Porting to new MCU platform

Jump to bottom

olikraus edited this page on Mar 4 · 78 revisions

Introduction

In order to port the U8G2 library to another MCU platform you need to provide the functions to interface directly with the MCU hardware so that the U8G2 Hardware Abstraction Layer (HAL) can issue commands etc. to the display controller. There are two interface points for the HAL.

1. The "uC specific" GPIO and Delay callback (the last argument of the setup function)
2. The u8x8 byte communication callback (the second to last argument of the setup function)

These functions are both used as callbacks by the U8G2/U8X8 library. They are setup when you call the first initialization function e.g. `u8x8_Setup(&u8x8, u8x8_d_ssd1306_128x64_noname, u8x8_cad_ssd13xx_i2c, u8x8_byte_hw_i2c, psoc_gpio_and_delay_cb);`

Writing a u8x8 byte communication callback is only required, if you want to use exisiting uC communication interfaces (I2C, SPI, etc). Several "bitbanging" communication callback

procedures are already available (see below).

Important: U8g2 defaults to 8 bit mode, which means that the display size is limited to 240x240 pixel. Activate 16 bit mode for larger displays (see https://github.com/olikraus/u8g2/blob/master/doc/faq.txt).

The "uC specific" GPIO and Delay callback

The "uC specific" GPIO and Delay callback function (in fact all of the HAL function) function must conform with the function prototype:

```
typedef uint8_t (*u8x8_msg_cb)(u8x8_t *u8x8, uint8_t msg, uint8_t arg_int, void *arg_ptr);
```

This function is used to set and reset GPIOs (in the case if software implemented interfaces) e.g. a software I2C, SPI, 8080 or 6800 interface. In addition this function is used to implement busy-wait delays for pin timing.
This function takes a "msg" which is one of many #defines found in u8x8.h that are of the form "U8X8_MSG_GPIO" or "U8X8_DELAY" and then acts on that message using a c-style argc/argv interface. In this specific case those are called "arg_int" and "arg_ptr".

There are three classes of messages sent by the HAL.

1. **Delay messages** of the form "U8X8_MSG_DELAY_". These messages are used to provide delay for the software implementation of I2C, SPI etc.
   In order for the software (aka. bit-banged) interfaces to work you need to implement the MCU specific busy-wait loop to provide a correct amount of delay.
   For the example implementation I used the Cypress PSoC specific delay functions of the form CyDelay*

2. **GPIO messages** of the form "U8X*_MSG_GPIO". These messages are used to write 1s and 0s to the GPIOs which are being used to interface to the device. i.e. the SCL/SDA or Reset or CS etc.
   For the example implementation I used the Cypress pin write functions which all take the form of "pinname_Write()".

3. **GPIO menu pins** are used to get the state of an input pin. These messages are only required for the build in menu function and can be ignored, if the U8G2/U8X8 menu functions are not used.

Template for the GPIO and Delay callback

```
uint8_t u8x8_gpio_and_delay_template(u8x8_t *u8x8, uint8_t msg, uint8_t arg_i
void *arg_ptr)
{
  switch(msg)
  {
    case U8X8_MSG_GPIO_AND_DELAY_INIT:  // called once during init phase of
u8g2/u8x8
      break;                                                // can be used
to setup pins
    case U8X8_MSG_DELAY_NANO:                   // delay arg_int * 1 nano second
      break;
    case U8X8_MSG_DELAY_100NANO:                // delay arg_int * 100 nano
seconds
      break;
    case U8X8_MSG_DELAY_10MICRO:                // delay arg_int * 10 micro
seconds
      break;
    case U8X8_MSG_DELAY_MILLI:                  // delay arg_int * 1 milli
second
      break;
    case U8X8_MSG_DELAY_I2C:                            // arg_int is the I2C
speed in 100KHz, e.g. 4 = 400 KHz
      break;                                                // arg_int=1:
delay by 5us, arg_int = 4: delay by 1.25us
    case U8X8_MSG_GPIO_D0:                              // D0 or SPI clock pin:
Output level in arg_int
    //case U8X8_MSG_GPIO_SPI_CLOCK:
      break;
    case U8X8_MSG_GPIO_D1:                              // D1 or SPI data pin:
Output level in arg_int
    //case U8X8_MSG_GPIO_SPI_DATA:
      break;
    case U8X8_MSG_GPIO_D2:                              // D2 pin: Output level
in arg_int
      break;
    case U8X8_MSG_GPIO_D3:                              // D3 pin: Output level
in arg_int
      break;
    case U8X8_MSG_GPIO_D4:                              // D4 pin: Output level
in arg_int
      break;
    case U8X8_MSG_GPIO_D5:                              // D5 pin: Output level
in arg_int
      break;
    case U8X8_MSG_GPIO_D6:                              // D6 pin: Output level
in arg_int
      break;
    case U8X8_MSG_GPIO_D7:                              // D7 pin: Output level
```

```
    case U8X8_MSG_GPIO_D7:                                  // D7 pin: Output level
in arg_int
      break;
    case U8X8_MSG_GPIO_E:                                   // E/WR pin: Output
level in arg_int
      break;
    case U8X8_MSG_GPIO_CS:                                  // CS (chip select) pin:
Output level in arg_int
      break;
    case U8X8_MSG_GPIO_DC:                                  // DC (data/cmd, A0,
register select) pin: Output level in arg_int
      break;
    case U8X8_MSG_GPIO_RESET:                   // Reset pin: Output level in
arg_int
      break;
    case U8X8_MSG_GPIO_CS1:                                 // CS1 (chip select)
pin: Output level in arg_int
      break;
    case U8X8_MSG_GPIO_CS2:                                 // CS2 (chip select)
pin: Output level in arg_int
      break;
    case U8X8_MSG_GPIO_I2C_CLOCK:               // arg_int=0: Output low at I2C
clock pin
      break;                                                    // arg_int=1:
Input dir with pullup high for I2C clock pin
    case U8X8_MSG_GPIO_I2C_DATA:                            // arg_int=0: Output low
at I2C data pin
      break;                                                    // arg_int=1:
Input dir with pullup high for I2C data pin
    case U8X8_MSG_GPIO_MENU_SELECT:
      u8x8_SetGPIOResult(u8x8, /* get menu select pin state */ 0);
      break;
    case U8X8_MSG_GPIO_MENU_NEXT:
      u8x8_SetGPIOResult(u8x8, /* get menu next pin state */ 0);
      break;
    case U8X8_MSG_GPIO_MENU_PREV:
      u8x8_SetGPIOResult(u8x8, /* get menu prev pin state */ 0);
      break;
    case U8X8_MSG_GPIO_MENU_HOME:
      u8x8_SetGPIOResult(u8x8, /* get menu home pin state */ 0);
      break;
    default:
      u8x8_SetGPIOResult(u8x8, 1);                          // default return value
      break;
  }
  return 1;
}
```

Notes

The return value of the GPIO and Delay callback should be 1 (true) for successful handling of the message.

U8X8_MSG_GPIO_SPI_CLOCK is an alias for U8X8_MSG_GPIO_D0 and U8X8_MSG_GPIO_SPI_DATA is an alias for U8X8_MSG_GPIO_D1.

Not all messages are required. If a pin is not connected to your hardware, then the message can be ignored (the case can be removed from the code). This is also true, if there is a special byte communication callback in which the GPIO is controlled by a uC communicaton subsystem (I2C, SPI).

Most messages just require an action without return value. For the menu messages, the level of the input pin has to be provided via the `u8x8_SetGPIOResult` function. The second argument should be the level at the input pin. The I2C message also do not poll any state from their pins: The build-in I2C procedures ignore the ACK signal from the I2C device.

Communication Callback (e.g. u8x8_byte_hw_i2c)

In order to interface to the communication port of the display controller you need to have a byte orientated interface i.e. SPI, I2C, etc. This interface may either be implemented as a bit-banged software interface or using the MCU specific hardware. Several software bit-banged interface are provided as part of the U8X8 library in u8x8_byte.c:

Byte Procedure	Description
u8x8_byte_4wire_sw_spi	Standard 8-bit SPI communication with "four pins" (SCK, MOSI, DC, CS)
u8x8_byte_3wire_sw_spi	9-bit communication with "three pins" (SCK, MOSI, CS)
u8x8_byte_8bit_6800mode	Parallel interface, 6800 format
u8x8_byte_8bit_8080mode	Parallel interface, 8080 format
u8x8_byte_sw_i2c	Two wire, I2C communication
u8x8_byte_ks0108	Special interface for KS0108 controller

The above functions use the uC specific gpio and delay functions defined by you.

If you want to use the MCU specific hardware for these communication interfaces you can create your own function. This function must conform to the prototype:

```
typedef uint8_t (*u8x8_msg_cb)(u8x8_t *u8x8, uint8_t msg, uint8_t arg_int, void *arg_ptr);
```

The HW interface function needs to handle message from the rest of the system. The messages that you need to implement are:

Message	Description
U8X8_MSG_BYTE_INIT	Send once during the init phase of the display.
U8X8_MSG_BYTE_SET_DC	Set the level of the data/command pin. `arg_int` contains the expected output level. Use `u8x8_gpio_SetDC(u8x8, arg_int)` to send a message to the GPIO procedure.
U8X8_MSG_BYTE_START_TRANSFER	Set the chip select line here. `u8x8->display_info->chip_enable_level` contains the expected level. Use `u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_enable_level)` to call the GPIO procedure.
U8X8_MSG_BYTE_SEND	Send one or more bytes, located at `arg_ptr`, `arg_int` contains the number of bytes.
U8X8_MSG_BYTE_END_TRANSFER	Unselect the device. Use the CS level from here: `u8x8->display_info->chip_disable_level`.

Hardware SPI Communication

The following code lists a typical SPI implementation. The messages are translated to calls to the Arduino SPI library.

```
extern "C" uint8_t u8x8_byte_arduino_hw_spi(u8x8_t *u8x8, uint8_t msg, uint8_t
  uint8_t *data;
  uint8_t internal_spi_mode;
  switch(msg) {
    case U8X8_MSG_BYTE_SEND:
      data = (uint8_t *)arg_ptr;
```

```
      while( arg_int > 0 ) {
        SPI.transfer((uint8_t)*data);
        data++;
        arg_int--;
      }
      break;
    case U8X8_MSG_BYTE_INIT:
      u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_disable_level);
      SPI.begin();
      break;
    case U8X8_MSG_BYTE_SET_DC:
      /* Wait for SPI transfer completion might be required here */
      u8x8_gpio_SetDC(u8x8, arg_int);
      break;
    case U8X8_MSG_BYTE_START_TRANSFER:
      /* SPI mode has to be mapped to the mode of the current controller, at least
      internal_spi_mode =  0;
      switch(u8x8->display_info->spi_mode) {
        case 0: internal_spi_mode = SPI_MODE0; break;
        case 1: internal_spi_mode = SPI_MODE1; break;
        case 2: internal_spi_mode = SPI_MODE2; break;
        case 3: internal_spi_mode = SPI_MODE3; break;
      }
      SPI.beginTransaction(SPISettings(u8x8->display_info->sck_clock_hz, MSBFIRST,
      u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_enable_level);
      u8x8->gpio_and_delay_cb(u8x8, U8X8_MSG_DELAY_NANO, u8x8->display_info->post_
      break;
    case U8X8_MSG_BYTE_END_TRANSFER:
      /* Wait for SPI transfer completion might be required here */
      u8x8->gpio_and_delay_cb(u8x8, U8X8_MSG_DELAY_NANO, u8x8->display_info->pre_cl
      u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_disable_level);
      SPI.endTransaction();
      break;
    default:
      return 0;
  }
  return 1;
}
```

DMA SPI Communication

"Direct Memory Access" (DMA) is available on many modern uC. DMA can transfer data independently from the uC to a target device. This sounds good, but the problem with display devices is, that the data is not directly available for transfer:

- Data for the display has to be calculated on the fly

- Extra commands are required to position the data within the RAM of the display controller
- Sometimes the data itself has to be converted for the target device
- Data transfer has to be in sync with the CD (command/data) signal

As a result, this means, that the data argument ("arg_ptr") provided to the "byte" function becomes invalid after leaving the byte function. The "arg_ptr" can't be used as soruce for any background data transfer.

The following template code shows a solution for this: The data is copied to a backup array, from where DMA takes the soruce data.

```
uint8_t dma_buffer[256]; /* required for DMA transfer */

extern "C" uint8_t u8x8_byte_dma_spi(u8x8_t *u8x8, uint8_t msg, uint8_t arg_int, v
  uint16_t i;
  switch(msg) {
    case U8X8_MSG_BYTE_SEND:
      /* wait for DMA completion */
      /* ... */
      /* create a copy of the input data */
      for( i = 0; i < arg_int; i++ )
        dma_buffer[i] = ((uint8_t *)arg_ptr)[i];
      /* create DMA SPI Transfer for arg_int bytes, located at dma_buffer */
      /* ... */
      break;
    case U8X8_MSG_BYTE_INIT:
      /* setup SPI & DMA */
      /* ... */
      u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_disable_level);
      break;
    case U8X8_MSG_BYTE_SET_DC:
      /* Wait for DMA SPI transfer completion */
      /* ... */
      u8x8_gpio_SetDC(u8x8, arg_int);
      break;
    case U8X8_MSG_BYTE_START_TRANSFER:
      u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_enable_level);
      u8x8->gpio_and_delay_cb(u8x8, U8X8_MSG_DELAY_NANO, u8x8->display_info->post_
      break;
    case U8X8_MSG_BYTE_END_TRANSFER:
      /* Wait for DMA SPI transfer completion */
      /* ... */
      u8x8->gpio_and_delay_cb(u8x8, U8X8_MSG_DELAY_NANO, u8x8->display_info->pre_cl
      u8x8_gpio_SetCS(u8x8, u8x8->display_info->chip_disable_level);
      break;
    default:
      return 0;
```

```
    return 0;
  }
  return 1;
}
```

In the above template the uC can now continue with the calculation of the next data block, while the transfer of the previous data block is still ongoing. However some preformance benefit is lost by the required `memcpy` operation. Overall around 5% speed improvement is possible:

Constructor	SysClk	Transfer	FPS	BSS (RAM)
uc1609_slg19264_f	2MHz	HW SPI	4.7	1692
uc1609_slg19264_f	2MHz	DMA SPI	5.0	1948
uc1609_slg19264_1	2MHz	HW SPI	2.6	348
uc1609_slg19264_1	2MHz	DMA SPI	2.6	604

Constructor	SysClk	Transfer	FPS	BSS (RAM)
uc1609_slg19264_f	32MHz	HW SPI	73.8	1692
uc1609_slg19264_f	32MHz	DMA SPI	76.7	1948
uc1609_slg19264_1	32MHz	HW SPI	39.6	348
uc1609_slg19264_1	32MHz	DMA SPI	40.7	604

The performance has been measured on a STM32L031 system, full code is available here: https://github.com/olikraus/u8g2/tree/master/sys/arm/stm32l031x4

Hardware I2C Communication

Hardware abstraction layers for a microcontroller may provide the following ways to use the I2C subsystem:

1. Start-Send-End Interface, which usually contains three function calls. A popular example is the Arduino Wire library.
2. Transfer Interface, which is one function call for the transfer of the I2C data.

The code below is an example for the Start-Send-End Interface. It shows the U8g2 implementation for the Arduino Environment. The following functions will be called:

- `Wire.begin()` : Init the I2C interface.

- `Wire.beginTransmission()` : Provide the I2C address and start the transfer.
- `Wire.write()` : Send some data.
- `Wire.endTransmission()` : Finish I2C communication.

```
uint8_t u8x8_byte_arduino_hw_i2c(u8x8_t *u8x8, uint8_t msg, uint8_t arg_int, 
{
  switch(msg)
  {
    case U8X8_MSG_BYTE_SEND:
      Wire.write((uint8_t *)arg_ptr, (int)arg_int);
      break;
    case U8X8_MSG_BYTE_INIT:
      Wire.begin();
      break;
    case U8X8_MSG_BYTE_SET_DC:
      break;
    case U8X8_MSG_BYTE_START_TRANSFER:
      if ( u8x8->display_info->i2c_bus_clock_100kHz >= 4 )
      {
        Wire.setClock(400000L);
      }
      Wire.beginTransmission(u8x8_GetI2CAddress(u8x8)>>1);
      break;
    case U8X8_MSG_BYTE_END_TRANSFER:
      Wire.endTransmission();
      break;
    default:
      return 0;
  }
  return 1;
}
```

On other systems you may only have one transfer function. Let us assume the following prototype:

```
void i2c_transfer(uint8_t adr, uint8_t cnt, uint8_t *ptr);
```

- `adr` : I2C address (0..127)
- `cnt` : Number of bytes, which should be sent
- `ptr` : Pointer to a memory area, which contains the values to be sent

The byte callback procedure could look like this:

```
uint8_t u8x8_byte_i2c(u8x8_t *u8x8, uint8_t msg, uint8_t arg_int, void *arg_p
{
  static uint8_t buffer[32];            /* u8g2/u8x8 will never send more than 32 b
  static uint8_t buf_idx;
  uint8_t *data;

  switch(msg)
  {
    case U8X8_MSG_BYTE_SEND:
      data = (uint8_t *)arg_ptr;
      while( arg_int > 0 )
      {
        buffer[buf_idx++] = *data;
        data++;
        arg_int--;
      }
      break;
    case U8X8_MSG_BYTE_INIT:
      /* add your custom code to init i2c subsystem */
      break;
    case U8X8_MSG_BYTE_SET_DC:
      /* ignored for i2c */
      break;
    case U8X8_MSG_BYTE_START_TRANSFER:
      buf_idx = 0;
      break;
    case U8X8_MSG_BYTE_END_TRANSFER:
      i2c_transfer(u8x8_GetI2CAddress(u8x8) >> 1, buf_idx, buffer);
      break;
    default:
      return 0;
  }
  return 1;
}
```

U8g2 Startup

The startup sequence for U8g2 with plain C code is described here.

System specific U8g2 ports

The following section links/notes to target specific ports. It is not guaranteed that the information below is complete, but you may find some additional hints.

Atmel SAM

Discussion: https://github.com/olikraus/u8g2/issues/117

Atmel AVR

- If you're using avr-libc (and avr-binutils, avr-gcc):
  - sys/avr/lib contains common code that should work on several uCs:
    - Busy loop delays.
    - Hardware SPI implementation.
    - TODO: Hardware SPI.
  - Please refer to the README and the examples there to learn how to use it.
- If you're using Atmel Studio:
  - Detailed instructions here https://github.com/olikraus/u8g2/wiki/u8g2as7.
  - More recent step by step tutorial on instructables.com: https://www.instructables.com/ST7920-LCD-With-ATmega328-in-Atmel-Studio-Using-SP
  - Software SPI example here https://github.com/olikraus/u8g2/issues/175.
  - Video Tutorial (I2C SSD1306): https://youtu.be/TO7-yxT6Gvw (see also issue 1578)
  - Video Tutorial AVR, HW SPI: https://www.youtube.com/watch?v=LapgGY27OD8
  - Hardware SPI with ST7920: https://github.com/olikraus/u8g2/issues/1954

STM32

Discussions: https://github.com/olikraus/u8g2/issues/179, https://github.com/olikraus/u8g2/issues/840, https://github.com/olikraus/u8g2/issues/2564

External Blog: https://elastic-notes.blogspot.com/2018/10/u8g2-library-usage-with-stm32-mcu.html

U8g2 Template for the STM32F103: https://github.com/nikola-v/u8g2_template_stm32f103c8t6

U8g2 Template for STM32L031@2MHz with SW I2C and HW SPI: https://github.com/olikraus/u8g2/blob/a0eb1bada3cd49b3808915069e661c5b336c0b2a/sys/arm/stm32l031x4/u8g2_test/u8x8cb.c

Video tutorial with good introduction to the callback functions: https://www.youtube.com/watch?v=LyiZdyTcakA

/watch?v=LjLdyYouk4

ESP32 ESP-IDF

Discussion: https://github.com/olikraus/u8g2/issues/187

Code: https://github.com/nkolban/esp32-snippets/tree/master/hardware/displays/U8G2

Video: https://www.youtube.com/watch?v=MipOGBStBbI

PSoC4200M/CY8CKIT-044

External link: https://iotexpert.com/2017/02/01/pinball-driving-oled-using-u8g2-library/

Raspberry Pi

Discussion: https://github.com/olikraus/u8g2/issues/457

Code: https://github.com/ribasco/u8g2-rpi-demo

Arm Linux

Description: Port for general arm linux board such as raspberry pi, orange pi, nano pi, and etc.

Code: https://github.com/wuhanstudio/u8g2-arm-linux

This port for arm-linux is now also avaiable in the upstream repo:

Code: https://github.com/olikraus/u8g2/tree/master/sys/arm-linux

Raspberry Pi Library Package

Project: libu8g2arm

Description: A simple solution for using U8g2 on the Raspberry Pi. It packages U8g2 and the Arm Linux port to build as a regular C and C++ library.

Code: https://github.com/antiprism/libu8g2arm

RT-Thread

Code: https://github.com/wuhanstudio/rt-u8g2

This port for RT-Thread is now also avaiable in the upstream repo:

Code: https://github.com/olikraus/u8g2/tree/master/sys/rt-thread

nRF52

I2C, see here: https://github.com/olikraus/u8g2/issues/1377 SPI, see here: https://github.com/olikraus/u8g2/issues/1381

RISCV

https://github.com/M-Minhaj/u8g2-with-RISC-V-MCU---CH32V305-7-MCU (https://github.com/olikraus/u8g2/discussions/1963)

▸ **Pages** 77

https://raw.githubusercontent.com/wiki/olikraus/u8g2/ref/u8g2_logo_transparent_orange.png

Home Page and Gallery

Installation (Arduino IDE)

Hardware Setup and Wiring

Font Groups and Icon Fonts

U8g2

U8g2 Reference Manual

U8g2 Fonts

U8g2 C++/Arduino Setup

U8g2 C Setup

U8x8

U8x8 Reference Manual

U8x8 Fonts

U8x8 C++/Arduino Setup

U8x8 C Setup

MUI

MUI User Manual

MUI Reference

Clone this wiki locally

```
https://github.com/olikraus/u8g2.wiki.git
```