AVR135: Using Timer Capture to Measure PWM Duty Cycle

8-bit AVR® Microcontrollers

Application Note

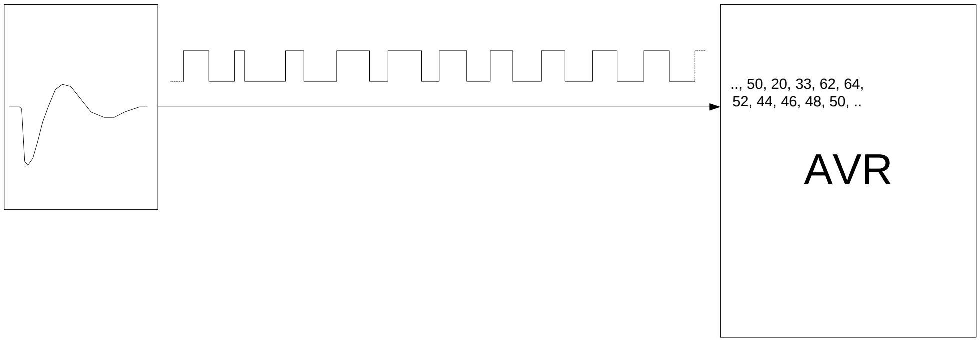
Features

- Scaled Duty-Cycle output (range set at compile time).
- Self-configuring and self-clocking via run-time PWM period computation.
- Requires 1 timer (with Input Capture): 2 interrupts, 1 external pin.
- 10 bytes SRAM, 336 bytes FLASH; 403 clock cycles/pulse.

1 Introduction

Historically Pulse Width Modulation (PWM) was used in a scenario (e.g. DC motors control) where an MCU merely generated a PWM signal and sent it to a target device. However, the examples of the ADXL202 (sensor to MCU) and J1850-PWM (MCU to MCU) show that there exist cases where an MCU has a need to decode a PWM signal. Indeed, given PWM's virtues (simple, robust, self-clocking) it would not be unreasonable to design new sensors, which convey readings via PWM. Although most Atmel AVR microcontrollers have ADCs, there may be a market advantage to designing a device, which is also usable by other MCUs, which lack ADCs.

Figure 1-1. Demodulation of PWM signal from analog sensor.



Rev. 8014A-AVR-10/05

2 Theory of operation

Pulse Width Modulation (PWM) is an encoding technique where a logic signal, on (typically) a single wire, is cycled to provide a sequence of pulses. A single PWM cycle consists of an active time span (the pulse proper) followed by an inactive time span, the sum of which comprises the full cycle period. The active (pulse) time span of the cycle may be indicated by a low (0) signal, known as Inverted PWM, or by a high (1) level, known as non-Inverted PWM. The information is encoded using the time width of the pulse as a proper fraction of the cycle period, normally referred to in the form of a percentage and termed the duty cycle. The PWM cycle is presumed to repeat constantly a pulse train, with the duty cycle varying only as the underlying information varies.

The cycle period is typically fixed for a given application device, though this is not strictly a requirement, since the encoding, being a ratio of pulse width to cycle period, does not depend on the raw timing itself. Indeed, the cyclic nature of the signal, along with the notion of the duty cycle, makes the PWM encoding self-clocking. This means that the PWM period needn't be known a priori, it needn't (with certain restrictions) be constant, and can be made largely immune from clock skew and variations due to temperature and voltage.

2.1 PWM Applications

PWM is routinely used to control the speed of DC motors using an H-bridge. In this application, the duty cycle is varied between 0% and 100%, and the physical properties of the motor itself effectively average the values of the pulses such that they are interpreted as a voltage varying between stopped (0VDC) and full speed (e.g. 12VDC).

This technique has also been used to provide a dimming mechanism for LEDs in which, by using a pulse rate faster than the LED can flash, the effective voltage (and thus brightness) can be varied by varying the PWM duty cycle.

These are examples of PWM being generated by an MCU for use by a device. In a reversal of this notion, the ADXL202 accelerometer (Analog Devices, Inc.) is a sensor, which produces PWM to convey the value of its current analog reading (between -2g (12.5%) and +2g (87.5%)) to an MCU. By using PWM for this application, there is no need for the MCU to have an Analog to Digital Converter (ADC).

While these are examples of encoding of analog information, PWM can also be used to encode digital information. SAE protocol J1850-PWM (used in intra-vehicle networks) defines an encoding where a single bit 0 is represented by a (nominal) 33% duty cycle, and a bit 1 by a 66% duty cycle.

Most models of the Atmel AVR microcontroller product line are able to generate PWM signals using one or more timers.

2.2 Variations

An encoding method, which is related to PWM is Pulse Width Coding (PWC). This also uses a pulse train to convey encoded information, but its encoding is different. For a PWC pulse, the information is encoded in the raw (timed) width of the pulse itself, and the cycle period is (within some limits) irrelevant. Perhaps the simplest comparison between PWM and PWC is that, if a PWM period is doubled, the pulse width is also doubled, but if a PWC period is doubled, the pulse width is unaffected. The AVR's PWM generation unit may be used to generate a valid PWC signal.

PWC is perhaps most visibly used in the control of RC servomotors. In this application, the pulse width is varied (typically) between 1ms and 2ms to position the servo mechanism; the refresh rate (PWC cycle period), however, may be freely (with some variation between units) chosen between 50Hz and 2kHz.

PWC is also used to convey digital information in the MD5 and RECS80 protocols used by infrared-based television remote controls. While these protocols are different, both depend on the raw pulse width, and thus a calibrated clock, to discern between bits 0 and 1. The PWC cycle time, which includes the time between bits, is not defined and thus may (in theory) be arbitrarily long.

While some of the techniques used in this Application Note may also apply to PWC, PWC implementation per se will not be discussed.

3 Implementation

The software for this application is written in C, and all functions, variables, defines, enums and typedefs are documented in the doxygen documentation (`readme.html`) that is downloaded with the source. Please also see the documentation for details on complier(s) and settings. The following subchapters describe various issues with, and parts of, the implementation.

3.1 Basic implementation

The PWM Decoder uses the Atmel AVR's Input Capture Unit (ICP) to measure the PWM pulse width and period. The ICP provides an edge-triggered interrupt based on the state of the AVR's ICPn pin, along with an internal register (ICRn), which contains a snapshot of the respective Timer/Counter (TCNTn) register at the moment of the trigger. By subtracting successive trigger timestamps, the pulse width and period may be computed.

Specifically, the timer to be used is configured initially (using the ICESn flag) to recognize the leading edge of a pulse (the setting of ICESn depends on whether Inverted or non-Inverted PWM is to be recognized). When the leading-edge interrupt is triggered, software in the Interrupt Service Routine (ISR) inverts the sense of the ICESn flag so that the next ICP interrupt will trigger on the trailing edge of the pulse. Similarly, during the trailing-edge interrupt, ICESn is inverted to detect the next leading edge. During each of these interrupts, the value of ICRn is saved for later computation.

The leading-edge interrupt also performs additional computation, since it is simultaneously the beginning of a new cycle and the end of the previous cycle. This is thus the point where the pulse width and period are computed by subtracting the stop and start times for the pulse and the current and previous start times, respectively. Once these values are known, the duty cycle for the previous cycle is computed and stored.

Pseudo-code for the ISR is:

```
Icp_isr()
  Reverse sense of ICP interrupt edge
  If (ICP interrupt was for rising edge)
    Previous_Period := ICR - Start_Time
    Previous_Pulse_Width := Stop_Time - Start_Time
    Start_Time := ICR
    Previous_Duty_Cycle :=
         Scale_Factor *
             (Previous_Pulse_Width/Previous_Period)
  Else
    Stop_Time := ICR;
```

No special provision is made for Inverted PWM, since the Inverted duty cycle may be simply computed as ICP_SCALE - icp_rx().

3.2 Implementation Issues

While this implementation plan, naively coded, will produce generally reasonable results, there are some boundary conditions, which must be considered.

The first is that it is possible to have a PWM duty cycle of 0%, or of 100%. These both have meaning, but they are anomalous, since the former cycle consists only of a (constant) inactive signal, and the latter only of an active signal -- in neither case is there any edge for the ICP to trigger on. To deal with these cases, the demodulator employs a heuristic, noting that for most applications the period is indeed constant, give or take some clock drift, which typically happens slowly. The solution is a timeout mechanism, using the timer's Output Compare Unit. At the beginning of each period, the comparator (OCRnA) is set to the expected end of the cycle, that being the edge timestamp (ICRn) plus the PWM period that was most recently measured. If the Output Compare triggers, the cycle is presumed to have completed, and a sample reading of 0% or 100% as appropriate is stored.

The second issue is a race condition, which follows from the fact that the ICP sense-transition (ICESn inversion) is done in software. This means that there is some minimum latency between the time a pulse edge is recognized by the ICP and the time the ICP is "re-armed" to recognize the subsequent edge. As a consequence of this, if a pulse is either very short (close to 0% duty cycle) or very long (close to 100% duty cycle) the subsequent edge may be missed. This latency includes:

1. Completion of the current instruction, or completion of any active ISR
2. Four cycles of internal MCU interrupt processing as defined by the Data Sheet.
3. Entry code (prior to the first executable statement) for the ICP ISR.

While item (3) can be reduced, it can't be eliminated, and items (1)-(2) can't be controlled at all; thus there is some minimum (and maximum) pulse width, outside of which range the pulse simply can't be measured. Knowing these minimum and maximum measurable pulses is an important part of the overall system design. For purposes of the demodulator, however, the prime concern is to recognize the condition so as to avoid producing outlandish results due to loss of synchronization with the signal.

Thus the capture ISR tests for a condition where (a) the relevant PINx register does not reflect the level to be expected after the "current" edge and (b) the ICP interrupt flag (ICFn) is not set. For these cases, it immediately performs the processing for the subsequent edge, effectively declaring the sample to be either 0% or 100% as appropriate.

3.3 "Analog" vs. "Digital"

The demodulator may be used on PWM streams representing either analog or digital data. While the PWM pulses per se have no notion of representing digital or analog data, there are subtle differences in the way the samples are processed. This is controlled using a C preprocessor symbol (ICP_ANALOG).

Analog samples are presumed to represent relatively smooth, though possibly noisy readings, and are presumed to arrive continuously, where the most recent values are taken to be the most important. Thus, when the demodulator is configured for analog data:

1. Samples are queued continuously, and queue overruns are ignored.
2. A moving average is maintained over the queue elements (the most recent N samples), and this value is returned by icp_rx().
3. icp_rx() does not "consume" queue elements, so the queue never empties; two calls within a single PWM period will return the same value.

Digital samples are treated as distinct, ordered items, with no relevance to one another. Thus, when the demodulator is configured for digital data:

1. Samples are queued in a circular fashion, and queue overruns are not permitted (new elements are thrown away).
2. No smoothing (moving average) is performed.
3. icp_rx() always returns the oldest queued item.
4. Each call to icp_rx() "consumes" a queue element. If the queue is empty, an "idle" indicator (100% duty cycle) is returned.

These are the only differences between the two schemes. Some applications, which use PWM for analog data, might yet prefer that data be handled according to the rules for "digital" data (the reverse is harder to imagine.

3.4 Application Program Interface

To use the demodulator, the application should

```
#include "icp.h"
```

This header file declares the functions below, and defines a required type: icp_sample_t. This type reflects the type used for computing the duty cycle, so it is wide enough to hold values [0:ICP_SCALE).

Two functions comprise the demodulator's API.

Table 3-1. Function declarations in “icp.h”

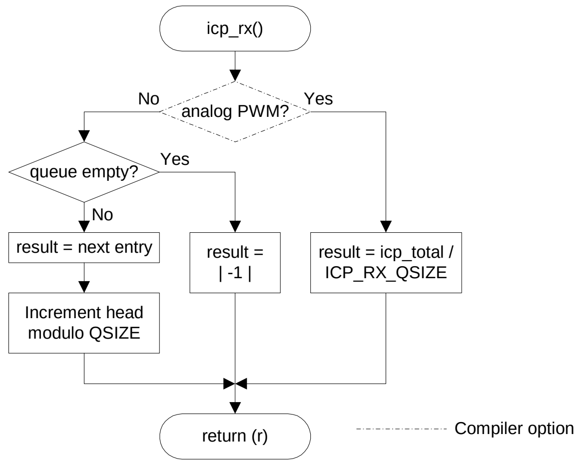
Function	Description
void icp_init(void)	Should be called during program initialization, before global interrupts are enabled. It configures the timer and does other setup tasks.
icp_sample_t icp_rx(void)	May be called to fetch a sample from the demodulator. This sample is an unsigned value in the range [0:ICP_SCALE), reflecting the duty cycle as a fraction of ICP_SCALE.

“Icp.h” also defines the preprocessor symbol ICP_ANALOG, which must be set to 1 to select analog mode and to 0 for digital.

3.5 ICP_RX

Figure 3-1 shows the flowchart for the icp_rx routine.

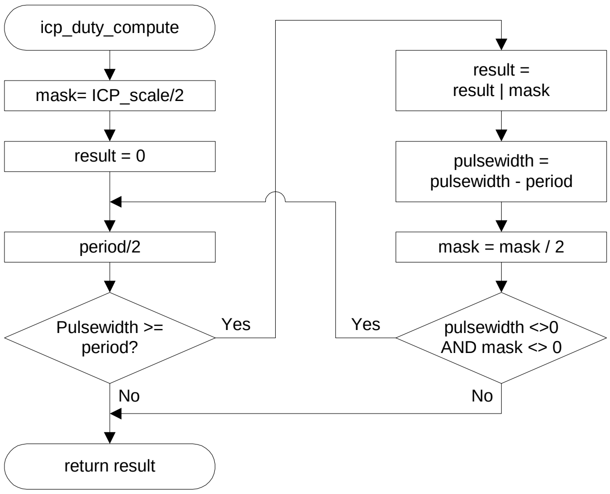
Figure 3-1. icp_rx flowchart.



3.6 Duty Cycle Scale

The computed duty cycle is in the form of a fraction of the value of ICP_SCALE. (The notion is analogous to a percentage, but using scale different from 100.) ICP_SCALE should be set to a power of 2. This setting affects the storage and CPU cost of computing and storing samples. If ICP_SCALE is greater than 256, samples are stored as a 16-bit quantity, and otherwise as an 8-bit quantity. Figure 3-2 shows how the duty cycle is computed.

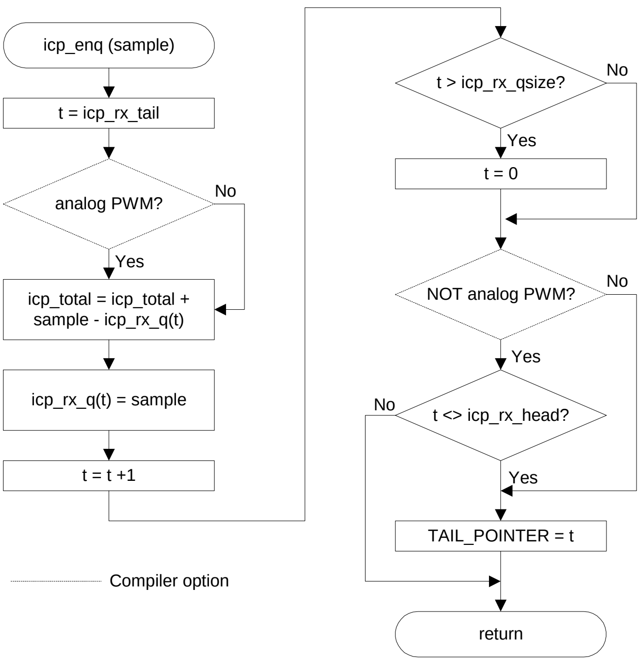
Figure 3-2. icp_duty_compute flowchart.



3.7 Queuing

The demodulator maintains a queue of sampled items. This queue is of size ICP_RX_QSIZE (in icp.h). The treatment of queued items is according to the section "Analog vs. Digital". Figure 3-3 shows the program flow.

Figure 3-3. icp_enq flowchart



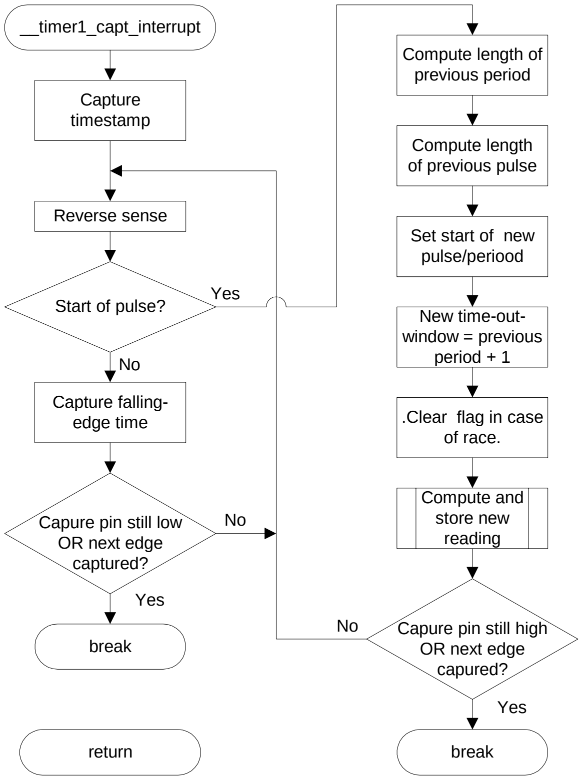
3.8 Capture with automatic calibration

The PWM demodulator dynamically re-computes the PWM period on every cycle. It should be mentioned here that, in most applications, the PWM period is considered to be constant, and there is a certain temptation to simply use this “well-known” constant rather than re-computing the period at the end of each cycle. The counterpoint to this notion is that (a) per-cycle computation of the period may be redundant, but it is also inexpensive, costing only a subtraction of two values which must be collected for other purposes, (b) using a compile-time, rather than a runtime, period value for computing the duty cycle gains little in CPU usage and (c) dynamic computation of the period provides robustness:

1. The demodulator is self-configuring, so the period needn’t be known ahead of time.
2. There are no concerns about the precision of the constant used.
3. The demodulator is self-correcting in the face of clock drift due to temperature or voltage.
4. Should a device be designed which doesn’t use constant period, the demodulator will support it trivially.

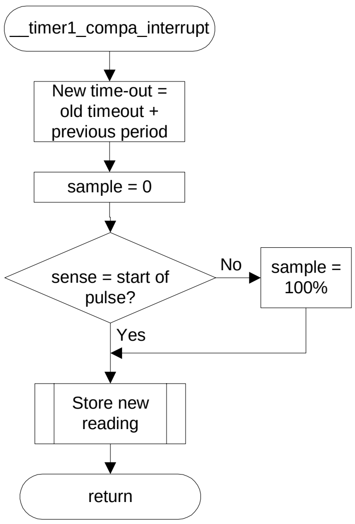
Indeed, there is a third option possible, wherein a calibration test is run periodically (seconds/minutes) to re-compute the period. However, this mechanism will inevitably affect the “core” timing code, and the per-cycle period computation is so inexpensive as to compare favorably with any additional complexity from a separate calibration mechanism.

Figure 3-4. __TIMER1_Capt_ interrupt flowchart.



The interrupt routine TIMER1_COMPA handles the 0% and 100% duty cycle cases. It's flowchart is shown in Figure 3-5.

Figure 3-5. __Timer1_COMPA_interrupt flowchart



4 Final considerations

Finally, there is the fact that decoding the PWM train takes a certain amount of processing time; this defines the minimum PWM period, which can be used. This is a significant system design consideration. For the ADXL202, e.g., this is not a concern since its minimum configurable period is 1 ms (1 kHz). At the other extreme, however, J1850-PWM defines a minimum PWM period of 24µs (41.666 kHz). The demodulator implementation requires 403 CPU cycles to decode a single pulse; to process a 24µs PWM period; the MCU must therefore be run at a minimum of 16.8MHz.

5 Demonstration Program

main.c contains a simple program to demonstrate the demodulator operation. It generates a PWM signal on OC2, then samples the demodulator output using calls to icp_rx() and writes the values obtained to PORTC.

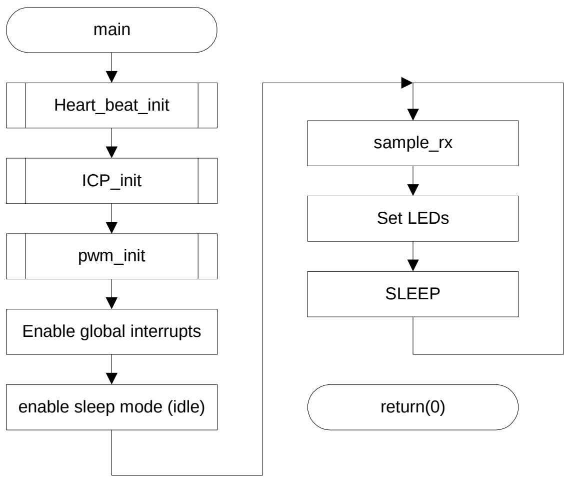
Approximately twice per second, OCR2 is incremented such that the PWM duty cycle steps through the entire [0:256) range.

The intended use is on the STK500/STK501, with:

1. A 10-wire jumper connecting the PORTC header with the LEDs header.
2. A single-wire jumper connecting PB7 (OC2) with PD4 (ICP1).

The expected result is that the LED display cycles repeatedly from 0x00 to 0xFF. Figure 5-1 shows the main program flow, except for the interrupt routine that updates OCR2.

Figure 5-1. Flowchart for demonstration program



6 Table of Contents

Atmel Corporation

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

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

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

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

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature