# AN3108
# Application note

## STLM75 firmware library for the STM32F10x

## Introduction

This application note describes the firmware library implementing the STLM75 interface for the STM32F10xxx (STM32F101x, STM32F103x, STM32F105x, STM32F107x) microcontroller.

This library is a firmware package which contains a collection of routines, data structures and macros covering the features of the STLM75 temperature sensor device. The firmware library allows the STLM75 sensor to be used in the user application without the need for an in-depth study of STLM75 registers and $I^2C$ read/write operation steps. As a result, using the firmware library saves significant time which would otherwise be spent in coding, while reducing the application development and integration costs.

The firmware library source code is developed in 'Strict ANSI-C' (relaxed ANSI-C for the example application). Writing the whole library in 'Strict ANSI-C' makes it independent from the software toolchain. In addition, the firmware architecture is developed in separate layers and the HAL (hardware abstraction layer) makes it independent from the microcontroller used in the final application.

Even though the firmware library source code is developed in 'ANSI-C', the code architecture follows an OOP (object oriented programming) approach.

*Section 1* describes document and library rules.

*Section 2* highlights the features of the STLM75 sensor and explains its hardware interface with a master device microcontroller (STM32 in this case).

*Section 3* and *4* highlight the features of the firmware library and describe its architecture and its exported APIs (application programming interfaces) in detail.

*Section 5* contains an example application source code describing how to configure and use the library.

# Contents

# List of tables

# List of figures

# 1 Document and library rules

This document uses the conventions described in the sections below.

## 1.1 Acronyms

The following table lists the acronyms used in this document.

**Table 1. List of abbreviations**

| Acronym | Meaning |
|---|---|
| API | Application programming interface |
| HAL | Hardware abstraction layer |
| MCU | Microcontroller unit |
| $I^2C$ | Inter-integrated circuit |
| OOP | Object oriented programming |

# 2 STLM75 temperature sensor

## 2.1 Sensor introduction

The STLM75 is a high-precision digital CMOS temperature sensor IC with a sigma-delta temperature-to-digital converter and an $I^2C$-compatible serial digital interface. It is targeted at general applications such as personal computers, system thermal management, electronics equipment, and industrial controllers, and is packaged in the industry standard 8-lead TSSOP and SO8 packages. The device contains a band-gap temperature sensor and 9-bit ADC which monitor and digitize the temperature to a resolution up to 0.5 °C. The STLM75 is typically accurate to ±3°C - max over the full temperature measurement range of -55 °C to 125 °C with ±2 °C accuracy in the -25 °C to + 100 °C range (max). The STLM75 is factory-calibrated and requires no external components to measure temperature.

Refer to the STLM75 (Digital temperature sensor and thermal watchdog) datasheet for more information.

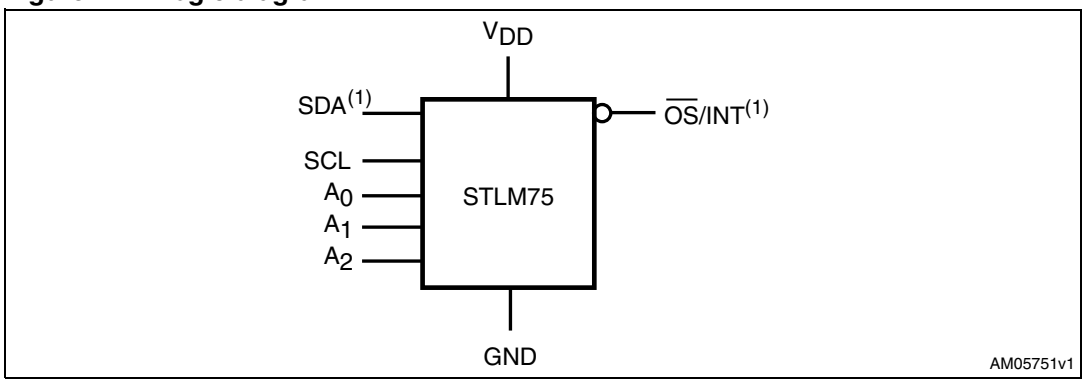
## 2.2 Interfacing the sensor with the microcontroller

The STLM75 has a simple 2-wire $I^2C$-compatible digital serial interface which allows the user to access the data in the temperature register at any time. It communicates via the serial interface with a master controller which operates at speeds up to 400 kHz. Three pins (A0, A1, and A2) are available for address selection, and enable the user to connect up to 8 devices on the same bus without address conflicts. In addition, the serial interface gives the user easy access to all STLM75 registers to customize the operation of the device.

*Figure 2* shows how the SMT32F10xxx microcontroller (master device) must be connected to the STLM75 device.

Refer to the STLM75 datasheet for more information.

**Figure 1. Logic diagram**

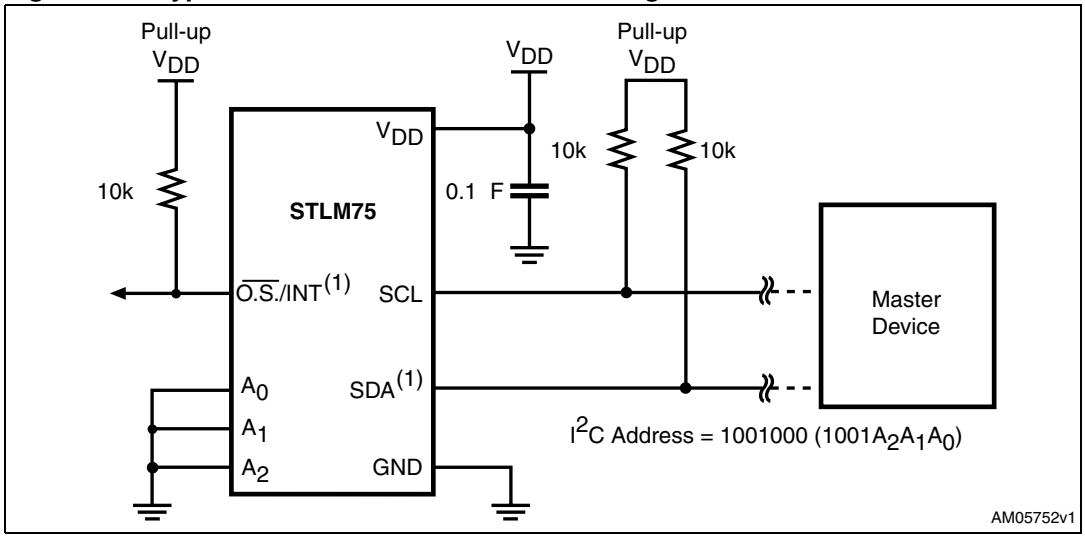



*Note: SDA and OS/INT are open drain.*

**Table 2. Signal names and pin descriptions**

| Pin | Sym | Type/direction | Description |
|---|---|---|---|
| 1 | SDA | Input/output | Serial data input/output |
| 2 | SCL | Input | Serial clock input |
| 3 | OS/INT | Output | Over-limit signal/interrupt alert output |
| 4 | GND | Supply ground | Ground |
| 5 | $A_2$ | Input | Address2 input |
| 6 | $A_1$ | Input | Address1 input |
| 7 | $A_0$ | Input | Address0 input |
| 8 | $V_{DD}$ | Supply power | Supply voltage (2.7 V to 5.5 V) |

**Figure 2. Typical 2-wire interface connection diagram**




*Note: SDA and OS/INT are open drain*

# 3 STLM75 library

## 3.1 Introduction

The STLM75 firmware library is fully developed in 'Strict ANSI-C' following an OOP approach. This means the final application using this library uses an instance of a temperature sensor object (TempSensor), and uses it according to its public methods and properties. The TempSensor is a structure containing public properties (data fields) and methods (functions pointers). The OOP encapsulation feature is assured.

The final application can create more than one TempSensor instance, and each instance can be matched with a different STLM75 temperature sensor assembled on the board. Therefore, the same library can be used to manage more temperature sensors simultaneously without communication and data conflict problems.

The library may be included in the final application as a library file (STLM75.a) and used as a black box through its exported public API, or can be included in the final application as source files (.c and .h), if the user wants to debug the library itself, or if it's necessary to change the HAL functions in order to port the library on an alternative microcontroller to the STM32F10xxx.

## 3.2 Library package

The library was developed using the IAR EWARM 5.20 and the related workspace/project files are included in the delivered package. As all the firmware is written in 'Strict ANSI-C', the library porting on another toolset doesn't require any change in the library.

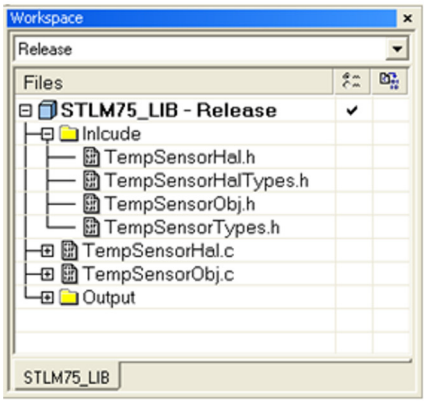
The library folder contains all the subdirectories and files that make up the core of the library:

- The included sub-folder contains the firmware library header files. They don't need to be modified by the user:
  - TempSensorObj.h: API layer file; this contains the Temperature Sensor Object structure description in terms of methods and properties; The API functions are declared in this file.
  - TempSensorTypes.h: API layer file; this contains all the defined types used by TempSensorObj.xxx files and related to the STLM75 temperature sensor.
  - TempSensorHal.h: HAL layer file; this contains all the Temperature Sensor functions declaration whose implementation depends on the MCU used (STM32 for this delivery). The final user should change these files in order to reuse this STLM75 library with other microcontrollers
  - TempSensorHalTypes.h HAL layer file; this contains all the Temperature Sensor types mapped on the used MCU library types (STM32 for this delivery). The final

user should change this type mapping in order to reuse this STLM75 library with other microcontrollers

- The source sub-folder contains the firmware library source files. They don't need to be modified by the user:
  - TempSensorObj.c: API layer file; this contains the exported public API (Application Programming Interface) and the related private internal functions. No direct reference to the Hardware and Micro firmware library occur in this file.
  - TempSensorHal.c: HAL layer file; this contains all the temperature sensor functions implementation whose source code depends on the MCU used (STM32 for this delivery). The final user should change these file in order to reuse this STLM75 library with other microcontrollers
  - The STM32_Include sub-folder contains the STM32F10xxxV2.0.3 firmware library included files. If the final user wants to use another microcontroller library version, replace this folder and check the HAL types and the microcontroller library function calls inside the HAL layer files (TempSensorHal.h, TempSensorHalTypes.h, TempSensorHal.c)
- EWARMv5 sub-folder contains the IAR EWARM 5.20 workspace and project files:
  - STLM75_Lib.eww: The IAR workspace file
  - STLM75_Lib.ewp: The IAR project file

**Figure 3. Firmware library project files**




## 3.3 Library architecture

The library architecture is devised and developed in two separate layers:

- API layer
- HAL layer

This layer architecture improves the code reusability splitting the application programming interface code (fully portable and reusable) from the hardware abstraction layer code (hardware dependent and written onto the STM32F10xxx libraries).

### 3.3.1 API layer

The application programming interface layer allows the final application to use the library as a black-box. The library firmware encapsulation feature and exported API allow full control of

the STLM75 temperature sensor without the need of an in-depth study of sensor registers and $I^2C$ read/write operation steps.

The API layer includes the following files:

- TempSensorObj.h
- TempSensorTypes.h
- TempSensorObj.c;

See *Section 4.1* for a more detailed description.

### 3.3.2 HAL layer

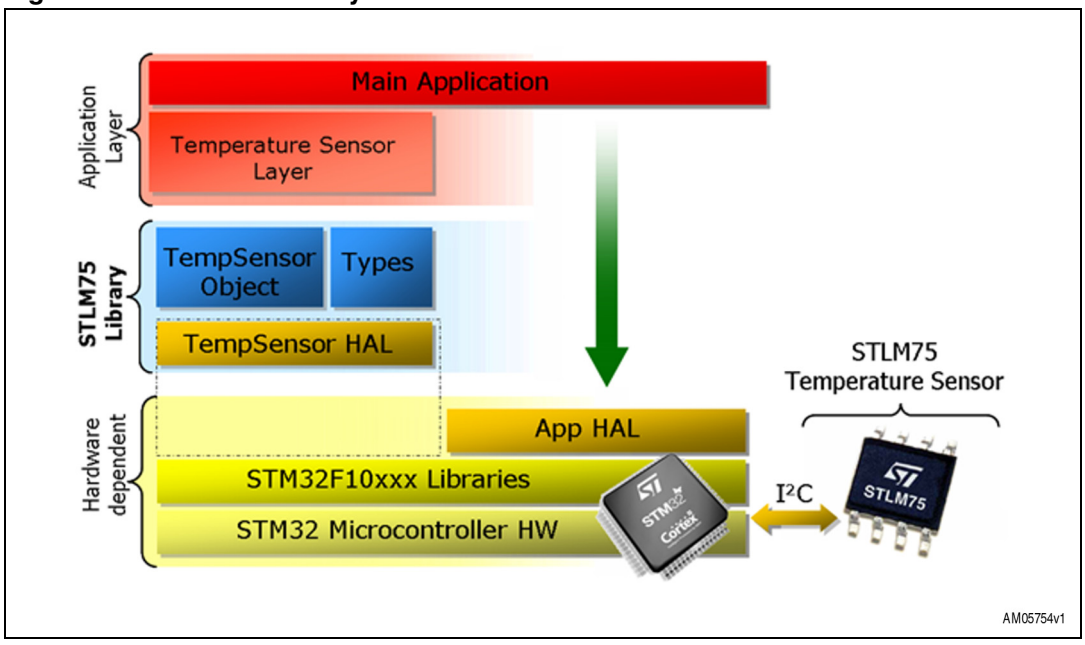
The hardware abstraction layer is directly built on the specific microcontroller firmware library and allows the built-upon layers, like the API layer, to implement its functions without the need of an in-depth study of the microcontroller used. This improves the library code reusability and guarantees easy portability on other microcontrollers.

The HAL layer includes the following files:

- TempSensorHal.h
- TempSensorHalTypes.h
- TempSensorHal.c

See *Section 4.2* for a more detailed description.

**Figure 4. Firmware library architecture**



# 4 STLM75 library firmware

This section describes the API and HAL layer implementation. Each library firmware function is described in detail. API and HAL layer functions are fully described. An example of how to use API functions is included. No example for HAL functions is provided because the final application should manage the STLM75 temperature sensor through the API layer functions only, without any direct access to the HAL functions.

The functions are described in the following format:

**Table 3. Function description format**

| Name | Description |
|---|---|
| Function name | The name of the peripheral function |
| Function prototype | Prototype declaration |
| Behavior description | Brief explanation of how the function is executed |
| Input parameter {x} | Description of the input parameters |
| Output parameter {x} | Description of the output parameters |
| Return value | Value returned by the function |
| Required preconditions | Requirements before calling the function |
| Called functions | Other library functions called |

## 4.1 API layer firmware overview

The application programming interface layer allows the final application to easily use the STLM75 temperature sensor. An OOP approach is used, making it possible for the application to create and use one or more instances of a TempSensor object.

The TempSensor structure is seen by the application as an object with encapsulate properties and methods. All read/write operations on the temperature sensors are executed through this object. It is an advanced structure containing:

- Properties as data fields
- Methods as functions pointers

In this way, each API function belongs to the related TempSensor object instance and more instances, and then more temperature sensors, can be managed without any conflicts.

In addition, the library exports two public API global functions in order to create/destroy a TempSensor structure instance:

- NewTempSensorObj function
- DelTempSensorObj function

### NewTempSensorObj API global function

*Table 4* describes the NewTempSensorObj function:

**Table 4. NewTempSensor API function**

| Name | Description |
|---|---|
| Function name | NewTempSensorObj |
| Function prototype | TempSensorType* NewTempSensorObj (void) |
| Behavior description | Create and initialize a new TempSensor object (a C structure) |
| Input parameter {x} | None |
| Output parameter {x} | None |
| Return value | The created object pointer or null if the object cannot be created |
| Required preconditions | None |
| Called functions | No API/HAL layers functions; |

### Example:

```
TempSensorType*  pObjTempSensor;
pObjTempSensor = NewTempSensorObj ();

/* Hardware Configuration: Signals: SCL, SDA, OS_INT pins */
TS_SignalsType TempSensorSignals;
...
/* Hardware Configuration: I2C pheripheral */
TS_I2C_SettingsType TempSensorI2C_Settings;
...
/* Initialize the temperature sensor according to previous Signals
and
   I2C_Settings */
pObjTempSensor->Init(pObjTempSensor, &TempSensorSignals,
                     &TempSensorI2C_Settings);
```

Once a TempSensor object instance is created using the NewTempSensorObj function, the TempSensor object itself provides all its features through its internal function pointers.

*Figure 2* shows the TempSensor object properties and methods in detail, which the final application can use to interact with the temperature sensor.

### DelTempSensorObj API global function

*Table 5* describes the DelTempSensorObj function:

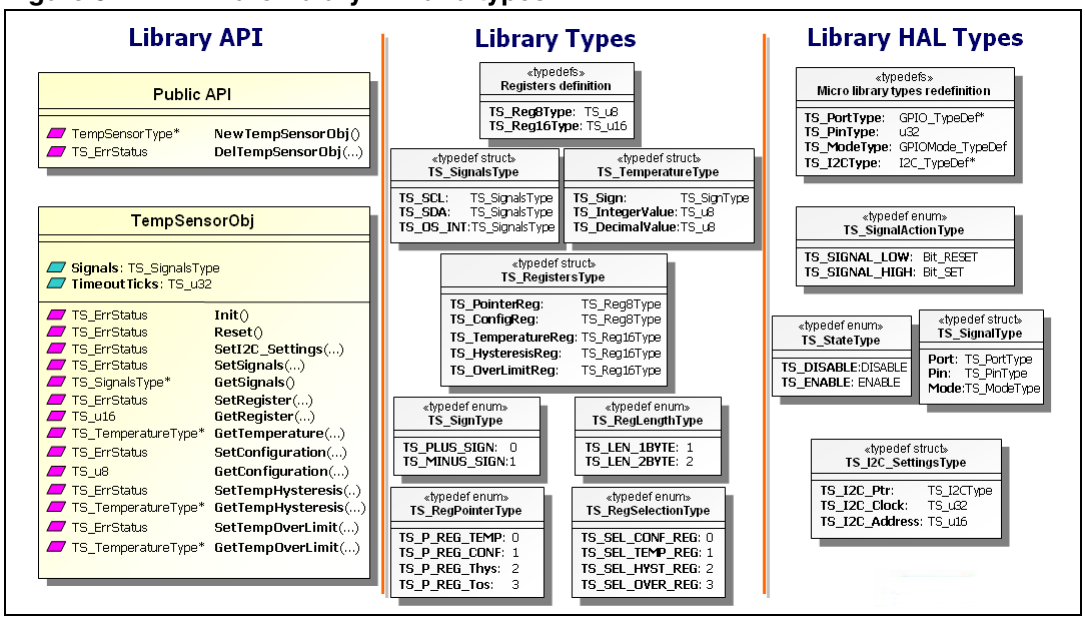
**Table 5. DelTempSensor API function**

| Name | Description |
|---|---|
| Function name | DelTempSensorObj |
| Function prototype | TS_ErrStatus DelTempSensorObj (TempSensorType** ppTempSensor) |
| Behavior description | Destroy the Object internal members and free the memory allocated when NewTempSensorObj function is called. |
| Input parameter {x} | ppTempSensor - Object pointer |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj must have been called before |
| Called functions | No API/HAL layers functions; |

### Example:

```
TempSensorType* pObjTempSensor;
TS_ErrStatus    errStatus;
...
errStatus = DelTempSensorObj(pTempSensorObj);
```

**Figure 5. Firmware library API and types**



### TempSensorObj API types/properties/functions

This section describes the TempSensor structure, all its exported API functions and all the defined types. See *Figure 2*.

Below you can find a description of the defined types used by the TempSensor object source code and a description of its properties and methods. Except for TempSensorType, all API layer types are defined in the TempSensorTypes.h file.

### TempSensorType type

The TempSensorType is defined in the following TempSensorObj.h file. The final TempSensor object is an instance of this defined structure.

The following described structure type can be considered as an OOP-class including private and public properties and public methods.

All the structure fields with a prefix name of 'm_' can be considered as private properties. This means these structure fields must not be directly accessed by the final application and Getxxx methods should be used instead. Clearly the final application could access these private members, but it should NEVER do so.

The final application must interact with this library using the following public properties and methods only, as described in *Figure 2* and in the following sections.

```
#define TEMP_SENSOR_OBJ        /* Temperature Sensor definition */ \
/*                    */ \
/* PRIVATE PROPERTIES */ \
TS_u8               m_Configuration;          \
TS_TemperatureType  m_Temperature;            \
TS_TemperatureType  m_TempHysteresis;         \
TS_TemperatureType  m_TempOverLimit;          \
TS_RegistersType    m_Registers;              \
TS_I2C_SettingsType m_I2C_Settings;           \
/*                    */ \
/* PUBLIC PROPERTIES */ \
TS_SignalsType      Signals;      /* The Micro/STLM75 signals */ \
TS_u32              TimeoutTicks; /* Timeout in ticks */ \
/*                    */ \
/* PUBLIC METHODS     */ \
TS_ErrStatus        (*Init)             (TempSensorType*);   \
TS_ErrStatus        (*Reset)            (TempSensorType*);   \
TS_ErrStatus        (*SetI2C_Settings)  (TempSensorType*,
TS_I2C_SettingsType* pI2C_Settings); \
TS_ErrStatus        (*SetSignals)       (TempSensorType*,
TS_SignalsType* pSignals); \
```

```
TS_SignalsType*     (*GetSignals)       (TempSensorType*); \
TS_ErrStatus        (*SetRegister)      (TempSensorType*,
TS_RegSelectionType, TS_u16);\
TS_u16              (*GetRegister)      (TempSensorType*,
TS_RegSelectionType eRegSelection); \
TS_TemperatureType* (*GetTemperature)   (TempSensorType*,
TS_Boolean bForceRegisterRead);     \
TS_ErrStatus        (*SetConfiguration) (TempSensorType*, TS_u8
uConfigValue);              \
TS_u8              (*GetConfiguration) (TempSensorType*, TS_Boolean
bForceRegisterRead);     \
TS_ErrStatus        (*SetTempHysteresis)(TempSensorType*,
TS_TemperatureType* pTemperature);  \
TS_TemperatureType* (*GetTempHysteresis)(TempSensorType*,
TS_Boolean bForceRegisterRead);     \
TS_ErrStatus        (*SetTempOverLimit) (TempSensorType*,
TS_TemperatureType* pTemperature);  \
TS_TemperatureType* (*GetTempOverLimit) (TempSensorType*,
TS_Boolean bForceRegisterRead);     \
/*                   */ \
/*  PUBLIC EVENTS    */ \
TS_ErrStatus        (*OnOverLimitIrq)   (TempSensorType*);


typedef struct TempSensor  TempSensorType;  /* Forward declaration
for circular typedefs */
struct TempSensor {
    TEMP_SENSOR_OBJ
};
```

### TS_Boolean type

```
/* Boolean types  */
typedef enum {
TS_FALSE = 0,
TS_TRUE  = !TS_FALSE
} TS_Boolean;
```

### TS_ErrStatus type

```
/* Error status types */
typedef enum {
TS_ERROR = 0,
```

```
TS_OK    = !TS_ERROR
} TS_ErrStatus;
```

### TS_Reg8Type and TS_Reg16Type types

```
/* Register types */
typedef TS_u8     TS_Reg8Type;
typedef TS_u16    TS_Reg16Type;
```

### TS_RegLengthType type

```
/* Register length type */
typedef enum{
    TS_REG_1BYTE_LEN   = 1,
    TS_REG_2BYTE_LEN
} TS_RegLengthType;
```

### TS_RegPointerType type

```
/* Command/Pointer register type */
typedef enum{
    TS_P_REG_TEMP =  0x00,   /* P1=0    P2=0 */
    TS_P_REG_CONF =  0x01,   /* P1=0    P2=1 */
    TS_P_REG_Thys =  0x02,   /* P1=1    P2=0 */
    TS_P_REG_Tos  =  0x03    /* P1=1    P2=1 */
} TS_RegPointerType;
```

### TS_RegSelection type

```
/* Register selection type */
typedef enum{
    TS_SEL_CONF_REG     = 0x00,
    TS_SEL_TEMP_REG     = 0x01,
    TS_SEL_HYST_REG     = 0x02,
    TS_SEL_OVER_REG     = 0x04
} TS_RegSelectionType;
```

### TS_SignType type

```
/* Temperature value sign type */
typedef enum{
    TS_PLUS_SIGN,
    TS_MINUS_SIGN
```

```
} TS_SignType;
```

### TS_RegistersType type

```
/* The STLM75 Registers */
typedef struct {
    TS_Reg8Type TS_PointerReg;     /* Command-Pointer register */
    TS_Reg8Type TS_ConfigReg;    /* Configuration register */
    TS_Reg16Type TS_TemperatureReg; /* Temperature register */
    TS_Reg16Type TS_HysteresisReg;  /* Hysteresis temp.register */
    TS_Reg16Type TS_OverLimitReg;   /* Over-limit temp.register */
} TS_RegistersType;
```

### TS_TemperatureType type

```
/* The STLM75 Temperature */
typedef struct {
    TS_SignType     TS_Sign;         /* Positive or negative sign */
    TS_u8           TS_IntegerValue; /* Integer part of the value */
    TS_u8           TS_DecimalValue; /* Decimal part of the value */
} TS_TemperatureType;
```

### TS_SignalsType type

```
/* The STLM75 Signals */
typedef struct {
    TS_SignalType     TS_SCL;
    TS_SignalType     TS_SDA;
    TS_SignalType     TS_OS_INT;
} TS_SignalsType;
```

### TempSensor:: Signals API property

The TempSensor structure exports the following public property:

```
TS_SignalsType  Signals;
```

The signals property contains the SCL, SDA, and OS_INT pin configuration. This property is redundant because the final application can use GetSignals and SetSignals API layer functions in order to manage signals property. The property has been left as public for utility scope only. It is recommended that the final application instead uses GetSignals and SetSignals functions to directly access signals public property.

### TempSensor::TimeoutTicks API property

The TempSensor structure exports the following public property:

```
TS_u32  TimeoutTicks;
```

The TimeoutTick property allows the final application to avoid application block in case of $I^2C$ communication problems. TimeoutTick is the number of g_TempSensorTick to wait during an $I^2C$ communication between the microcontroller and the STLM75 before considering that an error has occurred and resetting the $I^2C$ peripheral.

---

**Warning:** **TimeoutTicks must be set to >0 as following ONLY if g_TempSensorTick is defined as:** `extern volatile u32 g_TempSensorTick;` **in the final application source code, and it is incremented in a scheduler timer IRQ handler. If you don't want to use timeout feature, don't configure the TimeoutTicks property or set it to 0.**

---

The following is an example of TimeoutTicks property:

```
...
...
```

- File: main.c

```
int main(void)
{
...
/* configure SysTick timer in order to have a tick each ms */
...
/* configure the Temperature Sensor TimeoutTicks */
pObjTempSensor->TimeoutTicks = 2; /* 2ms:we have a g_TempSensorTick
                                     each millisecond */
...
}
```

- File: stm32f10x_it.c

```
extern volatile u32 g_TempSensorTick;
...
void SysTickHandler(void)
{
    // Increment Temperature Sensor tick for Timeout purpose
    g_TempSensorTick ++;
```

```
}
...
```

### TempSensor:: Init API function

*Table 6* describes the Init function of the TempSensor structure:

**Table 6. Init API function**

| Name | Description |
|---|---|
| Function name | Init |
| Function prototype | TS_ErrStatus Init (TempSensorType* pThis, TS_SignalsType* pSignals, TS_I2C_SettingsType* pI2C_Settings) |
| Behavior description | Initialize the TempSensor object. |
| Input parameter {x} | pThis - Object pointer;<br>pSignals - A Signal collection structure pointer;<br>pI2C_Settings - A I2C peripheral structure pointer; |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj, must have been called before. |
| Called functions | API function:<br>TS_ErrStatus SetSignals (TempSensorType* pThis, TS_SignalsType* pSignals);<br>TS_ErrStatus SetI2C_Settings (TempSensorType* pThis, TS_I2C_SettingsType* pI2C_Settings);<br>HAL function:<br>void TS_InitI2C_Peripheral(TS_I2C_SettingsType* pI2C_Settings); |

### Example:

```
TS_SignalsType TempSensorSignals;
TS_I2C_SettingsType TempSensorI2C_Settings;
...
/* Initialize the temperature sensor according to previous Signals
and I2C_Settings */
pObjTempSensor->Init(pObjTempSensor, &TempSensorSignals,
                     &TempSensorI2C_Settings);
```

### TempSensor:: Reset API function

*Table 7* describes the reset function of the TempSensor structure:

**Table 7. Reset API function**

| Name | Description |
|---|---|
| Function name | Reset |
| Function prototype | TS_ErrStatus Reset (TempSensorType* pThis) |
| Behavior description | Reset the Micro/Sensor $I^2C$ communication. |
| Input parameter {x} | pThis - object pointer |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>void TS_ResetI2C_Peripheral(TS_I2C_SettingsType* pI2C_Settings);<br>void TS_InitI2C_Peripheral(TS_I2C_SettingsType* pI2C_Settings); |

### Example:

```
...
/* Reset the I2C communication */
pObjTempSensor->Reset(pObjTempSensor);
```

### TempSensor:: SetI2C_Settings API function

*Table 8* describes the SetI2C_Settings function of the TempSensor structure.

This function is called by the TempSensor:Init function during the initialization phase, therefore it's not necessary to directly call it. The final application can specifically call this function in order to change the $I^2C$ peripheral settings after initialization.

**Table 8. SetI2C_Settings API function**

| Name | Description |
|---|---|
| Function name | SetI2C_Settings |
| Function prototype | TS_ErrStatus SetI2C_Settings (TempSensorType* pThis,<br>TS_I2C_SettingsType* pI2C_Settings) |
| Behavior description | Set the STLM75 I2C peripheral settings copying the right values from the passed parameter |
| Input parameter {x} | pThis - Object pointer;<br>pI2C_Settings - A I2C peripheral structure pointer; |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj function must have been called before. |
| Called functions | HAL function:<br>void TS_InitI2C_Peripheral(TS_I2C_SettingsType* pI2C_Settings); |

### Example:

```
TS_I2C_SettingsType TempSensorI2C_Settings;

/* Enable I2C2 clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C2, ENABLE);

/* I2C Peripheral settings initialization */
TempSensorI2C_Settings.TS_I2C_Ptr     = I2C2;
TempSensorI2C_Settings.TS_I2C_Clock   = 0x90;
TempSensorI2C_Settings.TS_I2C_Address = 400000;

pObjTempSensor->SetI2C_Settings(pObjTempSensor,
                                &TempSensorI2C_Settings);
```

### TempSensor:: SetSignals API function

*Table 9* describes the SetSignals function of the TempSensor structure.

This function is called by the TempSensor:Init function during the initialization phase, therefore it's not necessary to directly call it. The final application can specifically call this function in order to change the microcontroller signal (port, pin, mode) settings after initialization.

**Table 9. SetSignals**

| Name | Description |
|---|---|
| Function name | SetSignals |
| Function prototype | TS_ErrStatus SetSignals (TempSensorType* pThis, TS_SignalsType* pSignals) |
| Behavior description | Set the STLM75 signals (port and pin) using the right values from the passed parameter |
| Input parameter {x} | pThis - Object pointer;<br>pSignals - A Signal collection structure pointer; |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj function must have been called before. |
| Called functions | HAL function:<br>void TS_ConfigSignal(TS_SignalType* pSignal) |

### Example:

```
TS_SignalsType  TempSensorSignals;

/* Enable GPIOB clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

/* GPIO signals configuration */
/* Configure PB.10 as alternate function Open-Drain */
TempSensorSignals.TS_SCL.Port    = GPIOB;
TempSensorSignals.TS_SCL.Pin     = GPIO_Pin_10;
TempSensorSignals.TS_SCL.Mode    = GPIO_Mode_AF_OD;

/* Configure PB.11 as alternate function Open-Drain */
TempSensorSignals.TS_SDA.Port    = GPIOB;
TempSensorSignals.TS_SDA.Pin     = GPIO_Pin_11;
TempSensorSignals.TS_SDA.Mode    = GPIO_Mode_AF_OD;

/* Configure PB.12 as alternate function Open-Drain */
TempSensorSignals.TS_OS_INT.Port = GPIOB;
TempSensorSignals.TS_OS_INT.Pin  = GPIO_Pin_12;
TempSensorSignals.TS_OS_INT.Mode = GPIO_Mode_AF_PP;

pObjTempSensor->SetSignals(pObjTempSensor, &TempSensorSignals);
```

### TempSensor:: GetSignals API function

*Table 10* describes the GetSignals function of the TempSensor structure.

**Table 10. GetSignals**

| Name | Description |
|---|---|
| Function name | GetSignals |
| Function prototype | TS_SignalsType* GetSignals (TempSensorType* pThis) |
| Behavior description | Return the Signal collection structure pointer |
| Input parameter {x} | pThis - Object pointer |
| Output parameter {x} | None |
| Return value | The signals structure pointer |

**Table 10. GetSignals (continued)**

| Name | Description |
|---|---|
| Required preconditions | NewTempSensorObj function must have been called before. |
| Called functions | None |

```
TS_SignalsType*  pTempSensorSignals;


/* Get the Signals */
pTempSensorSignals = pObjTempSensor->GetSignals(pObjTempSensor);
```

### TempSensor:: SetRegister API function

*Table 11* describes the SetRegister function of the TempSensor structure.

This function can be considered as an advanced function for the final user who knows STLM75 internal registers. This function call can be avoided by calling one of the following specific functions:

```
SetConfiguration, SetTempHysteresis, SetTempOverLimit;
```

Note that calling the SetRegister advanced function instead of a specific one, internal TempSensor object data (m_Configuration, m_TempHysteresis, m_TempOverLimit) is not assigned.

**Table 11. SetRegister**

| Name | Description |
|---|---|
| Function name | SetRegister |
| Function prototype | TS_ErrStatus SetRegister (TempSensorType* pThis, TS_RegSelectionType eRegSelection, TS_u16 uRegValue) |
| Behavior description | Set the STLM75 register identified by eRegSelection type with the value passed as parameter |
| Input parameter {x} | pThis - Object pointer;<br>eRegSelection - The type associated to the register to be set;<br>uRegValue - The value to be assigned to the register. This is a 16-bit value, so for the 8-bit register, only the LSB must be considered |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillRegisterFromData(…); |

### Example:

```
...
pObjTempSensor->SetRegister(pObjTempSensor, TS_SEL_CONF_REG, 0);
```

### TempSensor:: GetRegister API function

*Table 12* describes the GetRegister function of the TempSensor structure.

This function can be considered as an advanced function for the final user who knows STLM75 internal registers. This function call can be avoided by calling one of the following specific functions:

```
GetConfiguration, GetTemperature, GetTempHysteresis,
GetTempOverLimit;
```

**Table 12. GetRegister**

| Name | Description |
|---|---|
| Function name | GetRegister |
| Function prototype | TS_u16 GetRegister (TempSensorType* pThis, TS_RegSelectionType eRegSelection) |
| Behavior description | Return a 16-bit register value. The right register is identified by eRegSelection type passed as parameter. For the 8-bit register, only the LSB must be considered in the return value. |
| Input parameter {x} | pThis - Object pointer;<br>eRegSelection - The type associated to the register to be set |
| Output parameter {x} | None |
| Return value | Register value if successful, TS_NULL_REG_VALUE otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillDataFromRegister (…); |

### Example:

```
TS_u16 uTemperatureRegisterValue;

uTemperatureRegisterValue = pObjTempSensor-
>GetRegister(pObjTempSensor, TS_SEL_TEMP_REG);
```

### TempSensor:: SetConfiguration API function

*Table 13* describes the SetConfiguration function of the TempSensor structure.

**Table 13. SetConfiguration**

| Name | Description |
|---|---|
| Function name | SetConfiguration |
| Function prototype | TS_ErrStatus SetConfiguration (TempSensorType* pThis, TS_u8 uConfigValue) |
| Behavior description | Set the 8-bit Configuration register value. |
| Input parameter {x} | pThis - Object pointer;<br>uConfigValue - The 8-bit value to write in the register |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillRegisterFromData(…); |

### Example:

```
u8 uTestConfigurationValue;
...
/* Write the following value for Configuration register: 0x00 */
uTestConfigurationValue = 0x00;
pObjTempSensor->SetConfiguration  (pObjTempSensor,
                                   uTestConfigurationValue);
```

### TempSensor:: GetConfiguration API function

*Table 14* describes the GetConfiguration function of the TempSensor structure.

**Table 14. GetConfiguration**

| Name | Description |
|---|---|
| Function name | GetConfiguration |
| Function prototype | TS_u8 GetConfiguration (TempSensorType* pThis, TS_Boolean bForceRegisterRead) |
| Behavior description | Return the 8-bit Configuration register value. |
| Input parameter {x} | pThis - Object pointer;<br>bForceRegisterRead - if TS_TRUE reads the value from the register otherwise the last read value returns |
| Output parameter {x} | None |

**Table 14. GetConfiguration (continued)**

| Name | Description |
|---|---|
| Return value | Configuration value if successful, TS_NULL_REG_VALUE otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillDataFromRegister (…); |

### Example:

```
u8 uTestConfigurationValue;


/* Get the configuration value forcing a STLM75 register reading */
uTestConfigurationValue = pObjTempSensor->GetConfiguration
                          (pObjTempSensor, TS_TRUE);
```

### TempSensor:: GetTemperature API function

*Table 15* describes the GetTemperature function of the TempSensor structure.

**Table 15. GetTemperature**

| Name | Description |
|---|---|
| Function name | GetTemperature |
| Function prototype | TS_TemperatureType* GetTemperature (TempSensorType* pThis, TS_Boolean bForceRegisterRead) |
| Behavior description | Return Temperature structure pointer. |
| Input parameter {x} | pThis - Object pointer;<br>bForceRegisterRead - if TS_TRUE reads the value from the register otherwise the last read value returns |
| Output parameter {x} | None |
| Return value | Temperature pointer if successful, TS_NULL_PTR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillDataFromRegister (…); |

### Example:

```
TS_TemperatureType* pStrTemperature;


/* Get the read temperature structure pointer forcing a STLM75
register reading */
```

```
pStrTemperature = pObjTempSensor->GetTemperature (pObjTempSensor,
                                                  TS_TRUE);
```

### TempSensor:: SetTempHysteresis API function

*Table 16* describes the SetTempHysteresis function of the TempSensor structure.

**Table 16. SetTempHysteresis**

| Name | Description |
|---|---|
| Function name | SetTempHysteresis |
| Function prototype | TS_ErrStatus SetTempHysteresis (TempSensorType* pThis, TS_TemperatureType* pTemperature) |
| Behavior description | Set the 16-bit Hysteresis temperature register value. |
| Input parameter {x} | pThis - Object pointer;<br>pTemperature - Hysteresis temperature structure pointer |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillRegisterFromData(…); |

### Example:

```
TS_TemperatureType strTestTemperature;
...
/* Write the following value for Hysteresis register: +75.5° C  */
strTestTemperature.TS_Sign         = TS_PLUS_SIGN;
strTestTemperature.TS_IntegerValue = 75;
strTestTemperature.TS_DecimalValue = 5;     /* 0 or 5 only */
pObjTempSensor->SetTempHysteresis (pObjTempSensor,
                                   &strTestTemperature);
```

### TempSensor:: GetTempHysteresis API function

*Table 17* describes the GetTempHysteresis function of the TempSensor structure.

**Table 17. GetTempHysteresis**

| Name | Description |
|---|---|
| Function name | GetTempHysteresis |
| Function prototype | TS_TemperatureType* GetTempHysteresis (TempSensorType* pThis, TS_Boolean bForceRegisterRead) |

**Table 17. GetTempHysteresis (continued)**

| Name | Description |
|---|---|
| Behavior description | Return the Hysteresis temperature structure pointer. |
| Input parameter {x} | pThis - Object pointer;<br>bForceRegisterRead - if TS_TRUE reads the value from the register otherwise the last read value returns |
| Output parameter {x} | None |
| Return value | Hysteresis temperature pointer if successful, TS_NULL_PTR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillDataFromRegister (…); |

### Example:

```
TS_TemperatureType* pStrTemperature;


/* Get the Hysteresis temperature structure pointer forcing a STLM75
   register reading */
pStrTemperature = pObjTempSensor->GetTempHysteresis
                  (pObjTempSensor, TS_TRUE);
```

### TempSensor:: SetTempOverLimit API function

*Table 18* describes the SetTempOverLimit function of the TempSensor structure.

**Table 18. SetTempOverLimit**

| Name | Description |
|---|---|
| Function name | SetTempOverLimit |
| Function prototype | TS_ErrStatus SetTempOverLimit (TempSensorType* pThis,<br>TS_TemperatureType* pTemperature) |
| Behavior description | Set the 16-bit Over-limit temperature register value. |
| Input parameter {x} | pThis - Object pointer;<br>pTemperature - Over-limit temperature structure pointer |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillRegisterFromData(…); |

### Example:

```
TS_TemperatureType strTestTemperature;
...
/* Write the following value for Over-Limit register: +75.5° C  */
strTestTemperature.TS_Sign         = TS_PLUS_SIGN;
strTestTemperature.TS_IntegerValue = 75;
strTestTemperature.TS_DecimalValue = 5;     /* 0 or 5 only */
pObjTempSensor->SetTempOverLimit (pObjTempSensor,
                                  &strTestTemperature);
```

### TempSensor:: GetTempOverLimit API function

*Table 19* describes the GetTempOverLimit function of the TempSensor structure.

**Table 19. GetTemperatureOverLimit**

| Name | Description |
|---|---|
| Function name | GetTempOverLimit |
| Function prototype | TS_TemperatureType* GetTempOverLimit (TempSensorType* pThis, TS_Boolean bForceRegisterRead) |
| Behavior description | Return the Over-limit temperature structure pointer. |
| Input parameter {x} | pThis - Object pointer;<br>bForceRegisterRead - if TS_TRUE reads the value from the register otherwise the last read value returns |
| Output parameter {x} | None |
| Return value | Over-limit temperature pointer if successful, TS_NULL_PTR otherwise |
| Required preconditions | NewTempSensorObj, TempSensor:: Init functions must have been called before. |
| Called functions | HAL function:<br>TS_ErrStatus TS_SetPointerRegister(…);<br>TS_ErrStatus TS_FillDataFromRegister (…); |

### Example:

```
TS_TemperatureType* pStrTemperature;


/* Get the Over-Limit temperature structure pointer forcing a STLM75
register reading */
pStrTemperature = pObjTempSensor->GetTempOverLimit (pObjTempSensor,
                                                    TS_TRUE);
```

## 4.2 HAL Layer firmware overview

This section describes the hardware abstraction layer function used by the upper API layer described in the previous section. All the library microcontroller hardware dependent functions and related defined types are described in this section. See *Figure 2*.

The final application should NEVER directly use these HAL functions, and it should manage the STLM75 temperature sensor through the API layer functions as described above.

### 4.2.1 HAL types

#### Standard types redefinition

```
/* Standard type redefinition in order
   to maintain code portability */
typedef signed long      TS_s32;
typedef signed short     TS_s16;
typedef signed char      TS_s8;
typedef unsigned long    TS_u32;
typedef unsigned short   TS_u16;
typedef unsigned char    TS_u8;
```

#### STM32 library types redefinition

```
/* Redefine micro specific types */
typedef GPIO_TypeDef*     TS_PortType;
typedef GPIOMode_TypeDef  TS_ModeType;
typedef u32               TS_PinType;
typedef I2C_TypeDef*      TS_I2CType;
```

#### TS_SignalActionType type

```
/* Signal state enumeration */
typedef enum
{ TS_SIGNAL_LOW   = Bit_RESET,
  TS_SIGNAL_HIGH  = Bit_SET
} TS_SignalActionType;
```

#### TS_StateType type

```
/* State type  */
typedef enum
{
  TS_DISABLE = DISABLE,
  TS_ENABLE  = ENABLE
```

```
} TS_StateType;
```

### TS_SignalType type

```
/* The STLM75 Single Signal type */
typedef struct {
    TS_PortType   Port;
    TS_PinType    Pin;
    TS_ModeType   Mode;
} TS_SignalType;
```

### TS_I2C_SettingsType type

```
/* The STLM75 I2C Peripheral settings */
typedef struct {
    TS_I2CType  TS_I2C_Ptr;
    TS_u32      TS_I2C_Clock;
    TS_u16      TS_I2C_Address;
} TS_I2C_SettingsType;
```

## 4.2.2 HAL functions

### TS_ConfigSignal HAL function

*Table 20* describes the TS_ConfigSignal function of the hardware abstraction layer.

**Table 20. TS_ConfigSignal**

| Name | Description |
|---|---|
| Function name | TS_ConfigSignal |
| Function prototype | void TS_ConfigSignal(TS_SignalType* pSignal) |
| Behavior description | Config the signal pin according to the passed parameters |
| Input parameter {x} | pSignal - Signal structure pointer |
| Output parameter {x} | None |
| Return value | None |
| Required preconditions | None |
| Called functions | None |

### TS_InitI2C_Peripheral HAL function

*Table 21* describes the TS_InitI2C_Peripheral function of the hardware abstraction layer.

**Table 21. TS_InitI2C_Peripheral**

| Name | Description |
|---|---|
| Function name | TS_InitI2C_Peripheral |
| Function prototype | void TS_InitI2C_Peripheral(TS_I2C_SettingsType* pI2C_Settings) |
| Behavior description | Initialize and configure the I2C peripheral used by the micro in order to manage the STLM75 temperature sensor device |
| Input parameter {x} | pI2C_Settings - I2C peripheral settings structure pointer |
| Output parameter {x} | None |
| Return value | None |
| Required preconditions | None |
| Called functions | None |

### TS_ResetI2C_Peripheral HAL function

*Table 22* describes the TS_ResetI2C_Peripheral function of the hardware abstraction layer.

**Table 22. TS_ResetI2C_Peripheral**

| Name | Description |
|---|---|
| Function name | TS_ResetI2C_Peripheral |
| Function prototype | void TS_ResetI2C_Peripheral(TS_I2C_SettingsType* pI2C_Settings) |
| Behavior description | Apply a software Reset on the I2C peripheral, e.g. in order to exit from an I2C blocking error |
| Input parameter {x} | pI2C_Settings - I2C peripheral settings structure pointer |
| Output parameter {x} | None |
| Return value | None |
| Required preconditions | None |
| Called functions | None |

### TS_CheckEventWithTimeout HAL function

*Table 23* describes the TS_CheckEventWithTimeout function of the hardware abstraction layer.

**Table 23. TS_CheckEventWithTimeout**

| Name | Description |
|---|---|
| Function name | TS_CheckEventWithTimeout |
| Function prototype | TS_ErrStatus TS_CheckEventWithTimeout(TS_I2C_SettingsType* pI2C_Settings, u32 I2C_EVENT, u32 uTimeoutTicks) |
| Behavior description | Check the I2C event inside a max timeout time. This function requires that the g_TempSensorTick variable is defined as extern in the main application, and it is incremented in SysTickHandler Interrupt or another scheduler time. |
| Input parameter {x} | pI2C_Settings - I2C peripheral settings structure pointer;<br>I2C_EVENT - The I2C event to check;<br>uTimeoutTicks - The g_TempSensorTick to wait before return an error;<br>uTimeoutTicks must be >=1. |
| Output parameter {x} | None |
| Return value | TS_OK if event occurs inside the timeout time, TS_ERROR otherwise |
| Required preconditions | g_TempSensorTick variable is defined as extern in the main application, and it is incremented in a scheduler time. |
| Called functions | None |

### TS_FillDataFromRegister HAL function

*Table 24* describes the TS_FillDataFromRegister function of the hardware abstraction layer.

**Table 24. TS_FillDataFromRegister**

| Name | Description |
|---|---|
| Function name | TS_FillDataFromRegister |
| Function prototype | TS_ErrStatus TS_FillDataFromRegister(TS_I2C_SettingsType* pI2C_Settings,TS_u16* pReadRegValue, TS_RegLengthType eRegLenType, u32 uTimeoutTicks) |
| Behavior description | Read from the specified STLM75 a register value (8 or 16 bits) and return the value in output parameter.<br>**Warning: SetPointerRegister function MUST be called before calling this one.** |
| Input parameter {x} | pI2C_Settings - I2C peripheral settings structure pointer;<br>eRegLenType - Register byte-length type (Can be 1-Byt or 2-Byte);<br>uTimeoutTicks - The g_TempSensorTick to wait before returning an error;<br>uTimeoutTicks must be >=1. |

**Table 24. TS_FillDataFromRegister (continued)**

| Name | Description |
|---|---|
| Output parameter {x} | pReadRegValue - Read register value pointer; |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | SetPointerRegister function MUST be called before calling this one. |
| Called functions | TS_ErrStatus TS_CheckEventWithTimeout(…) |

### TS_FillRegisterFromData HAL function

*Table 25* describes the TS_FillRegisterFromData function of the hardware abstraction layer.

**Table 25. TS_FillRegisterFromData**

| Name | Description |
|---|---|
| Function name | TS_FillRegisterFromData |
| Function prototype | TS_ErrStatus TS_FillRegisterFromData(TS_I2C_SettingsType* pI2C_Settings,TS_u16 uWriteRegValue, TS_RegLengthType eRegLenType, u32 uTimeoutTicks) |
| Behavior description | Write the data passed in input parameter on the specified STLM75 register value (8 or 16 bits).<br>**Warning: WARNING: SetPointerRegister function MUST be called before calling this one.** |
| Input parameter {x} | pI2C_Settings - I2C peripheral settings structure pointer;<br>eRegLenType - Register byte-length type (Can be 1-Byt or 2-Byte);<br>uWriteRegValue - Data value to write on the right register;<br>uTimeoutTicks - The g_TempSensorTick to wait before returning an error; uTimeoutTicks must be >=1. |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | SetPointerRegister function MUST be called before calling this one. |
| Called functions | TS_ErrStatus TS_CheckEventWithTimeout(…) |

### TS_SetPointerRegister HAL function

*Table 26* describes the TS_SetPointerRegister function of the hardware abstraction layer.

**Table 26. TS_SetPointerRegister**

| Name | Description |
|---|---|
| Function name | TS_SetPointerRegister |
| Function prototype | TS_ErrStatus TS_SetPointerRegister(TS_I2C_SettingsType* pI2C_Settings, TS_RegPointerType eRegPointer, u32 uTimeoutTicks) |
| Behavior description | Set the STLM75 Command/Pointer register, before executing a Read/Write operation on STLM75 register |
| Input parameter {x} | pI2C_Settings - I2C peripheral settings structure pointer; eRegLenType - Register byte-length type (Can be 1-Byt or 2-Byte); uTimeoutTicks - The g_TempSensorTick to wait before returning an error; uTimeoutTicks must be >=1. |
| Output parameter {x} | None |
| Return value | TS_OK if successful, TS_ERROR otherwise |
| Required preconditions | None |
| Called functions | TS_ErrStatus TS_CheckEventWithTimeout(…) |

# 5 Example application

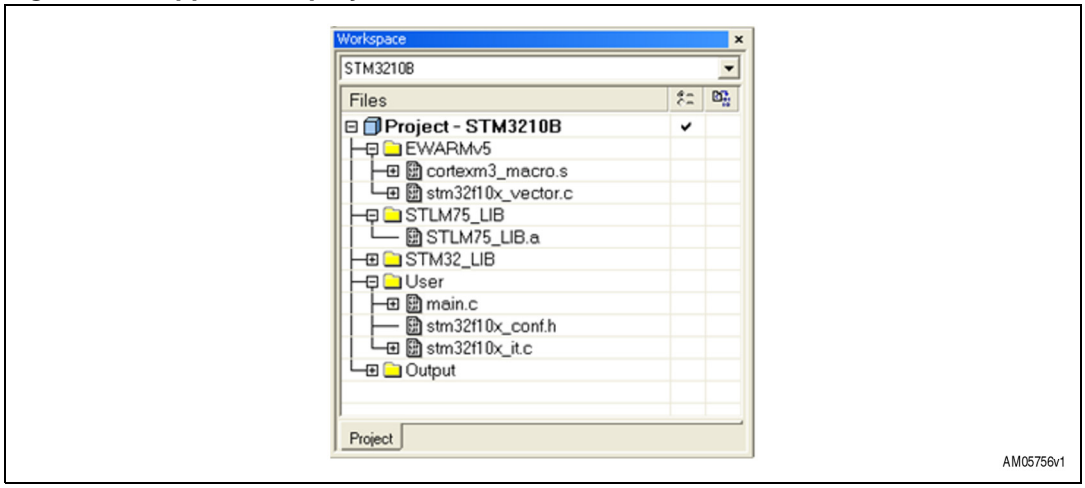
Together with the STLM75 firmware library package, an example application is delivered in order to provide the final user with a real example of STLM75 library use. See *Figure 6*.

The delivered example application has been developed using IAR EWARM 5.20 IDE and can be built for both STM32F10xxx Medium-density and High-density microcontroller families.

An application project which wants to use the STLM75 library must:

- Include the STLM75.a file in the project generated compiling the STLM75_LIB library firmware delivered in the final library package. See *Section 3.2*. The 2STLM75 library source code can be included instead of the STLM75.a object file
- Implement a main function as described in the following section.

**Figure 6. Application project files**




## 5.1 main.c

An example of a main application is reported below. The main function contains an example of the STLM75 library initialization/configuration and implements temperature sensor registers read/write operations:

```
/* Includes ------------------------------------------------------*/
#include "stm32f10x_lib.h"
#include "TempSensorObj.h"
#include <stdio.h>

/* Private typedef -----------------------------------------------*/
/* Private define ------------------------------------------------*/
```

```
// STLM75 TEMPERATURE SENSOR DEFINES
// Temperature sensor controller signal
#define TMP_I2C_PERIPHERAL   I2C2
#define TMP_I2C_SLAVE_ADDR7  0x90   // 1001 0000
#define TMP_I2C_CLOCK_SPEED  400000 //400Khz
#define TMP_SCL_PORT         GPIOB
#define TMP_SCL_PIN          GPIO_Pin_10
#define TMP_SCL_MODE         GPIO_Mode_AF_OD
#define TMP_SDA_PORT         GPIOB
#define TMP_SDA_PIN          GPIO_Pin_11
#define TMP_SDA_MODE         GPIO_Mode_AF_OD
#define TMP_OS_INT_PORT      GPIOB
#define TMP_OS_INT_PIN       GPIO_Pin_12
#define TMP_OS_INT_MODE      GPIO_Mode_AF_PP

/* Private macro -------------------------------------------------*/
/* Private variables ---------------------------------------------*/
ErrorStatus HSEStartUpStatus;

/* Private function prototypes -----------------------------------*/
void RCC_Configuration    (void);
void NVIC_Configuration   (void);
void SysTick_Configuration(void);
void STLM75_Configuration (TempSensorType* pObjTempSensor);

/* Private functions ---------------------------------------------*/

/*******************************************************************
* Function Name  : Main
* Description    : Main program.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************/
int main(void)
```

```
{
    TempSensorType*     pObjTempSensor;
    TS_TemperatureType* pStrTemperature;
    TS_TemperatureType  strTestTemperature;
    u8                  uTestConfigurationValue;
    char                cTemperatureSign;
    u16                 uReadingNumber = 0;

    #ifdef DEBUG
      debug();
    #endif

    /* System clocks configuration ------------------------------*/
    RCC_Configuration();

    /* NVIC configuration ---------------------------------------*/
    NVIC_Configuration();

    /* STLM75 configuration -------------------------------------*/
    pObjTempSensor = NewTempSensorObj ();
    STLM75_Configuration(pObjTempSensor);


/* You can configure timeout to avoid application block in case  of
I2C communication problems. TimeoutTick is the number of
g_TempSensorTick to wait during an I2C communication between Micro
and STLM75 before considering an error has occurred and resetting
the I2C peripheral.
WARNING: TimeoutTicks must be set >0 as following ONLY if
g_TempSensorTick is defined as "extern volatile u32
g_TempSensorTick" in this Application,and is incremented in
SysTickHandler or another timer irq handler. */
    pObjTempSensor->TimeoutTicks = 2; /* 2ms:we have a
                                         g_TempSensorTick each
                                         millisecond */
    /* STLM75 Library use example -------------------------------*/
```

```
    /* TEST: Write the following value for the Configuration
register:
             0x00 */
    uTestConfigurationValue = 0x00;
    pObjTempSensor->SetConfiguration  (pObjTempSensor,
                                       uTestConfigurationValue);


    /* TEST: Write the following value for the Hysteresis register:
             75.5° C  */
    strTestTemperature.TS_Sign         = TS_PLUS_SIGN;
    strTestTemperature.TS_IntegerValue = 75;
    strTestTemperature.TS_DecimalValue = 5;     /* 0 or 5 only */
    pObjTempSensor->SetTempHysteresis (pObjTempSensor,
                                       &strTestTemperature);


    /* TEST: Write the following value for the Over-Limit register:
             50.5° C  */
    strTestTemperature.TS_Sign         = TS_PLUS_SIGN;
    strTestTemperature.TS_IntegerValue = 50;
    strTestTemperature.TS_DecimalValue = 5;    /* 0 or 5 only */
    pObjTempSensor->SetTempOverLimit  (pObjTempSensor,
                                       &strTestTemperature);


  /* TEST: Read STLM75 values for the Configuration, Hysteresis
        and Over-Limit temp. registers */
    uTestConfigurationValue = pObjTempSensor->GetConfiguration
                              (pObjTempSensor, TS_TRUE);
    pStrTemperature         = pObjTempSensor->GetTempHysteresis
                              (pObjTempSensor, TS_TRUE);
    pStrTemperature         = pObjTempSensor->GetTempOverLimit
                              (pObjTempSensor, TS_TRUE);


    /* Infinite main loop ---------------------------------------*/
    while(1)
    {
        /* TEST: Read observed STLM75 temperature and show the read
                 value */
        pStrTemperature = pObjTempSensor->GetTemperature
                          (pObjTempSensor, TS_TRUE);
```

```
            cTemperatureSign = (pStrTemperature->TS_Sign ==
                                TS_PLUS_SIGN) ? '+' : '-';


            /* Send Temperature read data to the debugger Terminal Output
                Window */
            printf ("Read Temperature (%d): %c%d.%d\r\n",              \
                                        uReadingNumber++,              \
                                        cTemperatureSign,              \
                                       pStrTemperature->TS_IntegerValue, \
                                        pStrTemperature->TS_DecimalValue);
        }
    }



    /*******************************************************************
    * Function Name  : STLM75_Configuration
    * Description    : Configure the Micro I2C peripheral and GPIO pins
                       in order to use
    *                : the TempSensor hardware components present on the
                       application board
    * Input          : pObjTempSensor - The STLM75 temperature sensor
                       object pointer
    * Output         : None
    * Return         : None
    *******************************************************************/
    void STLM75_Configuration (TempSensorType* pObjTempSensor)
    {
        TS_SignalsType      TempSensorSignals;
        TS_I2C_SettingsType TempSensorI2C_Settings;

        /* Enable I2C2 clock */
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C2, ENABLE);
        /* Enable GPIOB clock */
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);


        /* GPIO signals configuration */
```

```
    /* Configure PB.10 as alternate function Open-Drain */
    TempSensorSignals.TS_SCL.Port    = TMP_SCL_PORT;
    TempSensorSignals.TS_SCL.Pin     = TMP_SCL_PIN;
    TempSensorSignals.TS_SCL.Mode    = TMP_SCL_MODE;

    /* Configure PB.11 as alternate function Open-Drain */
    TempSensorSignals.TS_SDA.Port    = TMP_SDA_PORT;
    TempSensorSignals.TS_SDA.Pin     = TMP_SDA_PIN;
    TempSensorSignals.TS_SDA.Mode    = TMP_SDA_MODE;

    /* Configure PB.12 as alternate function Open-Drain */
    TempSensorSignals.TS_OS_INT.Port = TMP_OS_INT_PORT;
    TempSensorSignals.TS_OS_INT.Pin  = TMP_OS_INT_PIN;
    TempSensorSignals.TS_OS_INT.Mode = TMP_OS_INT_MODE;

    //OverLimitConfigIrq();

    // I2C Peripheral settings configuration
    TempSensorI2C_Settings.TS_I2C_Ptr     = TMP_I2C_PERIPHERAL;
    TempSensorI2C_Settings.TS_I2C_Clock   = TMP_I2C_CLOCK_SPEED;
    TempSensorI2C_Settings.TS_I2C_Address = TMP_I2C_SLAVE_ADDR7;


    /* Initialize the Temperature Sensor Object */
    pObjTempSensor->Init(pObjTempSensor, &TempSensorSignals,
                         &TempSensorI2C_Settings);
}
```

# 6 References


1. STLM75; *Digital temperature sensor and thermal watchdog*, datasheet
2. RM0008; *STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs,* reference manual
3. STM32F10xFWLib 2.0.3, AN2953; *How to migrate from the STM32F10xxx firmware library V2.0.3 to the STM32F10xxx standard peripheral library V3.0.0*, application note

# 7 Revision history

Table 27. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 12-Oct-2010 | 1 | Initial release. |