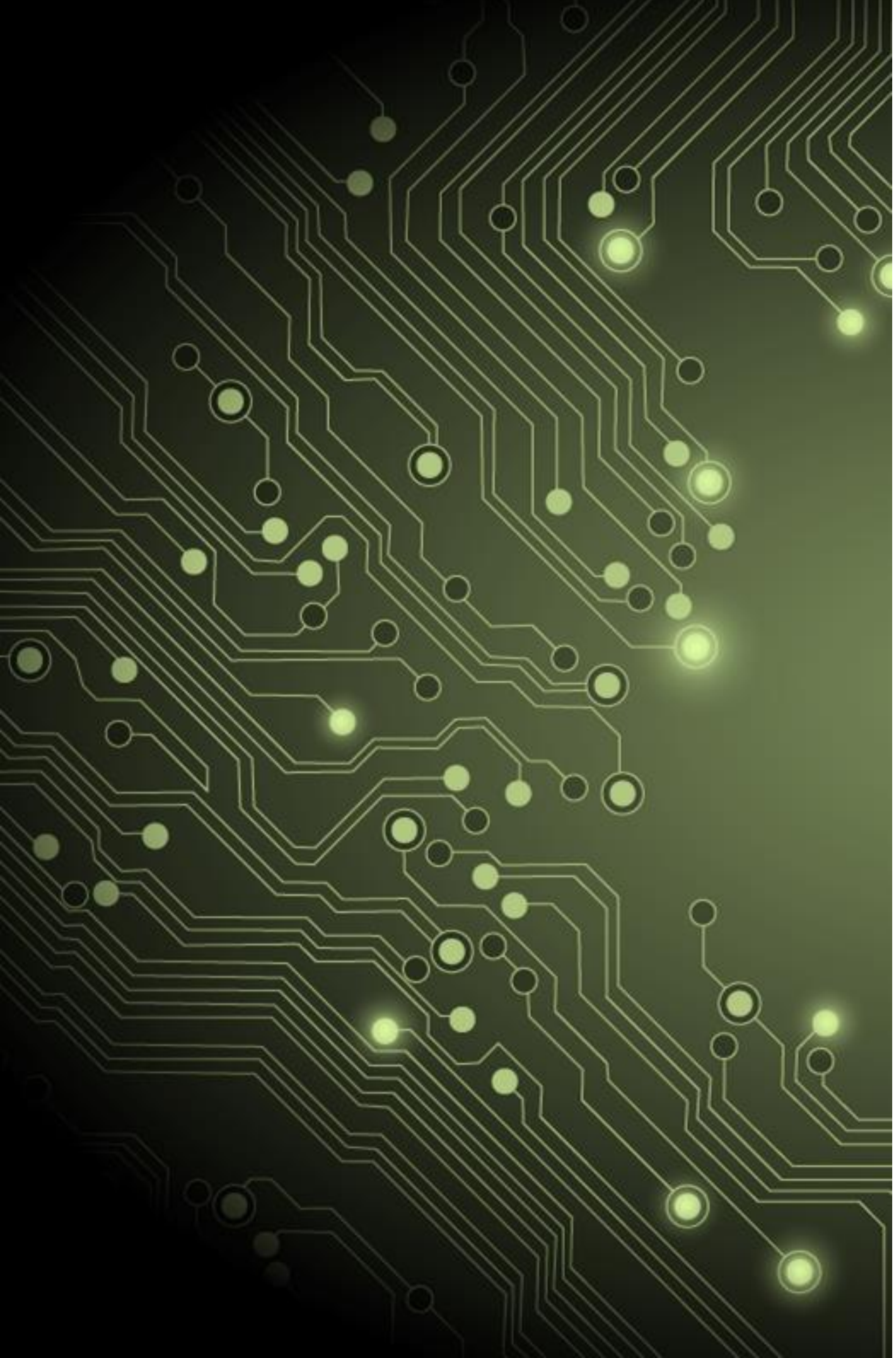


STM32 Microcontrollers Course
Hamed Jafarzadeh

Summer 2016

Final Tips

life.augmented

Serial Peripheral Interface

- SPI is a synchronous serial bus
- Similar to USART but with Master Slave architecture
- Every Slave device has a SS (Slave Select) pin similar to I2C device address
- At least four wires required for establishing a stable connection
- SPI Signals
  - SCLK : Serial Clock (output from master).
  - MOSI : Master Output, Slave Input (output from master).
    - Other names : SDO,DO,DOUT,SO
  - MISO : Master Input, Slave Output (output from slave).
    - Other names : SDI,DI,DIN,SI
  - SS : Slave Select (active low, output from master).
    - Other names : nCS,CSx,EN,nSS

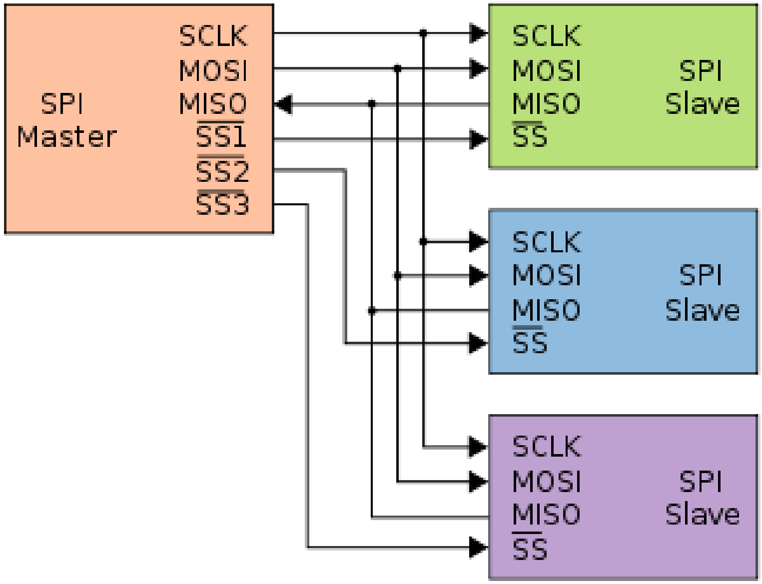
Serial Peripheral Interface

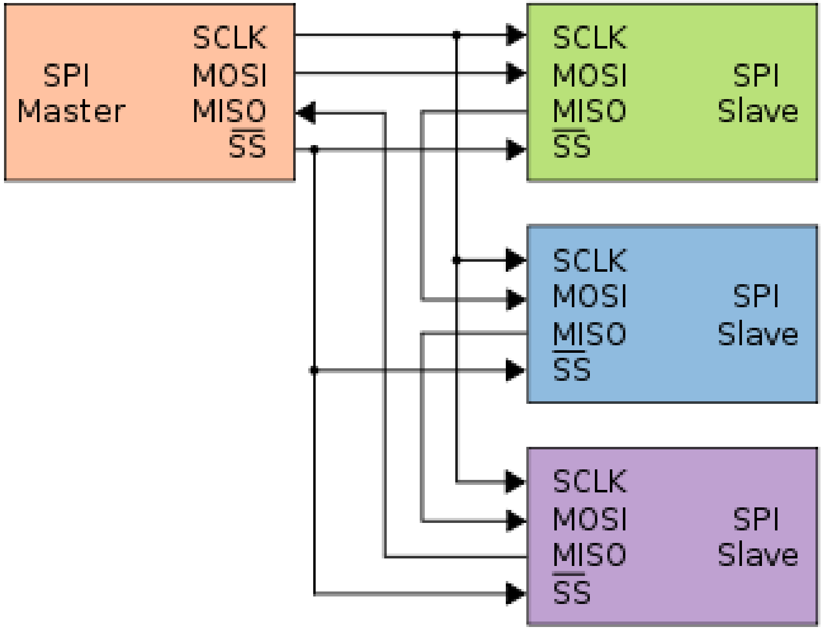
- **Advantages :**
- Full duplex communication in the default version of this protocol.
- Push-pull drivers (as opposed to open drain) provide good signal integrity and high speed
- Higher throughput than I^2C or SMBus
- Extremely simple hardware interfacing
  - Typically lower power requirements than I^2C or SMBus due to less circuitry (including pull up resistors)
  - Slaves use the master's clock, and do not need precision oscillators
  - Slaves do not need a unique address — unlike I^2C
  - Transceivers are not needed
- Uses only four pins on IC packages, and wires in board layouts or connectors, much fewer than parallel interfaces
- At most one unique bus signal per device (chip select); all others are shared
- Not limited to any maximum clock speed, enabling potentially high speed
- Simple software implementation

Serial Peripheral Interface

- **Disadvantages**
- Requires more pins on IC packages than I^2C
- No hardware flow control by the slave (but the master can delay the next clock edge to slow the transfer rate)
- No hardware slave acknowledgment (the master could be transmitting to nowhere and not know it)
- Typically supports only one master device (depends on device's hardware implementation)
- No error-checking protocol is defined
- Without a formal standard, validating conformance is not possible
- Only handles short distances compared to RS-232, RS-485, or CAN-bus
- Many existing variations, making it difficult to find convertors
- SPI does not support hot swapping (dynamically adding nodes).

Serial Peripheral Interface





STM32 HAL Drivers

- A set of rich APIs which provides easy interaction with STM32 microcontrollers
- Each driver uses a standard common API to interact with peripherals

```
HAL_SPI_Receive_DMA(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size)
HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc, uint32_t* pData, uint32_t Length)

HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size)
```

STM32 HAL Drivers

- Cross-family portable set of API
- Three API programming models , IT , DMA , Polling
- APIs are RTOS compatible
  - Systematic Time-Out generations
- All HAL APIs implement user-callbacks functions mechanism
- All HAL APIs provide Error Handling
- Object Locking Mechanism
- Assertion of possible errors

STM32 HAL Drivers

File	Description
system_stm32f4xx.c	This file contains SystemInit() which is called at startup just after reset and before branching to the main program. It does not configure the system clock at startup (contrary to the standard library). This must be performed using the HAL APIs in the user files. It allows to : • relocate the vector table in internal SRAM. • configure FSMC/FMC peripheral (when available) to use as data memory the external SRAM or SDRAM mounted on the evaluation board.
startup_stm32f4xx.s	Toolchain specific file that contains reset handler and exception vectors. For some toolchains, it allows adapting the stack/heap size to fit the application requirements.

STM32 HAL Drivers

stm32f4xx_it.c/.h	This file contains the exceptions handler and peripherals interrupt service routine, and calls HAL_IncTick() at regular time intervals to increment a local variable (declared in stm32f4xx_hal.c) used as HAL timebase. By default, this function is called each 1ms in Systick ISR. . The PPP_IRQHandler() routine must call HAL_PPP_IRQHandler() if an interrupt based process is used within the application.
main.c/.h	This file contains the main program routine, mainly: • the call to HAL_Init() • assert_failed() implementation • system clock configuration • peripheral HAL initialization and user application code.

STM32 HAL Drivers

Peripheral handle structures

The APIs have a modular generic multi-instance architecture that allows working with several IP instances simultaneous.

PPP_HandleTypeDef *handle is the main structure that is implemented in the HAL drivers. It handles the peripheral/module configuration and registers and embeds all the structures and variables needed to follow the peripheral device flow.

The peripheral handle is used for the following purposes:

- Multi-instance support: each peripheral/module instance has its own handle. As a result instance resources are independent.
- Peripheral process intercommunication: the handle is used to manage shared data resources between the process routines.
  Example: global pointers, DMA handles, state machine.
- Storage : this handle is used also to manage global variables within a given HAL driver.

STM32 HAL Drivers

An example of peripheral structure is shown below:

```
typedef struct
{
 USART_TypeDef       *Instance; /* USART registers base address   */
 USART_InitTypeDef     Init;       /* Usart communication parameters  */
 uint8_t           *pTxBuffPtr;/* Pointer to Usart Tx transfer Buffer */
 uint16_t           TxXferSize; /* Usart Tx Transfer size        */
 __IO uint16_t        TxXferCount;/* Usart Tx Transfer Counter       */
 uint8_t           *pRxBuffPtr;/* Pointer to Usart Rx transfer Buffer */
 uint16_t           RxXferSize;  /* Usart Rx Transfer size       */
 __IO uint16_t        RxXferCount; /* Usart Rx Transfer Counter      */
 DMA_HandleTypeDef     *hdmatx;   /* Usart Tx DMA Handle parameters */
 DMA_HandleTypeDef     *hdmarx;   /* Usart Rx DMA Handle parameters */
 HAL_LockTypeDef      Lock;     /* Locking object         */
 __IO HAL_USART_StateTypeDef  State;  /* Usart communication state     */
 __IO HAL_USART_ErrorTypeDef  ErrorCode;/* USART Error code          */
}USART_HandleTypeDef;
```

STM32 HAL Drivers

Initialization and configuration structure

These structures are defined in the generic driver header file when it is common to all part numbers. When they can change from one part number to another, the structures are defined in the extension header file for each part number.

```
typedef struct
{
uint32_t BaudRate;   /*!< This member configures the UART communication baudrate.*/
uint32_t WordLength;  /*!< Specifies the number of data bits transmitted or received
in a frame.*/
uint32_t StopBits;   /*!< Specifies the number of stop bits transmitted.*/
uint32_t Parity;    /*!< Specifies the parity mode. */
uint32_t Mode;     /*!< Specifies wether the Receive or Transmit mode is enabled or
disabled.*/
uint32_t HwFlowCtl;  /*!< Specifies wether the hardware flow control mode is enabled
or disabled.*/
uint32_t OverSampling; /*!< Specifies wether the Over sampling 8 is enabled or
disabled,
to achieve higher speed (up to fPCLK/8).*/
}UART_InitTypeDef;
```

STM32 HAL Drivers

HAL generic APIs

The generic APIs provide common generic functions applying to all STM32 devices. They are composed of four APIs groups:

- **Initialization and de-initialization functions:**HAL_PPP_Init(), HAL_PPP_DeInit()
- **IO operation functions:** HAL_PPP_Read(), HAL_PPP_Write(),HAL_PPP_Transmit(), HAL_PPP_Receive()
- **Control functions:**HAL_PPP_Set (), HAL_PPP_Get ().
- **State and Errors functions:** HAL_PPP_GetState (), HAL_PPP_GetError ().

STM32 HAL Drivers

Table 9: HAL generic APIs

Function Group	Common API Name	Description
Initialization group	*HAL_ADC_Init()*	This function initializes the peripheral and configures the low -level resources (clocks, GPIO, AF..)
	HAL_ADC_DeInit()	This function restores the peripheral default state, frees the low-level resources and removes any direct dependency with the hardware.
IO operation group	*HAL_ADC_Start ()*	This function starts ADC conversions when the polling method is used
	HAL_ADC_Stop ()	This function stops ADC conversions when the polling method is used
	HAL_ADC_PollForConversion()	This function allows waiting for the end of conversions when the polling method is used. In this case, a timout value is specified by the user according to the application.
	HAL_ADC_Start_IT()	This function starts ADC conversions when the interrupt method is used
	HAL_ADC_Stop_IT()	This function stops ADC conversions when the interrupt method is used
	HAL_ADC_IRQHandler()	This function handles ADC interrupt requests

STM32 HAL Drivers

HAL extension APIs

HAL extension model overview

The extension APIs provide specific functions or overwrite modified APIs for a specific family (series) or specific part number within the same family.

Table 10: HAL extension APIs

Function Group	Common API Name
HAL_ADCEx_InjectedStart()	This function starts injected channel ADC conversions when the polling method is used
HAL_ADCEx_InjectedStop()	This function stops injected channel ADC conversions when the polling method is used
HAL_ADCEx_InjectedStart_IT()	This function starts injected channel ADC conversions when the interrupt method is used
HAL_ADCEx_InjectedStop_IT()	This function stops injected channel ADC conversions when the interrupt method is used
HAL_ADCEx_InjectedConfigChannel()	This function configures the selected ADC Injected channel (corresponding rank in the sequencer and sample time)

STM32 HAL Drivers

HAL common resources

The common HAL resources, such as common define enumerations, structures and macros, are defined in *stm32f4xx_hal_def.h*.The main common define enumeration is *HAL_StatusTypeDef*.

- **HAL Status** The HAL status is used by almost all HAL APIs, except for boolean functions and IRQ handler. It returns the status of the current API operations. It has four possible values as described below:

```
Typedef enum
{ HAL OK = 0x00, HAL ERROR = 0x01, HAL BUSY = 0x02, HAL TIMEOUT = 0x03
} HAL_StatusTypeDef;
```

END