OBJECTIVES

- Understand general computer interrupt concepts and begin to utilize interrupts within the *ATxmega128A1U*.
- Learn how to use interrupts along with timer/counters to properly debounce a switch.

INTRODUCTION

Primarily, computer systems are comprised of multiple, independent components that communicate with one another.[1] For independence between components to remain effective, it is often desired that communication only take place when certain events are known to have occurred. To gain this ability, hardware components are generally designed to emit special digital signals, referred to here as *hardware flags* (or simply *flags*, if the context is clear[2]), that exactly identify the occurrence of specific events.[3]

For most computer systems, one generally flexible manner for handling an event of some peripheral component is to (1) identify an occurrence of the event through a hardware flag, either synchronously or asynchronously, and then (2) perform some set of instructions based on the event.[4] To limit how much an application program is responsible for keeping track of event occurrences, there is often the ability to additionally configure what is typically known as an *interrupt*. Essentially, an interrupt is *an automatic procedure for handling some event specified by a hardware flag*.[5, 6]

During an interrupt, the desired set of instructions to handle the relevant event are executed within what is generally known as an *interrupt service routine* (*ISR*), or *interrupt handler*. The manner in which an interrupt service routine is configured depends on the computer architecture (as do all other aspects of interrupts), however it is often either that there be a predefined section of memory for the instructions of the ISR to reside or that there be a predefined memory location, known as an *interrupt vector*, to contain information regarding where the desired set of instructions reside.[7] In the former case, the "size" of an interrupt service routine, or the amount of instructions that could be performed during the routine, is more or less predefined by the computer system designer, whereas in the latter case, the level of indirection achieved by an interrupt vector potentially allows a user to configure an arbitrarily-sized routine.

LAB STRUCTURE

In this lab, you will begin to utilize interrupts and the programmable interrupt controller (more specifically, the **p**rogrammable **m**ultilevel **i**nterrupt **c**ontroller [**PMIC**]) within the *ATxmega128A1U*. In § 1, you will learn fundamental information regarding interrupts within the *ATxmega128A1U* and then learn to how utilize an interrupt signal for a timer/counter module. Following this, in § 2, you will learn how to utilize an I/O port interrupt so that responses to a tactile switch on the *OOTB SLB* can be made asynchronously, as well as learn how to efficiently debounce such a switch in an asynchronous environment.

REQUIRED MATERIALS

- Atmel XMEGA AU Manual (doc8331)
- Atmel *ATxmega128A1U* Manual (doc8385)
- *OOTB µPAD v2.0* with USB A/B cable and accompanying schematic
- *OOTB Switch & LED Backpack* with accompanying schematic
- *Switch Debouncing with Software*
- **D**igilent **A**nalog **D**iscovery (**DAD**) kit with *WaveForms* software

SUPPLEMENTAL MATERIALS

- AVR Instruction Set (doc0856)

[1] There are various reasons for this modularity; some of the more common reasons involve efficiency, cost, and power, although there can be other factors as well.

[2] The term "flag" can also be used in contexts regarding software to describe a variable (i.e., a labeled set of memory locations) used to represent the value of some "logical" condition. In the specific discussion presented here, we do not consider such flags.

[3] Unfortunately, although hardware designers may try to provide a flag for each crucial or useful event within a given device, there is always some limit.

[4] Similar strategies can be implemented for computer systems without a central processing unit.

[5] In this context, hardware flags are often referred to as either *interrupt flags* or *interrupt signals*.

[6] When it is intended that there be multiple interrupts, some form of centralized control is often achieved through dedicated hardware such as a *programmable interrupt controller*.

[7] Typically, an interrupt vector is to contain the starting address of the desired interrupt handler, although this may not always be the case.

PRE-LAB PROCEDURE

REMINDER OF LAB POLICY

You must re-read the *Lab Rules & Policies* before submitting any pre-lab assignment and before attending any lab.

1. INTRODUCTION TO INTERRUPTS

Up until this point in this course, whenever it has been desired to determine when some event occurs within a peripheral component (e.g., when a tactile switch is pressed, when a TC counts for a specified amount of time, etc.), the CPU has been programmed to continually poll some input source. Ultimately, this technique wastes potentially useful processing time. In this section of the lab, you will be introduced to interrupts by implementing a simple overflow (OVF) interrupt for a timer/counter module within the *ATxmega128A1U*. However, before this, you must research all relevant information regarding interrupts within the *ATxmega128A1U*.

1.1. Read § 12 (*Interrupts and Programmable Multilevel Interrupt Controller*) of the 8331 manual, as well as § 14 (*Interrupts and Programmable Multilevel Interrupt Controller*) of the 8385 manual, to understand how interrupts are managed within the *ATxmega128A1U*.

1.2. Read all parts of § 14 (*TC0/1 – 16-bit Timer/Counter Type 0 and 1*) within the 8331 manual regarding timer/counter interrupts.

1.3. Create a simple assembly program, **`lab3_1.asm`**. Within this program, configure a timer/counter to trigger an overflow (OVF) interrupt every 250 ms. Within the necessary interrupt service routine, toggle an I/O port pin for which you have access to probe via the *µPAD*. Use your DAD to verify that the chosen pin toggles at the appropriate rate.

NOTE: An interrupt-driven program should have a format similar to what is shown in Figure 1.

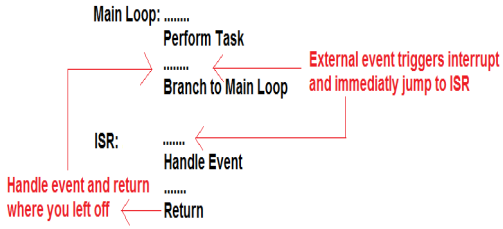

Figure 1: Proper program flow for an interrupt-driven program

PRE-LAB EXERCISES

i. Assuming that no interrupt has been previously configured, devise and describe a generalized series of steps for configuring any interrupt within the *ATxmega128A1U*, i.e., not just an interrupt within the TC system.

2. INTERRUPTS, CONTINUED

In this section of the lab, you will learn how to configure an I/O port interrupt so that responses to a tactile switch on the *OOTB SLB* can be made asynchronously. Additionally, you will learn how to efficiently debounce the relevant switch within the asynchronous context.

2.1. Read the relevant parts within § 13 (*I/O Ports*) of the 8331 manual to learn how to configure interrupts for an I/O port on the *ATxmega128A1U*.

2.2. Create an assembly program (**`lab3_2a.asm`**) to trigger an interrupt whenever tactile switch S1 is *pressed*, **without** debouncing the switch. Within the necessary ISR for the interrupt, increment a global counter, e.g., a register, and then display the updated count value on the LEDs available on the *OOTB SLB* such that the count value is displayed in binary notation with illuminated LEDs. (For example, if the current count value is equal to three, then LEDs *D1* and *D0* should be illuminated.) Additionally, within the main routine of the relevant program, after configuring the necessary interrupt, continually toggle on/off the blue LED within the RGB package available on your µPAD (labeled *BLUE_PWM* within the *µPAD* schematic) **as quickly as possible** (i.e., within a loop), to be able to easily highlight the fact that the separate, assigned tasks can appear to occur at the same time.

2.3. Verify that your program responds asynchronously to your tactile switch and that the toggling of the LED never appears to falter. Note that the relevant count value will most likely update *non-incrementally* due to switch bouncing.

Now, techniques for debouncing the relevant switch with interrupts will be explored. It is important to note that, in general, debouncing a switch in an asynchronous environment is *much different* than in a synchronous environment. In the *Switch Debouncing with Software* document available on our course website, two techniques are provided for environments where asynchronous responses to a switch are desired, however as the document further specifies, only one is allowable for our course.

2.4. Re-read and understand the pertinent sections of the *Switch Debouncing with Software* document.

2.5. Create another assembly program (**`lab3_2b.asm`**) to perform the same functionality as described in § 2.2, but additionally debounce tactile switch S1 with the relevant technique described in the *Switch Debouncing with Software* document.

2.6. Check that your switch is now correctly debounced by verifying that any count value displayed on your LEDs is exactly the same as the amount of times that the push-button has been pressed. (While remaining cautious, try pressing the button in a rougher-than-normal manner to attempt to ensure that the switch is appropriately

debounced even when placed under more "stressful" situations.)

NOTE: Now that you have become somewhat familiar with interrupts, do not let yourself get carried away when developing future applications – just like with anything, it should always be thoughtfully considered whether or not interrupts are necessary to accomplish some given task.

PRE-LAB PROCEDURE SUMMARY

1) Answer pre-lab exercises, when applicable.
2) Learn how to configure a timer/counter interrupt in § 1.
3) Understand how to configure an I/O port interrupt, as well as how to properly debounce a switch within an asynchronous environment, in § 2.