matrix_led ソースファイル

美都

2020 年 5 月 24 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 = "0.11.0"
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.bs0().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
```

```
109 struct WakeTimer(UnsafeCell<bool>);
110 const WAKE_TIMER_INIT: WakeTimer = WakeTimer(UnsafeCell::new(false));
111 impl WakeTimer {
112     pub fn set(&self, _cs: &cortex_m::interrupt::CriticalSection) {
113         unsafe { *self.0.get() = true };
114     }
115     pub fn reset(&self, _cs: &cortex_m::interrupt::CriticalSection) {
116         unsafe { *self.0.get() = false };
117     }
118     pub fn get(&self, _cs: &cortex_m::interrupt::CriticalSection) -> bool {
119         unsafe { *self.0.get() }
120     }
121 }
122 unsafe impl Sync for WakeTimer {}
123
124 /// システムクロックの初期設定
125 ///   クロック周波数  48MHz
126 fn init_clock(device: &stm32f401::Peripherals) {
127     // システムクロック  48MHz
128     // PLLCFGR設定
129     // hsi(16M)/8*192/8=48MHz
130     {
131         let pllcfgr = &device.RCC.pllcfgr;
132         pllcfgr.modify(|_, w| w.pllsrc().hsi());
133         pllcfgr.modify(|_, w| w.pllp().div8());
134         pllcfgr.modify(|_, w| unsafe { w.plln().bits(192u16) });
135         pllcfgr.modify(|_, w| unsafe { w.pllm().bits(8u8) });
136     }
137
138     // PLL起動
139     device.RCC.cr.modify(|_, w| w.pllon().on());
140     while device.RCC.cr.read().pllrdy().is_not_ready() {
141         // PLLの安定をただひたすら待つ
142     }
143
144     // フラッシュ読み出し遅延の変更
145     device
146         .FLASH
147         .acr
148         .modify(|_, w| unsafe { w.latency().bits(1u8) });
149     // システムクロックをPLLに切り替え
150     device.RCC.cfgr.modify(|_, w| w.sw().pll());
151     while !device.RCC.cfgr.read().sws().is_pll() { /*wait*/ }
152
153     // APB2のクロックを1/16
154     //device.RCC.cfgr.modify(|_,w| w.ppre2().div2());
155 }
156
157 /// gpioのセットアップ
158 fn gpio_setup(device: &stm32f401::Peripherals) {
159     // GPIOA 電源
160     device.RCC.ahb1enr.modify(|_, w| w.gpioaen().enabled());
161
162     // GPIOC セットアップ
163     let gpioa = &device.GPIOA;
164     gpioa.moder.modify(|_, w| w.moder1().output());
165     gpioa.moder.modify(|_, w| w.moder0().output());
166     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

```
//! matrix_ledの制御
//!  ledサイズ  32*8

use stm32f4::stm32f401;
use stm32f4::stm32f401::interrupt;

/// Matrix Ledの制御
pub struct Matrix<'a> {
    video_ram: [u32; 8], // 左上を基点(0,0)として、各u32のMSBと[0]が基点
    device: &'a stm32f401::Peripherals,
    spi: &'a stm32f401::SPI1,
}

impl<'a> Matrix<'a> {
    pub fn new(device: &stm32f401::Peripherals) -> Matrix {
        let led = Matrix {
            video_ram: [0; 8],
            device,
            spi: &device.SPI1,
        };
        led.gpio_setup();
        led.spi1_setup();
        led.dma_setup();
        led.init_mat_led();
        led
    }

    /// Video RAMをクリアする
    pub fn clear(&mut self) {
        for line in &mut self.video_ram {
            *line = 0;
        }
    }

    /// 指定の場所に、指定の矩形のビットマップを表示する。
    ///
    /// 原点は、左上隅(0,0)。
    /// ビットマップの最大サイズは8*8。
    ///
    /// 幅が8未満の場合は、LSBより詰めること。
    /// 矩形の高さは、bitmapの要素数に等しい。
    pub fn draw_bitmap(&mut self, px: i32, py: u32, width: u32, bitmap: &[u8]) {
        let width = if width <= 8 { width as i32 } else { 8 };
        let shift: i32 = 31 - px - width + 1;
        let mask: u32 = (1 << width) - 1;
        let mut y = if py >= 8 { return } else { py as usize };
        for line in bitmap {
            self.video_ram[y] |= if shift >= 0 {
                ((*line as u32) & mask) << shift
            } else {
                ((*line as u32) & mask) >> -shift
            };
            y += 1;
            if y >= 8 {
```

```
                break;
            }
        }
    }

    /// Matrix LEDにvideo_ramの内容を表示する。
    pub fn flash_led(&self) {
        while let Err(_) = DMA_BUFF.clear_buff(self.device) {}
        for x in 0..8 {
            self.send_oneline_mat_led(x);
        }
        self.send_request_to_dma();
    }

    /// Matrix LED BUFFに一行を送る
    /// # 引数
    ///     line_num:   一番上が0。一番下が7
    fn send_oneline_mat_led(&self, line_num: u32) {
        let digi_code: u16 = ((line_num + 1) << 8) as u16;
        let pat = self.video_ram[line_num as usize];
        let dat: [u16; 4] = [
            digi_code | (((pat >> 24) & 0x00FF) as u16),
            digi_code | (((pat >> 16) & 0x00FF) as u16),
            digi_code | (((pat >> 08) & 0x00FF) as u16),
            digi_code | (((pat) & 0x00FF) as u16),
        ];
        DMA_BUFF.add_buff(&dat, self.device).unwrap();
    }

    /// Matrix LED 初期化
    fn init_mat_led(&self) {
        const INIT_PAT: [u16; 5] = [
            0x0F00, // テストモード解除
            0x0900, // BCDデコードバイパス
            0x0A02, // 輝度制御  下位4bit MAX:F
            0x0B07, // スキャン桁指定 下位4bit MAX:7
            0x0C01, // シャットダウンモード  解除
        ];

        while let Err(_) = DMA_BUFF.clear_buff(self.device) {}
        for pat in &INIT_PAT {
            DMA_BUFF.add_buff(&[*pat; 4], self.device).unwrap();
        }
        self.send_request_to_dma();
    }

    /// SPI1 データのDMA送信要求
    ///   MatrixLED 4ブロック*行数 分のデータの送信を行う。
    ///   送信データは、事前にDMA_BUFFに投入済みのこと。
    fn send_request_to_dma(&self) {
        let dma = &self.device.DMA2;
        let mut i = DMA_BUFF.iter();
        if let Some(data) = i.next() {
            while dma.st[3].cr.read().en().is_enabled() {}
            let adr = data.as_ptr() as u32;
            dma.st[3].m0ar.write(|w| w.m0a().bits(adr));
            dma.st[3].ndtr.write(|w| w.ndt().bits(4u16));

```

```
113             Self::spi_enable(&self.device);
114             Self::dma_start(&self.device);
115         }
116         // 以降、2レコード目からの転送は、割込みルーチンにて
117     }
118
119     /// SPI送信終了待ちと送信終了時間の計測
120     ///   ループ回数が一定回数以上になると、緑のLEDを点灯する
121     fn wait_api_and_measurement(device: &stm32f401::Peripherals) {
122         let dma = &device.DMA2;
123         let spi = &device.SPI1;
124         let gpioa = &device.GPIOA;
125         const WAIT_LIMIT: u32 = 31;
126         let mut count_wait = 0;
127
128         while dma.lisr.read().tcif3().is_not_complete() {
129             count_wait += 1;
130         }
131         while spi.sr.read().txe().is_not_empty() {
132             count_wait += 0;
133         }
134         while spi.sr.read().bsy().is_busy() {
135             count_wait += 0;
136         }
137         if count_wait > WAIT_LIMIT {
138             gpioa.bsrr.write(|w| w.bs0().set());
139         }
140     }
141
142     /// DMAの完了フラグをクリアし、DMAを開始する
143     fn dma_start(device: &stm32f401::Peripherals) {
144         let dma = &device.DMA2;
145         dma.lifcr.write(|w| {
146             w.ctcif3().clear();
147             w.chtif3().clear();
148             w.cteif3().clear();
149             w.cdmeif3().clear()
150         });
151
152         dma.st[3].cr.modify(|_, w| w.en().enabled());
153     }
154
155     /// spi通信有効にセット
156     fn spi_enable(device: &stm32f401::Peripherals) {
157         let spi = &device.SPI1;
158         Self::cs_enable(&device);
159         spi.cr1.modify(|_, w| w.spe().enabled());
160     }
161
162     /// spi通信無効にセット
163     ///   LEDのデータ確定シーケンス含む
164     fn spi_disable(device: &stm32f401::Peripherals) {
165         let spi = &device.SPI1;
166         while spi.sr.read().txe().is_not_empty() {
167             cortex_m::asm::nop();
168         }
169         while spi.sr.read().bsy().is_busy() {
170             cortex_m::asm::nop(); // wait
```

```
171         }
172         Self::cs_disable(&device);
173         spi.cr1.modify(|_, w| w.spe().disabled());
174     }
175 
176     /// CS(DATA) ピンを 通信無効(HI)にする
177     /// CSピンは、PA4に固定(ハードコート)
178     fn cs_disable(device: &stm32f401::Peripherals) {
179         device.GPIOA.bsrr.write(|w| w.bs4().set());
180         for _x in 0..5 {
181             // 通信終了時は、データの確定待ちが必要
182             // 最低50ns 48MHzクロックで最低3クロック
183             cortex_m::asm::nop();
184         }
185     }
186 
187     /// CS(DATA) ピンを通信有効(LO)にする
188     /// CSピンは、PA4に固定(ハードコート)
189     fn cs_enable(device: &stm32f401::Peripherals) {
190         device.GPIOA.bsrr.write(|w| w.br4().reset());
191     }
192 
193     /// SPIのセットアップ
194     fn spi1_setup(&self) {
195         // 電源投入
196         self.device.RCC.apb2enr.modify(|_, w| w.spi1en().enabled());
197 
198         self.spi.cr1.modify(|_, w| {
199             w.bidimode().unidirectional().
200             dff().sixteen_bit().
201             lsbfirst().msbfirst().
202             br().div4(). // 基準クロックは48MHz
203             mstr().master().
204             cpol().idle_low().
205             cpha().first_edge().
206             ssm().enabled().
207             ssi().slave_not_selected()
208         });
209         self.spi.cr2.modify(|_, w| w.txdmaen().enabled());
210     }
211 
212     /// gpioのセットアップ
213     fn gpio_setup(&self) {
214         self.device.RCC.ahb1enr.modify(|_, w| w.gpioaen().enabled());
215         // SPI端子割付け
216         let gpioa = &self.device.GPIOA;
217         gpioa.moder.modify(|_, w| w.moder7().alternate()); // SPI1_MOSI
218         gpioa.afrl.modify(|_, w| w.afrl7().af5());
219         gpioa.ospeedr.modify(|_, w| w.ospeedr7().very_high_speed());
220         gpioa.otyper.modify(|_, w| w.ot7().push_pull());
221         gpioa.moder.modify(|_, w| w.moder5().alternate()); // SPI1_CLK
222         gpioa.afrl.modify(|_, w| w.afrl5().af5());
223         gpioa.ospeedr.modify(|_, w| w.ospeedr5().very_high_speed());
224         gpioa.otyper.modify(|_, w| w.ot5().push_pull());
225         gpioa.moder.modify(|_, w| w.moder4().output()); // NSS(CS)
226         gpioa.ospeedr.modify(|_, w| w.ospeedr4().very_high_speed());
227         gpioa.otyper.modify(|_, w| w.ot4().push_pull());
228     }
```

```
229
230     /// DMAのセットアップ
231     fn dma_setup(&self) {
232         self.device.RCC.ahb1enr.modify(|_, w| w.dma2en().enabled());
233         // DMAストリーム3のチャンネル3使用
234         let st3_3 = &self.device.DMA2.st[3];
235         st3_3.cr.modify(|_, w| {
236             w.chsel().bits(3u8);
237             w.mburst().incr4();
238             w.pburst().single();
239             w.ct().memory0();
240             w.dbm().disabled();
241             w.pl().medium();
242             w.pincos().psize();
243             w.msize().bits16();
244             w.psize().bits16();
245             w.minc().incremented();
246             w.pinc().fixed();
247             w.circ().disabled();
248             w.dir().memory_to_peripheral();
249             w.tcie().enabled();
250             w.htie().disabled();
251             w.teie().disabled();
252             w.dmeie().disabled()
253         });
254         st3_3.fcr.modify(|_, w| {
255             w.feie().disabled();
256             w.dmdis().disabled();
257             w.fth().half()
258         });
259         let spi1_dr = &self.device.SPI1.dr as *const _ as u32;
260         st3_3.par.write(|w| w.pa().bits(spi1_dr));
261         unsafe {
262             cortex_m::peripheral::NVIC::unmask(stm32f401::interrupt::DMA2_STREAM3);
263         }
264     }
265 }
266
267 /// DMA2 Stream3 割込み関数
268 #[interrupt]
269 fn DMA2_STREAM3() {
270     static mut ITER: DmaBuffIter = DmaBuffIter { cur_index: None };
271
272     let device;
273     unsafe {
274         device = stm32f401::Peripherals::steal();
275     }
276     let dma = &device.DMA2;
277     if dma.lisr.read().tcif3().is_complete() {
278         dma.lifcr.write(|w| w.ctcif3().clear());
279         if let None = ITER.cur_index {
280             ITER.cur_index = Some(0);
281         }
282         match ITER.next() {
283             Some(data) => {
284                 //次のデータの準備
285                 let adr = data.as_ptr() as u32;
286                 dma.st[3].m0ar.write(|w| w.m0a().bits(adr));
```

```
                dma.st[3].ndtr.write(|w| w.ndt().bits(4u16));

                //前データの確定終了処理
                Matrix::spi_disable(&device);

                //次のデータの送信開始
                Matrix::spi_enable(&device);
                Matrix::dma_start(&device);
            }
            None => {
                //前データの確定終了処理
                Matrix::spi_disable(&device);
                *ITER = DmaBuffIter { cur_index: None };
            }
        }
    } else {
        dma.lifcr.write(|w| {
            w.ctcif3().clear();
            w.chtif3().clear();
            w.cteif3().clear();
            w.cdmeif3().clear()
        });
    }
}

/// 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;
        }
        Ok(())
    }

    pub fn add_buff(&self, data: &[u16], device: &stm32f401::Peripherals) -> Result<()> {
        Self::is_dma_inactive(device)?;
        unsafe {
            if *self.data_count.get() < 8 {
```

```
345                 *self.data_count.get() += 1;
346             } else {
347                 return Err("Buffer over flow");
348             }
349             &(*self.buff.get())[*self.data_count.get() - 1].clone_from_slice(&data[0..4]);
350         }
351         Ok(())
352     }
353 
354     pub fn iter(&self) -> DmaBuffIter {
355         DmaBuffIter { cur_index: None }
356     }
357 
358     fn is_dma_inactive(device: &stm32f401::Peripherals) -> Result<()> {
359         if device.DMA2.st[3].cr.read().en().is_enabled() {
360             Err("DMA2 stream active")
361         } else {
362             Ok(())
363         }
364     }
365 
366     fn get_buff(&self, index: usize) -> Option<&[u16; 4]> {
367         unsafe {
368             if index < *self.data_count.get() {
369                 Some(&(*self.buff.get())[index])
370             } else {
371                 None
372             }
373         }
374     }
375 }
376 
377 /// DmaBuff用Iterator
378 struct DmaBuffIter {
379     cur_index: Option<usize>,
380 }
381 
382 impl Iterator for DmaBuffIter {
383     type Item = &'static [u16; 4];
384 
385     fn next(&mut self) -> Option<Self::Item> {
386         match &mut self.cur_index {
387             Some(i) => {
388                 *i += 1;
389             }
390             None => {
391                 self.cur_index = Some(0);
392             }
393         };
394         DMA_BUFF.get_buff(self.cur_index.unwrap())
395     }
396 }
```