Content

Software Setup 3

What is this setup doing? 3

Line 1 4

Line 2 4

⏱️ No Prescaler (CS10 = 1) 5

🎧 Noise Canceler Enabled (ICNC1 = 1) 5

🌊 Edge Selection (ICES1 = 0) 5

Line 3 5

⚖️ What does the Analog Comparator compare? 6

⚡ What happens in hardware 6

🧠 What we've built 6

How it works in the loop 7

What is the code measuring? 7

⏱️ Role of the 16 MHz clock 7

🔢 Converting ticks → time 7

🎯 Frequency formula 8

🤔 Why does the code use 1,600,000,000? 8

🔥 Example 9

🧠 Key insight 9

Why ICR1 and not just TCNT1? 10

🧩 First: what are these registers? 10

🎯 What is Input Capture? 10

⚡ Why not just read TCNT1 in software? 10

🚀 What Input Capture does better 11

🧠 Why this code still uses loops 12

🆚 TCNT1 vs ICR1 summary 12

🎯 Big picture 12

How the analog comparator is wired into the timer capture unit 13

🧠 The Big Idea 13

⚙️ Step 1 — The Analog Comparator 13
🪠 Step 2 — Internal Routing to Timer1 14
⏱️ Step 3 — What happens on an edge? 14
🎚️ Step 4 — Which edge triggers capture? 15
🧩 Why this is insanely powerful 15
🆚 Without this hardware path 16
🎯 Why this is perfect for FT8 16
📌 Summary 16
Hardware 17
Modulation input with low-pass filter 17

Software

Setup

```
// Timer1 configuration (used for frequency measurement)
  TCCR1A = 0x00;
  TCCR1B = 0x81;      // No prescaler, input capture enabled
  ACSR  |= (1 << ACIC); // Analog comparator triggers input capture
```

These lines configure **Timer1** and the **Analog Comparator** on the ATmega328P (Arduino Nano) to do **very precise frequency or period measurement** using hardware — almost no CPU involvement.

What is this setup doing?

We are building a system where:

1. **Timer1** runs as a high-speed clock.
2. The **Analog Comparator** watches an analog signal.
3. When the signal crosses a threshold, hardware:
    - instantly stores the current timer value.
4. From the time between two captures → you compute **period** → **frequency**.

This is extremely accurate because it avoids software timing delays.

Line 1

```
TCCR1A = 0x00;
```

TCCR1A = Timer/Counter1 Control Register A

Setting it to 0x00 clears all bits.

Effect:

Feature	Result
Waveform Generation Mode bits (WGM10, WGM11)	0 → Normal timer mode
Compare Output (COM1A/COM1B)	Disabled (no PWM on D9/D10)

👉 Timer1 is just a **free-running counter**. No PWM, no output compare behavior.

Line 2

```
TCCR1B = 0x81;   // No prescaler, input capture enabled
```

0x81 = 1000 0001 in binary.

TCCR1B = Timer/Counter1 Control Register B

Bit	Name	Value	Meaning
7	**ICNC1**	1	Input Capture Noise Canceler ON
6	ICES1	0	Capture on **falling edge**
5	–	0	–
4	WGM13	0	Normal mode
3	WGM12	0	Normal mode
2	CS12	0	Prescaler bits
1	CS11	0	Prescaler bits
0	**CS10**	1	Prescaler bits

⏱️ No Prescaler (CS10 = 1)

Timer1 runs at full CPU speed:

$16\,MHz \rightarrow 1\,tick = 62.5\,ns$

That gives **very high time resolution**, ideal for frequency measurement.

🎧 Noise Canceler Enabled (ICNC1 = 1)

The input capture signal must stay stable for **4 clock cycles** before triggering.

✓ Reduces false triggers from noise

✕ Adds a tiny delay (negligible for most uses)

🌊 Edge Selection (ICES1 = 0)

Capture occurs on a:

HIGH → LOW transition

If you want rising edge instead:

```
TCCR1B |= (1 << ICES1);
```

Line 3

```
ACSR |= (1 << ACIC); // Analog comparator triggers input capture
```

ACSR = Analog Comparator Control and Status Register

Bit **ACIC** = *Analog Comparator Input Capture Enable*

Normally, Timer1 Input Capture is triggered by the **ICP1 pin (D8)**.

This line reroutes the trigger:

“Use the **Analog Comparator output** as the capture trigger instead.”

⚖️ What does the Analog Comparator compare?

Input	Arduino Pin
AIN0 (+)	D6
AIN1 (–)	D7

When AIN0 crosses AIN1
the comparator output flips → this now triggers **Timer1 Input Capture**.

What happens in hardware

When the comparator switches:

1. Hardware instantly copies **TCNT1 → ICR1**
2. An interrupt can fire (if enabled)
3. You subtract two capture values:

$$\Delta t = ICR1_2 - ICR1_1$$

Then:

$$f = \frac{F_{CPU}}{\Delta t}$$

What we've built

We created a hardware measurement chain:

Analog Signal → Comparator → Timer1 Input Capture → Timestamp

✓ No digitalRead()
✓ No polling
✓ No timing jitter from code

This is **precision measurement using pure hardware** — very efficient and very accurate.

How it works in the loop

What is the code measuring?

The analog comparator (bit **ACO** in **ACSR**) sees an audio tone (FT8 signal).
The code measures parts of the waveform:

Variable	What it represents
d1	length of one HIGH period
d2	length of HIGH + LOW + next HIGH
d2 – d1	LOW + HIGH = **one full period**

So the code extracts the **signal period**.

⏱ Role of the 16 MHz clock

Timer1 runs at the **CPU clock = 16 MHz**

So:

$$1 \text{ timer tick} = \frac{1}{16{,}000{,}000} = 62.5 \text{ ns}$$

The timer is basically a very fast stopwatch.

🔢 Converting ticks → time

If the timer counts **N ticks** during one signal period:

$$T = \frac{N}{16{,}000{,}000}$$

Here:

$$N = d2 - d1$$

Frequency formula

Basic physics:

$$f = \frac{1}{T}$$

Insert T:

$$f = \frac{1}{(d2 - d1)/16{,}000{,}000} = \frac{16{,}000{,}000}{d2 - d1}$$

That would give frequency in **Hz**.

Why does the code use 1,600,000,000?

```
unsigned long codefreq = 1600000000UL / (d2 - d1);
```

Because:

$$1{,}600{,}000{,}000 = 16{,}000{,}000 \times 100$$

They scale by **×100** to keep **two decimal digits** without using floating point.

So:

$$\text{codefreq} = f \times 100$$

Real tone	**Stored value**
1000 Hz	100000
2500 Hz	250000

Later:

```
if (codefreq < 350000)   // = 3500 Hz
```

Example

Suppose:

d2 - d1 = 8000 ticks

$$f = \frac{16{,}000{,}000}{8000} = 2000\ Hz$$

Code:

1600000000 / 8000 = 200000

→ 200000 / 100 = **2000 Hz**

Key insight

The **loop speed does NOT matter** here.

Even if the CPU is stuck in those while() loops, **Timer1 keeps counting in hardware** at 16 MHz. The CPU is just waiting for signal edges; the timer measures time independently.

That's the big concept:

👉 **You measure frequency by measuring time, not by counting loop executions.**

Why ICR1 and not just TCNT1?

First: what are these registers?

Register	Meaning
TCNT1	Timer1 counter (live value, constantly changing)
ICR1	*Input Capture Register* (a hardware snapshot of TCNT1)

Think of it like this:

TCNT1 = a running stopwatch
ICR1 = a photo taken of the stopwatch at an exact instant

What is Input Capture?

Timer1 has special hardware:

When a selected signal edge occurs (rising or falling), the hardware:

1. **Instantly copies TCNT1 → ICR1**
2. Does this **in the same clock cycle**
3. No software delay

That's insanely precise — **cycle accurate**.

Why not just read TCNT1 in software?

Because software is **slow and jittery** compared to hardware.

In your loop:

```
while (ACSR & (1 << ACO)) { ... }
```

The CPU is polling:

- Read register
- Test bit
- Branch

That takes multiple cycles. When the signal edge happens, the CPU might detect it:

- 2 cycles late
- 8 cycles late
- sometimes more

That timing uncertainty = **measurement error**.

At 16 MHz:

1 cycle = 62.5 ns

10 cycles error = 625 ns

For a 2000 Hz tone:

Period = 500 µs

625 ns error ≈ 0.125% error

That's already noticeable for precise FSK/FT8.

🚀 What Input Capture does better

The analog comparator is internally connected to Timer1 capture.

When the comparator flips:

➞ hardware triggers capture
➞ **TCNT1 is frozen into ICR1 instantly**

No loop delay. No instruction timing. No jitter.

Precision = **±1 timer tick** (62.5 ns). Beautiful.

🧠 Why this code still uses loops

The loops are only used to:

- Wait for specific signal transitions
- Define where a period starts/ends

But the **actual timestamp** comes from hardware capture.

So the CPU is just acting as a traffic cop; the timer hardware does the stopwatch job.

🆚 TCNT1 vs ICR1 summary

Feature	TCNT1	ICR1
Live counter	✅	❌
Snapshot at signal edge	❌	✅
Software latency error	YES	NO
Cycle accurate	❌	✅
Best for frequency measurement	😬	🔥

🎯 Big picture

This system works like this:

1. Timer1 runs at 16 MHz
2. Comparator sees audio waveform
3. Hardware captures timestamps into ICR1
4. Code subtracts timestamps
5. Time → frequency

This is basically how oscilloscopes and frequency counters work internally — just simpler.

How the analog comparator is wired into the timer capture unit

This is one of those “hidden superpowers” of the AVR.

We’re talking about:

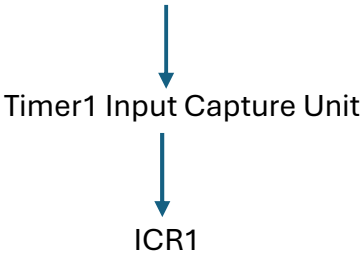
How the Analog Comparator is internally connected to Timer1 Input Capture

This is pure hardware — no Arduino magic.

The Big Idea

The AVR has an internal signal path:

Analog Comparator Output (ACO)



So an **analog voltage crossing** can directly trigger a **timer timestamp**.

No CPU involved. Zero software delay.

Step 1 — The Analog Comparator

The comparator compares two voltages:

Pin	Meaning
AIN0	Positive input
AIN1	Negative input

If:

AIN0 > AIN1 → ACO = 1

AIN0 < AIN1 → ACO = 0

So your audio signal becomes a **digital square wave** at the comparator output.

That bit is: ACSR register → ACO bit

🚦 Step 2 — Internal Routing to Timer1

Now comes the cool part.

There is a control bit in **ACSR**:

ACIC = Analog Comparator Input Capture Enable

When **ACIC = 1**:

The comparator output is wired internally to Timer1's **Input Capture trigger**.

So the timer can react to comparator edges just like it would to an external pin.

⏱️ Step 3 — What happens on an edge?

When the comparator output changes (depending on edge selection):

1. Timer hardware detects edge
2. Current value of **TCNT1** is copied to **ICR1**
3. **In the same clock cycle**
4. Optional interrupt can fire

This is called an **Input Capture Event**.

🎚 Step 4 — Which edge triggers capture?

Controlled by **TCCR1B register**:

ICES1 = Input Capture Edge Select

ICES1	Trigger
1	Rising edge
0	Falling edge

So you can measure:

- High time
- Low time
- Full period

Just by switching edges.

🧩 Why this is insanely powerful

You just built:

✓ A zero-crossing detector
✓ A frequency counter
✓ With 62.5 ns resolution
✓ Using only hardware

That's lab-grade measurement technique.

🆚 Without this hardware path

If the comparator wasn't connected to the timer:

You'd have to:

- Poll ACO in a loop
- Detect edges in software
- Read TCNT1 manually

Result = jitter, missed edges, bad precision.

🎯 Why this is perfect for FT8

FT8 tones are close in frequency (only ~6.25 Hz spacing).

To decode them reliably, you need:

- Very precise frequency measurement
- Low jitter
- Stable timing

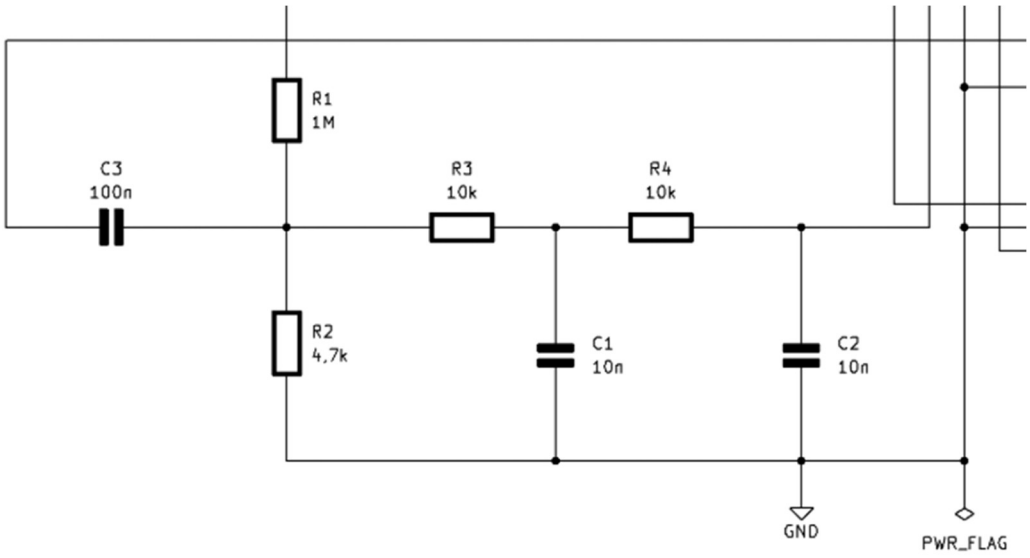
This hardware capture path makes that possible on an 8-bit MCU 🤯

📌 Summary

Component	**Role**
Comparator	Converts audio → digital edges
ACIC bit	Routes comparator to timer
Timer1	High-speed time base
Input Capture	Hardware timestamp
ICR1	Stores exact edge time

Hardware

Modulation input with low-pass filter



The 100n capacitor at the input filters out the remaining DC voltage.

The auxiliary voltage of 3.3 V is used to reliably detect the end of a transmission.

An RC filter with 10 k and 10 nF has a cutoff frequency of 1.6 kHz. Together, they are just above 1 kHz. This means that higher frequencies close to 2.5 kHz are already attenuated by a factor of 5. However, this is not a problem due to the high sensitivity of the comparator. At 2.5 kHz, a modulation voltage of 100 mV is sufficient to drive the transmitter.