AVR 8-bit Microcontrollers

AVR135: Using Timer Capture to Measure PWM Duty Cycle

APPLICATION NOTE

Introduction

This application note demonstrates the process to measure the pulse width of PWM waveform using Capture module of the Timer/Counter in Atmel® AVR® devices. Pulse Width Modulation (PWM) were originally predominant in motor control applications, where an MCU generates a PWM signal and send it to a target device (DC Motors). However, there are also some applications where PWM waveform is used to pass informaton from a sensor to the MCU or MCU to MCU. In such cases, MCU has to decode a PWM signal.

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
- Firmware provided for ATmega64 and ATmega328PB devices

Table of Contents

Introduction....1

Features....1

1. Abbreviations....3

2. Introduction....4
   - 2.1. PWM Applications....4
   - 2.2. PWM Variations....5

3. Implementation....6
   - 3.1. Basic Implementation....6
   - 3.2. Analog Input vs Digital Input....6
   - 3.3. Application Constraints....7

4. Software Implementation....9
   - 4.1. Device Selection....9
   - 4.2. Functional Description....9
   - 4.3. Software Modules....10
     - 4.3.1. Capture Pulse Width - `icp_rx()`....10
     - 4.3.2. Compute the Duty Cycle - `icp_duty_compute()`....11
     - 4.3.3. Queue the Sample - `icp_enq()`....12
     - 4.3.4. Capture with Automatic Calibration....13

5. Test Setup....16

6. Test Results....17

7. References....18

8. Revision History....19

1. Abbreviations

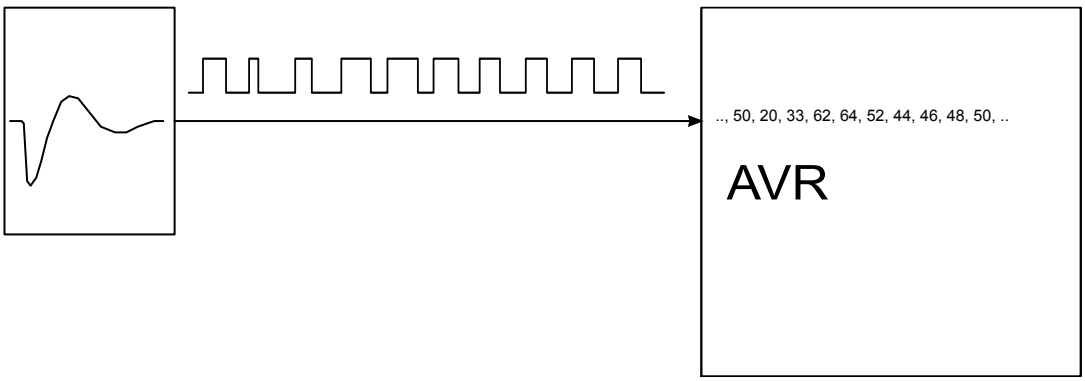
ADC	Analog to Digital Converter
CPU	Central Processing Unit
DC	Direct Current
Hz	Hertz
ICP	Input Capture Unit
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
kHz	kiloHertz
LED	Light Emitting Diode
MCU	Microcontroller Unit
MHz	megaHertz
PWC	Pulse Width Coding
PWM	Pulse Width Modulation
RC Servomotor	Radio Control Servomotor
RC Filter	Resistor-Capacitor Filter

2. Introduction

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 (Pulse ON time) followed by an inactive time span (OFF time), 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 (constant period) 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 need not be known entity, it need not (with certain restrictions) be constant, and can be made largely immune from clock skew and variations due to temperature and voltage.

Figure 2-1. Demodulation of PWM Signal from Analog Sensor



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 brightness control 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, there are some sensors that produces PWM to convey the value of its current analog reading to an MCU. By using PWM for this application, there is no need for the MCU to use an ADC module for measuring the duty cycle.

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 of the Atmel AVR microcontroller product line are capable of generating PWM signals using one or more timers.

2.2. PWM Variations

Another 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. Typically, the pulse width is varied between 1ms and 2ms to position the servo mechanism; the refresh rate (PWC cycle period), however, may be freely chosen between 50Hz and 2kHz, with some variation between units.

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, discussion on implementation will be limited to PWM.

3. Implementation

The software for this application is written in C, available for download as an attachment along with the application note. The details of hardware to be used for testing is explained in the subsections along with application constraints in the implementation. The firmware is a Atmel Studio project which can be used to test right away, by programming the hex (or elf) file to the device. It has been written and tested for the following devices:

1. ATmega64
2. ATmega328PB

Following sections will explain the theory behind 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 AVR's ICPn pin, along with an internal register, which contains a snapshot of the respective Timer/Counter 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 to recognize the leading edge of a pulse, which 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 edge sense flag so that the next ICP interrupt will trigger on the trailing edge of the pulse. Similarly, during the trailing-edge interrupt, the edge sense flag is inverted to detect the next leading edge. During each of these interrupts, the value captured by capture module 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. When these values are known, the duty cycle for the previous cycle is computed and stored.

The 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. Analog Input vs Digital Input

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. In the application note, 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 however may not be possible).

3.3. Application Constraints

While this implementation plan 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, there is 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 (with a minimal 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 is set to the expected end of the cycle, that being the edge timestamp 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 is done in software. This means a minimum latency exists 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, the prime concern is to recognize the condition to avoid producing undefined results due to loss of synchronization with the signal.

Thus the capture ISR tests for a condition where,

1. The relevant PINx register does not reflect the level to be expected after the current edge.
2. The ICP interrupt flag 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.

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 captured by the application. This is a significant system design consideration which depends on the number of clock cycles spent in the ISR and to calculate the duty cycle.

4. Software Implementation

The firmware provided has been verified on ATmega64 and ATmega328PB by changing the device in the Atmel Studio. The project contains the files as explained in the following table:

Table 4-1. Project file Information

File Name	Description
`main.c`	This file contains the main routine which has the application code. This includes the initialization and ISR for Timer 1 used as free running timer.
`device.h`	This file acts as a wrapper to have definitions for the two devices. This code can be used for other devices with no changes or minimal, if any.
`icp.c`	This file consists of the drivers necessary for the ICP module. The functions `icp_init` and `icp_rx` can be called respectively, to do initialization and fetch a sample for demodulator.
`icp.h`	This file should be included to use the functions in `icp.c`. This header file declares the functions defined in `icp.c` and a required data 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`]. It also defines the preprocessor symbol `ICP_ANALOG`, which must be set to 1 to select analog mode and to 0 for digital.

More details about the firmware and its functioning are discussed in the following sections.

4.1. Device Selection

This project is developed to work for two different devices (ATmega64 and ATmega328PB). The code has conditional compilation macros to use appropriate code based on device selection. The firmware for respective device shall be compiled by just selecting the appropriate device in Atmel Studio IDE. This is done in the `device.h` header file. This file acts as a wrapper between the application and the device register access.

There are some minor differences in the register names between the two devices and hence the segregation is necessary. Depending upon the device selected, the macro `DEVICE_ATMEGA328PB` or `DEVICE_ATMEGA64` gets defined. Based on the macro that gets defined, further definitions ensure error free compilation. The macro also limits the user to select either of the two devices, failing which a compilation error message is displayed. Application can be made to support more AVR devices with no change or minimal, if any. This can be handled by modifying the `device.h` file.

4.2. Functional Description

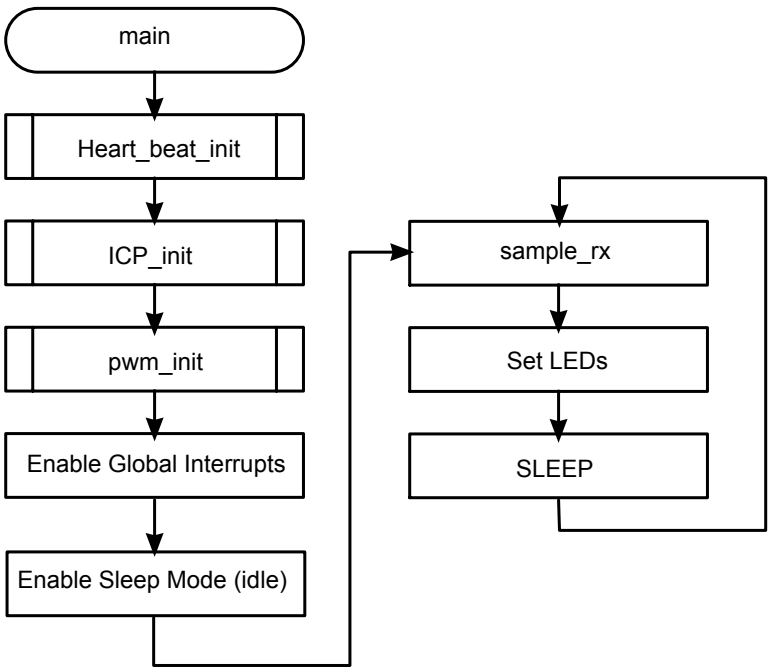
`main.c` contains a simple program to demonstrate the demodulator operation. The application uses three Timers available in the AVR for PWM generation, free running timer and pulsewidth capture. This implementation takes care of considerations mentioned in Application Constraints. The application flow is as follows:

1. Initialize the free running timer, Timer 0 as heart beat timer. This is used for generating interrupt approximately, every 1 second to change the duty cycle of PWM waveform generated. In a realtime application where only pulse width capture is needed, this may not be necessary. The Output Compare Register is incremented such that the PWM duty cycle steps through the entire [0:256) range.

2. Initialize the ICP module of Timer 1 to capture the pulse width. This is also configured to generate a compare interrupt when there is no edge detected for a duration more than the expected pulse period.
3. Initialize the PWM module of Timer 2 to generate and pulse train with constant period and varying duty cycle. In a realtime application where only pulse width capture is needed, this may not be necessary.
4. Enable global interrputs and enter sleep mode.
5. Periodically get the sampled values and calculate the duty cycle.
6. Depending upon the sample values, set the LEDs.

Following flowchart explains the logical flow of `main.c` file.

Figure 4-1. Flowchart for main()



4.3. Software Modules

The CPU is executed at 8MHz using Internal RC Oscillator. The heart beat timer used to generate interrupt is configured to overflow at the maximum value. The prescalar value is set to 1024 and still the highest overflow value of 255 (8-bit) will not be sufficient to have a delay of 1 second. A variable is used to count the entry into ISR before the duty cycle is actually changed.

The firmware can be grouped under 4 major modules based on the tasks performed, as follows:

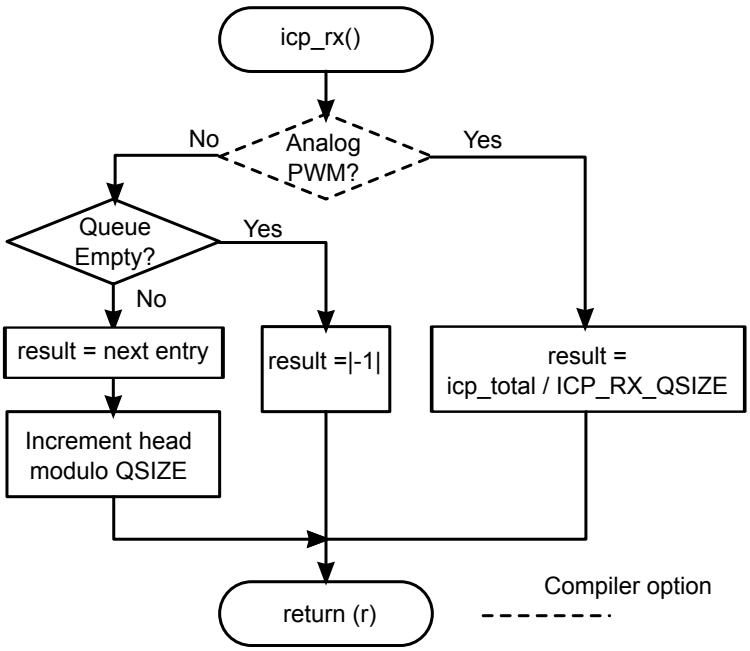
1. Capture Pulse Width - `icp_rx()`
2. Compute the Duty Cycle - `icp_duty_compute()`
3. Queue the Sample - `icp_enq()`
4. Capture with Automatic Calibration - `Compare ISR`

4.3.1. Capture Pulse Width - `icp_rx()`

The routine `icp_rx()` is called in the main routine, to get the sample values periodically. The selection of Analog Input or Digital Input is made before compilation. In case of analog input signal (done by enabling

ICP_ANALOG macro) moving average of last **n** are returned. If the selection is made as ICP_DIGITAL a queue is maintained to retrieve the oldest reading. The flow of software is shown in the following figure.

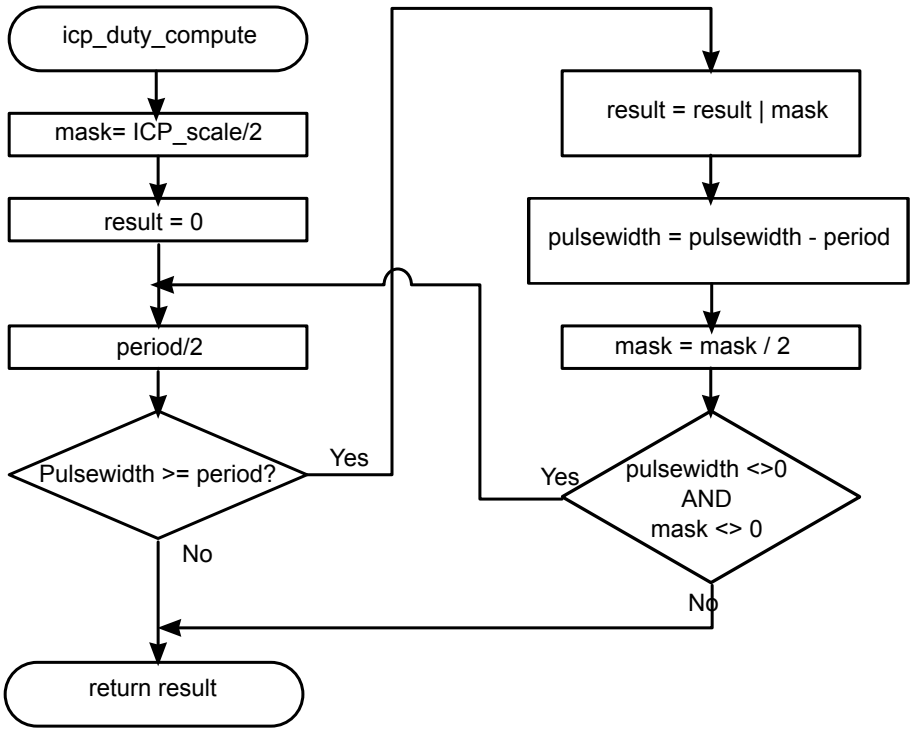
Figure 4-2. Flowchart for icp_rx()



4.3.2. Compute the Duty Cycle - `icp_duty_compute()`

The sample value received through icp_rx is in the form of a fraction of the value of ICP_SCALE. The notion is analogous to a percentage, but the scale used here is 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_SCALEis greater than 256, samples are stored as a 16-bit quantity, and otherwise as an 8-bit quantity. This also helps in easy display of duty cycle through the output LEDs by just sending the duty cycle value to the respective PORT. Following figure shows how the duty cycle is computed.

Figure 4-3. Flowchart for icp_duty_compute()

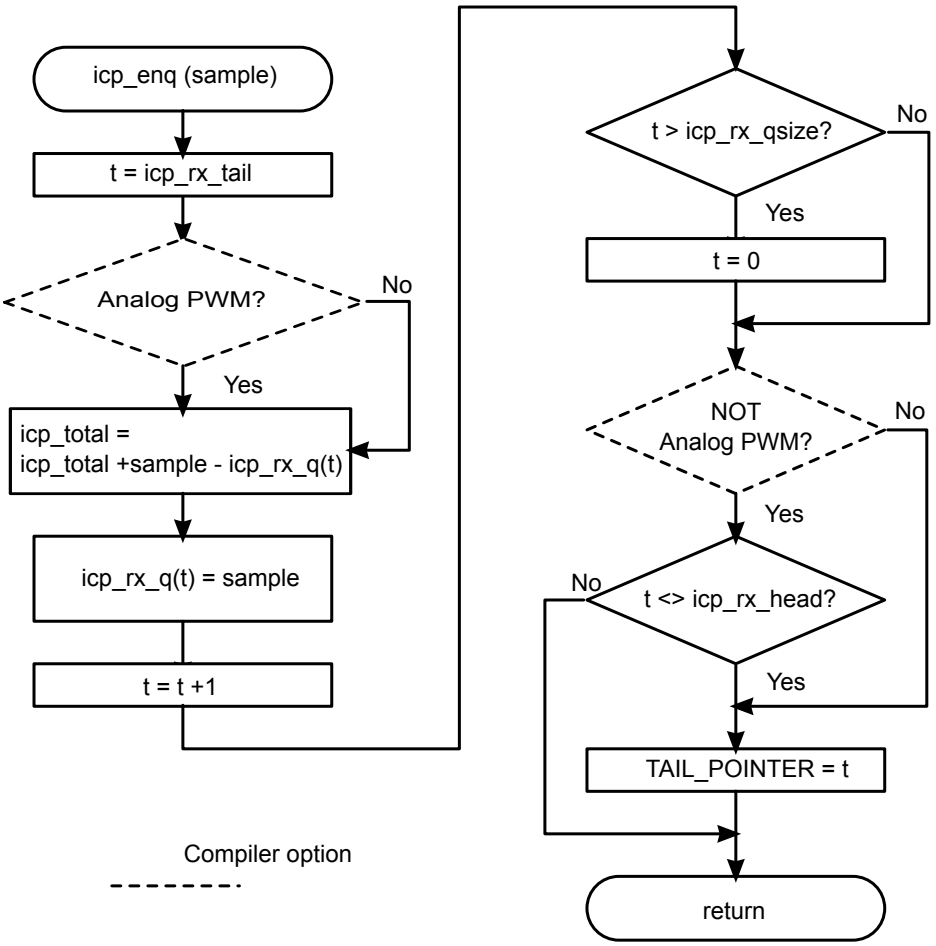


4.3.3. Queue the Sample - `icp_enq()`

The demodulator maintains a queue of sampled items. This queue is of size `ICP_RX_QSIZE` defined in `icp.h`. The treatment of queued items is based on the selection made during compilation.

The following figure shows the flow of the function `icp_enq()`.

Figure 4-4. Flowchart for icp_enq()



4.3.4. Capture with Automatic Calibration

The PWM demodulator dynamically re-computes the PWM period on every cycle. 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 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. Also, using a compile-time, rather than a runtime period value for computing the duty cycle gains little in CPU usage and dynamic computation of the period provides robustness:

1. The demodulator is self-configuring. Hence, the period need not 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 does not 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.

The following flowcharts explain the software flow in Capture and Compare Interrupt routines:

Figure 4-5. Flowchart for Timer 1 Capture ISR

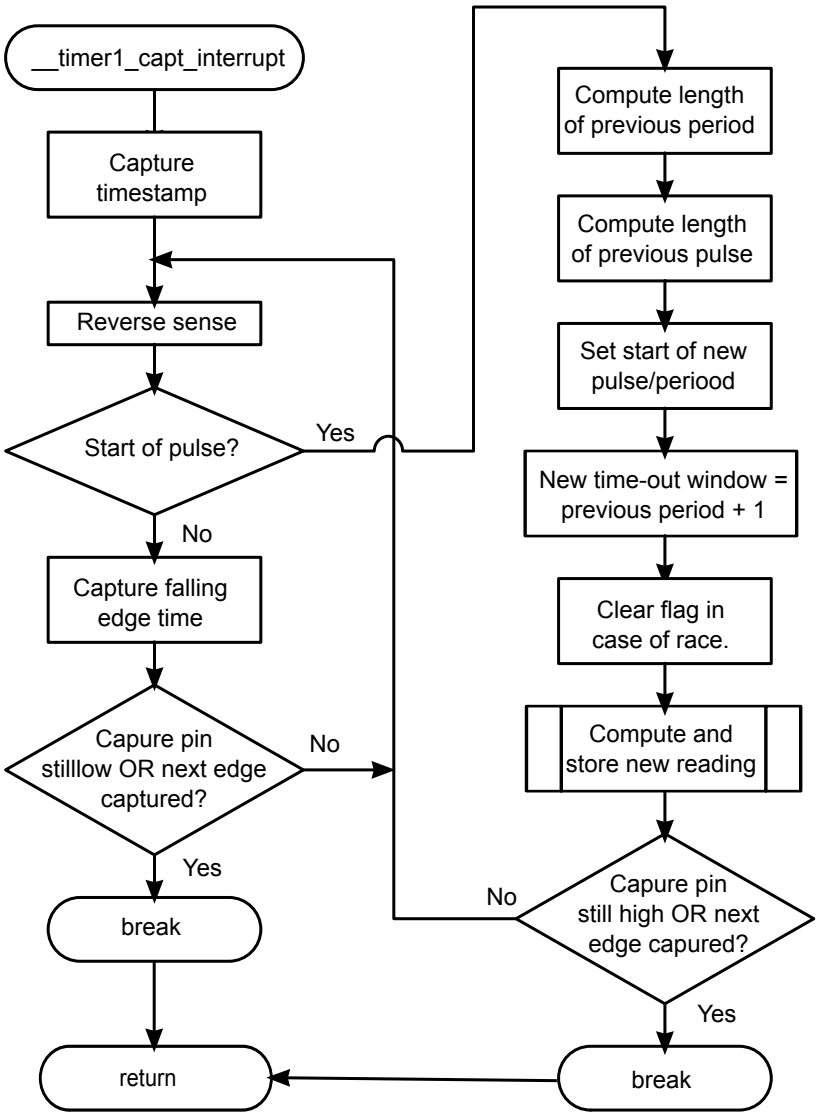
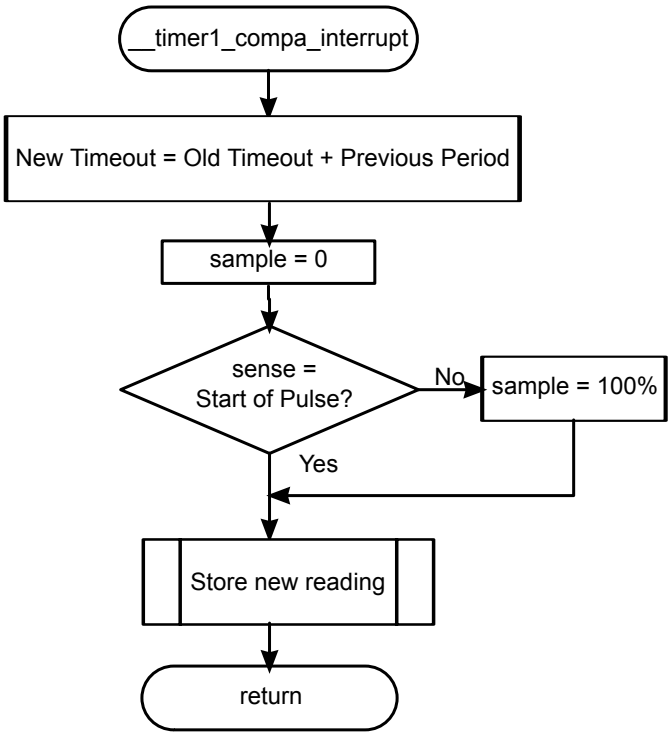


Figure 4-6. Flowchart for Timer 1 Compare ISR



5. Test Setup

The firmware provided along with this application note has been developed using Atmel Studio 7 and tested on STK600. Since some pin functionalities differ among both the devices, the configurations will be managed through `device.h` file.

Test Setup for ATmega64:

1. Timer 2 Ouput Compare (OC2) is used to generate PWM on pin PB7.
2. Timer 1 Input Capture (ICP1) is used to measure the pulse width through Pin PD4. Use a single-wire jumper to connect PB7 with PD4.
3. The measured Duty Cycle is displayed through the Port C pins that are connected to the LEDs available in STK600. Use a 10-wire jumper to connect PORTC header with LEDs header.

Test Setup for ATmega328PB:

1. Timer 2 Ouput Compare (OC2A) is used to generate PWM on pin PB3.
2. Timer 1 Input Capture (ICP1) is used to measure the pulse width through Pin PB0. Use a single-wire jumper to connect PB3 with PB0.
3. The measured Duty Cycle is displayed through the Port D pins that are connected to the LEDs available in STK600. Use a 10-wire jumper to connect PORTD header with LEDs header.

Note:

1. More information about programming an AVR microcontroller is available at References.
2. ATmega64 and ATmega328PB requires different Routing and Socket cards when used along with STK600. For more information, refer the links provided in References.

6. Test Results

The duty cycle of PWM generated to test the application varies over the range from 0 to 255. The range of the duty cycle can be modified based on the application. This helps in easy visualization of varying duty cycle through the LEDs available in the STK600. When the code is compiled and tested, the LEDs of STK600 shall display cycles repeatedly from 0x00 to 0xFF like a binary counter.

Note: In the application demonstrated, the input pulse width is less than the time spent by CPU in the capture ISR. As explained in Application Constraints, LED port shows the lower steps (0 & 1) of PWM's duty cycle as 0x00.

7. References

1. Simply AVR Webpage
2. AVR136: Low-Jitter Multi-Channel Software PWM
3. AVR1617: Frequency Measurement with Atmel AVR XMEGA Family Devices
4. Routing Card and Socket Card related information
5. Atmel Studio - Programming Dialog

8. Revision History

Doc Rev.	Date	Comments
8014B	04/2016	Updated for Atmel Studio 7 and added ATmega328PB device support
8014A	10/2005	Initial document release

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**