matrix_led ソースファイル

美都

2020 年 5 月 19 日

目次

1 Cargo.toml ソース

```
[package]
authors = ["mito␣<mito@laki.jp>"]
edition = "2018"
readme = "README.md"
name = "matrixled"
version = "0.1.0"

[dependencies]
cortex-m = "0.6.0"
cortex-m-rt = "0.6.10"
cortex-m-semihosting = "0.3.3"
panic-halt = "0.2.0"

# Uncomment for the panic example.
# panic-itm = "0.4.1"

# Uncomment for the allocator example.
# alloc-cortex-m = "0.3.5"

# Uncomment for the device example.
# Update `memory.x`, set target to `thumbv7em-none-eabihf` in `.cargo/config`,
# and then use `cargo build --examples device` to build it.
# [dependencies.stm32f3]
# features = ["stm32f303", "rt"]
# version = "0.7.1"

[dependencies.stm32f4]
version = "0.10.0"
features = ["stm32f401", "rt"]

[dependencies.misakifont]
path = "../misakifont/misakifont"

# this lets you use `cargo fix`!
[[bin]]
name = "matrixled"
test = false
bench = false

[profile.release]
codegen-units = 1 # better optimizations
debug = true # symbols are nice and they don't increase the size on Flash
lto = true # better optimizations
```

2　main.rs ソース

```
#![no_std]
#![no_main]

// pick a panicking behavior
extern crate panic_halt; // you can put a breakpoint on `rust_begin_unwind` to catch panics
                         // extern crate panic_abort; // requires nightly
                         // extern crate panic_itm; // logs messages over ITM; requires ITM support
                         // extern crate panic_semihosting; // logs messages to the host stderr; requires a debugger

use cortex_m::interrupt::free;
use cortex_m_rt::entry;
use stm32f4::stm32f401;
use stm32f4::stm32f401::interrupt;

//use cortex_m_semihosting::dbg;

use matrixled::matrix_led;
use misakifont::font88::FONT88;

const START_TIME: u16 = 1500u16;
const CONTICUE_TIME: u16 = 200u16;

static WAKE_TIMER: WakeTimer = WAKE_TIMER_INIT;

#[entry]
fn main() -> ! {
    let device = stm32f401::Peripherals::take().unwrap();

    init_clock(&device);
    gpio_setup(&device);
    tim11_setup(&device);

    let mut matrix = matrix_led::Matrix::new(&device);

    //device.GPIOA.bsrr.write(|w| w.bs10().set());
    //device.GPIOA.bsrr.write(|w| w.bs11().set());

    let chars = [
        0xa4, 0xb3, 0xa4, 0xf3, 0xa4, 0xcb, 0xa4, 0xc1, 0xa4, 0xcf, 0xa1, 0xa2, 0xc8, 0xfe, 0xc5,
        0xd4, 0xa4, 0xb5, 0xa4, 0xf3, 0xa1, 0xa1, 0xa1, 0xa1, 0xa1, 0xa1, 0xa1, 0xa1,
    ];

    //device.GPIOA.bsrr.write(|w| w.bs11().set());

    let tim11 = &device.TIM11;
    tim11.arr.modify(|_, w| unsafe { w.arr().bits(START_TIME) });
    tim11.cr1.modify(|_, w| w.cen().enabled());
    free(|cs| WAKE_TIMER.set(cs));

    let char_count = chars.len() / 2;
```

```
    let mut start_point = 0;
    loop {
        if free(|cs| WAKE_TIMER.get(cs)) {
            // タイマー割込みの確認
            if start_point == 0 {
                tim11.arr.modify(|_, w| unsafe { w.arr().bits(START_TIME) });
            } else {
                tim11
                    .arr
                    .modify(|_, w| unsafe { w.arr().bits(CONTICUE_TIME) });
            }

            // 漢字の表示位置算出と描画
            matrix.clear();
            let char_start = start_point / 8;
            let char_end = if (start_point % 8) == 0 {
                char_start + 3
            } else {
                char_start + 4
            };
            let char_end = core::cmp::min(char_end, char_count);
            let mut disp_xpos: i32 = -((start_point % 8) as i32);
            for i in char_start..char_end + 1 {
                // 各漢字の表示
                let font = FONT88.get_char(chars[i * 2], chars[i * 2 + 1]);
                matrix.draw_bitmap(disp_xpos, 0, 8, font);
                disp_xpos += 8;
            }
            matrix.flash_led(); // LED表示の更新
            start_point += 1;

            if start_point > 8 * char_count - 32 {
                start_point = 0;
            }
            free(|cs| WAKE_TIMER.reset(cs));
        }

        device.GPIOA.bsrr.write(|w| w.br1().reset());
        cortex_m::asm::wfi();
        device.GPIOA.bsrr.write(|w| w.bs1().set());
    }
}

use core::cell::UnsafeCell;
/// TIM11割り込み関数
#[interrupt]
fn TIM1_TRG_COM_TIM11() {
    free(|cs| {
        unsafe {
            let device = stm32f401::Peripherals::steal();
            device.TIM11.sr.modify(|_, w| w.uif().clear());
        }
        WAKE_TIMER.set(cs);
    });
}

/// タイマーの起動を知らせるフラグ
/// グローバル　イミュータブル変数とする
```

```
struct WakeTimer(UnsafeCell<bool>);
const WAKE_TIMER_INIT: WakeTimer = WakeTimer(UnsafeCell::new(false));
impl WakeTimer {
    pub fn set(&self, _cs: &cortex_m::interrupt::CriticalSection) {
        unsafe { *self.0.get() = true };
    }
    pub fn reset(&self, _cs: &cortex_m::interrupt::CriticalSection) {
        unsafe { *self.0.get() = false };
    }
    pub fn get(&self, _cs: &cortex_m::interrupt::CriticalSection) -> bool {
        unsafe { *self.0.get() }
    }
}
unsafe impl Sync for WakeTimer {}

/// システムクロックの初期設定
///   クロック周波数  48MHz
fn init_clock(device: &stm32f401::Peripherals) {
    // システムクロック  48MHz
    // PLLCFGR設定
    // hsi(16M)/8*192/8=48MHz
    {
        let pllcfgr = &device.RCC.pllcfgr;
        pllcfgr.modify(|_, w| w.pllsrc().hsi());
        pllcfgr.modify(|_, w| w.pllp().div8());
        pllcfgr.modify(|_, w| unsafe { w.plln().bits(192u16) });
        pllcfgr.modify(|_, w| unsafe { w.pllm().bits(8u8) });
    }

    // PLL起動
    device.RCC.cr.modify(|_, w| w.pllon().on());
    while device.RCC.cr.read().pllrdy().is_not_ready() {
        // PLLの安定をただひたすら待つ
    }

    // フラッシュ読み出し遅延の変更
    device
        .FLASH
        .acr
        .modify(|_, w| unsafe { w.latency().bits(1u8) });
    // システムクロックをPLLに切り替え
    device.RCC.cfgr.modify(|_, w| w.sw().pll());
    while !device.RCC.cfgr.read().sws().is_pll() { /*wait*/ }

    // APB2のクロックを1/16
    //device.RCC.cfgr.modify(|_,w| w.ppre2().div2());
}

/// gpioのセットアップ
fn gpio_setup(device: &stm32f401::Peripherals) {
    // GPIOA 電源
    device.RCC.ahb1enr.modify(|_, w| w.gpioaen().enabled());

    // GPIOC セットアップ
    let gpioa = &device.GPIOA;
    gpioa.moder.modify(|_, w| w.moder1().output());
    gpioa.moder.modify(|_, w| w.moder10().output());
    gpioa.moder.modify(|_, w| w.moder11().output());
```

```
}

/// TIM11のセットアップ
fn tim11_setup(device: &stm32f401::Peripherals) {
    // TIM11 電源
    device.RCC.apb2enr.modify(|_, w| w.tim11en().enabled());

    // TIM11 セットアップ
    let tim11 = &device.TIM11;
    tim11.psc.modify(|_, w| w.psc().bits(48_000u16 - 1)); // 1ms
    tim11.dier.modify(|_, w| w.uie().enabled());
    unsafe {
        cortex_m::peripheral::NVIC::unmask(stm32f401::interrupt::TIM1_TRG_COM_TIM11);
    }
}
```

3　lib.rs ソース

```
#![no_std]

// matrix ledの制御
pub mod matrix_led;
```

4 matrix_led.rs ソース

```
1  //! matrix_ledの制御
2  //!  ledサイズ  32*8
3
4  use stm32f4::stm32f401;
5
6  /// Matrix Ledの制御
7  pub struct Matrix<'a> {
8      video_ram: [u32; 8], // 左上を基点(0,0)として、各u32のMSBと[0]が基点
9      device: &'a stm32f401::Peripherals,
10     spi: &'a stm32f401::SPI1,
11 }
12
13 impl<'a> Matrix<'a> {
14     pub fn new(device: &stm32f401::Peripherals) -> Matrix {
15         let led = Matrix {
16             video_ram: [0; 8],
17             device,
18             spi: &device.SPI1,
19         };
20         led.gpio_setup();
21         led.spi1_setup();
22         led.dma_setup();
23         led.init_mat_led();
24         led
25     }
26
27     /// Video RAMをクリアする
28     pub fn clear(&mut self) {
29         for line in &mut self.video_ram {
30             *line = 0;
31         }
32     }
33
34     /// 指定の場所に、指定の矩形のビットマップを表示する。
35     ///
36     /// 原点は、左上隅(0,0)。
37     /// ビットマップの最大サイズは8*8。
38     ///
39     /// 幅が8未満の場合は、LSBより詰めること。
40     /// 矩形の高さは、bitmapの要素数に等しい。
41     pub fn draw_bitmap(&mut self, px: i32, py: u32, width: u32, bitmap: &[u8]) {
42         let width = if width <= 8 { width as i32 } else { 8 };
43         let shift: i32 = 31 - px - width + 1;
44         let mask: u32 = (1 << width) - 1;
45         let mut y = if py >= 8 { return } else { py as usize };
46         for line in bitmap {
47             self.video_ram[y] |= if shift >= 0 {
48                 ((*line as u32) & mask) << shift
49             } else {
50                 ((*line as u32) & mask) >> -shift
51             };
52             y += 1;
53             if y >= 8 {
54                 break;
```

```
55              }
56          }
57      }
58
59      /// Matrix LEDにvideo_ramの内容を表示する。
60      pub fn flash_led(&self) {
61          while let Err(_) = DMA_BUFF.clear_buff(self.device) {}
62          for x in 0..8 {
63              self.send_oneline_mat_led(x);
64          }
65          self.send_request_to_dma();
66      }
67
68      /// Matrix LED BUFFに一行を送る
69      /// # 引数
70      ///     line_num:   一番上が0。一番下が7
71      fn send_oneline_mat_led(&self, line_num: u32) {
72          let digi_code: u16 = ((line_num + 1) << 8) as u16;
73          let pat = self.video_ram[line_num as usize];
74          let dat: [u16; 4] = [
75              digi_code | (((pat >> 24) & 0x00FF) as u16),
76              digi_code | (((pat >> 16) & 0x00FF) as u16),
77              digi_code | (((pat >> 08) & 0x00FF) as u16),
78              digi_code | (((pat) & 0x00FF) as u16),
79          ];
80          DMA_BUFF.add_buff(&dat, self.device).unwrap();
81      }
82
83      /// Matrix LED 初期化
84      fn init_mat_led(&self) {
85          const INIT_PAT: [u16; 5] = [
86              0x0F00, // テストモード解除
87              0x0900, // BCDデコードバイパス
88              0x0A02, // 輝度制御  下位4bit MAX:F
89              0x0B07, // スキャン桁指定 下位4bit MAX:7
90              0x0C01, // シャットダウンモード  解除
91          ];
92
93          while let Err(_) = DMA_BUFF.clear_buff(self.device) {}
94          for pat in &INIT_PAT {
95              DMA_BUFF.add_buff(&[*pat; 4], self.device).unwrap();
96          }
97          self.send_request_to_dma();
98      }
99
100     /// SPI1 データのDMA送信要求
101     ///   MatrixLED 4ブロック*行数 分のデータの送信を行う。
102     ///   送信データは、事前にDMA_BUFFに投入済みのこと。
103     fn send_request_to_dma(&self) {
104         let dma = &self.device.DMA2;
105         for data in DMA_BUFF.iter() {
106             while dma.st[3].cr.read().en().is_enabled() {}
107             let adr = data.as_ptr() as u32;
108             dma.st[3].m0ar.write(|w| w.m0a().bits(adr));
109             dma.st[3].ndtr.write(|w| w.ndt().bits(4u16));
110
111             self.spi_enable();
112             self.dma_start();
```

```
113             while dma.lisr.read().tcif3().is_not_complete() {}
114             dma.st[3].cr.modify(|_, w| w.en().disabled());
115             while self.spi.sr.read().bsy().is_busy() {}
116             self.spi_disable();
117         }
118     }
119
120     /// DMAの完了フラグをクリアし、DMAを開始する
121     fn dma_start(&self) {
122         let dma = &self.device.DMA2;
123         dma.lifcr.write(|w| {
124             w.ctcif3()
125                 .clear()
126                 .chtif3()
127                 .clear()
128                 .cteif3()
129                 .clear()
130                 .cdmeif3()
131                 .clear()
132                 .cfeif3()
133                 .clear()
134         });
135         dma.st[3].cr.modify(|_, w| w.en().enabled());
136     }
137
138     /// spi通信有効にセット
139     fn spi_enable(&self) {
140         self.cs_enable();
141         self.spi.cr1.modify(|_, w| w.spe().enabled());
142     }
143
144     /// spi通信無効にセット
145     ///   LEDのデータ確定シーケンス含む
146     fn spi_disable(&self) {
147         while self.spi.sr.read().txe().is_not_empty() {
148             cortex_m::asm::nop();
149         }
150         while self.spi.sr.read().bsy().is_busy() {
151             cortex_m::asm::nop(); // wait
152         }
153         self.cs_disable();
154         self.spi.cr1.modify(|_, w| w.spe().disabled());
155     }
156
157     /// CS(DATA) ピンを 通信無効(HI)にする
158     /// CSピンは、PA4に固定(ハードコート)
159     fn cs_disable(&self) {
160         self.device.GPIOA.bsrr.write(|w| w.bs4().set());
161         for _x in 0..10 {
162             // 通信終了時は、データの確定待ちが必要
163             // 最低50ns 48MHzクロックで最低3クロック
164             cortex_m::asm::nop();
165         }
166     }
167
168     /// CS(DATA) ピンを通信有効(LO)にする
169     /// CSピンは、PA4に固定(ハードコート)
170     fn cs_enable(&self) {
```

```
171             self.device.GPIOA.bsrr.write(|w| w.br4().reset());
172         }
173
174         /// SPIのセットアップ
175         fn spi1_setup(&self) {
176             // 電源投入
177             self.device.RCC.apb2enr.modify(|_, w| w.spi1en().enabled());
178
179             self.spi.cr1.modify(|_, w| {
180                 w.bidimode().unidirectional().
181                 dff().sixteen_bit().
182                 lsbfirst().msbfirst().
183                 br().div4(). // 基準クロックは48MHz
184                 mstr().master().
185                 cpol().idle_low().
186                 cpha().first_edge().
187                 ssm().enabled().
188                 ssi().slave_not_selected()
189             });
190             self.spi.cr2.modify(|_, w| w.txdmaen().enabled());
191         }
192
193         /// gpioのセットアップ
194         fn gpio_setup(&self) {
195             self.device.RCC.ahb1enr.modify(|_, w| w.gpioaen().enabled());
196             // SPI端子割付け
197             let gpioa = &self.device.GPIOA;
198             gpioa.moder.modify(|_, w| w.moder7().alternate()); // SPI1_MOSI
199             gpioa.afrl.modify(|_, w| w.afrl7().af5());
200             gpioa.ospeedr.modify(|_, w| w.ospeedr7().very_high_speed());
201             gpioa.otyper.modify(|_, w| w.ot7().push_pull());
202             gpioa.moder.modify(|_, w| w.moder5().alternate()); // SPI1_CLK
203             gpioa.afrl.modify(|_, w| w.afrl5().af5());
204             gpioa.ospeedr.modify(|_, w| w.ospeedr5().very_high_speed());
205             gpioa.otyper.modify(|_, w| w.ot5().push_pull());
206             gpioa.moder.modify(|_, w| w.moder4().output()); // NSS(CS)
207             gpioa.ospeedr.modify(|_, w| w.ospeedr4().very_high_speed());
208             gpioa.otyper.modify(|_, w| w.ot4().push_pull());
209         }
210
211         /// DMAのセットアップ
212         fn dma_setup(&self) {
213             self.device.RCC.ahb1enr.modify(|_, w| w.dma2en().enabled());
214             // DMAストリーム3のチャンネル3使用
215             let st3_3 = &self.device.DMA2.st[3];
216             unsafe {
217                 st3_3.cr.modify(|_, w| w.chsel().bits(3u8));
218             }
219             st3_3.cr.modify(|_, w| {
220                 w.mburst()
221                     .incr4()
222                     .pburst()
223                     .single()
224                     .ct()
225                     .memory0()
226                     .dbm()
227                     .disabled()
228                     .pl()
```

```
                .medium()
                .pincos()
                .psize()
                .msize()
                .bits16()
                .psize()
                .bits16()
                .minc()
                .incremented()
                .pinc()
                .fixed()
                .circ()
                .disabled()
                .dir()
                .memory_to_peripheral()
                .tcie()
                .enabled()
                .htie()
                .disabled()
                .teie()
                .enabled()
                .dmeie()
                .enabled()
        });
        st3_3
            .fcr
            .modify(|_, w| w.feie().enabled().dmdis().disabled().fth().half());
        let spi1_dr = &self.device.SPI1.dr as *const _ as u32;
        st3_3.par.write(|w| w.pa().bits(spi1_dr));
    }
}

/// DMAバッファ領域
///   グローバル変数・matrix_ledモジュール以外での操作禁止
///   DMA2_S3CR.ENビットが0の時のみ操作可能
static DMA_BUFF: DmaBuff = DMA_BUFF_INIT;

type Result<T> = core::result::Result<T, &'static str>;

use core::cell::UnsafeCell;
struct DmaBuff {
    buff: UnsafeCell<[[u16; 4]; 8]>,
    data_count: UnsafeCell<usize>,
}

const DMA_BUFF_INIT: DmaBuff = DmaBuff {
    buff: UnsafeCell::new([[0u16; 4]; 8]),
    data_count: UnsafeCell::new(0),
};

unsafe impl Sync for DmaBuff {}

impl DmaBuff {
    pub fn clear_buff(&self, device: &stm32f401::Peripherals) -> Result<()> {
        Self::is_dma_inactive(device)?;
        unsafe {
            *self.data_count.get() = 0;
        }
```

```
287         Ok(())
288     }
289 
290     pub fn add_buff(&self, data: &[u16], device: &stm32f401::Peripherals) -> Result<()> {
291         Self::is_dma_inactive(device)?;
292         unsafe {
293             if *self.data_count.get() < 8 {
294                 *self.data_count.get() += 1;
295             } else {
296                 return Err("Buffer␣over␣flow");
297             }
298             &(*self.buff.get())[*self.data_count.get() - 1].clone_from_slice(&data[0..4]);
299         }
300         Ok(())
301     }
302 
303     pub fn iter(&self) -> DmaBuffIter {
304         DmaBuffIter { cur_index: None }
305     }
306 
307     fn is_dma_inactive(device: &stm32f401::Peripherals) -> Result<()> {
308         if device.DMA2.st[3].cr.read().en().is_enabled() {
309             Err("DMA2␣stream␣active")
310         } else {
311             Ok(())
312         }
313     }
314 
315     fn get_buff(&self, index: usize) -> Option<&[u16; 4]> {
316         unsafe {
317             if index < *self.data_count.get() {
318                 Some(&(*self.buff.get())[index])
319             } else {
320                 None
321             }
322         }
323     }
324 }
325 
326 /// DmaBuff用Iterator
327 struct DmaBuffIter {
328     cur_index: Option<usize>,
329 }
330 
331 impl Iterator for DmaBuffIter {
332     type Item = &'static [u16; 4];
333 
334     fn next(&mut self) -> Option<Self::Item> {
335         match &mut self.cur_index {
336             Some(i) => {
337                 *i += 1;
338             }
339             None => {
340                 self.cur_index = Some(0);
341             }
342         };
343         DMA_BUFF.get_buff(self.cur_index.unwrap())
344     }
```

```
345 }
```