Atmel®

APPLICATION NOTE

Atmel AVR1610: Guide to IEC 60730 Class B Compliance with XMEGA

8-bit Atmel Microcontroller

Features

- List of Class B component test requirements
- Firmware library for general Class B tests

Description

IEC 60730 is a safety standard for household appliances that addresses many aspects of both product design and operation. This standard is also referred to by other standards for safety-critical devices, for example, IEC 60335. System-wide compliance with this standard is necessary for an appliance to be certified as safe to operate.

This application note is a guide to compliance with Annex H of the standard, which regards electronic controls. Further, we supply a certified firmware library and usage examples for IAR Embedded Workbench® and Atmel® AVR® GCC. The firmware library is designed to cover the most general parts of the standard, while much depends on both the application and implementation choices.

The outline of this document is as follows. Chapter 1 presents some definitions from the IEC 60730 standard. Chapter 2 describes the requirements for Class B software. Chapter 3 presents an overview of the embedded self-tests. Chapter 4 introduces our certified library and the files that are included. Chapters 5 to 10 describe in detail the embedded self-tests in the library. Finally, Chapter 11 presents other safety features included in the Atmel AVR XMEGA®.

Table of contents

1. Definitions in Annex H of IEC 60730

1.1 Software classes

Annex H of the IEC 60730 safety standard defines three classes of control software for appliances:

- Class A – control functions which are not intended to be relied upon for the safety of the equipment (*H.2.21.1*)
- Class B – software that includes code intended to prevent hazards if a fault, other than a software fault, occurs in the appliance (*H.2.21.2*)
- Class C – software that includes code intended to prevent hazards without the use of other protective devices (*H.2.21.3*)

Software used in protective control functions are either Class B or Class C. This application note deals with Class B controls, which applies to most household appliances.

1.2 Control structures

Class B controls can be designed according to one of three defined structures (*H.11.12.2*):

- Single channel with functional test – a single channel structure in which test data is introduced to the functional unit prior to its operation (*H.2.16.5*)
- Single channel with periodic self-test – a single channel structure in which components of the control are periodically tested during operation (*H.2.16.6*)
- Dual channel without comparison – a structure which contains two mutually independent functional means to execute specified operations (*H.2.16.1*)

The term *channel* refers to the number of MCUs in the appliance. The single channel structure tends to be the preferred structure since it has the lowest cost.

A single channel structure with *functional* test means that the system is tested at the point of manufacture only. This structure can only be used if the appliance does not use any components that require periodic testing. However, periodic testing is a safer option because this allows for faults to be detected during operation of the product. Typically, the time interval between the tests must be shorter than it takes for a fault in the relevant component to cause hazard.

Dual channel without comparison is essentially a structure in which two MCUs operate independently, with different tasks, and either MCU can check that the other is operating correctly. One approach is to use one MCU strictly for supervision, rather than sharing control tasks between the two. In this case, a lower cost device can often be used as the supervisor.

This application note focuses on the single channel structure.

2. Class B requirements

For an appliance to comply with the Class B requirements, the control software must detect and handle the faults specified for the system components in Table 2-1. In addition to these tests and fault detections, the software must be properly documented to pass a certification. This includes the program sequence, control and data flow, timing, fault tree and the general design philosophy.

Table 2-1. Overview of components and faults/errors for test (*H.11.12.7*).

Component to test	Fault/error to test for
1 CPU	-
1.1 Registers	Stuck at
1.3 Program counter	Stuck at
2 Interrupt handling and execution	No/too frequent interrupts
3 Clock	Wrong frequency
4 Memory (1)	-
4.1 Invariable memory	All single bit faults
4.2 Variable memory	DC fault
4.3 Addressing	Stuck at
5 Internal data path (1)	-
5.1 Data	Stuck at
5.2 Addressing	Wrong address
6 External communication	-
6.1 Data	Hamming distance 3
6.3 Timing	Wrong point in time
7 I/O periphery	-
7.1 Digital I/O	Fault conditions specific in H.27.1 (2)
7.2.1 A/D- and D/A-converter	Fault conditions specific in H.27.1 (2)
7.2.2 Analog multiplexer	Wrong addressing
9 Custom chips	Any output outside static and dynamic functional specification

Notes: 1. These are essentially the same for AVRs since SRAM, FLASH and EEPROM are all internal.

2. This table lists various external components and indicates whether a short and/or open fault must be detected.

Several of these tests are inevitably application dependent. As an example, for I/O periphery it is required to do plausibility checking of the input/output signals. This in turn depends on both the application and its implementation. The firmware library supplied with this application note therefore cannot cover all of the requirements in Table 2-1.

3. Component tests

Acceptable fault detection measures and considerations for the individual components in Table 2-1 are explained below.

Note: The single channel structure with functional test cannot be used if functional test is not listed under acceptable measures for the components that are used in the appliance.

3.1 CPU registers – Component 1.1

Purpose of test: Detect stuck bits in the registers.

The CPU registers are the most vital part of the MCU, and must be tested since correct operation is impossible with faulty registers.

Acceptable measures for fault detection are functional test and/or periodic self-test using static memory testing or word protection with a parity check. In this library we have chosen the first method because the parity check would require a hardware implementation, which is not an option with the Atmel AVR XMEGA family. However, other methods for data redundancy could be implemented in software, for example, using two registers to keep the same value, in which case it would be convenient that the Atmel AVR CPU features 32 general-purpose registers.

3.2 Program counter – Component 1.3

Purpose of test: Detect stuck bits in the program counter.

A correctly functioning program counter is important for any software to successfully run on an MCU. However, the program counter cannot be tested directly for stuck bits since any such test will rely on the program counter to function correctly in the first place.

Acceptable measures for fault detection are functional test, periodic self-test, independent time-slot monitoring or logical monitoring of program sequence.

The preferred solution is to use the watchdog for indirect time-slot monitoring of the application. If a program counter fault occurs, the watchdog timer will either be reset at the wrong time or not reset at all, eventually causing a device reset. Since it is an integral safety feature, the XMEGA family's watchdog must be tested for proper operation in both regular and window mode before it can be relied on for catching program counter faults.

After it has been tested, the watchdog should be enabled in window mode at all times. Further, the closed period should be at least as long as the open period, that is, at least 50% of the total period.

For more information on the watchdog functionality of the XMEGA, please refer to the Atmel AVR1310 application note.

3.3 Interrupts – Component 2

Purpose of test: Verify that interrupts occur and are handled at expected rate.

Most applications rely on interrupts for their operation, and it is therefore important to verify that these occur at the time they are expected, and are handled accordingly.

Acceptable measures for fault detection include functional testing and/or time-slot monitoring of the interrupts. Time-slot monitoring is recommended since this will help detect faulty operation once the appliance is in use. Any interrupt testing will also indirectly test the XMEGA interrupt controller.

3.4 Clock – Component 3

Purpose of test: Detect unexpected deviations in system clock frequency.

For the MCU to operate correctly and with correct timing, the frequency of the system clock must be verified to be within specification.

Acceptable measures for fault detection are frequency or time-slot monitoring. For clocks based on crystal oscillators, the requirement is to detect that it is not oscillating at (sub)harmonics.

A reference clock is needed for verification of the system clock frequency. The Atmel AVR XMEGA family's Event System allows for timer/counter captures to be triggered by an external clock signal or the RTC (Real Time Counter), allowing for frequency comparison. It is also possible to use the Event System to count the timer/counters up or down.

As an extra security feature, the XMEGA family features clock failure detection for the external oscillator.

3.5 Static RAM – Components 4.2, 4.3 and 5

Purpose of test: Detect stuck bits and coupling faults in SRAM and on the data bus, and any addressing problems.

The internal SRAM is used for volatile storage of data and any faults related to this can be catastrophic for the appliance control.

Acceptable measures for fault detection include periodic testing and/or data redundancy, for example, parity bits. The latter is cumbersome without hardware implementation specific for this purpose, leaving periodic testing with, for example March algorithms as the best choice.

If the entire SRAM cannot be tested in one go, the tested memory sections must have some overlap to allow for detection of coupling faults.

3.6 Flash and EEPROM – Components 4.1

Purpose of test: Detect all single bit faults in non-volatile memory.

In all Atmel AVR microcontrollers, the application software is stored in Flash memory while EEPROM memory can be used for, for example, device-specific settings and constants. For the device to operate safely, these non-volatile memories must be checked for corruption.

Acceptable measures for fault detection are periodic self-test using a single or multiple checksums and/or single-bit data redundancy, for example, parity check in hardware. Multiple checksums is the recommended option, since this allows for one section of Flash or EEPROM to be checked at a time. The method is then to compute a checksum for the target memory range, then compare the result with a reference checksum that is stored elsewhere in non-volatile memory.

3.6.1 Note on self-programming

It is not recommended to perform self-programming of the Flash in harsh environments because, for example, power failure during writing will have unpredictable results.

If self-programming is absolutely necessary, it is recommended to do this in a boot loader that is protected by lock bits so accidental self-overwrites are impossible. In this case, the boot loader should check the application section of Flash before any code in it is executed.

All Atmel AVR XMEGA features a boot loader section in the Flash memory.

3.7 External communication – Component 6

Purpose of test: Verify correctness of transferred data, sequence and timing of communication.

Communication with external devices is an important part of many applications. This represents a potential source of faults since communication is susceptible to noise and either end of the communication line may be operating incorrectly. Consequently, steps must be taken to ensure that noise or faulty operation in one device does not cause faulty operation in another.

Acceptable measures for detecting faults in data are multi-bit data redundancy such as repetition, CRC or Hamming codes, or protocol tests. Acceptable measures for detecting faults in timing are time-slot monitoring or scheduled transmissions. Acceptable measures for detecting faults in communication sequence are the same as those for timing, but also include logical monitoring.

In short, a state machine based communications driver with time-outs, scheduling and transfer redundancy should be used.

3.8 Input/output periphery – Component 7

Purpose of test: Verify that input/output is as expected, and that signals are correctly routed.

Analog and/or digital I/O is needed in all applications, and can be used to detect faulty operation in either the periphery or in the MCU itself.

The acceptable measure for fault detection is plausibility checking, which means that the software verifies that it is getting the expected input and giving the desired output at any time.

4. Class B library for Atmel AVR XMEGA

The purpose of the library is to simplify the design of safe and reliable applications based on the XMEGA family of microcontrollers. Further, this library has been certified to simplify the design and certification processes for our customers.

This library has been developed to be flexible enough to embed the self-test modules in many different applications. The final user is then responsible to configure and use it in a way that the application complies with IEC 60730 Class B.

We have added support for the AVR GCC compiler, which is included in Atmel Studio 6, and for IAR™. A number of examples have been included in order to show how the self-diagnostic routines can be embedded in a user application.

The source code of the library has been prepared for automatic documentation generation with Doxygen (www.doxygen.org). This documentation complements the information provided in this document.

4.1 Error handling

In order for the test modules to be as general as possible, we have defined a number of error handlers that can be configured by the user. Errors have been divided into critical and non-critical, and have a default value for the error handlers.

Critical errors are those that cannot be handled, for example when the register that should return the result of the register self-diagnostic routine has a stuck bit. Critical errors hang the CPU by default, that is, the CPU is left executing and infinite loop. In principle this should lead to a Watchdog reset, and the actions to take after a system reset issued by the Watchdog Timer (hereafter referred to as WDT) can be configured as well.

Non-critical errors are those that, even if they prevent the MCU from working correctly, they can still be handled by the program. For example if one register is stuck (other than the one that returns the result of the test or the stack registers) the program could still take some actions to leave the system on a safe state. Non-critical errors set a global error flag by default. This flag is called *classb_error* and it can be used by the main application to leave the system on a safe state. This was the approach followed in the examples.

4.2 Source files

Common files:

- *avr_compiler.h*: this file contains some general definitions and macros to ensure code compatibility with IAR and GCC
- *classb_rtc_common.c/h*: this is an optional driver for the real time counter
- *error_handler.h*: macros and definitions related to error handlers and configurable actions

CPU registers:

- *classb_cpu.h*: settings, definitions and macros for the test
- *classb_cpu_gcc/iar.c*: implementation of the self-diagnostic routine
- *UserApplication.c*: an example application where the test is embedded

CPU program counter (Watchdog Timer test):

- *classb_wdt.c/h*: code and header file for the test
- *UserApplication.c*: an example application where the test is embedded

Interrupt manager:

- *classb_interrupt_monitor.c/h*: code and header file for the test
- *UserApplication.c*: an example application where the test is embedded

System Clock:

- *classb_freq.c/h*: code and header file for the test
- *UserApplication.c*: an example application where the test is embedded

Variable memory (March test for SRAM):

- *classb_sram*: code and header file for the test
- *UserApplication.c*: an example application where the test is embedded

Invariable memory (CRC test for Flash and EEPROM):

- *classb_crc.h*: header file with general configuration settings
- *classb_crc_hw.c/h*: code and header file for the hardware implementation
- *classb_crc_sw.c/h*: code and header file for the software implementation
- *UserApplication.c*: an example application where the test is embedded

Analog I/O:

- *classb_analog.c/h*: code and header file for the test
- *UserApplication.c*: an example application where the test is embedded

5. Registers

In our self-diagnostic routine the CPU registers are tested for stuck bits and some coupling faults. The basic algorithm works as follows: In the first step the register is set in a known value. In the second step register bits are forced to change state. In the third step the content of the register is verified. After that, register bits are forced to change value again and, finally, the content of the register is verified again. In both cases, if the content of the register is incorrect, the error variable is set.

We will test two types of registers: register file (R0 to R31) and I/O registers. These are located in separate memory spaces and are accessed with different instructions. In addition, there are a number of issues that have to be considered. Firstly, some registers cannot be manipulated destructively because they could be in use. The type of register and the specific compiler determines which registers must be preserved. We have used one auxiliary register to store the value of these registers. Secondly, registers from R0 to R15 and I/O registers cannot use immediate instructions, for example, LDI and CPI. Therefore, we have chosen to load and compare through another auxiliary register.

The registers are tested following this order (GCC compiler implementation):

1. Return value register: R24.
2. Auxiliary registers: R31, R30.
3. Stack pointer: SPL, SPH.
4. Register file: R29 to R25 and R23 to R0.
5. Extended addressing registers: RAMPD/X/Y/Z, EIND.
6. Status register: SREG (except interrupt flag).

In the first three steps we test those registers that are critical for the self-diagnostic routine:

- Return value register: used to deliver the result of this self-test
- Auxiliary register 1: used to store the value of those registers that must be preserved
- Auxiliary register 2: used to copy values when testing registers that cannot be used with the LDI or CPI instruction (R0 to R15)
- Stack pointer: necessary to return to the main program

The auxiliary registers are critical because they are required to test the stack pointer. If there is an error in these registers the device will by default stay executing an infinite loop. However, it is possible to call the error handler instead.

In the following steps the rest of the register file and the I/O registers are tested. If there should be an error, the self-diagnostic routine would call the error handler and return the value of the test. However, it is possible to configure the same behavior as in the critical registers, that is, the device would hang if there should be a fault in any register.

The self-diagnostic routine is coded with inline assembler to offer more flexibility for register manipulation. In order to simplify the assembly coding we defined a number of macros.

The example application lights an LED and sets up a switch that can interrupt the main program, which is basically an infinite loop. In the switch interrupt routine, the LED is toggled and the register test is called. In principle the CPU registers should be working correctly and, therefore, the register test will not find an error. The switch interrupt will return then to the main program, which continues execution normally: the global error is not set and the application stays in the infinite loop. The main program can then be interrupted again at anytime by pressing the button.

In order to simulate an error, we can set breakpoints in the self-diagnostic routine. The application can then be started and, after stopping in a breakpoint, the content of a register can be modified to simulate a stuck bit. The execution can then be resumed and we will find that the self-diagnostic routine has detected the error. Depending on the register that is modified and the value of the constants CLASSB_ERROR_CRIT and CLASSB_ERROR_NON_CRIT we will either find that the global error variable has been set to one (which leads to an exit in the endless loop and switching the LED off) or that the CPU is stuck in an endless loop inside the self-diagnostic routine. In any case, the application will not respond to pressing a button any more.

6. Program counter

The watchdog timer WDT is a system function for monitoring correct program operation that allows recovering from error situations such as runaway or deadlocked code. The WDT is a timer clocked independently from the CPU, it is configured to a predefined timeout period and it is constantly running when enabled.

If the WDT is not reset within the timeout period, it will issue a microcontroller reset. This can be done by executing the WDT reset instruction from the application code. In addition, the WDT in the Atmel AVR XMEGA has a window mode. This makes it possible to define a time slot or window inside the total timeout period during which the WDT must be reset. If the WDT is reset outside this window, either too early or too late, a system reset will be issued. Compared to the normal mode, this can also catch situations where a code error causes constant execution of the WDT reset instruction. Therefore, it is required that Class B software uses this windowed mode and that the closed period that is at least 50% of the total period.

The WDT will run in active mode and all sleep modes, if enabled. It is asynchronous, running from a CPU-independent clock source, and it will continue to operate and issue a system reset even if the main clocks fails.

There is a configuration change protection mechanism in XMEGA that ensures that WDT settings cannot be changed by accident. For increased safety, a fuse for locking the WDT settings is also available.

Since the WDT is an integral safety feature in the Atmel AVR XMEGA family, we have designed a self-diagnostic routine that tests this module in normal and window mode. This is executed after reset in the pre-init section of the application that is, before the main function. A flow diagram of the test is shown in Figure 6-1.

The self-diagnostic routine basically makes sure that:

- system reset is issued after WDT timeout
- WDT can be reset
- device is reset upon untimely WDT reset in window mode

The flow diagram shows that the device is reset a number of times during the WDT test. Therefore, a SRAM variable and the reset flags of the device are used by the self-diagnostic routine to keep track of the test phase. Further, the user can configure what to do for brown-out or software reset, or how to process a reset caused by the watchdog when the test is in the "ok" state.

Our self-diagnostic routine uses the real time counter (RTC) in order to check the timing of the WDT oscillator. The RTC has a clock source independent from the CPU and the WDT. Both modules have oscillators working at the nominal frequency of 32.768kHz. However, the WDT oscillator is optimized for low power consumption, at the cost of reduced accuracy. The oscillator of the RTC can be assumed accurate. The RTC is used to estimate the period of the WDT and the program checks that this estimate is within the interval (T/2, 3T/2), where T is the nominal period of the WDT.

Note: The RTC is implicitly tested by this self-diagnostic routine; if the difference in frequency between RTC and WDT is more than 50%, the error state is set.

The expected (error-free) execution flow is as follows:

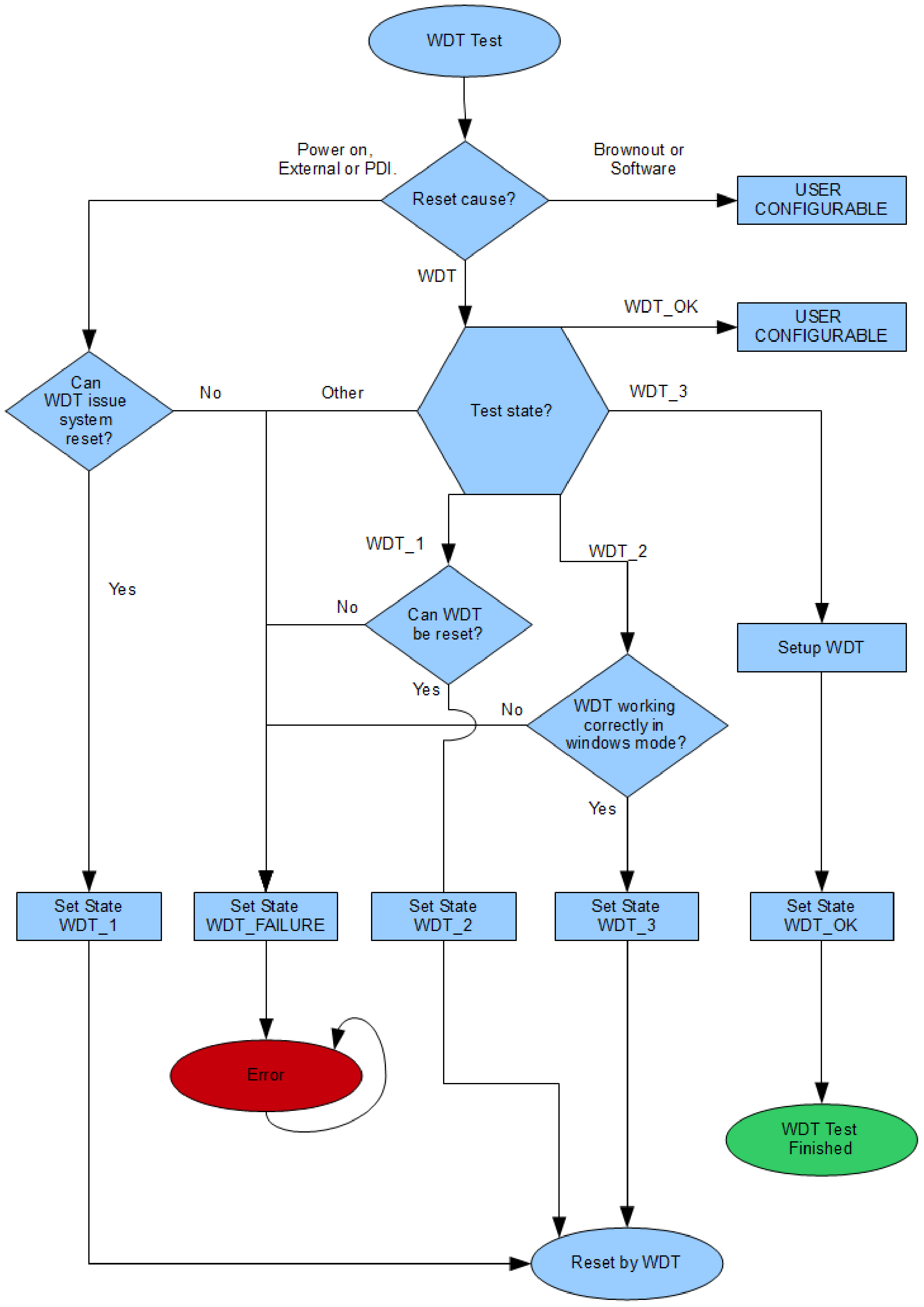
1. After power-on or external reset, check that WDT can issue a system reset. Set test state 1 and system reset by WDT.
2. Check that WDT can be reset. Set test state 2 and WDT issues system reset.
3. Check that window mode works correctly. Set test state 3 and WDT issues system reset.
4. Set up WDT according to settings. Set test state "ok" and continue to the main function.

The first step is to make sure that the WDT can issue a system reset. This is basically done by setting up the WDT as specified in the configuration file and waiting until it issues a reset. In addition, the RTC is used to estimate the watchdog period, which is required in later phases of the test. This is done by setting up the RTC with a small period (approx. 3ms) and counting the number of RTC-periods until system reset. There is a configurable maximum time to wait and, after this time, the program sets the error state.

The second step is to make sure that the WDT can be reset and to check the timing of the WDT. The error state is set temporarily and then it is checked that the estimated WDT period is higher than a minimum. Checking that the difference in frequency between the WDT and the RTC is within an interval gives us confidence that both modules are working as expected. Next, the WDT is set up and we use the RTC to wait for approximately ¾ of the WDT period, which was estimated in the previous step. This checks that the WDT does not expire earlier than expected. After this the WDT is reset and the program waits again ¾ of the period. Note that if there was any problem in the mechanism that resets the WDT, there would be a system reset while the program is waiting for the second time. This is because the total waiting time would be 1.5 times of the estimated WDT period. In addition, this early system reset would leave the test in the error state. Further, assuming that the WDT was reset correctly, a system reset should come in approximately ¼ of the period. The next test state is then set and the program wait for the system reset.

The third step checks that the WDT works correctly in window mode. This basically consists on setting up the WDT and the next state of the test and doing an early reset of the WDT. Given that the time constraints are not followed, the WDT should issue a system reset. Therefore, after the untimely WDT reset, the program waits for the system reset during ¼ of the WDT period. If there was any problem with the window, the device would not be reset and the program would set the error state.

Figure 6-1. Watchdog test.



In the fourth and final step the program simply sets up the WDT in window mode and the test in the ok state. Note that after this the main application is responsible to reset the WDT according to the configured settings.

If the test was in the error state, a user-configurable error handler would be called. By default the device would simply hang, because a working WDT is critical for a reliable software application.

The counter and the test state variables are declared so that the compiler does not initialize them after reset. This allows us to use them across resets. The Atmel AVR XMEGA has a register that stores the reset cause, which is used to decide whether it is the first iteration of the test.

The demo application sets up an interrupt for SW0, switches an LED *on* and then stays in a loop where the WDT is reset as long as the variable *classb_error* is not set. If this variable is set then the LED is turned *off* and the application ends.

When the SW0 button is pressed, the program stops resetting the WDT, which leads to a WDT timeout and, therefore, a system reset. This "unexpected" reset should be caught by the self-diagnostic routine, which in this demo is configured to set the variable *classb_error*, set up the WDT and continue to the main application. Therefore, after pressing the button, the LED will be switched *off* and the application will not be responsive.

6.2 How the self-diagnostic routine can be tested

The first step of the self-diagnostic routine will set the error state unless there is a system reset. A problem in the WDT that prevents it from being able to issue system resets could be replicated simply by not starting the WDT. The error state would be set and the device would hang.

Frequency failure of the WDT or the RTC could be simulated by setting a break point and modifying the value of the *rtc_count* variable, so that it is out of the interval of confidence.

Failure in the WDT reset mechanism could be simulated by removing the line where the WDT is reset. Given the structure of the program, the second step in the self-diagnostic routine will set the error state unless the WDT follows the time constraints that are imposed.

Failure in the WDT window mode could be simulated by removing the setup of that mode. The code would then reset the WDT and wait for ¼ of the WDT period. At that point the WDT period would be greater or equal to the estimated WDT period (the total timeout period is the sum of the open and close periods). Therefore, the device would not be reset before setting the error state.

The configured actions for a system issued by the WDT can be tested through the demo application. Pressing the button will lead to the WDT expiring and then the self-diagnostic routine would execute the configured actions, which in this case are to set up the WDT and set the variable *classb_error*.

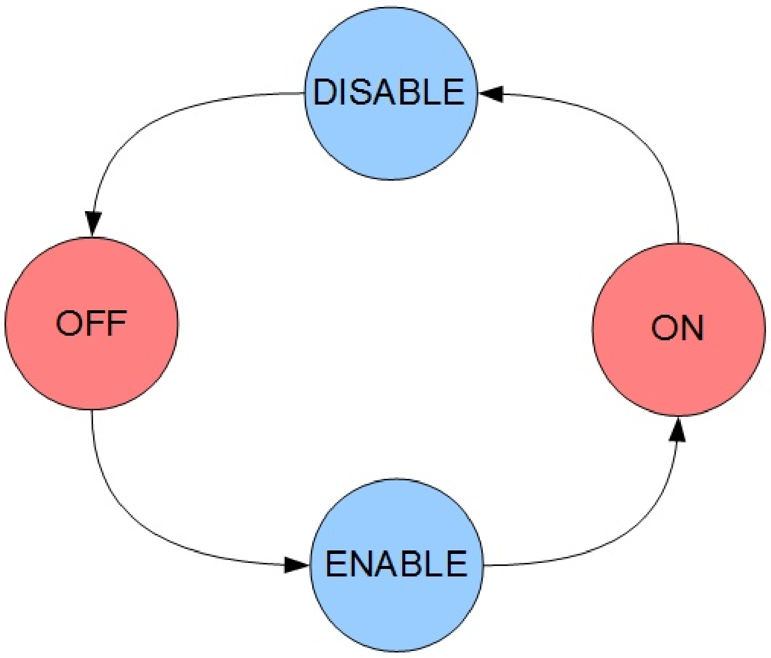
7. Interrupt handling

An interrupt signals a change of state in peripherals, and this can be used to alter program execution. Embedded applications use interrupts, for example, to respond to events in real-time. Therefore, it is important that a system can detect errors in the interrupt functionality. Specifically, Class B applications should be able to detect whether an interrupt is not being executed as often as expected (or not at all).

The chosen method is time-slot monitoring with the Real Time Counter (RTC, clock independent from the CPU clock). In short, any interrupt to be monitored has to increase a counter every time it is executed. Further, the RTC generates an interrupt periodically where the interrupts counters are checked. If any counter should be out of an interrupt-specific configurable range, the error handler would be called.

The interrupt monitor checks the frequency of those interrupts that are both registered and activated. Registering an interrupt means to give the interrupt monitor information the following information on the interrupt to check: identifier, expected frequency and deviation tolerance. The interrupt monitor creates a data structure with this information. Activating an interrupt means to tell the monitor that it should start checking a registered interrupt. In order to activate the monitoring of interrupts dynamically, we have implemented a state variable related to each registered interrupt. The possible transitions between states are shown in Figure 7-1.

Figure 7-1. Interrupt states.



The default state for any interrupt is OFF and in this state the interrupt manager does not check the frequency of the interrupt. The state of the interrupt should be set to ENABLE when the main application wants the monitor to start checking the interrupt frequency. The next time the interrupt monitor is executed it will then change the interrupt state to ON. This ensures that the interrupt counter is synchronized with the interrupt monitor, that is, the interrupt counter starts being increased exactly after a period of the interrupt monitor. Similarly, if the main application decides that an interrupt should not be monitored anymore, the interrupt state should then be set to DISABLE. The interrupt monitor will change the state to OFF the next time it is executed.

Note: There should only be one enable/disable request per RTC period.

The implemented interrupt monitor has two features that further increase the robustness of the main application. The first one is an interrupt counter that increases only if the interrupt is ON and, therefore, the counter should be zero while an interrupt is OFF. This is checked by the interrupt counter for all registered interrupts and otherwise the error handler is called. The second feature is that enabling an interrupt that is ON or disabling an interrupt that is OFF will call the error handler if the constant CLASSB_STRICT is defined.

The RTC calls the interrupt monitor function periodically. All registered interrupts are processed according to their state. Active interrupts are checked for their frequency. If there is no error, the interrupt monitor resets the counter. Otherwise the error handler is called and the monitor ends immediately (this is configurable). The counter of inactive interrupts is compared to zero for coherence. If the main application has requested to activate an interrupt, its state is set to ON, and vice versa. Note that in the latter case the interrupt counter is reset.

In order to monitor an interrupt, the following steps should be followed:

1. The interrupt should be declared in *classb_int_identifiers* by giving an identifier to the interrupt.
2. The main application has to register the interrupt by calling *classb_intmon_reg_int()*. A structure is then created for the interrupt.
3. The RTC has to be set up to generate an interrupt periodically and to call back the interrupt monitor.
4. The interrupts that have to be monitored should call *classb_intmon_increase()* on each execution.
5. The main application has to request that the monitor starts checking the interrupt. This is done by changing the interrupt state to ENABLE with *classb_intmon_set_state()*.
6. If at some point an interrupt should not be monitored any more, the main application can change the state to DISABLE.

In the example application a Timer/Counter (TC) is set up to generate an overflow interrupt periodically, which is checked by the monitor. In addition, the application sets up two switch interrupts. The first one changes the frequency of the TC interrupt. The second one deactivates the interrupt in the monitor. Further, an LED is switched on to show that the application is working correctly. The application stays on a loop with the LED on as long as the buttons are not pressed. If the first button is pressed first, the frequency of the TC interrupt will change and, therefore, the monitor should set the error flag. That will result on the main application leaving the loop and switching the LED off. However, the second button can be pressed in order to deactivate the TC interrupt in the monitor. In this case, pressing the first button still leads to a change in the frequency of the interrupt, but the monitor will not generate an error.

The interrupt monitor relies on a working RTC. The latter supports other components of the library (WDT and CPU frequency tests) and it is checked in the related software modules. Therefore, it can be assumed that there is no failure in the RTC. However, in order to increase the reliability of the application, registered interrupts could have a maximum value for the counter, which could be tested within the interrupt. If the counter reached this value, it could be assumed that the interrupt monitor was not working.

The monitor can be tested as follows. Firstly, an interrupt could be set up and then its frequency could be modified in order to generate an error. This is implemented in the example application. Secondly, the counter of an interrupt could be modified while debugging. This could be done both for active and inactive interrupts. Further, if the symbol CLASSB_STRICT is defined, the program could activate or deactivate an interrupt twice, which should lead to an error.

8. System clock

The self-diagnostic routine has been implemented using the Real Time Counter (RTC) and a Timer/Counter (TC). We have chosen a TC because this peripheral has the same clock domain as the CPU. However, the RTC can be clocked from an independent clock source. For example the CPU could be clocked from the internal 2MHz oscillator and the RTC from the internal 32.768kHz oscillator.

The RTC is set up to generate an interrupt periodically. In this interrupt, the count in the TC is compared to its expected value and, if it were not within a configurable range, the error handler would be called.

In the TC overflow interrupt a 16-bit overflow counter variable is increased. This is because the TC will run in most cases at a much higher frequency than the RTC and, therefore, it is necessary to have a 32-bit counter. In addition, the error handler would be called if the overflow counter should be higher than a configurable threshold. Given that the overflow counter is cleared in the RTC interrupt, an overflow count higher than the threshold would mean that the RTC interrupt did not occur as often as expected.

Considering these two measures (check of the RTC against the TC and vice versa), the self-test module would detect a failure in either the RTC or the TC. The risk of an undetected error is reduced to the scenario where there were failures in both the RTC and the TC but the ratio of their frequencies was correct.

The RTC interrupt calls back a function where, firstly, the value of the overflow counter is appended to the TC so that the effective counter has 32 bits. After that the 32-bit count is compared to the reference. If the difference were higher than expected, the error handler would be called. After that the overflow counter and the TC count are reset.

The example application is similar to previous examples. By default an XMEGA device runs from an internal 2Mhz oscillator, which is the system frequency that is considered when setting up the parameters of the self-diagnostic routine. If the button is pressed, the system clock is set to run from a 32MHz oscillator. This will make the test fail and the LED will be turned off.

The implementation of the system frequency self-test is flexible so that it is possible to use different RTC and TC configurations. The RTC frequency and the RTC interrupt period can be configured in the files related to the RTC. Note that different RTC setups will be compatible with our self-diagnostic routine as long as the clock source is independent from the CPU and the constants CLASSB_RTC_INT_PERIOD and CLASSB_RTC_FREQ are defined according to the RTC settings. Further, it is also possible to configure the TC module, the prescaler, the tolerance for the test (in percent) and the system frequency. The latter has to correspond to the actual system frequency set by the main application, that is, the self-diagnostic routine will not change the clock system according to that setting.

There are a number of methods to test this self-diagnostic routine. Firstly, the example application changes the system frequency after pushing the button. Secondly, the system frequency constant F_CPU could be modified so that it does not match the real system frequency, which is 2MHz. This can be combined with modification in the tolerance value. Thirdly, in order to simulate problems in the RTC or the TC the setup functions could be commented.

9. Memory

In this chapter we describe the standard requirements for memories and the self-diagnostic tests that we have implemented. Section 9.1 refers to invariable memories, that is, internal Flash and EEPROM in the Atmel AVR XMEGA family. Section 9.2 refers to variable memory, which is the internal SRAM.

9.1 Invariable memory

A cyclic redundancy check (CRC) is an error detection technique test algorithm used to find accidental errors in data. This method is commonly used to determine the correctness of data transmissions and data present in the data and program memories.

The CRC algorithm processes an input data stream or block of data and generates an output checksum, which can be used to detect errors later. Two common methods to do this consist on the following:

- Compute a first checksum of the data and store it. In order to detect errors a second checksum can be computed and compared with the previous one. If they are different, there is an error
- Compute a first checksum of the data and append it to the data section. In order to detect errors, a second checksum can be computed for the new data section. The resulting checksum should be a fixed implementation-specific value, otherwise there is an error

Typically, an n-bit CRC applied to a data block of arbitrary length will detect any single error burst not longer than n bits and will detect the fraction 1-2-n of all longer error bursts. If there is an error in the data, the application should take a corrective action, for example, requesting the data to be sent again or simply not using the incorrect data.

We have developed self-diagnostic modules based on hardware or software implementations for CRC. Further, two commonly used CRC standards are supported:

- 16-bit CRC CCITT
- 32-bit CRC IEEE® 802.3

The software implementation can be used by all XMEGA devices. In this case the CPU reads the data and computes the CRC checksum. It is possible to choose between two software implementations:

- Lookup table: this uses a CRC lookup table to speed up the computations. The lookup table requires 512 (for 16 bit) or 1024 (for 32 bit) bytes of flash memory
- Direct computation: this calculates the checksum for each byte using a polynomial division. This version occupies no space in the flash memory, but is slower than the lookup table method

In the software implementations, the CRC32-polynomial used is 0xEDB88320, the initial remainder is 0xFFFFFFFF, and the generated checksum is bit-reversed and complemented. The CCITT polynomial used is 0x1021, with 0x0000 as initial remainder. In this case the checksum is neither bit reversed nor complemented.

The hardware implementation relies on a driver to interface the CRC module included in the new XMEGA AU devices. This means that other XMEGA devices have to use the software implementation. In this case, even if the CRC hardware module computes the checksum, the CPU can still be required to read the data and feed it to the module. For 32-bit CRC computation in flash memory, the whole process can be run without support from the CPU. However, for the 16-bit computation the CPU has to read from the Flash memory and transfer data to the CRC module. For CRC computations of data in SRAM, EEPROM or peripherals, the CPU has to transfer data to the CRC module unless a DMA transaction is set up. The CRC-CCITT polynomial is 0x1021 and IEEE 802.3 one is 0x04C11DB7. The setup for the initial values and the final checksum is the same as described above.

The following functions are available to compute checksums for data stored in the EEPROM:

- CLASSB_CRC16_EEPROM_SW
- CLASSB_CRC16_EEPROM_HW

- CLASSB_CRC16_EEPROM_SW
- CLASSB_CRC32_EEPROM_HW

Similarly, the following functions are available to compute checksums for data stored in the Flash:

- CLASSB_CRC16_FLASH_SW
- CLASSB_CRC16_FLASH_HW
- CLASSB_CRC16_FLASH_SW
- CLASSB_CRC32_FLASH_HW

Note: The hardware and software implementations have been configured so that equivalent CRC algorithms produce the same checksum. However, there are significant differences in processing time.

As an example, we describe the function that computes 32-bit CRC for Flash using the hardware module. Firstly, the function receives the information needed: type of computation, start address, number of bytes of the data and the pointer to the "correct" checksum stored in the EEPROM. Secondly, the driver of the CRC module is used to compute the "new" checksum. Finally, the checksums are compared and the error handler is called if there should be any error.

In the example application the pre-calculated checksums and some data are stored in the EEPROM memory. Then, the program lights up a LED that signals correct function of the system and sets up a button to produce an interrupt. After that the program stays in a loop as long as the global error flag is not set. If the button is pressed the Flash and the application section in Flash and the data stored in EEPROM are checked. If there were errors, the error flag would be set and the main application would switch off the LED. In normal conditions, the content of the application section in the Flash and the data stored in the EEPROM will be correct. This means that pressing the button will not lead to the error flag being set and then the program will return to the main loop and the LED stays on. The button can be pressed again and again with the same results.

In order to check that the self-diagnostic tests are working correctly, the data stored in the EEPROM or the content of the application section in the Flash could be modified. For convenience, the example application has two symbols that can be defined to change the data stored in the EEPROM or to compile an extra loop after the line that switches off the LED. In this case, the LED will be on until the button is pressed. After that the CRC test will fail because the data has changed and the error flag will be set. This will lead to the main application leaving the loop and the LED being switched off.

Pressing the button once more will have no effect. Finally, in order to check that all equivalent CRC computations lead to the same checksum, it is possible to call them and compare the results. Note that the algorithm for the software implementation (lookup or direct computation) is chosen by a setting in the corresponding header file and only one algorithm can be called during one execution. Since choosing one algorithm or the other will modify the content in Flash, the checksums across different executions for EEPROM should be compared instead.

9.2 Variable memory

The specific March algorithm implemented is known as March X test, which can be described as follows:

Algorithm 1. March X test.

$$\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)$$

The first phase is to write a 0 to all memory locations, in any order. The second phase consists of three operations that are performed on each bit, starting with the lowest address:

- Read a bit and verify that it is 0. If it is 1, a fault has occurred
- Write 1 to its location
- Repeat for the next bit

The third phase also consists of three operations that are done in the opposite order of addresses (with respect to the second phase):

- Read a bit and verify that it is 1. If it is 0, a fault has occurred
- Write 0 to its location
- Repeat for the next bit

The fourth and final phase consists in verifying that all bits are 0, in any order. Note that the actual address order used in phases two and three does not matter as long as they are done in the exact reverse order. Further, this test is equivalent to the well-known March C test where steps 3 and 4 are skipped.

March X can detect the following faults:

- Address decoder
- Single cell faults: stuck-at, transition or data retention faults
- Faults between memory cells: some coupling faults (CFs)

The March test that we have described is defined for bit-oriented memories (BOMs). However, the SRAM in XMEGA is a word-oriented memory (WOM). Replacing $r0$, $r1$, $w0$ and $w1$ by, respectively, r^{D} , $r^{\bar{D}}$, w^{D} and $w^{\bar{D}}$, where D can be any data background, the BOM march test is converted to WOM march test that covers inter-word CFs. Specifically, in our implementation we have chosen the data background D=0x00. In order to cover intra-word CFs it is possible to concatenate an intra-word March test with different data backgrounds.

It is important to consider the physical location of bits in a row of the memory cell array. In the case of XMEGA the bits from one word are interleaved, that is, physically separated by bits from other words. Therefore, the probability of CFs between cells from different words is diminished. Some sources do not consider intra-word CFs for interleaved memories, probably because in some fault models the aggressor cell must be physically adjacent to the victim cell. In addition, the inter-word march test covers some intra-word CFs, for example, intra-word inversion CFs. Taking these factors into consideration, the inter-word part of the test is probably enough to fulfill the requirement of the standard. However, the proposed self-diagnostic routine can be configured to append an additional march element with the background sequence {0x55, 0xAA, 0x33, 0xCC, 0x0F, 0xF0}. This will add coverage for the intra-word state CFs considered in the unrestricted intra-word CFs model.

In order to speed up the test, the memory is divided into a configurable number of sections that are tested in turns. The simplest behavior of the test is when there is no overlap between memory sections. In this case all sections have the same size, except possibly the last one. The first memory section (referred to as the buffer) is reserved: it is used by the test to store the content of the other sections while they are being tested. This is necessary given that the implemented march test is destructive.

Given that the March X algorithm is run on one memory section at a time, there is a user-configurable overlap between memory sections that will decrease the probability that an inter-word CFs is undetected. Every time a memory section is tested, a part of the previous section is tested as well. Note that this does not apply to the buffer, since it is the first section. The size of the buffer needs to be expanded with respect to the previous case (the size of the second section is decreased correspondingly).

An example application that shows how the memory test can be embedded in an application is included. Basically, a LED that signals correct behavior of the system is lighted and then the program stays on a loop where the SRAM memory is tested as long as there are no errors.

The self-diagnostic routine can be tested as follows. Firstly, a number of breakpoints can be set in the function that implement the March X algorithm. Secondly, the content of some memory locations can be modified to model different types of inter-word coupling faults. The test module will then set the error flag, which leads to the application exiting the main loop and the LED being switched off.

9.3 Further coverage

Considering that SRAM, flash and EEPROM are internal memories in XMEGA, the previously described self-diagnostic routines cover the specifications of the standard for memory addressing and internal data path (subcomponent 4.3 and component 5. in Table H.11.12.7).

10. Analog I/O

A plausibility check is done to make sure that ADC and DAC work as expected. The test consists basically of the following steps:

1. The DAC is configured to use 1V as a reference and to output five values (0%, 25%, 50%, 75% and 100% of the scale).
2. The ADC is configured to use internal inputs and the same reference as the ADC.
3. The multiplexer in the ADC is set up so that the ADC reads from the DAC for each generated value.
4. The expected values are set according to the ADC configuration and the threshold according to the ADC characterization.
5. If the read values should deviate from the expected values more than the threshold, the test would call the error handler. This is set on the error handling header file.

We have included an example application that shows how the analog I/O test can be embedded in an application. A LED that signals correct behavior of the system is lit and then the program stays on a loop as long as there are no errors. When a button is pressed, the ADC and DAC are tested and the program returns to the main loop.

There are different methods to test the self-diagnostic routine. Firstly, the value written to the ADC or the expected value of the ADC could be modified, which would lead to the ADC read value being wrong. Alternatively the threshold could be decreased so that the read measure is almost certainly out of range. In all these cases the test module would set the error flag, which would lead to the application exiting the main loop and the LED being switched off. Secondly, failure in the modules could also be simulated by clearing the enable bits of the DAC and ADC modules (this would disable the modules). This could be done by setting a breakpoint inside the self-diagnostic routine and clearing the bits manually. Further, this would lead to the self-test being stuck waiting for the end of conversion, which would ultimately lead to the WDT issuing a system reset.

11. Additional safety features of XMEGA

The Atmel AVR XMEGA family has some additional features that can be used for increased safety, but do not directly address any of the Class B requirements:

- Configuration Change Protection (CCP) prevents change of certain I/O registers and writing/reading to nonvolatile memory if a timed code sequence is not followed
- Fuses can be used to lock some configuration settings and sections of Flash from overwrites
- Fuses can easily be read and verified by the application software
- Brown-Out Detection (BOD), Power-On Reset (POR) and Spike Detection (SD) can prevent code execution when power fails, fluctuates too much, etc.
- Internal temperature sensor can be used to detect wrong operating conditions or hardware faults
- The clock system checks for oscillator stability before switching clock sources
- XOSC failure detection ensures that the device does not stop operating even if the external oscillator does

12. Acknowledgements

In this document "IEC 60730 standard" and all other definitions and sections from this standard refer to:

IEC 60730-1 ed.3.2, copyright© 2007 IEC Geneva, Switzerland, www.iec.ch.

The author thanks the International Electrotechnical Commission (IEC) for permission to reproduce information from its International Publication IEC 60730-1 ed.3.2 (2007).

All such extracts are copyright of IEC, Geneva, Switzerland. All rights reserved. Further information on the IEC is available from www.iec.ch.

IEC has no responsibility for the placement and context in which the extracts and contents are reproduced by the author, nor is IEC in any way responsible for the other content or accuracy therein.

13. References and suggested literature

- "IEC 60730-1: Automatic electrical controls for household and similar use", International Electrotechnical Commission, Ed. 3.2, 2007-03
- "Essentials of electronic testing for digital, memory, and mixed-signal VLSI" by Michael Lee Bushnell and Vishwani D. Agrawal
- "A Designer's Guide to Built-In Self-Test", C. E. Stroud, Kluwer Academic Publishers, 2002
- Atmel AVR040: EMC Design Considerations
- Atmel AVR042: AVR Hardware Design Considerations
- Atmel AVR1310: Using the XMEGA Watchdog Timer

14. Revision history

Doc. Rev.	Date	Comments
42008A	06/2012	Initial document release

Enabling Unlimited Possibilities®

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan G.K.
16F Shin-Osaki Kangyo Building
1-6-4 Osaki
Shinagawa-ku, Tokyo 141-0032
JAPAN
Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370