Generic EEPROM driver library

Guide

Visibility and dissemination of the document:

Can be widely distributed

Language: **C**
Language version: **C99**
Endianness: **Little endian only**

Synchronous: **yes**
Asynchronous: **no**
OS: **need adaptation**

MCU compatibility:
no limit except endianness

Tests: **full**

MISRA C: **to be determined**
CERT C: **to be determined**

PBIT: **yes**
CBIT: **possible**

Generic I2C EEPROM driver library

Version: 1.0.0   date: 24 August 2020

This library has been tested and is compatible with components:
- AT24C01, AT24C02, AT24C04, AT24C08, AT24C16
- 24xx256
- AT24MACx02 (only EEPROM part)
- 47L/C04, 47L/C16 (only RAM/EEPROM part)

This library is fully compatible with I2C EEPROMs with an address 1010xxx_ compatibility

Established by		Reviewed		Approved	
Name:	**FMA**	Name:	FMA	Name:	FMA
Date:	**24 August 2020**	Date:	24 August 2020	Date:	24 August 2020

Change history

Issue	Date	Nature / comment	Paragraph	Writer
1.0.0	24 August 2020	**Initial release**	-	FMA

Content

Figure summary

Table summary

Aucune entrée de table d'illustration n'a été trouvée.

1. INTRODUCTION

1.1. Purpose

The purpose of this document is to explain how the driver library works and how to use it. It can work with every memory with an address 1010xxx_ compatibility.

The driver features are:

- Can be use with any MCU if its CPU use little endian
- Only take care of the memory, not the communication with it
- All functions and functionalities are implemented
- Configuration is very simplified
- Can communicate with virtually an infinite count of memories
- Different configurations can be used with different memories (no duplication of the driver needed)
- Direct communication with the memories, the driver has no buffer
- Can use the driver defines, enums, structs to create your own functions
- Contiguous memories can be used as 1 unique memory by the driver (under certain conditions)
- Driver will take care of page access to minimize write process and save time
- Driver will take care of address composition of the data

1.2. Documents, References, and abbreviations

1.2.1. Applicable documents

IDENTIFICATION	TITLE	DATE
-	-	-

1.2.2. Reference documents

IDENTIFICATION	TITLE	DATE
-	-	-

1.2.3. Abbreviations and Acronyms

This is the list of all the abbreviations and acronyms used in this document and their definitions. They are arranged in alphabetical order.

CBIT	Continuous Built-In Test
CERT	Computer Emergency Response Team
CLK	Clock
I2C	Inter-Integrated Circuit
MCU	Micro-Controller Unit
MISO	Master In Slave Out
MISRA	Motor Industry Software Reliability Association
MOSI	Master Out Slave In
OS	Operating System
PBIT	Power Up Built-In Test
PIO	Programmable Input/Output
RAM	Random Access Memory

2. PRESENTATION

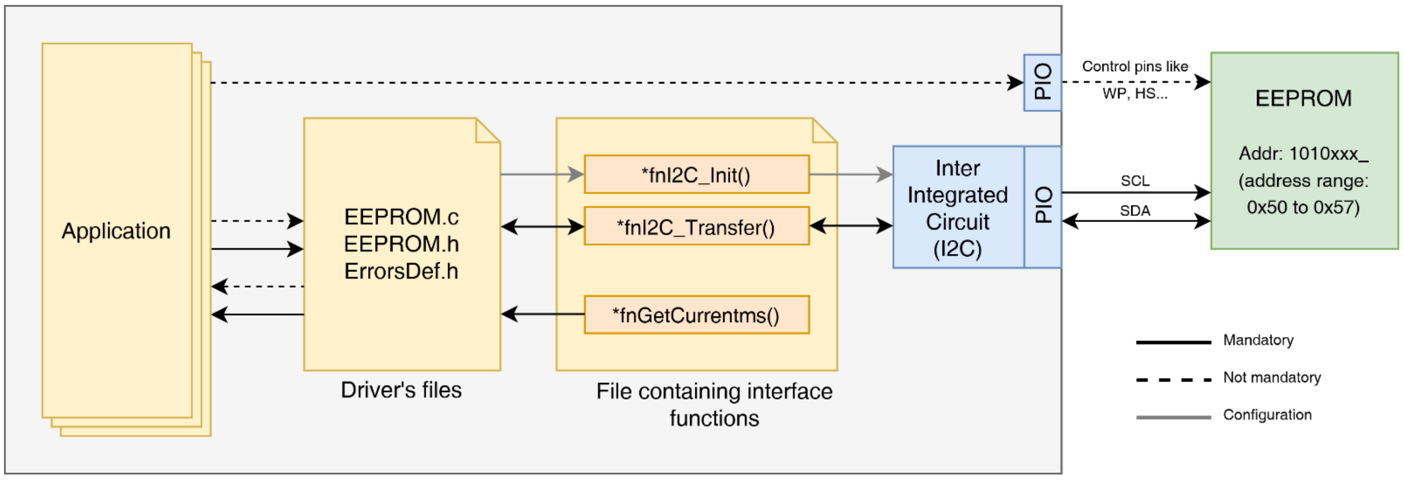


Figure 1 - Driver overview

This driver only takes care of configuration and check of the internal registers and the formatting of the communication with the device. That means it does not directly take care of the physical communication, there are functions interfaces to do that.
Each driver's functions need a device structure that indicate with which device he must threat and communicate and the configuration of the memory. Each device can have its own configuration.

2.1. Setup

To set up one or more devices in the project, you must:

- Create a memory configuration (only one per device part number because they can be shared)
- Create and define the configuration of as many device structures as there are devices to use

2.2. Contiguous memories

The driver can use multiple devices as only one “virtual memory”. To do this, you need to take into account some limitation of the possibility. Devices needs:

- Memories shall be on the same I2C bus
- Addressing to use all bits in the address bytes of the device i.e.:
    - If 1-byte address, the device uses 8, 9, 10 bits
    - If 2-bytes address, the device uses 16, 17, 18 bits
    - If 3-bytes address, the device uses 24, 25, 26 bits
- No unused bits in place of A0, A1, and A2 in the device chip address i.e., fixed as ‘0’ or ‘1’
- Chip addresses shall follow each other and start at the lowest Ax possible chip address pin
- Shall have 2^x component

3. MEMORY CONFIGURATION

To work with a memory, the driver needs to know:

- How to address the device
- How the memory address is composed to access data
- The memory size
- The memory page size (if any)
- Time to write memory (for timeout purpose)

The device configuration type is EEPROM_Conf. All information to fill the structure can be found in the datasheet of the target device.

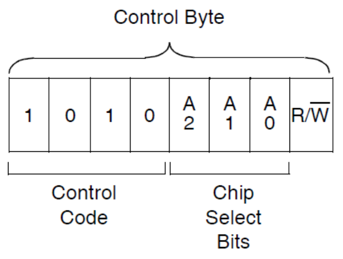
3.1. Example of how to fill an EEPROM_Conf of a device with A0, A1 and A2 pins

The following example was performed on a 24LC256 device.

```
const EEPROM_Conf _24LC256_Conf =
{
  .ChipAddress   = 0xA0,                       // Chip base address
  .ChipSelect    = EEPROM_CHIP_ADDRESS_A2A1A0, // Device have A2, A1 and A0 pins
  .AddressType   = EEPROM_ADDRESS_2Bytes,      // Memory address takes 2 bytes
  .PageWriteTime = 5,                          // 5ms of page write
  .PageSize      = 64,                         // Page size is 64 bytes
  .ArrayByteSize = 512/*Pages*/ *64,           // Memory size is 512*64 = 32768 bytes
  .MaxI2CclockSpeed = 400000,                  // Max I2C SCL speed is 400kHz
};
```

3.1.1. Fill the EEPROM_Conf.ChipAddress

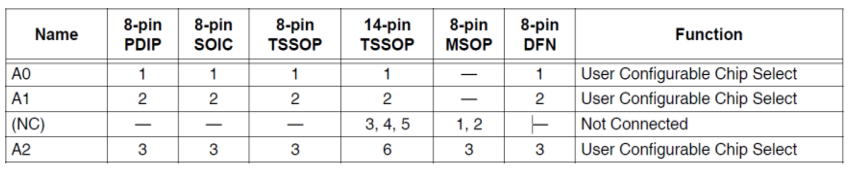
In the 24LC256 datasheet, the information is indicated at §5.0 "DEVICE ADDRESSING" in the Figure 5-2:



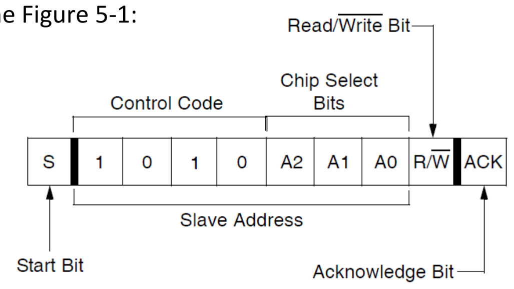
We are looking for the fixed information called here "Control Code", without A2, A1, and A0 bits. The R/$\overline{W}$ bit need to be ignored too. These bits have to be set to '0'. So, we configure EEPROM_Conf.ChipAddress with 0b10100000 or 0xA0.

3.1.2. Fill the EEPROM_Conf.ChipSelect

Here we are looking for which chip select bits are present on this device. In the 24LC256 datasheet, the information is indicated at §2.0 "PIN DESCRIPTIONS" in Table 2-1 and at §5.0 "DEVICE ADDRESSING" in the Figure 5-1:



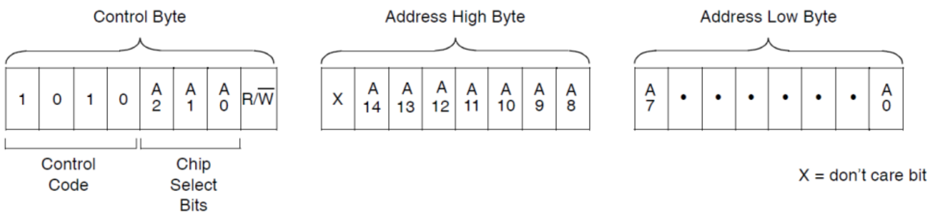
Name	8-pin PDIP	8-pin SOIC	8-pin TSSOP	14-pin TSSOP	8-pin MSOP	8-pin DFN	Function
A0	1	1	1	1	—	1	User Configurable Chip Select
A1	2	2	2	2	—	2	User Configurable Chip Select
(NC)	—	—	—	3, 4, 5	1, 2	—	Not Connected
A2	3	3	3	6	3	3	User Configurable Chip Select



The device has A0, A1, and A2 as user configurable chip select. They are all configurable at '0' or '1'.
So, we configure EEPROM_Conf.ChipSelect with EEPROM_CHIP_ADDRESS_A2A1A0.

3.1.3. Fill the EEPROM_Conf.AddressType

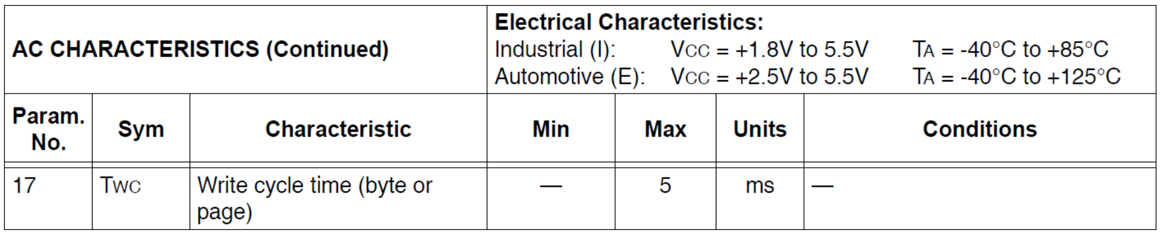
In the 24LC256 datasheet, the information is indicated at §5.0 “DEVICE ADDRESSING” in the Figure 5-2:



The address takes 2 bytes. The first is for “Address High Byte” and the second is for “Address Low Byte”.
So, we need to configure EEPROM_Conf.AddressType with EEPROM_ADDRESS_2Bytes.

3.1.4. Fill the EEPROM_Conf.PageWriteTime

In the 24LC256 datasheet, the information is indicated at §1.0 “ELECTRICAL CHARACTERISTICS” in the Table 1-2:



AC CHARACTERISTICS (Continued)			**Electrical Characteristics:** Industrial (I): VCC = +1.8V to 5.5V TA = -40°C to +85°C Automotive (E): VCC = +2.5V to 5.5V TA = -40°C to +125°C			
Param. No.	**Sym**	**Characteristic**	**Min**	**Max**	**Units**	**Conditions**
17	TWC	Write cycle time (byte or page)	—	5	ms	—

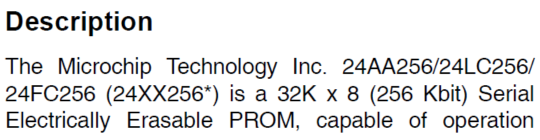
The parameter 17 indicate that the maximum write cycle time is 5ms. We need to always use the maximum time.
So, we configure EEPROM_Conf.PageWriteTime with 5.

3.1.5. Fill the EEPROM_Conf.PageSize

In the 24LC256 datasheet, the information is indicated in the Features of the device or the Description of the device in the first page of the datasheet: • 64-byte Page Write mode available
So, we configure EEPROM_Conf.PageSize with 64

3.1.6. Fill the EEPROM_Conf.ArrayByteSize

In the 24LC256 datasheet, the information is indicated in the Description of the device in the first page of the datasheet:



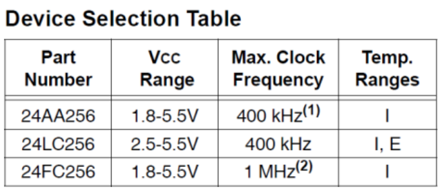
Description

The Microchip Technology Inc. 24AA256/24LC256/24FC256 (24XX256*) is a 32K x 8 (256 Kbit) Serial Electrically Erasable PROM, capable of operation

It is indicated 32K x 8 (256 Kbit). This memory is $256 \times 1024 = 262144\ bits \rightarrow 256144/8 = 32768\ bytes$. The memory organization is $32768/64 = 512$ page of 64 bytes
So, we configure EEPROM_Conf.ArrayByteSize with 32768.

3.1.7. Fill the EEPROM_Conf.MaxI2CclockSpeed

In the 24LC256 datasheet, the information is indicated in the Device Selection Table of the device in the first page of the datasheet:



Device Selection Table

Part Number	Vcc Range	Max. Clock Frequency	Temp. Ranges
24AA256	1.8-5.5V	400 kHz(1)	I
24LC256	2.5-5.5V	400 kHz	I, E
24FC256	1.8-5.5V	1 MHz(2)	I

It is indicated that the 24LC256 has an I2C Maximum Clock Frequency of 400kHz with no restriction of voltage.
So, we configure EEPROM_Conf.MaxI2CclockSpeed with 400000.

3.2. Example of how to fill an EEPROM_Conf of a device with A0, A1 and A2 pins that can be address pin

The following example was performed on a AT24C08A device.

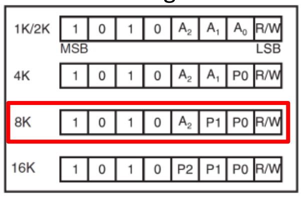
```
const EEPROM_Conf AT24C08A_Conf =
{
  .ChipAddress   = 0xA0,                          // Chip base address
  .ChipSelect    = EEPROM_CHIP_ADDRESS_A2,        // Device have only the A2 pins
  .AddressType   = EEPROM_ADDRESS_1Byte_plus_A1A0, // Memory address takes 1 bytes + A0 and A1 pin function
  .PageWriteTime = 5,                             // 5ms of page write
  .PageSize      = 16,                            // Page size is 16 bytes
  .ArrayByteSize = 64/*Pages*/ *16,               // Memory size is 64*16 = 1024 bytes
  .MaxI2CclockSpeed = 400000,                     // Max I2C SCL speed is 400kHz
};
```

The AT24C08A device is a "8K SERIAL EEPROM" as say in the datasheet:

> **AT24C08A, 8K SERIAL EEPROM:** Internally organized with 64 pages of 16 bytes each, the 8K requires a 10-bit data word address for random word addressing.

3.2.1. Fill the EEPROM_Conf.ChipAddress

In the AT24C08A datasheet, the information is indicated in the Figure 7 "Device Address":



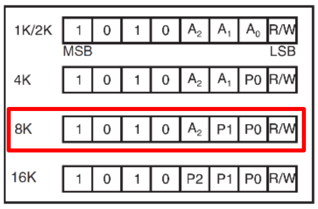
1K/2K	1	0	1	0	A_2	A_1	A_0	R/W
	MSB							LSB
4K	1	0	1	0	A_2	A_1	P0	R/W
8K	1	0	1	0	A_2	P1	P0	R/W
16K	1	0	1	0	P2	P1	P0	R/W

We are looking for the fixed information for the 8K device, without A2, P1, and P0 bits. The $\mathrm{R}/\overline{\mathrm{W}}$ bit need to be ignored too. These bits have to be set to '0'. So, we configure EEPROM_Conf.ChipAddress with 0b10100000 or 0xA0.

3.2.2. Fill the EEPROM_Conf.ChipSelect

Here we are looking for which chip select bits are present on this device. In the AT24C08A datasheet, the information is indicated at "PIN DESCRIPTION", at "DEVICE ADDRESSING" and in the Figure 7 "Device Address":

> The AT24C08A only uses the A2 input for hardwire addressing and a total of two 8K devices may be addressed on a single bus system. The A0 and A1 pins are no connects and can be connected to ground.
>
> The 8K EEPROM only uses the A2 device address bit with the next 2 bits being for memory page addressing. The A2 bit must compare to its corresponding hard-wired input pin. The A1 and A0 pins are no connect.



1K/2K	1	0	1	0	A_2	A_1	A_0	R/W
	MSB							LSB
4K	1	0	1	0	A_2	A_1	P0	R/W
8K	1	0	1	0	A_2	P1	P0	R/W
16K	1	0	1	0	P2	P1	P0	R/W

The device has only A2 as user configurable chip select. It is configurable at '0' or '1'.
So, we configure EEPROM_Conf.ChipSelect with EEPROM_CHIP_ADDRESS_A2.

3.2.3. Fill the EEPROM_Conf.AddressType

With the information said in §3.2.2, the device uses A0 as 9th bit, and A1 as 10th bit of the addressing.
So, we need to configure EEPROM_Conf.AddressType with EEPROM_ADDRESS_1Byte_plus_A1A0.

3.2.4. Fill the EEPROM_Conf.PageWriteTime

In the AT24C08A datasheet, the information is indicated in Table 5 "AC Characteristics":

Symbol	Parameter	1.8-volt		2.7, 5.0-volt		Units
		Min	Max	Min	Max	
t_{WR}	Write Cycle Time		5		5	ms

The parameter t_{WR} indicate that the maximum write cycle time is 5ms. We need to always use the maximum time.
So, we configure EEPROM_Conf.PageWriteTime with 5.

3.2.5. Fill the EEPROM_Conf.PageSize

In the AT24C08A datasheet, the information is indicated in the Memory Organization of the device:

> **AT24C08A, 8K SERIAL EEPROM:** Internally organized with 64 pages of 16 bytes each, the 8K requires a 10-bit data word address for random word addressing.

So, we configure EEPROM_Conf.PageSize with 16

3.2.6. Fill the EEPROM_Conf.ArrayByteSize

With the information said in §3.2.5:
It is indicated that the memory organization is 64 pages of 16 bytes = 1024 bytes
So, we configure EEPROM_Conf.ArrayByteSize with 1024.

3.2.7. Fill the EEPROM_Conf.MaxI2CclockSpeed

In the AT24C08A datasheet, the information is indicated in Table 5 “AC Characteristics”:

Symbol	Parameter	1.8-volt		2.7, 5.0-volt		Units
		Min	Max	Min	Max	
f_{SCL}	Clock Frequency, SCL		100		400	kHz

It is indicated that the AT24C08A has an I2C Maximum Clock Frequency of 400kHz with some restriction of voltage. If you use the device between 2.7V and 5.0V, then it is 400kHz. If you use the device at 1.8V then it is 100kHz.
So, we configure EEPROM_Conf.MaxI2CclockSpeed following the device voltage.

3.3. Example of how to fill an EEPROM_Conf of a EERAM device

The following example was performed on a 47L16 device.

```
const EEPROM_Conf EERAM47L16_Conf =
{
  .ChipAddress   = 0xA0,                    // Chip base address
  .ChipSelect    = EEPROM_CHIP_ADDRESS_A2A1, // Device have A2 and A1 pins
  .AddressType   = EEPROM_ADDRESS_2Bytes,    // Memory address takes 2 bytes
  .PageWriteTime = 25,                      // 25ms of page write
  .PageSize      = 2048,                    // Page size is 2048 bytes
  .ArrayByteSize = 2048,                    // Memory size is 2048 bytes
  .MaxI2CclockSpeed = 1000000,              // Max I2C SCL speed is 1MHz
};
```

3.3.1. Fill the EEPROM_Conf.ChipAddress

In the 47L16 datasheet, the information is indicated at §2.2 “Device Addressing” in the Figure 2-3:

Operation	Op Code	Chip Select	$R/\overline{W}$ Bit
SRAM Read	1010	A2 A1 0	1
SRAM Write	1010	A2 A1 0	0
Control Register Read	0011	A2 A1 0	1
Control Register Write	0011	A2 A1 0	0

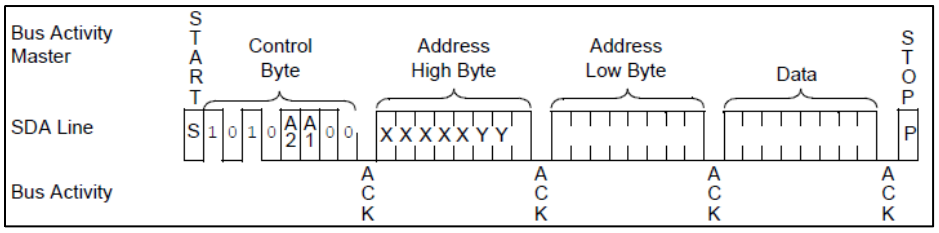
The memory Chip Address we are looking for is the SRAM. The Control Register is not what we are looking for.
We are looking for the fixed information called here “Op Code”, without Chip Select bits (A2 and A1). The $R/\overline{W}$ bit need to be ignored too. Take care that the normally A0 pin is not used and always at ‘0’, this have to be reported in the ChipAddress.
These bits have to be set to ‘0’. So, we configure EEPROM_Conf.ChipAddress with 0b10100000 or 0xA0.

3.3.2. Fill the EEPROM_Conf.ChipSelect

Here we are looking for which chip select bits are present on this device. In the 47L16 datasheet, the information is indicated at §2.2 “Device Addressing” in the Figure 2-3 (see §3.3.1):
The device has A1, and A2 as user configurable chip select. They are all configurable at ‘0’ or ‘1’.
So, we configure EEPROM_Conf.ChipSelect with EEPROM_CHIP_ADDRESS_A2A1.

3.3.3. Fill the EEPROM_Conf.AddressType

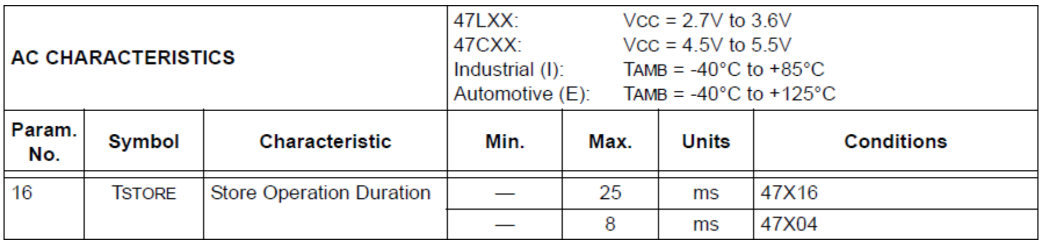
In the 47L16 datasheet, the information is indicated at §2.3 “SRAM Array” in the Figure 2-4:



The address takes 2 bytes. The first is for “Address High Byte” and the second is for “Address Low Byte”.
So, we need to configure EEPROM_Conf.AddressType with EEPROM_ADDRESS_2Bytes.

3.3.4. Fill the EEPROM_Conf.PageWriteTime

In the 47L16 datasheet, the information is indicated at §1.0 “ELECTRICAL CHARACTERISTICS” in the Table 1-2:



AC CHARACTERISTICS			47LXX: Vcc = 2.7V to 3.6V 47CXX: Vcc = 4.5V to 5.5V Industrial (I): TAMB = -40°C to +85°C Automotive (E): TAMB = -40°C to +125°C			
Param. No.	**Symbol**	**Characteristic**	**Min.**	**Max.**	**Units**	**Conditions**
16	TSTORE	Store Operation Duration	—	25	ms	47X16
			—	8	ms	47X04

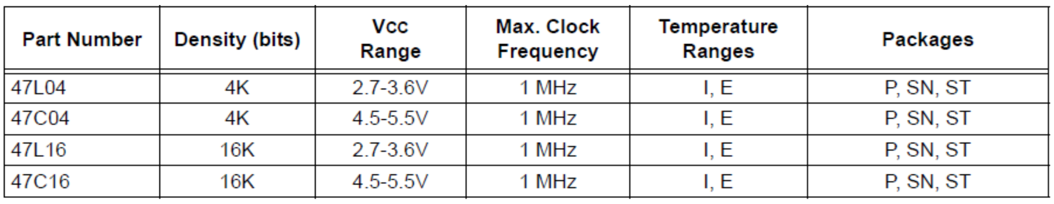
The parameter 16 indicate that the maximum write cycle time is 25ms for the 47X16. We need to always use the maximum time.
So, we configure EEPROM_Conf.PageWriteTime with 25. In reality, due to the operation of SRAM, it will take no time to store data. You can set this parameter to 0.

3.3.5. Fill the EEPROM_Conf.PageSize

In the 47L16 device, when the driver communicates with it, it always sees the SRAM, never the EEPROM. So, there is no page with this component, thus we configure EEPROM_Conf.PageSize with 2048 (see §3.3.6) which is the byte memory size of the SRAM. Never set ‘0’.

3.3.6. Fill the EEPROM_Conf.ArrayByteSize

In the 47L16 datasheet, the information is indicated in the “Device Selection Table” of the device in the first page of the datasheet:

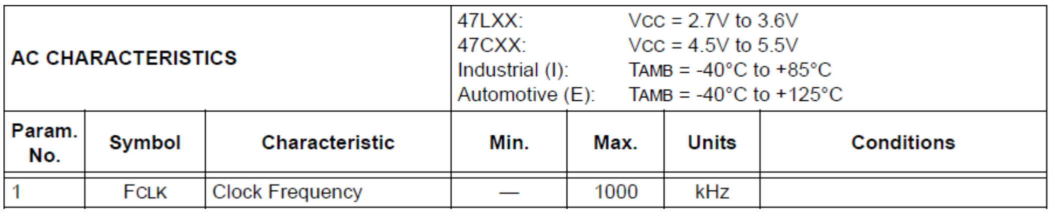


Part Number	Density (bits)	Vcc Range	Max. Clock Frequency	Temperature Ranges	Packages
47L04	4K	2.7-3.6V	1 MHz	I, E	P, SN, ST
47C04	4K	4.5-5.5V	1 MHz	I, E	P, SN, ST
47L16	16K	2.7-3.6V	1 MHz	I, E	P, SN, ST
47C16	16K	4.5-5.5V	1 MHz	I, E	P, SN, ST

It is indicated 16 Kbits. This memory is $16 \times 1024 = 16384\ bits \rightarrow 16384/8 = 2048\ bytes$.
So, we configure EEPROM_Conf.ArrayByteSize with 2048.

3.3.7. Fill the EEPROM_Conf.MaxI2CclockSpeed

In the 47L16 datasheet, the information is indicated in the Device Selection Table of the device in the first page of the datasheet:

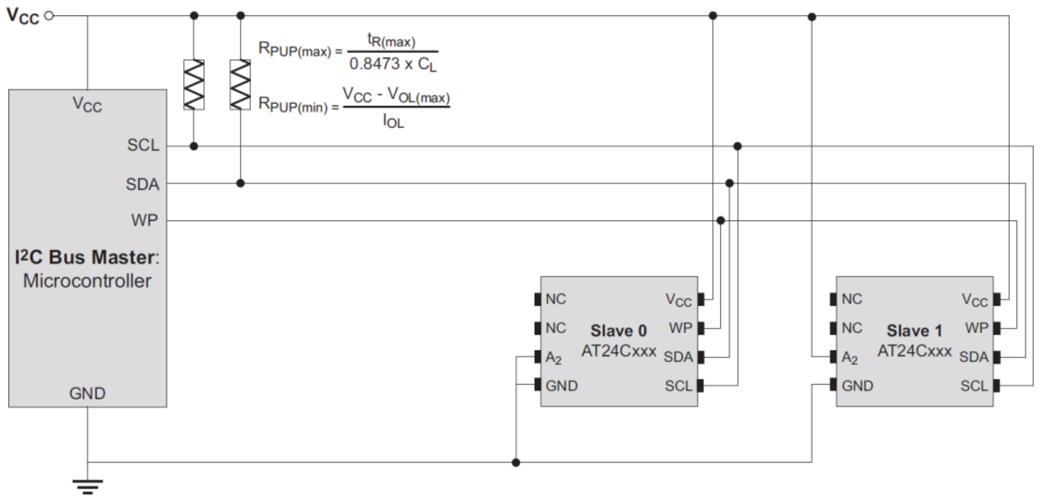


AC CHARACTERISTICS			47LXX: Vcc = 2.7V to 3.6V 47CXX: Vcc = 4.5V to 5.5V Industrial (I): TAMB = -40°C to +85°C Automotive (E): TAMB = -40°C to +125°C			
Param. No.	**Symbol**	**Characteristic**	**Min.**	**Max.**	**Units**	**Conditions**
1	FCLK	Clock Frequency	—	1000	kHz	

It is indicated that the 47L16 has an I2C Maximum Clock Frequency of 1MHz with no restriction of voltage.
So, we configure EEPROM_Conf.MaxI2CclockSpeed with 1000000.

3.4. Example of how to fill an EEPROM_Conf of dual EEPROM devices in series

The following example was performed with two AT24CM02 devices on the same bus. One with A2 at ‘0’ and the other with A2 as ‘1’. Schematic of the connection:



3.4.1. One device configuration

Here is the configuration of one AT24CM02 device (see §3.2 to understand how to configure a device):

```
const EEPROM_Conf AT24CM02_Conf =
{
  .ChipAddress    = 0xA0,                        // Chip base address
  .ChipSelect     = EEPROM_CHIP_ADDRESS_A2,      // Device have only the A2 pins
  .AddressType    = EEPROM_ADDRESS_2Byte_plus_A1A0, // Memory address takes 2 bytes + A0 and A1 pin function
  .PageWriteTime  = 10,                          // 10ms of page write
  .PageSize       = 256,                         // Page size is 256 bytes
  .ArrayByteSize  = 1024/*Pages*/ *256,          // Memory size is 1024*256 = 262144 bytes
  .MaxI2CclockSpeed = 1000000,                   // Max I2C SCL speed is 1MHz
};
```

3.4.2. Two devices configuration to see EEPROM as 1 bank of memory

Here is the configuration of two AT24CM02 device as 1 memory bank:

```
const EEPROM_Conf DualAT24CM02_Conf =
{
  .ChipAddress    = 0xA0,                          // Chip base address
  .ChipSelect     = EEPROM_NO_CHIP_ADDRESS_SELECT, // The virtual device have no chip address left
  .AddressType    = EEPROM_ADDRESS_2Byte_plus_A2A1A0, // Memory address takes 2 bytes + A0, A1 and A2 pin function
  .PageWriteTime  = 10,                            // 10ms of page write (remain the same)
  .PageSize       = 256,                           // Page size is 256 bytes (remain the same)
  .ArrayByteSize  = 2*1024/*Pages*/ *256,          // Memory size is 2*1024*256 = 524288 bytes
  .MaxI2CclockSpeed = 1000000,                     // Max I2C SCL speed is 1MHz (remain the same)
};
```

The EEPROM_Conf.ChipAddress, EEPROM_Conf.PageWriteTime, EEPROM_Conf.PageSize and EEPROM_Conf.MaxI2CclockSpeed are the same and are not changed.

We will use the A2 pin as the 19th address pin, the “virtual device” will have no chip address pin left so, EEPROM_Conf.ChipSelect is set to EEPROM_NO_CHIP_ADDRESS_SELECT, and EEPROM_Conf.AddressType with EEPROM_ADDRESS_2Byte_plus_A2A1A0.

With two same devices, the size of the new bank will double so, EEPROM_Conf.ArrayByteSize is $2 \times 262144 = 524288\ bytes$.

4. DEVICE CONFIGURATION

Each first parameter of a function is the device configuration. The purpose of this device configuration is to specify to the driver how and by which interface to communicate with the device selected. The device configuration type is EEPROM.

The EEPROM.fnI2C_Init, EEPROM.fnI2C_Transfer, and EEPROM.fnGetCurrentms are explained at §8.1.3 and an example at §10.

The EEPROM.Conf is the desired frequency of the I2C clock in Hertz. This allows the driver to verify the I2C speed.

The EEPROM.I2C_ClockSpeed is the desired frequency of the I2C clock in Hertz. This allows the driver to verify the I2C speed.

The EEPROM.UserDriverData is a generic pointer to what the user need. It can be used as context or bringing information for the interface functions for example. This variable is never touch by the driver.

The EEPROM.AddrA2A1A0 is the chip address pin configuration of A2, A1, and A0 pins.

4.1. Example

Example of driver configuration in a .c file:

```
struct EEPROM EEPROM24C08A_V71 =
{
  .UserDriverData  = NULL,
  //--- EEPROM configuration ---
  .Conf            = &AT24C08A_Conf,
  //--- Driver configuration ---
  .I2C_ClockSpeed  = 400000, // 400kHz
  //--- Interface driver call functions ---
  .InterfaceDevice = TWIHS0, // Here this point to the address memory of the peripheral TWIHS0
  .fnI2C_Init      = HardI2C_InterfaceInit_V71,
  .fnI2C_Transfer  = HardI2C_Tranfert_V71,
  //--- Time call function ---
  .fnGetCurrentms  = GetCurrentms_V71,
  //--- Interface clocks ---
  .AddrA2A1A0      = EEPROM_ADDR(0, 0, 1), // Here A2 = '0', A1 = '0', and A0 = '1'
};
```

Example of driver configuration in a .h file:

```
extern struct EEPROM EEPROM24C08A_V71;
#define EEPROM_24C08A  &EEPROM24C08A_V71
```

5. DEVICE INITIALIZATION

After configuring the device, you need to apply its configuration. To do this, it is pretty simple, first you need to call the Init_EEPROM() function with the EEPROM struct. If all went well, the function would return ERR_OK.

After that, the device is ready to use.

5.1. Example

Example of device initialization:

```
eERRORRESULT Error = Init_EEPROM(EEPROM_24C08A);
if (Error == ERR_OK)
{
  LOGTRACE("Device detected: 24C08A [1kB EEPROM (64x16)]");
}
```

This example is based on the example of configuration showed in §4.1.

6. READ DATA FROM THE EEPROM

To read data from the EEPROM, just use the EEPROM_ReadData() function. The address can be any byte in the memory area, even inside a page. The size of data to read can be any value, even the full memory size.
The only limitation is that $Address + Size \leq Size\ EEPROM$.

6.1. Example

Example of how to read data from the EEPROM.

```
uint32_t Address = 0x0000; // Address for the example
uint8_t ReadBuffer[16];    // Buffer for the example, it can be of any size
eERRORRESULT Error = EEPROM_ReadData(EEPROM_24C08A, Address, &ReadBuffer[0], sizeof(ReadBuffer));
// Test Error, Error should be ERR_OK if the read is complete without errors
// Do stuff
```

7. WRITE DATA TO THE EEPROM

To write data to the EEPROM, just use the EEPROM_WriteData() function. The address can be any byte in the memory area, even inside a page. The size of data to read can be any value, even the full memory size.
The only limitation is that $Address + Size \leq Size\ EEPROM$.

7.1. Example

Example of how to write data to the EEPROM.

```
uint32_t Address = 0x0000; // Address for the example
uint8_t WriteBuffer[16];   // Buffer for the example, it can be of any size
// Fill buffer with what you want
eERRORRESULT Error = EEPROM_WriteData(EEPROM_24C08A, Address, &WriteBuffer[0], sizeof(WriteBuffer));
// Test Error, Error should be ERR_OK if the write is complete without errors
// Do stuff
```

8. CONFIGURATION STRUCTURES

8.1. Device object structure

The EEPROM device object structure contains all information that is mandatory to work with a device. It is always the first parameter of each functions of the driver.

Source code:

```
typedef struct EEPROM EEPROM;

struct EEPROM
{
  void *UserDriverData;

  //--- EEPROM configuration ---
  const EEPROM_Conf *Conf;

  //--- Interface clocks ---
  uint32_t I2C_ClockSpeed;

  //--- Interface driver call functions ---
  void *InterfaceDevice;
  EEPROM_I2CInit_Func fnI2C_Init;
  EEPROM_I2CTranfert_Func fnI2C_Transfer;

  //--- Time call function ---
  GetCurrentms_Func fnGetCurrentms;

  //--- Device address ---
  uint8_t AddrA2A1A0;
};
```

8.1.1. Data fields

void ***UserDriverData**

Optional, can be used to store driver data related to the project and the device or NULL. This field is not used or modified by the driver.

const EEPROM_Conf ***Conf**

This is the EEPROM configuration, this parameter is mandatory.

Type

struct *EEPROM_Conf*

uint32_t **I2C_ClockSpeed**

Clock frequency of the I2C interface in Hertz.

void ***InterfaceDevice**

This is the pointer that will be in the first parameter of all interface call functions.

EEPROM_I2CInit_Func **fnI2C_Init**

This function will be called at driver initialization to configure the I2C interface driver.

Type

typedef *eERRORRESULT (*EEPROM_I2CInit_Func)(void *, const uint32_t)*

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the function Init_EEPROM().

EEPROM_I2CTranfert_Func **fnI2C_Transfer**

This function will be called when the driver needs to transfer data over the I2C communication with the device.

Type

typedef *eERRORRESULT (*EEPROM_I2CTranfert_Func)(void *, const uint8_t, uint8_t *, size_t, bool, bool)*

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned when using a function that require to communicate with the device.

GetCurrentms_Func **fnGetCurrentms**

This function will be called when the driver needs to get current millisecond. Some functions need a timeout, without, they can be stuck forever.

Type

typedef *uint32_t (*GetCurrentms_Func)(void)*

Initial value / default

This function has to point to a function else an ERR__PARAMETER_ERROR is returned by the functions EEPROM_ReadData() or EEPROM_WriteData().

uint8_t **AddrA2A1A0**

Device configurable address A2, A1, and A0. You can use the macro EEPROM_ADDR() to help filling this parameter. Only these 3 lower bits are used: ....210_ where 2 is A2, 1 is A1, 0 is A0. '.' and '_' are fixed by device. If the device has a fixed A2, A1, and/or A0 then set them in the macro EEPROM_ADDR().

8.1.2. Defines

#define ***EEPROM_ADDR(A2, A1, A0)***

Generate the EEPROM chip configurable address following the state of A0, A1, and A2. You shall set '1' (when corresponding pin is connected to +V) or '0' (when corresponding pin is connected to Ground) on each parameter. If the device has a fixed A2, A1, and/or A0 then set them in this macro.

8.1.3. Driver interface handle functions

```
eERRORRESULT (*EEPROM_I2CInit_Func)(
        void *pIntDev,
        const uint32_t sclFreq)
```

Function for interface driver initialization of the EEPROM. This function will be called at driver initialization to configure the interface driver.

Parameters

Input	*pIntDev	Is the EEPROM.InterfaceDevice of the device that call the interface initialization
Input	sclFreq	Is the SCL frequency in Hz to set at the interface initialization

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user. It is recommended, during the implement of the pointer interface function, to return only ERR__I2C_NACK, ERR__I2C_NACK_DATA, ERR__I2C_PARAMETER_ERROR, ERR__I2C_COMM_ERROR, ERR__I2C_CONFIG_ERROR, ERR__I2C_TIMEOUT, ERR__I2C_DEVICE_NOT_READY, ERR__I2C_INVALID_ADDRESS, ERR__I2C_INVALID_COMMAND, or ERR__I2C_FREQUENCY_ERROR when there is an error and ERR_OK when all went fine.

```
eERRORRESULT (*EEPROM_I2CTranfert_Func)(
        void *pIntDev,
        const uint8_t deviceAddress,
        uint8_t *data,
        size_t byteCount,
        bool start,
        bool stop)
```

Function for interface transfer of the EEPROM. This function will be called when the driver needs to transfer data over the I2C communication with the device. Can be a read of data or a transmit of data. It also indicates if it needs a start and/or a stop.

Parameters

Input	*pIntDev	Is the EEPROM.InterfaceDevice of the device that call the I2C transfer
Input	deviceAddress	Is the device address on the bus (8-bits only). The LSB bit indicate if it is a I2C Read (bit at '1') or a I2C Write (bit at '0')
In/Out	*data	Is a pointer to memory data to write in case of I2C Write, or where the data received will be stored in case of I2C Read (can be NULL if no data transfer other than chip address)
Input	byteCount	Is the byte count to write over the I2C bus or the count of byte to read over the bus
Input	start	Indicate if the transfer needs a start (in case of a new transfer) or restart (if the previous transfer has not been stopped)
Input	stop	Indicate if the transfer needs a stop after the last byte sent

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user. It is recommended, during the implement of the pointer interface function, to return only ERR__I2C_NACK, ERR__I2C_NACK_DATA, ERR__I2C_PARAMETER_ERROR, ERR__I2C_COMM_ERROR, ERR__I2C_CONFIG_ERROR, ERR__I2C_TIMEOUT, ERR__I2C_DEVICE_NOT_READY, ERR__I2C_INVALID_ADDRESS, ERR__I2C_INVALID_COMMAND, or ERR__I2C_FREQUENCY_ERROR when there is an error and ERR_OK when all went fine.

```
uint32_t (*GetCurrentms_Func)(void)
```

Function that gives the current millisecond of the system to the driver. This function will be called when the driver needs to get current millisecond.

Return

Returns the current millisecond of the system

8.2. EEPROM configuration object structure

The EEPROM configuration object structure contains all information that is mandatory to work with the memory of the device.

Source code:

```
typedef struct EEPROM_Conf
{
  uint8_t ChipAddress;
  eEEPROM_ChipSelect ChipSelect;
  eEEPROM_AddressType AddressType;
  uint8_t PageWriteTime;
  uint16_t PageSize;
  uint32_t ArrayByteSize;
  uint32_t MaxI2CclockSpeed;
} EEPROM_Conf;
```

8.2.1. Data fields

uint8_t **ChipAddress**

This is the base chip address. Generally, it is 0xA0, this is the general EEPROM I2C address.

eEEPROM_ChipSelect **ChipSelect**

Indicate which chip select pins are used by the chip.

Type

enum *eEEPROM_ChipSelect*

eEEPROM_AddressType **AddressType**

Indicate the EEPROM address type.

Type

enum *eEEPROM_AddressType*

uint8_t **PageWriteTime**

Maximum time to write a page (for timeout) in millisecond.

uint16_t **PageSize**

This is the page size of the device memory in bytes.

uint32_t **ArrayByteSize**

This is the memory total size in bytes.

uint32_t **MaxI2CclockSpeed**

This is the maximum I2C SCL clock speed of the device in Hertz.

8.2.2. Enumerators

enum **eEEPROM_ChipSelect**

Enumerator of all possible user configurable chip address selection pin that the device can have.

Enumerator

EEPROM_NO_CHIP_ADDRESS_SELECT	**0x00**	No chip select address
EEPROM_CHIP_ADDRESS_A0	**0x01**	User configurable chip select use A0
EEPROM_CHIP_ADDRESS_A1	**0x02**	User configurable chip select use A1
EEPROM_CHIP_ADDRESS_A2	**0x04**	User configurable chip select use A2
EEPROM_CHIP_ADDRESS_A1A0	**0x03**	User configurable chip select use A1, and A0
EEPROM_CHIP_ADDRESS_A2A0	**0x05**	User configurable chip select use A2, and A0
EEPROM_CHIP_ADDRESS_A2A1	**0x06**	User configurable chip select use A2, and A1
EEPROM_CHIP_ADDRESS_A2A1A0	**0x07**	User configurable chip select use A2, A1, and A0

enum **eEEPROM_AddressType**

Enumerator of all possible address configuration type.

Enumerator

EEPROM_ADDRESS_1Byte	**0x01**	EEPROM Address is 8-bits: S (1010$A_2A_1A_0$_) (xxxxxxxx)
EEPROM_ADDRESS_1Byte_plus_A0	**0x21**	EEPROM Address is 9-bits: S (1010A_2A_1x _) (xxxxxxxx)
EEPROM_ADDRESS_1Byte_plus_A1A0	**0x61**	EEPROM Address is 10-bits: S (1010A_2x x _) (xxxxxxxx)
EEPROM_ADDRESS_1Byte_plus_A2A1A0	**0xE1**	EEPROM Address is 11-bits: S (1010x x x _) (xxxxxxxx)
EEPROM_ADDRESS_2Bytes	**0x02**	EEPROM Address is 16-bits: S (1010$A_2A_1A_0$_) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_2Byte_plus_A0	**0x22**	EEPROM Address is 17-bits: S (1010A_2A_1x _) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_2Byte_plus_A1A0	**0x62**	EEPROM Address is 18-bits: S (1010A_2x x _) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_2Byte_plus_A2A1A0	**0xE2**	EEPROM Address is 19-bits: S (1010x x x _) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_3Bytes	**0x03**	EEPROM Address is 24-bits: S (1010$A_2A_1A_0$_) (xxxxxxxx) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_3Byte_plus_A0	**0x23**	EEPROM Address is 25-bits: S (1010A_2A_1x _) (xxxxxxxx) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_3Byte_plus_A1A0	**0x63**	EEPROM Address is 26-bits: S (1010A_2x x _) (xxxxxxxx) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_3Byte_plus_A2A1A0	**0xE3**	EEPROM Address is 27-bits: S (1010x x x _) (xxxxxxxx) (xxxxxxxx) (xxxxxxxx)
EEPROM_ADDRESS_4Bytes	**0x04**	EEPROM Address is 32-bits: S (1010$A_2A_1A_0$_) (xxxxxxxx) (xxxxxxxx) (xxxxxxxx) (xxxxxxxx)

8.3. Function’s return error enumerator

`enum eERRORRESULT`

There is only one error code at the same time returned by the functions. The only code that indicates that all went fine is ERR_OK.

Enumerator

ERR_OK	**0**	Succeeded
ERR__NO_DEVICE_DETECTED	**1**	No device detected
ERR__OUT_OF_RANGE	**2**	Value out of range
ERR__OUT_OF_MEMORY	**6**	Out of memory
ERR__DEVICE_TIMEOUT	**8**	Device timeout
ERR__PARAMETER_ERROR	**9**	Parameter error
ERR__NOT_READY	**10**	Device not ready
ERR__I2C_NACK	**210**	Received a I2C not acknowledge
ERR__I2C_NACK_DATA	**211**	Received a I2C not acknowledge while transferring data
ERR__I2C_PARAMETER_ERROR	**212**	I2C parameter error
ERR__I2C_COMM_ERROR	**213**	I2C communication error
ERR__I2C_CONFIG_ERROR	**214**	I2C configuration error
ERR__I2C_TIMEOUT	**215**	I2C communication timeout
ERR__I2C_DEVICE_NOT_READY	**216**	I2C device not ready
ERR__I2C_INVALID_ADDRESS	**217**	I2C invalid address
ERR__I2C_INVALID_COMMAND	**218**	I2C invalid command
ERR__I2C_FREQUENCY_ERROR	**219**	I2C frequency error

9. DRIVER’S FUNCTIONS

Here is the use of all functions related to the driver.

9.1. Initialization

```
eERRORRESULT Init_EEPROM(
        EEPROM *pComp)
```

This function initializes the EEPROM driver and call the initialization of the interface driver (I2C). It also checks the presence of the device.

Parameters

Input	*pComp	Is the pointed structure of the device to be initialized

Return

Returns an eERRORRESULT value enumerator. Below are some returned by the function itself but not errors returned by called functions.

- ERR__PARAMETER_ERROR when pComp or pComp->pConf is NULL, or Interface functions are NULL
- ERR__I2C_FREQUENCY_ERROR when pComp->I2C_ClockSpeed > pComp->Conf->MaxI2CclockSpeed
- ERR__NO_DEVICE_DETECTED when no communication with the device is possible

```
bool EEPROM_IsReady(EEPROM *pComp)
```

Poll the acknowledge from the EEPROM.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns 'true' if ready else 'false'

9.2. Read data

```
eERRORRESULT EEPROM_ReadData(
        EEPROM *pComp,
        uint32_t address,
        uint8_t* data,
        size_t size)
```

This function reads data from the EEPROM area of a EEPROM device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address to read (can be inside a page)
Output	*data	Is where the data will be stored
Input	size	Is the size of the data array to read

Return

Returns an eERRORRESULT value enumerator.

- ERR__PARAMETER_ERROR when pComp, pComp->Conf or data is NULL, or Interface functions are NULL, or Address is too high
- ERR__OUT_OF_MEMORY when you want to send too many data at the address specified or address is too high for the device
- ERR__DEVICE_TIMEOUT when it takes more than pComp->Conf->PageWriteTime+1 to write data to a page

9.3. Write data

```
eERRORRESULT EEPROM_WriteData(
       EEPROM *pComp,
       Uint32_t address,
       uint8_t* data,
       size_t size)
```

This function writes data to the EEPROM area of a EEPROM device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written (can be inside a page)
Input	*data	Is the data array to store
Input	size	Is the size of the data array to write

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp, pComp->Conf or data is NULL, or Interface functions are NULL, or Address is too high
- ERR__OUT_OF_MEMORY when you want to send too many data at the address specified or address is too high for the device
- ERR__DEVICE_TIMEOUT when it takes more than pComp->Conf->PageWriteTime+1 to write data to a page

10. EXAMPLE OF DRIVER INTERFACE HANDLE FUNCTIONS

This is an extract from the project for the SAMV71 Xplained Ultra board.

Example of driver interface handle functions in a .c file:

```
//=============================================================================
// Get millisecond
//=============================================================================
uint32_t GetCurrentms_V71(void)
{
  return msCount;
}


//**************************************************************************************************
//=============================================================================
// Hardware I2C driver interface configuration for the ATSAMV71
//=============================================================================
eERRORRESULT HardI2C_InterfaceInit_V71(void *pIntDev, const uint32_t sclFreq)
{
  if (pIntDev == NULL) return ERR__I2C_PARAMETER_ERROR;
  Twihs *I2C = (Twihs *)pIntDev;      // MCU specific: #define TWIHS0  ((Twihs*)0x40018000U) // (TWIHS0) Base Address

  //--- Reset ---
  if (SDA0_SOFT_Status == 0) // A device is stuck in communication and wait for a missing clocks
  {
    SDA0_SOFT_PIO_En;
    SDA0_SOFT_High;   // SDA = 1
    SDA0_SOFT_In;     // SDA in
    SCL0_SOFT_PIO_En;
    SCL0_SOFT_High;   // SCL = 1
    SCL0_SOFT_Out;    // SCL out
    size_t z = 9;                  // Clock up to 9 cycles. Here we force I2C SCL to clock until a device stuck in
                                   // communication respond
    while (SDA0_SOFT_Status == 0) // Look for SDA high in each cycle while SCL is high and then break
    {
      delay_us(1);
      SCL0_SOFT_Low;  // SCL = 0
      delay_us(1);
      SCL0_SOFT_High; // SCL = 1
      if (--z == 0) break;
    }
    ioport_set_pin_mode(TWIHS0_DATA_GPIO, TWIHS0_DATA_FLAGS); // Restore SDA pin function
    ioport_disable_pin(TWIHS0_DATA_GPIO);                     // Restore SDA pin function
    ioport_set_pin_mode(TWIHS0_CLK_GPIO, TWIHS0_CLK_FLAGS);   // Restore SCL pin function
    ioport_disable_pin(TWIHS0_CLK_GPIO);                      // Restore SCL pin function
  }

  //--- Configuration of the TWI interface ---
  twihs_options_t opt;
  opt.speed = sclFreq;
  if (twihs_master_setup(BOARD_AT24MAC_TWIHS, &opt) != TWIHS_SUCCESS) return ERR__I2C_CONFIG_ERROR;
  I2C->TWIHS_CR |= TWIHS_CR_STOP;

  //--- Clear receive buffer ---
  uint8_t Data = BOARD_AT24MAC_TWIHS_INSTANCE->TWIHS_RHR;
  (void)Data; // Unused data
  //--- clear registers ---
  Data = BOARD_AT24MAC_TWIHS_INSTANCE->TWIHS_SR;
  (void)Data; // Unused data

  I2Cconfigured = true;
  return ERR_OK;
}
```

```
//==============================================================================
// Hardware I2C - Transfer data through an I2C communication for the ATSAMV71
//==============================================================================
eERRORRESULT HardI2C_Tranfert_V71(void *pIntDev, const uint8_t deviceAddress, uint8_t *data, size_t byteCount, bool
start, bool stop)
{
  if (pIntDev == NULL) return ERR__SPI_PARAMETER_ERROR;
  if (data == NULL) return ERR__SPI_PARAMETER_ERROR;
  Twihs *I2C = (Twihs *)pIntDev;      // MCU specific: #define TWIHS0  ((Twihs*)0x40018000U) // (TWIHS0) Base Address
  size_t RemainingBytes = byteCount;
  uint32_t Status;
  uint32_t Timeout = TWIHS_TIMEOUT;
  bool DeviceWrite  = ((deviceAddress & 0x01) == 0);
  I2C->TWIHS_MMR  = 0;
  I2C->TWIHS_MMR  = TWIHS_MMR_DADR(deviceAddress >> 1) | (DeviceWrite ? 0 : TWIHS_MMR_MREAD);
  I2C->TWIHS_IADR = 0; // Not used
  I2C->TWIHS_CR   = TWIHS_CR_MSEN | TWIHS_CR_SVDIS;

  //--- Device polling ? ---
  if ((data == NULL) || (byteCount <= 0))                    // Device polling only
  { // Little hack because TWI of V71 does not support device polling without using SMBus
    I2C->TWIHS_MMR &= ~TWIHS_MMR_MREAD;                      // The SMBus of this device does not support quick
                                                             // read command (no Stop will be sent)
    I2C->TWIHS_CR |= TWIHS_CR_SMBEN + TWIHS_CR_PECDIS;       // Enable SMBus
    I2C->TWIHS_CR |= TWIHS_CR_STOP;                          // Send a stop
    I2C->TWIHS_CR |= TWIHS_CR_QUICK;                         // Start the polling with a quick command

    Timeout = TWIHS_TIMEOUT;
    while (true)                                             // Wait the polling to finish
    {
      Status = I2C->TWIHS_SR;
      if ((Status & TWIHS_SR_NACK) > 0) return ERR__I2C_NACK;
      if (!Timeout--) return ERR__I2C_TIMEOUT;               // Timeout ? return an error
      if ((Status & TWIHS_SR_TXCOMP) > 0) break;
    }
    return ERR_OK;
  }

  //--- Transfer data ---
  if (start) I2C->TWIHS_CR |= TWIHS_CR_START;                // Send a start if asked
  if (DeviceWrite) // Device write
  {
    while (true)
    {
      Status = I2C->TWIHS_SR;
      if ((Status & TWIHS_SR_NACK) > 0) return ERR__I2C_NACK_DATA;
      if (!Timeout--) return ERR__I2C_TIMEOUT;               // Timeout ? return an error
      if ((Status & TWIHS_SR_TXRDY) == 0) continue;
      Timeout = TWIHS_TIMEOUT;

      if (RemainingBytes == 0) break;                        // No data remaining to send, then break the loop
      I2C->TWIHS_THR = *data;                                // Send next data byte
      data++;
      RemainingBytes--;
    }
    if (stop) I2C->TWIHS_CR |= TWIHS_CR_STOP;                // Send a stop if asked
  }
  else // Device read
  {
    while (RemainingBytes > 0)
    {
      if ((RemainingBytes == 1) && stop)
        I2C->TWIHS_CR |= TWIHS_CR_STOP;                      // Last byte ? Send a stop if asked

      Timeout = TWIHS_TIMEOUT;
      while (true)                                           // Wait the polling to finish
      {
        Status = I2C->TWIHS_SR;
        if ((Status & TWIHS_SR_NACK) > 0)
        {
          if (RemainingBytes == byteCount)
               return ERR__I2C_NACK;
          else return ERR__I2C_NACK_DATA;
        }
        if (!Timeout--) return ERR__I2C_TIMEOUT;             // Timeout ? return an error
        if ((Status & TWIHS_SR_RXRDY) > 0) break;
      }

      *data = I2C->TWIHS_RHR;                                 // Get next data byte
      data++;
      RemainingBytes--;
    }
  }
  if (stop) while ((I2C->TWIHS_SR & TWIHS_SR_TXCOMP) == 0);    // Wait until both holding register and internal
                                                              // shifter are empty and STOP condition has been sent
  return ERR_OK;
}
```

Example of driver interface handle functions in a .h file:

```
/*! @brief Get millisecond
 *
 * This function will be called when the driver needs to get current millisecond
 */
uint32_t GetCurrentms_V71(void);


//*****************************************************************************************************************

/*! @brief Hardware I2C driver interface configuration for the ATSAMV71
 *
 * This function will be called at driver initialization to configure the interface driver soft I2C
 * @param[in] *pIntDev Is the EERAM47x16.InterfaceDevice of the device that call this function
 * @param[in] sclFreq Is the SCL frequency in Hz to set at the interface initialization
 * @return Returns an #eERRORRESULT value enum
 */
eERRORRESULT HardI2C_InterfaceInit_V71(void *pIntDev, const uint32_t sclFreq);


/*! @brief Hardware I2C - Transfer data through an I2C communication for the ATSAMV71
 *
 * This function will be called when the driver needs to transfer data over the I2C communication with the device
 * Can be read data of transmit data. It also indicate if it needs a start and/or a stop
 * @param[in] *pIntDev Is the Interface Device pointer of the device that call the I2C transfer
 * @param[in] deviceAddress Is the device address on the bus (8-bits only). The LSB bit indicate if it is a I2C Read
(bit at '1') or a I2C Write (bit at '0')
 * @param[in,out] *data Is a pointer to memory data to write in case of I2C Write, or where the data received will be
stored in case of I2C Read (can be NULL if no data transfer other than chip address)
 * @param[in] byteCount Is the byte count to write over the I2C bus or the count of byte to read over the bus
 * @param[in] start Indicate if the transfer needs a start (in case of a new transfer) or restart (if the previous
transfer have not been stopped)
 * @param[in] stop Indicate if the transfer needs a stop after the last byte sent
 * @return Returns an #eERRORRESULT value enum
 */
eERRORRESULT HardI2C_Tranfert_V71(void *pIntDev, const uint8_t deviceAddress, uint8_t *data, size_t byteCount, bool
start, bool stop);
```