Serial Interfaces

Lecture 11

Josh Brake
Harvey Mudd College

Outline

- Serial Interfaces Overview
  - Advantages over parallel
  - Major considerations
  - Overview of protocols
- Serial Peripheral Interface
  - Description
  - MCU configuration
- DS1722 SPI temperature sensor
  - Datasheet overview
- CMSIS

Learning Objectives

By the end of this lecture you should be able to…

- See how the SPI peripheral works on the STM32L432KC
- See how to verify the output using a logic analyzer

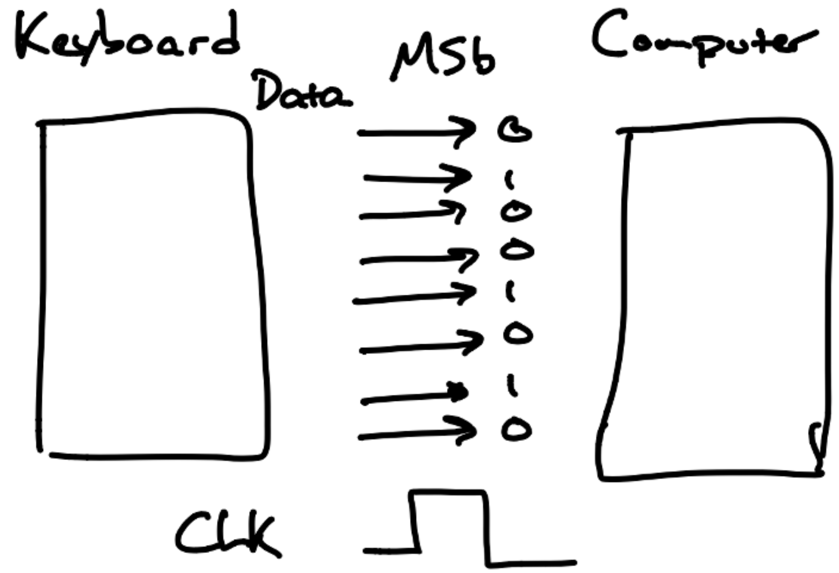
Serial Interfaces Overview

Motivation

How can we interface a peripheral?

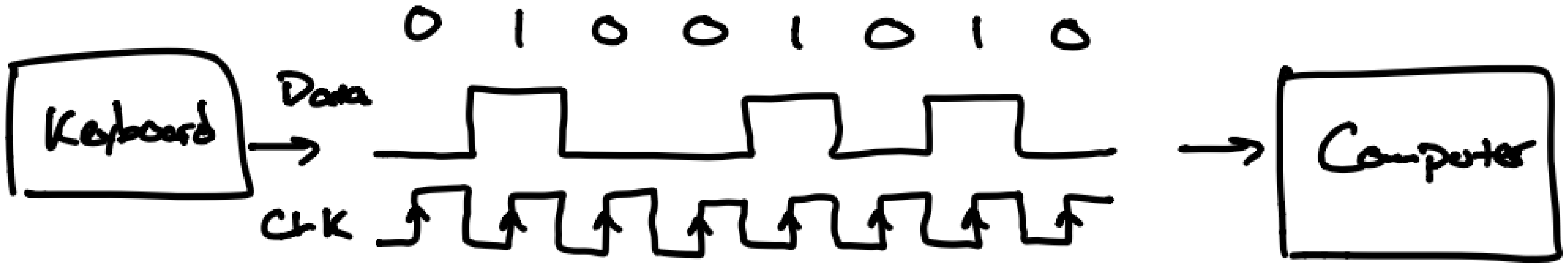
Imagine transmitting a character on a keyboard.

Capital J in ASCII is $74_{10} = 01001010_2$



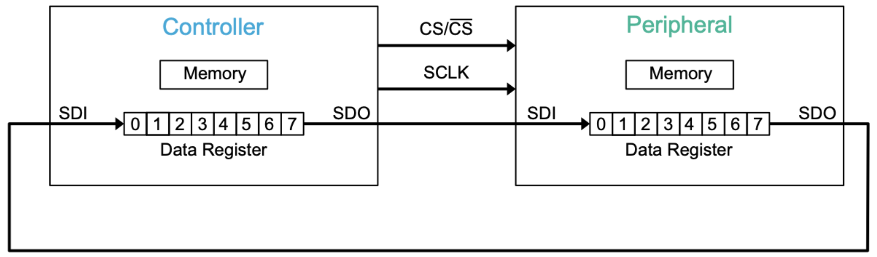
What if we repackage data in a stream?

- Multiplexing in time
- To send N bits, we only need 2 lines (CLK + Data) instead of 9
- Price we pay is time – but often worth it.



Serial Peripheral Interface (SPI)

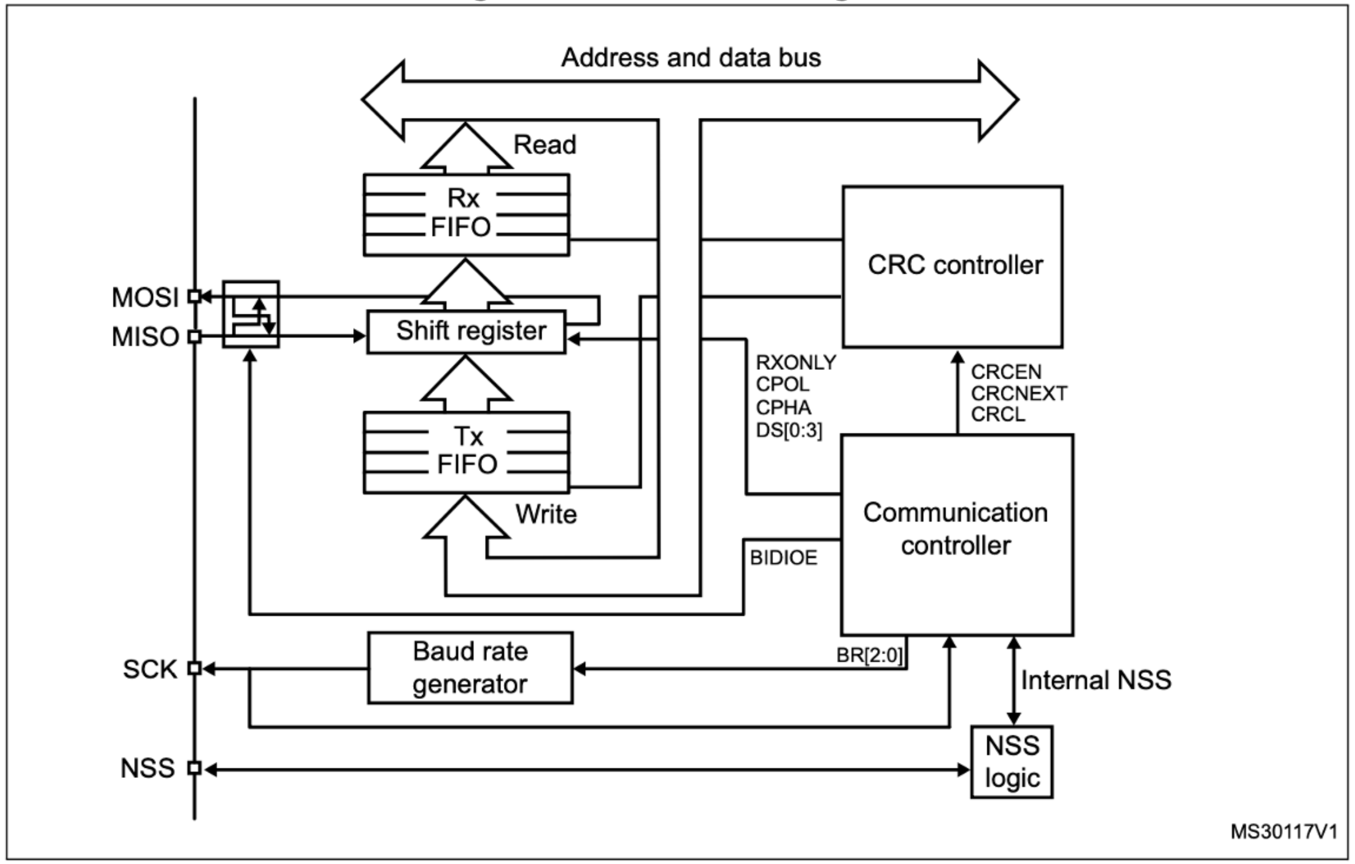
- Developed in the mid-1980s by Motorola
- Used to interface with many peripherals like memory (SD cards, flash), displays, sensors (accelerometers, gyroscopes, temperature sensors, ADCs and DACs).
- Four-wire, synchronous serial bus



SCLK: Serial clock
MOSI: Master Out Slave In
MISO: Master In Slave Out
CE/CS/nCE/nCS: Chip select/enable

SPI Block Diagram on STM32L432KC

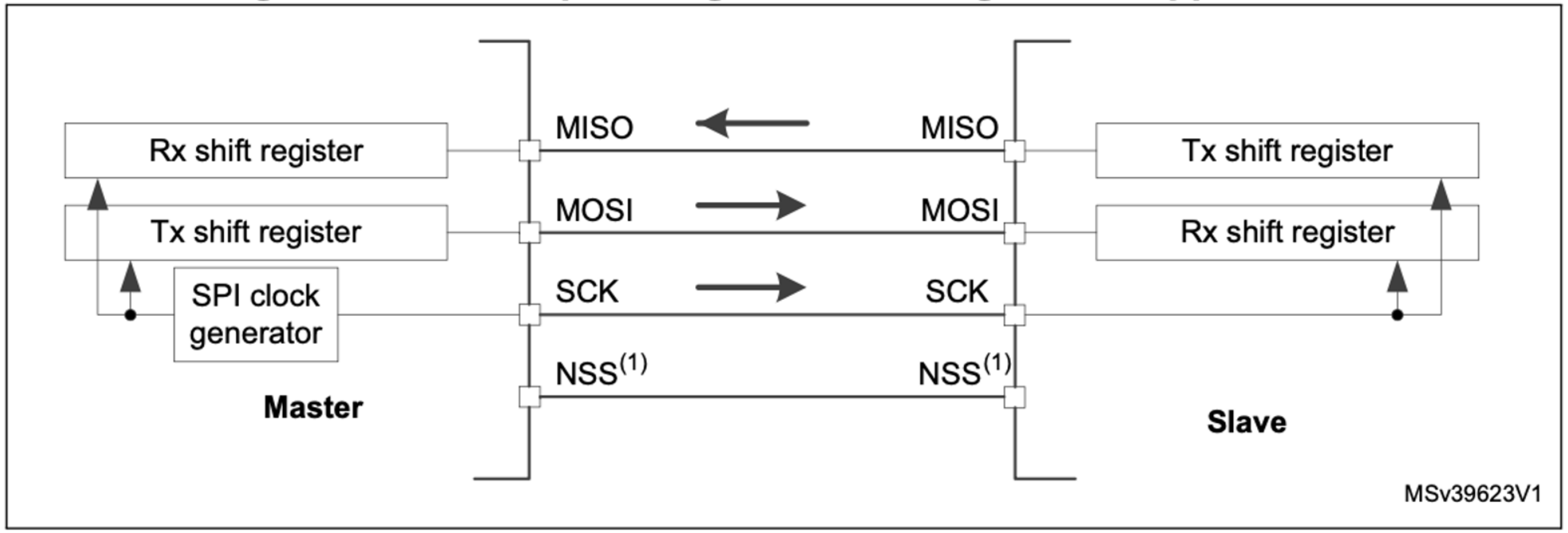
Figure 419. SPI block diagram



RM0394 p. 1305

SPI Block Diagram

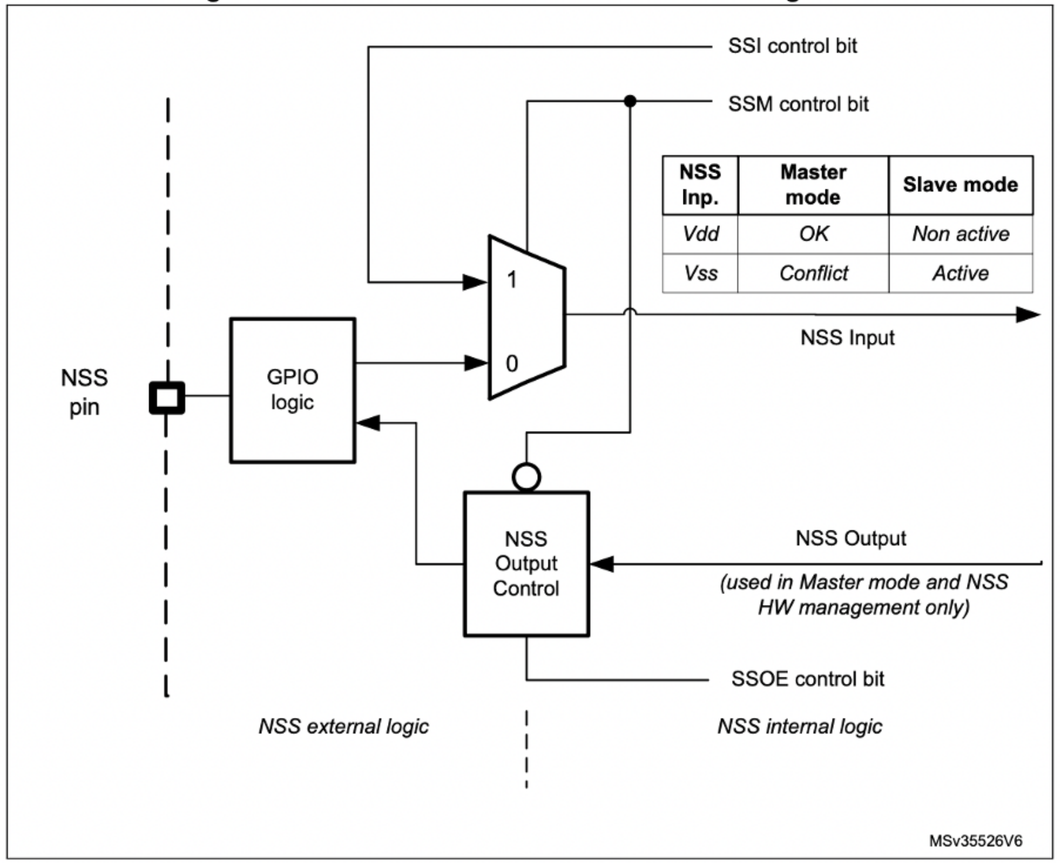
Figure 420. Full-duplex single master/ single slave application



RM0394 p. 1306

SPI Hardware NSS Management

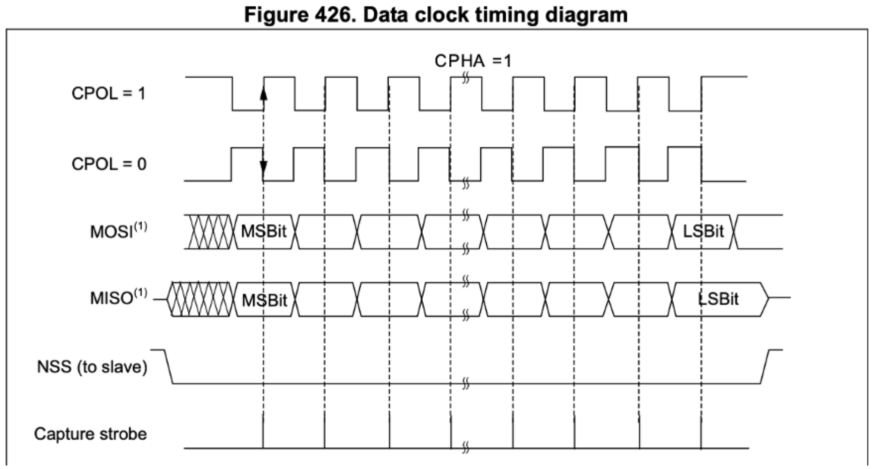
Figure 425. Hardware/software slave select management

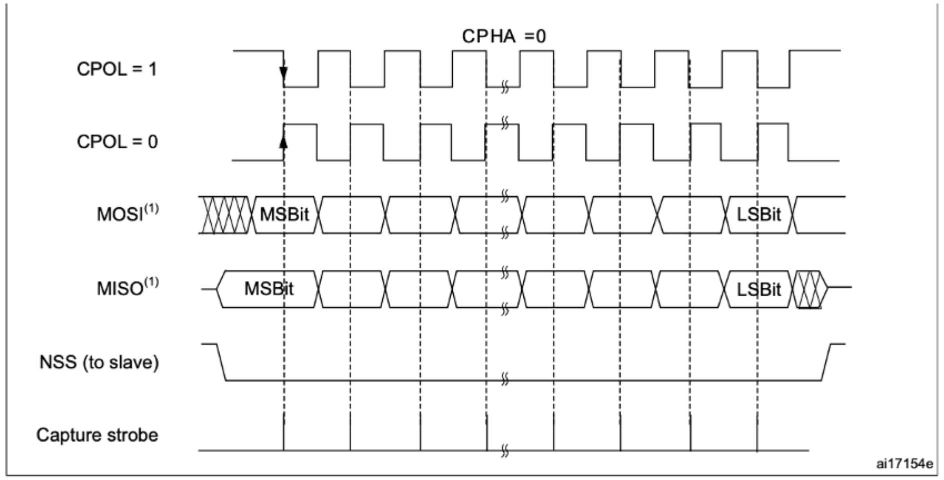


NSS Inp.	Master mode	Slave mode
Vdd	*OK*	*Non active*
Vss	*Conflict*	*Active*

RM0394 p. 1311

Example SPI Traces

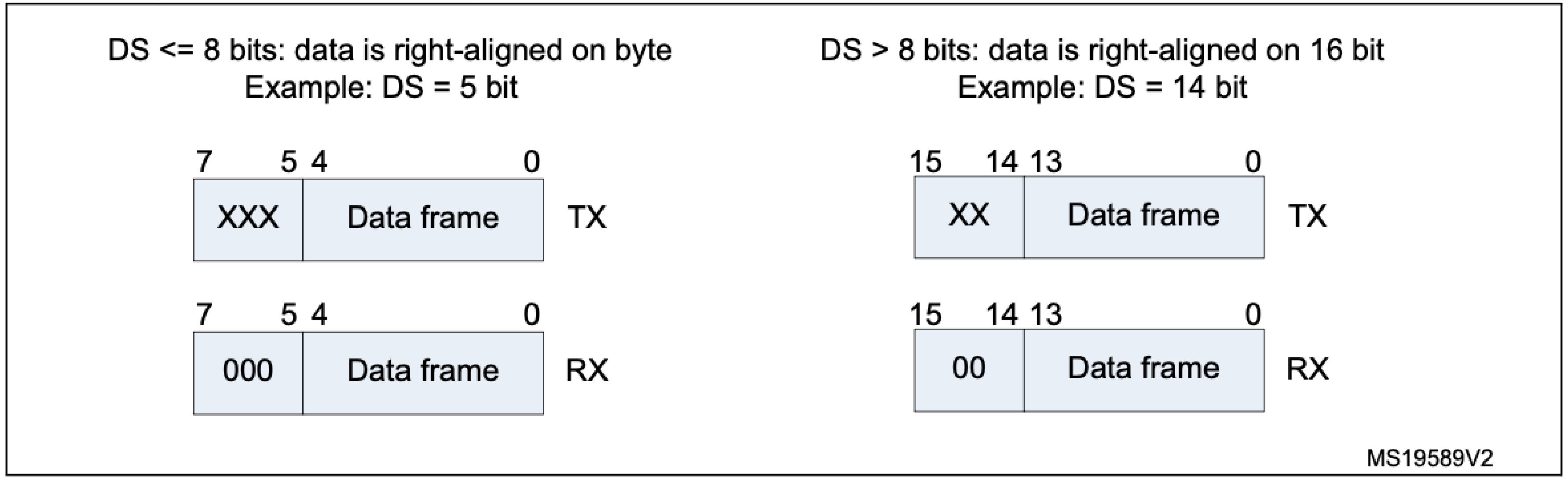




RM0394. p. 1312

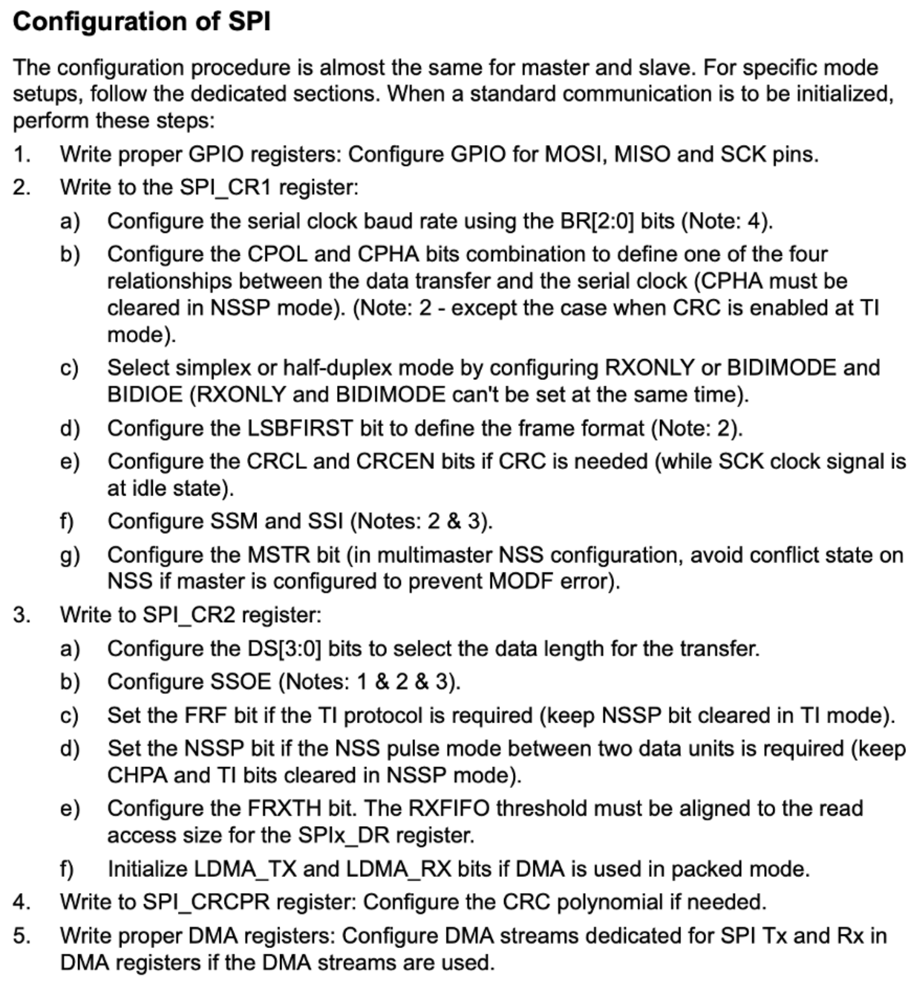
SPI Data Frame Sizes

Figure 427. Data alignment when data length is not equal to 8-bit or 16-bit



The minimum data length is 4 bits. If a data length of less than 4 bits is selected, it is forced to an 8-bit data frame size.

SPI Configuration



Configuration of SPI

The configuration procedure is almost the same for master and slave. For specific mode setups, follow the dedicated sections. When a standard communication is to be initialized, perform these steps:

1. Write proper GPIO registers: Configure GPIO for MOSI, MISO and SCK pins.
2. Write to the SPI_CR1 register:
   a) Configure the serial clock baud rate using the BR[2:0] bits (Note: 4).
   b) Configure the CPOL and CPHA bits combination to define one of the four relationships between the data transfer and the serial clock (CPHA must be cleared in NSSP mode). (Note: 2 - except the case when CRC is enabled at TI mode).
   c) Select simplex or half-duplex mode by configuring RXONLY or BIDIMODE and BIDIOE (RXONLY and BIDIMODE can't be set at the same time).
   d) Configure the LSBFIRST bit to define the frame format (Note: 2).
   e) Configure the CRCL and CRCEN bits if CRC is needed (while SCK clock signal is at idle state).
   f) Configure SSM and SSI (Notes: 2 & 3).
   g) Configure the MSTR bit (in multimaster NSS configuration, avoid conflict state on NSS if master is configured to prevent MODF error).
3. Write to SPI_CR2 register:
   a) Configure the DS[3:0] bits to select the data length for the transfer.
   b) Configure SSOE (Notes: 1 & 2 & 3).
   c) Set the FRF bit if the TI protocol is required (keep NSSP bit cleared in TI mode).
   d) Set the NSSP bit if the NSS pulse mode between two data units is required (keep CHPA and TI bits cleared in NSSP mode).
   e) Configure the FRXTH bit. The RXFIFO threshold must be aligned to the read access size for the SPIx_DR register.
   f) Initialize LDMA_TX and LDMA_RX bits if DMA is used in packed mode.
4. Write to SPI_CRCPR register: Configure the CRC polynomial if needed.
5. Write proper DMA registers: Configure DMA streams dedicated for SPI Tx and Rx in DMA registers if the DMA streams are used.

RM0394 p. 1313

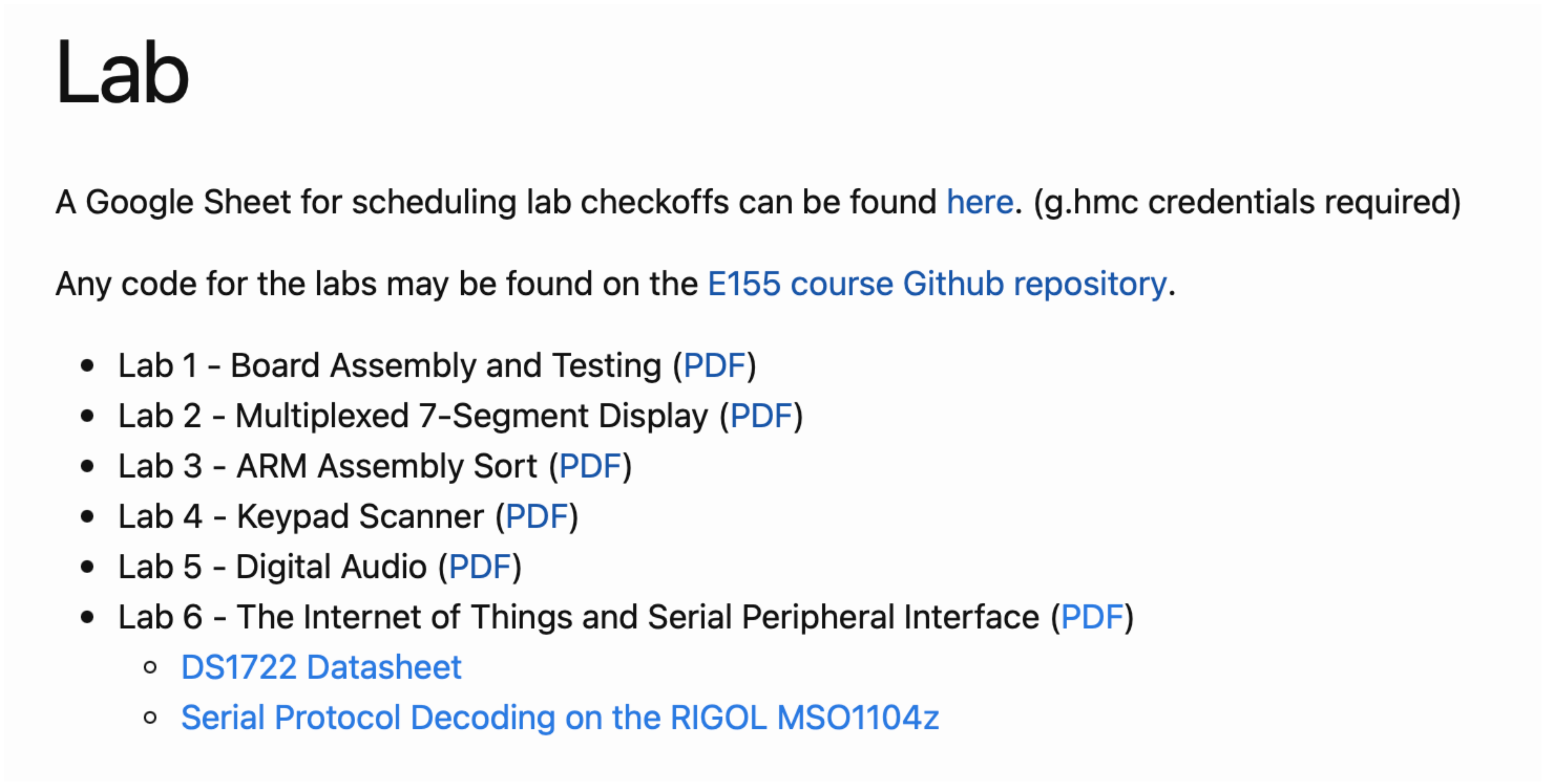
SPI Clock Polarity and Phase

- Clock polarity (CPOL) refers to the state of the clock line at idle
  - 0: clock is low when idle
  - 1: clock is high when idle
- Clock phase (CPHA) refers to when data is sampled vs. when new data is shifted out
  - 0: the first clock transition is the first data capture edge
  - 1: the second clock transition is the first data capture edge
- The clock transition (rising or falling) depends on the clock polarity
- 4 combinations or modes (CPOL,CPHA) = (0,0), (0,1), (1,0), (1,1)
- Must pay attention to match this mode to the peripheral!

Basic Configuration in Master/Controller Mode

- Configure clock tree
- Turn on SPI clock domain
- Set SPI parameters
  - Clock rate using baud rate divisor
  - CPOL and CPHA to match slave
  - DFF to 8- or 16-bit data frame format
  - Set LSBFIRST bit to set whether lsb or msb is sent first (normally msb)
  - Configure the NSS pin (can either use software management or a separate GPIO set as an output and manually toggle it)
  - Set to master mode MSTR
- Enable SPI – Set SPE bit to 1

SPI Demo



Lab

A Google Sheet for scheduling lab checkoffs can be found here. (g.hmc credentials required)

Any code for the labs may be found on the E155 course Github repository.

- Lab 1 - Board Assembly and Testing (PDF)
- Lab 2 - Multiplexed 7-Segment Display (PDF)
- Lab 3 - ARM Assembly Sort (PDF)
- Lab 4 - Keypad Scanner (PDF)
- Lab 5 - Digital Audio (PDF)
- Lab 6 - The Internet of Things and Serial Peripheral Interface (PDF)
  - DS1722 Datasheet
  - Serial Protocol Decoding on the RIGOL MSO1104z

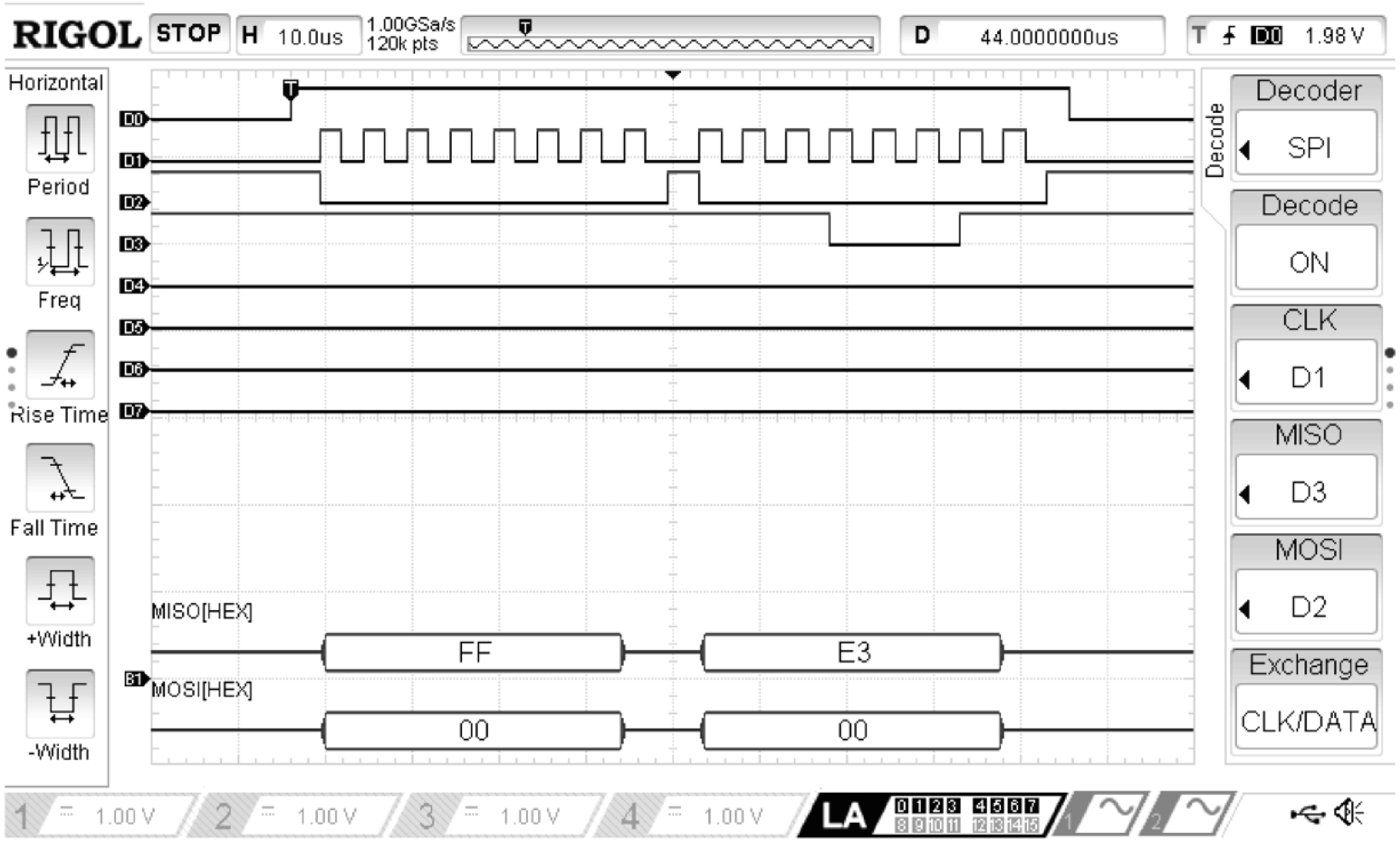
spi_demo.c: main()

```c
int main(void) {
  configureFlash();
  configureClock();

  gpioEnable(GPIO_PORT_A);
  gpioEnable(GPIO_PORT_B);
  gpioEnable(GPIO_PORT_C);

  RCC->APB2ENR |= (RCC_APB2ENR_TIM15EN);
  initTIM(TIM15);

  initSPI(2, 0, 0);

  while(1) {
    digitalWrite(PA11, PIO_HIGH);
    spiSendReceive(0xAB);
    digitalWrite(PA11, PIO_LOW);
    delay_millis(TIM15, 10);
  }
}
```

SPI.h Function Prototypes

```c
 /* Enables the SPI peripheral and intializes its clock speed (baud rate), polarity, and phase.
 *    -- br[2:0]: (0x0 to 0x7). The SPI clk will be the master clock / clkdivide.
 *    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive state is logical 1).
 *    -- cpha: clock phase (0: the first clock transition is the first data capture edge,
 *                          1: the second clock transition is the first data capture edge)
 * Refer to the datasheet for more low-level details. */
void spiInit(int br, int cpol, int cpha);

/* Transmits a character (1 byte) over SPI and returns the received character.
 *    -- send: the character to send over SPI
 *    -- return: the character received over SPI */
char spiSendReceive(char send);
```

An Example SPI Transaction



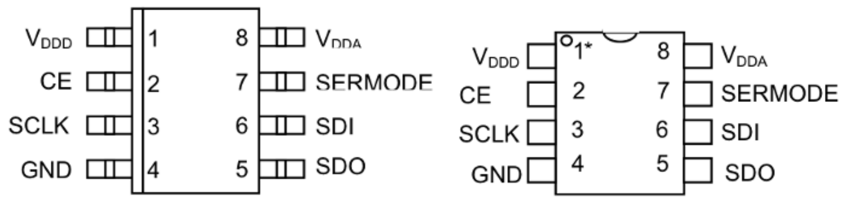
DS1722 SPI Temperature Sensor

DALLAS SEMICONDUCTOR MAXIM

DS1722
Digital Thermometer with SPI/3-Wire Interface

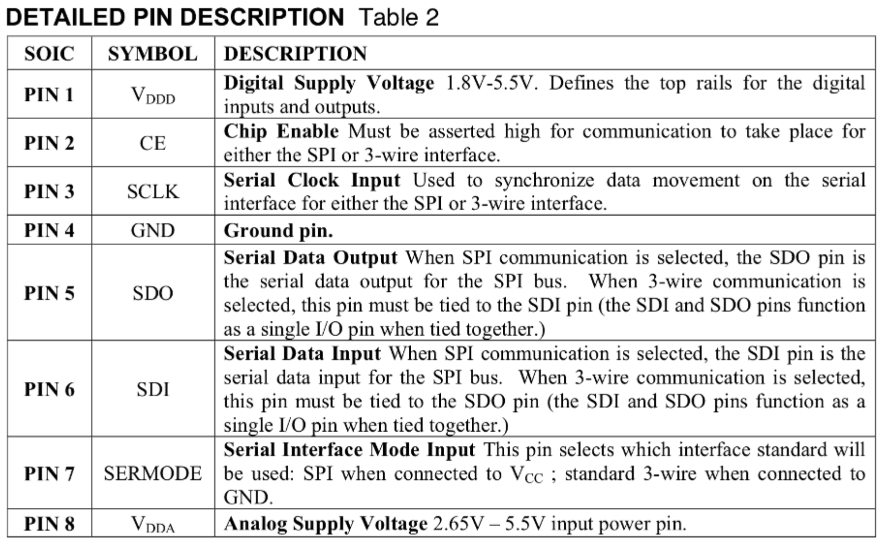
www.maxim-ic.com

PIN ASSIGNMENT



DS1722S
8-Pin SO (150mil)

DS1722U
8-PIN µMAX

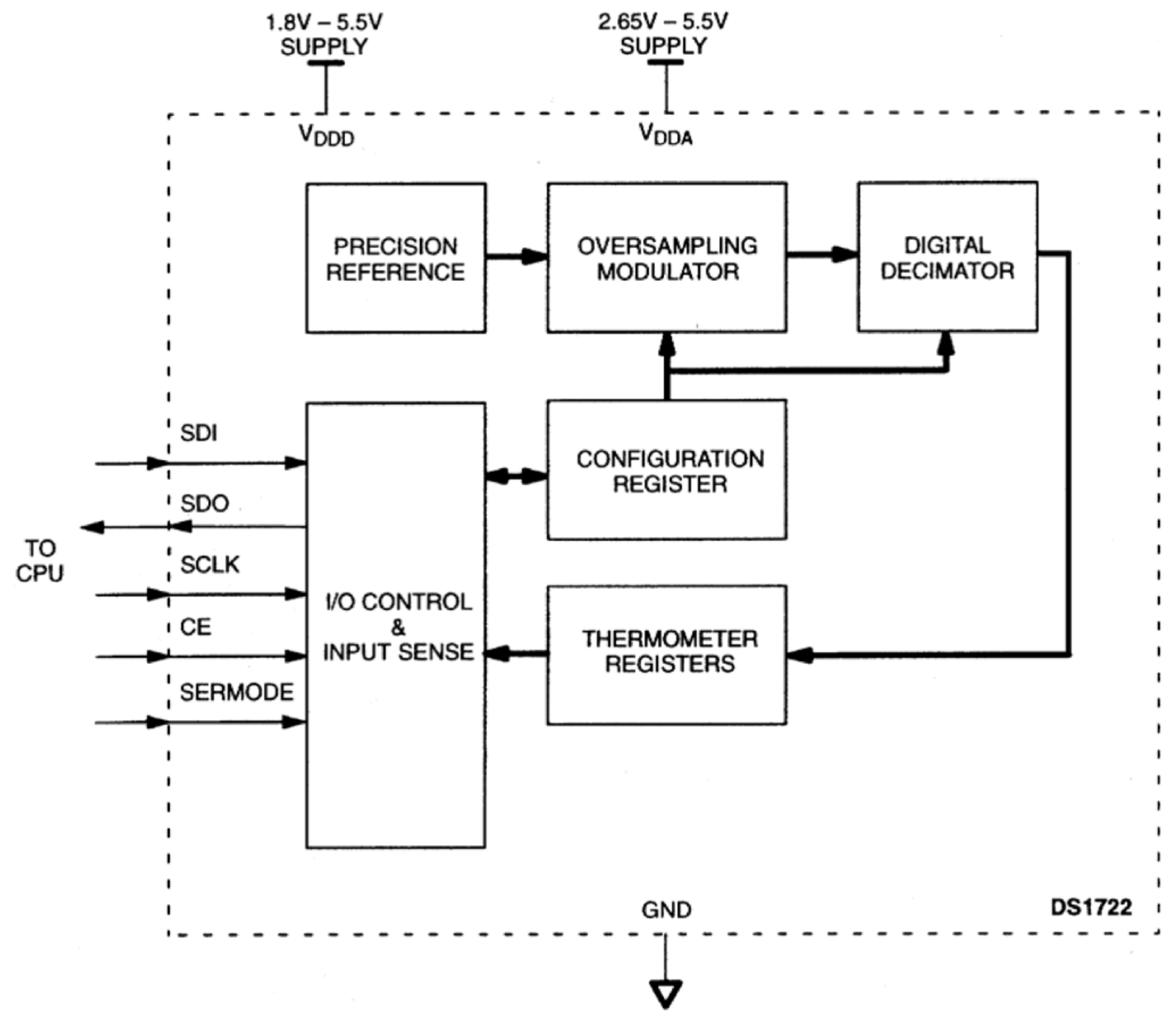


DETAILED PIN DESCRIPTION Table 2

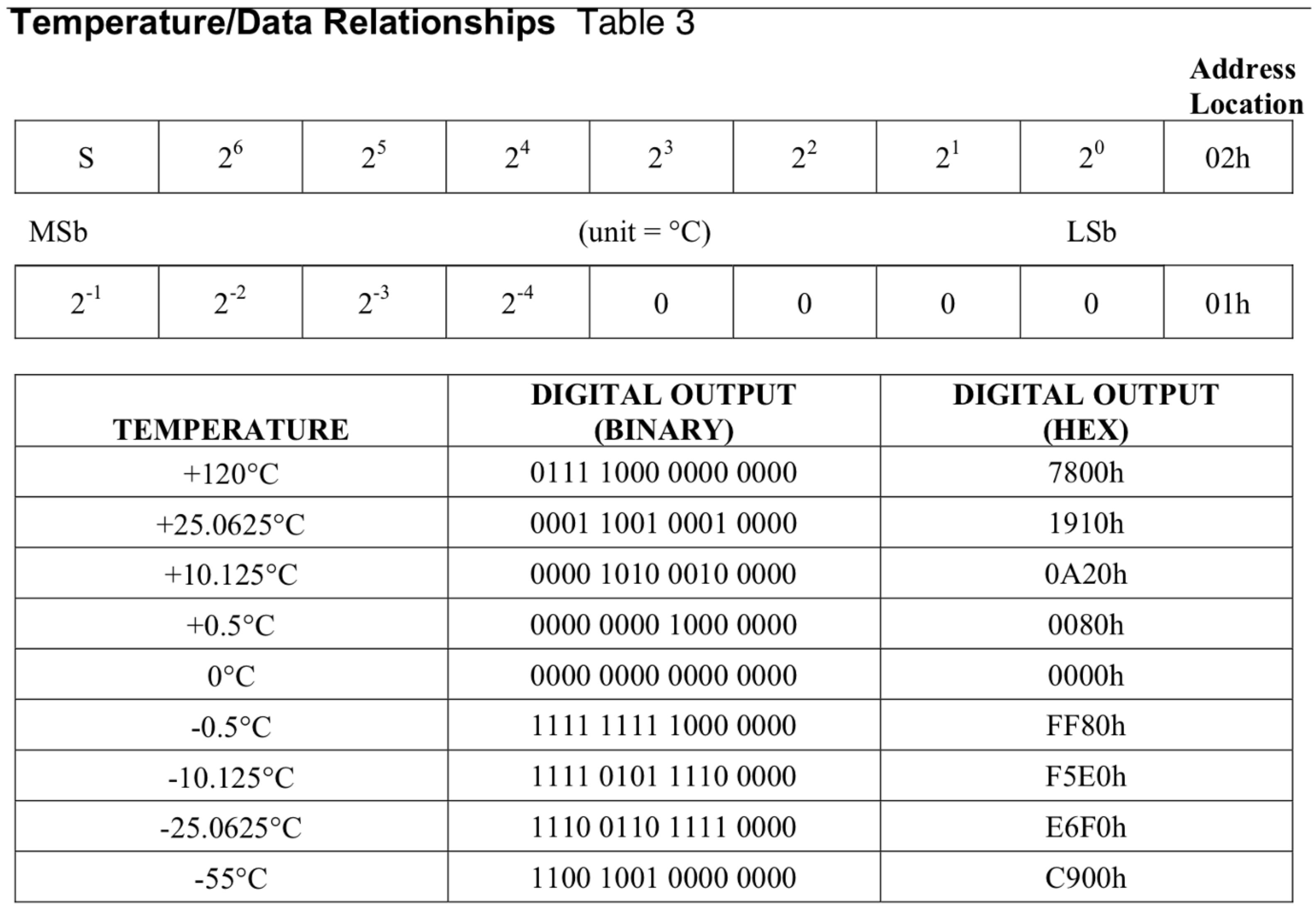
SOIC	SYMBOL	DESCRIPTION
PIN 1	V_{DDD}	**Digital Supply Voltage** 1.8V-5.5V. Defines the top rails for the digital inputs and outputs.
PIN 2	CE	**Chip Enable** Must be asserted high for communication to take place for either the SPI or 3-wire interface.
PIN 3	SCLK	**Serial Clock Input** Used to synchronize data movement on the serial interface for either the SPI or 3-wire interface.
PIN 4	GND	**Ground pin.**
PIN 5	SDO	**Serial Data Output** When SPI communication is selected, the SDO pin is the serial data output for the SPI bus. When 3-wire communication is selected, this pin must be tied to the SDI pin (the SDI and SDO pins function as a single I/O pin when tied together.)
PIN 6	SDI	**Serial Data Input** When SPI communication is selected, the SDI pin is the serial data input for the SPI bus. When 3-wire communication is selected, this pin must be tied to the SDO pin (the SDI and SDO pins function as a single I/O pin when tied together.)
PIN 7	SERMODE	**Serial Interface Mode Input** This pin selects which interface standard will be used: SPI when connected to V_{CC} ; standard 3-wire when connected to GND.
PIN 8	V_{DDA}	**Analog Supply Voltage** 2.65V – 5.5V input power pin.

DS1722 Functional Block Diagram

DS1722 FUNCTIONAL BLOCK DIAGRAM Figure 1



Temperature Data Register Format



Temperature/Data Relationships Table 3

								Address Location
S	2^6	2^5	2^4	2^3	2^2	2^1	2^0	02h

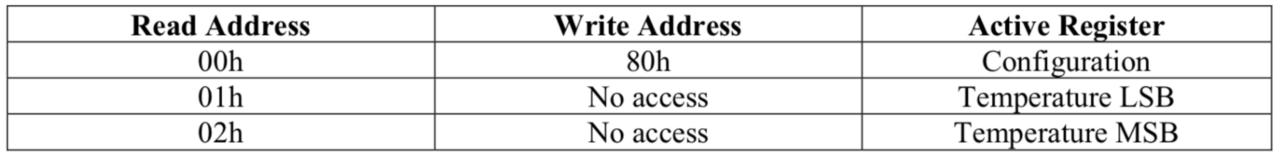
MSb (unit = °C) LSb

2^{-1}	2^{-2}	2^{-3}	2^{-4}	0	0	0	0	01h

TEMPERATURE	DIGITAL OUTPUT (BINARY)	DIGITAL OUTPUT (HEX)
+120°C	0111 1000 0000 0000	7800h
+25.0625°C	0001 1001 0001 0000	1910h
+10.125°C	0000 1010 0010 0000	0A20h
+0.5°C	0000 0000 1000 0000	0080h
0°C	0000 0000 0000 0000	0000h
-0.5°C	1111 1111 1000 0000	FF80h
-10.125°C	1111 0101 1110 0000	F5E0h
-25.0625°C	1110 0110 1111 0000	E6F0h
-55°C	1100 1001 0000 0000	C900h

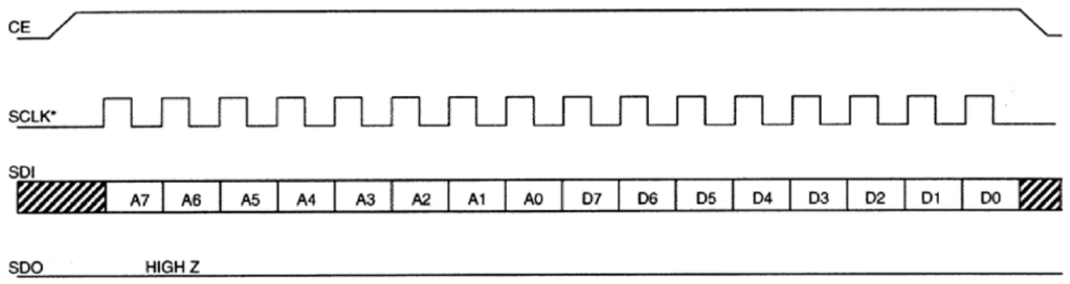
SPI Transactions

Register Address Structure Table 4

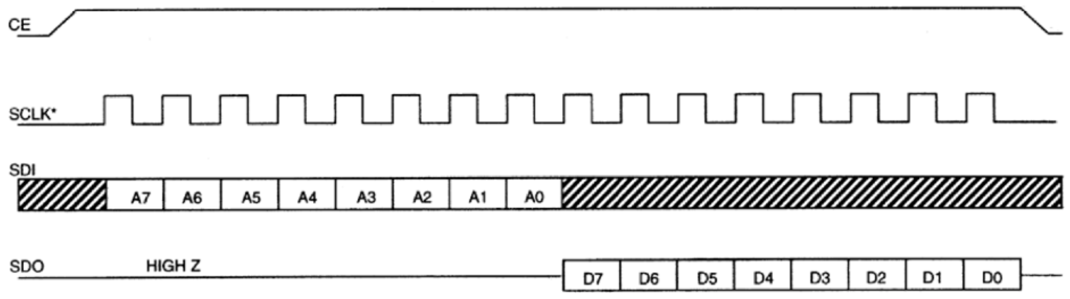


Read Address	Write Address	Active Register
00h	80h	Configuration
01h	No access	Temperature LSB
02h	No access	Temperature MSB

SPI SINGLE BYTE WRITE Figure 4

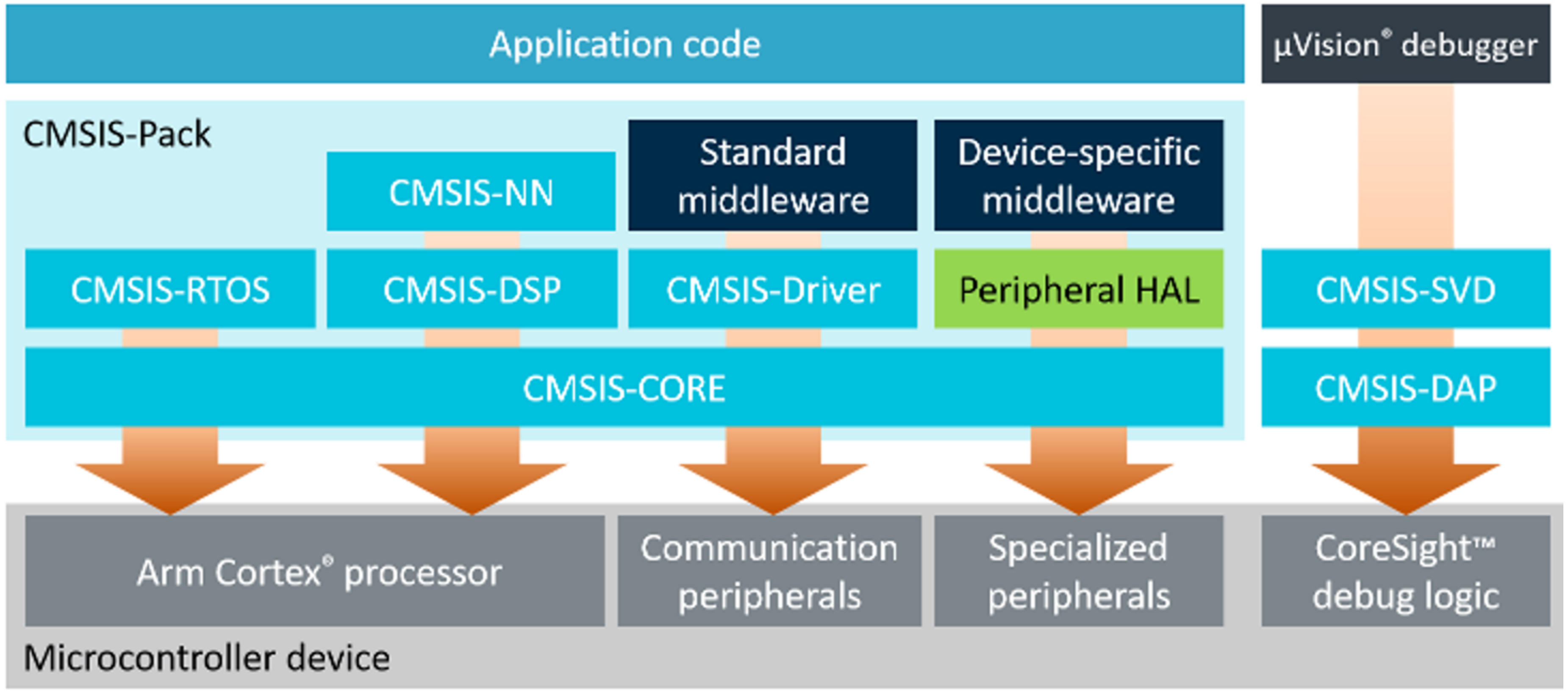


SPI SINGLE-BYTE READ Figure 5



Common Microcontroller Software Interface Standard (CMSIS)

CMSIS Major Components



The benefits of CMSIS

- Software reusability
- Software compatibility
- Easy to learn and use
- Toolchain independent
- Openness

CMSIS-Core Structure

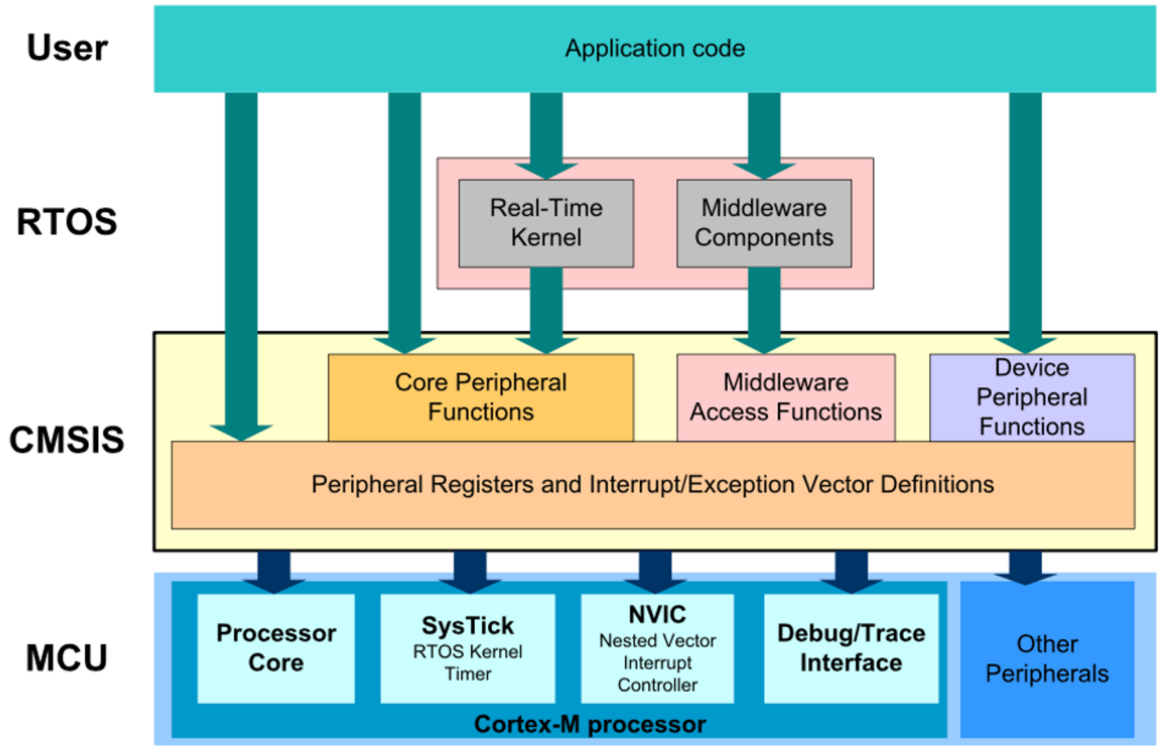

User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick
RTOS Kernel Timer
NVIC
Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals
Cortex-M processor

FIGURE 2.13

CMSIS-Core structure

p. 51 The Definitive guide to ARM Cortex-M3 and Cortex-M4 Processors

Using CMSIS-Core

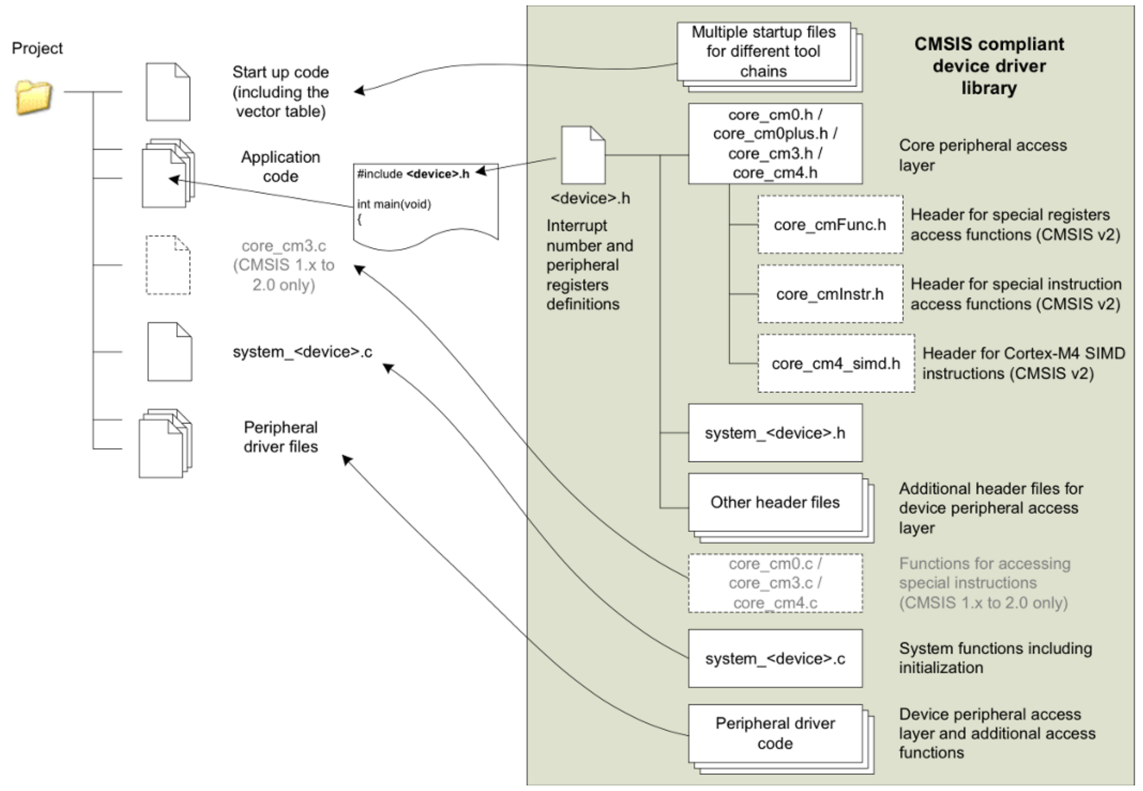

Project
Start up code (including the vector table)
Application code
#include <device>.h
int main(void)
{
core_cm3.c (CMSIS 1.x to 2.0 only)
system_<device>.c
Peripheral driver files
Multiple startup files for different tool chains
CMSIS compliant device driver library
<device>.h
Interrupt number and peripheral registers definitions
core_cm0.h / core_cm0plus.h / core_cm3.h / core_cm4.h
Core peripheral access layer
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm4_simd.h
Header for Cortex-M4 SIMD instructions (CMSIS v2)
system_<device>.h
Other header files
Additional header files for device peripheral access layer
core_cm0.c / core_cm3.c / core_cm4.c
Functions for accessing special instructions (CMSIS 1.x to 2.0 only)
system_<device>.c
System functions including initialization
Peripheral driver code
Device peripheral access layer and additional access functions

FIGURE 2.14

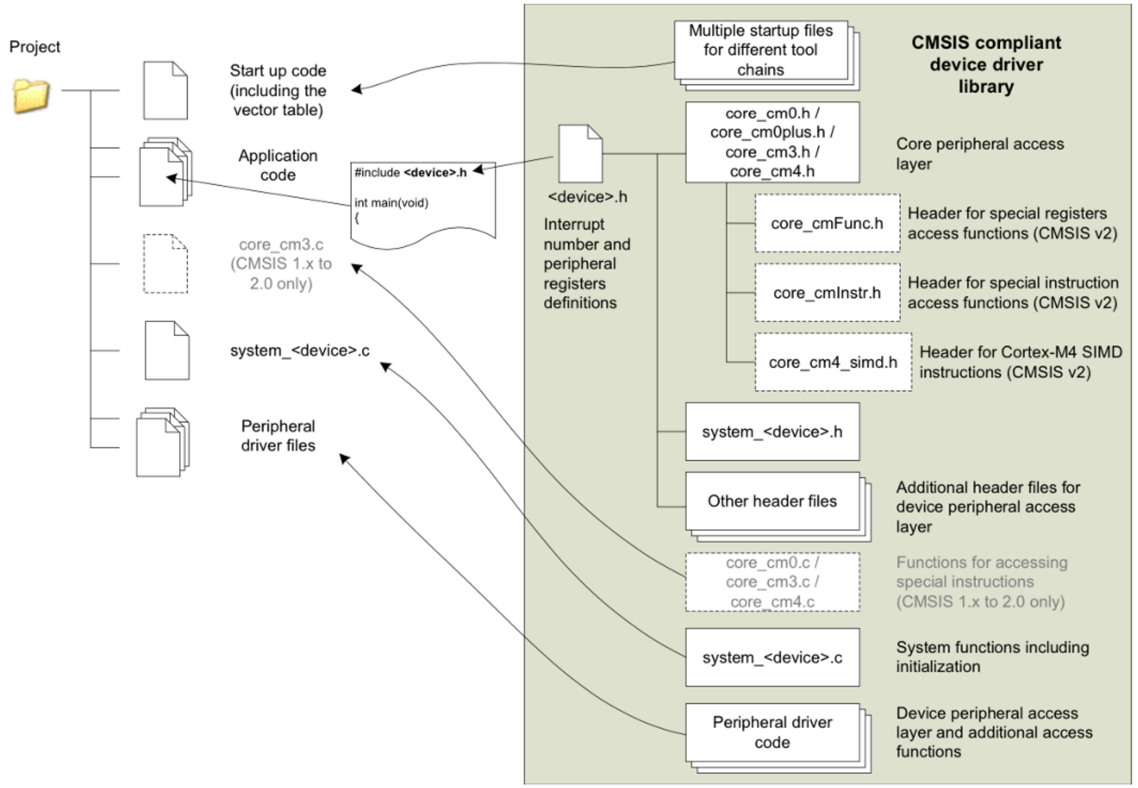
Using CMSIS-Core in a project

p. 52 The Definitive guide to ARM Cortex-M3 and Cortex-M4 Processors

Files to Include in a project

- Startup Code (typically `startup_<device>.c/.s`)
- Application code (`main.c`)
- `<device>.h`
  - `core_cm4.h`
  - `system_<device>.h`
  - `system_<device>.c`
- Peripheral Driver files (custom drivers you write or import)
  - For example, the drivers you are writing for lab.

Using CMSIS-Core: Startup Files



Project
Start up code (including the vector table)
Application code
#include <device>.h
int main(void)
{
core_cm3.c (CMSIS 1.x to 2.0 only)
system_<device>.c
Peripheral driver files
Multiple startup files for different tool chains
CMSIS compliant device driver library
<device>.h
Interrupt number and peripheral registers definitions
core_cm0.h / core_cm0plus.h / core_cm3.h / core_cm4.h
Core peripheral access layer
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm4_simd.h
Header for Cortex-M4 SIMD instructions (CMSIS v2)
system_<device>.h
Other header files
Additional header files for device peripheral access layer
core_cm0.c / core_cm3.c / core_cm4.c
Functions for accessing special instructions (CMSIS 1.x to 2.0 only)
system_<device>.c
System functions including initialization
Peripheral driver code
Device peripheral access layer and additional access functions

FIGURE 2.14

Using CMSIS-Core in a project

p. 52 The Definitive guide to ARM Cortex-M3 and Cortex-M4 Processors

`startup_<device>.c/.s`

`STM32L4xx_Startup.s` - Set the initial SP - Set the initial PC == Reset_Handler, - Set the vector table entries with the exceptions ISR address - Branches to main in the C library (which eventually calls main()).

stm32l432xx_Vectors.s

```
_vectors:
        //
        // Internal exceptions and interrupts
        //
        VECTOR __stack_end__
        VECTOR Reset_Handler
        EXC_HANDLER NMI_Handler
        VECTOR HardFault_Handler
        ISR_RESERVED
        ISR_RESERVED
        ISR_RESERVED
        ISR_RESERVED
        ISR_RESERVED
        ISR_RESERVED
        ISR_RESERVED
        EXC_HANDLER SVC_Handler
        ISR_RESERVED
        ISR_RESERVED
        EXC_HANDLER PendSV_Handler
        EXC_HANDLER SysTick_Handler
 ...
```

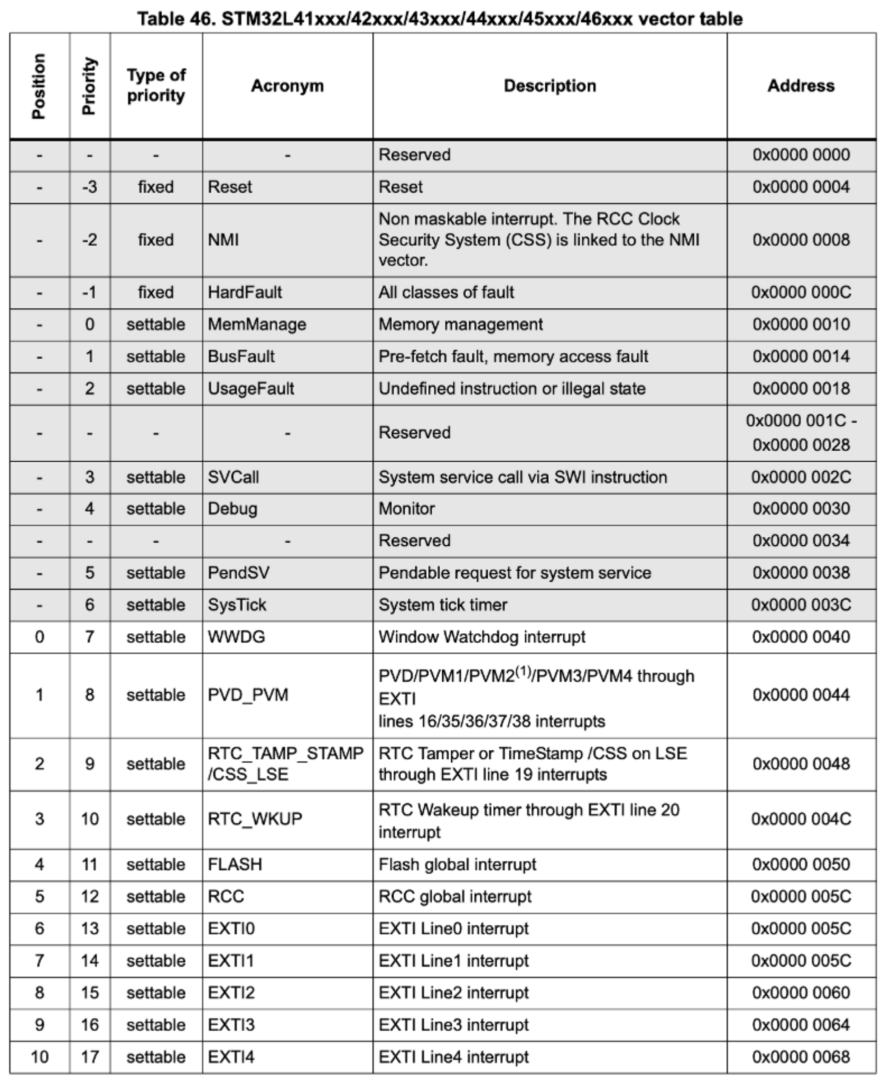
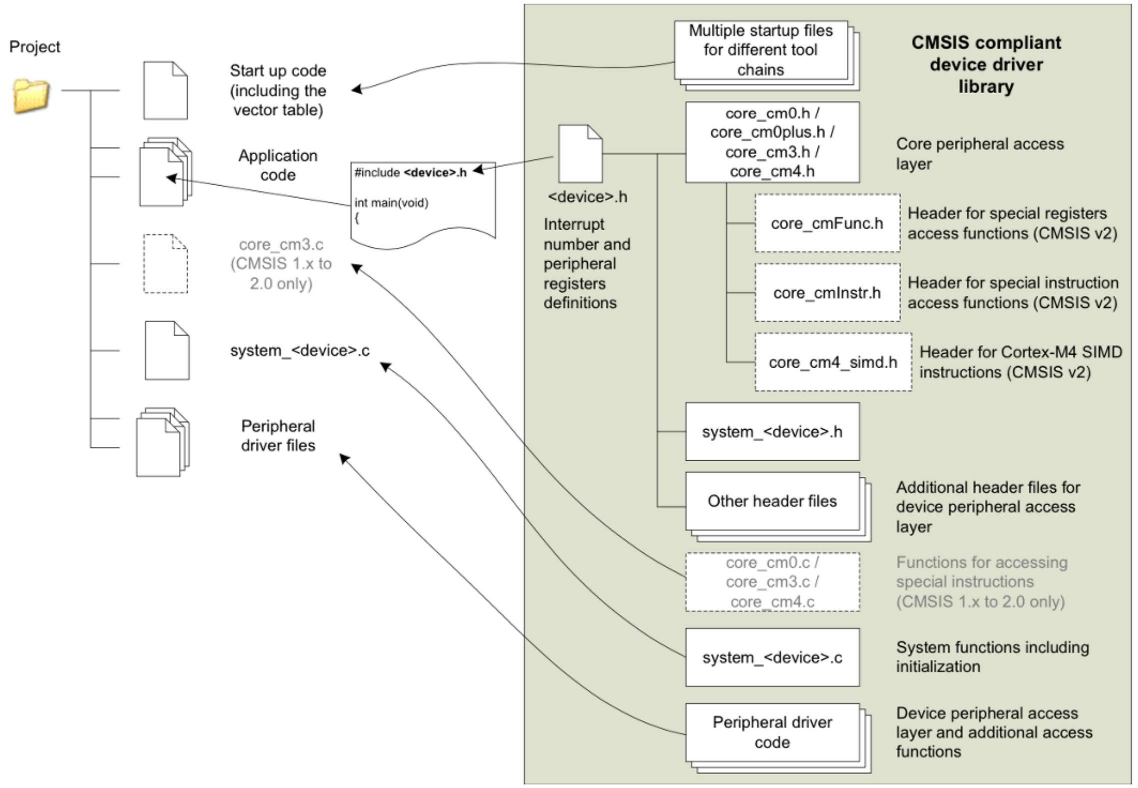


Table 46. STM32L41xxx/42xxx/43xxx/44xxx/45xxx/46xxx vector table

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-1	fixed	HardFault	All classes of fault	0x0000 000C
-	0	settable	MemManage	Memory management	0x0000 0010
-	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
-	-	-	-	Reserved	0x0000 001C - 0x0000 0028
-	3	settable	SVCall	System service call via SWI instruction	0x0000 002C
-	4	settable	Debug	Monitor	0x0000 0030
-	-	-	-	Reserved	0x0000 0034
-	5	settable	PendSV	Pendable request for system service	0x0000 0038
-	6	settable	SysTick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	PVD_PVM	PVD/PVM1/PVM2(1)/PVM3/PVM4 through EXTI lines 16/35/36/37/38 interrupts	0x0000 0044
2	9	settable	RTC_TAMP_STAMP /CSS_LSE	RTC Tamper or TimeStamp /CSS on LSE through EXTI line 19 interrupts	0x0000 0048
3	10	settable	RTC_WKUP	RTC Wakeup timer through EXTI line 20 interrupt	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 005C
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 005C
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068

RM0394 p. 321

Using CMSIS-Core: Device Files



Project
Start up code (including the vector table)
Application code
#include <device>.h
int main(void)
{
core_cm3.c (CMSIS 1.x to 2.0 only)
system_<device>.c
Peripheral driver files
Multiple startup files for different tool chains
CMSIS compliant device driver library
core_cm0.h / core_cm0plus.h / core_cm3.h / core_cm4.h
Core peripheral access layer
<device>.h
Interrupt number and peripheral registers definitions
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm4_simd.h
Header for Cortex-M4 SIMD instructions (CMSIS v2)
system_<device>.h
Other header files
Additional header files for device peripheral access layer
core_cm0.c / core_cm3.c / core_cm4.c
Functions for accessing special instructions (CMSIS 1.x to 2.0 only)
system_<device>.c
System functions including initialization
Peripheral driver code
Device peripheral access layer and additional access functions

FIGURE 2.14

Using CMSIS-Core in a project

p. 52 The Definitive guide to ARM Cortex-M3 and Cortex-M4 Processors

`<device>.h`

- Data structures and the address mapping for all peripherals
- Peripherals registers declarations and bits definition
- Macros to access peripheral’s registers hardware

stm32l432xx.h

Base Addresses

```c
** @addtogroup Peripheral_memory_map
  * @{
  */
#define FLASH_BASE              (0x08000000UL) /*!< FLASH(up to 256 KB) base address */
#define FLASH_END               (0x0803FFFFUL) /*!< FLASH END address                 */
#define FLASH_BANK1_END         (0x0803FFFFUL) /*!< FLASH END address of bank1        */
#define SRAM1_BASE              (0x20000000UL) /*!< SRAM1(up to 48 KB) base address  */
#define SRAM2_BASE              (0x10000000UL) /*!< SRAM2(16 KB) base address */
#define PERIPH_BASE             (0x40000000UL) /*!< Peripheral base address */
#define QSPI_BASE               (0x90000000UL) /*!< QUADSPI memories accessible over AHB base address */

#define QSPI_R_BASE             (0xA0001000UL) /*!< QUADSPI control registers base address */
#define SRAM1_BB_BASE           (0x22000000UL) /*!< SRAM1(96 KB) base address in the bit-band region */
#define PERIPH_BB_BASE          (0x42000000UL) /*!< Peripheral base address in the bit-band region */
```

stm32l432xx.h

SPI Register Mapping

```
/**
  * @brief Serial Peripheral Interface
  */

typedef struct
{
  __IO uint32_t CR1;        /*!< SPI Control register 1,                           Address offset: 0
  __IO uint32_t CR2;        /*!< SPI Control register 2,                           Address offset: 0
  __IO uint32_t SR;         /*!< SPI Status register,                              Address offset: 0
  __IO uint32_t DR;         /*!< SPI data register,                                Address offset: 0
  __IO uint32_t CRCPR;      /*!< SPI CRC polynomial register,                      Address offset: 0
  __IO uint32_t RXCRCR;     /*!< SPI Rx CRC register,                              Address offset: 0
  __IO uint32_t TXCRCR;     /*!< SPI Tx CRC register,                              Address offset: 0
} SPI_TypeDef;
```

Bit definitions

```c
#define GPIO_MODER_MODE0_Pos            (0U)
#define GPIO_MODER_MODE0_Msk            (0x3UL << GPIO_MODER_MODE0_Pos)         /*!< 0x00000003 */
#define GPIO_MODER_MODE0                GPIO_MODER_MODE0_Msk
#define GPIO_MODER_MODE0_0              (0x1UL << GPIO_MODER_MODE0_Pos)         /*!< 0x00000001 */
#define GPIO_MODER_MODE0_1              (0x2UL << GPIO_MODER_MODE0_Pos)         /*!< 0x00000002 */
```

```c
/**
  * @brief General Purpose I/O
  */

typedef struct
{
  __IO uint32_t MODER;       /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;      /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;     /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;       /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;         /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;         /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;        /*!< GPIO port bit set/reset  register,     Address offset: 0x18      */
  __IO uint32_t LCKR;        /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];      /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
  __IO uint32_t BRR;         /*!< GPIO Bit Reset register,               Address offset: 0x28      */

} GPIO_TypeDef;
```

```c
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
```

Using bit definitions

```c
/****************** Bits definition for GPIO_MODER register *****************/
#define GPIO_MODER_MODE0_Pos (0U)
#define GPIO_MODER_MODE0_Msk (0x3U << GPIO_MODER_MODE0_Pos) /*!< 0x00000003 */
#define GPIO_MODER_MODE0 GPIO_MODER_MODE0_Msk
#define GPIO_MODER_MODE0_0 (0x1U << GPIO_MODER_MODE0_Pos) /*!< 0x00000001 */
#define GPIO_MODER_MODE0_1 (0x2U << GPIO_MODER_MODE0_Pos)

#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
```

Example code to set PA0 to OUTPUT (`0x1=0b01`)

```c
GPIOA->MODER &= ~(0b11 << GPIO_MODER_MODE0_Pos) // Clear bits
GPIOA->MODER |= (0b01 << GPIO_MODER_MODE0_Pos)  // Set bit 0
```

Macros in C (`#define`)

Object-like Macros

```
#define <TOKEN_NAME> <TOKEN_VALUE>
#define BUFFER_SIZE 2056
foo = (char *) malloc (BUFFER_SIZE);
```

Function-like Macros

```
#define <MACRO_NAME>(<param1>,<param2>,…) (<stuff to do>)

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

x = min(a, b); → x = ((a) < (b) ? (a) : (b));
y = min(1, 2); → y = ((1) < (2) ? (1) : (2));
```

https://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros

_VAL2FLD Macro

```c
/**
  \brief   Mask and shift a bit field value for use in a register bit range.
  \param[in] field  Name of the register bit field.
  \param[in] value  Value of the bit field. This parameter is interpreted as an uint32_t type.
  \return           Masked and shifted value.
*/
#define _VAL2FLD(field, value)    (((uint32_t)(value) << field ## _Pos) & field ## _Msk)
```

```c
/****************** Bits definition for GPIO_MODER register *****************/
#define GPIO_MODER_MODE3_Pos           (6U)
#define GPIO_MODER_MODE3_Msk           (0x3UL << GPIO_MODER_MODE3_Pos)         /*!< 0x000000C0 */
#define GPIO_MODER_MODE3               GPIO_MODER_MODE3_Msk
#define GPIO_MODER_MODE3_0             (0x1UL << GPIO_MODER_MODE3_Pos)         /*!< 0x00000040 */
#define GPIO_MODER_MODE3_1             (0x2UL << GPIO_MODER_MODE3_Pos)         /*!< 0x00000080 */
```

```
0b00000000000000000000000000000011

0b00000000000000000000000000000011 << 6U = 0b00000000000000000000000011000000
```

_VAL2FLD Macro

```c
/**
  \brief   Mask and shift a bit field value for use in a register bit range.
  \param[in] field  Name of the register bit field.
  \param[in] value  Value of the bit field. This parameter is interpreted as an uint32_t type.
  \return           Masked and shifted value.
*/
#define _VAL2FLD(field, value)    (((uint32_t)(value) << field ## _Pos) & field ## _Msk)
```

Example: Set MODER3 to 0b01 (output)

```c
_VAL2FLD(GPIO_MODER_MODE3, 0b01)
```

Expands to

```c
(((uint32_t)(0b01) << GPIO_MODER_MODE3_Pos) & GPIO_MODER_MODE3_Msk)
```

```c
(0b00000000000000000000000000000001 << 6U) = 0b00000000000000000000000001000000
```

_FLD2VAL Macro

Similar idea to _VAL2FLD but going the other way

```
/**
  \brief     Mask and shift a register value to extract a bit filed value.
  \param[in] field  Name of the register bit field.
  \param[in] value  Value of register. This parameter is interpreted as an uint32_t type.
  \return           Masked and shifted bit field value.
*/
#define _FLD2VAL(field, value)    (((uint32_t)(value) & field ## _Msk) >> field ## _Pos)
```

`system_stm32f4xx.c`

- `SystemInit()`: This function is called at startup just after reset and before branch to main program. This call is made inside the `startup_stm32f4xx.s` file.
- `SystemCoreClock` variable: Contains the core clock (HCLK), it can be used by the user application to setup the SysTick timer or configure other parameters.
- `SystemCoreClockUpdate()`: Updates the variable SystemCoreClock and must be called whenever the core clock is changed during program execution.