Usart driver Documentation

Release 0.7.0

ANSSI

Oct 10, 2019

TABLE OF CONTENTS

This library is an implementation of the STM32F4 U(S)ART hardware devices.

It provide a light abstraction for all the SoC U(S)ART devices in order to map and manipulate a serial interface in a userspace task.

CHAPTER

ONE

ABOUT THE HASH DRIVER

1.1 Principles

STM32F4xx SoCs support up to 6 U(S)ART devices, including 4 potentially synchronous serial devices (USART1, 2, 3, 6) and 2 asynchronous UARTs (4 and 5).

These devices have to be connected to the external world through 2 GPIOs ports. The GPIO backend connection depends on the board HW design and is a part of the driver that is considered as generated by the SDK. See tataouine memory layout informations to understand how the board-specific parts of the drivers are handled.

In the USART driver case, the following are included to get back the board configuration:

```
#include "generated/usart1.h"
#include "generated/usart2.h"
#include "generated/usart3.h"
#include "generated/uart4.h"
#include "generated/uart5.h"
#include "generated/usart6.h"
```

This driver does not take any consideration on the way the device will be used, leaving the task handling the serial in various contexts, including serial console, serial interface to external devices (e.g. ISO7816 serial interface) or any other type of serial communication.

CHAPTER

TWO

ABOUT THE U(S)ART DRIVER API

2.1 Initializing the USART driver

Initializing the usart driver is done with the following API:

```
typedef void (*cb_usart_irq_handler_t) (uint32_t status, uint32_t data);
typedef char (*cb_usart_getc_t) (void);
typedef void (*cb_usart_putc_t) (char);

typedef enum {
  UART       = 0,
  SMARTCARD,
  CUSTOM
} usart_mode_t;

typedef enum {
  USART_SET_BAUDRATE        = 0b000000001,
  USART_SET_WORD_LENGTH     = 0b000000010,
  USART_SET_PARITY          = 0b000000100,
  USART_SET_STOP_BITS       = 0b000001000,
  USART_SET_HW_FLOW_CTRL    = 0b000010000,
  USART_SET_OPTIONS_CR1     = 0b000100000,
  USART_SET_OPTIONS_CR2     = 0b001000000,
  USART_SET_GUARD_TIME_PS   = 0b010000000,
  USART_SET_CB_RCV_IRQ      = 0b100000000,
  USART_SET_ALL             = 0b111111111
} usart_mask_t;

typedef enum {
    USART_MAP_AUTO,
    USART_MAP_VOLUNTARY
} usart_map_mode_t;


typedef struct __packed {
    uint32_t set_mask;
    uint8_t usart;
    usart_mode_t mode;
    uint32_t baudrate;
    uint32_t word_length;
    uint32_t parity;
    uint32_t stop_bits;
    uint32_t hw_flow_control;
    uint32_t options_cr1;
```

(continues on next page)

(continued from previous page)

```
    uint32_t options_cr2;
    uint32_t guard_time_prescaler;
    cb_usart_irq_handler_t callback_irq_handler;
    cb_usart_getc_t *callback_usart_getc_ptr;
    cb_usart_putc_t *callback_usart_putc_ptr;
} usart_config_t;


uint8_t usart_early_init(usart_config_t * config, usart_map_mode_t map_mode);
uint8_t usart_init(usart_config_t * config);
```

Warning: Keep the usart config in the task context, as it is used multiple time in the driver lifecycle !

At early init time (during the task initialization phase), the driver declare the device, requesting its mapping for its nominal mode. The USART driver support both automatic and voluntary mapping (see kernel API for device registration). In voluntary mapping, beware to map the device before using it !

At init time, the device is mapped. The *user_init()* must be called after the end of the task initialization phase and the device must be mapped if it was previously declared as voluntary mapped. This function configure the device using the *config* argument, setting the various USART informations such as the speed, the parity, the flow control and so on.

The USART driver handle one user-defined callback:

- the *callback_irq_handler*, handling IRQ event when a char is received on the USART line.

The USART driver handle two driver-defined callbacks:

- *the usart_getc_ptr*, which is updated in the config structure with the corresponding USART getc function. The task **must** use this function to get a char on the USART line
- *the usart_putc_ptr*, which is updated in the config structure with the corresponding USART putc function. The task **must** use this function to put a char on the USART line

These three callbacks are set by *usart_init()*.

2.2 Mapping and unmapping the USART device

When declaring the USART device in USART_MAP_VOLUNTARY mode, the USART device has to be map and unmap depending on the task memory constraints. The USART driver provides easy-to-use API for this:

```
#include "libusart.h"

int usart_map(void);
int usart_unmap(void);
```

Danger: Use these API only when declaring the device in USART_MAP_VOLUNTARY mode. The kernel refuses to (un)map a device that has been declared in USART_MAP_AUTO mode, making these functions fail

2.3 Enabling and disabling a device

It is possible to enable or disable a U(S)ART device. Disabling a device will block any incoming or outgoing communication. Enabling a device will re-enabling such communication.

Hint: The U(S)ART device does not hold any effective memory of the currently being processed character. When disabling or enabling a device, the current communication is dropped and lost

Danger: Disabling the U(S)ART does not mean deactivating the device input clock. From a hardware point of vue, the device still has its clock input working and is still configurable

2.4 Writing and reading from usart

2.4.1 getc and putc

Using getc and putc can be done easily using the getc and putc pointer set at init time by the USART driver. These functions are blocking functions waiting for the character to be received or to be sent:

```c
#include "libusart.h"

cb_usart_getc_t my_getc = NULL;
cb_usart_putc_t my_putc = NULL;

usart_config_t config;
[...]
config.callback_usart_getc_ptr = &my_getc;
config.callback_usart_getc_ptr = &my_putc;

[...]
usart_init(config);

[...]
char out = 'A';
char in;

in = my_getc();
my_putc(out);
```

2.4.2 Higher read and write access

The USART driver provides higher level abstraction to communicate on serial devices. This permit to read and write buffers directly on the serial line, holding correctly the USART constraints.

This permits, for example, to implement a serial console.

This API is the following:

```c
#include "libusart.h"
```

(continues on next page)

(continued from previous page)

```
void usart_write(uint8_t usart, char *msg, uint32_t len);
uint32_t usart_read(uint8_t usart, char *buf, uint32_t len);
```

Warning: Be sure to use the correct usart identifier, set in your config.usart field, otherwise you will try to access an unmapped device !

Todo: Describes the way U(S)ART DMA are handled

Todo: Describes each field of the config structure correctly

CHAPTER

THREE

USART DRIVER FAQ

3.1 Can I declare multiple U(S)ART devices in the same time ?

Yes, as far as you handle correctly your various configs. You may need, depending on you memory constraints, to map them voluntary.

3.2 In voluntary mode where only my ISR handle the device, should I keep (un)mapping the device ?

No! The map mode is for the main thread only. In ISR mode, the corresponding device is always mapped.