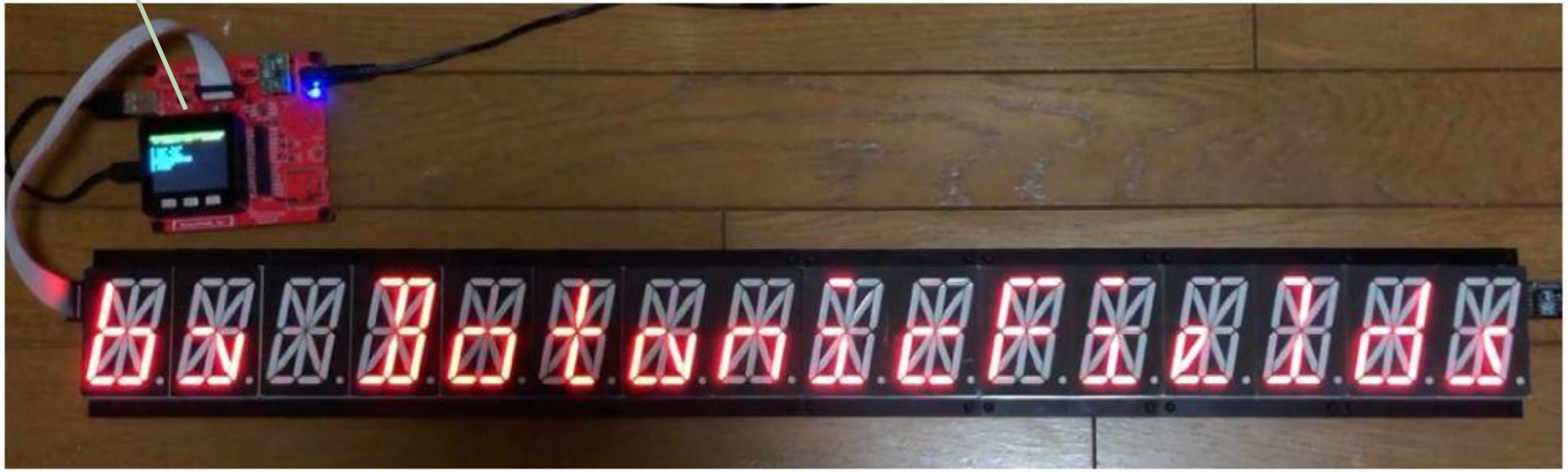
How to make GPIO Signals to Control Hardware

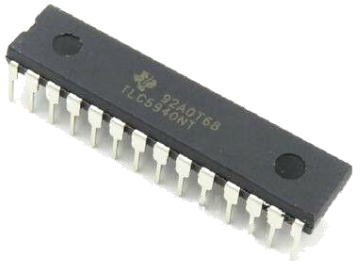
M5Stack User Meeting Vol. 7
At Shiodome, Tokyo
2020/2/4

16 Segment LED Display

M5Stack



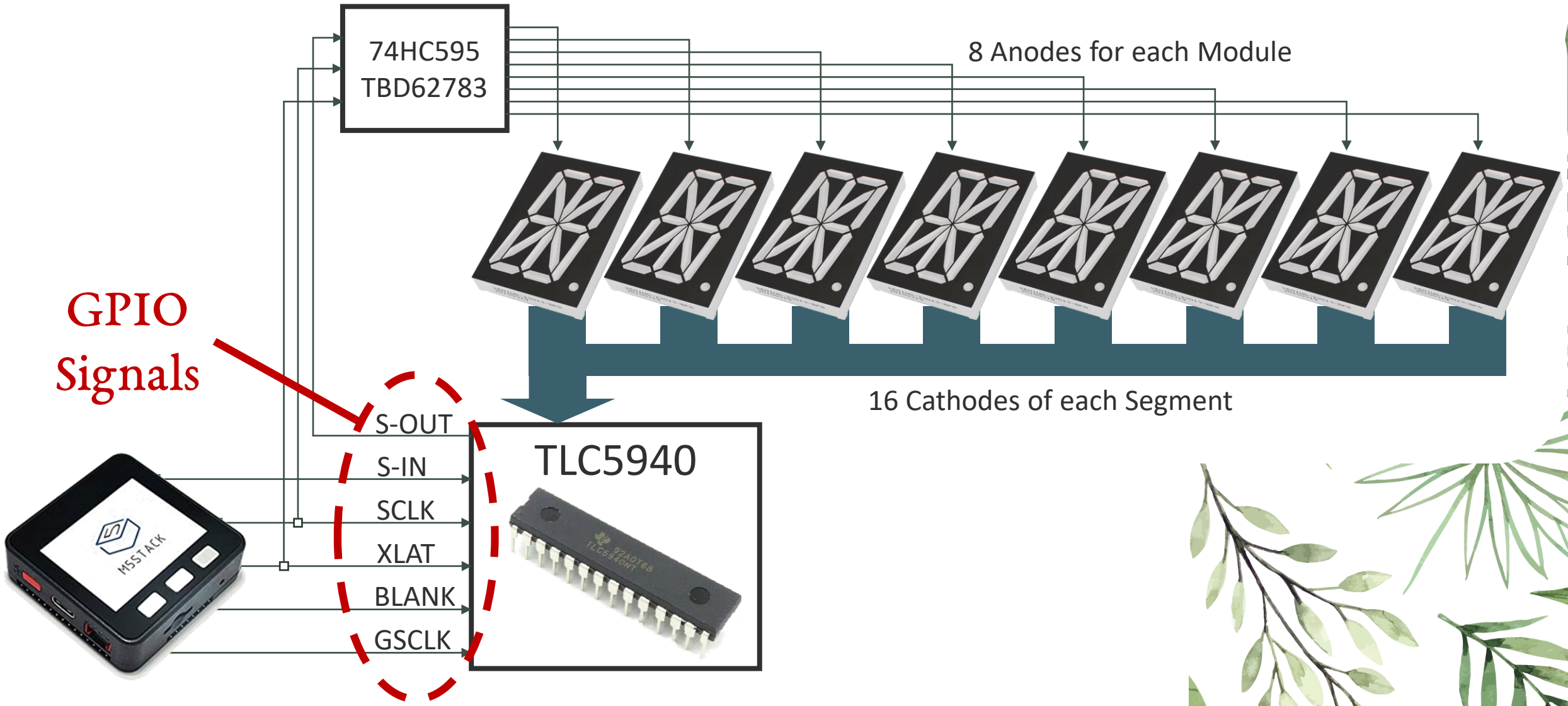
Controlled by M5Stack and TLC5940



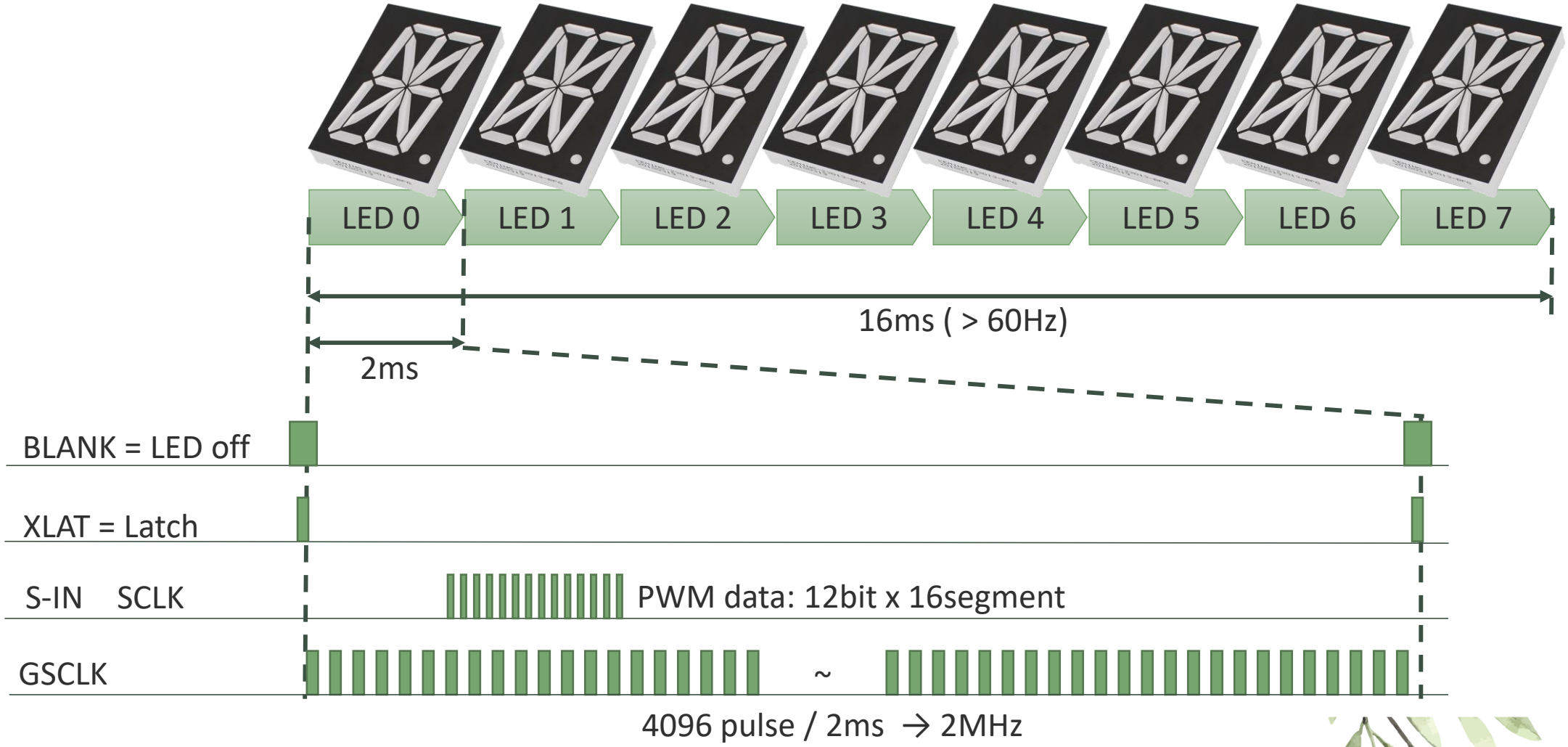
I.

Structure
Interface
Timing

Structure and Interface



Timing


LED 0
LED 1
LED 2
LED 3
LED 4
LED 5
LED 6
LED 7
16ms (> 60Hz)
2ms
BLANK = LED off
XLAT = Latch
S-IN SCLK
PWM data: 12bit x 16segment
GSCLK
~
4096 pulse / 2ms → 2MHz

2.

Wave Forms

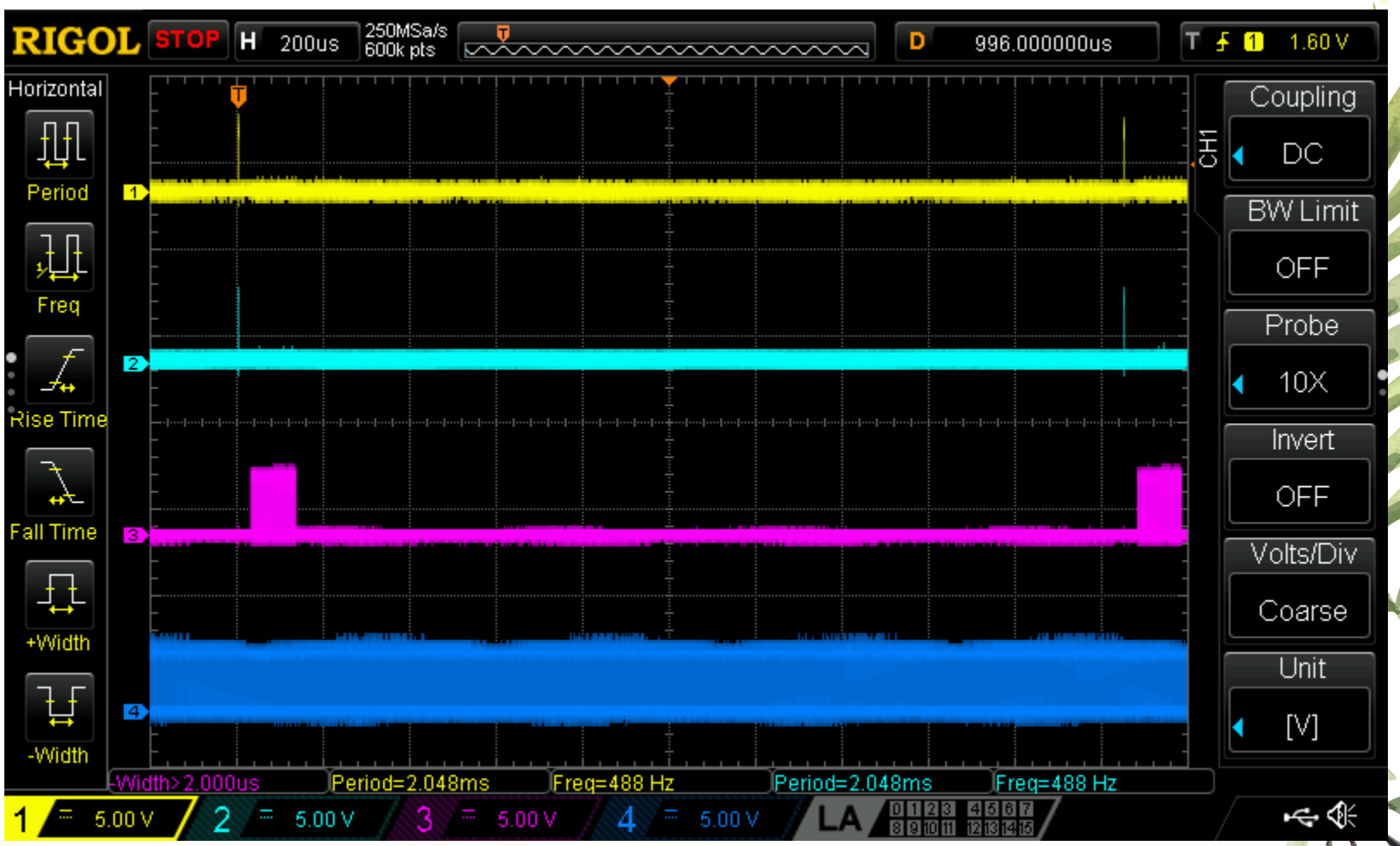
Wave Forms

BLANK

XLAT

SCLK

GSCLK



BLANK, XLAT: 2.048ms

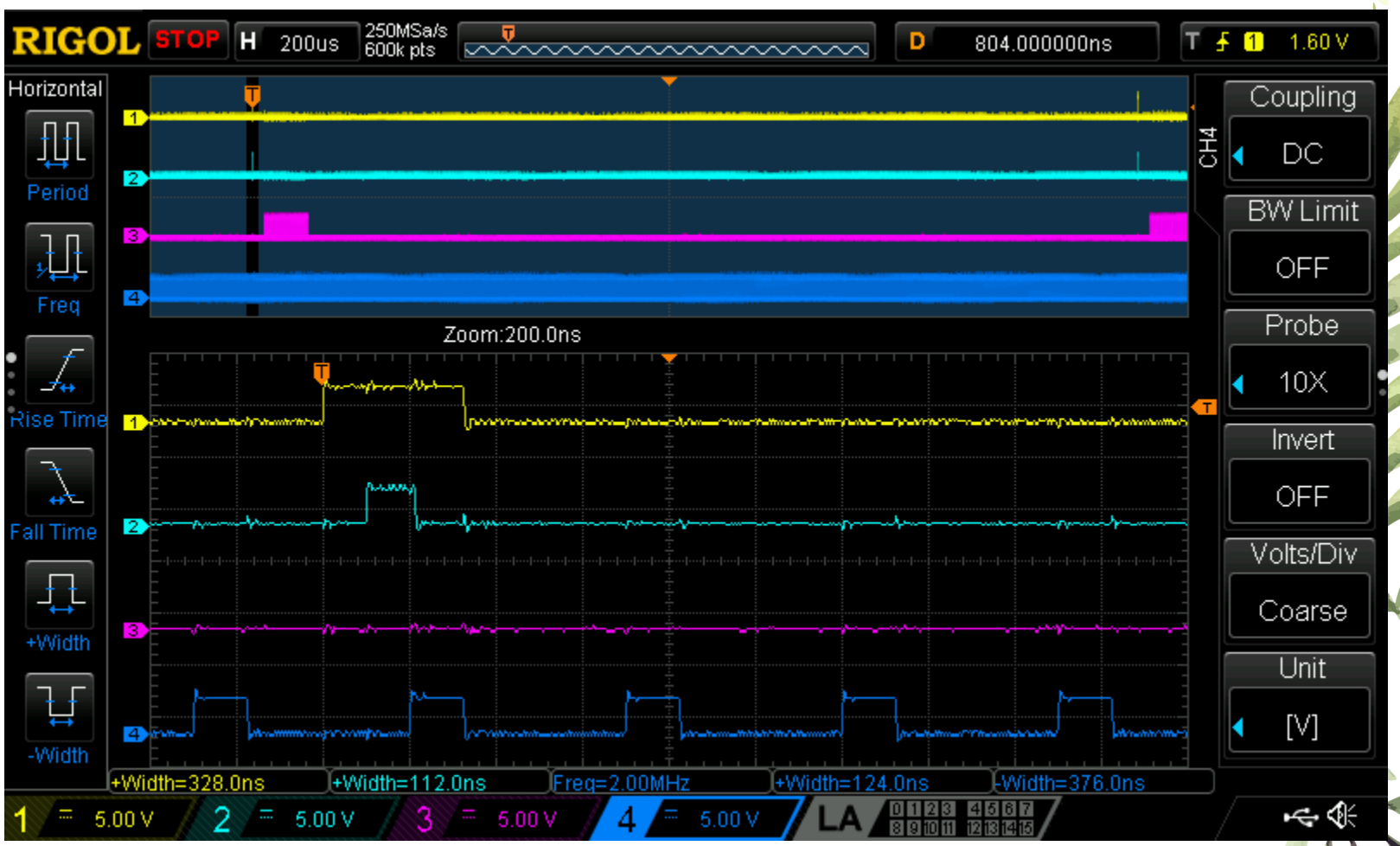
Wave Forms

BLANK

XLAT

SCLK

GSCLK



BLANK: 328ns ⟷

XLAT: 112ns ↔

GSCLK: 2MHz 124ns/376ns

Wave Forms

BLANK

XLAT

SCLK

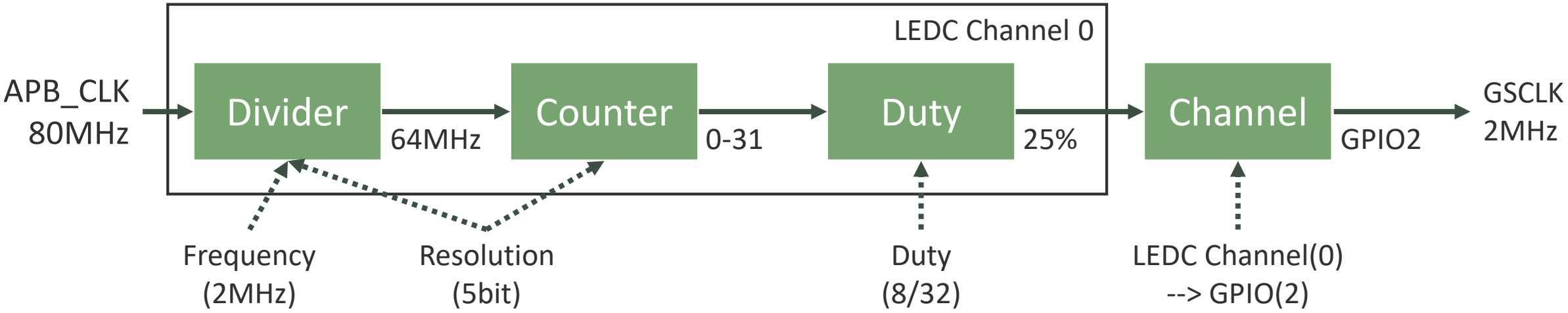
GSCLK

SCLK: 8MHz 60ns/65ns

3.

Genarate Signals with ESP32

LEDC for GSCLK



sketch_feb02a §

```
// pin definition
const uint8_t PIN_GSCLK =  2;  // GPIO2

// LEDC for GS(Gray-scale) clock
const uint8_t  CHN = 0;     // channel
const double   FRQ = 2e6;   // frequency 2MHz
const uint8_t  RSL = 5;     // resolution 5bit = 32
const uint32_t DTY = 8;     // duty 25% .. 8 / 32

// start GSCLK
void ledc_init() {
  ledcSetup(CHN, FRQ, RSL);
  ledcAttachPin(PIN_GSCLK, CHN);
  ledcWrite(CHN, DTY);
}
```

$$\text{Frequency} \times 2^{\text{Resolution}} <= \text{APB\_CLK}$$

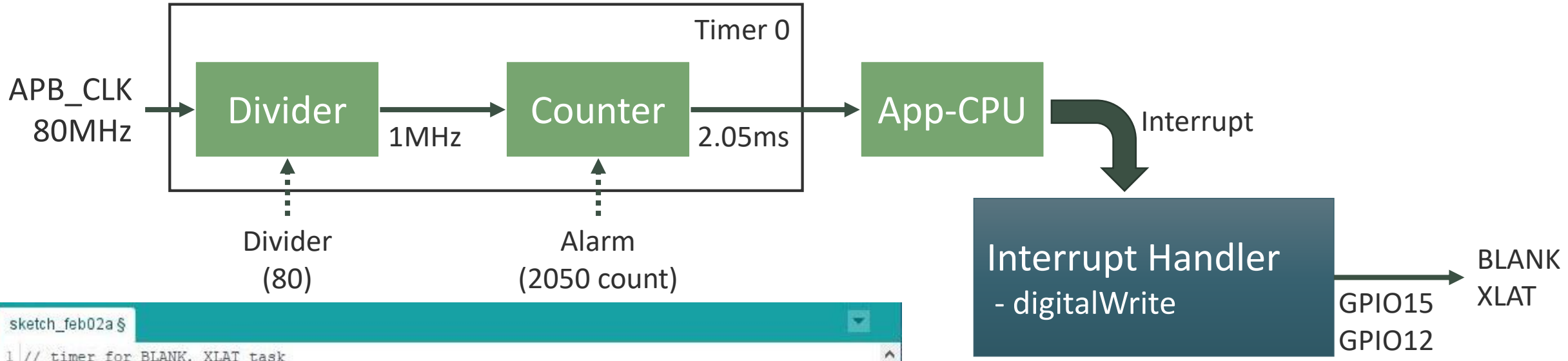
$$2\text{MHz} \times 32 = 64\text{MHz} < 80\text{MHz}$$

Qiita

ESP32のLED_PWMにおいて元クロックを生成する仕組み

@BotanicFields

Timer Interrupt for BLANK and XLAT



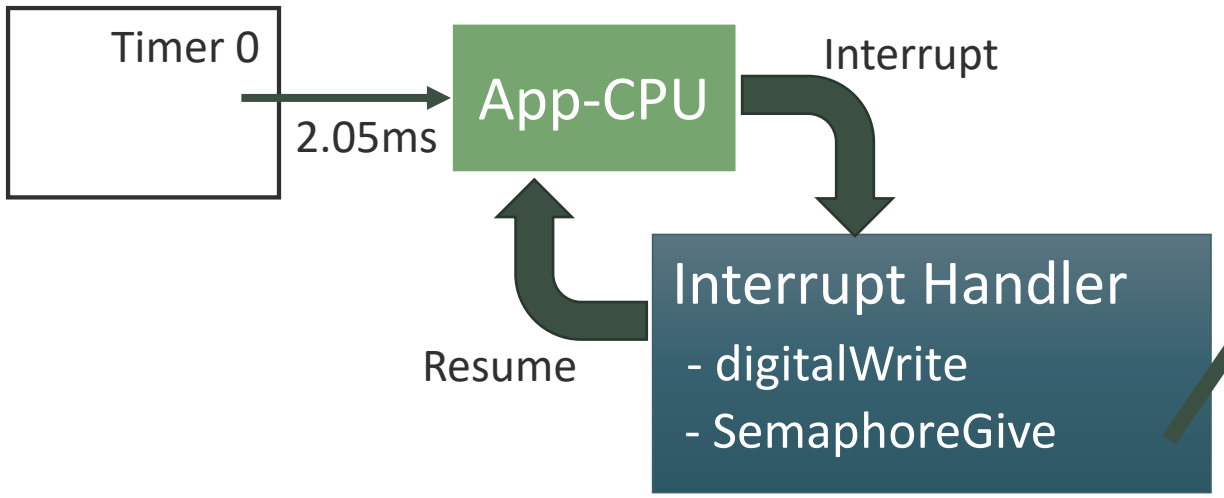
sketch_feb02a §

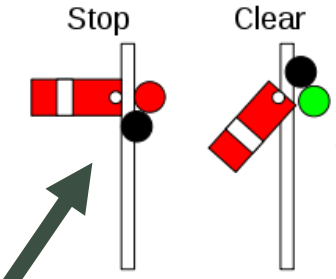
```
// timer for BLANK, XLAT task
const uint8_t  TIMER_NUM = 0;     // hardware timer number
const uint16_t TIMER_DIV = 80;    // timer divider to make 1MHz from 80MHz
const uint64_t TIMER_ALM = 2050;  // interrupt every 2.050ms

// timer interruption
hw_timer_t* TLC5940_timer = NULL;

// set up timer
void timer_init() {
  // start timer
  TLC5940_timer = timerBegin(TIMER_NUM, TIMER_DIV, true);       // increment mode
  timerAttachInterrupt(TLC5940_timer, &TLC5940_onTimer, true);  // edge mode
  timerAlarmWrite(TLC5940_timer, TIMER_ALM, true);              // auto-reload mode
  timerAlarmEnable(TLC5940_timer);
}
```

sketch_feb02a §

```
// pin definition
const uint8_t PIN_XLAT  = 12;  // GPIO12
const uint8_t PIN_BLANK = 15;  // GPIO15

/// handle timer interruption
void IRAM_ATTR TLC5940_onTimer(){
  digitalWrite(PIN_BLANK, HIGH);
  digitalWrite(PIN_XLAT,  HIGH);
  digitalWrite(PIN_XLAT,  LOW);
  digitalWrite(PIN_BLANK, LOW);
```

Semaphore for PWM Data





Non-privileged Task
- SemaphoreTake
- Make data
- Send data

sketch_feb02a §

```
// handle timer interruption
void IRAM_ATTR TLC5940_onTimer(){
  // BLANK, XLAT
  digitalWrite(PIN_BLANK, HIGH);
  digitalWrite(PIN_XLAT,  HIGH);
  digitalWrite(PIN_XLAT,  LOW);
  digitalWrite(PIN_BLANK, LOW);
  // wake TLC5940-task up
  BaseType_t xHigherPriorityTaskWoken;
  xHigherPriorityTaskWoken = pdFALSE;
  xSemaphoreGiveFromISR(TLC5940_timerSemaphore, &xHigherPriorityTaskWoken);
}
```

sketch_feb02a §

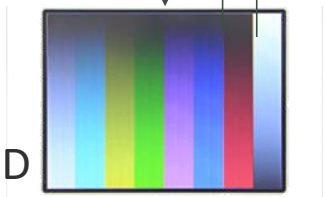
```
// create semaphore
TLC5940_timerSemaphore = xSemaphoreCreateBinary();

// standby TLC5940 task
xTaskCreatePinnedToCore(TLC5940_task,   // Function to implement the task
                        "TLC5940_task", // Name of the task
                        4096,           // Stack size in words
                        NULL,           // Task input parameter
                        2,              // Priority of the task
                        NULL,           // Task handle.
                        1);             // Core where the task should run
```

sketch

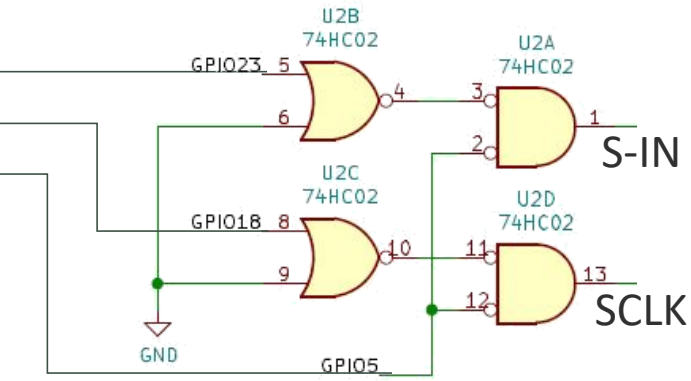
```
// send GS data through SPI
void TLC5940_task(void * pvParameters) {
  for(;;) {
    // wait for semaphore
    xSemaphoreTake(TLC5940_timerSemaphore, portMAX_DELAY);
    // update gs
    TLC5940_update(count);
    TLC5940_spi_gs_send(count);
  }
}
```

SPI(Serial Peripheral Interface) for S-IN, SCLK

GPIO23(MOSI)
GPIO18(SCK)
GPIO5
GPIO4
GPIO14



LCD



TLC5940

```sketch_feb02a §
// initialize SPI
void spi_init() {
  TLC5940_vspi = new SPIClass(VSPI);
  TLC5940_vspi->begin(PIN_SCLK, PIN_MISO, PIN_MOSI, PIN_SS);
  pinMode(PIN_SS, OUTPUT);
}
```

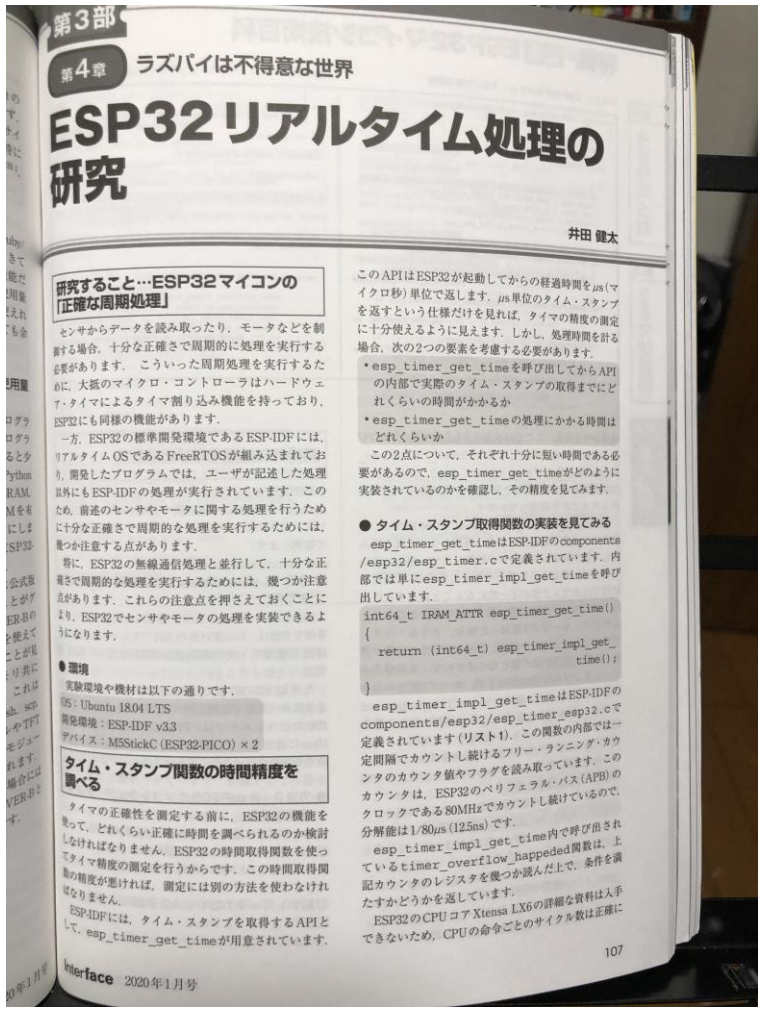
If some devices share an SPI,
negatively affects the performance.

```
// make and send GS data through SPI
void spi_gs_send(uint16_t count) {
  digitalWrite(PIN_VPRG, LOW);  // select GS
  TLC5940_vspi->beginTransaction(SPISettings(SPI_CLK, MSBFIRST, SPI_MODE0));
  digitalWrite(PIN_SS, LOW); // active

  // for each groups
  for(int k = TLC5940_GRP - 1; k >= 0; k--) {
    TLC5940_vspi->transfer(0x80 >> col);                    // column selected
    TLC5940_vspi->transfer(TLC5940_image_dot[k][col] >> 8);  // dot
    for(int i = 0; i < TLC5940_SEG * 3 / 2; i++)
      TLC5940_vspi->transfer(gs_reg[i]);
  }
  digitalWrite(PIN_SS, HIGH); // inactive
  TLC5940_vspi->endTransaction();
}
```

4.

Conclusion and Reference

Reference


第3部
第4章 ラズパイは不得意な世界

ESP32リアルタイム処理の研究

井田 健太

研究すること…ESP32マイコンの「正確な周期処理」

センサからデータを読み取ったり，モータなどを制御する場合，十分な正確さで周期的に処理を実行する必要があります．こういった周期処理を実行するために，大抵のマイクロ・コントローラはハードウェア・タイマによるタイマ割り込み機能を持っており，ESP32にも同様の機能があります．

一方，ESP32の標準開発環境であるESP-IDFには，リアルタイムOSであるFreeRTOSが組み込まれており，開発したプログラムでは，ユーザが記述した処理以外にもESP-IDFの処理が実行されています．このため，前述のセンサやモータに関する処理を行うために十分な正確さで周期的な処理を実行するためには，幾つか注意する点があります．

特に，ESP32の無線通信処理と並行して，十分な正確さで周期的な処理を実行するためには，幾つか注意点があります．これらの注意点を押さえておくことにより，ESP32でセンサやモータの処理を実装できるようになります．

● 環境

実験環境や機材は以下の通りです．

OS：Ubuntu 18.04 LTS
開発環境：ESP-IDF v3.3
デバイス：M5StickC（ESP32-PICO）×2

タイム・スタンプ関数の時間精度を調べる

タイマの正確性を測定する前に，ESP32の機能を使って，どれくらい正確に時間を調べられるのか検討しなければなりません．ESP32の時間取得関数を使ってタイマ精度の測定を行うからです．この時間取得関数の精度が悪ければ，測定には別の方法を使わなければなりません．

ESP-IDFには，タイム・スタンプを取得するAPIとして，esp_timer_get_timeが用意されています．このAPIはESP32が起動してからの経過時間をμs（マイクロ秒）単位で返します．μs単位のタイム・スタンプを返すという仕様だけを見れば，タイマの精度の測定に十分使えるように見えます．しかし，処理時間を計る場合，次の2つの要素を考慮する必要があります．

- esp_timer_get_timeを呼び出してからAPIの内部で実際のタイム・スタンプの取得までにどれくらいの時間がかかるか
- esp_timer_get_timeの処理にかかる時間はどれくらいか

この2点について，それぞれ十分に短い時間である必要があるので，esp_timer_get_timeがどのように実装されているのかを確認し，その精度を見てみます．

● タイム・スタンプ取得関数の実装を見てみる

esp_timer_get_timeはESP-IDFのcomponents/esp32/esp_timer.cで定義されています．内部では単にesp_timer_impl_get_timeを呼び出しています．

```
int64_t IRAM_ATTR esp_timer_get_time()
{
  return (int64_t) esp_timer_impl_get_
                                time();
}
```

esp_timer_impl_get_timeはESP-IDFのcomponents/esp32/esp_timer_esp32.cで定義されています（**リスト1**）．この関数の内部では一定間隔でカウントし続けるフリー・ランニング・カウンタのカウンタ値やフラグを読み取っています．このカウンタは，ESP32のペリフェラル・バス（APB）のクロックである80MHzでカウントし続けているので，分解能は1/80μs（12.5ns）です．

esp_timer_impl_get_time内で呼び出されているtimer_overflow_happeded関数は，上記カウンタのレジスタを幾つか読んだ上で，条件を満たすかどうかを返しています．

ESP32のCPUコアXtensa LX6の詳細な資料は入手できないため，CPUの命令ごとのサイクル数は正確に

107

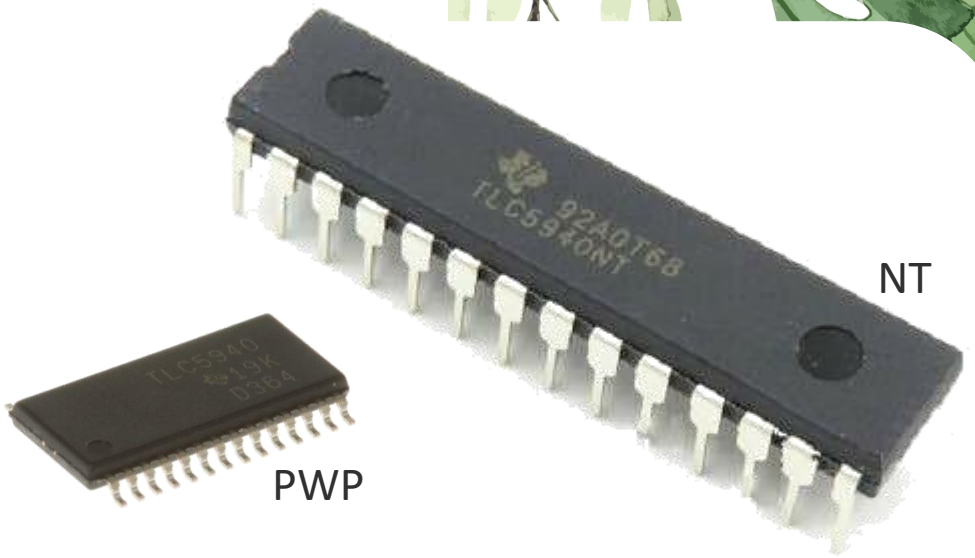
Interface 2020年1月号

ハードウェア・タイマがダントツで優秀

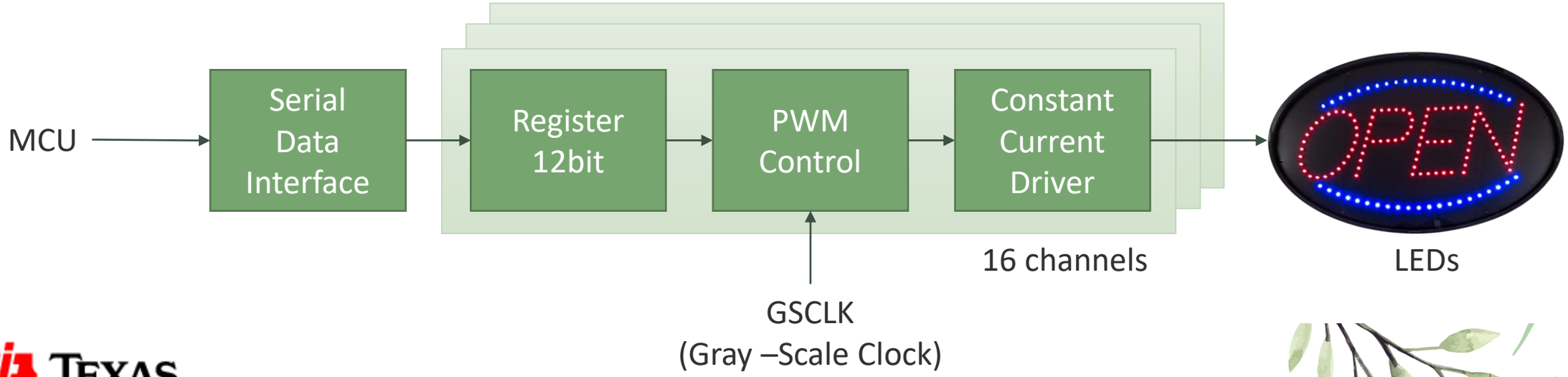
Hardware timers are excellent by far.

TLC5940 PWM

TLC5940



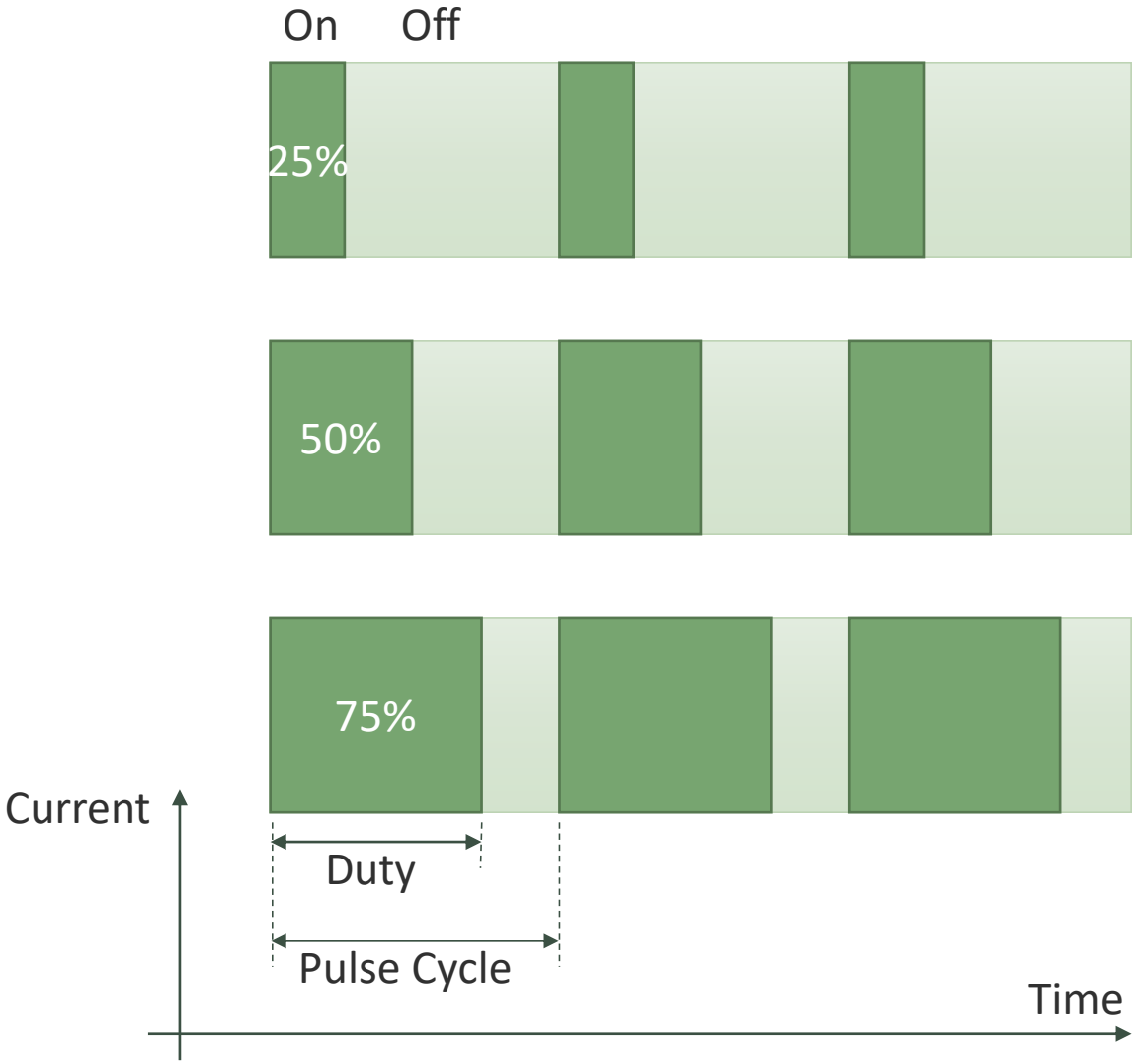
- 16 Channel PWM controller for LED
- 4096 Levels of Brightness
- Constant Current Sink Drivers up to 120mA

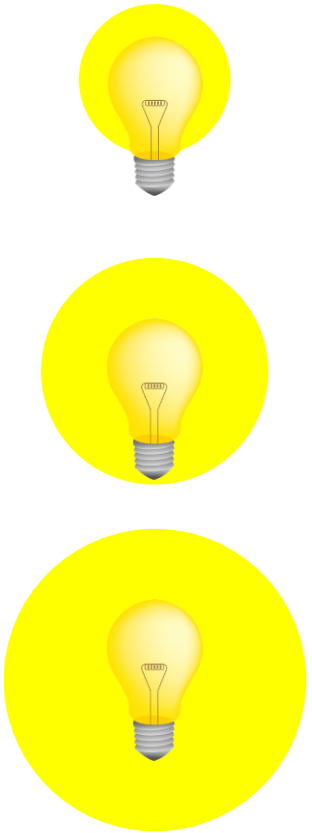


Texas Instruments

It has been available since 2005.

Pulse Width Modulation





Thanks!