AVR1305: XMEGA Interrupts and the Programmable Multi-level Interrupt Controller

Features

- **3 interrupt levels**
- **Round-robin scheduling for low-level interrupts**
- **Programmable priority for low-level interrupts**

1 Introduction

Microcontrollers use interrupts to prioritize between the tasks and to ensure that certain peripheral modules are serviced fast. Further, interrupts can be used to reduce the power consumption of a microcontroller, so that the device is in low power mode until a certain interrupt causing condition occurs.

The XMEGA™ Interrupt mechanisms and the Programmable Multi-level Interrupt Controller (PMIC) are described in this application note. The application note also offers a C code example that shows how the PMIC can be accessed.

8-bit AVR® Microcontrollers

Application Note

Rev. 8043A-AVR-02/08

2 Theory of Operation

Most (peripheral) modules have status flags, or *interrupt flags*, that can trigger execution of an interrupt. An *interrupt* is when a module signals to the AVR® CPU that the flow of the program code execution should be interrupted and a specific Interrupt Service Routine (ISR) program code should be executed. An interrupt is generated if all of the following requirements are meet:

- The condition that sets the interrupt flag is fulfilled.
- The specific interrupt is enabled in the module.
- The *interrupt level* is enabled in the *Interrupt Controller.*
- Global interrupts are enabled for the CPU.

The execution order of interrupts is determined first by their *Interrupt Level* and then their *Interrupt Priority*. Interrupt Level is specified individually for the Interrupts in the module where they originate, while the Interrupt Priority is determined by their fixed *Interrupt Vector Address.* Alternatively, for low-level interrupts, the Interrupt Priority can be based on a *Round-robin* scheme controlled by the Interrupt Controller.

The Non-Maskable Interrupt (NMI) is a special interrupt that cannot be disabled, and is used to for system critical interrupts. NMI is used for critical failures such as a failure in the crystal oscillator.

2.1 Interrupt Level – enabling a module interrupt

When a peripheral module needs to use an interrupt, it sets the Interrupt Level of the corresponding interrupt to something else than “off”. An interrupt can be given one of three levels: low, medium and high. The bits controlling the enabling and level of an interrupt are typically located in the INTCTRL registers.

When having multiple interrupt levels one should understand that when executing an interrupt with a given level it is interruptible by an interrupt with higher level. This means that an interrupt with low interrupt level can be interrupted by an interrupt with medium or high level. Evidently, nested interrupts can occur and one should take this into considerations when assessing the stack depth required by the firmware.

Also, one has to consider protection of access to registers and variables that are more than 8 bits wide. Since the AVR data bus is 8-bit wide and access to multi-byte variables requires execution of more than one instruction, it is possible that an interrupt can occur between accessing the first and the second byte of a 16-bit (or 32-bit variable). If a 16-bit variable used in the main loop of the code is modified by an interrupt, the variable can be corrupted. The same situation applies if an interrupt is interrupted by another interrupt with higher level. Corruption can be avoided in several ways: One method is to ensure that a variable that can be modified by ISRs is accessed in an atomic operation. This can by achieved through global disabling of interrupts while accessing the variable. Another method, which applies when accessing 16-bit registers (or rather, two 8-bit registers that together represent a 16-bit value), is to preserve and restore the temporary register that is used by hardware when accessing these registers. This method can be relevant e.g. when accessing the 16-bit timer registers. More information about temporary registers and 16-bit access to registers can be found in the datasheet.

2.2 Non Maskable Interrupt (NMI)

The XMEGA A1 device has one NMI source, the Crystal Oscillator Failure NMI.

If a crystal is used as system clock, the clock from the crystal oscillator is monitored. If for some reason the crystal clock stops (e.g. due to physical damage to the crystal) an NMI is generated and the internal 2 MHz RC oscillator takes over as system clock.

Since an NMI interrupt by definition cannot be disabled, there is no available mechanism to disable the NMI from software once enabled. The only way to disable the NMI is to reset the MCU, which will bring back the initial state where the NMI is disabled. Refer to application note AVR1003 covering the XMEGA Clock System for more information about crystal failure detection and the clock system in general.

2.3 Enabling Interrupt Levels in PMIC

Each interrupt level can be enabled individually. This can e.g. be desired in cases where atomic operations, like the above-mentioned examples of reading 16-bit registers, should not block high level interrupts. Consider that only low level interrupts writes to the variable that should be protected, there is no reason to block medium and high level interrupts, since they do not affect the variable anyway. The HILVLEN, MEDLVLEN and LOLVLEN bits in PMIC CTRL register enable the individual levels.

It is important to take into consideration that the Global Interrupt bit (I-bit in SREG) is not cleared when entering an interrupt, as opposed to the single level interrupt system in megaAVR™ microcontrollers. The PMIC contains the mechanism that ensures that interrupts having same level cannot interrupt each other. Please refer to the description of the PMIC STATUS register found in the datasheet for more information.

2.4 Global Enabling of Interrupts

Even if interrupt levels are enabled in both the module and the PMIC, no interrupts are executed unless the Global Interrupt bit is set for the CPU. The assembly instructions SEI and CLI respectively enable and disable the global interrupts (please refer to the AVR instruction set for more information about these instructions).

When programming in C the enabling/disabling is typically done using inline assembly or compiler specific intrinsic functions. Refer to your compiler reference guide to find information about this.

2.5 Interrupt Vectors

Interrupt vectors are locations (addresses) in the program memory that are executed when an interrupt is triggered. The memory location will typically hold a JMP (or RJMP) instruction to the program memory address where the corresponding Interrupt Service Routine (ISR) is located. The addresses of the individual interrupt vectors are specified in the datasheet.

As a safety precaution it is recommended to not leave unused interrupt vectors unprogrammed. Unprogrammed memory will contain illegal instruction opcode, and the effect is undefined. “Filling” unused interrupt vectors with RETI instructions will ensure that the consequence of executing an interrupt that has no ISR is less critical. However, if employing this during development it may conceal the fact that an undefined interrupt is executed. One should consider implementing a dummy ISR for all unused interrupts during development and debugging, so that is possible to notice this kind of bugs.

If you are using the boot section of the program memory to (re)program the application section of the program memory while the interrupts should still operate, it is necessary to move the interrupt vectors to the boot section. If the vectors are not moved while the application section is being written an interrupt would cause the execution of a program memory address that cannot be read (due to the write process). Therefore, if interrupts are enabled while the application section of the program memory is written to, one has to move the vectors to the boot section to ensure that the interrupts executes correctly. The vectors are moved to the boot section by setting the IVSEL bit in the PMIC CTRL register. More information about bootloaders can be found in the application note AVR109 and AVR1316.

2.6 Interrupt Priority

The priority of interrupts is determined by their vector address (or vector number). The vector address is fixed for a given device. The interrupt priority determines which of two interrupts with same interrupt level that would be executed first if they occurring at the same time.

The PMIC offers a Round-robin scheduling scheme for low-level interrupt, which is enabled by setting the RREN bit in the PMIC CTRL register. When enabling the Round-Robin priority scheme the CPU will update the INTPRI register automatically so that it always holds the vector number of the interrupt that has been executed last. The value in INTPRI defines which low-level interrupt has the lowest priority. In this way the Round-Robin scheme ensures that no low-level interrupts are “starved”, as the priority changes continuously. In other words, the low-level interrupt last executed will always get lowest priority to ensure that other low-level interrupts are also serviced.

Note that the INTPRI register will not return to its default value (0x00), when disabling the Round-robin scheme: it will contain the number of the last low-level interrupt executed. If it is desired to return to the default priority order, one has to write 0x00 to the INTPRI register after disabling the Round-Robin scheme. The value in the INTPRI register will still affect the priority order of low-level interrupts even if the Round-Robin scheme is not used.

2.7 Interrupt Flags

A number of status bits in the AVR modules are referred to as interrupt flags. These interrupt flags provides information about a change of state in the module, e.g. that a byte has been received on the SPI. Interrupt flags will be set even if the corresponding interrupt is not enabled. This makes it possible to poll interrupt flags in the firmware main loop. Polling can be preferred in some cases, but are often handled by an ISR that are executed when the event occurs. One advantage of ISR handling is that important modules can be given immediate attention by temporarily stopping less important tasks. In the case where the interrupt controller has multiple interrupt levels, “less important tasks” includes interrupt with lower level than the “important” higher level interrupt. Another advantage is that the AVR can enter sleep mode, to reduce power consumption, and wake up when interrupts occur. Please refer to the datasheet to find out which interrupts that can wake the CPU from the different sleep modes.

Interrupts can be cleared manually, typically by writing a logic one to the flag, or automatically, when the interrupt vector is executed. Other interrupt flags are cleared when certain registers are accessed, e.g. the USART data register. Another example

is the interrupt flag for EEPROM writing, which will generate interrupts continuously while the EEPROM is not busy writing.

Further, some flags are not automatically cleared by execution the interrupt vector. Please refer to the datasheet to for information about clearing the individual flags.

2.8 Interrupts versus Event System

Note that it is not always necessary to let the CPU handle interrupts. The XMEGA Event System provides an alternate way to handle typical interrupt conditions in hardware without CPU intervention. The Event System can e.g. start an ADC conversion every time a timer overflow occurs. This offers unique timing opportunities, which would not be possible to obtain if CPU intervention was required. The Event System is described in application note AVR1303.

3 Driver Implementation

This application note includes a source code package with a basic driver implemented in C. It is written for the IAR Embedded Workbench® compiler.

Note that this driver is written to be highly readable and as general example how to use the peripheral module. If using the driver in an application it may be desirable to copy relevant parts of the code to where it is needed, to reduce to number of function calls. This will both speed up the code and reduce the code footprint.

3.1 Files

The source code package consists of three files:

- *pmic_driver.c* – driver source file
- *pmic_driver.h* – driver header file
- *pmic_example.c* – Example code using the driver

For a complete overview of the available driver interface functions and their use, please refer to the source code documentation.

3.2 Doxygen documentation

All source code is prepared for automatic documentation generation using Doxygen. Doxygen is a tool for generating documentation from source code by analyzing the source code and using special keywords. For more details about Doxygen please visit http://sourceforge.net/projects/doxygen. Precompiled Doxygen documentation is also supplied with the source code accompanying this application note, available from the *index.html* file in the source code folder.

Headquarters

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

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature