_BKPT` on page 16 shows the `.M51` file from the EFM8UB1 example, with the last four digits of the address field highlighted.

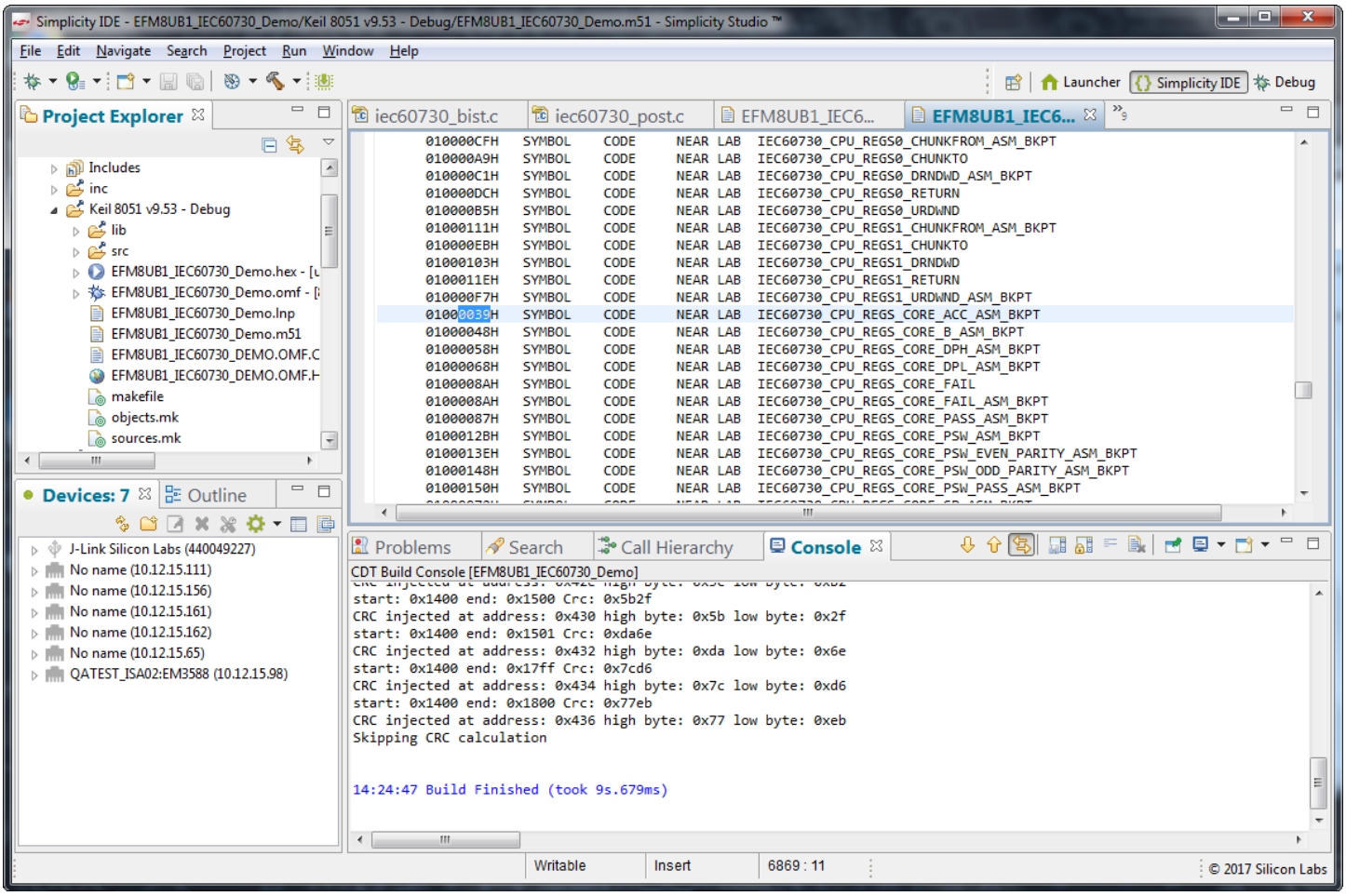


Figure 8.3. Last 4 Digits of the Address Field for `IEC60730_CPU_REGS_CORE_ACC_ASM_BKPT`

Debug the project under development. To set a breakpoint at this address location, write the last 4 of the address into the [**Disassembly**] window, and then left-click to the left of the address displayed in the [**Disassembly**] window. The following figure shows where to input the address. Left-click where indicated.

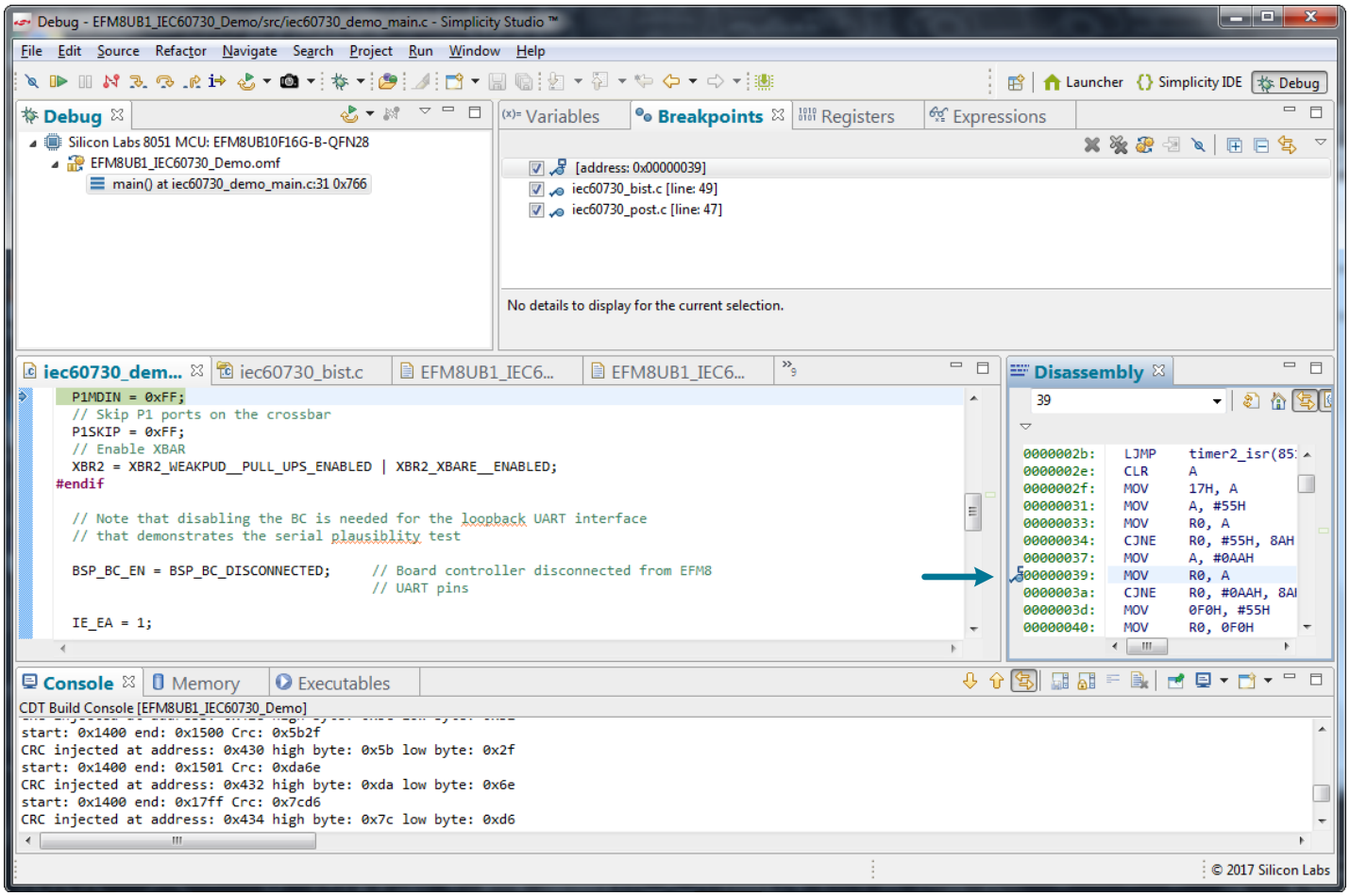


Figure 8.4. Setting a Breakpoint at IEC60730_CPU_REGS_CORE_ACC_ASM_BKPT

8.3 Watchdog Reset POST

During debugging, expect code before Watchdog Reset POST to execute twice.

9. Integration Testing as Part of Certification

For integration testing, an OEM must demonstrate to a certification house that the IEC library is properly called by their firmware. This section covers how to use Simplicity Studio to prove to a certification house that the library is checking on the health of the system.

The detailed IEC60730 library test specification documentation provided at http://devtools.silabs.com/studio/doc/EFM8/software/Lib/group___i_e_c60730___v_e_r_i_f_i_c_a_t_i_o_n.html has links to the automated tests performed during the build process for the library. These procedures are simplified and done manually during certification to prove that the API is properly called.

9.1 CPU Register Test

1. Debug the project under development.
2. Set a breakpoint at label `IEC60730_CPU_REGS_CORE_ACC_ASM_BKPT`.
3. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
4. Resume execution by pressing [**F8**].
5. Change the value of the [**Core register A**] to [**0xFF**], as shown in the following figure.

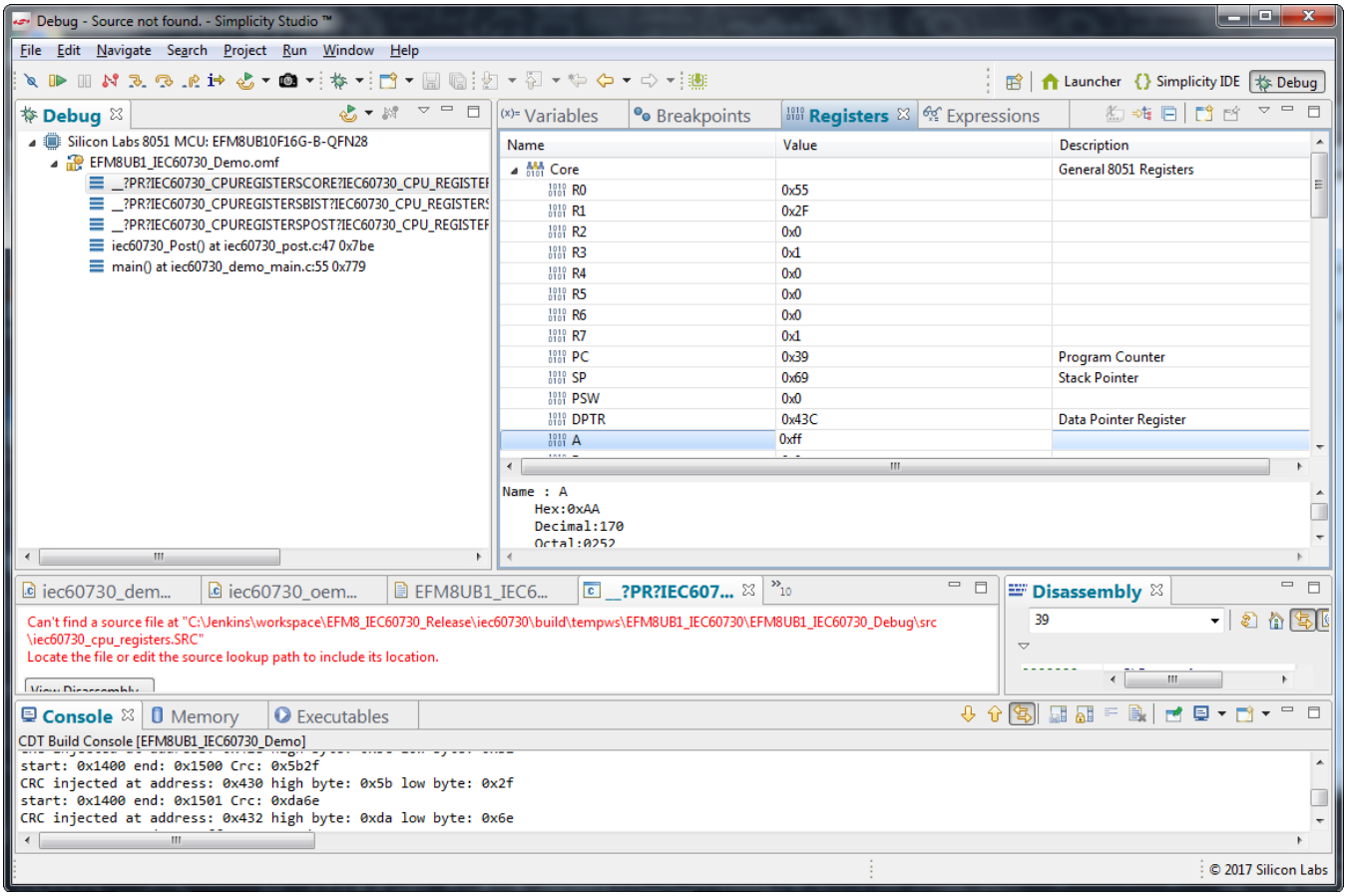


Figure 9.1. Corrupting the Accumulator Register to Cause a Failure

6. Resume execution by pressing [**F8**].
7. The device will now halt at `iec60730_SafeState()`.
8. This verifies that the library is correctly integrating the CPU Registers test into the device firmware.

9.2 Programme Counter Check

1. Debug the project under development. Set a breakpoint at label `IEC60730_PC_BKPT`.
2. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
3. Resume execution by pressing [**F8**].
4. The code will stop at `IEC60730_PC_BKPT`.
5. Add variable `iec60730_programmeCounterCheck` to the watch expressions.
6. Clear variable `iec60730_programmeCounterCheck` to 0x0000.
7. Resume execution again.
8. The device will now halt at `iec60730_SafeState()`.

9.3 Watchdog

1. Debug the project under development.
2. Set a breakpoint at label `IEC60730_WATCHDOG_POST_WATCHDOG_RESET_BKPT`.
3. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
4. Resume execution by pressing [**F8**].
5. The code will stop at `IEC60730_WATCHDOG_POST_WATCHDOG_RESET_BKPT`.
6. Set variable `iec60730_WatchdogState` to 0x02.
7. Resume execution again.
8. The device will now halt at `iec60730_SafeState()`.

9.4 Interrupt Handling

1. Debug the project under development.
2. Set a breakpoint at label `IEC60730_IRQ_BKPT`.
3. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
4. Resume execution by pressing [**F8**].
5. The code will stop at `IEC60730_IRQ_BKPT`.
6. Set variable `iec60730_IRQExecCount[0]` to 0x00.
7. Resume execution again.
8. The device will now halt at `iec60730_SafeState()`.

9.5 Clock

1. Debug the project under development.
2. Set a breakpoint at label `IEC60730_SYSTEM_CLOCK_FREQ_ADJUSTMENT_BKPT`.
3. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
4. Resume execution by pressing [**F8**].
5. The code will stop at `IEC60730_SYSTEM_CLOCK_FREQ_ADJUSTMENT_BKPT`.
6. Set `SFR CLKSEL` so that the `SYSCLK` frequency is divided by two.
7. Resume execution again.
8. The device will now halt at `iec60730_SafeState()`.

9.6 Invariable Memory

1. Debug the project under development.
2. Set a breakpoint at label `IEC60730_IDATA_URDWND_BKPT`.
3. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
4. Resume execution by pressing [**F8**].
5. The code will stop at `IEC60730_IDATA_URDWND_BKPT`.
6. Corrupt IDATA memory location 0x10 by setting it to 0xFF.
7. Resume execution again.
8. The device will now halt at `iec60730_SafeState()`.

9.7 Variable Memory

1. Debug the project under development.
2. Set a breakpoint at label `IEC60730_CRCBLOCK_BKPT`.
3. Set another breakpoint at `IEC60730_SAFE_STATE_BKPT` in the project under development.
4. Resume execution by pressing [**F8**].
5. The code will stop at `IEC60730_CRCBLOCK_BKPT`.
6. Corrupt the CRC calculation by setting `iec60730_CrcCalc` to 0x0000.
7. Resume execution again.
8. The device will now halt at `iec60730_SafeState()`.

9.8 Other Modules

Other modules, such as ADC/ADC Mux, I/O, and Communications, will require OEMs to demonstrate module testing for certification. Since these modules are custom modules developed by the OEM, the OEM must create sufficient tests to demonstrate to the certification house that the modules are functioning correctly.

10. Rebuilding the IEC60730 Library

Generally, an OEM will use the pre-built certified EFM8 IEC60730 library. However, in special situations an OEM might wish to add debug code, customize and rebuild the library, or integrate the library directly in their system. Recertification of the whole library will be required.

10.1 Importing the SLSPROJ file

From the Simplicity Studio IDE, choose [**Project**]>[**Import**]>[**MCU Project**]. Select [**Browse**], and navigate to `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\8051\v4.0.3\Lib\efm8_iec60730`. Open the SLSPROJ file for your EFM8 device. Finish the import wizard with the default settings.

10.2 Setting the Library Version

Set the library version to something other than 0.0.0. Each digit of the library version must be between 0 and 255. In `iec60730.h`, set `@version` on line 3 and `IEC60730_LIBRARY_VERSION` on line 150.

10.3 Building the Library

Build the library project normally.

10.4 Replacing the Pre-built Version

The output .LIB will be located in your Simplicity Studio workspace, visible in the [**Console**] output window. Rename the original `.LIB` file to `.LIB.CERT`, located at `\SiliconLabs\SimplicityStudio\v4\developer\sdks\8051\v4.0.3\Lib`, and replace it with the new version.

11. Revision History

11.1 Revision 0.2

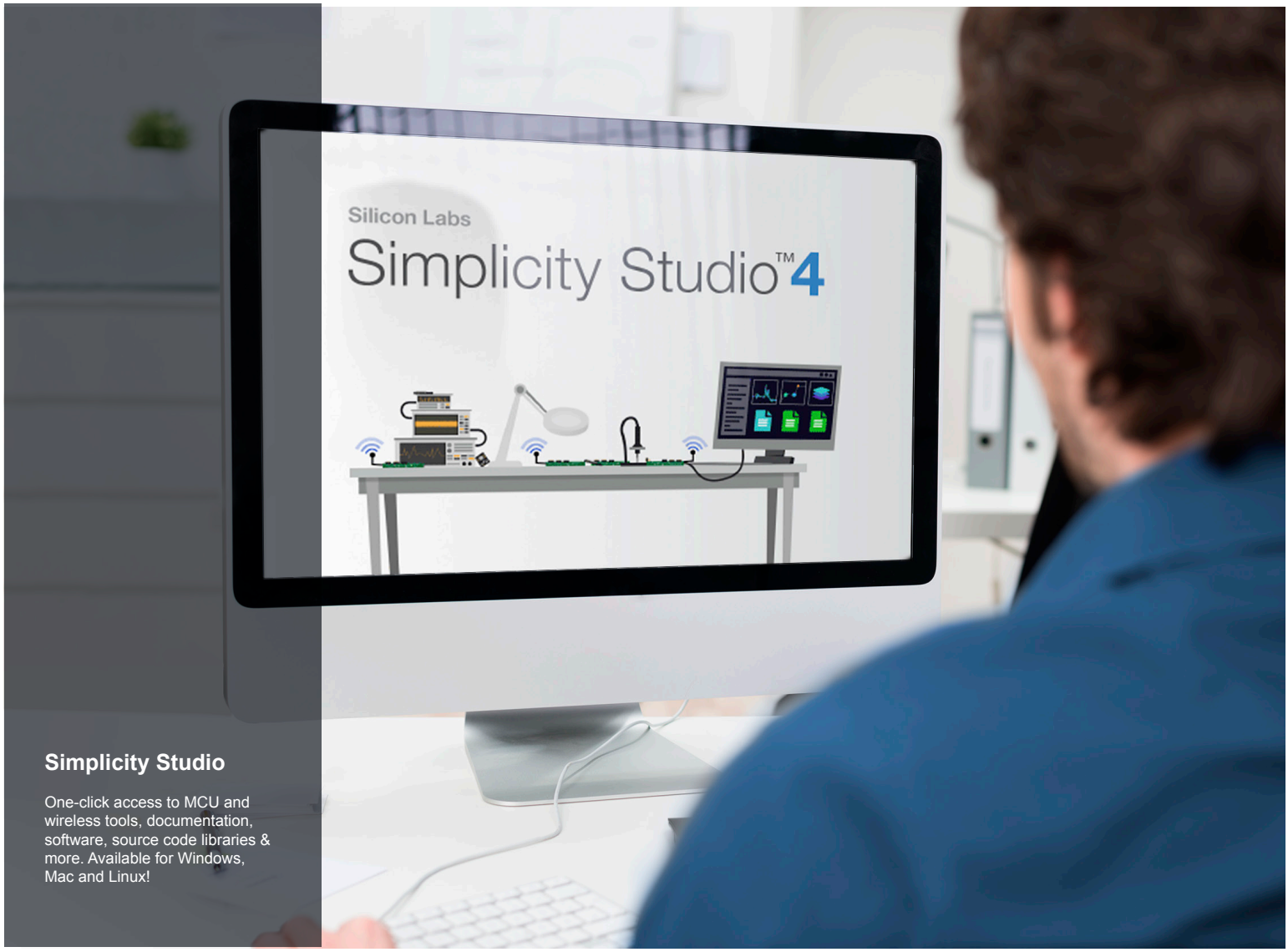
December 6th, 2017

Added 1. Device Compatibility.

11.2 Revision 0.1

March 13th, 2017

Initial release.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com