Emulating PWM Capture via MA35D1 ECAP

Example Code Introduction for 64-bit NuMicro® Family

Document Information

Application	This example code uses the ECAP function to emulate PWM Capture mode.
BSP Version	Linux-6.6.y
Hardware	NuMaker-HMI-MA35D1 V1.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

In real applications, the ECAP (Enhanced Input Capture Timer) is usually used together with QEI (Quadrature Encoder Interface). This is commonly used in motor detection and control systems. QEI decodes the encoder A/B phase signals to get the motor position and speed. By combining ECAP and QEI, the system can get both the motor mechanical position and speed at the same time.

However, this sample program uses the ECAP module to emulate the PWM capture mode function. It measures the frequency and duty cycle of an external PWM or pulse signal, and extends the ECAP usage range and flexibility in real systems.

The ECAP module built into the MA35D1 series microprocessor (MPU) can provide a measurement function similar to EPWM Capture mode. It can measure parameters such as the frequency and duty cycle of an external PWM or pulse signal.

The ECAP module provides multiple independent Input Capture units. Each unit includes the following features:

- One 16-bit up-count timer/counter
- Auto-reload and Compare-Match mechanism
- Three independent input capture channels

Each capture channel can detect digital signal edge changes on the input pin (rising edge, falling edge, or both edges). When a configured edge event is detected, ECAP latches the current counter value into the related capture register.

By continuously reading the captured counter value difference between different edges (for example: rising-to-next-rising, rising-to-falling), the system can calculate:

- Input PWM period (Period)
- High level time (High Time)
- Low level time (Low Time)
- Duty cycle (Duty Cycle)

In this way, the ECAP module can emulate PWM input capture mode. It can accurately measure the timing features of an external PWM or other pulse signals. It can also work with interrupts to automatically move the measurement result to memory, reduce CPU load, and improve measurement efficiency.

1.1 Principle

In Figure 1-1, the red line shows the main path used by ECAP when this sample measures the PWM signal. The PWM signal is input through the ECAPn_IC0/1/2 pins, filtered by the Noise Filter, and then selected to CAP0, CAP1, or CAP2 via CAPSEL0/1/2 before being sent to the Input Capture Control Unit. When a configured rising edge / falling edge is detected, ECAP

latches the current ECAP_CNT counter value into ECAP_HLD0/1/2 registers. These latched values are used later to calculate the PWM period and duty cycle.

The memory map of the original application code and its CRC32 checksum are shown in Figure 1-1.

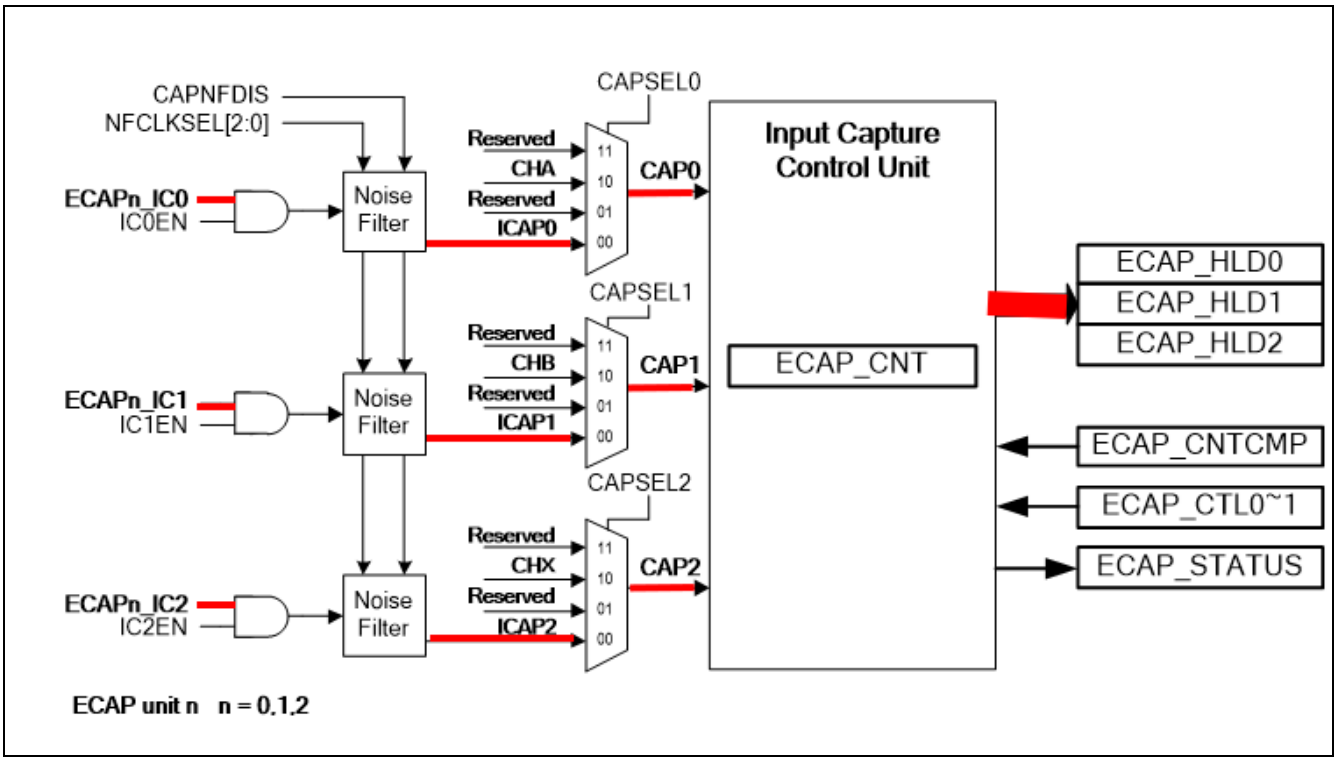

Figure 1-1 ECAP Flowchart

ECAP (Input Capture Timer/Counter) uses a "timer + edge detect" mechanism to measure the PWM high level time and the full period in one cycle and then calculate the duty cycle.

1. **Signal input and front-end processing**
    - The PWM signal is input from external pins into the ECAP channels (ECAPn_IC0, ECAPn_IC1, ECAPn_IC2).
    - Each input channel first goes through the Noise Filter to remove glitches and short noise pulses to avoid false edge triggers.
    - After filtering, a multiplexer (CAPSEL0 / CAPSEL1 / CAPSEL2) selects the signal path and connects it to the three capture input paths (CAP0, CAP1, CAP2), then sends it into the Input Capture Control Unit.
2. **ECAP counter operation**
    - Internally, ECAP shares one ECAP_CNT timer/counter. It uses the system clock, or a prescaled clock, as the count source.
    - ECAP_CNT is usually set to up-count mode. It counts up from 0. The counter value maps to discrete “time” ticks.

- When a specified input edge event occurs (rising edge or falling edge), the current ECAP_CNT value is captured into the related register (ECAP_HLDx).

3. Edge setup and capture flow

To measure PWM duty cycle, the system must get the high level time and the full period time:

- Period: Usually measured by the time difference between two continuous same edges, for example:

  Rising edge → next rising edge (ECAP_CNT delta)

- High Time: Usually measured by the time difference between:

  Rising edge → falling edge

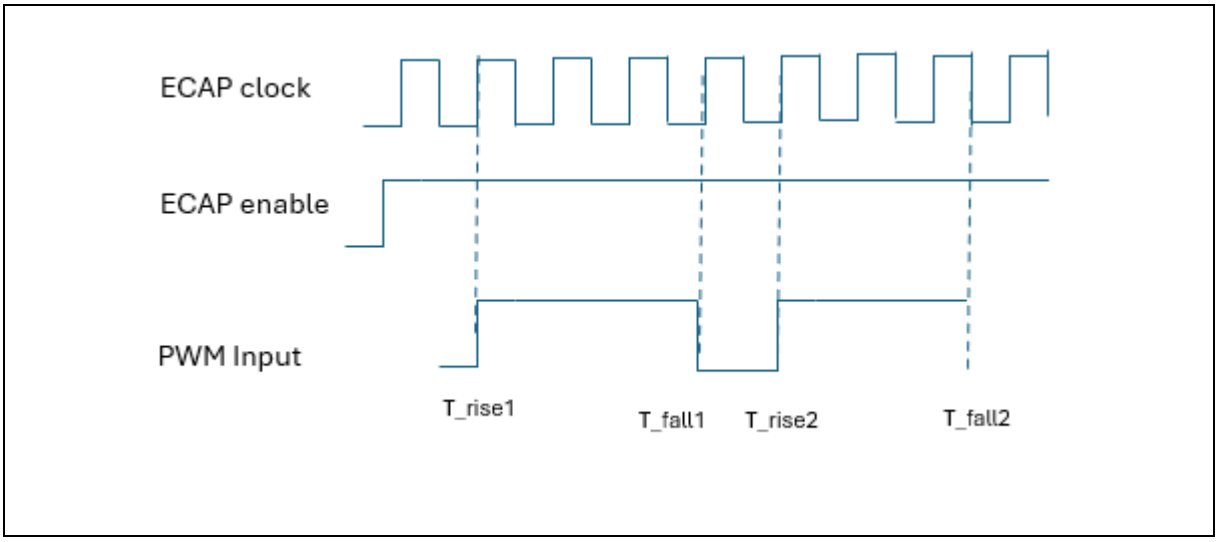

Figure 1-2 Timing Diagram

A typical configuration and flow is as follows (taking CAP0 as an example):

1) Edge-trigger configuration

   - Configure CAP0 to capture on both rising and falling edges (dual-edge capture).
   - When each capture event occurs, simultaneously sample the input pin level and record it into ECAP_Level (used to determine whether the measured time interval corresponds to a High or Low level).

2) Capture event sequence

   - Select an appropriate capture sequence (for example, rising → falling → rising or falling → rising → falling). When an event occurs, the Input Capture Control Unit latches the current ECAP_CNT value into ECAP_HLD0 register.
   - After each capture event latches ECAP_CNT into ECAP_HLD0, the hardware resets ECAP_CNT to 0 so that the next time interval measurement starts from 0. This prevents overflow of the 16-bit counter during long measurements.

3) Obtain two “interval time” count values (ticks) in time order after one complete capture sequence is finished (using rising → falling → rising as an example), two-time intervals (in ticks) between adjacent edges can be obtained:

   - Δt1_ticks: the counter value for the first interval between two adjacent edges Example: ticks from rising edge → falling edge (latched into ECAP_HLD0 register; the CPU reads ECAP_HLD0 and stores it into ECAP_TMP1).
   - Δt2_ticks: the counter value for the second interval between two adjacent edges Example: ticks from falling edge → next rising edge (latched into ECAP_HLD0 register; the CPU reads ECAP_HLD0 and stores it into ECAP_TMP2).

4) Calculate period and high/low time

   - The period (in ticks) is the sum of the two intervals:

     Period_ticks = Δt1_ticks + Δt2_ticks

   - The high-level time (High Time, ticks) and low-level time (Low Time, ticks) depend on ECAP_Level (or the starting edge of the sequence) to determine which interval corresponds to High and which corresponds to Low:

   Case A: The first interval measures High (e.g., sequence is rising → falling → rising)

   - High_ticks = Δt1_ticks
   - Low_ticks = Δt2_ticks

   Case B: The first interval measures Low (e.g., sequence is falling → rising → falling, or the initial level differs)

   - High_ticks = Δt2_ticks
   - Low_ticks = Δt1_ticks

5) Calculate duty cycle (Duty Cycle). The duty cycle is defined as “high-level time / period time” and is expressed as a percentage:

   - Duty(%) = (High_ticks / Period_ticks) × 100%

6) Convert to real time units If the ECAP counter clock frequency f_ECAP is known, the tick values can be converted to time:

   - Time(s) = ticks / f_ECAP
   - Or Time(μs) = ticks × 10^6 / f_ECAP

4. Use polling

Using polling, switch among IC0, IC1, and IC2 in sequence to measure the period and duty cycle of each input signal one by one.

5. Use interrupts or status registers to assist with reading.

Registers such as ECAP_STATUS / ECAP_CNTCMP / ECAP_CTL are used to indicate capture events, compare-match status, or control modes.

After a complete capture sequence is finished (for example, “rising → falling → rising”), an interrupt can be triggered, or a flag can be set so that the CPU reads the counter values in ECAP_HLDx and performs the calculations described above. For better efficiency, the driver notifies the user to read the period and duty cycle only after it has completed three interrupts for the “rising → falling → rising” sequence.

1.2 Demo Result

```
$ echo 1 > /sys/class/pwm/pwmchip0/export
$ echo 30000 > /sys/class/pwm/pwmchip0/pwm1/period
$ echo 20000 > /sys/class/pwm/pwmchip0/pwm1/duty_cycle
$ echo 1 > /sys/class/pwm/pwmchip0/pwm1/enable
$ cd /sys/class/pwm/pwmchip6/
$ echo 2 > export
# cat pwm2/capture
5398 3599
$
```

Set PWM0 output through PWM sysfs and use another controller (ECAP simulation) to do capture measurement for verification.

1.2.1 PWM Output Settings (pwmchip0/pwm1)

1. Go to the directory: */sys/class/pwm/pwmchip0/*.
2. Export PWM0: echo 1 > export (creates the pwm1/node).
3. Configure the parameters:
   - pwm1/period = 30000 (usually in ns): period = 30 µs (~33.33 kHz)
   - pwm1/duty_cycle = 20000 (ns): high-level time = 20 µs, duty cycle ~66.7%
4. Enable the output: echo 1 > pwm1/enable.

1.2.2 Capture Measurement (pwmchip6/IC2)

1. Switch to the ECAP-emulated PWM controller: */sys/class/pwm/pwmchip6/*.
2. On this platform, export maps to IC channels as follows:
   - 1 → IC1
   - 2 → IC2
   - 3 → IC3
3. Export IC2: echo 2 > export (maps to pwm2/).
4. Read the measurement: cat pwm2/capture.
   - If the output is 5398 3599, it can be interpreted as a captured period of about 5398 and a high time of about 3599 (duty cycle ~66.7%).

1.2.3 Why the Period Value Looks Different (Normal Behavior)

- PWM0 period/duty_cycle are configured in the time unit (ns) of the PWM output.
- The values read from pwmchip6:pwm2/capture are based on the capture module's sampling/counter clock (sample clock / counter clock).

Because different modules may use different clock sources, prescalers, and resolutions, the "read-back period value" may not numerically match the PWM configuration values. In this example, 3599/5398 ≈ 66.7% and 20000/30000 ≈ 66.7% match, indicating the duty-cycle relationship is correct. If you need an accurate time comparison (ns/us), you must convert the captured values using the capture module's clock and prescaler before comparing.

2. Code Description

2.1 Main ECAP Driver Code

drivers/pwm/pwm-ma35d1-ecap.c

```
// SPDX-License-Identifier: GPL-2.0
/*
 * Driver for the Nuvoton MA35 series ECAP controller
 *
 * Copyright (C) 2025 Nuvoton Corporation
 *               Shan-Chun Hung <schung@nuvoton.com>
 */

#include <linux/clk.h>
#include <linux/I/O.h>
#include <linux/math64.h>
#include <linux/mod_devicetable.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/pwm.h>

#define TIMEOUT_CNT       1000000

#define ECAP_CTL0     0x14
#define ECAP_CTL1     0x18
#define ECAP_STATUS       0x1C

#define ECAP_HLD(ch)  (0x04+ch*4)

#define SET_EDGE_SEL(base, ch, edge) \
    writel((readl(base)&~0x3>>(ch<<1))|(edge<<(ch<<1)), base)

struct ma35_ecap {
    struct pwm_chip chip;
    void __iomem *base;
    struct clk *clk;
};

static inline struct ma35_ecap *to_ma35_ecap(struct pwm_chip *chip)
{
    return container_of(chip, struct ma35_ecap, chip);
}

static int ma35_ecap_capture(struct pwm_chip *chip, struct pwm_device *pwm,
                  struct pwm_capture *result, unsigned long timeout)
{
    struct ma35_ecap *ecap = to_ma35_ecap(chip);
    int ch = pwm->hwpwm;
    unsigned long timeout_cnt;
    int ret = 0;

    writel(0, ecap->base + ECAP_CTL0);
    writel(0, ecap->base + ECAP_CTL1);
    writel(BIT(29) | BIT(4+ch), ecap->base + ECAP_CTL0);
```

```
50        writel(BIT(8+ch) | BIT(20+ch) | (BIT(1)<<(ch<<1)), ecap->base + ECAP_CTL1);
51
52        timeout_cnt = 0;
53        while ((readl(ecap->base + ECAP_STATUS) & BIT(8 + ch)))
54            if (++timeout_cnt > TIMEOUT_CNT)
55                goto TIMEOUT;
56
57        writel(BIT(ch), ecap->base + ECAP_STATUS);
58        writel(readl(ecap->base + ECAP_CTL0) | BIT(24), ecap->base + ECAP_CTL0);
59        timeout_cnt = 0;
60        while ((readl(ecap->base + ECAP_STATUS) & BIT(ch)) != BIT(ch))
61            if (++timeout_cnt > TIMEOUT_CNT)
62                goto TIMEOUT;
63
64        writel(BIT(ch), ecap->base + ECAP_STATUS);
65        timeout_cnt = 0;
66        while ((readl(ecap->base + ECAP_STATUS) & BIT(ch)) != BIT(ch))
67            if (++timeout_cnt > TIMEOUT_CNT)
68                goto TIMEOUT;
69
70        writel(BIT(ch), ecap->base + ECAP_STATUS);
71        result->duty_cycle = readl(ecap->base + ECAP_HLD(ch));
72
73        timeout_cnt = 0;
74        while ((readl(ecap->base + ECAP_STATUS) & BIT(ch)) != BIT(ch))
75            if (++timeout_cnt > TIMEOUT_CNT)
76                goto TIMEOUT;
77
78        result->period = result->duty_cycle + readl(ecap->base + ECAP_HLD(ch));
79        return ret;
80
81    TIMEOUT:
82        result->period = 0;
83        result->duty_cycle = 0;
84        return 0;
85    }
86
87    static int ma35_pwm_apply(struct pwm_chip *chip, struct pwm_device *pwm,
88                  const struct pwm_state *state)
89    {
90        return 0;
91    }
92
93    static const struct pwm_ops ma35_ecap_ops = {
94        .apply = ma35_pwm_apply,
95        .capture = ma35_ecap_capture,
96        .owner = THIS_MODULE,
97    };
98
99    static int ma35_ecap_probe(struct platform_device *pdev)
100   {
101       struct ma35_ecap *ecap;
102       int ret;
103
104       ecap = devm_kzalloc(&pdev->dev, sizeof(*ecap), GFP_KERNEL);
105       if (IS_ERR(ecap))
```

```
106             return PTR_ERR(ecap);
107
108         ecap->base = devm_platform_ioremap_resource(pdev, 0);
109         if (IS_ERR(ecap->base))
110             return PTR_ERR(ecap->base);
111
112         ecap->clk = of_clk_get(pdev->dev.of_node, 0);
113         if (IS_ERR(ecap->clk)) {
114             dev_err(&pdev->dev, "failed to get epwm clock\n");
115             ret = PTR_ERR(ecap->clk);
116             return -ENOENT;
117         }
118
119         ret = clk_prepare_enable(ecap->clk);
120         if (ret)
121             return -ENOENT;
122
123         platform_set_drvdata(pdev, ecap);
124         ecap->chip.dev = &pdev->dev;
125         ecap->chip.ops = &ma35_ecap_ops;
126         ecap->chip.base = -1;
127         ecap->chip.npwm = 3;
128
129         ret = pwmchip_add(&ecap->chip);
130         if (ret < 0)
131             return dev_err_probe(&pdev->dev, ret, "unable to add ecap chip");
132
133         return 0;
134     }
135
136     static const struct of_device_id ma35_ecap_of_match[] = {
137         {.compatible = "nuvoton,ma35d1-ecap" },
138         { }
139     };
140
141     MODULE_DEVICE_TABLE(of, ma35_ecap_of_match);
142
143     static struct platform_driver ma35_ecap_driver = {
144         .probe = ma35_ecap_probe,
145         .driver = {
146                 .name = "ma35-ecap",
147                 .of_match_table = ma35_ecap_of_match,
148                  },
149     };
150
151     module_platform_driver(ma35_ecap_driver);
152
153     MODULE_LICENSE("GPL");
154     MODULE_AUTHOR("Shan-Chun Hung <schung@nuvoton.com>");
155     MODULE_ALIAS("platform:ma35-ecap");
```

- This file implements the Linux driver for the ECAP controller of the MA35 family MPU.
- It wraps the ECAP module as a pwm_chip and registers it under the Linux PWM framework.

- Main functions are:
  - Only do "PWM capture" measurement (ma35_ecap_capture()).
  - It does not implement PWM generation (ma35_pwm_apply() just returns 0).

In practice, this allows upper layers (other drivers or user-space tools) to call the PWM subsystem to measure an external PWM signal's "period + duty cycle."

1. Turn ECAP into a "PWM measurement device" by registering the ECAP driver with the Linux PWM subsystem.

- In ma35_ecap_probe():
  - Retrieve the ECAP register base address (ecap->base) and clock (ecap->clk).
  - Initialize ecap->chip and declare that it has 3 channels.
  - Register it with the Linux PWM framework using pwmchip_add().
- After that, the system treats this ECAP as a PWM controller. Upper layers can call .capture to "measure PWM" rather than generate PWM.

2. How does ma35_ecap_capture() measure PWM?

The overall flow of a single capture is roughly:

- Clear all configuration and old status flags
  - Clear ECAP_CTL0 and ECAP_CTL1 to 0, then reconfigure the selected channel's capture mode and edge settings.
  - Clear the relevant bits in ECAP_STATUS to prevent events from the previous run from interfering.
- Start the ECAP counter and wait for 3 edge events
  - Enable the counter (the enable bit in ECAP_CTL0).
  - Wait for the event bit in ECAP_STATUS to become 1 three times in sequence; each time indicates that another edge has been captured:
    - 1st time: used only for synchronization; the driver does not read any value.
    - 2nd time: read ECAP_HLD(ch) and treat it as result->duty_cycle (high-level time).
    - 3rd time: read ECAP_HLD(ch) again and add it to the previous duty_cycle value to obtain result->period (a full period).
- If it times out, return 0
  - If any wait loop exceeds TIMEOUT_CNT, set both period and duty_cycle to 0, indicating that no valid PWM was captured this time.

2.2 Add MA35D1 ECAP Driver into Linux PWM Subsystem

drivers/pwm/Makefile

```
obj-$(CONFIG_PWM_MA35D1_ECAP)        += pwm-ma35d1-ecap.o
```

The purpose of this line:

- When Kconfig selects CONFIG_PWM_MA35D1_ECAP=y or m,
- It builds *pwm-ma35d1-ecap.o* into the kernel or as a loadable module, i.e., it includes your MA35 ECAP driver in the PWM framework build process.

drivers/pwm/Kconfig

```
config PWM_MA35D1_ECAP
        tristate "Nuvoton MA35D1 ECAP support"
        help
          Captures waveform driver for Nuvoton MA35D1.

          To compile this driver as a module, choose M here: the module
          will be called pwm-ma35d1-ecap.
```

- Define a new Kconfig option PWM_MA35D1_ECAP.
- tristate means it can be set to y/m/n:
  - y: built into the kernel
  - m: built as a module *pwm-ma35d1-ecap.ko*
  - n: not built
- The help text explains that this driver is used for waveform capture on the MA35D1.

2.3 Declare ECAP Hardware Node in the Device Tree

arch/arm64/boot/dts/nuvoton/ma35d1.dtsi

```
ecap0: ecap0@40b40000 {
        compatible = "nuvoton,ma35d1-ecap";
        reg = <0x0 0x40b40000 0x0 0x10000>;
        clocks = <&clk ECAP0_GATE>;
        status = "disabled";
};
```

This section in the dtsi defines the ECAP0 hardware block:

- ecap0@40b40000: The MMIO base address of ECAP0 is 0x40b40000, with a size of 0x10000.
- compatible = "nuvoton,ma35d1-ecap";
  This corresponds to ma35_ecap_of_match` in your driver, allowing the kernel to match this device to the correct driver using the compatible string.

- clocks = <&clk ECAP0_GATE>;
  Specifies which clock gate is used.
- status = "disabled";
  Disabled by default; it can be enabled later in the board-specific .dts when actually needed.

2.4 Configure Pins and Enable ECAP0 in the SOM Board DTS

The ***pinctrl*** configuration in ***ma35d1-som-256m.dts.***

```
ecap0 {
        pinctrl_ecap0: ecap0grp{
                nuvoton,pins =
                        <SYS_GPD_MFPH_PD13MFP_ECAP0_IC0 &pcfg_default>,
                        <SYS_GPD_MFPH_PD14MFP_ECAP0_IC1 &pcfg_default>,
                        <SYS_GPD_MFPH_PD15MFP_ECAP0_IC2 &pcfg_default>;
        };
};
```

This section does the following:

- Create a pinctrl group pinctrl_ecap0.
- Switch the three physical pins PD13 / PD14 / PD15 to the three ECAP0 input functions:
  - PD13 → ECAP0_IC0
  - PD14 → ECAP0_IC1
  - PD15 → ECAP0_IC2
- &pcfg_default indicates that these pins use the default electrical characteristics (pull-up/down, drive strength, etc.).

In other words, this config “wires” the external PWM/pulse signals to the three ECAP0 capture channels.

Add an ECAP0 node in the same DTS.

```
&ecap0 {
        status = "okay";
        #pwm-cells = <2>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_ecap0>;
};
```

This section enables the ecap0 node that was previously declared in the dtsi:

- status = "okay";
  Tells the kernel that ECAP0 is routed out and used on this board.

- #pwm-cells = <2>;
  Informs the PWM framework that this controller's PWM specifier requires 2 cells (e.g., a format like <channel period>), which is consistent with common PWM bindings.
- pinctrl-names = "default";
- pinctrl-0 = <&pinctrl_ecap0>;
  Applies the previously configured pin group at boot so the pins are switched to the ECAP function.

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - Linux-6.6.y
- IDE version
  - Buildroot_2024

3.2 Hardware Requirements

- Circuit components
  - NuMaker-HMI-MA35D1 V1.0

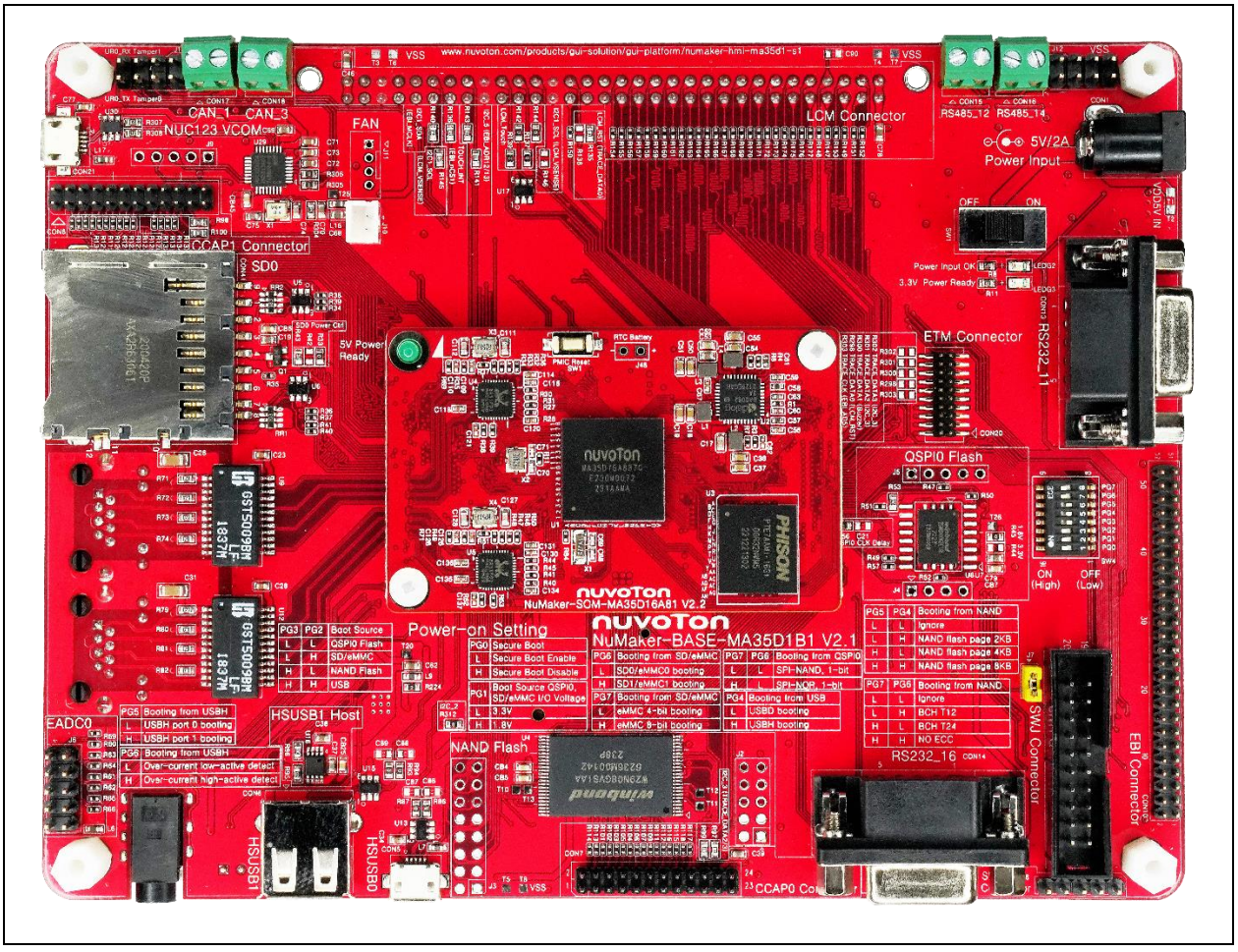

Figure 3-1 NuMaker-HMI-MA35D1 V1.0

- Wiring
  - The blue box is ECAP0_IC1 (PD14)
  - The yellow box is ECAP0_IC0 (PD15)

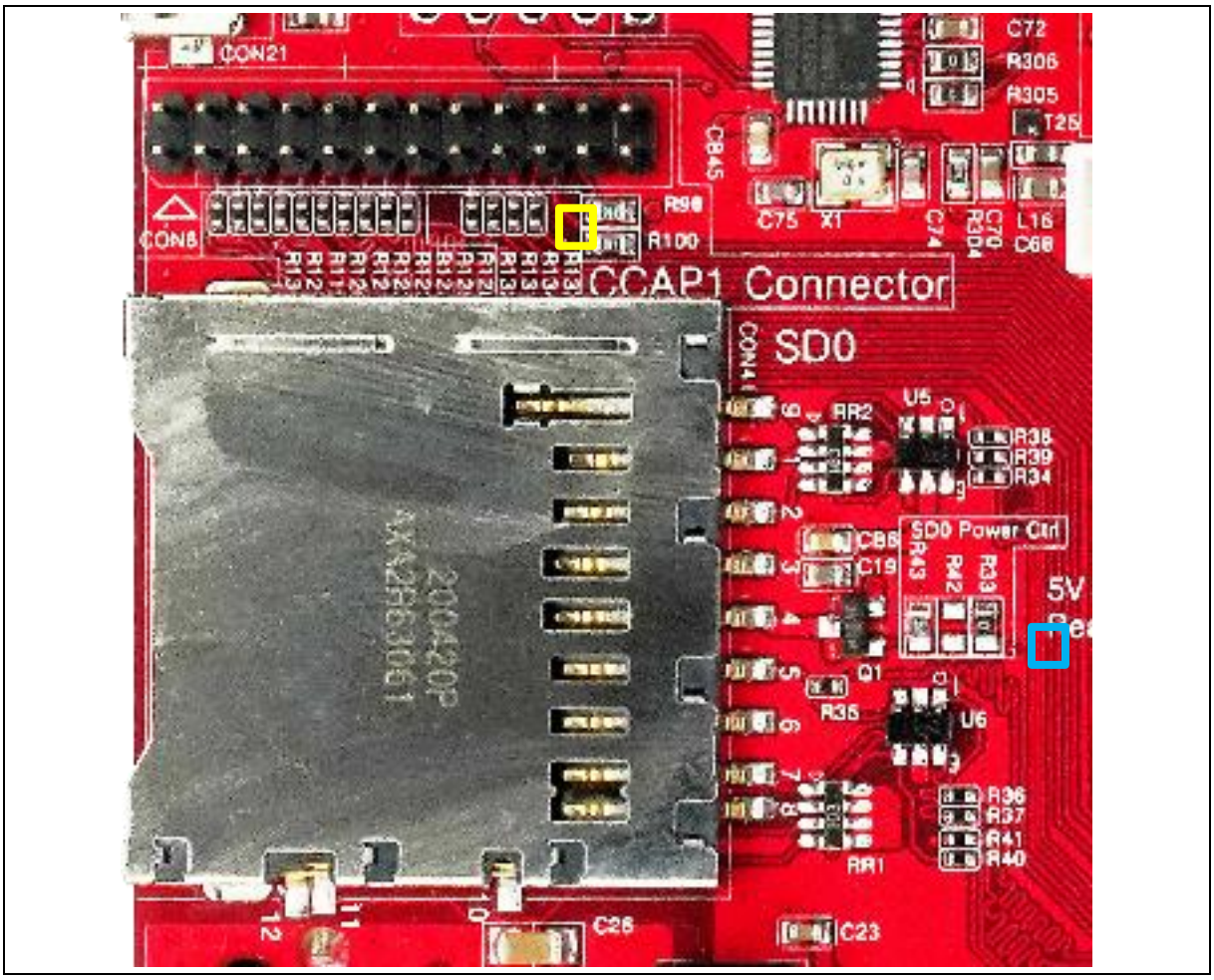

Figure 3-2 ECAP Pins

- LCD connector pin 37 is currently EPWM CH1. You can use an external wire to connect it to ECAP0_IC0 (PD15), so the PWM signal output from EPWM can be fed into ECAP0_IC0 for measurement. Other pins can be connected in the same way to the other ECAP input pins. Since this is a relatively special wiring/use case, there is currently no standard board that supports it directly—you need to implement the wiring yourself.

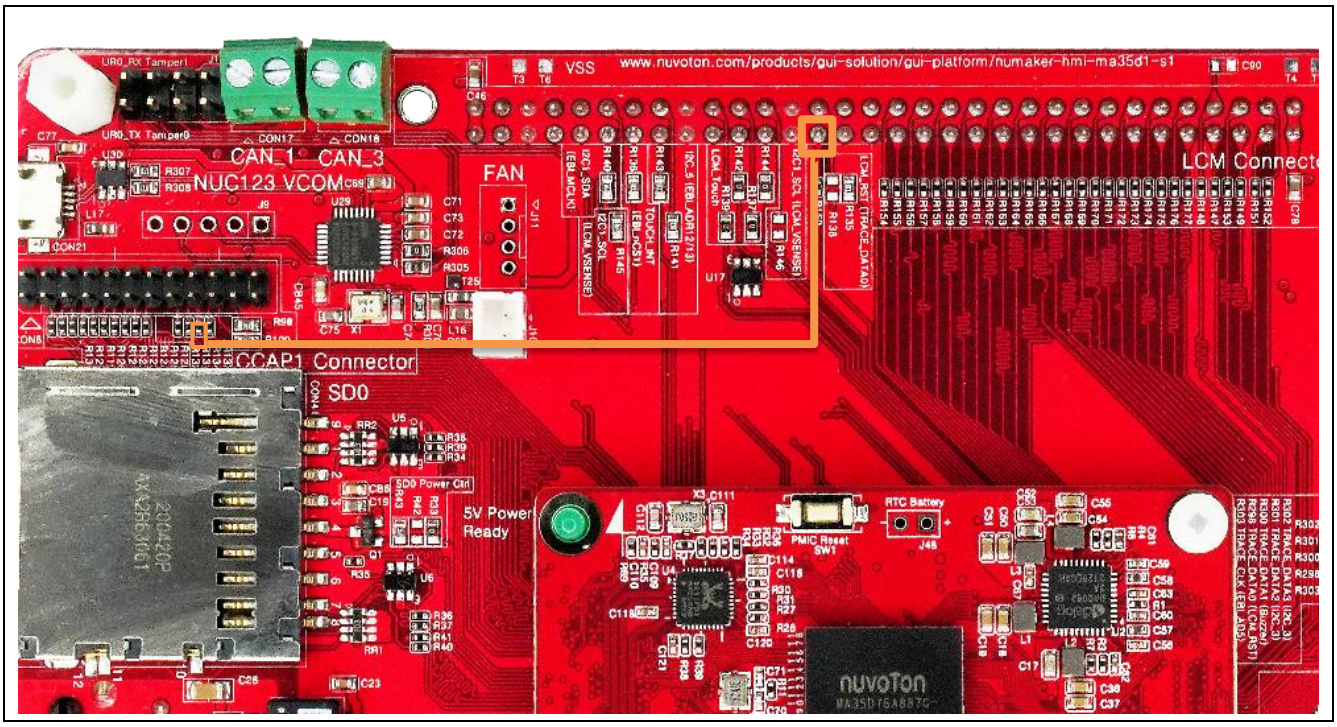

Figure 3-3 EPWM_CH1Connect ECAP0_IC0

4. Directory Information

The directory structure is shown below.

EC_MA35D1_ECAP_Emulate_PWM_Capture_V1.00	
linux-6.6.y	Linux source code
arch/arm64/boot/dts/nuvoton	Nuvoton ARM64 device tree files
drivers/pwm	PWM drivers

Figure 4-1 Directory Structure

5. Example Code Execution

The MA35D1 will build a Linux image by default. For this example to run successfully, you must modify the relevant source code under the Linux directory. Below are the Buildroot-related steps, using an SD card as the external storage device:

1. After entering the MA35D1 Buildroot environment, perform the first build. This will generate the required source code and image files.

```
user@bb867ccf5905:~/shared/buildroot_2024$ make numaker_som_ma35d16a81_defconfig
user@bb867ccf5905:~/shared/buildroot_2024$ make
```

2. Go to the corresponding linux-6.6.y path (it may differ depending on your setup):

```
user@bb867ccf5905:~/shared/buildroot_2024/output/build/linux-custom$
```

3. Copy all files from each linux-6.6.y folder in the example project and overwrite the corresponding files under linux-custom. **Note:** This is a replacement/overwrite, not copying the entire example folder directly into linux-custom.

4. Enter the MA35D1 linux-6.6.y configuration menu.

```
user@bb867ccf5905:~/shared/buildroot_2024$ make linux-menuconfig
```

5. Enable the ECAP feature under PWM in linux-menuconfig.

```
Device Drivers  --->
[*] Pulse-Width Modulation (PWM) Support  --->
<*>   Nuvoton MA35D1 ECAP support
```

6. Rebuild linux-custom.

```
make linux-rebuild
```

7. Rebuild the image.

```
make
```

8. Use NuWriter to program the packed image.

9. Set the evaluation board's Power-on setting DIP switches to boot from SD0, then power on the MA35D1 evaluation board.

6. Revision History

Date	Revision	Description
2026.04.13	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.